1. Welcome to Inform

§1.1 Preface
§1.2 Acknowledgements
§1.3 The facing pages
§1.4 The Go! button
§1.5 The Replay button
§1.6 The Index and Results panels
§1.7 The Skein

§1.1 Preface

Welcome to Inform, a design system for interactive fiction based on natural language.

Interactive fiction is a literary form which involves programming a computer so that it presents a reader with a text which can be explored. Inform aims to make the burden of learning to program such texts as light as possible. It is a tool for writers intrigued by computing, and computer programmers intrigued by writing. Perhaps these are not so very different pursuits, in their rewards and pleasures.

> The sheer joy of making things... the fascination of fashioning complex puzzle-like objects of interlocking moving parts and watching them work in subtle cycles... the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. (Frederick P. Brooks, "The Mythical Man-Month", 1972)

Writing with Inform is one of two interlinked books included with Inform: a concise but complete guide to the system. The other book is **The Inform Recipe Book**, a comprehensive collection of examples, showing its practical use.

These notes are arranged so that the reader can, in principle, write whole works of fiction as early as the end of Chapter 3. Each subsequent chapter then extends the range of techniques available to make livelier and more intriguing situations.

Today's Inform language (sometimes called "Inform 7") is very different from its 20th-century predecessor, which was called Inform 6. A few advanced sections of this book show how unusual effects can be achieved by mixing low-level coding in Inform 6 notation with more usual Inform text. However, most users will never need this. For information about Inform 6, see **inform-fiction.org**.

This book is also a guide to the Inform language, rather than a manual on how to use its supporting tools. Those tools, when used at the command line rather than inside the

Inform app, have numerous features not covered here. Manuals for them are all available online: see **github.com/ganelson/inform**.

> Programming is best regarded as the process of creating works of literature, which are meant to be read... so we ought to address them to people, not to machines. (Donald Knuth, "Literate Programming", 1981)

Example

1. **About the examples**
   An explanation of the examples in this documentation, and the asterisks attached to them.

RB 1.1 Preface

§1.2 Acknowledgements

Inform 7 is dedicated to Emily Short and Andrew Plotkin, whose shrewd and sceptical suggestions made a contribution which can hardly be overstated. A long email correspondence with Andrew entirely subverted my original thoughts about natural-language IF, as he convinced me that the "new model" of rule-based IF was a truer foundation; while Emily's wry, witty analysis and how-about-this? cheered me at low moments, besides providing the impetus and often the specifics for a lot of the best ideas.

From the outset, I have thought of Inform 7 as no longer being a command-line compiler, but a compiler in combination with a humanising user interface. All credit for the reference implementation under Mac OS X belongs to Andrew Hunter. How simple the metaphor of an interactive book with facing pages may seem, but the coding was an enormous challenge. In 2014 Toby Nelson, my brother, put months of time into the project by rewriting and modernising the Mac OS X application: sandboxing it for the Mac App Store, giving it a more contemporary design, and much more. He continues to maintain it today.

Though David Kinder's Windows application does indeed visually follow the OS X original, the two programs were coded independently, and the programming task taken up by David was formidable indeed. Philip Chimento's Gnome-based user interface for Linux became officially part of the project in November 2007, when the first easy-to-install packages for Ubuntu and Fedora were offered. Adam Thornton gave invaluable assistance in the closed-source age of Inform to make generic Unix binaries available, too.

While Inform is not strictly speaking a project of the Interactive Fiction Technology

Foundation (IFTF), it benefits enormously from the Foundation's good work. In particular, the Narrascope conferences were invaluable in the period 2017-2022, and I thank Judith Pintar and Andrew Plotkin for arranging speaking slots at them.

Inform in its widest sense incorporates work by so many people that it's simply impossible to thank all of them, but Zed Lopez, Dannii Willis, Mark Musante, Brian Rushton, Dan Fabulich, Hugo Labrande, Erik Temple, Ron Newcomb, Eric Eve, Justin de Vesine and Juhana Leinonen all deserve special mention. Many hundreds of users have filed patient and careful bug reports, keeping us on the straight and narrow. They're contributors, too.

The original development of Inform 7 was a long haul, and I would particularly like to thank Sonja Kesserich, David Cornelson and other volunteers for their early testing of a then-fragile system. The final months before the Public Beta release of Inform 7 were made more enjoyable, as well as more productive, by fruitful discussions leading to a cross-platform standard for bibliographic data and cover art. Special mentions to L. Ross Raszewski, who wrote frighteningly efficient reference software in frighteningly little time; the librarians of the IF-Archive, Andrew Plotkin, David Kinder and Paul Mazaitis; and my fellow authors of IF design systems - Mike Roberts (of the Text Adventure Development System); Kent Tessman (of Hugo); and Campbell Wild (of ADRIFT).

§1.3 The facing pages

At the start the only panels available are a blank space in which to write the first lines of a new interactive fiction - the Source panel - and this one, the Documentation. Clicking on the other choices will do nothing.

The exception is the Settings panel, which contains some preference settings for the individual project - not the whole application. This is always available, but it controls settings which can be left alone almost all of the time.

§1.4 The Go! button

Clicking the Go button translates the text in the Source panel into a computer program which enacts the interactive fiction, and automatically sets it going (in the Story panel, which opens as needed).

If the Source is empty of text, Inform will be unable to create anything: it needs at least one name of a location where the drama can unfold. For reasons of tradition, such locations are normally called "rooms", though people have used them to represent anything from grassy fields to states of mind and other metaphorical places.

```
"Midsummer Day"

The Gazebo is a room.
```

Clicking Go with this text in the Source panel will result in a short delay, after which the Story panel will appear, from which we can explore this newly created world: an interactive fiction called "Midsummer Day". It will not be very exciting, since Inform has only five words to go on, but we can add more detail to the source at any point and then click Go again to try out the changes. (Note that there is no need to "quit" these explorations in the Story panel. When Go is clicked, any story already in progress is discarded in favour of the new version.)

§1.5 The Replay button

Replay works identically to Go, except that it does something further: once the story is created, it automatically plays through the same commands as were typed into the previous version. For instance: suppose we click Go to bring Midsummer Day into being, and find ourselves playing the story. We type "look" and find that there is not much to see. Going back to the source, we add

```
"A white canvas parasol raised up on stakes driven into the grass."
```

so that the source now reads

```
"Midsummer Day"

The Gazebo is a room. "A white canvas parasol raised up on stakes driven into the grass."
```

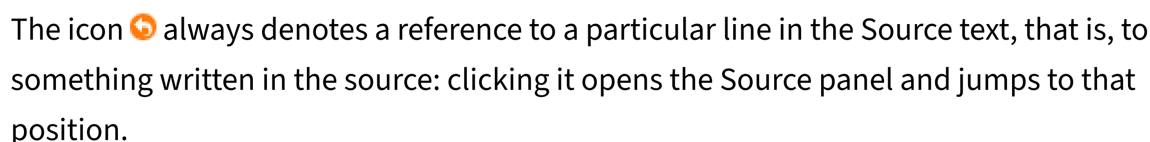
Instead of clicking Go, we click Replay, and can sit back and watch what has changed. In this example, it only saves us the trouble of typing "look", but once stories become long and elaborate, Replay is invaluable: and especially when we notice in play that something very minor is wrong - a spelling error, say - and want to fix it immediately, without fuss.

§1.6 The Index and Results panels

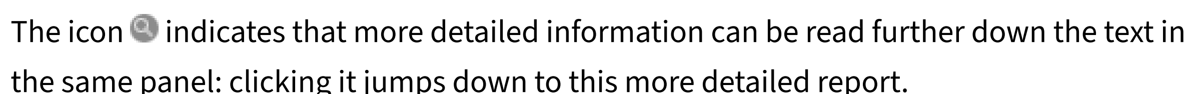
If, when Go! is clicked, the text in the Source panel is not fully understood, then Inform will generate a report of the problems it found, which will open in the "Errors" panel. (Other information is also available in "Errors", but most of it is used for debugging Inform, and can be ignored.)

On the other hand, if the text was fully understood then another new panel will become available: the "Index". This is a cross-referenced index of the source, or rather, of the

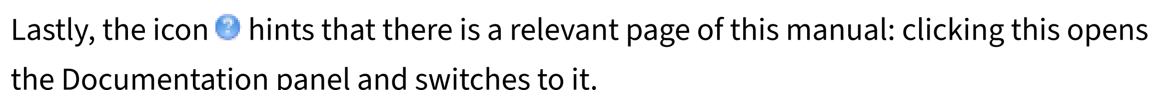
interactive fiction which has been generated. The Index is only an optional convenience, but becomes more and more helpful as the fiction grows larger. Its exact format does not matter for now.



The icon always denotes a reference to a particular line in the Source text, that is, to something written in the source: clicking it opens the Source panel and jumps to that position.



The icon indicates that more detailed information can be read further down the text in the same panel: clicking it jumps down to this more detailed report.



Lastly, the icon hints that there is a relevant page of this manual: clicking this opens the Documentation panel and switches to it.

§1.7 The Skein

The Replay button demonstrates that Inform must be quietly remembering the commands typed into the last run through the story. In fact it remembers, and automatically organises, *every* previous run.

Inform's approach to testing interactive fiction is to treat it as being like the analysis of other turn-based games, such as chess. It would be prohibitively difficult to work out every possible combination of moves: instead, we analyse those which go somewhere, and look for significant choices. Every Queen's Gambit begins with the same first three moves (1. d4, d5; 2. c4), but then there is a choice, as the next move decides whether we have a Queen's Gambit Accepted (dxc4) or Declined (e6). Books about chess often contain great tables of such openings, which run together for a while but eventually diverge. To learn chess, one must explore all of these variations.

Inform's Skein panel is just such a table, built automatically. If we think of the list of typed commands as a thread, then the skein is (as the name suggests) braided together from all these threads. In the display, time begins at the top, with the **start** knot, and the threads of different play-throughs hang downwards from it.

Double-clicking on a command translates the source afresh and replays the story from **start** down to that command, and then stops. We are then free to continue play by typing commands into the Story panel, of course, and these commands will automatically be recorded in the Skein as a new variation of play, diverging from the previous threads.

The user interface for the Skein looks slightly different on different versions of the Inform apps (that is, the MacOS version is not quite the same as the Windows version, and so on), so this manual is not the best place to describe it. In any case, the best way to find out about it is probably to experiment.

2. The Source Text

§2.1 Creating the world
§2.2 Making rules
§2.3 Punctuation
§2.4 Problems
§2.5 Headings
§2.6 Why using headings is a good idea
§2.7 The SHOWME command
§2.8 The TEST command
§2.9 Material not for release
§2.10 Installing extensions
§2.11 Including extensions
§2.12 Use options
§2.13 Administering classroom use
§2.14 Limits and the Settings panel
§2.15 What to do about a bug
§2.16 Does Inform really understand English?

§2.1 Creating the world

Designing an interactive fiction can be divided into two related activities. One is the creation of the world as it appears at the start of play: where and what everything is. The other is to specify the rules of play, which shape how the player interacts with that initially created world. A new Inform project is void and without form, so to speak, with nothing created: but it starts with hundreds of standard rules already in place.

The same division between creating things, and laying down rules, is visible in Inform source text. The creation of the world is done by making unconditional factual statements about it. For example,

```
The wood-slatted crate is in the Gazebo. The crate is a container.
```

Inform calls sentences like these "assertions". The verb is always written in the present tense (thus the crate "is", not "will be"). Further examples are:

```
Mr Jones wears a top hat. The crate contains a croquet mallet.
```

The words "is", "wears" and "contains" are forms of three of the basic verbs built in to Inform. There are only a few built-in assertion verbs, of which the most important are *to be*, *to have*, *to carry*, *to wear*, *to contain* and *to support*. (As we shall see, further assertion

verbs can be created if needed.)

The world described by these assertions is the starting condition of the story: what happens when play begins is another matter. If somebody picks up the crate and walks off with it, then it will no longer be in the Gazebo. Mr Jones may remove his hat.

§2.2 Making rules

The other kind of sentence tells Inform what should happen in certain circumstances, and reads like an instruction issued to someone:

```
Instead of taking the crate, say "It's far too heavy to lift."
```

This is a "rule", and it changes the crate's behaviour. The player who tries typing "take crate", "pick up the crate" or similar will be met only with the unhelpful reply "It's far too heavy to lift." The many different kinds of thing which the player can do are called "actions", and are always written as participles: "taking …", for instance, or "putting … on …".

Inform is built on a mass of several hundred rules, some quite complex, and it could even be said that Inform *is* that mass of rules. We never see the complexity behind the scenes because the whole aim is to provide a basic, penny-plain, vanilla flavoured sort of realism. It would be surprising if one could put the crate inside itself, so a rule exists to forbid this. It would be surprising if one could drop something which was already on the ground, and so on. These basic rules of realism are the ones which every new Inform project starts with.

A rule always starts with a situation which it applies to, and then follows with one or more things to do. Here's an example where the situation is "Before taking the crate" - the player is just starting to try to pick the box up - and there's a three-step process to follow, but steps 2 and 3 happen only if step 1 comes out in a particular way:

```
Before taking the crate:
    if the player is wearing the hat:
        now the hat is in the crate;
        say "As you stoop down, your hat falls into the crate."
```

The steps to follow here are called "phrases". Inform knows about 400 built-in phrases, but most of them are needed only occasionally. These three are used over and over again:

if tells Inform to do something only if some "condition" holds, here "the player is wearing the hat";
now tells Inform to change the situation, here so that the hat moves to the crate; and
say tells Inform to say something, that is, to write some text for the player to read.

Every one of the built-in phrases has a definition somewhere in this book. The full definition of "say" will come later, but in the simple form above it writes out the given text for the player to read. (Normally this text is simply shown on screen, not spoken aloud, unless software adapted for partially sighted people is being used.) Phrase definitions are all linked to in the Phrases page of a project's Index.

§2.3 Punctuation

An example rule from the previous section demonstrates one of Inform's conventions about punctuation, and is worth pausing to look at again.

```
Instead of taking the crate, say "It's far too heavy to lift."
```

In English grammar, it's usual to regard a full stop as closing its sentence even when it occurs inside quotation marks, provided there is no indication to the contrary, and this is also the rule used by Inform. Thus:

```
The description is "Shiny." It is valuable.
```

is read as equivalent to

```
The description is "Shiny.". It is valuable.
```

Sentence breaks like this occur only when the final character of the quoted text is a full stop, question mark or exclamation mark (or one of these three followed by a close bracket) and the next word begins, in the source code, with a capital letter. A paragraph break also divides sentences, behaving as if it were a full stop.

Material in square brackets [like so] is "comment", in computing jargon: it is considered as being an aside, a private note by the author, and not read in by Inform. This allows us to make notes to ourselves like so:

```
The China Shop is a room. [Remember to work out what happens if the bull gets in here!]
```

Inform is all about text, so pieces of text are often quoted in Inform source. This example is typical:

```
The description is "Shiny." It is valuable.
```

Quotations always use double-quotation marks, which aren't part of the text. So the description here is just the five letters and full stop in between the marks:

```
Shiny.
```

That seems straightforward, but there are three conventions to watch out for.

1. Square brackets [and] inside quoted text don't literally mean [and]. They're used to describe what Inform should say, but in a non-literal way. For example,

```
"Your watch reads [time of day]."
```

might produce

```
Your watch reads 9:02 AM.
```

These are called "text substitutions". They're highly flexible, and they can take many different forms.

2. Single quotation marks at the edges of words are printed as double. So:

```
"Simon says, 'It's far too heavy to lift.'"
```

produces

```
Simon says, "It's far too heavy to lift."
```

3. Texts which end with sentence-ending punctuation - full stop, question mark, exclamation mark - are printed with a line break after them. So:

```
say "i don't know how this ends";
say "I know just how this ends!";
```

would come out quite differently - this doesn't affect the appearance of the text, but only the position where the next text will appear. Something to be careful about is that this only applies when the punctuation occurs at the end of a "say", as in these examples. (It doesn't apply when a varying textual value is printed, using some text substitution, because then the pattern of where line breaks occur would be unpredictable - sometimes the value might end in a punctuation mark, sometimes not.)

These three punctuation rules for texts feel very natural with practice, and Inform users sometimes don't realise the third rule is even there, because it just seems the right thing to happen. But occasionally the rules get in the way of what we want to do. (For instance, how do we get a literal [or]? What if we want a single quote mark where Inform thinks we want a double, or vice versa?) So we'll come back to these rules in more detail in the chapter on Text.

Inform also reads other punctuation marks. Colon ":" and semicolon ";" turned up in the previous section, in the writing of rules.

As these examples begin to show, Inform source imitates the conventions of printed books and newspapers whenever there is a question of how to write something not easily fitting into words. The first example of this is how Inform handles headings, but to see why these are so useful we first look at Problems.

See Also

How Inform reads quoted text for a fuller exploration of the punctuation rules for text.

§2.4 Problems

The language used in the source reads as if it were English aimed at a human reader (and this is intentional: the designer, after all, is a human reader and needs to be able to understand his or her own source), but in reality Inform can only understand a very modest range of sentences and will complain if its limits are passed. Subtler problems arise if the source contains contradictions. For instance, the following "Problem" might be produced:

> **Problem.** You wrote 'A starting pistol is in the cup', but in another sentence 'A Panama hat is on the cup': the trophy cup cannot both contain things and support things, which is what you're implying here. If you need both, the easiest way is to make it either a supporter with a container attached or vice versa. For instance: 'A desk is here. On the desk is a newspaper. An openable container called the drawer is part of the desk. In the drawer is a stapler.'

This is a rather discursive error message, and if a similar problem were to occur in the same run through, it would be curtailed to:

> **Problem.** You wrote 'A firing pistol is in the box', but in another sentence 'A fedora hat is on the box': again, the croquet box cannot both contain things and support things.

§2.5 Headings

Inform provides for us to organise the source code in just the way that a printed book would be organised: with headings and subheadings. Firstly, we can put the title at the top. If the first paragraph consists only of a single quoted piece of text, then that's the title; and an author can also be given, as follows:

```
"Spellbreaker" by Dave Lebling
```

We will later see that more bibliographic information can also be placed here, in the same way that the imprint page of a novel comes before the text gets going. The author's name can normally be given without quotation marks, so long as it contains no punctuation. For instance:

```
"Three Men in a Boat" by "Jerome K. Jerome"
```

needs quotes as otherwise the full stop after the K will be mistaken for the end of a sentence.

A sentence which is the only one in its paragraph and which begins with any of the words "volume", "book", "part", "chapter" or "section" is considered to be a heading or a sub-heading. It must not contain a typed line break, and in order to stand alone in its paragraph there should be a skipped line both before and after it. For instance:

```
Section 2 - Flamsteed's Balloon
```

Headings can be written in any format, provided they start with one of the five indicator words, and they are hierarchical: a "Part …" heading is considered more significant than a "Chapter …" heading but not so significant as a "Book …" heading, and so on. (We do not need to use all five kinds of heading.)

§2.6 Why using headings is a good idea

Reports of problems, as we have seen, often quote back the source to justify themselves. Rather than quoting line numbers ("Midsummer Day, line 2017" or something similar) Inform uses the icon. The down side of this is that a glance at the list of problems might give little hint of whereabouts in the source the difficulties lie. Inform therefore makes use of headings to give a general indication:

```
In Part the First, Chapter 1 - Attic Area:

Problem. You wrote 'South of the Attic is the Winery', but in another sentence 'South of the Attic is the Old Furniture': this looks like a contradiction, which might be because I have misunderstood what was meant to be the subject of one or both of those sentences.

In Chapter 2 - Deeper In:

Problem. You wrote 'The Disused Observatory is south of the Dark Room', but in another sentence 'South of the Dark Room is the Cupboard': again, this looks like a contradiction.
```

Secondly, headings are used in the Contents page of the Index, and they allow rapid navigation through the source, by jumping to any heading or subheading with a single click.

Finally, headings are used when working out what a name refers to. Suppose the source contains both a "four-poster bed" and also a "camp bed", and we write something like "The pillow is on the bed." Inform decides which bed is meant by giving priority to whichever is defined in the current section (so far), or failing that the current chapter, or current part, or current book, or finally the current volume. This allows us to write, for instance,

```
The four-poster bed is in the Boudoir. The pillow is on the bed.
```

and not have the pillow mysteriously turn up on the camp bed, which hasn't been mentioned since way back in Chapter 2.

§2.7 The SHOWME command

Problem messages are generated when the source text does not make sense to Inform. Even if it does make sense, though, there is no guarantee that it does what the author intends, and the only way to find out is to test the result by playing through it (or asking others to). For the most part one plays as if one were the eventual reader of the work, but sometimes it is highly convenient to have the god-like powers which are an author's prerogative. These are provided by the testing commands, which are present at every stage until the final release version (generated by the Release button). They will be introduced in this manual as they become relevant: here is the first.

The testing command SHOWME prints out a brief summary about a room or thing, and any contents or parts it may have. Typing SHOWME on its own shows the current room, but any item or room in the story, however distant, can be named instead. For instance:

```
>showme
Boudoir - room
    four-poster bed - supporter
    yourself - person
    pillow

>showme diamonds
diamonds - thing
location: in the strongbox on the dresser in the Drawing Room
unlit; inedible; opaque; portable; singular-named; improper-named
description: The diamonds glitter dangerously.
printed name: diamonds
```

Much of this can be seen, and seen more easily, in the World tab of the Index panel: but that only shows the initial state of play, whereas the SHOWME command reveals the situation in mid-story. ("Room", "supporter" and so on are kinds, of which more in Chapter 3.)

See Also

High-level debugging commands for more convenient testing commands like this one.

§2.8 The TEST command

The only way to thoroughly test a work of IF is to run a complete solution through it, and carefully check the resulting transcript of dialogue. The Skein and Transcript tools of the Inform application are provided for exactly this purpose.

All the same, most works of interactive fiction contain occasional vignettes, either in terms of short scenes of narrative, or in the behaviour of particular things or rooms, which we would like to test without the fuss of using the full story-level Skein tool. The examples in the documentation are like this: in almost every example, typing TEST ME puts the story through its paces.

Solutions or sequences for testing ("scripts") can be defined with sentences like so:

```
Test balloon with "get balloon / blow balloon / drop balloon".
```

This has no effect on the design itself, but ensures that when the story is played, typing "test balloon" will run through the given three commands in sequence, as if we had typed "get balloon" and then "blow balloon" and then "drop balloon".

The name for the test (balloon in this example) has to be a single word. Typing just "test" at the story prompt gives a list of all the test scripts known to the story. Test scripts can

make use of each other, for instance:

```
Test all with "test balloon / test door".
```

One convenient way to keep track of the solution for a work being written is to include a test script at the end of each section, and to place a master test script (like "test all") at the top of the source. But different designers will prefer different approaches, and this testing system is no more than an optional convenience.

Many tests will only be sensible in given places, which may be hard to reach from the initial position; or with the aid of given things, which may be difficult to obtain. We are therefore allowed to add stipulations to test scripts:

```
Test balloon with "get balloon / blow balloon / drop balloon" holding the balloon.
```

The "… holding the balloon" means that the balloon will be transferred to the player's ownership immediately before the test script is run, unless it is already held. Similarly:

```
Test jam with "get jam / taste jam / eat jam" in the Kitchen.
```

Or we might want to say both:

```
Test jam with "get jam / taste jam / eat jam" in the Kitchen holding the jam.
```

(Single quotation marks in test scripts are interpreted the same way in test scripts as they are in other text: that is, they are sometimes read as double-quotes unless they appear to be present as apostrophes. The notation ['] forces a single quotation mark if necessary. Similarly, [/] forces a literal forward slash, and prevents the / from being read as dividing up two commands.)

Sometimes when testing it's convenient to get hold of something not easily available at the moment. The testing command "PURLOIN" does this:

```
The jewelled Turkish clockwork hat is in the sealed glass box.

> PURLOIN HAT
[Purloined.]
```

This can also make test scripts shorter, but of course it's important to make sure that people without PURLOIN powers can still play through.

§2.9 Material not for release

Special testing commands, like "TEST" and "SHOWME", are automatically excluded from the story if it is exported from the Inform application using the Release button. We sometimes want to write our own for-testing-purposes-only code, though, and for this purpose we are allowed to designate whole headings as being "not for release":

```
Section 10 - Open sesame - Not for release

Universal opening is an action applying to nothing.
Understand "open sesame" as universal opening.
Carry out universal opening: now all doors are open.
Report universal opening: say "Open Sesame!"
```

Clearly we do not wish the final reader to be able to type "OPEN SESAME", so this whole heading will be disregarded in the Release version, as will any heading whose name includes "not for release".

Note that if a chapter, say, is marked as "not for release", then its subheadings (mere sections) will also not be for release. If in doubt, check the "Contents" index: if any section is "not for release" then so are all of its subheadings.

The reverse effect is produced by:

```
Section 10 - Open sesame - For release only
```

That is, it marks material included only in a Release version.

§2.10 Installing extensions

The original Inform of 1993 provided no special facilities for "extensions" - in effect, additional packets of rules providing extra features - but the creation and circulation of these extensions soon became a flourishing part of Inform culture. Today's Inform actively promotes sharing of such extensions, both to bring writers together and to support good practice. For the user of an extension, the advantage is clear: why go to great trouble to (say) work out how to make doors open automatically as needed, when somebody else has already perfected this? For the writer of an extension, there is the satisfaction of producing a good solution to a ticklish problem, and contributing to the public good.

Newcomers will probably not need extensions for quite some while, but there is nothing difficult about using them, so a few brief notes are worth giving here. (The final chapter of the documentation covers the writing of new extensions.)

Extensions are identified by name (say "Following People") and also by author (say "Mary Brown"). They need to be installed before they can be used, which means downloading them from the Internet. By far the easiest way to do this is to use the Public Library feature of Inform: then the application can do everything, letting us either choose individual extensions or download them en masse. But it's also possible to install extensions by hand.

In fact, though, Inform can automatically install extensions for us: we need only select the "Install Extension…" item on the File menu.

The actual extension file should always be named with a ".i7x" suffix, meaning "I7 extension" - for instance, "Following People.i7x".

To provide an example, Emily Short's useful extension "Locksmith" is one of a small number of extensions which come ready-installed as part of the basic Inform package, and need not be downloaded and installed.

Each time that Inform translates any source text, it performs a quick check of the extensions available, and updates its own internal records. A directory of the extensions currently installed can be found by clicking on "Installed Extensions" from the Extensions panel. This is also worth visiting in order to browse the Public Library, a selection of extensions contributed by Inform users.

§2.11 Including extensions

We talk about "including" such an extension into a work of IF because the process merges rules and behaviours from the extension with those we have described ourselves. It's not uncommon for contributions by five or six different people to be pooled together this way.

Including an extension is only a matter of writing a single sentence in the source. For instance:

```
Include Locksmith by Emily Short.
```

Note that it is compulsory to name both extension and author.

Many extensions come with their own documentation. Again, follow the "Installed Extensions" link to see what's available from them.

§2.12 Use options

One more preliminary. Inform has a small number of optional settings which affect the result of translating the source. The sentence:

```
Use American dialect.
```

makes the resulting work of IF use American spellings (except where the designer spells otherwise) and the American convention for spelling out numbers (thus, "one hundred seventeen" not "one hundred and seventeen"). Similarly:

```
Use the serial comma.
```

uses a comma when printing lists: thus "Julian, Dick, George, and Anne" rather than "Julian, Dick, George and Anne". A more profound change is made by

```
Use scoring.
```

which introduces the concept of a numerical score - something which modern authors of interactive fiction often feel is inappropriate, which is why Inform only provides it on request. Two alternative options:

```
Use full-length room descriptions.
Use abbreviated room descriptions.
```

change the normal way room descriptions are shown: normally they are given in full, but in abbreviated mode, they're never given. (The latter is a bad idea in any publicly released story, but is provided for completeness and in case it may help testing.) Alternatively, we can set the traditional Infocom-style of room description to any of VERBOSE, BRIEF and SUPERBRIEF:

```
Use VERBOSE room descriptions.
Use BRIEF room descriptions.
Use SUPERBRIEF room descriptions.
```

The default is now VERBOSE, but until 2010 it was BRIEF.

Next we have:

```
Use undo prevention.
```

which disables the UNDO verb, both in play and after death, for the benefit of stories which are heavily randomised and where we do not want players to keep on UNDOing until they get a random outcome which is to their taste. (Many players consider UNDO to be their birthright, and that any work using this option is an abomination: indeed, it has even been suggested that this section of the Inform documentation be censored. To use

the option is to court controversy if not outright hostility.)

We can combine any number of options in a single "Use" sentence, so for example:

```
Use American dialect and the serial comma.
```

brings about both of these changes.

§2.13 Administering classroom use

Inform is increasingly used in education, where teachers sometimes need to install it on a whole room of computers at once, and want to monitor their students' progress. There is no special "classroom" version of Inform, but a couple of small administration features in the standard Inform - usually never needed - might be helpful to teachers.

When Inform starts up, it now looks for a file called Options.txt inside the user's home folder for Inform. (On Mac OS X, this is "~/Library/Inform"; on Windows, "My Documents\Inform", and so on.) If the file is present, then the text in it is added to the source text of everything Inform translates.

This must be used only to set use options, specify test commands, and give release instructions. For example, the following is a valid "Options.txt":

```
Use American dialect.
Test fish with "fish/fish with pole/angle".
Release along with source text.
```

The idea is that this file can be used for setting up a standard configuration on multiple machines in a classroom setting. Here the instructor can make sure the Release button will do what she would like, and can arrange for each student's copy of Inform to respond to given Test commands: for instance, if the class has an assignment to create a simulation of a camera, the instructor could set up "Options.txt" so that TEST CAMERA would run through some commands the camera ought to respond to.

A new use option, "Use telemetry recordings.", causes Inform to copy its outcome and problem messages to files in its home folder (see above) as they occur. These files are dated, so that for instance

```
Telemetry 2009-03-25.txt
```

contains all of the recorded activity on 25 March 2009. Telemetry only records the contents of the "Problems" panel - notes of success or failure, and problem messages -

and nothing is transmitted via any network, so it isn't really surveillance. The user can deliberately add a note to the current telemetry file by writing something like this in source text:

```
* "I don't get it! What's a kind? Why can't the lamp be lighted?"
```

(This is a way to make a note for the benefit of someone who will read the telemetry file - for instance, to comment on a problem message that has just appeared. Note the double-quotes. Otherwise, it's meant to look like the standard way that beta-testers mark up IF transcripts.)

These two features have been added in response to requests from education users. Let's suppose that Mr Lebling, who teaches 5th grade in Minnesota, wants to set things up just right for his class. He installs Inform on the ten computers they will use, and also copies an Options.txt file from his memory stick onto each one. The Options.txt file reads:

```
Use serial comma.
Use American dialect.
Use telemetry recordings.
```

Now Mr Lebling's class won't be confronted with English spellings, and so on. And most of the kids are happy, but Mr Lebling gets the feeling that young Marc wasn't really paying attention, so after class he checks that day's Telemetry file for that computer to see what Marc was up to, and whether he was stuck on something.

§2.14 Limits and the Settings panel

No computer has unlimited capacity, and a large, complex project may eventually bump its head against the ceiling.

Inform is a system for translating textual descriptions of interactive fiction into "story files". No single format of story file is standard to the IF community. The formats developed over the history of IF differ in three key respects:

```
- the range of computers or devices capable of playing them;
- how large they are, that is, how much play they can express;
- what extra-textual effects they can bring off.
```

Inform can write to two different formats. Neither of these is proprietary, and neither was created by the authors of Inform: each format is a community property, defined by published standards documents. An individual Inform project can make its own choice of story file format, using that project's Settings panel. Outside the Inform app, Inform can

even be used at the command line to generate C programs rather than story files, and those can be compiled to run on almost any computer.

Newly created projects are set up with the Glulx format. This has largely taken over from an earlier format called the Z-machine, but Inform can still generate a version 8 Z-machine file (a so-called "z8") if required. The Z-machine is of historic importance, and may continue to be useful for certain tasks where Glulx support is not yet available, but most users will want to keep the Glulx format set all of the time.

Internally, the Inform application uses a tool called Inform 6 (which was once the entire Inform system) as the final stage in manufacturing the story file. Inevitably, though, this can go wrong if the story is so large or complex that it exceeds some fundamental limitation of the current story file format. This is only likely to happen with the Z-machine format, since Glulx has a huge capacity; so the cure here is to switch to Glulx in the Settings. But if that's not possible for some reason - say, if we want a story file playable on a tiny handheld computer unable to manage Glulx - we still have a few options. Unless the story is very large (in which case there is little we can do), the "z8" format is most likely to be exhausted for lack of what is called "readable memory", with a message like so:

```
This program has overflowed the maximum readable-memory size of the Z-machine format. See the memory map below: the start of the area marked "above readable memory" must be brought down to $10000 or less.
```

followed by a tabulation of how the Z-machine's storage has been used, a large but not very useful diagram. The first time one runs into the problem on a large project, it can be postponed, by adding the following to the source:

```
Use memory economy.
```

(Economy cuts down the verbosity of some of the testing commands, but otherwise subtracts no performance.) Writing this into the source is the equivalent of a diver switching to an emergency oxygen tank: it gives us a generous safety margin, but also tells us that now is the time to wrap things up.

If we hit the problem again, genuine cuts must be made. As a general rule, the most memory-expensive ingredients of an Inform design are various-to-various relations between large kinds such as "thing" or, if there are many rooms, "room". Other than that, if a kind has been festooned with new properties and we have created dozens of items of that kind, then we can get a fairly large saving simply by doing without one of those properties; and so on.

The ultimate memory-saving device, of course, is the one used by book publishers when there are too many pages to bind: to cut the design into two stories, Part I and Part II.

§2.15 What to do about a bug

All software has bugs, and Inform is no exception. The most obvious bugs are the ones which Inform catches itself, when it confesses that it has halted in failure, or translated the source text into a program which cannot be compiled further. But sometimes it will also happen that Inform will issue a misleading Problem message, or appear to work normally but to produce a story which does not do what it should have done.

It is very helpful for users to report faults, so that the program can be improved for everyone else. To report a fault, please first check with the Inform home page to make sure that the version of Inform you have used to detect the fault is the latest version available. You can find the latest versions at

```
inform7.com/downloads/
```

If the bug is still present in the latest version, please report the bug using Inform's bug tracking database. Links for this can be found from the Inform source code page:

```
github.com/ganelson/inform
```

It may be that someone else has already identified the bug and even that a workaround for users is suggested. If not, please make an account at the bug tracking system and submit the requested information to help Inform's maintainers track and fix the fault.

§2.16 Does Inform really understand English?

No. No computer does, and Inform does not even try to read the whole wide range of text: it is a practical tool for a particular purpose, and it deals only with certain forms of sentence useful to that purpose. Inform source text may look like "natural language", the language we find natural among ourselves, but in the end it is a computer programming language. Many things which seem reasonable to the human reader are not understood by Inform. For instance, Inform understands

```
something which is carried by the player
```

but not (at present, anyway)

```
something which the player carries
```

even though both are perfectly good English. So it is not always safe to assume that Inform will understand any reasonable instruction it is given: when in doubt, we must go back to the manual.

More philosophically, to "understand" involves contextual knowledge. Just because Inform recognises and acts on a sentence, does it really understand what we meant? It will turn out that Inform is both good and bad at this. For instance, from

```
Mr Darcy wears a top hat.
```

Inform will correctly deduce that Darcy is a person, because inanimate objects do not ordinarily wear clothes, and that the top hat is clothing. But it will not automatically know that Darcy is a man rather than a woman because it does not know the social convention implied by "Mr". Moreover, if instead we had written

```
Mr Darcy carries a top hat.
```

then Inform would not guess that the top hat is clothing. This is because it does not have the vast vocabulary and experience of a human reader: it is probably discovering the word "hat" for the first time.

Finally, it is best to avoid ambiguities rather than rely on Inform to know which meaning is patently absurd. For instance, in

```
Heatwave bone breaks clog hospital.
```

(a headline once printed by the *Oxford Mail* newspaper) a human reader quickly realises that there is no clog hospital being broken. But if Inform had been taught the verbs *to break* and *to clog* then that is exactly the conclusion it would have drawn. Or an example which genuinely arose in beta-testing:

```
The life support unit fits the egg.
```

in which Inform construed the verb as *support* and not *fits*, and then created items called "the life" (plural) and "unit fits the egg".

That disclaimer completes the groundwork, and we are ready to begin on simulating a world to explore.

3. Things

§3.1 Descriptions
§3.2 Rooms and the map
§3.3 One-way connections
§3.4 Regions and the index map
§3.5 Kinds
§3.6 Either/or properties
§3.7 Properties depend on kind
§3.8 Scenery
§3.9 Backdrops
§3.10 Properties holding text
§3.11 Two descriptions of things
§3.12 Doors
§3.13 Locks and keys
§3.14 Devices and descriptions
§3.15 Light and darkness
§3.16 Vehicles and pushable things
§3.17 Men, women and animals
§3.18 Articles and proper names
§3.19 Carrying capacity
§3.20 Possessions and clothing
§3.21 The player's holdall
§3.22 Food
§3.23 Parts of things
§3.24 Concealment
§3.25 The location of something
§3.26 Directions

§3.1 Descriptions

At its simplest, the interactive fiction will be simulating a physical world to explore. The forerunner of today's IF is generally agreed to be a computer simulation by Will Crowther of the exploration of a cave system in the Mammoth and Flint Ridge chain of caves in Kentucky, a part of which might be described in Inform thus:

```
"Cave Entrance"

The Cobble Crawl is a room. "You are crawling over cobbles in a low passage. There is a dim light at the east end of the passage."

A wicker cage is here. "There is a small wicker cage discarded nearby."

The Debris Room is west of the Crawl. "You are in a debris room filled with stuff washed in from the surface. A low wide passage with cobbles becomes plugged with mud and debris here, but an awkward canyon leads upward and west. A note on the wall says, 'Magic word XYZZY'."

The black rod is here. "A three foot black rod with a rusty star on one end lies nearby."

Above the Debris Room is the Sloping E/W Canyon. West of the Canyon is the Orange River Chamber.
```

Here we sketch in four of Crowther's locations, and two objects: just enough to be able to walk around the caves and pick up the rod and the cage. The text in quotation marks will appear verbatim as paragraphs shown to the player as the caves are explored. The first paragraph, as we have seen, is the title of the work. The other quotations describe the places and objects introduced.

If we play this story, we find that we can type TAKE CAGE or TAKE WICKER CAGE, for instance, but not TAKE SMALL CAGE. Inform saw that we called this "a wicker cage" when it first appeared in the source text, and assumed that the player would call it that, too. (Whereas it didn't look inside the descriptive text to allow for TAKE SMALL CAGE or TAKE DISCARDED CAGE or TAKE NEARBY CAGE.) A small limitation here is that probably only the first 9 letters of each word are read from the player's command. This is plenty for handling the wicker cage and the black rod, but it might be embarrassing at a meeting of the Justice League to find that KISS SUPERHERO and KISS SUPERHEROINE read as if they are the same command.

So we have already found that Inform has made some assumptions about what we want, and imposed some limitations on how much computational effort to go to when the work of IF is finally played. If Inform guesses what we need wrongly, we need to know more advanced features of the language in order to overcome these problems. (We shall see how to change the way the player's commands are read in the chapter on Understanding.)

This is often how Inform works: make the standard way of doing things as simple as possible to describe, but allow almost any behaviour to be altered by more elaborate source text. As an example of that, the player begins in the Cobble Crawl because it was the first room created in the source text, but we could instead have written text like:

```
The player is in the Cobble Crawl.
```

to override that. This can make the source text easier to follow if the rooms are sometimes being created in a less obvious way. For example, if we write:

```
The silver bars are in the Y2 Rock Room.
The Cobble Crawl is a room. South of the Crawl is Y2.
```

then the first room to be created will actually be the Y2 Rock Room, so that's where the player will be starting unless we say otherwise.

Examples

2. **Bic**
   Testing to make sure that all objects have been given descriptions.
   RB 13.1 Testing

3. **Verbosity 1**
   Making rooms give brief room descriptions when revisited.
   RB 6.4 Looking

4. **Slightly Wrong**
   A room whose description changes slightly after our first visit there.
   RB 3.1 Room Descriptions

§3.2 Rooms and the map

Rooms are joined together at their edges by "map connections", most of which are pathways in one of the eight cardinal compass directions: north, northeast (written without a hyphen), east, southeast, south, southwest, west, northwest. We also have up and down, suitable for staircases or ladders. In real life, people are seldom conscious of their compass bearing when walking around buildings, but it makes a concise and unconfusing way for the player to say where to go next, so is generally accepted as a convention of the genre.

Two more directions are provided by Inform: "inside" and "outside". These are best used when one location is, say, a meadow and the other is a woodcutter's hut in the middle of it; we might then say

```
Inside from the Meadow is the woodcutter's hut.
```

The "from" is important, as it clarifies that we intend to link two different locations, not

to create an item - the hut - in a single location - the meadow.

A problem which sometimes arises when laying out maps is that Inform allows short forms of room names to be used as abbreviations. This is usually a good idea, but has unfortunate results if we write:

```
The Airport Road is west of the Fish Packing Plant. The Airport is west of the Airport Road.
```

…because "Airport" is taken as a reference to "Airport Road", so Inform makes only two locations, one of which supernaturally leads to itself. We can avoid this by writing:

```
The Airport Road is west of the Fish Packing Plant. A room called the Airport is west of the
Airport Road.
```

Using "called" is often a good way to specify something whose name might give rise to confusion otherwise. It always makes something new, and it is also neatly concise, because we can establish something's kind and name in the same sentence. As another example, suppose we want to create a room called "South of the Hut", to south of the Hut. We can't do so like this:

```
South of the Hut is a room. South of the Hut is south of the Hut.
```

…because Inform will read that first sentence as placing a (nameless) room to the south of a room called "Hut". Once again "called" can save the day:

```
South of the Hut is a room called South of the Hut.
```

It is best to use "called" in the simplest way possible, and in particular, best not to use "called" twice in the same sentence. Consider:

```
The kitchen cabinet contains a container called a mixing bowl and a portable supporter
called a platter.
```

It is unlikely that anyone would want to name something "a mixing bowl and a portable supporter called a platter", but not impossible, and Inform tends not to be a good judge of what is likely.

(If we really want to get rid of this issue once and for all, starting the source text with the use option "Use unabbreviated object names." will do it, but the effect is drastic. This instructs Inform not to recognise names other than in full. For example:

```
West of the Kitchen is the Roaring Range. South of the Range is the Pantry.
```

is ordinarily read by Inform as constructing three rooms (Kitchen, Roaring Range, Pantry); but with this use option set, it makes four (Kitchen, Roaring Range, Range, Pantry), in two disconnected pieces of map. Handle with care.)

Examples

5. **Port Royal 1**
   A partial implementation of Port Royal, Jamaica, set before the earthquake of 1692 demolished large portions of the city.
   RB 3.2 Map

6. **Up and Up**
   Adding a short message as the player approaches a room, before the room description itself appears.
   RB 6.9 Going, Pushing Things in Directions

7. **Starry Void**
   Creating a booth that can be seen from the outside, opened and closed, and entered as a separate room.
   RB 3.3 Position Within Rooms

§3.3 One-way connections

Connections are ordinarily two-way, but do not have to be. One of the map connections in the Mammoth Cave simulation was made by the sentence:

```
The Debris Room is west of the Crawl.
```

Besides reading this sentence at face value, Inform also deduced that the Crawl was probably meant to be east of the Debris Room: in other words, that the path between them is a two-way one. When Inform makes guesses like this, it treats them as being less certain than anything explicitly stated in the source. Inform will quietly overturn its assumption if information comes to hand which shows that it was wrong. That might happen in this case if another sentence read:

```
The Hidden Alcove is east of the Debris Room.
```

These two sentences are not contradictory: Inform allows them both, simply accepting that the world is more complicated than it first assumed. There are relatively few situations where Inform has to make educated guesses, but when it does, it tries always

to follow Occam's Razor by constructing the simplest model world consistent with the information in the Source text.

We can even explicitly make a route which turns around as it leads between two rooms:

```
West of the Garden is south of the Meadow.
```

If we want to establish a route which cannot be retraced at all, we can specify that a particular direction leads nowhere:

```
East of the Debris Room is nowhere.
```

Finally, note that Inform's assumptions about two-way directions are only applied to simple sentences. When the source text seems to be saying something complicated, Inform takes it as a precise description of what's wanted. So, for example, in:

```
The Attic is above the Parlour.
The Attic is a dark room above the Parlour.
```

Inform makes guesses about the first sentence, and makes a two-way connection; but it accepts the second sentence more precisely, with just a one-way connection.

Examples

8. **Port Royal 2**
   Another part of Port Royal, with less typical map connections.

   RB 3.2 Map

9. **The Unbuttoned Elevator Affair**
   A simple elevator connecting two floors which is operated simply by walking in and out, and has no buttons or fancy doors.

   RB 8.2 Ships, Trains and Elevators

§3.4 Regions and the index map

Rooms represent individual places to which one can go, but we tend to think of the world around us in larger pieces: we think of a house and a garden, rather than each of the single rooms of the house and all corners of its garden. To Inform a collection of rooms is called a "region", and we can create one like so:

```
The Arboretum is east of the Botanical Gardens. Northwest of the Gardens is the Tropical Greenhouse.

The Public Area is a region. The Arboretum and Gardens are in the Public Area.
```

The real usefulness of creating regions like "Public Area" will only appear later, when we begin defining rules of play which apply in some areas but not others, but in the mean time we can see the effect by turning to the World tab of the Index. In the World Index, Inform draws a map - or at least a stylised attempt at a diagram of the rooms and their connections: this will not always correspond to how we imagine things, but with any luck it should mostly be right.

Rooms are represented by coloured squares, and the colour-coding is done by region. In the above example, the two "Public Area" rooms are coloured green (as it happens); the Greenhouse, since it belongs to no region, is a neutral grey.

Regions can be put inside each other:

```
The University Parks is a region. The Public Area is in the University Parks.
```

but they are not allowed to overlap other than by one being entirely inside the other.

See Also

Improving the index map for ways to adjust the way the index map is drawn or exported for publication.

Example

10. **Port Royal 3**
    Division of Port Royal into regions.

RB 3.2 Map

§3.5 Kinds

The following description runs to only 33 words, but makes a surprisingly intricate design. It not only places things within rooms, but also places them very specifically with respect to each other:

```
"Midsummer Day"

East of the Garden is the Gazebo. Above is the Treehouse. A billiards table is in the Gazebo. On it is a trophy cup. A starting pistol is in the cup.
```

Inform needs to identify the places and objects being described by the nouns here, and to guess what it can about them. For instance, the pistol can be picked up but not walked inside, whereas the Treehouse is the reverse. (This is obvious to someone who knows what these words mean, less obvious to a computer which does not, but the text contains sufficient clues.) Inform does this by sorting the various nouns into different categories, which are called "kinds". For instance:

```
Garden, Gazebo, Treehouse - room
billiards table - supporter
cup - container
starting pistol - thing
East, up (implied by "above") - direction
```

(A container is something which can contain other things, and a supporter similarly.) For instance Inform knows that if one thing is in another, then the second thing is either a room or a container, and if one thing is on another, the second thing is a supporter. This worked nicely for the design above, but:

```
In the Treehouse is a cardboard box.
```

results in the cardboard box being made only a "thing": because nothing has been put inside it, there is no reason for Inform - which does not know what a cardboard box looks like - to guess that it is a "container". So we need to add:

```
The box is a container.
```

It is rather clumsy to have to write two sentences like this, so we would normally write this instead:

```
In the Treehouse is a container called the cardboard box.
```

Examples

11. **First Name Basis**
    Allowing the player to use different synonyms to refer to something.
    RB 2.2 Varying What Is Read

12. **Midsummer Day**
    A few sentences laying out a garden together with some things which might be found in it.
    RB 1.1 Preface

§3.6 Either/or properties

Some containers, like bottles, can be opened: others, like buckets, cannot. If they can be opened, then sometimes they will be open, and sometimes closed. These are examples of properties, which can change during play. The following source sets some properties:

```
The cardboard box is a closed container. The glass bottle is a transparent open container.
The box is fixed in place and openable.
```

There are only four different properties referred to here. Closed means not open, and vice versa, so these two adjectives both refer to the same property. (As might be expected, when a container is open, one can see inside and place things within, or take them out.) The glass bottle and the box being containers is a matter of their kinds, which is something fundamental and immutable, so "container" does not count as a property.

A "transparent" container is one which we can see inside even when it is closed, and the opposite is an "opaque" container.

The property of being "fixed in place" ensures that the player cannot pick the item up and walk away with it: this is useful for such things as oak trees or heavy furniture. The opposite condition is to be "portable".

A container which is "openable" can be opened or closed by the player; as might be expected, the opposite is "unopenable".

With a really large cardboard box, we might imagine that the player could get inside: such a container should be declared "enterable".

Example

13. **Tamed**
    Examples of a container and a supporter that can be entered, as well as nested rooms.

RB 8.4 Furniture

§3.7 Properties depend on kind

Properties depend very much on kind. It makes no sense to ask whether a room is transparent or opaque, for instance, so Inform will not allow this either to be specified or queried.

Another way that kind influences properties can be seen from an earlier example:

```
The Gazebo is a room. A billiards table is in the Gazebo. On it is a trophy cup. A starting pistol is in the cup.
```

The cup, the pistol and the table are all allowed to have the "fixed in place" property, but in fact only the table actually has it: the cup and the pistol are created as "portable" instead. This is because Inform knows that most things are portable, but that supporters - such as the table - are usually fixed in place. If this assumption is wrong, we need only add the line:

```
The table is portable.
```

Example

14. **Disenchantment Bay 1**
    A running example in this chapter, Disenchantment Bay, involves chartering a boat. This is the first step: creating the cabin.

RB 1.3 Disenchantment Bay

§3.8 Scenery

As we have just seen, making something "fixed in place" will prevent it from being picked up or moved. But it remains substantial enough to be described in its own paragraph of text when the player visits its location. This can be unfortunate if it has also been described already in the body of the main description for that location. For instance, if we wrote:

```
The Orchard is a room. "Within this quadrille of pear trees, a single gnarled old oak remains as a memory of centuries past." The gnarled old oak tree is fixed in place in the Orchard.
```

This would end up describing the oak twice, once in the paragraph about the Orchard, then again in a list of things within it:

```
Orchard
Within this quadrille of pear trees, a single gnarled old oak remains as a memory of centuries past.

You can see a gnarled old oak tree here.
```

We avoid this by making it "scenery" instead of "fixed in place":

```
The gnarled old oak tree is scenery in the Orchard.
```

Any thing can be scenery, and this does not bar it from playing a part in the story: it simply means that it will be immobile and that it will not be described independently of its room. Being immobile, scenery should not be used for portable objects that are meant to be left out of the room description.

If a supporter is scenery, it may still be mentioned in the room description after all, but only as part of a paragraph about other items, such as

```
On the teak table are a candlestick and a copy of the Financial Times.
```

If the player takes the candlestick and the Times, the teak table will disappear from mention. (Scenery containers do not behave in this way: their contents are assumed to be less immediately visible, and will be mentioned only if the player looks inside them.)

Examples

15. **Disenchantment Bay 2**
    Disenchantment Bay: creating some of the objects in the cabin's description.
    RB 1.3 Disenchantment Bay

16. **Replanting**
    Changing the response when the player tries to take something that is scenery.
    RB 6.8 Taking, Dropping, Inserting and Putting

§3.9 Backdrops

It is a cardinal rule that nothing can be in more than one place at the same time, but rules were made to be broken, and an exception is allowed for a special kind of thing called a "backdrop". For instance:

```
"Streaming"

The Upper Cave is above the Rock Pool.

The stream is a backdrop. It is in the Upper Cave and the Rock Pool.
```

Backdrops are ordinarily in the background: if the sky needed to be referred to in the course of play, it might be represented by a backdrop, for instance. Here we have a stream of water running through two rooms, though it might be any number. Backdrops are always fixed in place.

Backdrops can be put in regions as well as rooms, and if so, then they are present at every room in the given region (or regions), as well as any specific rooms they may also be put into. For instance:

```
The Outdoors Area is a region. The Moon is a backdrop. The Moon is in the Outdoors Area.
The Moon is in the Skylight Room.
```

The special place "everywhere" can be given as the location of a backdrop to make it omnipresent:

```
The sky is a backdrop. The sky is everywhere.
```

Inform assumes that backdrops are also scenery unless told otherwise, so this will not result in messages like "You can also see the sky here." being included in room descriptions. In the case of the stream above, we could artfully mention it in passing in the room descriptions of the Upper Cave and the Rock Pool.

See Also

Moving backdrops for ways to place backdrops in dynamically changing selections of rooms.

Example

17. **Disenchantment Bay 3**
    Disenchantment Bay: adding a view of the glacier.

RB 1.3 Disenchantment Bay

§3.10 Properties holding text

The properties we have seen so far have all been either/or: either open or closed, either transparent or opaque, either fixed in place or portable, either openable or not openable. However, some properties can have a much wider range of possibilities. For instance, the "description" of a room is the text revealed when the player first enters it, or types "look". This needs to be textual: Inform would complain if, for instance, we tried to set the description of something to the number 42. We have already seen a concise way to set the description of a room:

```
The Painted Room is north of the Undertomb. "This is the Painted Room, where strange wall drawings leap out of the dark at the gleam of your candle: men with long wings and great eyes, serene and morose."
```

This does the same thing as:

```
The Painted Room is north of the Undertomb. The description of the Painted Room is "This is the Painted Room, where strange wall drawings leap out of the dark at the gleam of your candle: men with long wings and great eyes, serene and morose."
```

Or even:

```
The Painted Room is north of the Undertomb. The description is "This is the Painted Room, where strange wall drawings leap out of the dark at the gleam of your candle: men with long wings and great eyes, serene and morose."
```

§3.11 Two descriptions of things

The player's first sight of something is the text used as its "initial appearance":

```
The plain ring is here. "Cast aside, as if worthless, is a plain brass ring."
```

This text appears as a separate paragraph in the text describing the Painted Room. It will continue to be used until the first time player picks the ring up (if this ever happens), so it normally describes things in their original, undisturbed context. (Inform uses an either/or property called "handled" for this: something is "handled" if it has at some point been held by the player.)

Thus when a piece of text stands alone as a sentence in its own right, then this is either the "description" of the most recently discussed room, or the "initial appearance" of the most recently discussed thing. Either way, it is used verbatim as a paragraph in the text shown to the player visiting the room in question.

But a thing also has an ordinary "description", which is used to give a close-up look at it. This text is ordinarily only revealed to the player when a command like "examine ring" is keyed in:

```
The description of the plain ring is "No better than the loops of metal the old women use for fastening curtains."
```

See Also

Creating a scene for the description of a scene, which is set in the same way.

Examples

18. **Disenchantment Bay 4**
    Disenchantment Bay: fleshing out the descriptions of things on the boat.
    RB 1.3 Disenchantment Bay

19. **Laura**
    Some general advice about creating objects with unusual or awkward names, and a discussion of the use of printed names.
    RB 2.2 Varying What Is Read

§3.12 Doors

The map of an interactive fiction is the layout of rooms and the entrances and exits which connect them. So far, these map connections have always run from one room to another, like so:

```
The Painted Room is north of the Undertomb.
```

However, we can also interpose doors between rooms, like so:

```
The heavy iron grating is east of the Orchard and west of the Undertomb. The grating is a door.
```

The second sentence is needed since otherwise Inform will take "heavy iron grating" to be the name of a third room, whereas what we want is for the grating to be something physically present in both the Orchard and in the Undertomb, and acting as a conduit between them. To this end it needs to be a "door", a kind we have not so far seen. In the absence of any other instruction, a newly created door will be fixed in place, closed and openable.

The grating really does come in between the two rooms: the grating is what lies immediately east of the Orchard, not the Undertomb room. So if we wrote the following:

```
The Undertomb is east of the Orchard. The heavy iron grating is east of the Orchard and west of the Undertomb. The grating is a door.
```

then Inform would say that this is a contradiction: we said the Undertomb was east of the Orchard, but then we said that the grating was east of the Orchard.

Inform's "door" kind can be used for all manner of conduits, so the word door need not be taken literally. In Ursula K. Le Guin's beguiling novel "The Tombs of Atuan", from which the above rooms are stolen, it is not a grating which interposes, but:

```
The red rock stair is east of the Orchard and above the Undertomb. The stair is an open door. The stair is not openable.
```

In real life, most doors are two-sided, and can be used from either of the rooms which they join, but this is not always convenient for interactive fiction. Here is a one-sided door:

```
The blue door is a door. It is south of Notting Hill. Through it is the Flat Landing.
```

(Note the use of "it" here as an optional abbreviation.) This will make a door visible only on the Notting Hill side; no map connection will be made in the reverse direction, unless we ask for one.

So much for creating and describing individual doors. Once we need to write about doors in general, we are likely to want a way to find out where a given door sits in the map. The following phrases reveal this:

front side of (object) ⇒ *room*

This phrase produces the first of the one or two rooms containing a door - first in the order given in the source text. Example: if

```
The red rock stair is east of the Orchard and above the Undertomb.
```

then "front side of the red rock stair" produces the Orchard. For a one-sided door, this produces the only room containing the door.

back side of (object) ⇒ *room*

This phrase produces the last of the one or two rooms containing a door - last in the order given in the source text. Example: if

```
The red rock stair is east of the Orchard and above the Undertomb.
```

then "back side of the red rock stair" produces the Undertomb. A one-sided door has no "back side."

More often, we are dealing with a door and want to know what it leads to, but that depends where we're standing:

other side of (door) **from** (room) ⇒ *object*

This phrase produces the room on the other side of the door, as seen from the given vantage point, which needs to be one of its sides. Example: if

```
The red rock stair is east of the Orchard and above the Undertomb.
```

then "other side of the red rock stair from the Undertomb" produces the Orchard, and vice versa.

direction of (door) **from** (room) ⇒ *object*

This phrase produces the direction in which the door leads, as seen from the given vantage point, which needs to be one of its sides. Example: if

```
The red rock stair is east of the Orchard and above the Undertomb.
```

then "direction of the red rock stair from the Undertomb" produces up.

See Also

Adjacent rooms and routes through the map for more phrases which can look at the current map layout.

Examples

20. **Disenchantment Bay 5**
    Disenchantment Bay: adding the door and the deck to our charter boat.
    RB 1.3 Disenchantment Bay

21. **Escape**
    Window that can be climbed through or looked through.
    RB 3.6 Windows

22. **Garibaldi 1**
    Providing a security readout device by which the player can check on the status of all doors in the game.
    RB 3.5 Doors, Staircases, and Bridges

§3.13 Locks and keys

It seems unwise for a door in Notting Hill to be unlocked, so:

```
The blue door is lockable and locked. The matching key of the blue door is the brass Yale key.
```

Since the second sentence here is a little clumsy, we can equivalently say

```
The brass Yale key unlocks the blue door.
```

Yet a third way to say this is:

```
The blue door has matching key the brass Yale key.
```

This introduces three new properties: a door can be locked or unlocked; lockable or not lockable; and it can have a matching key, which must be another thing. The same thing can be the matching key of many different locks: and note that a door can be locked and even lockable without having a matching key at all, in which case the player trying to open it will be permanently out of luck. Doors are ordinarily unlocked, not lockable, and without a matching key.

Containers can also have locks, in exactly the same way, and are allowed to have the same properties. On the other hand supporters never have locks: it makes no sense to be able to lock a tabletop, for instance, and Inform will not allow any discussion of the matching key of a supporter, or of a supporter being locked or unlocked.

Examples

23. **Disenchantment Bay 6**
    Disenchantment Bay: locking up the charter boat's fishing rods.
    RB 1.3 Disenchantment Bay

24. **Neighborhood Watch**
    A locked door that can be locked or unlocked without a key from one side, but not from the other.
    RB 3.5 Doors, Staircases, and Bridges

§3.14 Devices and descriptions

A "device" is another of the standard kinds of thing, and should be used for anything which can be switched on or off: a light switch, say, or a slide projector. Devices are generally machines, clockwork or electrical. A device is always either "switched on" or "switched off", but is switched off unless we specify otherwise.

That makes three kinds of thing which will likely change their appearance according to which of their two possible states they are in: doors and containers, which can be open or closed; and devices, which can be switched on or switched off. We would like to produce text accordingly, and we can do this using Inform's ability to make (almost) any piece of text change with circumstances. For instance:

```
The coffin is an openable container in the Undertomb. "[if open]The lid of a plank coffin yawns open.[otherwise]A plank coffin lies upon the dirt floor of the Tomb."
```

We could use a similar trick to make the appearance of a device change "if switched on". There will be much more about text substitutions, as instructions in square brackets like these are called, in later chapters.

See Also

Text with substitutions for more on varying what is printed.

Examples

25. **Disenchantment Bay 7**
    Disenchantment Bay: making the radar and instruments switch on and off.
    RB 1.3 Disenchantment Bay

26. **Down Below**
    A light switch which makes the room it is in dark or light.
    RB 3.7 Lighting

§3.15 Light and darkness

Rooms can be "dark" or "lighted", though they are lighted by default, and are lighted in all the examples we have seen so far.

```
The Sinister Cave is a dark room. "A profoundly disquieting rock formation, apparently sculptured by some demonic hand, this is not a cave in which to relax."
```

When the player is in a dark room, he can still go in various directions, but he cannot see the room description or interact with any of the objects in the room, except those he is holding. This means that, unless we should change the Cave in some way during play, the text above ("A profoundly…") will only be read if the player succeeds in bringing light into the Cave, perhaps by bringing along the following:

```
The flaming torch is in the Sandy Passage. "Stuck loosely into the sand is a flaming torch." The flaming torch is lit.
```

A thing with the property of being "lit" will enable the player to see inside dark rooms, and to carry out other activities requiring light, such as examining items. A lit thing in an open container will still light up a room; a lit thing in a closed container will not, unless the container has been given the "transparent" property.

It is possible to adjust the way darkness behaves, and we will see more on this topic in the chapter on Activities.

See Also

Printing a refusal to act in the dark for the first of several ways to control what is printed in the dark.

§3.16 Vehicles and pushable things

Next in the tour of standard kinds is the "vehicle". This behaves like (indeed, is) an

enterable container, except that it will not be portable unless this is specified.

```
In the Garage is a vehicle called the red sports car.
```

The player can enter the sports car and then move around riding inside it, by typing directions exactly as if on foot: and the story will print names of rooms with "(in the red sports car)" appended, lest this be forgotten.

We have already seen that some things are portable, others fixed in place. In fact we can also make a third sort of thing: those which, although not portable, can be pushed from one room to another with commands like "push the wheelbarrow north". At a pinch, we might just be willing to allow:

```
The red sports car is pushable between rooms.
```

But of course this is a property which almost any thing can have, not just a vehicle. (Only "almost" because Inform will not allow a door to be pushable between rooms, in the interests of realism rather than surrealism.)

If we need vehicles which the passenger sits on top of, like a horse or a tractor, the standard "vehicle" kind will not be ideal. However, by loading one of the extensions which comes ready-installed:

```
Include Rideable Vehicles by Graham Nelson.
```

…we are provided with two more kinds, "rideable vehicle" and "rideable animal", just right for the tractor and the horse respectively. (As with all extensions, the documentation can be seen by clicking Go on some source which contains the above line, and then turning to the Contents index; or from the Installed Extensions tab of the Extensions panel.)

See Also

Going by, going through, going with for further ways to customize vehicle behaviour.

Examples

27. **Peugeot**
    A journey from one room to another that requires the player to be on a vehicle.
    RB 8.1 Bicycles, Cars and Boats

28. **Disenchantment Bay** 8
    Disenchantment Bay: a pushable chest of ice for the boat.
    RB 1.3 Disenchantment Bay

29. **Hover**
    Letting the player see a modified room description when he's viewing the place from inside a vehicle.
    RB 8.1 Bicycles, Cars and Boats

§3.17 Men, women and animals

Rounding out the standard kinds provided by Inform are four for living things: "person", which is a kind of thing, and "man", "woman" and "animal", all kinds of person. For instance:

```
In the Ballroom is a man called Mr Darcy.
```

For the time being, men and women will be little more than waxworks: they will come to life only when we go beyond the present stage of creating an initial state of the world.

People can be male or female: this is an either/or property for the "person" kind, and it affects play at run-time a little, because the player can use "him" and "her" to refer to male or female people encountered. Men and women are always male and female respectively, and for animals we can choose either way, for example making a stallion male or a nanny goat female. Animals are male unless we say otherwise.

If our animal is instead something like a beetle or an earthworm, where gender doesn't seem to matter or even to exist, we can use the further property "neuter":

```
The spider is a neuter animal in the Bathroom.
```

The Standard Rules don't make people behave differently according to their genders, and the main difference comes down to language: whether we want the animal to be called "her", or "it". Because of the existence of "neuter", we sometimes need to be cautious about the use of the adjective "male": since Inform, partly for historical reasons, uses an either/or property for masculinity, neuter animals are also "male".

Example

30. **Disenchantment Bay 9**
    Disenchantment Bay: enter the charter boat's Captain.

RB 1.3 Disenchantment Bay

§3.18 Articles and proper names

Suppose we have said that:

```
In the Ballroom is a man called Mr Darcy.
```

When the Ballroom is visited, the man is listed in the description of the room as "Mr Darcy", not as "a Mr Darcy". This happened not because Inform recognised that Darcy is a proper name, or even because men tend to have proper names, but because Inform noticed that we did not use "a", "an", "the" or "some" in the sentence which created him. The following shows most of the options:

```
The Belfry is a room. A bat is in the Belfry. The bell is in the Belfry. Some woodworm are in the Belfry. A man called William Snelson is in the Belfry. A woman called the sexton's wife is in the Belfry. A man called a bellringer is in the Belfry.
In the Belfry is a man called the vicar. The indefinite article of the vicar is "your local".
```

In the resulting story, we read:

```
You can see a bat, a bell, some woodworm, William Snelson, the sexton's wife, a bellringer and your local vicar here.
```

The subtlest rule here is in the handling of "the". We wrote "The bell is in the Belfry", but this did not result in the bell always being called "the" bell: in fact, writing "A bell is in the Belfry" would have had the same effect. On the other hand, "A woman called the sexton's wife is in the Belfry." led to the wife always being known as "the" sexton's wife, not "a" sexton's wife, because Inform thinks the choice of article after "called" shows more of our intention than it would elsewhere. These rules will never be perfect in all situations, so we are also allowed to specify indefinite articles by hand, as the vicar's case shows.

"Some" is worth a closer look, because English uses it in several different ways. By introducing the woodworm with "some", above, we established that it was plural. We might imagine that there are many worms, even though they are represented by a single thing in Inform. We can expect to see text in the story such as:

```
You can see some woodworm here.
The woodworm are fixed in place.
```

But suppose we wanted something which there is an amount of, but which is not made up of individual items - a so-called mass noun like "water", or "bread". Now we can write:

```
The water is here. The indefinite article is "some".
```

and this time Inform does not treat the "some water" thing as a plural, so we might read:

```
You can see some water here.
The water is hardly portable.
```

rather than "The water are hardly portable."

Finally, we can override these settings, if they still come out not as we intend, by explicitly changing the either/or properties "singular-named" (vs "plural-named") and "proper-named" (vs "improper-named").

Examples

31. **Belfry**
You can see a bat, a bell, some woodworm, William Snelson, the sexton's wife, a bellringer and your local vicar here.
RB 7.1 Getting Acquainted

32. **Gopher-wood**
Changing the name of a character in the middle of play, removing the article.
RB 7.1 Getting Acquainted

§3.19 Carrying capacity

The containers and supporters created so far have been boundlessly capacious: or rather, though we seldom notice the difference, have had a maximum carrying capacity of 100 items. This is clearly unrealistic for a small purse or a modest mantelpiece. We can impose upper limits with sentences like so:

```
The carrying capacity of the jewelled purse is 2.

The bijou mantelpiece has carrying capacity 3.
```

Attempts by the player to overfill, or overload, will now be rebuffed with a message such as "There is no room on the mantelpiece".

The player is not a container or a supporter, but nevertheless does have a carrying capacity: this is interpreted to mean the maximum number of items which can be carried at once.

```
The carrying capacity of the player is 4.
```

These restrictions only apply to the player (and other in-world characters): as the omnipotent creators, we are not restrained by them. Nothing prevents this:

```
The carrying capacity of the jewelled purse is 2. The diamond, the ruby and the sapphire
are in the purse.
```

The player will be able to remove all three items, but only put two of them back. (This is probably something we only want very occasionally: perhaps to create a sack stuffed almost to bursting point.)

§3.20 Possessions and clothing

We have seen how to place objects in rooms, and in containers or on supporters. But what about people? Perhaps it could be said that they "contain" the fillings in their teeth, or "support" a top hat, but this is not very natural. Inform therefore never speaks of things being "in" or "on" people. Instead, they have two sorts of possessions: the things they carry, and the things they wear. (Body parts, such as arms and legs, are different again: see "parts" below for a clue to how to do these.) Thus:

```
Mr Darcy wears a top hat. Mr Darcy carries a silver sword.
```

In fact, Inform deduces from this not only who owns the hat and the sword, but also that Darcy has the kind "person", because only people can wear or carry.

As all the assertion verbs do, "to wear" and "to carry" have participles which Inform knows about. So we could equally well write:

```
The scarlet coat is worn by Mr Wickham. The duelling pistol is carried by Mr Wickham.
```

If we do not specify who does the wearing, or carrying, then this is assumed to be the player. Thus:

```
A brass lantern and a rusty iron key are carried. The mosquito-repellent hat is worn.
```

It would make no sense to "wear" the key, for instance, so Inform needs to distinguish

between what is clothing and what is not. It does this with an either/or property called "wearable": if something has this property then the player will be allowed to wear it, provided it can first be picked up. Anything which is worn by somebody at the start of play is assumed to be wearable (unless we say otherwise). But if nobody is initially wearing the item in question, then we have to be explicit:

```
The player carries a scarlet gown. The gown is wearable.
```

(When we come to asking questions about the current situation, we will need to remember that "to carry" and "to wear" are different. Thus "if Lancelot carries the plate armour" will not be true if he is wearing it rather than carrying it under his arm. As we will later see, we can instead vaguely say "if Lancelot has the plate armour" to mean either carrying or wearing.)

See Also

To carry, to wear, to have for a more detailed explanation of carrying, wearing, and possessing as Inform understands them.

Example

33. **Disenchantment Bay 10**
    Disenchantment Bay: things for the player and the characters to wear and carry.

RB 1.3 Disenchantment Bay

§3.21 The player's holdall

When the player has only limited carrying capacity, play is likely to be tiresome, but we can make life easier by providing a way for the player to carry endless items without dozens of free hands to hold them all:

```
"Sackcloth"

The Attic is a room. The old blue rucksack is a player's holdall. The player is wearing the rucksack.

The carrying capacity of the player is 3.

In the Attic are a CD entitled No Smoke Without Fire, a 70s photograph of an American winning Wimbledon, a fraxinus branch, an urn holding your late great-aunt's remains, a convention badge from the American Society of Hypertension and a ghost story by M R James.
```

This example story introduces a new kind of container, the "player's holdall". This is a

kind of which most stories will contain at most one example, but in principle there can be any number. A player's holdall is a capacious bag into which the player automatically places surplus items whenever his or her hands are full: trying the above example story and getting the items one by one will give the general idea.

Of course, if the carrying capacity of the player is never reached then there will never be any surplus items and a player's holdall will behave just like any other (portable, usually openable) container.

See Also

Units for the tools to implement a more sophisticated capacity system.

Example

34. **Disenchantment Bay 11**
    Disenchantment Bay: making a holdall of the backpack.

RB 1.3 Disenchantment Bay

§3.22 Food

We have nearly reached the end of the chapter on Things, but one either/or property for things remains: every thing is either "edible" or "inedible". Unless we say otherwise, things are inedible. But for instance we might write:

```
The player carries a Macintosh apple. The Macintosh is edible.
```

(The type of computer is named after a variety of apple descended from a tree cultivated in 1811 by John McIntosh of Ontario.) Edible things are just like inedible ones, except that the player can EAT them. This will usually only consume the foodstuff in question, effectively destroying it, but using techniques from later chapters we could make the consequences more interesting.

§3.23 Parts of things

Everything has one and only one kind. This is both good and bad: good for clarity, bad if something needs to behave in two different ways at once. How might we simulate a car with an ignition key, given that no single thing can be both a "vehicle" and a "device" at the same time?

The Inform world model takes the view that such a car is too complicated to be simulated with a single thing. Instead it should be simulated as a vehicle (the car) which has a device (the ignition) attached. This is done using a third kind of containment to those

seen so far ("in…" and "on…"): "part of".

```
"Buttons"

The Confectionary Workshop is a room. The Chocolate Machine is here. "The Chocolate Machine has pride of place. A lever and two buttons, one white, the other brown, seem to be the only controls. On top is a hopper."

A container called the hopper is part of the Chocolate Machine. The lever, the white button and the brown button are parts of the Chocolate Machine.

The Chocolatier's desk is here. "The Chocolatier evidently works at the imposing green-leather topped desk facing the Machine. It has three drawers with brass handles."

The upper drawer, the middle drawer and the lower drawer are parts of the desk. The upper drawer, the middle drawer and the lower drawer are openable closed containers. In the middle drawer is a sugared almond. In the lower drawer is a Battenburg cake. On the desk is a liquorice twist.

The cake, the twist and the almond are edible.
```

The machine and the desk each have several "parts" representing subsidiary pieces of themselves. The desk is a "supporter" (it needs to be, for the liquorice twist to be on top) but also has three "containers" attached, each of which can be opened or closed independently.

In the interests of realism, the standard rules of play protect these composite things. Thus if the desk were to be moved elsewhere (rolling on sugar casters perhaps) then its parts would move with it, and the player is not allowed to detach parts of things: the drawers can be opened or closed, but not pulled out altogether.

Note that rooms and regions are not allowed to have parts. (Rooms are already parts of regions, and to divide up rooms, we can either make several rooms or place containers or other obstacles in a single one.)

Examples

35. **Fallout Enclosure**
    Adding an enclosure kind that includes both containers and supporters in order to simplify text that would apply to both.
    RB 9.2 Bags, Bottles, Boxes and Safes

36. **Brown**
    A red sticky label which can be attached to anything in the game, or removed again.
    RB 9.7 Painting and Labeling Devices

37. **Disenchantment Bay 12**
    A final trip to Disenchantment Bay: the scenario turned into a somewhat fuller scene, with various features that have not yet been explained.
    RB 1.3 Disenchantment Bay

§3.24 Concealment

Though realism can become tiresome in interactive fiction, there are times when we cannot go along with Inform's normal assumption that all of a person's possessions are visible to everybody else. People are not like containers, which either show all of their holdings or not, according to whether they are open or transparent. If a man is carrying a fishing rod and a wallet, one will be on open show, the other not. Some clothing is outwardly visible, but not all.

Whether or not something is concealed is not like the either/or properties we have seen so far - such as being "open" or "closed" - because it is not really a property of the thing itself, but depends on the habitual behaviour of its current owner. To talk about behaviour we have to use sentences of a kind not seen so far, and which will not fully be explained for some chapters to come.

But straightforward cases are easy to write, if only by imitating the following examples.

Here we make the Cloaked Villain invariably conceal anything she is holding or wearing:

```
Rule for deciding the concealed possessions of the Cloaked Villain: yes.
```

At which point we think about it more carefully, and then rewrite:

```
Rule for deciding the concealed possessions of the Cloaked Villain: if the particular
possession is the sable cloak, no; otherwise yes.
```

(A rule which says neither "yes" nor "no" will decide yes, but it's best to spell out exactly

what's wanted.)

Parts are treated exactly as if clothes or items being held, and the following will make the face and inscription on a coin invisible unless the player is holding it - the idea being that they are too small to be seen from farther away.

```
The coin is in the Roman Villa. The face and inscription are parts of the coin. Rule for deciding the concealed possessions of the coin: if the coin is carried, no; otherwise yes.
```

There is also an either/or property called "described"/"undescribed", intended to be used only as a last resort, but which has the ability to hide something from room descriptions. This not really hiding: the idea is that "undescribed" should be used only for cases where some other text already reveals the item, or where its presence is implicit. Even then, it should only be used when the item is intended to be taken or moved by the player at some point - if the item isn't intended to move, it's much better to make it "scenery". (There's only one commonly-found example - the player's own body, the "yourself", is undescribed.)

Note that the "undescribed" property is automatically removed from anything carried by, worn by or part of the player, even indirectly; and that nothing on top of an "undescribed" supporter will be visible in a room description, even if it itself is "described". (Scenery supporters don't suffer from that restriction, which is one reason scenery is a better option when possible.)

Example

38. **Search and Seizure**
    A smuggler who has items, some of which are hidden.

RB 7.3 Reactive Characters

§3.25 The location of something

The model world created by Inform is partitioned into rooms. This means that everything which exists in the model world, exists in one of the rooms. If we write a sentence such as

```
Professor Wilderspin is a man.
```

and say nothing more about Wilderspin, then he does not physically exist at the start of the story: he is said to be "out of play", and stays that way until we move him into one of the rooms. A better metaphor might be that he is waiting in the wings, ready to come onto the stage.

Every thing is either out of play, or can be found in one of the rooms, and the property "location of X" gives us the room in question. The following condition tests, in effect, whether Wilderspin is in play:

```
if the location of Wilderspin is a room, ...
```

Which uses a new phrase:

location of (object) ⇒ *room*

This phrase produces the room which, perhaps indirectly, contains the object given. Example: if the player stands in Biblioll College and wears a waistcoat, inside which is a fob watch, then

```
location of the fob watch
```

is Biblioll College. In general, a thing cannot be in two rooms at once, but there are two exceptions: two-sided doors, present on both sides, and backdrops. The "location of" a door is its front side, but a backdrop has no location. (Objects which are not things at all, such as rooms and directions, also have no location.)

We very often want to know the location of the player, and this is more simply called just "the location". (This is actually a value that varies rather than a phrase, but that's a technicality we can ignore here.)

The idea of indirect containment is useful enough to have a name: Inform calls it "enclosure". A thing encloses whatever is a part of itself, or inside itself, or on top of itself, and it also encloses anything that they enclose. And when something moves around, anything it encloses will move with it. In the example above, Biblioll College (a room) and the player (a person) both enclose the fob watch and the waistcoat. (The small print: a door is enclosed by the rooms on both sides; a backdrop is never enclosed.)

Enclosure is only useful when being used as a question. So the following is fine:

```
if the player encloses the fob watch, ...
```

But these will produce problem messages:

```
The player encloses the fob watch. The location of the trilobite is the Museum.
```

because they are too vague. Inform needs to know exactly where the fob watch and the trilobite will begin the story, whereas these sentences leave room for doubt about who or what is actually holding them.

Addendum

The verb "to hold" means the holding relation. X holds Y if one of the following are true (they're mutually exclusive, so it's not possible for more than one of them to be true): X contains Y, X supports Y, X wears Y, X carries Y, X incorporates Y, i.e., Y is part of X). Enclosure is most easily understood as direct or indirect holding. X encloses Y if X holds Y *or* X holds something that holds Y, and so on, through however many levels. (In assertions, "hold" simply means "carries", e.g., "The player holds the spatula.". "to be held by" means the reversed holding relation, but "to be held inside" and "to be held within" both mean the reversed *containment* relation.)

Example

39. **Van Helsing**
    A character who approaches the player, then follows him from room to room.

RB 7.13 Traveling Characters

§3.26 Directions

"Direction" is a kind which is quite unlike most of those seen so far. While it has to do with the physical world, a direction does not exactly belong to it. One cannot find "southeast" sitting on a shelf. "Direction" is not a kind of thing, nor a kind of room: it is a kind in its own right.

Every direction has an "opposite" property, which is always another direction. These occur in matched pairs. The opposite of north is south, just as the opposite of south is north. The opposite of southeast is northwest, the opposite of inside is outside, and so on. When Inform reads a sentence like…

```
Bangkok is south of Nakhon Sawan.
```

…it assumes that the opposite map connection is probably also valid, so that

```
Nakhon Sawan is north of Bangkok.
```

The chapter began with the twelve directions built into Inform:

```
north, northeast, east, southeast, south, southwest, west, northwest, up, down, inside, outside
```

But the built-in set is not always appropriate. Sometimes this is too many; if we wanted to write about a Flatland, for instance, then up and down ought to go. But in practice it is better not to abolish them as directions but instead to forbid travelling in them. (See the Recipe Book for examples.)

But away from our familiar Earth, the usual frame of reference loses its meaning. Terry Pratchett's "Discworld" comedies, set on a rotating disc, use the directions turnwise, widdershins, hubwards and rimwards. On board a Zeppelin airship, which constantly changes its course, the cockpit has no fixed compass bearing from the passenger cabin: it is not very naturally "north". In zero gravity, there is no up or down. Mars does not have a magnetic core, so a compass doesn't work there.

New directions must always be created in opposing pairs, and each **must** be declared with a clear simple sentence of the form "X is a direction." For instance:

```
Turnwise is a direction. The opposite of turnwise is widdershins.
Widdershins is a direction. The opposite of widdershins is turnwise.
Hubwards is a direction. The opposite of hubwards is rimwards.
Rimwards is a direction. The opposite of rimwards is hubwards.
```

It is then possible to write, say, that:

```
Ankh-Morpork is hubwards of Lancre and turnwise from Borogravia.
```

Of course the Map page of the Index for the project normally draws a map based on compass bearings, so it will get a little befuddled by this. But the map drawn in the Index can be given hints to improve its legibility. More on this later, but for now note that

```
Index map with turnwise mapped as east.
```

maps turnwise directions as if they were east, that is, pointing rightwards on the page. (This has no effect on the story file produced; it does not mean turnwise is simply a new name for east; it affects only the look of the Index map, which is only a convenience for the author in any case.)

At one time, directions had to have shortish names (up to three words only), but that's no longer true:

```
Just the tiniest smidge off magnetic north is a direction. The opposite of
just the tiniest smidge off magnetic north is just the tiniest smidge
off magnetic south.
Just the tiniest smidge off magnetic south is a direction. The opposite of
just the tiniest smidge off magnetic south is just the tiniest smidge
off magnetic north.
```

Examples

40. **Prisoner's Dilemma**
    A button that causes a previously non-existent exit to come into being.

    RB 3.2 Map

41. **The World of Charles S. Roberts**
    Replacing the ordinary compass bearings with a set of six directions to impose a hexagonal rather than square grid on the landscape.

    RB 3.2 Map

42. **Fore**
    Understand "fore", "aft", "port", and "starboard", but only when the player is on a vessel.

    RB 8.2 Ships, Trains and Elevators

4. Kinds

§4.1 New kinds
§4.2 Using new kinds
§4.3 Degrees of certainty
§4.4 Plural assertions
§4.5 Kinds of value
§4.6 Properties again
§4.7 New either/or properties
§4.8 New value properties
§4.9 Using new kinds of value in properties
§4.10 Conditions of things
§4.11 Default values of kinds
§4.12 Values that vary
§4.13 Values that never vary
§4.14 Duplicates
§4.15 Assemblies and body parts
§4.16 Names made in assembly
§4.17 Postscript on simulation

§4.1 New kinds

Values are to Inform what nouns are to English sentences. They represent numbers, times of day, pieces of text, places, people, doors, and so on. Because they have such an enormous variety, and because we often want to talk about what some of them have in common, we need a way to sort all of these different ideas out. That's the main aim of Inform's concept of "kind".

Every value has a kind. The kind of 10 is "number"; the kind of 11:30 PM is "time"; the kind of "jinxed wizards pluck ivy from my quilt" is "text"; and so on. The Kinds index panel shows the kinds present in the current Inform project, which will always include a wide range of built-in kinds, and usually also some new ones created in that project.

Some kinds are more general than others. For example, if we write:

```
Growler is an animal in the Savannah.
```

then Growler is an "animal", which is a kind of "thing", which is a kind of "object". When we talk about "the" kind of Growler, we mean "animal", the most specific one, but actually he belongs to all of those kinds.

As we see from this example, kinds have a whole hierarchy. Some are specialised versions of other kinds; some are not. Browsing the Kinds index shows that Inform builds its model world out of "objects". (That's really what objects are: "object" is a kind of value used to make the ingredients of the model world.) The objects fall into four fundamental kinds, "room", "thing", "direction" and "region", and "thing" is more subdivided still.

All the same, Inform starts out with a fairly simple arrangement. Because taxonomy - setting up kinds for everything - is so difficult, and depends so much on what you want it for, Inform creates relatively few kinds in advance: it has "animal" built in, but not "mammal" or "Bengal tiger". When we need more, we must make them ourselves. Let's see how we might invent these new kinds. The easy one is:

```
A Bengal tiger is a kind of animal.
```

Given that, we can then write:

```
Growler is a Bengal tiger in the Savannah.
```

That's easy enough. Adding "mammal" now looks awkward, though, because it seems to belong in between the two. All Bengal tigers are mammals, but not all animals are. But Inform can sort this out:

```
A mammal is a kind of animal. A Bengal tiger is a kind of mammal.
```

If we look at the Kinds index, we should indeed see a hierarchy:

```
object > person > animal > mammal > Bengal tiger
```

though the diagram is laid out as a sort of tree, just as zoologists do.

As another example, it may seem curious that Inform doesn't usually think "man" is a kind of "animal". This is really because, in English, people don't usually expect something like "if an animal is in the garden" to be true when it's only the Revd Mr Beebe going for a walk. People expect the word "animal" not to mean a human being, despite our common genetic ancestry. But if we want to rearrange Inform's default ideas, we can do that quite easily:

```
A man is a kind of animal. A woman is a kind of animal.
```

Or indeed we could say:

```
A human being is a kind of mammal. A man and a woman are kinds of human being.
```

While this is an ideal way to make new kinds for the model world, we are more restricted in what we can do outside of objects. For instance,

```
A glob is a kind of number.
```

isn't allowed. The numbers are fixed and all exist already; they aren't like Bengal tigers which we can simply imagine, and fill the world with. "Number" is not a concept we can specialise any further. But what we can do is to invent entirely new concepts, like so:

```
A distance is a kind of value.
```

We will see more of these later. (This isn't specialising anything - "value" is a sort of everything-category, and is too big and vague to be a kind.)

Example

43. Vouvray
    Adding synonyms to an entire kind of thing.

RB 2.2 Varying What Is Read

§4.2 Using new kinds

This seems a good point to see what we can do with new kinds. Here we invent a new kind to provide a new sort of room:

```
A dead end is a kind of room.
```

Any dead end that we make is also a room, so it has all of the properties and behaviour of a room. For instance, every room is either "lighted" or "dark", and the default is to be lighted. But we can reverse that convention for dead ends, and we can also fill in some other properties:

```
A dead end is a kind of room with printed name "Dead End" and description "This is a dead end. You'll have to go back the way you came." A dead end is usually dark.

The Undertomb is a dark room. East is a dead end. South is a dead end with printed name "Collapsed Dead End". Northwest is a dead end called the Tortuous Alcove.

In the Undertomb is the candle lantern. It is lit.
```

As a result of this, three different rooms adjoin the Undertomb, all dead ends. This is

much more concise than spelling them out one at a time would be.

Inform often doesn't mind in what order it is told about the world, but it may need to know the name of a kind before that kind can be used. For example,

```
A coffer is a kind of container. In the Crypt is an open coffer.
```

makes sense to Inform and results in the creation of a new thing, just called "coffer" in the absence of any other name to give it, whose kind is "coffer" and which is initially open. Whereas if Inform reads:

```
In the Crypt is an open coffer.
```

without knowing that "coffer" is a kind, it simply makes a thing called "open coffer" (and which is not a container). Inform has to be careful like this: English is simply too overflowing with multiple meanings. An "open railway ticket", for instance, is not a "railway ticket" that one can put objects into.

§4.3 Degrees of certainty

When we write:

```
A dead end is usually dark.
```

we are saying that it will be dark rather than lighted unless we should specify otherwise. So it would be fine to add:

```
The Tortuous Alcove is lighted.
```

because although dead ends are usually dark, this one is evidently an exception. On the other hand, if we had originally written

```
A dead end is always dark.
```

then Inform would not have permitted any exception to be made, and would have reported a problem if we had tried to make the Tortuous Alcove lighted. Besides "usually" and "always", we can also employ "seldom" and "never", which are their negatives. Thus, "never lighted" means the same as "always dark".

Examples

44. **Odin**
    Replacing "You see nothing special…" with a different default message for looking at something nondescript.
    RB 6.5 Examining

45. **Something Narsty**
    A staircase always open and never openable.
    RB 3.5 Doors, Staircases, and Bridges

§4.4 Plural assertions

As the following examples show, sentences can make several assertions at once by using the plural. Suppose we have defined a kind called "high-up fixture", for instance like so:

```
A high-up fixture is a kind of thing. A high-up fixture is usually fixed in place.
```

Then the following sentence creates two such objects and puts them in their place:

```
The high shelf and the skylight window are high-up fixtures in the Lumber Room.
```

since it is equivalent to saying:

```
The high shelf is a high-up fixture. The skylight window is a high-up fixture. The high shelf
is in the Lumber Room. The skylight window is in the Lumber Room.
```

Such plurals are allowed in almost any context, and we could even define two kinds at once:

```
Bucket and basket are kinds of container.
```

Inform constructs plurals by a form of Conway's pluralisation algorithm, which is quite good - for example, it gets oxen, geese (but mongooses), sheep, wildebeest, bream, vertebrae, quartos, wharves, phenomena, jackanapes and smallpox correct. But English is a very irregular language, and multiple-word nouns sometimes pluralise in unexpected ways. So we sometimes need to intervene:

```
A brother in law is a kind of man. The plural of brother in law is brothers in law.
```

We are allowed to define more than one plural for the same singular text, and for the

names of things, rooms or kinds, all alternative versions will be used interchangeably. (For instance, Inform defines both "people" and "persons" as plurals of "person".)

Example

46. **Get Me to the Church on Time**
    Using kinds of clothing to prevent the player from wearing several pairs of trousers at the same time.

RB 9.3 Clothing

§4.5 Kinds of value

So much for making new and more specialised kinds of object - for example, new kinds of room, or new kinds of animal. This allows us to describe the physical world in elegant ways, but what about concepts which aren't so physical?

Without getting into philosophy, we can probably agree that numbers like 1, 2, 3, …, and texts like "Jackdaws love my big sphinx of quartz", are not physical. Inside Inform, those are values, but not objects. Inform already has a good stock of this sort of concept built in, so it may not immediately seem clear why we need to create new ones. But in fact this is very useful. To describe the physical world, we need concepts like (for example) distance and brightness. We want to say that two armchairs are 12 feet apart, or that a given light-bulb is very dim. Here, "twelve feet" and "very dim" are clearly not physical objects; they need to be values, but not objects.

As these two examples suggest, sometimes we want a quantitative way to measure things, sometimes not. Thomas Hardy, in his novel "The Return of the Native", writes:

> When he drew nearer he perceived it to be a spring van, ordinary in shape, but singular in colour, this being a lurid red.

Hardy doesn't tell us that the wavelength of the light is 700nm, he tells us that the colour is "lurid red", and we understand. Later in the same chapter, Hardy writes:

> The loads were all laid together, and a pyramid of furze thirty feet in circumference now occupied the crown of the tumulus.

and now we do have a quantitative measurement: thirty feet. This is how people write about the world, and how they read about it. So Inform needs to provide both sorts of measurement.

(a) Here is a qualitative example. Suppose we would like a candle lantern to burn down, gradually diminishing in brightness. Then we'll need a way to talk about the current

strength of the flame, but only in vague terms. Here goes:

```
Brightness is a kind of value. The brightnesses are guttering, weak, radiant and blazing.
```

"Brightness" is now a kind of value on a par with (for instance) "number" or "text". There are only four possible values, named as above. Kinds of value like this, where there are just a few named possibilities, are extremely useful, as we'll see.

(b) Now a quantitative example:

```
Weight is a kind of value. 1kg specifies a weight.
```

The difference here is not the way we create the kind, but the way we tell Inform what the possible values are. Instead of a list, we teach Inform some notation. As a result, "26kg" is now a value, for instance. Quantitative kinds like this are sometimes called "units", because - as in this example - they're often units in the sense of measuring things. Many Inform projects never need units, but they can still be very useful, and they're described in detail in the chapter on "Numbers and Equations".

§4.6 Properties again

So now we have seen two fundamental ideas: "value" and "kind". We have seen how to make a source text which refers to many values - for example, Growler the Bengal tiger, 23kg, "Collapsed Dead End", blazing, 7, all values of different kinds.

But we don't just want a way to refer to values, we want to lay out facts about them. Inform understands two sorts of fact, which it calls properties and relations. Properties are about single values in isolation: Growler is male. Relations are about how values interact with each other: Growler likes Bambi. (Or would like to eat Bambi, anyway.) Relations are really the central organising idea of Inform, and we've seen them many times already:

```
Growler is in the Savannah.
```

expresses a relation called "containment" between Growler and the Savannah. Much more about this in the chapter on Relations: for now, let's go back to the simpler idea of properties.

In Inform terms, a "property" is any fact about a value (other than its kind) which the author is allowed to choose. For example,

```
Growler is an animal. Growler is male. The description of Growler is "What immortal hand or eye could frame thy fearful symmetry?".
```

The first of these sentences talks about Growler's kind, but the other two sentences tell Inform about his properties. Properties are divided into either/or properties - "male" versus "female" - and value properties - such as the description of something, which can be any text.

The Kinds index shows which kinds of object are allowed to have properties. Every object is, so there's no problem with Growler. In general, if Inform can find a sensible way to store properties, then it will allow them. But it won't allow (for example) properties of numbers. There are only a finite number of Bengal tigers in the world (fewer than three thousand, alas), so Inform can easily store individual description texts for each one of them. But there are an infinite range of numbers. (Inform does allow adjectives like "even" or "odd" to be used about number - saying "if 7 is odd" is fine, for example - but they are not properties in the Inform sense, because the author doesn't get to choose. The author can choose whether Growler is a male or female tiger, but not whether 7 is an even or an odd number.)

§4.7 New either/or properties

Properties can't be handed out completely freely. In the previous chapter, we saw that we were allowed to make a chair "portable" and to make a room "dark". But if we try this the other way round, Inform produces a Problem message. This is because every property must be created in a way which lays out what values are allowed to have it. The Standard Rules built into Inform say that

```
A thing can be fixed in place or portable.
```

and as a result it won't allow "The Savannah is portable" because the Savannah is a room, not a thing.

We must do the same. To go back to our example "dead end" kind:

```
A dead end is either secret or ordinary.
```

This creates just one new property, not two. The names are taken as the two states of a single either/or property: secret means not ordinary, ordinary means not secret. Alternatively, we could just say:

```
A dead end can be secret.
```

in which case the opposite of "secret" would be "not secret".

Now we have a property which can be given to any value of the kind "dead end". We're also free to add to the definitions of kinds which already exist, including those built into Inform: for instance,

```
A room is either indoors or outdoors.
```

If we make the above definitions then all dead ends will be "ordinary" and all rooms "outdoors" unless the source text says otherwise. That is, in the absence of other information it's assumed that an either/or property is not true. We could reverse by writing:

```
A dead end is usually secret. A room is usually indoors.
```

A property can be used by several kinds at once. For example, the built-in either/or property "open" is used by both doors and containers, even though door isn't a kind of container and container isn't a kind of door. In fact, although it's more usual to declare properties for whole kinds, they can actually be given to single values one at a time, if we like:

```
The umbrella is carried by the player. The umbrella can be open.
```

And now the umbrella, which is a thing and not a door or container, can also have the property.

Example

47. **Change of Basis**
    Implementing sleeping and wakeful states.

RB 6.11 Waiting, Sleeping

§4.8 New value properties

So much for either/or properties. Now we move on to properties which have values attached. The same principles apply, but the wording is different. For example,

```
A dead end has some text called the river sound. The river sound of a dead end is usually
"a faint whispering of running water". The Tortuous Alcove has river sound "a gurgle of
running water".
```

The property "river sound" is now applicable only to dead ends, so we would not be

allowed to talk about "the river sound of the Savannah", say. Moreover, it's required to hold a piece of text. If we tried the following:

```
The river sound of the Tortuous Alcove is 7.
```

…then Inform would object, because the number 7 is the wrong kind of value to go into the "river sound" property. If we need a numerical property, we can try this instead:

```
A dead end has a number called the difficulty rating. The Tortuous Alcove has difficulty rating 7.
```

Suppose that we were to add:

```
The Exquisitely Narrow Defile is a dead end.
```

The Defile must have a river sound, of course, because we said that every dead end would have one. We haven't said what that river sound will be, but Inform can work it out, because we did say this:

```
The river sound of a dead end is usually "a faint whispering of running water".
```

If there are no instructions at all about the value of a property, Inform fills in the default value of the appropriate kind - in this case, it would be a blank text. (A table of the kinds which can be used for properties, and their default values, can be found in the Kinds index.)

Examples

48. **Would you...?**
    Adding new properties to objects, and checking for their presence.
    RB 9.1 Food

49. **Straw Boater**
    Using text properties that apply only to some things and are not defined for others.
    RB 8.1 Bicycles, Cars and Boats

§4.9 Using new kinds of value in properties

It turns out to be very useful to create a new kind of value, and then create a property to hold it. So useful, in fact, that Inform provides two shortened forms for doing so. Here is the first, and the second (making "conditions") is in the section following.

Suppose we go back to our example of the candle lantern whose brightness we have to measure. It's clear that what we want to do is to define:

```
Brightness is a kind of value. The brightnesses are guttering, weak, radiant and blazing.
```

And now we can use the technique of the previous section:

```
The lantern has a brightness called the flame strength. The flame strength of the lantern
is blazing.
```

This works very nicely. The "flame strength" property is now only allowed to have one of four values we allowed: guttering light, weak light, radiant light or blazing light. So we have succeeded in recording our measurement.

But it seems artificial to call the brightness of the lantern "flame strength", when we could instead simply call it "brightness". Much simpler to write:

```
The lantern has a brightness. The lantern is blazing.
```

Now "brightness" is the name of both the property and the kind of value. What's particularly nice is that we can now use the names of the possible brightnesses - "weak", "blazing" and so on - as adjectives. Inform knows that "The lantern is blazing" must be talking about the brightness property, because "blazing" is a brightness.

Now we can improve our dead ends:

```
A dead end is a kind of room with printed name "Dead End" and description "This is a
dead end, where crags in the uneven rock are caught by the [brightness of the lantern]
flame you hold aloft. Despite [river sound] there is no sign of the stream." A dead end is
usually dark.
```

The "[brightness of the lantern]" is printed not as literal text, but as whatever the brightness currently is. (The square brackets mark it as what is called a text substitution, which will be the subject of the next chapter.) So we get something like this:

```
This is a dead end, where crags in the uneven rock are caught by the blazing flame you
hold aloft. Despite a faint whispering of running water there is no sign of the stream.
```

So now we have a lantern, which has a brightness as a property. But we can build on this further if we like. A brightness such as "guttering" is a value, so it can have properties in its own right. That can be quite useful, in fact:

```
A brightness can be adequate or inadequate. A brightness is usually adequate. Guttering is inadequate.
```

This is convenient because it divides up the brightnesses:

```
The player carries a book. The description of the book is "[if the brightness of the lantern is adequate]Many secrets are now yours.[otherwise]No, the print's too tiny by this awful light."
```

And while we're at it, let's give each brightness its own corresponding temperature:

```
Temperature is a kind of value. 100C specifies a temperature.

A brightness has a temperature. The temperature of a brightness is usually 700C. The temperature of blazing is 1400C. The temperature of radiant is 1100C.

The description of the lantern is "The lantern shines with a flame at [temperature of the brightness of the lantern]."
```

(Candle flames are hotter than most people think.)

See Also

Text with substitutions for more on varying what is printed.

Examples

50. **The Undertomb 1**
    A small map of dead ends, in which the sound of an underground river has different strengths in different caves.

RB 3.8 Sounds

51. **The Undertomb 2**
    Flickering lantern-light effects added to the Undertomb.

RB 3.7 Lighting

52. **The Crane's Leg 1**
    A description text that automatically highlights the ways in which the object differs from a standard member of its kind.

RB 5.6 Viewpoint

53. **Signs and Portents**
    Signpost that points to various destinations, depending on how the player has turned it.

RB 9.8 Simple Machines

§4.10 Conditions of things

Now for an even more abbreviated way to create a new kind of value, and at the same time create a property to hold it. Suppose we have something, say a wine cask, which we know is always in one of three different states. We can write:

```
The cask is either customs sealed, liable to tax or stolen goods.
```

This is just like our example of the lantern having possible brightnesses, but it's quicker to do, because we don't need to create or name the kind of value. (The trade-off is that we can't use it for anything else as well.)

Initially the cask will be "customs sealed", the first value we gave. We could now write, for instance,

```
The description of the cask is "A well-caulked Spanish wine cask.
[if liable to tax] It really is a shame to have to pay duty on it!"
```

Or, as a second example, here we're going to allow a whole kind to have the property, not just a single object:

```
Colour is a kind of value. The colours are red, green and white.
A colour can be bright, neutral or flat. Green is neutral.
```

Now in fact these properties are not anonymous: Inform has worked out names for them, even though we didn't give any. The usual arrangement is that the name is the name of the object with the word "condition" tacked on: for instance, "cask condition". So we could write:

```
The printed name of the cask is "wine cask ([cask condition])".
```

so that sometimes this would be "wine cask (liable to tax)", sometimes "wine cask (stolen goods)" and so on.

But only usually, because we might need to define several different conditions of the same thing, and then the names would collide. For instance, suppose we write:

```
A fruit is a kind of thing. A fruit can be citrus, berry, melon, or pome.
```

This makes a property and a kind of value each called "fruit condition". But now suppose we add that:

```
A fruit can be unripened, ripe, overripe, or mushy.
```

This is a quite unrelated property - a fruit could have any combination of these two properties, in fact. Left to itself, Inform will call the second one "fruit condition 2", which isn't really ideal if we ever do need to refer to it in other source text. So we are also allowed to give these conditions names of our own choosing:

```
A fruit can be unripened, ripe, overripe, or mushy (this is its squishiness property).
```

And now the resulting property and kind of value would be called "squishiness".

§4.11 Default values of kinds

Just about every kind has a "default value". Inform needs this when it knows that something has to be a value of a given kind, but it hasn't been told what the value is. For example, in the previous chapter we saw that every thing has a "description" text, but we also created plenty of things without describing them. So if Inform reads

```
The conference pear is in the bowl.
```

and it isn't told anything else about the pear, what should it set the description of the pear to?

The answer is that Inform knows the description has to be a value of the kind "text", so it uses the default value of "text". Not very interestingly, this is just the blank text "".

Being uninteresting is exactly the idea, of course. The default number is 0, for instance. (Default values are tabulated in the Kinds index.)

It's sometimes useful to be able to refer to the default value of a kind without having to spell out what this is (especially if the kind is something obscure, or we're trying to write a rule for an extension which has to work in situations we don't fully know about).

default value of (name of kind) ⇒ *value*

Produces the default value of the kind named. Examples:

```
The silver repeater is here. "You catch sight of a silver repeater watch, hands immobile at [default value of time]."
```

produces the output:

```
You catch sight of a silver repeater watch, hands immobile at 9:00 am.
```

because nine in the morning is the default time in Inform. If we have:

```
Brightness is a kind of value. The brightnesses are guttering, weak, radiant and blazing.
```

then "default value of brightness" is guttering, the first brightness created. When it comes to kinds of object, we sometimes have to be a little careful. For example,

```
default value of room
```

is always going to be fine (it's always the first room created in the source text). But

```
default value of vehicle
```

would produce a Problem message if there were no vehicles in the world.

§4.12 Values that vary

Sometimes a value important to the simulated world will not naturally belong to any thing or room, and should not be kept in a property. In fact, we have seen a value that varies already: "location", which holds the room in which the story is presently taking place. Here's how we might make a new one:

```
The prevailing wind is a direction that varies. The prevailing wind is southwest.
```

Or "which varies" would also be allowed, as would the more traditional computing term "variable":

```
The prevailing wind is a direction variable. The prevailing wind is southwest.
```

A briefer way to do this is to use the word "initially", which alerts Inform to the possibility that the value will change in future:

```
The prevailing wind is initially southwest.
```

This creates the variable and gives it an initial value all in one sentence.

It's not compulsory to give an initial value. If we do not, Inform will use the default value for its kind. (See the table in the Kinds index.) For example, writing just

```
The grand tally is a number that varies.
```

will start it at the value 0, because that's the default value for numbers.

We can have variables of any of the kinds of value, including new ones, but should watch out for a potential error. If we write:

```
The receptacle is a container that varies.
```

in a world which has no containers at all, Inform will object, because it will be unable to put any initial value into the receptacle variable. A similar complaint will be made if we write:

```
Colour is a kind of value. The fashionable shade is a colour that varies.
```

without ever having defined any colours. Something else we are not permitted is:

```
The receptacle is an open container that varies.
```

because the openness of a given container may change during play, so that the value in the variable might suddenly become invalid even though the variable itself had not changed.

As a final note on kinds, when Inform reads something like this:

```
Peter is a man. The accursed one is initially Peter.
```

it has to make a decision about the kind of "accursed one". Peter is a "man", so that seems like the right answer, but Inform wants to play safe in case the variable later needs to change to a woman called Jane, say, or even a black hat. So Inform in fact creates "accursed one" as an object that varies, not a man that varies, to give us the maximum freedom to use it. If we don't want that then we can override it:

```
Peter is a man. The accursed one is initially Peter.
The accursed one is a man that varies.
```

thus telling Inform exactly what is intended.

Example

54. **Real Adventurers Need No Help**
Allowing the player to turn off all access to hints for the duration of a game, in order to avoid the temptation to rely on them overmuch.

RB 11.3 Helping and Hinting

§4.13 Values that never vary

It's sometimes useful to name even values which don't change. For example, suppose the story involves driving, and the same speed limit value comes up in many places. Rather than typing "55" (say) every time it comes up, we might prefer to write:

```
The speed limit is always 55.
```

at the start of the source text, and then talk about "the speed limit" every time we would otherwise have typed "55". Just as the word "initially" alerts Inform that we want the named value to change during play, the word "always" tells it that we don't.

This might seem pointless, because "speed limit" only means the same thing as "55" and takes more typing. But there are two reasons why authors might want to use this feature

anyway. One is that it's easier for a human reader to understand the significance of a line like:

```
if the speed is greater than the speed limit, ...
```

Another is that it makes it easier to change our minds about the value, because if we decide we want 70 as the limit and not 55, we only need to make one change at the start of the source text:

```
The speed limit is always 70.
```

which is much easier than combing through a long source text trying to find many individual things which need changing.

"Speed limit" is then a number constant. Any attempt to set this elsewhere, or change its value, will result in a Problem message, and moreover it can be used in contexts where only constant values are allowed. For example,

```
The generic male appearance is always "He is a dude."

Trevor is a man. The description of Trevor is the generic male appearance.
```

means that the SHOWME TREVOR testing command produces, among other data:

```
description: "He is a dude."
```

§4.14 Duplicates

Although it is only useful to a limited extent, we can make any number of copies of something:

```
"Polygons"

A shape is a kind of thing. A square is a kind of shape. A triangle is a kind of shape.

The Geometry Lab is a room. In the Geometry Lab are three triangles and two squares.
```

The description "three triangles" makes three identical things, each of the kind "triangle", and similarly for the squares. When the above is compiled, the player can type TAKE TWO TRIANGLES or TAKE ALL THE TRIANGLES and so forth.

Four caveats. Firstly, a counted-out description like "two squares" is only allowed if it combines a number with the name of a kind which is already known (perhaps modified

with adjectives, so "two open doors" is fine). If we say:

```
Two circles are in the Lab.
```

without having defined "circle" as a kind in advance, then only a single object will be created - whose name is "two circles". (This is because many natural names start with numbers: "six of clubs", for instance, referring to a single playing card, or "12 Hollywood Close" meaning a single house. We wouldn't want such names to be misinterpreted.)

The second caveat is that excessive duplication is expensive in memory and running time. It is perfectly legal to say

```
In the Lab are 75 triangles.
```

but the resulting story may be a little sluggish: and Inform draws the line at 100, refusing to create more duplicates than that in any single place. If we really need more than about fifty duplicated objects - say, a tombola containing raffle tickets numbered 1 to 1000 - it is usually better to find some less literal way to simulate this: for instance, only having a single raffle ticket, but with a randomly chosen number on it.

If there are very many items in the same place, commands like TAKE ALL and DROP ALL may mysteriously not quite deal with all of them - this is because the parser, the run-time program which deciphers typed commands, has only limited memory to hold the possibilities. It can be raised with a use option like so:

```
Use maximum things understood at once of at least 200.
```

(The default is, as above, 100. Note the "at least".)

Thirdly, note that Inform's idea of "identical" is based on what the player could type in a command to distinguish things. In a few cases this can make items unexpectedly identical. For example:

```
The Lab is a room. A chemical is a kind of thing. Some polyethylene and polyethylene-terephthalate are chemicals in the Lab.
```

results surprisingly in "You can see two chemicals here", because the run-time system truncates the words that are typed - POLYETHYLENE and POLYETHYLENE-TEREPHTHALATE look like the same word in a typed command. So Inform decides that these are indistinguishable chemicals. Typically words are truncated after 9 letters, though (unless the Glulx setting is used) punctuation inside a word, such as an

apostrophe, can make this happen earlier. The best way to avoid trouble is simply to use more easily distinguishable names. For example:

```
Some polyethylene and polyethylene terephthalate are chemicals in the Lab.
```

works fine, because now only one chemical can be called TEREPHTHALATE, and that means they can be distinguished.

Finally: numbers up to twelve may be written out in words in the source text, but larger ones must be written as numerals. So "twelve" or "12", but "13" only.

Example

55. **Early Childhood**
    A child's set of building blocks, which come in three different colours - red, green and blue - but which can be repainted during play.

RB 9.7 Painting and Labeling Devices

§4.15 Assemblies and body parts

In the previous chapter, we saw that it was possible to make sub-parts of things. For instance,

```
The white door is in the Drawing Room. The handle is part of the white door.
```

creates a door with an attached handle. But what if we want to say that not just this door, but every door, should have a handle? To do this we first need to create a kind called "handle", since there will clearly need to be many handles. The solution is:

```
A handle is a kind of thing. A handle is part of every door.
```

"Every" is a loaded word and best used sparingly. A sentence like "A handle is part of every handle" would, if taken literally, mean that a handle takes forever to make and is never finished. Inform will reject this, but the moral is clear: we should think about what we are doing with "every".

We will usually want to work with smaller collections - not literally every room, but with a whole set of them all the same. We can do that like so:

```
A silver coin is a kind of thing. A banking room is a kind of room. Five silver coins are in every banking room.
```

The effect of sentences like these is to make what we might call "assemblies" instead of single things. When a banking room is created, so are five more silver coins; when a door is created, so is another handle. Such sentences act not only on items created later on in the source text, but also on all those created so far.

This is especially useful for body parts. If we would like to explore Voltaire's suggestion that history would have been very different if only Cleopatra's nose had been shorter, we will need noses:

```
A nose is a kind of thing. A nose is part of every person.
```

Of course, if we make an assembly like this then we had better remember that the player is also a person and also gets a nose. In fact slightly odd things can happen if we combine this with changing the identity of the player. This works:

```
Cleopatra is a woman in Alexandria. The player is Cleopatra.
A nose is a kind of thing. A nose is part of every person.
```

but if those lines are in reverse order then Cleopatra's nose is assembled before she becomes the player, with the result that it ends up called "Cleopatra's nose" rather than "your nose" in play - which is very regal but probably not what we want. To avoid this, settle the player's identity early on in the source text.

All of the assemblies above make objects. Most make these new objects "part of" existing ones, but as we saw, they can also be "in" or "on" them. In fact, though, assemblies work in much more general ways: they can assemble values of almost any kind, placed in almost any relationship. To make use of that, we need to create a new verb, a topic which won't be covered properly until a later chapter, but here goes:

```
A colour is a kind of value. The colours are red, green and blue.

Liking relates various people to various colours. The verb to like means the liking relation.

Every person likes a colour.
```

Now every time a person is created, so is a colour which that person will like. If there are two people in the world, the player and Daphne, then we now have five colours: red, green, blue, Daphne's colour and the player's colour. Alternatively, we can assemble the other way around:

```
A person likes every colour.
```

Now we're telling Inform that every time a colour is made, a new person is also made - someone who will like that colour. So this sentence effectively makes three new people, one who likes red, one who likes green, and one who likes blue.

Examples

56. **Being Prepared**
A kind for jackets, which always includes a container called a pocket.
RB 9.3 Clothing

57. **Model Shop**
An "on/off button" which controls whatever device it is part of.
RB 9.8 Simple Machines

58. **U-Stor-It**
A "chest" kind which consists of a container which has a lid as a supporter.
RB 8.4 Furniture

59. **The Night Before**
Instructing Inform to prefer different interpretations of EXAMINE NOSE, depending on whether the player is alone, in company, or with Rudolph the Red-nosed Reindeer.
RB 5.1 The Human Body

§4.16 Names made in assembly

Something skated over in the previous section is the question of how Inform gives names to objects (or other values) it creates in an assembly. The standard thing naming combines the names of what's being assembled. For example:

```
A nose is a kind of thing. A nose is part of every person. Antony and Cleopatra are people.
```

might result in the creation of "Antony's nose", part of Antony, and "Cleopatra's nose", part of Cleopatra. In this way, Inform names the noses after their owners. It will always do this unless there are multiple indistinguishable things being created, as in the "five silver coins are in every banking room" example: those will all just be called "silver coin".

A small pitfall of this is that if we write:

```
Marcus Tullius Cicero is a person.
```

then although "Marcus Tullius Cicero's nose" and "Cicero's nose" are both valid names

for the consular nose, "Marcus's nose" is not.

The standard naming scheme is often about right, but as usual Inform offers a way to improve it in particular cases. For example, if we write:

```
Every room contains a vehicle (called its buggy).
```

then we will find the world full of, say, the Garden buggy, the Patio buggy and so on - instead of the Garden vehicle, the Patio vehicle and so on, which is what we would have had without the "called…" part. Similarly, we could write:

```
A person (called its fan) likes every colour.
Every person likes a colour (called his favourite colour).
```

The former would produce new people with names like "Green's fan", whereas the latter would produce new colours with names like "Daphne's favourite colour".

So much for an informal description. Here is exactly what Inform does:

> (1a) If there is a "called..." text, Inform uses it, expanding out "its" (or "his" or "her" or "their") to a possessive form of the name of the owner, so to speak, and "it" (or "he" or "she" or "they" or "him" or "them") to the name itself.
>
> (1b) If there's no "called..." text, Inform behaves as if we had written "(called its K)", where K is the name of the kind.
>
> (2) If this results in a value which isn't an object being given a name which already exists, Inform tacks on a number to force the new name to be different from existing ones: e.g., "Daphne's colour 2", "Daphne's colour 3", ...

(The reason that (2) doesn't affect objects is that objects are allowed to have names clashing with other objects, or no name at all, whereas other values have to have names belonging to themselves alone.)

§4.17 Postscript on simulation

That concludes our tour through the design of the initial state of a simulated world. We have seen how to create rooms and to stock them with containers, supporters, devices, doors, men and women. The player of such a simulation can explore, move things around, open and close certain doors and containers, lock or unlock them provided a suitable key is found, switch machines on or off, and so on.

But that is about all. There is as yet no element of surprise, no aim or sense of progress to be earned, and no narrative thread. We have painted the backcloth, and laid out the

properties, but the actors have yet to take the stage.

5. Text

§5.1 Text with substitutions
§5.2 How Inform reads quoted text
§5.3 Text which names things
§5.4 Text with numbers
§5.5 Text with lists
§5.6 Text with variations
§5.7 Text with random alternatives
§5.8 Line breaks and paragraph breaks
§5.9 Text with type styles
§5.10 Accented letters
§5.11 Unicode characters
§5.12 Displaying quotations
§5.13 Making new substitutions

§5.1 Text with substitutions

In the previous chapter, we gave properties to certain kinds of things in order to change their appearance and behaviour, and saw brief glimpses of one of Inform's most useful devices: text substitution. The following gives a more complete example:

```
"The Undertomb"

A dead end is a kind of room with printed name "Dead End" and description "This is a dead end. You'll have to go back the way you came, consoled only by [river sound]." A dead end is usually dark.

The Undertomb is a dark room. East is a dead end. South is a dead end with printed name "Collapsed Dead End". Northwest is a dead end called the Tortuous Alcove. In the Undertomb is the lantern. It is lit.

A dead end has some text called river sound. The river sound of a dead end is usually "a faint whispering of running water". The Tortuous Alcove has river sound "a gurgle of running water".
```

The novelty here is the text in square brackets in the first paragraph. They imply more or less what they would when a journalist is quoting something in a newspaper article. The actual words "river sound" are not part of the text. Instead, when Inform prints up the description of a dead end, it will substitute the appropriate river sound in place of these words.

Thus the description of the Collapsed Dead End is "This is a dead end. You'll have to go

back the way you came, consoled only by a faint whispering of running water.", whereas the description of the Tortuous Alcove is "This is a dead end. You'll have to go back the way you came, consoled only by a gurgle of running water." As the player explores these dead ends, subtle differences will appear in their room descriptions.

§5.2 How Inform reads quoted text

Text is so fundamental to Inform that the basics had to be covered back in Chapter 2, so let's begin this new chapter with a recap.

Literal text is written in double-quotation marks. It's mostly true that what you see is what you get: the literal text "The Hands of the Silversmith" means just

```
The Hands of the Silversmith
```

But four characters are read in unexpected ways: [,], ' and ". The rules are as follows:

Exception 1. Square brackets [and] are used to describe what Inform should say, but in a non-literal way. For example,

```
"Your watch reads [time of day]."
```

might produce

```
Your watch reads 9:02 AM.
```

These are called "text substitutions". They're highly flexible, and they can take many different forms. But as useful as they are, they do seem to stop us from making actual [and] characters come through on screen. To get around that:

```
say "[bracket]"
```

This text substitution expands to a single open square bracket, avoiding the problem that a literal [in text would look to Inform like the opening of a substitution. Example:

```
"He [bracket]Lord Astor[close bracket] would, wouldn't he?"
```

prints as "He [Lord Astor] would, wouldn't he?".

```
say "[close bracket]"
```

This text substitution expands to a single close square bracket, avoiding the problem that a literal] in text would look to Inform like the closing of a substitution. Example:

```
"He [bracket]Lord Astor[close bracket] would, wouldn't he?"
```

prints as "He [Lord Astor] would, wouldn't he?".

Exception 2. Single quotation marks at the edges of words are printed as double. So:

```
"Simon says, 'It's far too heavy to lift.'"
```

produces

```
Simon says, "It's far too heavy to lift."
```

This is good because typing a double quotation mark inside the quote wouldn't work - it would end the text then and there. Single quotation marks inside words, such as the one in "it's", remain apostrophes.

The rule looks odd at first, but turns out to be very practical. The only problem arises if we need an apostrophe at the start or end of a word, or a double inside one. Again, substitutions can fix this:

say "[apostrophe/']"

This text substitution expands to a single quotation mark, avoiding Inform's ordinary rule of converting literal single quotation marks to double at the edges of words. Example:

```
Instead of going outside, say "Lucy snaps, 'What's the matter? You don't trust my cookin[apostrophe] mister?'"
```

produces:

```
Lucy snaps, "What's the matter? You don't trust my cookin' mister?"
```

A more abbreviated form would be:

```
Instead of going outside, say "Lucy snaps, 'What's the matter? You don't trust my cookin['] mister?'"
```

which has exactly the same meaning.

say "[quotation mark]"

This text substitution expands to a double quotation mark. Most of the time this is unnecessary because of Inform's rule of converting literal single quotation marks to double at the edges of words, so it's needed only if we want a double-quote in the middle of a word for some reason. Example:

```
"The compass reads 41o21'23[quotation mark]E."
```

which produces: The compass reads 41o21'23"E. (Note that ["] is not allowed; a double-quotation mark is never allowed inside double-quoted text, not even in a text substitution.)

Exception 3. Texts which end with sentence-ending punctuation - full stop, question mark, exclamation mark - are printed with a line break after them. So:

```
say "i don't know how this ends";
say "I know just how this ends!";
```

would come out quite differently - this doesn't affect the appearance of the text, but only the position where the next text will appear. Again, sometimes this is not what we want - the full rules are complicated enough to be worth a whole section later in the chapter.

§5.3 Text which names things

We can put almost any description of a value in square brackets in text, and Inform will work out what kind of value it is and print something accordingly. (Only almost any, because we aren't allowed to use commas or more quotation marks inside a square-bracketed substitution.)

say "[(sayable value)]**"**

This text substitution takes the value and produces a textual representation of it. Most kinds of value, and really all of the useful ones, are "sayable" - numbers, times, objects, rules, scenes, and so on. Example:

```
The description of the wrist watch is "The dial reads [time of day]."
```

Here "time of day" is a value - it's a time that varies, and time is a sayable kind of value, so we might get "The dial reads 11:03 AM."

The values we say most often are objects. If we simply put the name of what we want into square brackets, this will be substituted by the full printed name. We might find:

```
"You admire [lantern]."
= "You admire candle lantern."
```

But this reads oddly - clearly "the" or "a" is missing. So the following substitutions are used very often:

say "[a (object)]"

or…

say "[an (object)]"

This text substitution produces the name of the object along with its indefinite article. Example:

```
Instead of examining something (called the whatever):
    "You can only just make out [a whatever]."
```

which might produce "You can only just make out a lamp-post.", or "You can only just make out Trevor.", or "You can only just make out some soldiers." The "a" or "an" in the wording is replaced by whatever indefinite article applies, if any.

say "[A (object)]"

or…

say "[An (object)]"

This text substitution produces the name of the object along with its indefinite article, capitalised. Example:

```
Instead of examining something (called the whatever):
    "[A whatever] can be made out in the mist."
```

which might produce "A lamp-post can be made out in the mist.", or "Trevor can be made out in the mist.", or "Some soldiers can be made out in the mist." The "A" or "An" in the wording is replaced by whatever indefinite article applies, if any.

say "[the (object)]"

This text substitution produces the name of the object along with its definite article. Example:

```
Instead of examining something (called the whatever):
    "You can only just make out [the whatever]."
```

which might produce "You can only just make out the lamp-post.", or "You can only just make out Trevor.", or "You can only just make out the soldiers." The "the" in the wording is replaced by whatever definite article applies, if any.

say "[The (object)]"

This text substitution produces the name of the object along with its definite article, capitalised. Example:

```
Instead of examining something (called the whatever):
    "[The whatever] may be a trick of the mist."
```

which might produce "The lamp-post may be a trick of the mist.", or "Trevor may be a trick of the mist.", or "The soldiers may be a trick of the mist." The "The" in the wording is replaced by whatever definite article applies, if any.

This may not look very useful, because why not simply put "the", or whatever, into the ordinary text? The answer is that there are times when we do not know in advance which object will be involved. For instance, as we shall later see, there is a special value called "the noun" which is the thing to which the player's current command is applied (thus, if the player typed TAKE BALL, it will be the ball). So:

```
After taking something in the Classroom:
    "You find [a noun]."
```

might produce replies like "You find a solid rubber ball.", "You find an ink-stained blouse.", "You find some elastic bands.", or even "You find Mr Polycarp." (the school's pet hamster, perhaps).

§5.4 Text with numbers

When a numerical value is given in a square-bracketed substitution, it is ordinarily printed out in digits. Thus:

```
"You've been wandering around for [turn count] turns now."
```

might print as "You've been wandering around for 213 turns now.", if the story has been played out for exactly that many commands. But if we prefer:

say "[(number) in words]"

This text substitution writes out the number in English text. Example:

```
"You've been wandering around for [turn count in words] turns now."
```

might produce "You've been wandering around for two hundred and thirteen turns now." The "and" here is natural on one side of the Atlantic but not the other - so with the "Use American dialect." option in place, it disappears.

Either way, though, there is some risk of the following:

```
You've been wandering around for one turns now.
```

We can avoid this using the special substitution:

say "[s]"

This text substitution prints a letter "s" unless the last number printed was 1. Example:

```
"You've been wandering around for [turn count in words] turn[s] now."
```

produces "… for one turn now." or "… for two turns now." as appropriate. Note that it reacts only to numbers, not to other arithmetic values like times (or, for instance, weights from the "Metric Units" extension).

This only solves one case, but it's memorable, and the case is one which turns up often.

Example

60. **Ballpark**
    A new "to say" definition which allows the author to say "[a number in round numbers]" and get verbal descriptions like "a couple of" or "a few" as a result.
    RB 2.1 Varying What Is Written

§5.5 Text with lists

We often want running text to include lists of items.

say "[list of (description of objects)**]"**

This text substitution produces a list, in sentence form, of everything matching the description. Example:

> "Mr Darcy glares proudly at you. He is wearing [list of things worn by Darcy] and carrying [list of things carried by Darcy]."

And, if this were from a dramatisation of the novel by Miss Fielding rather than Miss Austen, we might find:

> Mr Darcy glares proudly at you. He is wearing a pair of Newcastle United boxer shorts and carrying a self-help book.

If the description matches nothing - for instance, if Darcy has empty hands - then "nothing" is printed.

As with all lists in Inform, the serial comma is only used if the "Use serial comma." option is in force. So by default we would get "a fishing pole, a hook and a sinker", rather than "a fishing pole, a hook, and a sinker".

We then need variations to add indefinite or definite articles, and to capitalise the first item. For example,

> "Mr Darcy impatiently bundles [the list of things carried by Darcy] into your hands and stomps out of the room."

might result in

```
Mr Darcy impatiently bundles the self-help book and the Christmas card into your hands and stomps out of the room.
```

say "[a list of (description of objects)]"

This text substitution produces a list, in sentence form, of everything matching the description. Each item is prefaced by its indefinite article. Example:

```
a maritime bill of lading, some hemp rope and Falconer's Naval Dictionary
```

say "[A list of (description of objects)]"

This text substitution produces a list, in sentence form, of everything matching the description. Each item is prefaced by its indefinite article, and the first is capitalised, so that it can be used at the beginning of a sentence. Example:

```
A maritime bill of lading, some hemp rope and Falconer's Naval Dictionary
```

say "[the list of (description of objects)]"

This text substitution produces a list, in sentence form, of everything matching the description. Each item is prefaced by its definite article. Example:

```
the maritime bill of lading, the hemp rope and Falconer's Naval Dictionary
```

say "[The list of (description of objects)]"

This text substitution produces a list, in sentence form, of everything matching the description. Each item is prefaced by its definite article, and the first is capitalised, so that it can be used at the beginning of a sentence. Example:

```
The maritime bill of lading, the hemp rope and Falconer's Naval Dictionary
```

So much for articles. A more insidious problem comes with something like this:

```
"The places you can go are [list of rooms]."
```

The trouble is that the list may end up either singular or plural. We might be expecting something like:

```
The places you can go are Old Bailey, Bridget's Flat and TV Centre.
```

But if there is only one room, then the result might be:

```
The places you can go are Bridget's Flat.
```

which is wrong. We can get around this with careful wording and a slightly different substitution:

```
"Nearby [is-are list of rooms]."
```

say "[is-are list of (description of objects)**]"**

This text substitution produces a list, in sentence form, of everything matching the description. The whole list starts with "is" (if there's one item or none) or "are" (more than one). Examples:

```
is marlin-spike
are maritime bill of lading, hemp rope and Falconer's Naval Dictionary
```

say "[is-are a list of (description of objects)**]"**

This text substitution produces a list, in sentence form, of everything matching the description. Each item is prefaced by its indefinite article, and the whole list starts with "is" (if there's one item or none) or "are" (more than one). Examples:

```
is a marlin-spike
are a maritime bill of lading, some hemp rope and Falconer's Naval Dictionary
```

say "[is-are the list of (description of objects)**]"**

This text substitution produces a list, in sentence form, of everything matching the description. Each item is prefaced by its definite article, and the whole list starts with "is" (if there's one item or none) or "are" (more than one). Examples:

```
is the marlin-spike
are the maritime bill of lading, the hemp rope and Falconer's Naval Dictionary
```

say "[a list of (description of objects) **including contents]"**

This text substitution produces a list, in sentence form, of everything matching the description, noting any contents in brackets. This is really intended only to be used by the Standard Rules.

Examples

61. **Control Center**
    Objects which automatically include a description of their component parts whenever they are examined.
    RB 9.8 Simple Machines

62. **Tiny Garden**
    A lawn made up of several rooms, with part of the description written automatically.
    RB 3.4 Continuous Spaces and The Outdoors

§5.6 Text with variations

Text sometimes needs to take different forms in different circumstances. Perhaps it needs an extra sentence if something has happened, or perhaps only one altered word.

say "[if (a condition)]"

This text substitution produces no text. It's used only for a side-effect: it says that the text following should be said only if the condition is true. That continues until the end of the text, or until an "[end if]" substitution, whichever comes first. If the "[otherwise]" and "[otherwise if]" substitutions are also present, they allow alternatives to be added in case the condition is false. Example:

```
The wine cask is a container. The printed name of the cask is "[if open]broached, empty cask[otherwise]sealed wine cask".
```

we find that the cask is described as "a broached, empty cask" when open, and "a sealed wine cask" when closed. A longer example which begins and ends with fixed text, but has two alternatives in the middle:

```
The Customs Wharf is a room. "Amid the bustle of the quayside, [if the cask is open]many eyes stray to your broached cask. [otherwise]nobody takes much notice of a man heaving a cask about. [end if]Sleek gondolas jostle at the plank pier."
```

say "[unless (a condition)]"

This text substitution produces no text. It's used only for a side-effect: it says that the text following should be said only if the condition is false. That continues until the end of the text, or until an "[end if]" substitution, whichever comes first. If the "[otherwise]" and "[otherwise if]" substitutions are also present, they allow alternatives to be added in case the condition is true. Example:

```
The Customs Hall is a room. "With infinite slowness, with ledgers and quill pens, the clerks ruin their eyesight.[unless the player is a woman] They barely even glance in your direction."
```

say "[otherwise]"

or…

say "[else]"

This text substitution produces no text, and can be used only following an "[if …]" or "[unless …]" text substitution. It switches from text which appears if the condition is true, to text which appears if it is false. Example:

```
The wine cask is a container. The printed name of the cask is "[if open]broached,
empty cask[otherwise]sealed wine cask".
```

say "[end if]"

This text substitution produces no text, and can be used only to close off a stretch of varying text which begins with "[if …]".

say "[end unless]"

This text substitution produces no text, and can be used only to close off a stretch of varying text which begins with "[unless …]".

say "[otherwise/else if (a condition)**]"**

This text substitution produces no text, and can be used only following an "[if …]" or "[unless …]" text substitution. It gives an alternative text to use if the first condition didn't apply, but this one does. Example:

```
The wine cask is a container. The printed name of the cask is "[if open]broached,
empty cask[otherwise if transparent]sealed cask half-full of sloshing
wine[otherwise]sealed wine cask".
```

say "[otherwise/else unless (a condition)**]"**

This text substitution produces no text, and can be used only following an "[if …]" or "[unless …]" text substitution. It gives an alternative text to use if the first condition didn't apply, and this one is false too.

We sometimes need to be careful about the printing of line breaks:

```
The Cell is a room. "Ah, [if unvisited]the unknown cell. [otherwise]the usual cell."
```

This room description has two possible forms: "Ah, the unknown cell. ", at first sight, and then "Ah, the usual cell." subsequently. But the second form is rounded off with a line break because the last thing printed is a ".", whereas the first form isn't, because it ended with a space. The right thing would have been:

```
The Cell is a room. "Ah, [if unvisited]the unknown cell.[otherwise]the usual cell."
```

allowing no space after "unknown cell."

When varying descriptions are being given for kinds of rooms or things, it can be useful to make use of a special value called "item described", which refers to the particular one being looked at right now. For example:

```
A musical instrument is a kind of thing. The tuba and the xylophone are musical instruments. The description of a musical instrument is usually "An especially shiny, well-tuned [item described]."
```

The tuba now has the description "An especially shiny, well-tuned tuba.", and similarly for the xylophone.

The "item described" value can similarly be used in any textual property of a room or thing, and in particular can be used with the "initial appearance" and "printed name" properties, which are also forms of description.

Examples

63. **When?**
    A door whose description says "…leads east" in one place and "…leads west" in the other.
    RB 3.5 Doors, Staircases, and Bridges

64. **Whence?**
    A kind of door that always automatically describes the direction it opens and what lies on the far side (if that other room has been visited).
    RB 3.5 Doors, Staircases, and Bridges

65. **Persephone**
    Separate the player's inventory listing into two parts, so that it says "you are carrying…" and then (if the player is wearing anything) "You are also wearing…".
    RB 6.7 Inventory

§5.7 Text with random alternatives

Sometimes we would like to provide a little quirky variation in text, especially in messages which will be seen often. We can achieve this with the "[one of]… [or] … [or] …" construction.

say "[one of]"

This text substitution produces no text. It's used only for a side-effect: it switches between a number of alternative texts, which follow it and are divided by "[or]" substitutions, according to a strategy given in a closing substitution. Example:

```
"You flip the coin. [one of]Heads[or]Tails[purely at random]!"
```

Here there are just two alternatives, and the strategy is "purely at random". Exactly half of the time the text will be printed as "You flip the coin. Heads!"; and the other half, "You flip the coin. Tails!".

say "[or]"

This text substitution produces no text, and can be used only in a "[one of]..." construction. It divides alternative wordings. Example:

```
"You flip the coin. [one of]Heads[or]Tails[purely at random]!"
```

There are seven possible endings, each making the choice of which text to follow in a different way:

say "[purely at random]"

This text substitution produces no text, and can be used only to end a "[one of]..." construction. It indicates that the alternatives are chosen uniformly randomly.

say "[then purely at random]"

This text substitution produces no text, and can be used only to end a "[one of]..." construction. It indicates that the alternatives are chosen in sequence until all have been seen, but that after that they are chosen uniformly randomly.

say "[at random]"

This text substitution produces no text, and can be used only to end a "[one of]..." construction. It indicates that the alternatives are chosen at random except that the same choice cannot come up twice running. This is useful to avoid the deadening effect of repeating the exact same message. Example:

```
"The light changes randomly again; now it's [one of]green[or]amber[or]red[at random]."
```

Here we can safely say the light "changes", because the new colour cannot be the same as the one printed the last time.

say "[then at random]"

This text substitution produces no text, and can be used only to end a "[one of]..." construction. It indicates that the alternatives are chosen in sequence until all have been seen, and then after that, at random except that the same choice cannot come up twice running. Example:

```
"Maybe the murderer is [one of]Colonel Mustard[or]Professor Plum[or]Cardinal Cerise[then at random]."
```

say "[sticky random]"

This text substitution produces no text, and can be used only to end a "[one of]..." construction. It indicates that a random choice is made the first time the text is printed, but that it sticks from there on. Example:

```
"The newspaper headline is: [one of]War Casualties[or]Terrorists[or]Banks[sticky random] [one of]Continue To Expand[or]Lose Out[sticky random]."
```

Although the newspaper headline will change with each playing, it will not alter during play.

say "[as decreasingly likely outcomes]"

This text substitution produces no text, and can be used only to end a "[one of]..." construction. It indicates that the alternatives are chosen at random, except that the first is most likely to be chosen, the second is next most likely, and so on down to the rarest at the end. Example:

```
"Zorro strides by, [one of]looking purposeful[or]grim-faced[or]deep in thought[or]suppressing a yawn[or]scratching his ribs[or]trying to conceal that he has cut himself shaving[as decreasingly likely outcomes]."
```

There are six outcomes here: the first is six times as likely as the last, and those in between are similarly scaled, so Zorro cuts himself shaving only once in 21 tries, while he looks purposeful almost a third of the time.

But suppose we want to tuck some useful information in these messages, and we want to be sure that the player will see it. Because all of the above options involve randomness, it's possible that an unlucky player might miss a clue placed into only one variant of the message. One fix for this is to make sure that everything turns up sooner or later:

say "[in random order]"

This text substitution produces no text, and can be used only to end a "[one of]..." construction. A random order is chosen for the alternative passages of text, and they are used in that order as the text is printed again and again. When one random cycle finishes, a new one begins. The effect is somewhat like the "shuffle album" feature on an iPod. Example:

```
"You dip into the chapter on [one of]freshwater fish[or]hairless mammals[or]extinct birds[or]amphibians such as the black salamander[in random order]."
```

One small restriction: if there are more than 32 variations, purely random choices will be printed, and there will be no guarantee that repeats are prevented.

Another fix is to avoid randomness altogether:

say "[cycling]"

This text substitution produces no text, and can be used only to end a "[one of]..." construction. It indicates that the alternatives are used one at a time, in turn: after the last one is reached, we start again from the first. Example:

```
"The pundits discuss [one of]the weather[or]world events[or]celebrity gossip[cycling]."
```

say "[stopping]"

This text substitution produces no text, and can be used only to end a "[one of]..." construction. It indicates that the alternatives are used one at a time, in turn: once the last one is reached, it's used forever after. Example:

```
"[one of]The phone rings[or]The phone rings a second time[or]The phone rings again[stopping]."
```

Finally, here's a convenient shorthand for one of the commonest things needed:

say "[first time]"

or...

say "[only]"

This pair of text substitutions causes whatever is between them to be printed only the first time the text is printed. Example:

```
"The screen door squeaks loudly as when you open it. [first time]Well, you'll get used to it eventually. [only]"
```

This is exactly equivalent to

```
"The screen door squeaks loudly as when you open it. [one of]Well, you'll get used to it eventually. [or][stopping]";
```

but easier to read.

Something to watch out for is that texts are sometimes being printed internally for purposes other than actual output which the player can see, and this is particularly true of names. For example:

```
Before printing the name of the traffic signal: say "[one of]green[or]amber[or]red[cycling] ".
```

This looks good for some purposes, but may not cycle in the sequence expected, and can result in incorrect indefinite articles being printed -- "an red traffic signal", for example. What's happening is that the name is being printed internally to see whether it begins

with a vowel; that prints "amber traffic signal", but invisibly to us, and since this does begin with a vowel, "an" is visibly printed; then the name is visibly printed, but now it has changed to "red traffic signal", and so the result on screen is "an red traffic signal". There are many ways to avoid this (for example, to give the traffic signal a state which changes every turn, not every time the name is printed), but it's a trap to look out for.

Examples

66. **Radio Daze**
    A radio that produces a cycle of output using varying text.
    RB 9.9 Televisions and Radios

67. **Camp Bethel**
    Creating characters who change their behavior from turn to turn, and a survey of other common uses for alternative texts.
    RB 7.2 Liveliness

§5.8 Line breaks and paragraph breaks

Inform controls the flow of text being said so that it will read, to the player, in a natural way. There are two principles:

(a) pieces of text ending with full stop, exclamation or question marks will be followed by line breaks (or "new lines", as some computer programming languages would call them); and

(b) pieces of text produced by different rules in Inform will be separated by paragraph breaks.

The effect is that authors can forget about paragraph spacing most of the time, but the mechanism is not impossible to fool, so text substitutions are provided to override the usual principles. First, to manipulate line breaks:

say "[line break]"

This text substitution produces a line break. Example:

```
"There is an endless sense of[line break]falling and[line break]falling."
```

Line breaks are not paragraph breaks, so the result is:

```
There is an endless sense of
falling and
falling.
```

with no extra vertical spacing between these lines.

say "[no line break]"

This text substitution produces no text. It's used only for a side-effect: it prevents a line break where Inform might otherwise assume one. Example:

```
"The chorus sing [one of]Jerusalem[or]Rule, Britannia![no line break][at random]."
```

Here the "[no line break]" stops Inform from thinking that the exclamation mark means a sentence ending - it's part of the name of the song "Rule, Britannia!". So we get

```
The chorus sing Rule, Britannia!.
```

with no line break between the "!" and ".".

And similarly for paragraph breaks. Because Inform can be pretty trigger-happy with these, the first need is for a way to stop them:

say "[run paragraph on]"

This text substitution produces no text. It's used only for a side-effect: it prevents a paragraph break occurring after the present text is printed, in case Inform might be tempted to place one there. Example:

```
Before taking something, say "Very well. [run paragraph on]".
```

This allows the reply to, say, TAKE ENVELOPE to be

```
Very well. Taken.
```

rather than

```
Very well.

Taken.
```

which is how texts produced by different rules would normally be shown. (It's a traditional printer's term. See Oldfield's Manual of Typography, 1892, under "When two paragraphs are required to be made into one, or, in technical language, 'to run on'.")

But sometimes we actually want paragraph breaks in unexpected places. One way is to force them outright:

say "[paragraph break]"

This text substitution produces a paragraph break. Example:

```
"This is not right.[paragraph break]No, something is terribly wrong."
```

Paragraph breaks have a little vertical spacing in them, unlike mere line breaks, so the result is:

```
This is not right.

No, something is terribly wrong.
```

More subtly, we can give Inform the option:

say "[conditional paragraph break]"

This text substitution either produces a paragraph break, or no text at all. It marks a place where Inform can put a paragraph break if necessary; in effect it simulates what Inform does every time a "before" or similar rule finishes. If there is text already printed, and text then follows on, a paragraph break is made. But if not, nothing is done. This is sometimes useful when producing a large amount of text which changes with the circumstances so that it is hard to predict in advance whether a paragraph break is needed or not.

Really finicky authors might possibly want to know this:

if a paragraph break is pending:

This condition is true if text has recently been said in such a way that Inform expects to add a paragraph break at the next opportunity (for instance when the present rule ends and another one says something, or when a "[conditional paragraph break]" is made).

Finally, there are two special sorts of paragraph break for special circumstances. They are mainly used by the Standard Rules, and imitate the textual layout styles of traditional IF.

say "[command clarification break]"

This text substitution produces a line break, and then also a paragraph break if the text immediately following is a room description brought about by having gone to to a different room and looking around, in which case a line break should be added. In traditional IF, this is used when clarifying what Inform thinks the player intended by a given command. Example:

```
say "(first opening [the noun])[command clarification break]";
```

might result in

```
(first opening the valise)
You rummage through the valise for tickets, but find nothing.
```

say "[run paragraph on with special look spacing]"

This text substitution produces no text. It's used only for a side-effect: it indicates that the current printing position does not follow a skipped line, and that further material is expected which will run on from the previous paragraph, but that if no further material turns up then a skipped line would be needed before the next command prompt. (It's very likely that only the Standard Rules will ever need this.)

Example

68. **Beekeeper's Apprentice**
    Making the SEARCH command examine all the scenery in the current location.
    RB 6.5 Examining

§5.9 Text with type styles

Inform does not go in for the use of fonts: a work of IF will be rendered with different fonts on different machines anyway, from tiny personal organisers up to huge workstations. However, it does allow for a modest amount of styling.

say "[bold type]"

This text substitution produces no text. It's used only for a side-effect: to make the text following it appear in bold face. "[roman type]" should be used to switch back to normal. Example:

```
"Jane looked down. [bold type]Danger[roman type], the sign read."
```

say "[italic type]"

This text substitution produces no text. It's used only for a side-effect: to make the text following it appear in italics. "[roman type]" should be used to switch back to normal. Example:

```
"This is [italic type]very suspicious[roman type], said Peter."
```

```
say "[roman type]"
```

This text substitution produces no text. It's used only for a side-effect: to return to ordinary Roman type after a previous use of "[bold type]" or "[italic type]".

but there is one other effect we can employ:

```
say "[fixed letter spacing]"
```

This text substitution produces no text. It's used only for a side-effect: to make the text following it appear with fixed letter spacing. In variable letter spacing, a lower case "m" is much wider than an "l", which is natural to the eye since it has been printing practice since the Renaissance. Fixed letter spacing is more like typewriting, and it is best used to reproduce typewritten text or printed notices; it can also be convenient for making simple diagrams. Example:

```
"On the door is written: [fixed letter spacing]J45--O-O-O[variable letter spacing]."
```

```
say "[variable letter spacing]"
```

This text substitution produces no text. It's used only for a side-effect: to return to ordinary letter spacing after a previous use of "[fixed letter spacing]".

Whichever effect we use, we should be careful to ensure that we return to normal -- roman type and variable letter spacing -- after any specially-treated text has been printed. Combining these effects (for, say, bold fixed-spaced lettering) is not guaranteed to work, though on some platforms it will.

Example

69. **Garibaldi 2**
    Adding coloured text to the example of door-status readouts.

RB 12.1 Typography

§5.10 Accented letters

Inform 7 is infused by the English language, so it's a challenge using it to write a work of IF in any other language. (With that said, extensions do exist which have made

considerable progress on this problem: nil desperandum.) But even a book in English contains occasional quotations or words borrowed from other tongues, so we are going to need more than plain A to Z.

The world has a bewildering range of letters, accents, diacritics, markers and signs. Inform tries to support the widest range possible, but the works of IF produced by Inform are programs which then have to be run on a (virtual) computer whose abilities are more constrained: few players will have an Ethiopian font installed, after all. So a degree of caution is called for.

(a) **Definitely safe to use.** Inform's highest level of support is for the letters found on a typical English typewriter keyboard, including both the $ and £ signs (but not the Yen or Euro symbols ¥ and €), and in addition the following:

```
ä, á, à, ã, å, â and Ä, Á, À, Ã, Å, Â
ë, é, è, ê and Ë, É, È, Ê
ï, í, ì, î and Ï, Í, Ì, Î
ö, ó, ò, õ, ø, ô and Ö, Ó, Ò, Õ, Ø, Ô
ü, ú, ù, û and Ü, Ú, Ù, Û
ÿ, ý and Ý (but not Ÿ)
ñ and Ñ
ç and Ç
æ and Æ (but not œ or Œ)
ß
¡, ¿
```

These characters can be typed directly into the Source panel, and can be used outside quotation marks: we can call a room the Église, for instance.

(b) **Characters which can safely be used, but will be simplified.** As it reads in the text, Inform silently converts all kinds of dash (en-rules, em-rules, etc.) to simple hyphens; converts the multiplication symbol to a lower case "x"; converts all kinds of space other than tabs (em-spaces, non-breaking spaces, etc.) to simple spaces, and all kinds of quotation marks to "straight" (non-smart) marks.

(c) **Characters which can be used provided they are in quoted text (other than boxed quotations), and which will probably but not certainly be visible to the player.** All other Latin letter-forms, including the œ ligature, East European forms such as ő, ş and ž, and Portuguese forms such as ũ; the Greek and Cyrillic alphabets, with their associated variants and accents; and the principal currency symbols, such as € and ¥. Such characters are not legal in unquoted text: so we could write

```
The Churchyard is a room. The printed name of the Churchyard is "Łodz Churchyard".
```

but not

```
Łodz Churchyard is a room.
```

Moreover, the player is not allowed to type these characters in commands during play: or rather, they will not be recognised if he does. They are for printing only.

(d) **Characters which might work in quoted text, or might not**. The Arabic and Hebrew alphabets are fairly likely to be available; miscellaneous symbols are sometimes legible to the player, sometimes not. Other alphabets are chancier still. (If a work of IF depends on these being visible, it may be necessary to instruct players to use specific interpreters, or to provide a way for the player to test that all will be well.)

§5.11 Unicode characters

As we have seen, Inform allows us to type a wide range of characters into the source text, although the more exotic ones may only appear inside quotation marks. But they become more and more difficult to type as they become more obscure. Inform therefore allows us to describe a letter using a text substitution rather than typing it directly.

Unicode characters can be named (or numbered) directly in text. For example:

```
"[unicode 321]odz Churchyard"
```

produces a Polish slashed L. Characters can also be named as well as numbered:

```
"[unicode Latin capital letter L with stroke]odz Churchyard"
```

The Unicode standard assigns character numbers to essentially every marking used in text from any human language: its full range is enormous. (Note that Inform writes these numbers in decimal: many reference charts show them in hexadecimal, or base 16, which can cause confusion.)

This means, for instance, that we can write text such as:

```
"Dr Zarkov unveils the new [unicode Hebrew letter alef] Nought drive."
"Omar plays 4[unicode black spade suit] with an air of triumph."
```

Admittedly, character names can get a little verbose:

```
"[unicode Greek small letter omega with psili and perispomeni and ypogegrammeni]"
```

Inform can "only" handle codes [unicode 32] up to [unicode 131071], and note that if the story settings are to compile to the Z-machine, this range stops at 65535: thus many emoji characters - say, [unicode fish cake with swirl design] - can only be used if the story will compile to Glulx or another modern target. But by default, stories are compiled the modern way, so this should not be a problem in practice.

There are far too many possible names to list here: formally, any character name in the Basic Multilingual Plane or the Supplementary Multilingual Plane of version 15.0.0 of the Unicode standard can be used. See:

```
https://en.wikipedia.org/wiki/Plane_(Unicode)
```

But before getting carried away, we should remember the hazards: Inform allows us to type, say, "[unicode Saturn]" (an astrological sign) but it appears only as a black square if the resulting story is played by an interpreter using a font which lacks the relevant sign. For instance, Zoom for OS X uses the Lucida Grande and Apple Symbol fonts by default, and this combination does contain the Saturn sign: but Windows Frotz tends to use the Tahoma font by default, which does not. (Another issue is that the fixed letter spacing font, such as used in the status line, may not contain all the characters that the font of the main text contains.) To write something with truly outré characters is therefore a little chancy: users would have to be told quite carefully what interpreter and font to use to play it.

At one time, Inform could only use named Unicode values in a story which had first included an extension:

```
Include Unicode Character Names by Graham Nelson.
Include Unicode Full Character Names by Graham Nelson.
```

This is no longer the case: no such inclusion need now be made, and indeed, those extensions have been removed from Inform as redundant.

Example

70. **The Über-complète clavier**
    This example provides a fairly stringent test of exotic lettering.

RB 12.1 Typography

§5.12 Displaying quotations

Text is normally printed in between the typed commands of the player, rolling upwards from the bottom of the screen, as if a dialogue is being typed by an old-fashioned teletype. But it can also be displayed in a bolder way, floating above the main text, and this is sometimes used to display quotations.

display the boxed quotation (text)

This phrase displays the given text on screen in an overlaid box. For reasons to do with the way such quotations are plotted onto the screen, their text is treated literally: no substitutions in square brackets are obeyed. The quotation will only ever appear once, regardless of the number of times the "display the boxed quotation ..." phrase is reached. Rather than being shown immediately - and thus, probably, scrolling away before it can be seen - the display is held back until the next command prompt is shown to the player. Example:

```
After looking in the Wabe, display the boxed quotation
    "And 'the wabe' is the grass-plot round
    a sun-dial, I suppose? said Alice,
    surprised at her own ingenuity.

    Of course it is. It's called 'wabe,'
    you know, because it goes a long way
    before it, and a long way behind it --

    -- Lewis Carroll".
```

This was the original example used in *Trinity*, by Brian Moriarty, which invented the idea. A player exploring Kensington Gardens comes upon a location enigmatically called The Wabe; and by way of explanation, this quotation pops up.

Note that exotic accented characters, such as the "Ł" in "Łodz", can't be displayed in boxed quotations. This is only a simple feature, and we should go in search of a suitable extension for fancier screen effects if we would like to do more.

§5.13 Making new substitutions

If we have some textual effect which needs to occur in several different messages, we might want to create a new text substitution for it. For instance:

```
The Missile Base is a room. "[security notice]Seems to be a futuristic missile base." M's Office is east of the Missile Base. "[security notice]Admiral Sir M.- M.- glares up from his desk."

To say security notice:
    say "This area is a Prohibited Place within the meaning of the Official Secrets Act 1939. "
```

This is only the tip of the iceberg in how to define ways to do things using "To…", as we shall see. The definition makes "say the security notice" a new phrase known to Inform. A text substitution is exactly a phrase whose name begins with "say" (well - except for the "say" phrase itself), so the effect is that "[security notice]" is a new text substitution. Several of the examples in this chapter make use of this trick.

Inform often ignores the casing of the text it reads, but sometimes uses it as a clue to meaning. We have already seen that "[an item]" and "[An item]" produce different results, for instance. Similarly, it's possible to define two text substitutions which are the same except for the initial casing. We might write:

```
To say Security Notice:
    say "THIS AREA IS A PROHIBITED PLACE WITHIN THE MEANING OF THE OFFICIAL SECRETS ACT 1939. "
```

And now Inform will act on "[Security Notice]" differently from "[security notice]".

See Also

The phrasebook for other forms of phrase besides To say....

Examples

71. **Fifty Ways to Leave Your Larva**
    Using text substitution to make characters reply differently under the same circumstances.
    RB 2.1 Varying What Is Written

72. **Fifty Times Fifty Ways**
    Writing your own rules for how to carry out substitutions.
    RB 2.1 Varying What Is Written

6. Descriptions

§6.1 What are descriptions?
§6.2 Adjectives and nouns
§6.3 Sources of adjectives
§6.4 Defining new adjectives
§6.5 Defining adjectives for values
§6.6 Whereabouts on a scale?
§6.7 Comparatives
§6.8 Superlatives
§6.9 Which and who
§6.10 Existence and there
§6.11 A word about in
§6.12 A word about nothing
§6.13 To be able to see and touch
§6.14 Adjacent rooms and routes through the map
§6.15 All, each and every
§6.16 Counting while comparing

§6.1 What are descriptions?

It is in describing circumstances that Inform really capitalises on the concise, expressive power of natural language, and this chapter brings together the facts about "descriptions".

The simplest descriptions consist of a noun alone. Some refer to single things ("lantern", or "wine cask"), others to kinds of thing ("dead end" or "container"). But we have also seen adjectives alone:

```
The oaken desk is fixed in place.
```

Here, "fixed in place" is a description which, to Inform's simple-minded grammar, is a single adjective. And of course adjectives and nouns can be combined:

```
The cargo trunk is an openable container.
```

The description "openable container" consists of the noun "container", meaning a kind of thing, and the adjective "openable", which means one of the two possible states of an either/or property held by that thing.

As the next chapter will show, rules also make great use of descriptions:

```
Instead of throwing something at a closed openable door, say "Or you could just use the handle like anyone else, of course."
```

We have already seen that we can list the items fitting a given description:

```
"You look down at [the list of things in the basket]."
```

It's also sometimes convenient to count them up:

number of (description of values) ⇒ *number*

This phrase counts the number of values matching the description, which may of course be 0. Example:

```
number of open doors
```

produces the number of doors, anywhere in the model world, which are currently open. A Problem message is produced if the number is potentially infinite, or impractical to count: for instance, Inform rejects "number of odd numbers".

It is because descriptions are so widely useful that they deserve a chapter of their own, and this is it.

§6.2 Adjectives and nouns

Descriptions can contain a noun, but need not, and can contain any number of adjectives:

```
supporter = the noun supporter
closed = the adjective closed
the open wine cask = the adjective open + the noun wine cask
something portable = (some) + the noun thing + the adjective portable
```

Note that we are not allowed to have more than one noun in the same description (something English occasionally does allow as a coded form of emphasis, as in "the man Jenkins" or "the harlot Helen").

Nouns are simple enough, referring either to kinds or specific things. The noun "something" means "some thing", so is actually a reference to the kind "thing". Inform treats this as having the same meaning as "anything", and all told there are eight special nouns of this kind, but with only three different meanings between them:

```
something = anything
someone = anyone = somebody = anybody
somewhere = anywhere
```

So for instance "anybody male" or "somewhere dark" are valid descriptions. These eight nouns are unusual in being allowed to come at the front of a description: nouns are usually expected to be at the end. (Inform also understands "nothing", "nowhere", "nobody", "no-one" and even "no one", which in a sense are opposites of "something" and the like, but for now we'll look at descriptions of things which do exist rather than don't.)

§6.3 Sources of adjectives

We have seen two sorts of adjectives so far: those which refer to either/or properties, like "open" and "closed", and those which come out of new kinds of value. If we define

```
Texture is a kind of value. The textures are rough, stubbly and smooth. Everything has a texture.
```

…then "rough", "stubbly" and "smooth" all become adjectives. (That last sentence "Everything has a texture" was essential, because without it Inform would not know that these words could meaningfully be applied to things.)

In addition to these adjectives, we can create new ones (as we shall see), and a few special adjectives such as "visible", "touchable" and "adjacent" are already defined for us by Inform.

§6.4 Defining new adjectives

Suppose we want to coin a word for supporters currently supporting something. We can do so with the following sentence:

```
Definition: A supporter is occupied if something is on it.
```

Note the colon, which is essential, and the usage of "it" in the definition part to refer to the object in question. (For this purpose we would write "it" even if we were defining a term about, say, a woman instead of a supporter, so that "she" or "her" might seem more appropriate - but see below.)

This creates the adjective "occupied", and gives it a definition valid for supporters. That restriction on validity means that non-supporters would always fail the description "something occupied"; which might be unfortunate if we wanted to talk about rooms being occupied. We could give a second definition thus:

```
Definition: A room is occupied if a person is in it.
```

These are entirely different senses of the word "occupied" - a mantelpiece is occupied if an invitation is on it, but for a drawing room to be occupied there must be human presence - and Inform applies whichever sense is relevant when deciding whether or not a given object is "occupied".

Often, though not always, we also want to give a name to the opposite possibility. We can do that as follows:

```
Definition: A room is occupied rather than unoccupied if a person is in it.
```

The "rather than…" part of the definition is optional, but it saves having to write a boringly similar definition of "unoccupied" out in longhand. (Note that Inform does not guess the meaning of "unoccupied" unless it has been explicitly told it. Such guesses are too risky, when so many "un-" words fail to conform to this pattern: "unified", "uncle", "ungulate" and so on.)

Newly defined adjectives cannot be used when creating things, because they are not explicit enough. Inform could not satisfy:

```
The Ballroom is occupied. The bucket is a large container.
```

because there is not enough information: by whom is the Ballroom occupied? How large, exactly? On the other hand, newly defined adjectives are very helpful in conditions and for rules, as we shall see later on.

It is occasionally clumsy having to refer to the subject of a definition using "it". We can avoid this and give the definition better legibility by supplying a name instead. For instance:

```
Definition: a direction (called thataway) is viable if the room thataway from the location is a room.
```

which is a good deal easier to read than

```
Definition: a direction is viable if the room it from the location is a room.
```

See Also

New conditions, new adjectives for giving more extensive definitions of new adjectives,

using phrases.

Example

73. **Finishing School**
    The "another" adjective for rules such as "in the presence of another person".

RB 5.3 Characterization

§6.5 Defining adjectives for values

In general, any noun can have adjectives applied to it, and this means that values can have adjectives just as objects can. We have already seen that they can (in some cases, at least) have either/or properties, and this gives them adjectives just as for objects. But we can also write out definitions which apply to values:

```
Definition: A number is round if the remainder after dividing it by 10 is 0.
Definition: A time is late rather than early if it is at least 8 PM.
```

That makes the numbers 20 and 170 but not 37 meet the description "a round number", and the times 8 PM and 11:23 PM but not 9 AM meet the description "a late time". Because they come up fairly often, Inform contains several adjectives for numbers built in:

```
positive - one which is greater than zero (but not 0 itself)
negative - one which is less than zero (but not 0 itself)
even - a number like ..., -4, -2, 0, 2, 4, ...
odd - a number like ..., -5, -3, -1, 1, 3, 5, ...
```

Similarly, two useful adjectives are built in to talk about text:

```
empty - the text "", with no characters in it, not even spaces
non-empty - any text which does have at least one character in
```

Adjectives can have multiple definitions and, as long as each applies to a different sort of noun, there will be no problem. We could write:

```
A thing can be round, square or funny-shaped.
A container can be odd or ordinary.
```

And these definitions of "round" and "odd" will not interfere with the ones applying to numbers, because Inform can always look at the noun to see which definition is meant in any given case. For instance,

```
if the score is round, ...
```

must mean "round" in the sense of numbers, because the score is a number. Inform itself makes good use of this; "empty" also has meanings applying to rulebooks, lists and activities, for instance, as will be seen later.

Although it's more usual to give a definition to apply to a whole kind, we can actually give a specific definition to apply to just a single object or named value. For example:

```
A colour is a kind of value. The colours are red, green and blue.
Definition: red is subtle if the player is female.
Definition: a colour is subtle if it is blue.
```

The first definition of "subtle" takes precedence, of course, since it has the more specific domain - it applies only to red. The effect of this is that, if the player's female, the subtle colours are red and blue; if not, just blue.

Example

74. **Only You...**
    Smoke which spreads through the rooms of the map, but only every other turn.

RB 10.1 Gases

§6.6 Whereabouts on a scale?

Adjectives are often used in English to give a sense of where something is on a sliding scale. We talk about "a tall man" and "a short man", but without meaning that all men are either tall or short. If pushed, we might say that tall means about 6 feet and up, short means about 5 feet 6 and down, but we more often compare one person's height against another's.

Inform allows us to use adjectives in the same way. For example, every container has a number called its "carrying capacity". We can define:

```
Definition: A container is huge if its carrying capacity is 20 or more.
Definition: A container is large if its carrying capacity is 10 or more.
Definition: A container is standard if its carrying capacity is 7.
Definition: A container is small if its carrying capacity is 5 or less.
```

These definitions are similar to those in the previous section, but have a very specific (and strictly enforced) shape to them. The adjective must be a single word. We have to say "its" (i.e., of it), not the ungrammatical "it's"; we have to specify a property, and a

literal value of it, and we must either give an exact value or else conclude with "or more" or "or less". If we create something with one of these properties:

```
The basket is a large container in the Shop. The thimble is a small container in the Shop.
The matchbox is a standard container in the Shop.
```

then they will have the most moderate values they can have, that is, the basket will have carrying capacity 10 and the thimble 5 (and of course the matchbox 7). Both of the following tests will then fail:

```
if the basket is huge ...
if the basket is a small container ...
```

because the basket is neither huge nor small, but somewhere in between.

Sometimes the meaning of adjectives must depend on their context, as we see from the following example, where we assess heights in inches:

```
A person has a number called height. Definition: A man is tall if his height is 72 or more.
Definition: A woman is tall if her height is 68 or more.
```

Inform then judges whether someone is or is not "tall" using different standards for men and for women, and

```
In the Shop are a tall man and a tall woman.
```

creates a man 72 inches tall and a woman 68 inches tall.

§6.7 Comparatives

The special definitions in the previous section have a further effect. When we define:

```
Definition: A container is large if its carrying capacity is 10 or more.
```

we not only say how to test if something is large (see if its capacity is at least 10) and how to create something large (give it a capacity of exactly 10), we also create a new form of comparison. Thus,

```
if the basket is larger than the thimble ...
if the thimble is not larger than the basket ...
```

are both true. If we also define "huge" and "small", as in the previous section, we also get

comparisons "huger than" and "smaller than". Note that "huger than" has exactly the same meaning as "larger than": we can use whichever wording seems more natural. (For bacilli, for instance, we would probably not say "huger than", even though the meaning would be unambiguous.)

We can also compare two things to see if they share the same value of a property. For instance, to go back to the heights example, once we define "tall" and "short", we get that exactly one of the following will be true:

```
if Adam is taller than Eve ...
if Adam is the same height as Eve ...
if Adam is shorter than Eve ...
```

Though it will not always seem natural wording, we can use the comparison "the same P as" for any property P which has a value. Do we think "if the basket is the same carrying capacity as the thimble" is good English? Maybe, maybe not. But we are always at liberty to spell things out in full:

```
if the carrying capacity of the basket is the carrying capacity of the thimble ...
```

§6.8 Superlatives

Lastly, if we define an adjective in this calibrating way, we also automatically benefit from the use of the superlative form. That is, if we define

```
Definition: A container is large if its carrying capacity is 10 or more.
Definition: A container is small if its carrying capacity is 5 or less.
```

then we can talk about things like this:

```
the largest container
the smallest open container
```

Though we should be careful, in the second case, because we might get nothing: maybe all the containers are closed at the moment this is used. And in general there might be several equally large largest containers, in which case we should not rely on getting any particular one of those rather than another.

Note that Inform constructs comparatives and superlatives by a pretty simplistic system. If we want to use these forms for an adjective expressing the relatively large size of a room, we had better go with "roomy" (roomier, roomiest) - not "spacious" (spaciouser,

spaciousest).

§6.9 Which and who

A description can not only talk about things in terms of themselves, but also in terms of their relationships to the rest of the world. For instance,

```
an open container on the table
a woman inside a lighted room
an animal carried by a man
a woman taller than Mark
something worn by somebody
```

are all valid descriptions. These are really abbreviations, having missed out the words "which is" or "who is", as appropriate:

```
an open container which is on the table
a woman who is inside a lighted room
an animal which is carried by a man
a woman who is taller than Mark
something which is worn by somebody
```

and indeed those are also valid descriptions. The other sentence verbs can all be used here, too. So for instance:

```
a man who does not wear anything
something which supports something
```

And sometimes we should spell out "who is" regardless:

```
a man who is not Sherlock Holmes
```

Since these clauses can be attached to the end of any valid description, descriptions can grow longer still:

```
something worn by a woman who is in a dark room
```

Pedants who flinch when "which" is used to introduce a restrictive clause are welcome to use "that" instead.

Example

75. **Versailles**
    A mirror which will reflect some random object in the room.

RB 8.5 Kitchen and Bathroom

§6.10 Existence and there

"There" is a curious word in English, which mostly refers to some place which is being talked about - but which can sometimes mean the whole world. In Ian Fleming's novel "From Russia With Love", a chapter narrating a committee meeting of SMERSH officers in Istanbul ends with one of the Russians saying:

```
There is a man called Bond.
```

What does this "there" mean? It really just means that Bond exists. In fact, he's watching the meeting through a concealed periscope, but the SMERSH general doesn't know that. All he is saying is that Bond is out there somewhere, and is not imaginary, or dead.

Inform also allows "there is" (or "there are") to talk about what exists, or does not. This is especially useful if, for some reason, we don't want to give a name to something. For example:

```
There is a door in the Summerhouse.
```

Another reason might be that we want to create something but not put it anywhere. If Inform reads the sentence:

```
There is a man called Bond.
```

then it creates a man, gives him the name Bond, but places him initially off-stage - not in any room, that is, but available to be brought into play later on, like an actor who is not needed until Act II.

"There" also provides a useful way to test what exists:

```
if there is a woman in the Summerhouse, ...
```

Or even:

```
if there is a woman, ...
```

which will be true if the model world contains even a single woman, on-stage or off. The alternative "there are" can also be used:

```
if there are women in the Summerhouse, ...
```

but note that this does not necessarily imply more than one woman is present, despite the plural. If we want that, we have to be more explicit:

```
if there is more than one woman in the Summerhouse, ...
```

or, of course, we needn't use "there is" at all:

```
if more than one woman is in the Summerhouse, ...
```

And we can also test non-existence:

```
if there is nobody in the Summerhouse, ...
if there is nothing on the mantelpiece, ...
```

§6.11 A word about in

What does "in" mean? It's worth just a brief diversion to cover this, because "in" has two subtly different meanings.

Meaning 1. Usually, if X is "in" Y then this is because of containment. A croquet ball is "in" a croquet box, which is "in" the Summerhouse. This is the standard meaning, and is the one which happens if we write something like:

```
The croquet ball is in the box.
```

or if we ask a question like:

```
if the croquet box is in the Summerhouse, ...
```

This kind of "in" talks only about direct containment. If we ask

```
if the croquet ball is in the Summerhouse, ...
```

then the answer is that it isn't - it is in the box which is itself in the Summerhouse, but that's not the same thing.

This is almost always the meaning of "in" that we intend. This is only one of a number of relationships between objects - there are also "part of", "on", "worn by" and "carried by", for example. If we have

```
The bird feed is on the sundial.
```

…then "if the bird feed is in the sundial" won't be true: the relationship here is one called support (being on top of, in effect), not containment. But there's no confusion because "on" and "in" are different words, so it's no problem that they have different meanings.

Meaning 2. Much less common. If X is "in" Y and Y is a region, then the meaning is slightly different. Suppose the Garden Area is a region, and contains several rooms - the Croquet Lawn, the Terrace and so on. Then

```
if the croquet box is in the Garden Area, ...
if the bird feed is in the Garden Area, ...
if the Terrace is in the Garden Area, ...
```

are all true. This seems very natural, but in fact is quite different from the first meaning of "in". It allows rooms (and even other regions) to be "in" a region, and it allows indirect containment.

How Inform decides. So which meaning does Inform use, and when? Since these two meanings are so different, it clearly matters.

The answer is that meaning 1 is always the meaning of "X is in Y" unless Y is explicitly the name of a region. Thus:

```
if the croquet box is in the Garden Area, ...
```

is meaning 2, because "Garden Area" is the name of a region. That seems fair enough, but values are indeed sometimes given names (becoming "variables", or values "that vary"). Suppose "mystery value" is a name for a value which is an object, but which has different identities at different times. Then Inform reads

```
if the croquet box is in the mystery value, ...
```

as meaning 1, because whatever "mystery value" is, it isn't explicitly a region name, even if from time to time it might happen to be equal to a region.

That sometimes makes meaning 2 difficult to express. If we ever need it, and this is fairly rare, we can write it like so:

```
if the croquet box is regionally in the mystery value, ...
```

because "regionally in" is always meaning 2 of "in".

§6.12 A word about nothing

Like "in", "nothing" has two slightly different meanings, though here there's much less potential for confusion.

Meaning 1. "Nothing" as "no thing". This is the meaning in sentences like:

```
Definition: a container is bare if nothing is in it.
```

And similar for conditions like "if the box contains nothing". It's a word which describes the absence of things: it says that, though there might have been many possible items here, it turned out that there were none.

Meaning 2. "Nothing" as a value. This is much less commonly seen, but sometimes Inform stores a value such as a property (or a variable) which always has to be an object. In some circumstances, "nothing" is then a special value meaning that this is not set at present. For instance,

```
Definition: a container is impossible if its matching key is nothing.
```

The "matching key" property of a container is always an object, but is allowed to be "nothing" when there isn't a matching key anywhere. (If such a container is locked, nobody will ever be able to unlock it.)

How Inform decides. So which meaning does Inform use, and when? The answer is that it depends on the relationship being talked about. When this is "is", values are being compared and we are using meaning 2. But when it is any other relationship, like "is in" - which talks about containment - then we are using meaning 1.

§6.13 To be able to see and touch

Two of the adjectives built into Inform are:

```
"visible" - the player can see this
"touchable" - the player can touch this
```

So we can write descriptions such as "someone visible" or "a touchable container". We also have adjectives "invisible" and "untouchable", as might be expected. The visibility adjectives are particularly useful because the following is likely to go wrong:

```
if Helen is in a dark room, ...
```

This tests whether the room is dark, of itself; Helen may in fact be able to see by means of a torch, but the room is still "dark".

We can also talk about what other people can see and touch:

```
something which can be seen by Helen
```

are synonymous. Similarly for touch; and we can write such conditions as

```
if Helen cannot see Agamemnon, ...
if Cressida can see Troilus, ...
```

Note that it is essential to establish who does the seeing and touching: so "something which can be seen" will not be allowed, whereas "something which can be seen by Helen" will.

In fact, inside Inform the adjective "invisible" (for instance) has the following straightforward definition:

```
Definition: Something is invisible if the player cannot see it.
```

The exact definitions of visibility and touchability are complicated, because there are so many ways in which vision and touch can be obstructed, but the gist is that they behave as one would expect. Note that in darkness, nothing is visible, and that nobody can see from one room to another. In general anything invisible is also untouchable, but there are a few exceptions to do with being in the dark. Lastly, the player's own body (usually called "yourself" during play) is both visible (in light) and touchable.

Example

76. **Lean and Hungry**
    A thief who will identify and take any valuable thing lying around that he is able to touch.

RB 7.2 Liveliness

§6.14 Adjacent rooms and routes through the map

Another useful adjective built into Inform is "adjacent". Two rooms are said to be adjacent if there is a map connection between them which does not pass through some barrier such as a door. This is easily tested:

```
if the Hallway is adjacent to the Study ...
```

We usually want to know about the places adjacent to the current scene of the action, so that is what the adjective "adjacent" means when applied to rooms. For instance:

```
if somebody is in an adjacent room, ...
```

As with the case of "visible", the adjective is a cut-down version of the more general relationship. This often happens: "worn" and "carried", for instance, imply "by the player" unless something else is specified.

If we want to ask a more direct question, we can obtain specific map connections as follows. (Recall that every map connection leads either to a door, to a room, or to nothing.) If we know which direction we want to look in, then the easiest thing is to use its relation - every direction in the map, say "north", has its own relation, say "mapped north of". So:

```
if the Ballroom is mapped north of the Hallway, ...
```

Alternatively, and particularly if the direction is not a constant,

room (direction) **from/of** (room) ⇒ *room*

This phrase produces the room which the given map direction leads to, or the special value "nothing" if it leads nowhere. If it leads to a door, the result is the room through that door. Examples:

```
say "You look north into [the room north from the Garden]."
if the room north from the Garden is nothing, say "The grass leads nowhere."
```

door (direction) **from/of** (room) ⇒ *door*

This phrase produces the door which the given map direction leads to, or the special value "nothing" if it leads nowhere or to a room. Examples:

```
let the barrier be the door north from the Garden;
if the barrier is a door, say "Well, [the barrier] is in the way.";
```

room-or-door (direction) **from/of** (room) ⇒ *object*

This phrase produces the object which the given map direction leads to, which will always be either a room, a door or the special value "nothing". The phrase is used mainly by the Standard Rules, for technical reasons, and usually it's better to use "room … from …" or "door … from …" instead.

The map can be a great sprawling mass of rooms and doors connected together, and it can be quite hard to find a way through it one step at a time.

best route from (object) **to** (object) ⇒ *object*

This phrase produces a direction to take in order to get from A to B by the shortest number of movements between rooms, or produces "nothing" if there is no way through at all. Example:

```
The description of the brass compass is "The dial points quiveringly to [best route from the location to the Lodestone Room]."
```

Best routes are ordinarily forbidden to go through doors, but if the suffix "using doors" is added as an option then any open or openable and unlocked door may be used on the way; and if "using even locked doors" is given, then any door at all will do. Since magnetism is no respecter of property, that seems right here:

```
The description of the brass compass is "The dial points quiveringly to [best route from the location to the Lodestone Room, using even locked doors]."
```

In practice this simple approach sometimes produces impossible journeys, rather the way Google Maps directions from New York to London would recommend driving down to the docks and then swimming. A more careful approach is to use:

best route from (object) **to** (object) **through** (description of objects) ⇒ *object*

This phrase produces a direction to take in order to get from A to B by the shortest number of movements between rooms which match the given description, or produces "nothing" if there is no way through at all. Example:

```
best route from the Drawbridge to the Keep through visited rooms
```

The condition - in this case, that "visited rooms" must be used - also applies to both ends of the journey, so if either Drawbridge or Keep are unvisited then this is "nothing". (Similarly, saying something like "...through containers" would mean there is never a route.)

Lastly, the following phrases can find out how long the journey would be. (They are quite a bit faster than using the "best route..." phrases repeatedly and counting.)

number of moves from (object) **to** (object) ⇒ *number*

This phrase produces the number of map connections which must be followed in order to get from A to B by the shortest number of movements between rooms. If A and B are the same, the answer is 0; if there is no route at all, the answer is -1. Example:

```
The description of the proximity gadget is "You are now [number of moves from the location to the Sundial] moves from the Sundial.";
```

number of moves from (object) **to** (object) **through** (description of objects) ⇒ *number*

This phrase produces the number of map connections which must be followed in order to get from A to B by the shortest number of movements between rooms matching the given description. If A and B are the same, the answer is 0; if there is no route at all, or if either A or B fail to match the description themselves, the answer is -1.

Route-finding makes it possible to write quite sophisticated conditions concisely. But these sometimes run slowly, because they call for large amounts of computation. How rapidly Inform can find routes depends on which of two methods it uses. Both have advantages - one is fast but needs large amounts of memory, the other is slow but economical. We can choose between them with one of these two use options:

```
Use fast route-finding.
Use slow route-finding.
```

If neither is specified, "fast" is used where the project uses the Glulx virtual machine (see the Settings panel), and "slow" on the Z-machine, where memory is tighter. Fast route-finding is ideally suited to situations where dozens of characters are constantly route-finding through the map as they meander around in a landscape.

See Also

Indirect relations for route-finding through a relation rather than the map.

Examples

77. Mistress of Animals
A person who moves randomly between rooms of the map.
RB 7.13 Traveling Characters

78. All Roads Lead to Mars
Layout where the player is allowed to wander any direction he likes, and the map will arrange itself in order so that he finds the correct "next" location.
RB 3.2 Map

79. Hotel Stechelberg
Signposts such as those provided on hiking paths in the Swiss Alps, which show the correct direction and hiking time to all other locations.
RB 3.4 Continuous Spaces and The Outdoors

80. A View of Green Hills
A LOOK [direction] command which allows the player to see descriptions of the nearby landscape.
RB 3.4 Continuous Spaces and The Outdoors

81. Unblinking
Finding a best route through light-filled rooms only, leaving aside any that might be dark.
RB 3.7 Lighting

§6.15 All, each and every

When testing conditions, we normally talk only about specific things, or else ask if a particular circumstance happens:

```
if the oaken door is open
if a woman is carrying an animal
```

But we can also use "all", "each" or "every" to check the whole range:

```
if each door is open
if anyone is carrying all of the animals
if everybody is in the Dining Room
```

Inform allows other English "determiners" (as they are sometimes called), as well:

```
if some of the doors are open
if most of the doors are open
if almost all of the doors are open
```

are true if at least one case is true, if a majority (any number greater than one half) or at least 80 per cent of the possible cases are true, respectively.

And we can also use "none" and "no". These three are all ways to say the same thing:

```
if no door is open
if all of the doors are not open
if none of the doors is open
```

though it may be clearer style to find a positive way of putting things:

```
if all of the doors are closed
```

All, each and every can be applied to values, too - but only in some cases. For example, suppose we write:

```
Colour is a kind of value. The colours are red, orange, yellow, green, blue, indigo and violet. A colour can be found or unfound.
```

And suppose that, during play, we assign the "found" property to any colour which the player notices on a wall. We might then want to write conditions like so:

```
if every colour is found
if most of the colours are found
if any colour is found
```

But we always have to bear in mind that Inform might have no reasonable way to decide these questions. It will refuse to allow these, for example:

```
if every number is positive
if any text is palindromic
```

(even supposing the adjective "palindromic" has been defined) - there are practically infinitely many possible numbers and texts, so the search cannot sensibly be done.

Example

82. **Revenge of the Fussy Table**
    A small game about resentful furniture and inconvenient objects.

RB 7.3 Reactive Characters

§6.16 Counting while comparing

Lastly we can also ask for a more specific number of possibilities, like so:

```
if two women are carrying animals
if at most three doors are open
if fewer than 10 portable containers are closed
if all but two of the devices are switched on
if there are more than six locked doors
```

Likewise for "less than", "at least", "all except". Something to watch out for is that

```
if two doors are open
```

will be found true if there are (say) three open doors: after all, if three doors are open, then certainly two doors are. So this is not quite counting. We can be more precise by writing

```
if exactly two doors are open
```

The "all but" counts - say, "if all but two doors are open" - are exact: if, in fact, all of the doors are open then this will be found false.

We can often use these counting forms with values, too. As with the use of "all", this is allowed only if the kind of value is one which can reasonably be searched through. For example:

```
if more than three scenes are happening
if there are more than two non-recurring scenes
```

are allowed because the built-in kind of value "scene" (of which much more later on) has only a small number of possible values.

Lastly, note that the "the" in text like "two of the doors" matters: without it, the phrase will not be recognised as a requirement on the number. (This is to make sure that names of things like "two of hearts" are not misinterpreted.)

Example

83. **Yolk of Gold**
    Set of drawers where the item the player seeks is always in the last drawer he opens, regardless of the order of opening.

RB 8.4 Furniture

7. Basic Actions

§7.1 Actions
§7.2 Instead rules
§7.3 Before rules
§7.4 Try and try silently
§7.5 After rules
§7.6 Reading and talking
§7.7 The other four senses
§7.8 Rules applying to more than one action
§7.9 All actions and exceptional actions
§7.10 The noun and the second noun
§7.11 In rooms and regions
§7.12 In the presence of, and when
§7.13 Going from, going to
§7.14 Going by, going through, going with
§7.15 Kinds of action
§7.16 Repeated actions
§7.17 Actions on consecutive turns
§7.18 Postscript on actions

§7.1 Actions

"Actions" are what we get if we try to break down a narrative into its irreducible parts. We might casually say that we are "going shopping", but this involves many smaller steps: going north, going east, entering the shop, examining a loaf of bread, taking it, giving money to the baker, and so on.

An action is an impulse to do something. This may or may not be a reasonable aspiration, and may or may not be achieved. The player's exploration of an interactive fiction is made by a sequence of actions, so much of the designing process comes down to responding to these actions.

We write actions using present participles. For instance, if the player types "take napkin" or "get the napkin" or something similar then the resulting action would be written as:

```
taking the napkin
```

The details of what words the player actually typed are unimportant to us: we deal only in actions.

Every action ends in success or failure. In this context, success means only that the player's intention has been fulfilled. If the player sets out to take the napkin, but finds a million-pound banknote in its folds instead, the action will be deemed to be a failure.

The testing command ACTIONS causes Inform to log every action as it happens, and what its outcome is. (ACTIONS OFF turns this off again.) For instance:

```
>s
[going south]
Security Vault
You can see a metal door here.
[going south - succeeded]
>close door
[closing metal door]
You close the metal door.
[closing metal door - succeeded]
>take door
[taking metal door]
That's fixed in place.
[taking metal door - failed the can't take what's fixed in place rule]
```

A good way to get a sense of the constant flow of actions is to use this command and then wander around an existing work, trying things out. ACTIONS can also give an insight into the web of rules governing play: there are more than ten different ways an attempt to take something can fail, for instance.

§7.2 Instead rules

An action is ordinarily handled by running it through Inform's extensive rulebooks of what might be called normal behaviour. An action such as "taking the napkin", for instance, will be run through numerous checks to see if it is physically reasonable, and then provided all is well, the napkin will be moved into the possession of the player.

Instead, though, we can bypass the rules to do with an action and do something else:

```
Instead of eating the napkin: say "Why not wait for the actual dinner to arrive?"
```

This is an example of a "rule": a set of circumstances followed by a list of instructions. When those circumstances apply, the instructions are carried out. In the case of an "instead" rule, after this is done the action is immediately ended (and counts as a failure, since the original intention has been thwarted).

A friendly alternative can be used when there is only a single instruction, as here: in such

rules the colon can be replaced with a comma. Thus:

```
Instead of eating the napkin, say "Why not wait for the actual dinner to arrive?"
```

Examples

84. **Grilling**
    A grill, from which the player is not allowed to take anything lest he burn himself.
    RB 10.9 Heat

85. **Bad Hair Day**
    Change the player's appearance in response to EXAMINE ME.
    RB 5.3 Characterization

§7.3 Before rules

Despite what was said in the previous section, instead rules do not quite bypass all of the usual rules. Inform knows that certain actions require light: for instance,

```
examining the napkin; looking; looking under the dining table
```

and if it is dark then none of these actions will be allowed, and any instead rules about them will not even be reached. Similarly, Inform knows that most actions require physical access to their objects: so "taking the napkin" would be blocked if the napkin were, say, inside a closed glass bottle, whereas "examining the napkin" would not. So an instead rule can only take effect if the action has already passed these basic reasonability tests.

"Before" rules genuinely precede checking of any kind. They also differ from instead rules in that they do not automatically stop the action in its tracks. Rather, they are provided as an opportunity to ensure that something else is done first. For example:

```
Before taking the napkin, say "(first unfolding its delicate origami swan)".
```

whence

```
>GET NAPKIN
(first unfolding its delicate origami swan)
Taken.
```

We have seen that instead rules automatically stop actions, whereas before rules automatically allow them to continue. We sometimes want to change this. The magic word "instead" can therefore be tacked on to any instruction in a before rule, and will

have the effect of immediately stopping the action at that instruction. Thus the following two rules are (almost) equivalent:

```
Before taking the key, instead say "It seems to be soldered to the keyhole."
Instead of taking the key, say "It seems to be soldered to the keyhole."
```

It is also possible to be explicit about stopping the action:

stop the action

This phrase stops the current rule, stops the rulebook being worked through, and finally stops the action being processed. Example:

```
Before taking the key:
    say "It seems to be soldered to the keyhole.";
    stop the action.
```

Finally, we can prevent Inform from stopping the action when it otherwise might:

continue the action

This phrase ends the current rule, but in a way which keeps its rulebook going, so that the action being processed will carry on rather than being stopped. Example:

```
Instead of taking the napkin:
    say "(first unfolding its delicate origami swan)[command clarification break]";
    continue the action.
```

An "instead" rule ordinarily stops the action when it finishes, so the "continue the action" is needed to make things carry on. (This rule would have been better written as a "before" rule, in fact, but it shows the idea.)

As a general principle, it is good style to use instead rules whenever blocking actions, and before rules only when it is genuinely necessary to do something first but then to continue: in fact, it is good style to use "stop the action" or "continue the action" as little as possible.

Examples

86. **Democratic Process**
    Make PUT and INSERT commands automatically take objects if the player is not holding them.
    RB 6.8 Taking, Dropping, Inserting and Putting

87. **Sand**
    Extend PUT and INSERT handling to cases where multiple objects are intended at once.
    RB 6.8 Taking, Dropping, Inserting and Putting

§7.4 Try and try silently

Chapter 2 noted that surveys of Inform source text showed that the three most popular phrases used by authors are "say", "if" and "now". The fourth most popular is "try", which allows us to trigger off actions ourselves, rather than waiting for the player to type something which generates them. Thus:

try (action)

This phrase makes the action, which has to be named literally, take effect now. Example:

```
Instead of entering the trapdoor, try going up.
```

It's as if the player had typed GO UP as a command. Note that the action has to be specific:

```
try eating something;
```

is not allowed, since it doesn't say exactly what is to be eaten.

The word "try" is intended to make clear that there is no guarantee of success. For example:

```
Before locking the front door, try closing the front door.
```

could go wrong in any number of ways - perhaps the door is closed already, perhaps it is not openable, perhaps somebody has wedged it open. It would be safer to write:

```
Before locking the front door:
    try closing the front door;
    if the front door is open, stop the action.
```

There's no need to say anything if closing didn't work, because the closing action will have done that already. A neater approach still is to use:

silently try (action)

or…

try silently (action)

This phrase makes the action, which has to be named literally, take effect now, under the "silent" convention which means that routine messages aren't printed. Example:

```
try silently taking the napkin;
```

Silence is maintained only if this new action, the taking of the napkin, is successful (so if the napkin is successfully taken, the text "Taken." will not appear): if the action should fail, a suitable objection will be voiced as usual.

So now we have:

```
Before locking the front door:
    try silently closing the front door;
    if the front door is open, stop the action.
```

And this is neater because it won't produce a pointless "You close the front door." message.

See Also

Stored actions for how to store up actions as values and try those, too, so that isn't necessary to name the action as literally as in the examples above.

Examples

88. **Fine Laid**
    Making writing that can be separately examined from the paper on which it appears, but which directs all other actions to the paper.
    RB 6.14 Remembering, Converting and Combining Actions

89. **Hayseed**
    A refinement of our staircase kind which can be climbed.
    RB 3.5 Doors, Staircases, and Bridges

§7.5 After rules

There is pleasantly little to be said about "after" rules. If an action has survived all the rules in its way, and has actually succeeded, then we need to give the player a response which acknowledges this. Inform's normal rules will be sufficient to say something undramatic: for instance, if "taking the napkin" has succeeded then it will reply "Taken." to the player.

An after rule is an opportunity to say something more interesting:

```
After taking the diamonds, say "Taken!"
```

(Well, slightly more interesting.) After rules automatically end the action (as a success), which is what we would want in the above case. Allowing it to continue would simply result in "Taken." being printed as well. However, should we really need to do something and then carry on:

```
After taking the diamonds: say "(Mr Beebe looks up sharply.) "; continue the action.
```

Example

90. **Morning After**
    When the player picks something up which he hasn't already examined, the object is described.
    RB 6.8 Taking, Dropping, Inserting and Putting

§7.6 Reading and talking

A few actions apply not to items alone, but also involve what might be called conversation. The first is the one used for looking things up in books (which is conversation of a kind, even if the author is not present): "consulting … about …". For example,

```
In the Grove is a book of sybilline verses.

After consulting the book about "grove", say "The Grove is a sacred yadda, yadda. There's a tree, that sort of thing. Wisdom."

After consulting the book about "future events", say "It's a bit, what's the word? Delphic."
```

Note that what follows "about" here is a piece of text in double-quotes, and not the name of something. It can be almost any text at all, and in fact we shall later see (in the chapter on "Understanding") that we can match complicated patterns of words, too.

Similar actions are used for conversing with people:

```
After asking the Sybil about "verses", say "She blushes."

After telling the Sybil about "persians", say "She nods gravely."

After answering the Sybil that "I am mad", say "She sighs."
```

These would be produced by commands like "ask sybil about verses", "tell sybil about persians" and "answer i am mad". Answering is little-used except that it also catches commands like "sybil, something unrecognized", which inexperienced players sometimes type. Asking and telling, however, are important actions and the difference between them is often worth preserving. If you would prefer to make "tell sybil about X" do the same as "ask sybil about X", the following rule would serve:

```
Instead of telling the Sybil about something, try asking the Sybil about it.
```

Games with a lot of conversation often involve great heaps of rules like the ones above, which can be repetitious to type out. We shall also later see (in the chapter on "Tables") that we can tabulate questions and answers in a much more concise way, if we prefer.

See Also

Topic columns for table-based ways to store and retrieve conversation.

Examples

91. **Sybil 1**
Direct all ASK, TELL, and ANSWER commands to ASK, and accept multiple words for certain cases.
RB 7.7 Saying Simple Things

92. **Lucy**
Redirecting a question about one topic to ask about another.
RB 6.14 Remembering, Converting and Combining Actions

93. **Sybil 2**
Making the character understand YES, SAY YES TO CHARACTER, TELL CHARACTER YES, ANSWER YES, and CHARACTER, YES.
RB 7.7 Saying Simple Things

94. **Costa Rican Ornithology**
A fully-implemented book, answering questions from a table of data, and responding to failed consultation with a custom message such as "You flip through the Guide to Central American Birds, but find no reference to penguins."
RB 9.6 Reading Matter

§7.7 The other four senses

The five senses are all simulated with actions. Sight is so informative that it is handled by a whole range of actions: "looking", which describes the general scene; "examining something", which takes a closer look at a specific thing; "looking under something", and so on.

The other senses have one action each: "listening to something", "touching something", "tasting something" and "smelling something". It makes no sense to touch or taste the general scene, but listening and smelling are a different matter: we often just listen, without listening to anything specific. If the player types the command "listen", Inform understands that as listening to the current location: similarly for the bare command "smell". Thus:

```
Instead of listening to the Seashore, say "The song of gulls."

Instead of smelling the Cave, say "Salt and old seaweed."
```

Example

95. **The Art of Noise**
    Things are all assigned their own noise (or silence). Listening to the room in general reports on all the things that are currently audible.

RB 3.8 Sounds

§7.8 Rules applying to more than one action

A description can include more than one choice of action. For instance:

```
examining or searching the desk
```

matches either of "examining the desk" or "searching the desk". We can have more than two actions, of course:

```
examining, looking under or searching the desk
```

The actions combined like this need to be compatible with each other, at least a little. For instance, this will generate a problem message:

```
waiting or searching the desk
```

because it makes no sense to "wait the desk". On the other hand, this is fine:

```
waiting or searching
```

The general rule is that if we specify one or more objects ("the desk" in the above example), then each of the actions we quote must take at least that many objects.

For example, the following saves us writing the same basic rule three times over:

```
Instead of examining, looking under or searching the desk: say "There's no use poking
around in that old desk."
```

§7.9 All actions and exceptional actions

The special description "doing something" (or "doing anything") matches any action, and "doing something to …" also allows the noun to be specified.

For instance, the following puts its object out of bounds:

```
Instead of doing something to the cucumber sandwich, say "Lady Bracknell stares disapprovingly down her pince-nez at you, in a way which no amount of hunger or curiosity could overcome."
```

We sometimes need to be a little careful here: "waiting" qualifies as "doing something", but not as "doing something to something", because there is no object. "Putting the handbag on the cucumber sandwich" would also not qualify as "doing something to the cucumber sandwich" - only to the handbag.

More often, we would like to restrict the range of allowable actions to a select few. For instance:

```
Instead of doing something other than looking, examining or waiting: say "You must learn patience."
```

(Or we can write "except" instead of "other than".) Or we might have an object, too:

```
Instead of doing something other than examining, taking or dropping with the dagger: say "Don't fool around with that dagger. It's exceedingly sharp."
```

Note the "with", which is crucial here. Without it, the rule is subtly different:

```
Instead of doing something other than examining, taking or dropping the dagger: say "Don't fool around with that dagger. It's exceedingly sharp."
```

This second version matches if the action is, say, taking a shield, or even just looking, because that would be an action other than examining the dagger, taking the dagger or dropping the dagger.

Example

96. **Zodiac**
    Several variations on "doing something other than…", demonstrating different degrees of restriction.

RB 7.3 Reactive Characters

§7.10 The noun and the second noun

Once we begin applying rules to actions which are not entirely known in advance, we have a problem: there's no way to find out what specifically is happening. Consider the following:

```
Instead of examining something, say "It is none of your concern!"
```

This is fine as far as it goes, but clumsy. What if the player had examined a human being? Then "it" would be inappropriate. A better approach would be this:

```
Instead of examining something, say "[The noun] is none of your concern!"
```

The "noun" and, when necessary, the "second noun" are values which can be used in any rule about actions, and it follows that they can also be substituted into text, as this example demonstrates. Results might include:

```
Lady Bracknell is none of your concern!
The silver cigarette case is none of your concern!
```

This seems a good moment to mention that if you use "The" in a substitution, then a capitalised "The" will be used so long as this is grammatically correct (Lady Bracknell, as a proper noun, takes no article); "the" becomes a lower-case "the" along the same lines; and "a" a lower-case indefinite article.

```
Instead of examining something in the Drawing Room, say "Under Lady Bracknell's eye, you feel constrained. Besides, it is only [a noun]."
```

Example

97. **Ming Vase**
    ATTACK or DROP break and remove fragile items from play.
    RB 10.4 Glass and Other Damage-Prone Substances

§7.11 In rooms and regions

Three elaborations of action descriptions increase the range of possibilities further.

```
Instead of taking something in the Supernatural Void, say "In this peculiar mist you feel unable to grasp anything."
```

Like the objects to which the action applies, this location - the "in" clause - can take any description, not just an explicit place like "Supernatural Void":

```
Instead of listening in a dead end, say "You strain to hear further clues as to the course of the underground river, but to no avail."
```

But we often want a rule to apply in any of a set of rooms: and where, unlike the "dead end" example above, the rooms have nothing much in common except where they happen to lie on a map. For instance, we might want a rule to apply only inside a given building, or a garden consisting of five miscellaneous rooms. If so, we can create a "region" as a convenient way to refer to that group of rooms:

```
The Arboretum is east of the Botanical Gardens. Northwest of the Gardens is the Tropical Greenhouse.

The Public Area is a region. The Arboretum and Gardens are in the Public Area.

Instead of eating in the Public Area, say "The curators of the Gardens are ever among you, eagle-eyed and generally cussed."
```

§7.12 In the presence of, and when

Relative location can also be important: relative to other people, that is -

```
Instead of eating something in the presence of Lady Bracknell, say "Lady Bracknell disapproves thoroughly of gentlemen who snack between meals, and there are few disapprovals in this world quite so thorough as Lady Bracknell's."
```

As might be guessed, this applies when the action takes place in the same location as the person named: and of course that person can also be described more vaguely ("... in the presence of a woman", say), and can just as easily be an inanimate thing ("... in the presence of the radio set").

Lady Bracknell is a pushover compared to some matriarchs:

```
Instead of doing something other than looking, examining or waiting in the presence of the Queen: say "I'm afraid they take what you might call a zero tolerance approach to breaches of court etiquette here."; end the story saying "You have been summarily beheaded".
```

The last of the optional clauses we can tack on to the description of an action is the most general of all. We can add "when" and then any condition at all, as in:

```
Instead of eating something when the radio set is switched on, say "Something about the howling short-wave static puts you right off luncheon."
```

This supposes that the radio is so loud that it can be heard from any room: we could muffle it so that it's only audible from the room it is in like so:

```
Instead of eating something in the presence of the radio set when the radio set is switched on, say "Something about the howling short-wave static puts you right off luncheon."
```

Examples

98. **Beachfront**
An item that the player can't interact with until he has found it by searching the scenery.
RB 6.6 Looking Under and Hiding

99. **Today Tomorrow**
A few notes on "In the presence of" and how it interacts with concealed objects.
RB 8.3 Animals

§7.13 Going from, going to

Going is an action defined like any other: it is the one which happens when the player tries to go from one location to another. But it is unlike other actions because it happens in two locations, not just one, and has other complications such as vehicles and doors to contend with. To make it easier to write legible and flexible rules, "going" is allowed to be described in a number of special ways not open to other actions, as demonstrated by the following example story:

```
"Going Going"

The Catalogue Room is east of the Front Stacks. South of the Catalogue Room is the Musicology Section.

Instead of going nowhere from the Front Stacks, say "Bookcases obstruct almost all passages out of here."

Instead of going nowhere, say "You really can't wander around at random in the Library."

Before going to the Catalogue Room, say "You emerge back into the Catalogue Room."
```

Note that "going nowhere" means trying a map connection which is blank, and if no rules intervene then "You can't go that way" is normally printed. Unless "nowhere" is specified, descriptions of going apply only when there is a map connection. So "going from the Musicology Section" would not match if the player were trying to go east from there, since there is no map connection to the east. Similarly, "going somewhere" excludes blank connections.

The places gone "from" or "to" can be specific named regions instead of rooms. This is convenient when there are several different ways into or out of an area of map but a

common rule needs to apply to all: so, for example,

```
Before going from the Cultivated Land to the Wilderness, ...
Before going nowhere from the Wilderness, say "Tangled brush forces you back."
```

Note that it must be "going nowhere from the Wilderness", not "…in the Wilderness". (Note also the caveat that the regions must be named: "going from a region", or something similarly nonspecific, will not work.)

An important point about "going… from" is that, as mentioned in general terms above, it requires that there is actually a map connection that way: whereas "going… in" does not. Suppose there is no map connection north from the Wilderness. Then:

```
Instead of going north from the Wilderness, say "You'll never read this."
Instead of going north in the Wilderness, say "Oh, it's too cold."
```

The first of these never happens, because it is logically impossible to go north from the Wilderness: but the second does happen. (Technically, this is because "going north" is the action, and "in the Wilderness" a separate condition tacked onto the rule.) This distinction is often useful - it allows us to write rules which apply only to feasible movements.

This may be a good place to mention a small restriction on the ways we can specify an action for a rule to apply to, and how it can be overcome. The restriction is that the action should only involve constant quantities, so that the following does not work:

```
The Dome is a room. The Hutch is north of the Dome. The rabbit is in the Hutch. Before going to the location of the rabbit, say "You pick up a scent!"
```

because "the location of the rabbit" is a quantity which changes in play (the player can pick up the rabbit and take him to the Dome, for instance). However, we can get around this restriction by defining a suitable adjective, like so:

```
The Dome is a room. The Hutch is north of the Dome. The rabbit is in the Hutch.
Definition: a room is rabbit-infested if it is the location of the rabbit. Before going to a rabbit-infested room, say "You pick up a scent!"
```

Examples

100. **Veronica**
    An effect that occurs only when the player leaves a region entirely.
    RB 6.9 Going, Pushing Things in Directions

101. **A&E**
    Using regions to block access to an entire area when the player does not carry a pass, regardless of which entrance he uses.
    RB 3.2 Map

102. **Bumping into Walls**
    Offering the player a list of valid directions if he tries to go in a direction that leads nowhere.
    RB 6.9 Going, Pushing Things in Directions

103. **Polarity**
    A "go back" command that keeps track of the direction from which the player came, and sends him back.
    RB 6.9 Going, Pushing Things in Directions

§7.14 Going by, going through, going with

Adding to the previous example story, we apply rules which depend on travelling by a particular vehicle:

```
The book trolley is in the Musicology Section. "The book trolley, a sort of motorised tractor for trundling around through the stacks, is parked here." The trolley is a vehicle. Instead of going nowhere by the trolley, say "Don't go crashing the trolley into walls."

Instead of going to the Front Stacks by the trolley, say "The Front Stacks are far too confined for the trolley to manoeuvre into them."
```

And, lastly, rules which apply to movements through particular doors:

```
The green baize door is east of the Catalogue Room and west of the Clerk's Office. The green baize door is an open door.

Before going through the green baize door, say "Through you go..." After going through the green baize door: try looking; say "...and here you are."
```

(Note that these apply whether the action is "going east" or "entering the green baize door", each having the same effect.) The last rule is worth a second look: the normal way that a "going" action is reported is to produce the room description of the new location.

So if an "after" rule stops the action before we get to reporting, we have to produce any room description by hand (hence the "try looking" to cause the looking action). Alternatively, we could simply say something and let the normal course of events take place:

```
After going through the green baize door: say "...and here you are:"; continue the action.
```

Finally, going is an action which can also happen while the player is pushing something from one room to another, and we can describe this like so:

```
Instead of going from the Office with the trolley, say "But it looks perfectly placed here.
Why push any further?"
```

"Going" is not the only action which moves the player. Another is "exiting", an action which moves the player out of whatever he/she is currently in or on. This action is often caused by the player typing just OUT or GET DOWN, and there's no noun as such. But Inform allows the syntax "exiting from" to make it easier to write rules about the exiting of particular containers or supporters:

```
After exiting from the Mini Cooper:
    say "You painstakingly unpack your limbs from the tiny car."
```

Examples

104. **No Relation**
    A car which must be turned on before it can be driven, and can only go to roads.
    RB 8.1 Bicycles, Cars and Boats

105. **Mattress King**
    Adding extra phrasing to the action to PUSH something in a direction.
    RB 6.9 Going, Pushing Things in Directions

106. **One Short Plank**
    A plank bridge which breaks if the player is carrying something when he goes across it. Pushing anything over the bridge is forbidden outright.
    RB 3.5 Doors, Staircases, and Bridges

107. **Provenance Unknown**
    Allowing something like PUSH TELEVISION EAST to push the cart on which the television rests.
    RB 6.9 Going, Pushing Things in Directions

108. **Zorb**
    Replacing the message the player receives when attempting to push something that isn't pushable, and also to remove the restriction that objects cannot be pushed up or down.
    RB 6.9 Going, Pushing Things in Directions

§7.15 Kinds of action

Especially when people need to react to events going on around them, it is helpful to be able to categorise actions into whole areas of behaviour. For instance:

```
Kissing Mr Carr is unmaidenly behaviour.
Doing something to the painting is unmaidenly behaviour.

Instead of unmaidenly behaviour in the Inn, say "How unmaidenly!"
```

Here a new kind of action called "unmaidenly behaviour" has been created and then used in the description of an instead rule. The convenience of this approach is that when further actions suddenly occur to us as also being unmaidenly - say, attacking Mr Carr - we only need to add a single line:

```
Attacking Mr Carr is unmaidenly behaviour.
```

And this will automatically be reflected in any rules which concern the consequences of failing to be ladylike.

(Note that we were only allowed to say that "Kissing Mr Carr is unmaidenly behaviour." because Inform already knew from earlier sentences - see the example below - that Mr Carr was a person, and therefore that "kissing Mr Carr" made sense as a description of an action.)

Examples

109. **Dearth and the Maiden**
Our heroine, fallen among gentleman highwaymen, is restrained by her own modesty and seemliness.
RB 5.3 Characterization

110. **Mimicry**
People who must be greeted before conversation can begin.
RB 7.6 Getting Started with Conversation

§7.16 Repeated actions

We come at last to the final thing which can be specified about an action, and appropriately enough it must be specified with the final words of the description. This is the way to talk about repeated activity:

```
Instead of examining the tapestry for the third time, say "All right, so it's a masterpiece, but is this really the time to make a detailed study?"

Instead of examining the urn at least twice, say "It's an urn. What do you want from me?"

Instead of going nowhere for the 20th time, say "Do stop walking into walls, there's a good fellow."
```

Note that we are allowed to spell out numbers up to twelve in English words, but beyond that must use digits (thus "twelfth" is allowed but not "thirteenth": "13th" should be used instead). The following example is instructive:

```
Instead of taking something for the fourth time, say "No. I'm capricious."
```

This means that it is the fourth time a "taking…" action has been tried, and does not mean that the same item was taken each time. Also, note that we are counting the number of times the action has been tried, not the number of times it succeeded.

Examples

111. **Y ask Y?**
   Noticing when the player seems to be at a loss, and recommending the use of hints.
   RB 11.3 Helping and Hinting

112. **A Day For Fresh Sushi**
   A complete story by Emily Short, called "A Day for Fresh Sushi", rewritten using Inform 7. Noteworthy is the snarky commenter who remarks on everything the player does, but only the first time each action is performed.
   RB 7.3 Reactive Characters

§7.17 Actions on consecutive turns

We can also reckon the number of consecutive turns on which an action has been repeated, by talking about "turns" instead of "times", as demonstrated in the following example story. Note also that we are allowed to use the phrase "doing it" to mean "the same description as the previous one except for the part about turns or times", like so:

```
"Waiting Room"

The Antechamber is a room. The tattered copy of Women's Wear Daily is in the Antechamber. Instead of taking the Daily, say "It is stamped NOT TO BE TAKEN AWAY."

Instead of examining the Daily for the first time, say "The best article seems to be about how your star sign affects your best swimsuit colour. Really: that's the best article."

Instead of doing it for the second time, say "You now know a generous amount about a typical week in the life of a weather forecaster."

Instead of doing it for the third time, say "You would now know how to cook herb bread, except that you have already forgotten the names of both of the herbs."

Instead of doing it more than three times, say "Nope, you've drained it of all conceivable sustenance, even the small ads about French farmhouses to let (sleeps 7) and breast reduction surgery (with alarming photographs in sallow light)."

After waiting for four to six turns, say "This is getting mighty dull." After waiting for seven to eight turns, say "Really, exceptionally dull." After waiting for nine turns, end the story saying "You have died of boredom, something previously thought medically impossible".
```

Note once again that numbers above twelve must not be written out, so "more than twelve times" would be acceptable, but we would write "more than 13 times".

§7.18 Postscript on actions

In this chapter, all actions have been carried out by the player, all have been drawn from the standard stock of built-in actions ("unlocking", "taking", "going" and so forth), and all

of those built-in actions have been allowed to work in the standard way - we have seen how to prevent the taking of something, and how to give this unexpected consequences, but not how to make taking work in an entirely different way.

All three of those restrictions will later be lifted in the chapter on "Advanced Actions", but otherwise we have covered the ground thoroughly, and it is time to move on to the techniques enabling us to do more than make tart replies to the player: it is time to change the world.

8. Change

§8.1 Change of values that vary
§8.2 Changing the command prompt
§8.3 Changing the status line
§8.4 Change of either/or properties
§8.5 Change of properties with values
§8.6 Whose property?
§8.7 Moving things
§8.8 Moving backdrops
§8.9 Moving the player
§8.10 Removing things from play
§8.11 Now...
§8.12 Increasing and decreasing
§8.13 Checking on whereabouts
§8.14 More flexible descriptions of whereabouts
§8.15 Calling names
§8.16 Counting the number of things
§8.17 Looking at containment by hand
§8.18 Randomness
§8.19 Random choices of things

§8.1 Change of values that vary

So far, what we have done in response to the player's commands amounts to little more than a few ripostes. The simulated world does change during play, as the player moves from room to room or picks up things, but all of this is happening automatically, not at our direct instruction. How then can we make the world change?

Recall that the world consists of rooms, in which are things, and that all of these have properties appropriate to their kinds. Some properties are either/or ("open" or "closed" but not both and not neither), while others have values (the "matching key" of a lockable door, for instance). Finally, we may also have created some free-standing values or "variables".

We take the last example first, as it is the simplest. Suppose we have:

```
"Winds of Change"

The prevailing wind is a direction that varies. The prevailing wind is northwest.

The Blasted Heath is a room. "Merely an arena for the play of witches and kings, my dear, where the [prevailing wind] wind blows."

Instead of waiting when the prevailing wind is northwest:
    say "A fresh gust of wind bowls you over.";
    now the prevailing wind is east.
```

The new phrase here is "now". This automatically checks that the new value is one which makes sense in the given context, so for instance it would not allow either of these:

```
now the prevailing wind is 25;
now the prevailing wind is the Heath;
```

the former being a number, and the latter a room, so that neither is a direction. Similarly, "now" will not allow constant values to be changed. So

```
Colour is a kind of value. The colours are blue, red and mauve.

After pulling the psychedelic lever:
    now blue is mauve.
```

…will result in a problem message; it's like writing "now 1 is 2". The difference between "the prevailing wind" and "blue" is that the wind was declared to be a "direction that varies", whereas blue wasn't.

§8.2 Changing the command prompt

The command prompt is the text printed by Inform to ask the player for another command. Ordinarily this is simply a greater-than-sign, ">", so we tend not to notice it as text at all. Internally, though, it is a variable value called "command prompt", which means we can change it.

For example, this will be a more conversational sort of prompt:

```
When play begins: now the command prompt is "What now? ".
```

Whereas this will be more up-to-the-minute and demanding:

```
When play begins: now the command prompt is "[time of day] >".
```

("Time of day" is another variable value, which is fairly self-explanatory, but will be

covered in detail later on.) The prompt can be changed at any point, so can be used to indicate the current situation, or even as a sly way to introduce a sort of conversation between computer and player.

Example

113. **Don Pedro's Revenge**
    Combat scenario in which the player's footing and position changes from move to move, and the command prompt also changes to reflect that.
    RB 7.5 Combat and Death

§8.3 Changing the status line

The status line is the black bar along the top of a story being played, which ordinarily displays the current position; in a story with scoring, it also usually shows the score and number of moves taken. Like the command prompt, it is not fixed but results from values which can be changed: the "left hand status line" and "right hand status line".

The default values are "[the player's surroundings]" for the left hand status line and "[score]/[turn count]" for the right hand status line (if there's scoring; it's blank otherwise). Score and turn count are numbers which vary in play (more about scoring later); "[the player's surroundings]" is a text substitution really intended for just this purpose:

say "[the/-- player's surroundings]"

This text substitution produces a succinct description of where the player is, be this in darkness, in a lighted room or inside an opaque container such as a large packing case. Example:

```
now the left hand status line is "You: [the player's surroundings]";
```

These make useful elements to juggle in redesigning the status line, as in the following example:

```
When play begins:
    now the left hand status line is
        "[the player's surroundings] / [turn count] / [score]";
    now the right hand status line is "Time: [time of day]".
```

The text in the right hand status line should be kept no more than 14 letters long,

including any spaces. The left hand status line has more leeway, but should still be kept brief.

See Also

Awarding points for scoring.

Examples

114. **Politics as Usual**
   Have the status line indicate the current region of the map.
   RB 12.2 The Status Line

115. **Centered**
   Replacing the two-part status line with one that centers only the room name at the top of the screen.
   RB 12.2 The Status Line

§8.4 Change of either/or properties

When we have an either/or property, we can set it like so:

```
Instead of waiting when the oaken door is closed:
    say "There is a slow, creaky click! sort of noise as the door swings open, apparently all by itself.";
    now the oaken door is open.
```

If it is open already, nothing changes: in any case nothing is said to the player unless we give explicit instructions to that effect, as we've done here.

Inform protects its model world from accidental damage in several ways, one of which is to ensure that things are not given properties which they are not allowed to have. So this, for instance, will not be accepted:

```
now the oaken door is unvisited
```

More subtle problems arise if it is not possible to tell, when the story is being constructed, what the object in question will be: for instance, if we try to change a randomly chosen object to be "unvisited". Inform therefore makes additional checks during play, printing up a suitable message only if the rules are violated. The net effect is that it is impossible for the oaken door ever to have the "unvisited" property.

Example

116. **Vitrine**
    An electrochromic window that becomes transparent or opaque depending on whether it is currently turned on.

RB 3.6 Windows

§8.5 Change of properties with values

Changing properties with values is very similar:

```
now the printed name of the Closet is "Suddenly Spooky Closet"
```

Inform checks three different things to ensure that this change is safe to perform. Firstly, the value must be the right kind for the property in question, so this for instance would be rejected:

```
now the printed name of the Closet is 7
```

Secondly, the object in question has to be allowed to have the given property. This, for instance, would be disallowed:

```
now the initial appearance of the Closet is "Dusty"
```

(since "initial appearance" is a property which only things can have, not rooms). Finally, the object has to actually have the property, not just have the right to have that property. Thus:

```
now the printed name of the Closet is "Suddenly Spooky Closet"
```

…is only permitted if the Closet is designed with a "printed name". In fact this is certain to be true: all rooms and things automatically have a printed name, which is the short boldface description in the case of rooms, and the usual text briefly describing something in the case of things.

"Now" is a simple way to change many things in Inform, but it's cumbersome to change the map of the model world using "now", because the map is such a complicated arrangement. (It's not a property: it's a sort of mesh of relations.) So a special phrase exists to change map connections:

change (direction) **exit of** (room) **to** (room)

This phrase alters the map so that the given map connection is made. Note that connections can be made to rooms, but not doors: the positions of doors are fixed. Example:

```
change the east exit of the Closet to the Tsar's Imperial Dining Salon
```

Since "nothing" is not a room, this doesn't allow us to change the exit to nothing, so there is a separate definition of:

```
change the west exit of the Closet to nothing
```

change (direction) **exit of** (room) **to nothing/nowhere**

This phrase alters the map so that the given map connection is unmade. Example:

```
change the west exit of the Closet to nowhere
```

Altering the map itself is not a very subtle way to adjust when and where the player can move - writing suitable rules is usually a cleaner solution - so this phrase is best avoided unless really needed.

Examples

117. Thirst
A waterskin that is depleted as the player drinks from it.
RB 10.2 Liquids

118. Thirst 2
A campfire added to the camp site, which can be lit using tinder.
RB 10.8 Fire

§8.6 Whose property?

This seems a useful point to clarify something already seen. We normally call a property with a value something like:

```
the printed name of the West Ballroom
```

We are sometimes allowed to omit the "of the …" part, and simply call it "the printed name", for the sake of brevity. For instance, the following room description:

```
The West Ballroom is a room. "A handsome sweep of chequered floor beckons the eye
into the [printed name]."
```

will result in "West Ballroom" being substituted for "[printed name]". Since the text belongs to the West Ballroom, that is assumed to be the owner of any properties named in its description. Similarly:

```
Instead of examining something, say "Hmm, let me see: [printed name]..."
```

Here the owner of the "printed name" is assumed to be the noun referred to in the action - in other words, the "something" alluded to in the rule.

§8.7 Moving things

We have now seen how to change the properties of rooms and things, and also any freestanding values which may have a bearing on the model world. We are not allowed to change the kind of anything during play. Our remaining freedom is to move things around. It would make no sense to move rooms around, because rooms are the fixed reference points in our geography, but anything else is mobile. This even includes things which are supposedly "fixed in place", for unlike the player, we have god-like powers. (There are minor restrictions: backdrops are trickier to move, since they are present in several rooms at once - see the next section. And doors, at the junction between two rooms, cannot be moved.)

Here is how to move something:

move (object) **to** (object)

This phrase moves the first-named object to the second. Example:

```
move the genie's lamp to Aladdin's Cave;
```

The first object named has to be a thing; the destination must be a room, as here, a container, a supporter, or a person. When something is moved, all its parts and contents (and all their contents, and so on) move with it. If the thing being moved is a person, then the destination is required to be a room or an enterable container. (In particular, a person cannot be carried by another person.)

Two options can be used if the object being moved is the player.

```
move the player to Aladdin's Cave, without printing a room description
```

omits the description which would otherwise be produced. A compromise is to use:

```
move the player to Aladdin's Cave, printing an abbreviated room description
```

which gives a full description if the player has never been here before, but only a brief one if it is a familiar scene. These options have no effect for any other objects being moved.

If the destination is a person, like so:

```
move the genie's turban to Aladdin;
```

then it will be carried rather than worn. We could arrange for it to be worn instead by writing

```
now the genie's turban is worn by Aladdin;
```

"Now…" is a much more flexible phrase than "move": more on this shortly.

Example

119. **Meteoric I and II**
    A meteor in the night sky which is visible from many rooms, so needs to be a backdrop, but which does not appear until 11:31 PM.

RB 4.4 Scene Changes

§8.8 Moving backdrops

A backdrop can be in several rooms at once. When created, its position can be given as any specific collection of rooms, or as a region, or even as "everywhere". For instance:

```
The Upper Cave is above the Rock Pool. The Ledge is east of the Pool.

The stream is a backdrop. It is in the Upper Cave and the Ledge.
```

Moving backdrops is not like moving other things, because there's no single destination. There are several possibilities:

(a) A backdrop can be moved to a region. If we define:

```
Lower Level is a region. The Rock Pool and the Ledge are in the Lower Level.
```

then we can write either of

```
move the stream to the Lower Level;
now the stream is in the Lower Level;
```

and either way, the stream is now found in the Rock Pool and the Ledge but nowhere else.

(b) A backdrop can be moved to a category of rooms:

move (object) **backdrop to all** (description of objects)

This phrase moves the backdrop so that it is now present in every room matching the given description. Example: If we define

```
A room can be wet or dry. A room is usually dry. The Rock Pool is wet.
```

then we can write

```
move the stream backdrop to all wet rooms;
```

This phrasing, "move the … backdrop to all …" is deliberately meant to look unlike the simpler "move … to …", to emphasise that this kind of movement is possible only for backdrops.

What then happens is that the stream is present in whichever rooms are currently wet. But the stream's position is ordinarily checked only after movements, for efficiency's sake. So if the player is in a room which suddenly changes from being dry to being wet, the stream will not magically appear (though it will be there if the player goes out and comes in again). If this is not good enough, the phrase "update backdrop positions" can be used to ensure the accuracy of all backdrop locations after a dramatic change:

update backdrop positions

This phrase runs through all backdrops in the model world and makes sure they are correctly in, or not in, the current location, so that everything appears right from the player's point of view. Example:

```
The Upper Cave is above the Rock Pool. The Ledge is east of the Pool. The stream is a backdrop.

When play begins:
    move the stream backdrop to all wet rooms.

A lever is in the Cave. The lever is fixed in place.

Instead of pulling the lever when the Cave is dry:
    now the Cave is wet;
    now the lever is in the Rock Pool;
    now the lever is portable;
    update backdrop positions;
    say "The old rusty lever pulls away, and the thin cave wall goes with it, so that a stream bursts into the cave, falling to the pool below."
```

(c) A backdrop can be moved to be either everywhere or nowhere:

```
After sleeping:
    say "It's a bright new day!";
    now the stars are nowhere.

After waiting:
    say "Darkness falls rapidly here.";
    now the stars are everywhere.
```

Example

120. **Orange Cones**
Creating a traffic backdrop that appears in all road rooms except the one in which the player has laid down orange cones.
RB 3.9 Passers-By, Weather and Astronomical Events

§8.9 Moving the player

The player is a thing, too, and can also be moved, which has the effect of instantaneous transportation, without the need for a suitable map connection to the new location. For instance, these are equivalent:

```
move the player to the Bodleian Library;
now the player is in the Bodleian Library;
```

This will ordinarily result in a room description of the Bodleian Library being printed up, but that might not always be desirable. For instance:

```
Instead of waiting in the Schola Maleficorum:
	say "A bored demon catches your eye (they really do have very inquisitive fingers) and throws you back out into the Antechamber.";
	move the player to the Antechamber, without printing a room description.
```

Thus tacking on the option "without printing a room description", remembering to add the comma, omits the description which would otherwise be produced. A compromise is to use the option "printing an abbreviated room description": this gives a full description if the player has never been here before, but only a brief one if it is a familiar scene.

The player's point of view can also be moved by shifting to another character. Suppose the story features two people, Alice and Bob, and the player at the keyboard is giving commands to Alice, and seeing everything from her point of view. The phrase:

```
now the player is Bob
```

switches the perspective so that now Bob is the one controlled by the human player, and it's Bob's point of view which counts. The human being at the keyboard may feel a sense of having jumped abruptly from place to place, but in fact neither Alice nor Bob has moved.

A change of player can sometimes cause confusing things to happen, if it takes place as part of a successful action. Suppose there's an action called "possessing", which enables the player to possess somebody else's body; and suppose the player types POSSESS ADELE. The action succeeds, so that the player moves into the mind of Adele. But that means that at the end of the action, the player is no longer the actor - that is, no longer the person who began the action; and consequently, Inform won't use the report rulebook to say what has just happened. It's a strange business, moving into another body.

Example

121. **Terror of the Sierra Madre**
    Multiple player characters who take turns controlling the action.

RB 5.6 Viewpoint

§8.10 Removing things from play

Some things will occasionally be in a limbo state called being "off-stage": like actors or props not needed in Act II, but perhaps to be brought back on-stage later, they wait on the sidelines. Anything created with no apparent location will start the story off-stage, as in the case of the lamp here:

```
Aladdin's Cave is a room. The genie's lamp is a container.
```

(Such things are easy to see in the World index because they are listed after all of the rooms and their contents, not belonging inside any room.) If we wanted to make this clearer to a human reader, we could add:

```
The lamp is nowhere.
```

to emphasise the point. In this context, "nowhere" means "in no room". Moving the lamp onto the stage-set, so to speak, is easy:

```
now the lamp is in the Cave;
```

or perhaps:

```
now the player is carrying the lamp;
```

and we can whisk it away again like so:

```
now the lamp is nowhere;
```

(We can't say "now the lamp is somewhere" because that's too vague about exactly where it is.) In older builds of Inform, the usual thing was to write "remove the lamp from play", but that's now a deprecated phrase: better to use "nowhere" instead.

remove (object) **from play**

Removes the given object from play, so that it is not present in any room. We are not permitted to remove rooms, or doors, or the player, from play; but we are permitted to remove backdrops, making them disappear from all rooms in which they are present. Example:

```
remove the gold coin from play;
```

We can test whether something is on-stage or off-stage with:

```
if the gold coin is somewhere, ...
if the gold coin is nowhere, ...
```

Inform also understands two adjectives for this:

```
if the gold coin is on-stage, ...
if the gold coin is off-stage, ...
```

Because these are adjectives, they can be used in a few ways which "nowhere" and "somewhere" can't, such as:

```
say "Ah, so many absent friends. Who now remembers [list of off-stage people]?"
```

Note that "on-stage" and "off-stage" apply only to things. Rooms, directions and regions are the stage itself: so it makes no sense to ask the question of whether they are "on-" or "off-". Doors are always on-stage; a backdrop, say "the sky", is always on-stage unless it has been taken off by writing something like "now the sky is nowhere".

Examples

122. Beverage Service
A potion that the player can drink.
RB 10.2 Liquids

123. Spring Cleaning
A character who sulks over objects that the player has broken (and which are now off-stage).
RB 10.4 Glass and Other Damage-Prone Substances

124. Extra Supplies
A supply of red pens from which the player can take another pen only if he doesn't already have one somewhere in the game world.
RB 10.3 Dispensers and Supplies of Small Objects

§8.11 Now...

"Now" has already appeared several times in this chapter, being used like a Swiss army knife to change values of all kinds:

```
now the score is 100;
```

In fact, "now" is by far the most flexible phrase known to Inform.

now (a condition)

This phrase makes the condition become true. Examples:

```
now the score is 100;
now the player is Kevin;
now the front door is open;
now Mr Darcy is wearing the top hat;
now all the doors are open;
now all of the things in the sack are in the box;
```

Inform issues a problem message if the condition asks to do the impossible ("now 3 is an even number") or is vague ("now the duck is not in the Lily Pond") or not in the present tense ("now the front door had been open").

We've now seen all three things which can be done with a condition S which describes the world:

```
S. - The relation holds at the start of play.
if S, ...; - Does the relation hold right now?
now S; - Make the relation hold from now on.
```

For instance,

```
The apple is in the basket.
if the apple is in the basket, ...;
now the apple is in the basket;
```

Examples

125. **Bee Chambers**
   A maze with directions between rooms randomized at the start of play.
   RB 3.2 Map

126. **Hatless**
   It's tempting to use "now…" to distribute items randomly at the start of play, but we need to be a little cautious about how we do that.
   RB 11.1 Start-Up Features

127. **Technological Terror**
   A ray gun which destroys objects, leaving their component parts behind.
   RB 7.5 Combat and Death

§8.12 Increasing and decreasing

Once we begin to deal with named values (or table entries, list entries or other ways to describe places where values are kept), we find that we often want to change them. We could if we wanted always use "now" for this, but it can be a little clumsily worded if we want to increase or decrease something:

```
now the score is the score plus six;
```

Because of that, we have some convenient abbreviations which have the advantage that the value being changed only has to be named once:

increase (a stored value) **by** (value)

This phrase increases the variable, table entry, list entry, or property by the given amount, which must be of a compatible kind. Example:

```
increase the score by 8;
increase the time of day by 5 minutes;
```

decrease (a stored value) **by** (value)

This phrase decreases the variable, table entry, list entry, or property by the given amount, which must be of a compatible kind. Example:

```
decrease the score by 6;
decrease the carrying capacity of the player by 10;
```

An even greater abbreviation can be made when we are changing a number by 1 either way:

increment (a stored value)

This phrase increases the variable, table entry, list entry, or property by 1. Example:

```
increment the score;
```

decrement (a stored value)

This phrase decreases the variable, table entry, list entry, or property by 1. Example:

```
decrement the score;
```

"Increment" and "decrement" are traditional computing terms, though they have been used in engineering for at least a century and in finance for longer still.

§8.13 Checking on whereabouts

We have seen that while rooms are fixed, their contents move around, so we will need ways to examine the current whereabouts of things. The following examples show the kind of conditions allowed:

```
if the genie's lamp is in Aladdin's Cave ...
if Aladdin is not in Aladdin's Cave ...
if Aladdin's Cave contains the genie's lamp ...
if the genie's lamp is carried by Aladdin ...
if Aladdin is carrying the genie's lamp ...
if Aladdin does not have the genie's lamp ...
if the table supports the genie's lamp ...
if the table is supporting the genie's lamp ...
if the genie's lamp is supported by the table ...
if the genie's lamp is on the table ...
if the genie's lamp is on top of the table ...
if the genie's lamp is in the cupboard ...
if the genie's lamp is contained in the cupboard ...
if the genie's lamp is inside the cupboard ...
if the genie's lamp is within the cupboard ...
if the wick is part of the genie's lamp ...
```

These are exactly like the assertions which we use to set up the world, except that we make them questions by placing "if" in front. But we shall later see that we can also use three other tenses, not to mention plural forms, so that new verbal forms like "had not been inside" and "were not supported by" are legal here (which they would not be in assertions). What we are not allowed is to contract these verbs with apostrophes: "isn't", "hasn't" and "hadn't" are forbidden.

Overwhelmingly the condition we check most is whether the player is carrying something. The following are therefore equivalent:

```
if the genie's lamp is carried by the player ...
if the genie's lamp is carried ...
```

And similarly for "not carried", "worn" and "not worn". To be precise, if a form of *to be carried* or *to be worn* is not followed by any other description, then "the player" is assumed to be doing the carrying or wearing.

§8.14 More flexible descriptions of whereabouts

The examples just given were all basically of the form "X *relation* Y" where X and Y were specific names of things. For example,

```
if the genie's lamp is carried by Cinderella ...
if the genie's lamp is inside the cupboard ...
```

Just as actions could be described with patterns to be matched ("taking an open

container", say), so can the positions of things. Giving subtler descriptions of our X and Y sometimes broadens the possibilities, sometimes narrows them:

```
if the genie's lamp is carried by a woman ...
if the genie's lamp is inside the closed cupboard ...
```

In the first case, Y is allowed to be one of a whole range of things - any of the women existing in the world. This makes for a broader condition. In the second case, Y has not only to be the cupboard, but at a time when it is closed: which makes for a narrower condition. We can, of course, also vary X:

```
if an animal is inside the cupboard ...
if a container is carried ...
```

And we can even vary both X and Y at once:

```
if a woman is holding an animal ...
```

a condition which will be true if, anywhere in the story's world, any woman is holding any animal.

§8.15 Calling names

Conditions like "if somebody is in an adjacent room" allow complicated tests to be performed with a minimum of fuss, but it's rare that we want to know only whether they are true: more likely we also want to know *which* person, and *which* room.

For this purpose, we are allowed to supply a name for any such vaguely-described object which comes up, and then to use that name thereafter.

```
if somebody is in an adjacent room (called the Hiding Place), say "You hear distant
breathing from [the Hiding Place]."
```

We can even name more than one of the things discovered:

```
Instead of waiting when a woman (called the kidnapper) is holding an animal (called the
pet), say "How can you think of rest when, somewhere out there, [pet] has been cruelly
kidnapped by [the kidnapper]?"
```

Note the brackets, which are essential. The result of typing "wait" is then

```
How can you think of rest when, somewhere out there, a lapdog has been cruelly
kidnapped by Baroness Orczy?
```

Of course, that might be just one of many animals held by women in the story. We shall later see ways to go through all of the possibilities found, performing some action with each in turn.

A calling, if we can use that word, should be made immediately after the noun it refers to, and not left to hang back after any relative clauses. For instance,

```
if something (called the penitential object) held by the player is hot
```

is allowed, but not

```
if something held by the player (called the penitential object) is hot
```

because there is too much potential ambiguity - are we trying to call the player something?

See Also

Repeat running through for systematically working on everything matching a description.

Example

128. **Higher Calling**
All doors in the game automatically attempt to open if the player approaches them when they are closed.
RB 3.5 Doors, Staircases, and Bridges

§8.16 Counting the number of things

It is very often useful to know how many things are in a given situation, and for this purpose we have the "number of ..." construction. For instance:

```
the number of edible things carried
the number of things on the table
the number of people in the Dining Room
```

Whereas "a woman is holding an animal" makes the same test as "an animal is held by a woman", getting the same result, counting is not so even-handed:

```
the number of women holding animals
the number of animals held by women
```

are different questions and, unless the ration is strictly one lapdog per baroness, will have different answers. If Cruella de Vil has 101 dalmatians, they may be very different indeed.

It can also be helpful to count things with no particular location, like so:

```
the number of rooms
the number of closed doors
```

For instance:

```
When play begins:
    now the right hand status line is "Explored: [number of visited rooms]/[number of rooms]".
```

Provided that the possible range is finite, we can also use "number of" to count values which match a description. For instance:

```
the number of non-recurring scenes
```

or if we were to define

```
Colour is a kind of value. The colours are red, orange, yellow, green, blue, indigo and violet.
```

then "the number of colours" would evaluate to 7. As with other ways of talking about whole ranges of values, this only works if the range is manageable. "The number of numbers" cannot sensibly be worked out: there are infinitely many, for all practical purposes, and similarly for "the number of texts".

§8.17 Looking at containment by hand

The descriptions outlined in the last few sections are intended to deal with almost all of the routine questions we might have about what currently resides where. It should be a last resort to use the following more primitive way to inspect the world.

holder of (object) ⇒ *object*

This phrase produces the container, supporter, carrier, wearer or room in which the object resides.

It's sometimes useful to go the other way. When something has possessions, we can find them out one at a time by running through a list.

first thing held by (object) ⇒ *object*

This phrase produces the first of the list of things held by the object. Example:

```
first thing held by Baroness Orczy
```

next thing held after (object) ⇒ *object*

This phrase produces the next item of the list of things held by something. Example: suppose Baroness Orczy is carrying a lapdog and a string of pearls.

```
next thing held after the lapdog
```

is then the string of pearls.

§8.18 Randomness

Sometimes we want to introduce random behaviour into play. We usually do this by generating random values, and then acting differently depending on what they are. The following:

```
a random number from 2 to 5
```

produces, as it suggests, a random number drawn from the choices 2, 3, 4 or 5, each of which is equally likely to come up. In fact, this isn't limited to numbers:

a random (name of kind) **between** (arithmetic value) **and** (arithmetic value) ⇒ *value*

or…

a random (name of kind) **from** (arithmetic value) **to** (arithmetic value) ⇒ *value*

or…

a random (name of kind) **between** (enumerated value) **and** (enumerated value) ⇒ *value*

or…

a random (name of kind) **from** (enumerated value) **to** (enumerated value) ⇒ *value*

This phrase produces a uniformly random value in the range given. Examples:

```
a random number from 10 to 99
a random time from 2:31 PM to 2:57 PM
```

If we make a new kind of value:

```
A cloud pattern is a kind of value. The cloud patterns are cumulus, altocumulus,
cumulonimbus, stratus, cirrus, nimbus, nimbostratus.
```

then we can also take random values from it:

```
a random cloud pattern between stratus and nimbus
```

which has three possible outcomes, all equally likely.

We can also use random conditions:

if a random chance of (number) **in** (number) **succeeds:**

This condition is true X/Yths of the time, where X and Y are the numbers. Example:

```
if a random chance of 2 in 3 succeeds, ...
```

Here is a rule which applies only 15% of the time:

```
Instead of waiting when a random chance of 15 in 100 succeeds: ...
```

Testing IF which makes random choices can be rather frustrating, because a problem showing up on one attempt may not show up on another. We can get around this by making use of the fact that computers do not actually generate true randomness, but instead make a sequence of apparently random numbers by applying a complicated formula to each one in order to make the next. The starting point is a number called the "seed", because the next choice grows out of it.

seed the random-number generator with (number)

This phrase changes the seed number as specified. Any random numbers generated after that depend only on the seed. Example: the following sentence will "fix" the process of generating these random numbers so that they are not random at all - the same sequence of random numbers will be produced on each run.

```
When play begins, seed the random-number generator with 1234.
```

The seed value "1234" can be anything positive; a different sequence of random numbers will be produced for each different seed value. A seed value of 0 restores the RNG to properly random behaviour again.

Alternatively, it's possible the "fix" the RNG by clicking the "Make random outcomes predictable when testing" option on the Settings panel. This makes the behaviour predictable whenever the story is played within Inform, but (unlike the rule above) has no effect on the story file once released.

Examples

129. **Do Pass Go**
    A pair of dice which can be rolled, and are described with their current total when not carried, and have individual scores when examined.
    RB 9.5 Dice and Playing Cards

130. **Lanista 1**
    Very simple randomized combat in which characters hit one another for a randomized amount of damage.
    RB 7.5 Combat and Death

131. **Weathering**
    The automatic weather station atop Mt. Pisgah shows randomly fluctuating temperature, pressure and cloud cover.
    RB 3.9 Passers-By, Weather and Astronomical Events

132. **Uptown Girls**
    A stream of random pedestrians who go by the player.
    RB 3.9 Passers-By, Weather and Astronomical Events

§8.19 Random choices of things

Writing "a random number" is not allowed, because the possible range is too large, but that was the only reason why not.

a/-- random (description of values) ⇒ *value*

This phrase makes a uniformly random choice from values satisfying the description given. Example:

```
a random visited room
a random scene
```

A problem message is issued if the range is too large (for instance, "a random text"). Unexpected results may follow if no value fits the description, unless we are describing objects, in which case the result is the special value "nothing".

For instance:

```
say "You can see [number of adjacent rooms] way[s] from here; how about [random adjacent room]?"
```

But it's important to worry about the possibility that nothing qualifies - here, that no adjacent rooms exist. The above would then say:

```
You can see 0 ways from here; how about nothing?
```

Examples

133. Candy
One of several identical candies chosen at the start of play to be poisonous.
RB 9.1 Food

134. Zork II
A "Carousel Room", as in Zork II, where moving in any direction from the room leads (at random) to one of the eight rooms nearby.
RB 3.2 Map

9. Time

§9.1 When play begins
§9.2 Awarding points
§9.3 Introducing tables: rankings
§9.4 When play ends
§9.5 Every turn
§9.6 The time of day
§9.7 Telling the time
§9.8 Approximate times, lengths of time
§9.9 Comparing and shifting times
§9.10 Calculating times
§9.11 Future events
§9.12 Actions as conditions
§9.13 The past and perfect tenses
§9.14 How many times?
§9.15 How many turns?

§9.1 When play begins

With the material from previous chapters, we are now able to set up a simulated world, to respond to the player's actions within it, and to make it change in perhaps unexpected ways.

The resulting experience can be as lively as we want to make it, but so far we have no very good ways to give it any structure: a beginning and an end, for instance, or a sense of plot. This next chapter is all about the passage of time, and it begins at the beginning.

We have already seen an example of how to write a rule which applies just once, at the start of the story:

```
When play begins: say "Welcome to Old Marston Grange, a country house cut off by fog."
```

The "when play begins" rules are checked only at the start of a story, not when a saved session is restored from disc. What happens is that these rules are followed, then the story's banner is printed, then the initial room description is printed up, and then the player is asked for a first command.

Example

135. **Clueless**
   A murderer for the mystery is selected randomly at the beginning of the game.
   RB 7.1 Getting Acquainted

§9.2 Awarding points

Traditionally-written stories award points throughout play, as an indication of progress. If we want to be traditional, we must first write:

```
Use scoring.
```

Without this, the SCORE, NOTIFY ON and NOTIFY OFF commands do not work; the final score is not shown at the end of a story; and the status line above the player's text area shows only the turn count, not (as is more usual) both the score and the turn count. Changing the "score" (see below) has no visible effect, though it is not actually illegal.

With "Use scoring" in place, we can award points as follows:

```
increase the score by 5;
```

substituting whatever number we feel is appropriate. We should be careful not to give out the same points over and over, that is, not to reward the same basic achievement many times over if the player simply repeats the same action. This, for instance, is open to abuse:

```
After taking the trophy:
    increase the score by 5;
    say "Well done!"
```

The player may simply take the trophy, drop it again, take it again, … and win five points every time around. We can prevent this by phrasing the rule more carefully:

```
After taking the trophy when the trophy was not handled:
    increase the score by 5;
    say "Well done!"
```

("Was handled", not "is handled", because this rule happens after the trophy has been taken - so by the time this rule has been considered, the trophy is always handled.) Rather than being an open-ended scoring system, IF normally has a maximum possible score, which can be specified with a sentence like so:

```
The maximum score is 10.
```

The score and maximum score are just numbers that vary, so we can freely change them:

```
After eating the poisoned mushroom:
    now the score is -100.
```

Examples

136. **Mutt's Adventure**
Awarding points for visiting a room for the first time.
RB 11.4 Scoring

137. **No Place Like Home**
Recording a whole table of scores for specific treasures.
RB 11.4 Scoring

§9.3 Introducing tables: rankings

Another tradition of interactive fiction is that the player has a current 'rank' according to how far his or her score has got. We can (but need not) choose to provide such rankings, and should do so by specifying a table like this:

Table 1 - Rankings

Score	Rank
0	"Beginner"
25	"Amateur Adventurer"
50	"Novice Adventurer"
100	"Junior Adventurer"
200	"Adventurer"
300	"Master"
330	"Wizard"
350	"Master Adventurer"

Typographically, tables in Inform look as much as possible like those found in non-fiction books: they can have many columns, so this is only a simple example (drawn from the actual rankings used by Infocom's *Zork I*, 1979). Each line in the source represents one row in the table, and the entries on a line must be separated by at least one tab character. (An entry might of course have several words with spaces in between, so a space is not

enough to separate entries: this is the only context when Inform distinguishes between spaces and tabs.) The table must occupy a single whole paragraph, with no skipped lines or missing entries. We are free to use extra tabs to indent it if we like.

Ordinarily we must explicitly ask to use the information presented in a table, but the table of rankings is a special case: Inform uses it automatically, if it is provided at all. To be recognised it must have the name "Rankings" and must have two columns named and arranged as shown. The scores should be given in ascending order. Customarily, the score in the final row should be the maximum possible achieved in the story - so that only a player with maximum possible score can be awarded the final ranking - and the value of "maximum score" is automatically set to this bottom-row value if it has not been explicitly set anywhere else in the source text.

§9.4 When play ends

Short of something like a power cut, the story can only end when one of the two participants chooses to end it: either the player, by deciding that enough is enough, or us.

In story-telling, there are many kinds of ending: happy, sad, decisive, bittersweet, surprise. Inform doesn't try to interfere in that kind of artistic choice, but it does need to know one thing about the ending: is it final, or not? Many authors like to make additional menu items available if the player has completed the story right through, but not if she has reached an early or wrong ending. (See the activity "amusing a victorious player", for example.)

end the story

This phrase ends the story at the next opportunity (typically as soon as the current rule ends), with the closing message "The End." The end is not considered final.

end the story finally

This phrase ends the story at the next opportunity (typically as soon as the current rule ends), with the closing message "The End." The end is considered final, and any hidden menu options will be revealed.

end the story saying (text)

This phrase ends the story at the next opportunity (typically as soon as the current rule ends), with the closing message given in the text. The end is not considered final. Example:

```
end the story saying "You have been stymied"
```

end the story finally saying (text)

This phrase ends the story at the next opportunity (typically as soon as the current rule ends), with the closing message given in the text. The end is considered final, and any hidden menu options will be revealed. Example:

```
end the story finally saying "You have defeated Sauron"
```

The closing message is traditionally set out in asterisks:

```
*** The End ***
```

This style is traditional in IF, and goes back to 1980 if not earlier.

We can test the current state like so:

if story has ended:

This condition is true if an end has been declared using one of the "end the story..." phrases.

if story has not ended:

This condition is true if no end has been declared using one of the "end the story..." phrases.

if story has ended finally:

This condition is true if an end has been declared using one of the "end the story finally…" phrases, so that an ending has been reached which the author feels is a completion of the player's experience.

if story has not ended finally:

This condition is true if an end has been declared using one of the "end the story…" phrases, but not "finally", so the author feels that the player can get further experience by playing again and trying different approaches.

The rulebook "when play ends" is the matching bookend to "when play begins". It is followed when the story decides to end (not when the player simply gives up and quits), and before any epitaph like

```
*** You have been poisoned ***
```

appears. For example:

```
When play ends, say "Oh dear."
```

Surprisingly, the end is not always the end:

resume the story

This phrase causes an ended story to resume exactly as if no "end the story…" phrase had been used. Example:

```
When play ends:
    if the story has not ended finally:
        say "Oh dear. Still, here's another chance.";
        resume the story.
```

The phrase is likely to be sensible only as part of a "when play ends" rule. Other traditional uses include giving the player three lives, as in an old-school arcade machine.

Example

138. **Big Sky Country**
     Allowing the player to continue play after a fatal accident, but penalizing him by scattering his possessions around the game map.

RB 11.6 Ending The Story

§9.5 Every turn

The passage of time in interactive fiction is broken up into a succession of turns, in each of which the player types a request and is given a response. Usually each such request triggers one action, but sometimes a whole sequence are fired off, as when the player types "get all" in a cluttered room.

As we've seen, the variable "turn count" holds the number of turns of play so far. By convention turn number 0 is the time when Inform prints up the banner and any initial text; it becomes turn number 1 when the player's first command is typed.

One of the last things to happen in each turn is that Inform will apply any rules which have been set to occur "every turn", like so:

```
Every turn, say "The summer breeze shakes the apple-blossom."
```

This is equivalent to writing:

```
An every turn rule: say "The summer breeze shakes the apple-blossom."
```

Note that the text about blossom, which will quickly become tiresome, is said at the end of every turn, not at the beginning, and in particular not before the player's first opportunity to type a command.

As usual when defining rules, we can add stipulations: any condition can be attached using "when".

```
Every turn when the location is the Orchard, say "The summer breeze shakes the apple-blossom."

Every turn when the player can see the rotting fish, say "Your nose twitches involuntarily."
```

Examples

139. **Witnessed 1**
   A kind of battery which can be put into different devices, and which will lose power after extended use.
   RB 10.7 Electricity and Magnetism

140. **Text Foosball**
   A game of foosball which relies heavily on every-turn rules.
   RB 7.2 Liveliness

§9.6 The time of day

Inform keeps track of the time of day automatically: play ordinarily begins at 9 AM and each turn takes one minute. In some works of interactive fiction, time of day is crucial: in others, it is irrelevant or even, by a sort of tacit convention, non-existent. So Inform does nothing to act upon the time, or to reveal it to the player, without instruction. Nevertheless it is there.

A sentence like the following allows the initial time to be set up as something other than 9 AM:

```
The time of day is 3:13 PM.
```

Here, "3:13 PM" is a constant value of a kind not seen before: it's a kind of value called "time", and the value "time of day" is a time that varies. After one turn it will be 3:14 PM, then 3:15 PM and so on.

Note that the sentence above is an assertion (a statement about the initial state of affairs), not an instruction which can be part of a rule. It would be equivalent to write:

```
When play begins: now the time of day is 3:13 PM.
```

We more often change "time of day" to take care of drastic events:

```
At the time when the player loses consciousness:
    now the time of day is 10:12 AM;
    say "A mist comes over your vision, and when you come to, it is morning and you are in bed."
```

Example

141. IPA
Shops which each have opening and closing hours, so that it is impossible to go in at the wrong times, and the player is kicked out if he overstays his welcome.
RB 4.3 Event Scheduling

§9.7 Telling the time

Now that we have the time of day, we can of course use this value in room descriptions and the like:

```
The Clock Chamber is a room. "The dark chamber behind the clock face, a mill-room of gears which grind down the seconds. Through the glass you can see the reversed hands reading [the time of day]."
```

It seems odd, though, to read a precise numerical description of the time here: after all, it isn't a digital clock. A friendlier version would use:

say "[(time) in words]"

This text substitution produces the given time written out in English sentence form. For example:

```
"Through the glass you can see the reversed hands reading [the time of day in words]."
```

might produce

```
Through the glass you can see the reversed hands reading twenty to nine.
```

To reiterate an example which came up earlier, we could even work the time of day into the command prompt, which would lend the proper sense of urgency to a story played out against the clock:

```
When play begins: now the command prompt is "[time of day] >".
```

Example

142. **Situation Room**
    Printing the time of day in 24-hour time, as in military situations.

RB 4.1 The Passage Of Time

§9.8 Approximate times, lengths of time

Clocks and watches vary considerably in how much detail they show, and we tend not to report the time over-precisely: half-past ten is an elastic concept. The following room description for the Clock Chamber comes across much more naturally:

```
The Clock Chamber is a room. "The dark chamber behind the clock face, a mill-room of gears which grind down the seconds. Through the glass you can see the reversed hands reading [the time of day to the nearest five minutes in words]."
```

The phrase "... to the nearest ..." rounds off the given time, just as it sounds; as we'll see later, it can actually round off any arithmetic values, not just times. For instance, "9:58 PM to the nearest ten minutes" is 10:00 PM.

In talking about lengths of time, rather than times of day, it's useful to have these:

(number) **minutes** ⇒ *time*

This phrase converts numbers into lengths of time. Example:

```
15 minutes
```

Because it's a phrase, not just a notation for writing constants down, the number doesn't have to be given literally:

```
let X be 5;
if the player is in the Slow Room, now X is 10;
let deadline be the time of day plus X minutes;
```

Note that lengths of time can't exceed 1440 minutes.

(number) **hours** ⇒ *time*

This phrase converts numbers into lengths of time. Example:

```
10 hours
```

Note that lengths of time can't exceed 24 hours.

§9.9 Comparing and shifting times

Carrying out easy calculations with times is straightforward:

```
The chronometer is in the Clock Chamber. "On one wall is a terribly self-important chronometer showing the time in major world cities. London: [time of day]. Paris: [one hour after the time of day]. Tokyo: [9 hours after the time of day]. Cupertino, California: [7 hours before the time of day]."
```

Here we are using two phrases:

(time) **before** (time) ⇒ *time*

This phrase produces a time earlier by the amount given, keeping within the 24 hour clock. Example:

```
7 hours before 5:30 AM
```

produces 10:30 PM.

(time) **after** (time) ⇒ *time*

This phrase produces a time later by the amount given, keeping within the 24 hour clock. Example:

```
9 hours after 11 AM
```

produces 8 PM.

Similarly, we have conditions:

if (time) **is before** (time):

This condition is true if the first time occurs earlier in the day than the second. In recognition of the fact that very few stories begin before 4 AM, whereas many run on past midnight, the start of the day is taken to be 4 AM: thus 3:59 AM is after 11:10 PM, but 4:04 AM is before it.

if (time) **is after** (time):

This condition is true if the first time occurs later in the day than the second. In recognition of the fact that very few stories begin before 4 AM, whereas many run on past midnight, the start of the day is taken to be 4 AM: thus 3:59 AM is after 11:10 PM, but 4:04 AM is before it.

§9.10 Calculating times

We will occasionally need to perform more complex calculations with time, and in order to do that, we have a way to convert the time of day to numbers. Thus the phrase "the minutes part of ..." takes a time and produces a number from 0 to 59; similarly "the hours part of ..." extracts a number from 0 to 23, using the twenty-four hour clock.

minutes part of (time) ⇒ *number*

This phrase converts a time to a number, then takes the result mod 60, which in effect produces the number of minutes after the hours are thrown away. Example:

```
minutes part of 12:41 PM
```

produces 41.

hours part of (time) ⇒ *number*

This phrase converts a time to a number, then divides the result by 60, which in effect produces the number of hours after minutes are thrown away. Example:

```
hours part of 8:21 AM
```

produces 8.

To go the other way, we can convert any number to a duration by writing "minutes" or "hours" after it. For instance:

```
The clock error is a number that varies. To thump the mechanism: now the clock error is a random number from -10 to 10.

The broken grandfather clock is in the Chamber. "An erratic grandfather clock seems to say it is [clock error minutes after the time of day]."

When play begins, thump the mechanism. Instead of attacking the broken clock: thump the mechanism; say "You thump the clock, which now reads [clock error minutes after the time of day].".
```

Note that "clock error" is a number, but "clock error minutes" is a time.

§9.11 Future events

We often want to arrange for something to happen at some point in the future. Here is yet another timepiece:

```
An egg-timer is in the Chamber. "A plastic egg timer in the shape of a chicken can be pressed to set it going."

Instead of pushing the egg-timer:
    say "It begins to mark time.";
    the egg-timer clucks in four turns from now.

At the time when the egg-timer clucks:
    say "Cluck! Cluck! Cluck! says the egg-timer."
```

The event here is called "the egg-timer clucks". It only happens if we instruct so, using one of the following phrases:

(rule) **in** (time) **from now**

This phrase causes the given rule to be run at a given time offset from the current time of day. Example:

```
the egg-timer clucks in 18 minutes from now;
```

(rule) **in** (number) **turn/turns from now**

This phrase causes the given rule to be run at a given number of turns after the current one. Example:

```
the egg-timer clucks in four turns from now;
```

(rule) **at** (time)

This phrase causes the given rule to be run at a given time of day. Example:

```
the egg-timer clucks at 11:35 AM;
```

If we know in advance what time we want something to happen, we can more simply write:

```
At 4 PM: say "The great bells of the clock tower chime four."
```

(Note that in either case such rules begin with the word "at": they are the only rules allowed to begin with the word "at".)

A small warning: timed events like these only have a chance to occur during the turn sequence, that is, once every turn. In most stories, one turn takes one minute, so there will in due course be a turn happening at exactly (say) 11:35 AM. But if the clock is being advanced faster than this, it's possible that there are turns at (say) 11:32 AM and then not until 11:37 AM. But an event set for 11:35 AM will nevertheless happen -- it will run at the first available turn after that time, which will be 11:37 AM. Events can thus happen up to half an hour late, though Inform cancels them if the elapsed time is greater than that.

The Scenes panel of the Index can be a useful way to see what events have been set.

Examples

143. **MRE**
Hunger that eventually kills the player, and foodstuffs that can delay the inevitable by different amounts of time.
RB 9.1 Food

144. **Totality**
To schedule an eclipse of the sun, which involves a number of related events.
RB 3.9 Passers-By, Weather and Astronomical Events

145. **Empire**
A train which follows a schedule, stopping at a number of different locations.
RB 8.2 Ships, Trains and Elevators

146. **Hour of the Wren**
Allowing the player to make an appointment, which is then kept.
RB 4.3 Event Scheduling

§9.12 Actions as conditions

There are two ways that descriptions of actions can be used as conditions. First, we can simply describe an action, and then the condition will be true if that is what the player is trying to do, and not otherwise:

```
if taking a container, ...
```

This is actually an abbreviation for the longer, some would say preferable form:

```
if we are taking a container, ...
```

Secondly, we can talk about the past as well as the present, which is very useful since interactive fiction often contains situations which are changed by earlier events.

```
Instead of waiting when we have taken the lantern, say "No, your acquisitive nature is roused now, and simply waiting will no longer do."
```

More on the past tense later follows in the next section: note that "we are taking" has become "we have taken". For the rule to apply, it is not enough that the action "taking the lantern" has been tried: it must have succeeded. Note also that it's enough for any actor

in the story to have successfully taken the lantern: it doesn't have to be the player.

Examples

147. Night Sky
 A room which changes its description depending on whether an object has been examined.
 RB 3.1 Room Descriptions

148. Zero
 A box which called "horribly heavy box" after the player has tried to take it the first time.
 RB 5.5 Memory and Knowledge

§9.13 The past and perfect tenses

The remaining sections of this chapter go into more technical ways to think about the progress of the story through time, and can be skipped at a first reading.

Conditions are clauses which require Inform to make a decision: is such-and-such true, or not true? We have already seen conditions attached to rules using "when":

```
Instead of waiting when the Sorting Hat is in the Hall: ...
```

and, as we shall later see, we will often want to write instructions like:

```
if the Sorting Hat is in the Hall, say "Hermione blinks apprehensively."
```

The condition is "the Sorting Hat is in the Hall", and during play this will sometimes be true and sometimes false.

A condition in the form "X is Y" is of course written in the present tense, and refers to the current state of affairs. Three other tenses are allowed. First, the present perfect:

```
if X has been Y ...
```

is true if it has ever been the case that "X is Y" at the start of any turn (or any action). So, for instance,

```
if the gate has been open ...
```

will be valid if and only if the gate has ever been made open by any action (even if it is

closed now), or if it started out by being open when play began.

Next is the past tense:

```
if X was Y ...
```

holds if and only if "X is Y" was true at the start of the most recent action. This is convenient when trying to describe what has changed in the course of the action, but sometimes also when making the action itself happen. For instance:

```
if the lantern was switched on, now the lantern is switched off;
if the lantern was switched off, now the lantern is switched on;
```

Completing the set is the past perfect:

```
if X had been Y ...
```

which records whether "X has been Y" was true at the start of the most recent action. All these verbs can of course be negated (though "wasn't" and "hadn't" are disallowed as poor style: we use "was not" and "had not" instead). So for example,

```
if the player had not been in the Ballroom ...
```

is true if the player hadn't visited the Ballroom at the start of the most recent action.

Something we must watch out for is that variables might not have the same values in the past that they have now. As a result, writing conditions such as "if the noun has been open" is a bad idea, because in the past "the noun" very likely referred to something different. It is really only safe to talk in the past tense about definite, fixed things: "if the Great Gates of Kiev have been open" would be fine.

Examples

149. **Tense Boxing**
    An overview of all the variations of past and present tenses, and how they might be used.
    RB 5.5 Memory and Knowledge

150. **Bruneseau's Journey**
    A candle which reacts to lighting and blowing actions differently depending on whether it has already been lit once.
    RB 10.8 Fire

151. **Elsie**
    A door that closes automatically one turn after the player opens it.
    RB 3.5 Doors, Staircases, and Bridges

§9.14 How many times?

There are two further ways to examine the historical record. Given any condition, we can say

```
if (...condition...) for the second time ...
if (...condition...) twice ...
if (...condition...) 2 times...
if (...condition...) two times...
```

(all of which are synonymous: the words once, twice, thrice, one, two, three, four, five, six, seven, eight, nine, ten, first, second, third, fourth, fifth, sixth, seventh, eighth, ninth and tenth all mean what they obviously should). The result is true if the condition holds now and has held for only one previous spell in the past. A condition holding for, say, fifteen consecutive turns without a break counts as only one "time" - so what we mean by "twice" here is that it is true now, was previously false for a while, and was previously true for a while before that, but no more. In effect, then,

```
if the player is in the Ballroom for the third time ...
```

is true if this is the third visit to the Ballroom. We can also say

```
if the player is in the Ballroom for more than the third time ...
```

or similarly "less than", "at least", "at most". It would be more natural, though, to say

```
if the player has been in the Ballroom three times ...
```

The adjective "only" (or equivalently "exactly") can be added to obtain

```
if the player has been in the Ballroom only three times ...
```

To recap, this means there have been exactly three visits to the Ballroom in history, whereas

```
if the player is in the Ballroom for the third time ...
```

means there have been exactly three visits, the third of which is still going on - an important distinction.

Example

152. **Infiltration**
     A room whose description changes depending on the number of times the player has visited.

RB 3.1 Room Descriptions

§9.15 How many turns?

So much for "times" - spells in which a condition is true. We can also test the length of time, in turns of play, that something has been true. Thus:

```
if ... for three turns;
```

means that the condition holds now, and held at the start of this turn, at the start of last turn, and at the start of the turn before that. In particular:

```
if the floppy hat has been worn for three turns ...
```

will be false if the hat is not currently worn (even if it has been often in the past) and, on the other hand, will be true if the hat has been worn for twenty turns. Here again we can be more specific. These are synonymous:

```
if the floppy hat is worn for the third turn ...
if the floppy hat has been worn for only 3 turns ...
if the floppy hat has been worn for exactly three turns ...
```

all requiring that the hat wasn't worn four turns ago. As before, "more than", "less than", "at least" and "at most" so forth can also be used - say, "for at least 21 turns".

A warning: we must be careful when writing something like

```
if the noun has been open ...
```

since this tests whether it has ever been true that the noun of the then action was open: not whether the current noun-object has ever been open.

Lastly, note that the beginning of play - when (usually) initial text and a banner is printed, followed by a room description - counts towards these counts. In effect, this is a turn: one in which the player compulsorily performs the looking action, rather than being asked for a command. (By convention it is numbered as turn number 0, and doesn't contribute towards the turn count.)

Example

153. Annoyotron Jr
    A child who after a certain period in the car starts asking annoying questions.

RB 7.2 Liveliness

10. Scenes

§10.1 Introduction to scenes
§10.2 Creating a scene
§10.3 Using the Scene index
§10.4 During scenes
§10.5 Linking scenes together
§10.6 More general linkages
§10.7 Multiple beginnings and repeats
§10.8 Multiple endings
§10.9 Why are scenes designed this way?

§10.1 Introduction to scenes

As we have seen, Inform divides up space into individual places called "rooms", and allows us to group rooms together into "regions" if we find that convenient. And Inform also divides time up, into individual turns. These too we can group together: the equivalent of a region is a "scene".

To put this another way, if we think of the interactive fiction as a stage play, then up to now it has simply contained endless dialogue and stage directions - there has been no convenient way to divide up its running time into dramatic episodes, in the same way that a playwright might make Act II take place in the same drawing-room as Act I, but (let us say) six months later, after many things have changed. The script contains cues for one scene to end and another to begin: when those cues are reached, the stage hands rearrange props, actors reposition themselves and so on.

Inform also allows us to create scenes, with cues for them to start and end, and some stage machinery (so to speak) making it easy to move the action on. But interactive fiction is *interactive*, so the metaphor of the theatre only goes so far. We can have several different scenes going on at once - perhaps with the relevant events taking place in different rooms, which the player is free to walk between. And the player may make a choice which changes the story-line, causing scenes to happen which otherwise would not have happened, and so on. Scenes can even be "recurring", that is, can repeat themselves.

So organising the story-line into scenes is not simply a matter of making a list (Scene 1, then Scene 2, then Scene 3, *finis*). It is more like a chart in which one scene can lead in several possible ways to others - a sort of map of time, which as we shall see Inform displays in its "Scenes" index.

§10.2 Creating a scene

As usual, we only need to say that something is a scene to make it so:

```
Train Stop is a scene.
```

We conventionally write scene names with capital letters, as this demonstrates.

This works, and shows up in the "Scenes" index, but does nothing. We have given no instructions on when it begins - no cue, in stage-play terms - so it never will begin, and even if it did, nobody would notice since it does nothing. First, to give it a beginning:

```
Train Stop begins when the player is in the Station for the third turn.
```

In theory any condition can be used to cue the scene - here, it's "the player is in the Station for the third turn" - but it's wise to look for a state of affairs which will last at least a brief time, because scene changes only happen at the start and end of turns. (Something like "…when examining the timetable" may be true only for a part of the middle of a turn, and so go unnoticed.)

Every scene has two rulebooks attached, one at each end, so to speak. These look very like "when play begins" and "when play ends", and work in the same way. Thus:

```
When Train Stop begins:
    now the Flying Scotsman is in the Station;
    say "The Flying Scotsman pulls up at the platform, to a billow of steam and hammering."

When Train Stop ends:
    now the Flying Scotsman is nowhere;
    if the player is in the Station, say "The Flying Scotsman inches away, with a squeal of released brakes, gathering speed invincibly until it disappears around the hill. All is abruptly still once more."
```

Thus when the scene begins, our imaginary stage-hands wheel in a steam train; when it ends, they get rid of it again. Note that we know where the player will be at the start of the scene, but by the end he may have wandered off across the fields, so we must be careful not to report something he might not be in a position to see.

When Train Stop begins, we printed some text, but we did this by hand. We didn't need to, because Inform automatically prints out the description of a scene (if it has one) when the scene begins. Scenes can have properties, just like objects, and in particular they have the "description" property. For example, we could write:

```
Arrival is a scene. "There's a flourish of trumpets."
```

which saves us the trouble of writing the rule:

```
When Arrival begins: say "There's a flourish of trumpets."
```

We can also write rules like this which apply to a whole variety of scenes at once. For instance:

```
A scene can be bright or dim. A scene is usually dim. Dawn is a bright scene.
When a scene which is bright ends: say "So passes the bright [scene being changed]."
```

Here, instead of naming a scene ("Train Stop"), we've given a description ("a scene which is bright"). When a scene begins, these general rules come before those which name the scene exactly; when it ends, the reverse is true.

Examples

154. **Pine 1**
Pine: Using a scene to watch for the solution of a puzzle, however arrived-at by the player.
RB 7.3 Reactive Characters

155. **Entrapment**
A scene in which the player is allowed to explore as much as he likes, but another character strolls in as soon as he has gotten himself into an awkward or embarrassing situation.
RB 4.2 Scripted Scenes

§10.3 Using the Scene index

But when we test the previous section's example, we find that after a brief wait, the train pulls up: but it never goes away again. We have given instructions on how the scene ends, but not when it ends, and as a result the scene goes on forever once started.

Even with simple story-lines, and this one could hardly be simpler, it is surprisingly easy to overlook something so that the whole story-line is derailed.

The Scenes page of the index is intended to help with this. The Plot section shows all of the scenes and how they are to begin, along with a key to the symbols used on it. One scene always included is "Entire Game", a special scene which, as its name implies, is always being played out. But if we look at the Scene index for the previous example, we

will also see our Train Stop scene, and find that it is marked with the red warning symbol for "never ends". Let us fix this:

```
Train Stop ends when the time since Train Stop began is 3 minutes.
```

Note the useful value "time since Train Stop began":

time since (scene) **began** ⇒ *time*

This phrase produces the time since the named scene began, which only makes sense, of course, if it has indeed begun. Example:

```
time since Entire Game began
```

time since (scene) **ended** ⇒ *time*

This phrase produces the time since the named scene ended, which only makes sense, of course, if it has indeed ended. Example:

```
time since Formal Dinner ended
```

The actual times, in case they are needed, can be obtained with:

time when (scene) **began** ⇒ *time*

This phrase produces the time (i.e., the value of the "time of day" variable) at the moment when the given scene began.

time when (scene) **ended** ⇒ *time*

This phrase produces the time (i.e., the value of the "time of day" variable) at the moment when the given scene ended.

The testing command SCENES causes Inform to monitor the beginning and ending of scenes, and report on them. For instance:

```
>ask monk about lodging
"Welcome a poor traveler for the night?" you ask, rubbing your fingers together to restore a little feeling.

The monk looks you up and down for a moment and you sense his reaction hanging in the balance; then he slaps you on the back, hard enough to drive the air from your lungs. "In."

[Scene 'Greeting' ends]
The monk takes your elbow and pushes you imperiously toward dinner.
[Scene 'Banquet' begins]
```

Example

156. **Age of Steam**
    The railway-station examples so far put together into a short game called "Age of Steam".

RB 4.4 Scene Changes

§10.4 During scenes

Scenes are not only useful for changing the setting, by moving items or people around and providing a little narration. We can also make the rules different in one scene from another. For instance, at a sleepy country halt there is no reason why one should not walk across the tracks: but if there is a train in the way, that would be impossible.

```
Before going north during the Train Stop, say "The train blocks your way." instead.
```

Any rule can have the clause "during ..." attached, provided that clause goes at the end and either explicitly names a scene, or gives a description of which scenes would match. This is especially useful with "every turn":

```
Every turn during the Train Stop, say "Water is sluiced out of the tank and into the engine."
```

We can test whether a scene is happening with the adjective "happening":

```
if Train Stop is happening, ...
```

if (scene) **has happened:**

This condition is true if the given scene has both begun and ended.

if (scene) **has not happened:**

This condition is true if the given scene has not ended (or never started).

if (scene) **has ended:**

This condition is true if the given scene ended at least once.

if (scene) **has not ended:**

This condition is true if the given scene has never ended.

We need to be a bit careful: it's possible to set things up so that the Train Stop scene will play out more than once, so "Train Stop is happening" and "Train Stop has happened" might both be true at once.

The kind of value "scene" is one which is allowed to have properties - it has a tick in the "properties" column in the chart in the Kinds index - and this can be very useful in describing scenes. For instance, we could write:

```
A scene can be thrilling or dull. Train Stop is dull.
A scene has a text called cue speech. The cue speech of Train Stop is "All aboard!".
```

Inform has the adjectives "recurring", "non-recurring" and "happening" all built in to describe scenes, and the above would add "thrilling" and "dull". Moreover, the "during" clause of a rule can give a description of a scene as easily as a specific scene name. For instance:

```
Before going north during a dull non-recurring scene, ...
```

Examples

157. **Full Moon**
Random atmospheric events which last the duration of a scene.
RB 3.9 Passers-By, Weather and Astronomical Events

158. **Space Patrol - Stranded on Jupiter!**
We'll be back in just a moment, with more exciting adventures of the… Space Patrol!
RB 4.4 Scene Changes

159. **Bowler Hats and Baby Geese**
Creating a category of scenes that restrict the player's behavior.
RB 4.2 Scripted Scenes

160. **Day One**
A scene which plays through a series of events in order, then ends when the list of events is exhausted.
RB 4.2 Scripted Scenes

§10.5 Linking scenes together

Let us suppose that somebody gets off the train, after all, so that a second scene follows on.

```
Brief Encounter is a scene. Brief Encounter begins when Train Stop ends.
```

The effect of this is that they occur in sequence. If we add a third to the chain of scenes:

```
Village Exploration is a scene. Village Exploration begins when Brief Encounter ends.
```

…we find another chance to fool ourselves: if we check the Scenes index again, we can see the linkages between these scenes, but we also see that Brief Encounter never ends (despite its name). All we have said is that another scene begins where Brief Encounter leaves off, but it never does, so this is moot.

Example

161. **Pine 2**
Pine: Adding a conversation with the princess, in which a basic set of facts must be covered before the scene is allowed to end.
RB 7.12 Characters Following a Script

§10.6 More general linkages

We are allowed to link the beginning or end of any scene to the beginning or end of any other scene. So, for instance:

```
Luggage Trouble is a scene. Luggage Trouble begins when Brief Encounter begins.
```

Thus the two scenes run concurrently, at least for a while. We can also add that:

```
Luggage Trouble ends when Brief Encounter ends.
```

This can be useful when a large, complicated scene really contains several smaller sub-scenes.

A special exceptional case is that we can have any scene or scenes starting right at the outset:

```
Railway Meeting is a scene. Railway Meeting begins when play begins.
```

When play ends, of course, all scenes end, so there is no need to say that.

Examples

162. **The Prague Job**
Scenes used to provide pacing while the player goes through his possessions.
RB 4.2 Scripted Scenes

163. **Entrevaux**
Organizing the game by scenes, where each scene has a location and prop lists so that it can be set up automatically.
RB 4.4 Scene Changes

§10.7 Multiple beginnings and repeats

It is quite allowed for a scene to be linked to several other scenes, and this is useful if several alternate strands of plot are being brought together in a common resolution scene:

```
Bittersweet Ending begins when Stranger's Rejection ends.
Bittersweet Ending begins when Stranger's Acceptance ends.
```

and we can also have the same scene beginning when a condition holds. In general, it will begin the first time it gets any chance to do so.

All scenes are ordinarily set up so that they can happen only once. But sometimes we want them to repeat. Suppose the train calls not once only, but every twenty minutes. We could set this up with two scenes linked back to back like so:

```
Train Stop is a recurring scene. Train Wait is a recurring scene.
Train Wait begins when play begins.
Train Stop begins when Train Wait ends.
Train Wait begins when Train Stop ends.
```

The difference here is that these scenes have been declared as "recurring". In all other respects they are the same as any other scene.

Examples

164. **Night and Day**
Cycling through a sequence of scenes to represent day and night following one another during a game.
RB 3.9 Passers-By, Weather and Astronomical Events

165. **Pine 3**
Pine: Allowing the player to visit aspects of the past in memory and describe these events to the princess, as a break from the marriage-proposal scene.
RB 4.5 Flashbacks

§10.8 Multiple endings

Interactive fictions vary considerably in the extent to which the player is allowed to influence the story-line. Sometimes the appearance of making choices is wholly bogus, as only one possible line is given more than lip service. But in other works, the player can radically change the outcome, and whole rafts of plot differ between one person's experience and another's. Inform allows multiple endings to its scenes to make this kind of branching story-line easy to achieve.

Any scene can have up to 31 alternate endings, differentiated by name (unless the Z-machine format has been selected on the Settings panel, in which case, 15). These alternates are created as and when conditions are set for them:

```
Brief Encounter ends happily when ...
Brief Encounter ends wisely but sadly when ...
```

"Ends happily" and "ends wisely but sadly" behave just like "ends". We can have rules "When Brief Encounter ends happily, …" and so forth, in addition to rules "When Brief

Encounter ends, …" - if a rule doesn't specify any particular ending, it applies to all of them.

We can also link rules together from these branches, so

```
Stranger's Acceptance begins when Brief Encounter ends happily.
Stranger's Rejection begins when Brief Encounter ends wisely but sadly.
```

With this set-up and that of the previous section, there are now two possible paths through the story:

```
(i) Train Stop - Brief Encounter - Stranger's Acceptance - Bittersweet Ending
(ii) Train Stop - Brief Encounter - Stranger's Rejection - Bittersweet Ending
```

We might later need to know which of these paths has been taken, and to help with this Inform provides conditions like so:

```
if Brief Encounter ended happily ...
if Brief Encounter did not end happily ...
if Brief Encounter ended wisely but sadly ...
if Brief Encounter did not end wisely but sadly ...
```

(For a scene which repeats, note that these conditions apply only to the most recent repetition: and that such conditions are always false if the scene is currently going on. "Brief Encounter did not end happily" will be true only when the scene has finished but in a different way.)

Examples

166. **Panache**
Replacing the score with a plot summary that records the events of the plot, scene by scene.

RB 11.4 Scoring

167. **Pine 4**
Pine: Adding a flashback scene that, instead of repeating endlessly, repeats only until the Princess has understood the point.

RB 4.5 Flashbacks

§10.9 Why are scenes designed this way?

In the part it plays in stories, time is like space. The endings of a scene (along with its

beginning) are like the map connections leading out of a room. The Scenes index keeps track of the "map of time" through which these possible story-lines traverse. Some works of IF will have immensely complicated story-lines in only a few rooms, some will have no scenes at all despite a sprawling geography. The Scenes and World index tabs, side by side, show both kinds of map.

Just as Inform uses a simple but practical design for the boundaries between rooms (map connections and doors, that is), it also simplifies transitions between scenes. Scenes are based on states of things: we give circumstances for them to begin or end. There is no phrase with the power to say "make Act II begin right now", so perhaps it is worth explaining why not. The state-based approach was chosen because:

* it guarantees that each action falls entirely inside, or entirely outside, of any given scene (and therefore that "during..." clauses in the conditions for a rule are not affected by rule ordering);

* it ensures that scene changes occur outside actions, like every turn rules;

* it promotes a style of writing which makes it clearer to the reader of the source text when a scene begins and ends, and what conditions are guaranteed to be true during it;

* it makes it possible for the Scenes index page to show this information in a communicative way.

Settings in IF where one revisits the same location but at a different time, or after a dramatic change, have historically been difficult to test properly and prone to mistakes. (The classic example would be where a character killed during Act I reappears unharmed in Act II.) The design of scenes is an attempt to encourage a style of writing which minimises the risk of these accidents.

Since scenes are, in the end, only a convenient way to organise rules, and do nothing that cannot be done by other means, this simplified system of scene changing does not really restrict us.

Example

168. **Cheese-makers**
    Scenes used to control the way a character reacts to conversation and comments, using a TALK TO command.

RB 7.8 Saying Complicated Things

11. Phrases

§11.1 What are phrases?
§11.2 The phrasebook
§11.3 Pattern matching
§11.4 The showme phrase
§11.5 Conditions and questions
§11.6 If
§11.7 Begin and end
§11.8 Otherwise
§11.9 While
§11.10 Repeat
§11.11 Repeat running through
§11.12 Next and break
§11.13 Stop
§11.14 Phrase options
§11.15 Let and temporary variables
§11.16 New conditions, new adjectives
§11.17 Phrases to decide other things
§11.18 The value after and the value before

§11.1 What are phrases?

Phrases are instructions to Inform to do something, or to decide whether something is true or false, or to produce a value, or to say something. Inform has around 350 phrases built-in, and the chapters so far have already defined about 100 of those. In this chapter we'll see some key phrases for organising instructions of what to do, and also see how to define entirely new phrases.

Just to run through the four sorts of phrase with examples:

(a) Phrases to do something. These are the ones used in the body of a rule. For example,

```
When Train Stop begins:
    move the Flying Scotsman to the Station;
    say "The Flying Scotsman pulls up at the platform."
```

Rules like this begin with a "preamble", the beginning part which tells Inform when or how they apply, and then follow on with a list of instructions - here, just two of them. "move … to …" and "say …" are both phrases. Inform provides about 130 of these built-in. It's actually not quite true that they all do something, because one of them is:

do nothing

This phrase does nothing at all. It is very occasionally useful to make a rule which does nothing:

```
This is the largely ineffective rule:
    do nothing.
```

(b) Phrases to decide whether a condition is true. These are the ones which can be used in an "if":

```
if action requires light: ...
```

Not all conditions come from phrases. For example, "if the front door is closed" and "if Peter is wearing the sandals" have meanings which come from the verbs "to be" and "to wear". Inform provides about 60 built-in conditions, which give a friendly wording for questions which would be lengthy or difficult to write in any other way.

(c) Phrases to decide a value. For example:

```
square root of 16
```

produces a number, 4 of course, and can be used whenever a number is expected. Inform provides about 100 built-in phrases like this.

(d) Text substitutions. These are actually just phrases whose definition begins with "To say …". Example:

```
"It's now [time of day in words]."
```

Inform provides about 60 built-in text substitutions.

§11.2 The phrasebook

The Phrasebook is Inform's collection of recognised phrases, and it can always be browsed using the Index panel of the same name. Even the smallest project has a good-sized phrasebook, since it contains all of the built-in phrases. But most projects also define new phrases of their own.

Here is a simple definition of a new phrase:

```
To spring the trap:
	say "'Sproing!' go the hinges and, with a flash of silver, the enormous blades whisk together!";
	end the story.
```

Inform allows us to use whatever conventions of layout we prefer, but it's customary to use indentation like this, dividing off the preamble from the phrases which follow. As can be seen, definitions of new phrases look very like rules.

What makes this definition a simple one is that the wording is fixed. The only way to use this would be from another phrase or rule, like so:

```
Instead of entering the cage:
	spring the trap.
```

In the next section we'll see how to give more complicated definitions which, like "move … to …", allow for the wording to change with the circumstances.

§11.3 Pattern matching

In this section, let's make the following new phrase:

```
To admire (item - an object):
	say "You take a long look at [item].".
```

This does very little, of course, but it does allow the wording to be different each time the phrase is used:

```
admire the diamonds;
admire Mr Cogito;
admire the honey sandwich;
```

and our single definition covers all of these possibilities. The bracketed part of the definition, "(item - an object)", tells Inform to expect an object in that position, and Inform enforces this carefully. So this definition might tell Inform what "admire the barricade" means, but not what

```
admire "blue cheese";
admire 63;
```

mean. Unless some other definition sorts the matter out, Inform will reply to uses like this with a Problem message:

```
Problem. You wrote 'admire 63' , but '63' has the wrong kind of value: a number rather than an object.
```

The object does not need to be named literally, but can be anything which works out to be an object: for instance,

```
After dropping something in the Auction House:
    admire the noun.
```

which Inform allows because "noun", here, is a name for the object which is being acted on.

Inform decides which definition to apply in a process called "pattern matching".

The bracketed part of the example definition has the form "(name - kind)". The definition only applies if the text supplied agrees with the "kind" part - for instance, the diamonds agreed with "object", but 63 did not. If the definition does apply, then the Inform works through the rest of the phrase using "name" to mean whatever value matched. For example:

```
To slam shut (box - an open container):
    say "With great panache, you slam shut [the box].";
    now the box is closed.
```

When this phrase is followed, "box" means whatever open container the pattern-matcher found when it was called for. For example, if Inform reads

```
slam shut the Dutch armoire;
```

then it acts on this by following the definition of "slam shut ...", using the Dutch armoire object as the value of "box", so it prints:

```
With great panache, you slam shut the Dutch armoire.
```

and renders it closed.

In fact any description can be given in the definition, and that includes a single, specific value. For instance, we could define:

```
To grant (bonus - a number) points:
    increase the score by the bonus.

To grant (bonus - 7) points:
    say "You shiver uncontrollably."
```

which would withhold this unlucky bounty. That would mean that:

```
grant 7 points;
grant seven points;
```

would each produce uncontrollable shivers, because Inform uses the definition applying to the number 7; but

```
grant six points;
```

would increase the score by 6. In general Inform always follows the principle that more specific definitions take priority over more general ones. So although the definitions:

```
To grant (bonus - a number) points: ...
To grant (bonus - 7) points: ...
```

both apply to the case of "grant 7 points", Inform uses the second, because it's the more specific of the two possibilities.

Sometimes it will not be possible to tell if the value supplied meets the requirements until the story is actually playing. If, at run-time, no definition fits some phrase which has to be carried out, a run-time problem message is produced.

Finally, and more straightforwardly, we can specify variations in wording using slashes between alternative words in a "To …" definition. For instance:

```
To grant (bonus - a number) point/points: ...
```

allows the final word to be either "point" or "points". Slashes like this can only be used with literal words, not bracketed values, and give alternative forms only of a single word at a time; the alternative "--" means "no word at all", and thus makes it optional:

```
To grant (bonus - a number) point/points/--: ...
```

makes "grant 3" do the same as "grant 3 points".

If we need more variation than that, we should make more than one definition.

Examples

169. **Ahem**
   Writing a phrase, with several variant forms, whose function is to follow a rule several times.
   RB 2.1 Varying What Is Written

170. **Ferragamo Again**
   Using the same phrase to produce different results with different characters.
   RB 7.10 Character Emotion

§11.4 The showme phrase

We've already seen the SHOWME command, which can be typed into the Story panel to look at the state of something, usually a thing or room. SHOWME is a testing command which has no effect once the work is released; eventual players can't use it.

Inform also has a phrase called "showme", which works in much the same way:

showme (value)

This phrase is intended for testing purposes only. If used in a story file running inside the Inform application, it prints a line of text showing the given value and its kind; in a Released story file, it does nothing at all. Example:

```
When play begins: showme 11.
```

produces

```
number: 11
```

More usefully:

```
Every turn: showme the score.
```

Now, every turn, we get a line in the story's transcript like so:

```
"score" = number: 0
```

Inform uses the quotation marks and equals sign to show that it had to do some work to find the answer. "score" wasn't a constant value - it was a variable, and Inform had to look up the current value.

"showme" is a convenient way to see what's going on inside a phrase which isn't behaving as expected, or to find out the kind of a value. Here are some trickier examples. Suppose our design includes:

```
The matching key of the blue door is the brass Yale key.
```

If we then try this:

```
When play begins:
    showme matching key of the blue door.
```

we get, when the story starts up,

```
"matching key of the blue door" = object: brass Yale key
```

Why is this an "object", when we know that the key is actually a "thing"? After all, if we "showme key" instead, we get:

```
thing: brass Yale key
```

The answer is a little technical: it's because Inform guarantees that the matching key is always an object, but not that it's always a thing - it just happens to be a thing at the moment. There's not really a contradiction, because a "thing" is a kind of "object", so in fact the key is both. If we try "showme matching key", we get something like this:

```
objects valued property: property 23
```

which is even more technical - people never need to print the names of abstract property names during play, so Inform doesn't provide any good way of doing it. It is reduced to printing out an internal ID number ("property 23") instead of the name ("matching key"). This can't be helped: "showme" is a way to lift the lid and see what's going on inside Inform's machinery, and some of the corners are dark.

All the same, "showme" can be very useful in tinkering with rules to make them work properly. It prints nothing at all in a Release version of a project, so it's impossible for these private notes to be shown accidentally to our eventual readers.

§11.5 Conditions and questions

A variety of "conditions" have already appeared in this documentation. A condition is a phrase which describes a situation which might be true, or might be false, and examples might include:

```
Mr Kite is in Bishopsgate
the score is greater than 10
Sherlock Holmes suspects a woman
```

These are all examples of sentences, formed by putting nouns either side of a verb, and clearly a wide range of conditions can be written this way. But there are also a few special conditions built into Inform which have a fixed wording, and test questions difficult to address with ordinary sentences. For instance:

if in darkness:

This condition is true if the player currently has no light to see by. Note that the test is more complicated than simply testing

```
if the player is in a dark room, ...
```

since the player might have a torch, or be inside a cage which is itself in a dark room, and so on.

Another example of a condition not easily written as a sentence is:

if player consents:

This condition is unusual in doing something and not simply making a silent check: it waits for the player to type YES (or Y) or NO (or N) at the keyboard, and then is true if the answer was yes. Example:

```
say "Are you quite sure you want to kiss the Queen? ";
if the player consents:
    ...
```

Whether it's put to the player like this or not, testing a condition is really asking a question, and there is always a yes/no answer. In Inform this answer is not usually a value (unlike in some other computer programming languages), but it can be made into one.

Firstly, we need a special kind of value to hold answers like this. It's called "truth state", and it has just two possible values, written as "true" and "false". We then need:

whether or not (a condition) ⇒ *truth state*

This phrase converts a condition into its result as a value, which is always either "true" or "false". Example:

```
whether or not 20 is an odd number
```

produces the truth state "false". This is mostly useful for storing up results to look at later:

```
let victory be whether or not all the treasures are in the cabinet;
```

and then subsequently:

```
if victory is true, ...
```

As another example, in most stories this:

```
When play begins:
    showme whether or not in darkness.
```

…will produce a line:

```
"whether or not in darkness" = truth state: false
```

In short, "truth state" is a kind of value like any other. That means it can be the kind of a variable:

```
Salvation earned is a truth state that varies.
```

and it can similarly be used in table columns, lists, or anywhere else where values are allowed.

Example

171. **Proposal**
    Asking the player a yes/no question which he must answer, and another which he may answer or not as he chooses.

RB 7.7 Saying Simple Things

§11.6 If

Inform's most powerful phrases are those which control the others, making them repeat, or be skipped.

if (a condition) , (a phrase)

or…

if (a condition):

This phrase causes the single phrase, or block of phrases, following it to be obeyed only if the condition is true. (If the condition must contain a comma for some reason, the block form should be used.) Example:

```
if the red door is open, say "You could try going east?"
```

The sense of an "if" can be reversed by using the word "unless" instead:

unless (a condition) , (a phrase)

or…

unless (a condition):

This phrase causes the single phrase, or block of phrases, following it to be obeyed only if the condition is false. (If the condition must contain a comma for some reason, the block form should be used.) Example:

```
unless the red door is closed, say "You could try going east?"
```

"Unless" is clearly unnecessary, but it can be a good way to make the source text easier for humans to read.

As we have seen, there are many different forms of condition in Inform. They usually take a form quite like an assertion sentence, except that they're questions and not statements of fact. For example:

```
if the score is 10, ...
if all of the people are in the Atrium, ...
```

Questions like this are checked by Inform to see if they make sense. The following

doesn't, for instance:

```
if 10 is a door, say "Huzzah!";
```

This produces the baffled reply:

> **Problem.** In the line 'if 10 is a door, say "Huzzah!"', I can't determine whether or not '10 is a door', because it seems to ask if a number is some sort of door.

§11.7 Begin and end

In practice it is not enough to apply "if" to a single phrase alone: we want to give a whole list of phrases to be followed repeatedly, or to be followed only if a condition holds.

We do this by grouping them together, and there are two ways to do this. One is as follows:

```
To comment upon (whatever - a thing):
    if whatever is transparent, say "I see right through this!";
    if whatever is an open door:
        say "Oh look, an open door!";
        if whatever is openable, say "But you could always shut it."
```

Here we group two phrases together under the same "if". Note that the comma has been replaced by a colon, and that the indentation in the list of phrases shows how they are grouped together. In the example above, the source moves two tabs in from the margin; the maximum allowed is 25.

Indentation is the convention used in this manual and in the examples, but not everybody likes this Pythonesque syntax. So Inform also recognises a more explicit form, in which the beginning and ending are marked with the words "begin" and "end":

```
To comment upon (whatever - a thing):
    if whatever is transparent, say "I see right through this!";
    if whatever is an open door
    begin;
        say "Oh look, an open door!";
        if whatever is openable, say "But you could always shut it.";
    end if.
```

(Pythonesque because it's a style popularised by the programming language Python, named in turn after "Monty Python's Flying Circus".)

Examples

172. **Matreshka**
    A SEARCH [room] action that will open every container the player can see, stopping only when there don't remain any that are closed, unlocked, and openable.
    RB 6.6 Looking Under and Hiding

173. **Princess and the Pea**
    The player is unable to sleep on a mattress (or stack of mattresses) because the bottom one has something uncomfortable under it.
    RB 8.4 Furniture

§11.8 Otherwise

We often need code which does one thing in one circumstance, and another the rest of the time. We could do this like so:

```
if N is 2:
    ...
if N is not 2:
    ...
```

but this is not very elegant, and besides, what if the action we take when N is 2 changes N so that it becomes something else?

Instead we use "otherwise":

otherwise if (a condition)

or…

otherwise unless (a condition)

or…

otherwise (a phrase)

or…

else if (a condition)

or…

else unless (a condition)

or…

else (a phrase)

This phrase can only be used as part of an "if …:" or "unless: …", and provides an alternative block of phrases to follow if the first block isn't followed. Example:

```
if N is 2:
    ...
otherwise:
    ...
```

When there is only a single phrase we can use the shortened form:

```
if N is 2, say "Hooray, N is 2!";
otherwise say "Boo, N is not 2...";
```

We can also supply an alternative condition:

```
if N is 1:
    ...
otherwise if N is 2:
    ...
otherwise if N is greater than 4:
    ...
```

At most one of the "…" clauses is ever reached - the first which works out.

If the chain of conditions being tried consists of checking the same value over and over,

we can use a convenient abbreviated form:

if (value) **is:**

This phrase switches between a variety of possible blocks of phrases to follow, depending on the value given. Example:

```
if the dangerous item is:
    -- the electric hairbrush:
        say "Mind your head.";
    -- the silver spoon:
        say "Steer clear of the cutlery drawer."
```

One alternative is allowed to be "otherwise", which is used only if none of the other cases apply, and which therefore guarantees that in any situation exactly one of the blocks will be followed.

```
if N is:
    -- 1: say "1.";
    -- 2: say "2.";
    -- otherwise: say "Neither 1 nor 2.";
```

This form of "if" layout is not allowed to use "begin" and "end" instead of indentation: it would look too messy, and would scarcely be an abbreviation. It is also not allowed to use "unless" instead of "if", because the result would be too tangled to follow.

Example

174. **Numberless**
A simple exercise in printing the names of random numbers, comparing the use of "otherwise if...", a switch statement, or a table-based alternative.
RB 2.1 Varying What Is Written

§11.9 While

The next control phrase is "while", which has the form:

while (a condition):

This phrase causes the block of phrases following it to be repeated over and over for as long the condition is true. If it isn't even true the first time, the block is skipped over and nothing happens. Example:

```
while someone (called the victim) is in the Crypt:
    say "A bolt of lightning strikes [the victim]!";
    now the victim is in the Afterlife;
```

We must be careful not to commit mistakes like the following:

```
while eggs is eggs:
    say "again and ";
```

which, as sure as eggs is eggs (which is very sure indeed), writes out

```
again and again and again and again and again and ...
```

forever. (Inform won't prevent this: we will find out the hard way when the story is played.) While we would probably never write anything so blatant as that, the mistake is all too easy to commit in disguised form. We should never design a loop, as repetitions like this are called, without worrying about if and when it will finish.

As with "if", we can use "begin" and "end" instead of a tabulated layout if we want to --

```
while ...
begin;
    ...
end while.
```

(The "begin" of an "if" must of course match an "end if", not an "end while", and so on.)

Experience shows that it is much more legible to lay out "while" loops as blocks, even in these rare cases when only a single phrase forms the body of the block.

§11.10 Repeat

The other kind of loop in Inform is "repeat". The trouble with "while" is that it's not obvious at a glance when or whether the loop will finish, and nor is there any book-keeping to measure progress. A "repeat" loop is much more predictable, and is more or

less certain to finish.

There are several forms of "repeat", of which the simplest is similar to the old FOR/NEXT loop from the home-computer programming language BASIC, for those with long memories:

repeat with (a name not so far used) **running from** (arithmetic value) **to** (arithmetic value)

or…

repeat with (a name not so far used) **running from** (enumerated value) **to** (enumerated value):

This phrase causes the block of phrases following it to be repeated once for each value in the given range, storing that value in the named variable. (The variable exists only temporarily, within the repetition.) Example:

```
repeat with counter running from 1 to 10:
    ...
```

This, and runs through the given phrases ten times. Within those phrases, a special value called "counter" has the value 1 the first time through, then the value 2, then 3 and so on up to 10. (It can of course be called whatever we like: this is only an example.) The range can be from any kind where ranges make sense - anything on which arithmetic can be done, so for instance

```
repeat with moment running from 4 PM to 4:07 PM:
    ...
```

and also any enumeration:

```
Colour is a kind of value. The colours are red, orange, yellow, green, blue, indigo and violet.

...
    repeat with hue running from orange to indigo:
        ...
```

We are allowed to "nest" loops, that is, to put one inside another.

```
To plot a grid with size (S - a number):
    repeat with x running from 1 to S:
        say "Row [x]:";
        repeat with y running from 1 to S:
            say " [y]";
        say "."
```

If we then write

```
plot a grid with size 5;
```

then the result is

```
Row 1: 1 2 3 4 5.
Row 2: 1 2 3 4 5.
Row 3: 1 2 3 4 5.
Row 4: 1 2 3 4 5.
Row 5: 1 2 3 4 5.
```

Thus the innermost phrase, the say which mentions "y", happens 25 times.

Whenever dealing with numbers in Inform we may need to remember that if the Settings for the project are set to use the Z-machine, the range is restricted to -32768 up to 32767. Repeating with a counter up to exactly 32767 is hazardous, because the counter can never break through this barrier: it's infinity, so far as Inform is concerned, and that can cause the repetitions to go on forever. (On Glulx, numbers can be very much larger.)

Example

175. **Wonka's Revenge**
A lottery drum which redistributes the tickets inside whenever the player spins it.
RB 9.5 Dice and Playing Cards

§11.11 Repeat running through

Inform is not used very much for numerical work, so the kind of repeat loop described in the previous section is not much used. Inform's natural domain is really the world of things and rooms, so the following kind of repeat is much more useful.

repeat with (a name not so far used) **running through** (description of values):

This phrase causes the block of phrases following it to be repeated once for each value matching the description, storing that value in the named variable. (The variable exists only temporarily, within the repetition.) Example:

```
repeat with item running through open containers:
    ...
```

If there are no containers, or they are all closed, the phrases will not be followed at all. Inform will issue a Problem message if the range of the loop may be infinite: for example, it won't allow:

```
repeat with X running through odd numbers:
    ...
```

On the other hand it will allow:

```
repeat with T running through times:
    ...
```

which repeats 1440 times, starting with T at midnight and finishing at 11:59 PM. See the Kinds index for which kinds of value can be repeated through.

As with counting the "number of ..." objects satisfying some property, we can run through a wide variety of possibilities - any description whose range is possible for Inform to search. For example:

```
repeat with dinner guest running through the people in the Dining Room:
    ...
repeat with possession running through things carried:
    ...
repeat with event running through non-recurring scenes which are happening:
    ...
```

The following lists the whereabouts of all men in lighted rooms:

```
repeat with suspect running through the men who are in a lighted room:
    say "[The suspect] is in [the location of the suspect].";
```

One small note of caution: if what the "repeat" loop does is to change the things being repeated through, changing in particular whether items not yet reached will qualify to be repeated through, the results can be unexpected. Rather than writing "repeat with X running through D", it may be safer to try "while there is D (called X)", though note that this will only finish if X is always changed so that it no longer qualifies.

Example

176. **Strictly Ballroom**
    People who select partners for dance lessons each turn.

RB 7.16 Social Groups

§11.12 Next and break

So "repeat" and "while" phrases cause a block of other phrases to be repeated, over and over. The number of repetitions and the flow of "control" has so far been controlled only by the way the original loop was described.

But in fact it's also possible to change this from inside the block being repeated, using these:

next

This phrase can only be used inside a "repeat" or "while" block, and causes the current repetition of the block to finish immediately. That either means the next repetition begins, or (if we are already at the last one) the loop ends too. Example:

```
repeat with X running from 1 to 10:
    if X is 4, next;
    say "[X] ".
```

produces the text "1 2 3 5 6 7 8 9 10 ", with no "4" because the "say" phrase was never reached on the fourth repetition.

In Monopoly terms, "next" is "Advance to Go" rather than "go directly, do not pass Go, do not collect $200" - the next iteration begins with the variable, if there is one, having cleanly moved on to the next value, just as if the loop had been run through in the normal way. ("Next" is called "continue" in a fair number of programming languages, so Inform issues a specific problem message to help people who forget this.)

break

This phrase can only be used inside "repeat", "while" block, and causes both the current repetition and the entire loop to finish immediately. Example:

```
repeat with X running from 1 to 10:
    if X is 7, break;
    say "[X] ".
```

produces the text "1 2 3 4 5 6 ", with nothing after "6" because the loop was broken at that point. The "say" wasn't reached on the 7th repetition, and the 8th, 9th and 10th never happened.

§11.13 Stop

Now that it's possible to define phrases where different things are done in different circumstances, we sometimes want to halt early. This is what "stop" is for.

stop

This phrase causes the current rule to end immediately. It is most often used in the definition of other phrases:

```
To judge the score:
    if the score is 0, stop;
    say "The score is [score in words] more than it was a half-hour ago."
```

In the case when the score is 0, the "stop" ends the phrase immediately, so that the subsequent text is printed only if the score is not 0.

"Stop" can also be used in action rules, though this is not very good style - it's clearer to use "stop the action", which is exactly equivalent.

§11.14 Phrase options

There are sometimes several slightly different ways to perform a given task but which have substantially the same definition. In the following example:

```
To go hiking, into the woods or up the mountain:
    if into the woods, say "Watch out for badgers.";
    if up the mountain, say "Better take your compass.";
    say "You go hiking."
```

…a phrase has been set up which can be used in three ways:

```
go hiking;
go hiking, into the woods;
go hiking, up the mountain;
```

Note that commas must be used to divide these "phrase options" from the rest of the text of the phrase. Within the definition of the phrase, the option's name is a valid condition, and

```
if up the mountain, ...
```

tests whether it is set; we can also test if it is not set using:

```
if not up the mountain, ...
```

A more substantial example from the Standard Rules is given by a phrase used mostly for internal, technical reasons:

list the contents of (object)

This phrase produces a list of all things whose holder is the given object, according to Inform's traditional conventions for room descriptions and inventory listings. Example:

```
list the contents of Marley Wood, as a sentence, with newlines
and including all contents;
```

Where this is possible, it's generally better to use "[list of things in …]" instead, which produces the same result in an acceptable way for the middle of a sentence.

Note that this phrase is allowed to have multiple options specified, whereas "go hiking" above was not: this is because it was defined thus:

```
To list the contents of (something - an object), with newlines, indented, as a sentence, including contents, including all contents, giving inventory information, giving brief inventory information, using the definite article, listing marked items only, prefacing with is/are, not listing concealed items, suppressing all articles and/or with extra indentation:
...
```

The significant difference is the word "and/or" instead of "or", which signals that more than one option can apply at a time.

Example

177. **Equipment List**
   Overview of all the phrase options associated with listing, and examples of how to change the inventory list into some other standard formats.

RB 6.7 Inventory

§11.15 Let and temporary variables

A variable, as we have seen, is a name for a value which changes, though always remaining of the same kind. For instance, if "target" is a number variable (or "number that varies") then it may change value from 2 to 4, but not from 2 to "fishknife".

To make complicated decisions, phrases often need to remember values on a temporary basis. We have already seen this for the counter in a "repeat" loop, which exists only inside that loop, and then is no longer needed.

We can also make temporary variables using "let":

let (a name not so far used) **be** (value)

or…

let (a temporary named value) **be** (value)

This phrase creates a new temporary variable, starting it with the value supplied. The variable lasts only for the present block of phrases, which certainly means that it lasts only for the current rule. Examples:

```
let outer bull be 25;
let the current appearance be "reddish brown";
let the special room be Marley Wood;
```

The kinds of these are deduced from the values given, so that, for instance,

```
say "The outer bull scores [the outer bull in words] when you practice archery in
[special room]."
```

produces

```
The outer bull scores twenty-five when you practice archery in Marley Wood.
```

The variable name should be a new one; if it's the name of an existing one, then the kinds must agree. So:

```
let outer bull be 25;
let outer bull be 50;
```

is a legal combination, because the second "let" simply changes the value of the existing "outer bull" variable to a different number.

let (a name not so far used) **be** (name of kind)

This phrase creates a new temporary variable of the given kind. The variable lasts only for the present block of phrases, which certainly means that it lasts only for the current rule. Example:

```
let inner bull be a number;
```

The variable created holding the default value for that kind - in this case, the number 0. A handful of very obscure kinds have no default values, and then a problem message is produced. Inform also disallows:

```
let the conveyance be a vehicle;
```

because temporary variables aren't allowed to have kinds more specific than "object". (This is a good thing: suppose there are no vehicles in the world?) It's quite safe in such cases to use

```
let the conveyance be an object;
```

instead, which creates it as the special object value "nothing".

Temporary variables made by "let" are only temporarily in existence while a phrase is being carried out. Their values often change: we could say

```
let x be 10;
now x is 11;
```

for instance, or indeed we could "let x be 10" and then "let x be 11". But although we are allowed to change the value, we are not allowed to change the kind of value. The name "x" must always have the same kind of value throughout the phrase to which it belongs, so the following will not be allowed:

```
let x be 45;
now x is "Norway";
```

(The difference between using "let" and "now" here is that "let" can create a new temporary variable, whereas "now" can only alter things already existing: on the other hand, "now" can change many other things as well, whereas "let" applies only to

temporary variables.)

Example

178. **M. Melmoth's Duel**
   Three basic ways to inject random or not-so-random variations into text.

RB 2.1 Varying What Is Written

§11.16 New conditions, new adjectives

We can create new conditions by defining a phrase with "to decide whether" (or equivalently "to decide if"):

```
To decide whether danger lurks:
    if in darkness, decide yes;
    if the Control Room has been visited, decide no;
    decide yes.
```

If the player is indeed in darkness, the decision is "yes" because the "decide yes" stops the process right there. We can now write, for instance,

```
if danger lurks, ...
```

In fact, "danger lurks" is now a condition as good as any other, and can be used wherever a condition would be given. Rules can apply only "when danger lurks", for instance.

yes

or...

decide yes

This phrase can only be used in the definition of a phrase to decide whether a condition holds. It ends the decision process immediately and makes the condition true.

no

or…

decide no

This phrase can only be used in the definition of a phrase to decide whether a condition holds. It ends the decision process immediately and makes the condition false.

We can also supply definitions of adjectives like this. So far, new adjectives have been defined like so:

```
Definition: a supporter is occupied if it is described and something is on it.
```

If we want to give a definition which involves more complex logic, we can use a special form allowing us to make arbitrary decisions. In this longer format, the same definition would look like so:

```
Definition: a supporter is occupied:
    if it is undescribed, decide no;
    if something is on it, decide yes;
    decide no.
```

Here "it" refers to the supporter in question. Note that there are now two colons in this sentence, one after "Definition", the other after the clause being defined. But that apart, it's a phrase like any other: it must end in "yes" or "no" just as the "danger lurks" example must. "Decide no" and "decide yes" are needed so often that they can be abbreviated by leaving out "decide":

```
Definition: a supporter is occupied:
    if it is undescribed, no;
    if something is on it, yes;
    no.
```

Example

179. **Owen's Law**
    OUT always means "move to an outdoors room, or else to a room with more exits than this one has"; IN always means the opposite.
    RB 6.9 Going, Pushing Things in Directions

§11.17 Phrases to decide other things

A condition is a yes/no decision, but we can also take decisions where the result is a value. Suppose we want to create a concept of the "grand prize", which will have different values at different times in play. Each time the "grand prize" is referred to, Inform will have to decide what its value is, and the following tells Inform how to make that decision:

```
To decide which treasure is the grand prize:
    if the Dark Room has been visited, decide on the silver bars;
    decide on the plover's egg.
```

Note that we have to say what kind the answer will be: here it's a kind of thing called "treasure" (which we're supposing has already been created), and as it turns out only two treasures are ever eligible anyway (we're also supposing that the plover's egg and the silver bars are treasures already created, of course). And note also that the phrase must in all cases end with a "decide on …" to say what the answer is:

decide on (value)

This phrase can only be used in the body of a definition of a phrase to decide a value. It causes the calculation to end immediately, with the outcome being the given value, which must be of the kind expected. Example:

```
To decide which number is double (N - a number):
    let D be N times N;
    decide on D.
```

Now that we have "grand prize" created, we can use it just as we would use any other value, so for instance:

```
if taking the grand prize, ...
```

As this is something of a dialect difference between English speakers, "what" and "which" are synonymous here, i.e., we could equally well write something like:

```
To decide what number is the target score: ...
```

(A phrase to decide if something-or-other is exactly the same thing as a phrase to decide a truth state, and indeed, if we want to then we can use "decide on T", where T is a truth state, in its definition. For instance:

```
To decide if time is short:
    if the time of day is after 10 PM, decide on true;
    ...
    decide on whether or not Jennifer is hurried.
```

"Decide on true" is exactly equivalent to the more normally used "decide yes", and of course it is optional. The last line is more interesting since it effectively delegates the answer to another condition.)

Examples

180. **Witnessed 2**
     A piece of ghost-hunting equipment that responds depending on whether or not the meter is on and a ghost is visible or touchable from the current location.
     RB 9.11 Clocks and Scientific Instruments

181. **A Haughty Spirit**
     Windows overlooking lower spaces which will prevent the player from climbing through if the lower space is too far below.
     RB 3.6 Windows

§11.18 The value after and the value before

A point which has come up several times in recent chapters is that enumerated kinds of value have a natural ordering. For example, if we write:

```
Colour is a kind of value. The colours are red, orange, yellow, green, blue, indigo and violet.
```

…then we not only have seven possible values, we have put them into a sequence, in order of their naming. We can't perform arithmetic on colours, of course, but we can perform comparisons on them. Thus "red < yellow" is true, while "green >= violet" is not. (More on comparisons in the chapter on Numbers and Equations, which also covers arithmetic.)

It's also sometimes useful to get at the sequence directly. First, the two ends:

first value of (name of kind) ⇒ *value*

This phrase produces the first-created value of the given kind, which should be an enumeration. Example: if we have

```
Colour is a kind of value. The colours are red, orange, yellow, green, blue, indigo and violet.
```

then "first value of colour" is red.

last value of (name of kind) ⇒ *value*

This phrase produces the last-created value of the given kind, which should be an enumeration. Example: if we have

```
Colour is a kind of value. The colours are red, orange, yellow, green, blue, indigo and violet.
```

then "last value of colour" is violet.

And now how to step forward and back:

(name of kind) **after** (enumerated value) ⇒ *value*

This phrase produces the next-created value of the given kind, which should be an enumeration. Example: if we have

```
Colour is a kind of value. The colours are red, orange, yellow, green, blue, indigo and violet.
```

then "colour after orange" is yellow.

(name of kind) **before** (enumerated value) ⇒ *value*

This phrase produces the previous-created value of the given kind, which should be an enumeration. Example: if we have

> Colour is a kind of value. The colours are red, orange, yellow, green, blue, indigo and violet.

then "colour before blue" is green.

Examples

182. **Entropy**
All objects in the game have a heat, but if not kept insulated they will tend toward room temperature (and at a somewhat exaggerated rate).
RB 10.9 Heat

183. **The Hang of Thursdays**
Turns take a quarter day each, and the game rotates through the days of the week.
RB 4.1 The Passage Of Time

12. Advanced Actions

§12.1 A recap of actions
§12.2 How actions are processed
§12.3 Giving instructions to other people
§12.4 Persuasion
§12.5 Unsuccessful attempts
§12.6 Spontaneous actions by other people
§12.7 New actions
§12.8 Irregular English verbs
§12.9 Check, carry out, report
§12.10 Action variables
§12.11 Making actions work for other people
§12.12 Check rules for actions by other people
§12.13 Report rules for actions by other people
§12.14 Actions for any actor
§12.15 Out of world actions
§12.16 Reaching inside and reaching outside rules
§12.17 Visible vs touchable vs carried
§12.18 Changing reachability
§12.19 Changing visibility
§12.20 Stored actions
§12.21 Guidelines on how to write rules about actions

§12.1 A recap of actions

Actions are impulses to do something, which arise sometimes through typed commands:

```
>examine tapestry
```

and sometimes through "try" phrases occurring in other rules:

```
Before examining the tapestry, try switching the ultraviolet light on.
```

Every action either succeeds or fails, though failure may not be a bad thing (something better may have happened). Besides any rules applied in the source text, actions are subject to basic realism rules. A general rule ensures that actions are rejected if the actor would need to touch something which is out of reach, or see something which is invisible; and a couple of hundred other built-in rules police individual actions. For instance, if the ACTIONS testing command has been used to switch monitoring on, then:

```
>unlock cage with watermelon
[unlocking cage with watermelon]
That doesn't seem to fit the lock.
[unlocking cage with watermelon - failed the can't unlock without the correct key rule]
```

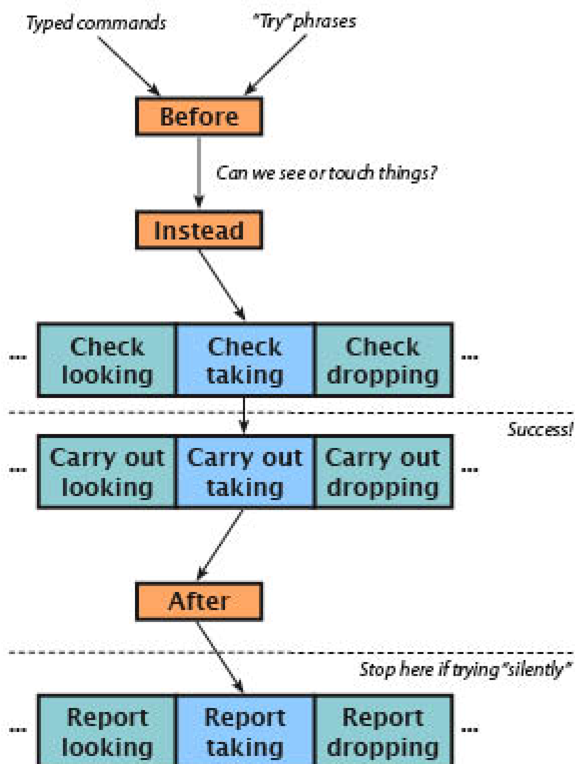
Actions generated by "try" phrases are allowed to run "silently", which means that if nothing out of the way happens and they succeed, then nothing is printed. For instance:

```
Before examining the tapestry: say "(Switching on the lamp first.)"; silently try switching the ultraviolet light on.
```

There are many ways to impose extra rules on actions, and we have seen three main kinds: Before rules, intended so that preliminary activities like the one above can happen before the action is tried; Instead rules, which block or divert the intention, or may cause something spectacularly different to happen; and After rules, which allow for unexpected consequences after the action has taken place.

§12.2 How actions are processed

The following flow chart shows the natural course of events when Inform deals with a new action - a "taking" action in the case drawn. For quite a long time, the action may still fail, and it may be that nothing actually happens: but eventually a deciding line is crossed, and once that happens the action will certainly succeed.



The coloured boxes on this chart represent "rulebooks", that is, collections of rules with a

common purpose. The orange boxes for Before, Instead and After were covered in the Basic Actions chapter, but the blue boxes are new. The orange boxes are where we put rules to handle unusual situations, or unexpected events: special rules to cover the opening of a container which happens to be booby-trapped, or walking through a doorway into a room where a surprise party is about to be sprung.

Blue boxes hold the mundane, everyday rules - the generic ways that particular actions behave. Every action provides these: "Check" rules, to see if it makes sense - for instance, to see that the player is not trying to take his or her own body, or a whole room, or something he or she already has; then "Carry out" rules, to actually do what the action is supposed to do - in the case of taking, to move an object into the player's possession; and finally "Report" rules, which tell the player what has happened - perhaps printing up "Taken."

When we create a new action, we add a new column to the blue rows in this diagram. As we shall see, we can also put new rules into the existing blue boxes: for instance, if we wanted to increase physical realism by forbidding the player to carry more than a certain weight, we would want to add a new "check taking" rule, and this is entirely legal.

In subsequent chapters, we will see ways to intervene at almost every point in the above diagram - from how "Can we see or touch things?" is reckoned, to each and every individual rule in all of these books. Action-processing may be the single most important thing Inform does, so the system is designed to be immensely flexible. On the other hand, that does make it a lot to take in at one look. Newcomers should probably concentrate on "Instead" and "After" as the basic tools for designing the situations turning up routinely in interactive fiction. There are guidelines at the end of this chapter offering advice on which tricks to use when it comes to more complicated needs.

§12.3 Giving instructions to other people

So far, all actions have been carried out by the player: which is fine for exploring the passive world of an empty warehouse, but less good for a drama in which other characters have to be contended with. In fact, an action can be carried out by anybody - by any instance of the "person" kind, that is, which includes all the men, women and animals in the story, and not only the player.

In interactive fiction, players conventionally ask other characters to do something with commands like so:

```
> will, go west
```

Clearly "will, go west" should not produce the same action as "go west", because a

different person will be trying it: this person is called the "actor", and while the actor is ordinarily the player, here it is the character called Will. Inform distinguishes these two actions like so:

```
going west
asking Will to try going west
```

As a result, we can write rules like so:

```
Instead of asking Will to try going west, say "Will scratches his head, baffled by this talk of westward. Is not the ocean without bound?"
```

To write rules like this, we sometimes want to generalise about who is supposed to do the deed. To do this we can refer to "person asked", just as the "noun" stands for whatever noun was typed:

```
Instead of asking somebody to try taking something, say "I don't think we ought to tempt [the person asked] into theft, surely?"
```

So if the player types "Algy, take sandwich", the "person asked" would be Algy; the "noun" would be the sandwich; and there would be no "second noun".

Examples

184. **Virtue**
Defining certain kinds of behavior as inappropriate, so that other characters will refuse indignantly to do any such thing.
RB 7.14 Obedient Characters

185. **Latris Theon**
A person who can accept instructions to go to new destinations and move towards them according to the most reasonable path.
RB 7.13 Traveling Characters

§12.4 Persuasion

"Asking … to try …" actions run through their Before and Instead rules like any other actions, but then (if no rule has intervened) something different happens: Inform has to decide whether the person asked consents to try the action or not. By default, the answer is always no, and text like the following will be printed:

```
> will, go west
Will has better things to do.
```

However, we can intervene to make the answer "yes", using a special kind of rule which produces a yes/no answer. The following examples show how we can give broad or narrow permission, as we choose:

```
Persuasion rule for asking people to try going: persuasion succeeds.
Persuasion rule for asking Will to try going west: persuasion succeeds.
```

Such a rule can either declare that "persuasion succeeds", or that "persuasion fails", or make no decision and leave it to another rule to say. If it decides that persuasion fails, it is also allowed to say something, describing why: in that event, the standard message ("Will has better things to do.") is suppressed. For example,

```
Persuasion rule for asking Will to try going:
    say "Will looks put out, and mutters under his breath.";
    persuasion fails.
```

The following rule, which is really only suitable for testing, makes everybody infinitely obliging:

```
Persuasion rule for asking people to try doing something: persuasion succeeds.
```

Supposing that Will does decide to cooperate, a new action is generated:

```
Will going west
```

and this is then subject to all of the usual action machinery. For instance, we could write a rule such as:

```
Instead of Will going west, say "He runs out into the waves, but soon returns, rueful."
```

So in this case the new action ("Will going west") failed: but the original action, "asking Will to try going west", is still deemed to have succeeded - after all, Will *did* try. To put it more formally, "asking X to try A" succeeds if the persuasion rules succeed, and otherwise fails.

Note also that "Instead of…" rules written for other people will be treated by Inform as failures, even if we write something like

```
Instead of Will pulling the cord:
    say "The bell rings."
```

and thus may produce unsatisfactory results such as

```
>WILL, PULL CORD
The bell rings.

Will is unable to do that.
```

If we wish to write new successful actions for another character, we will need to create appropriate carry out and report rules for them: these will be explained in the sections to follow.

(Finally, note that the mechanism Inform uses to see if we have printed a refusal message of our own, in the event of persuasion rules failing, can be fooled if we write a persuasion rule explicitly ending with a "[paragraph break]" text substitution.)

Examples

186. **The Hypnotist of Blois**
A hypnotist who can make people obedient and then set them free again.
RB 7.14 Obedient Characters

187. **Police State**
Several friends who obey you; a policeman who doesn't (but who takes a dim view of certain kinds of antics).
RB 7.3 Reactive Characters

§12.5 Unsuccessful attempts

Suppose, finally, that Will not only consents to try the action, but it also survives its passage through Before and Instead rules. What happens then? In principle, what happens to Will is exactly what would have happened to the player in his place. For instance:

```
> will, go east
Will leaves to the east.
```

If on the other hand Will's attempt is frustrated because one of the checking rules stops him, then Will's action fails. For instance, if Will tries going northeast but there is no room to northeast, one of the rules checking the "going" action will stop him. We will then see this:

```
> will, go northeast
Will is unable to do that.
```

This is rather a generic message, and we may want something more interesting. We can provide that using yet another special kind of rule:

```
Unsuccessful attempt by Will going: say "Will blunders around going nowhere, as usual."
```

Even that is still a little generic, though, because it treats all of the various ways that "going" can fail as the same. If we have ACTIONS switched on, we can see what goes on behind the scenes when we ask Will to walk into a door:

```
>will, go west
[asking Will to try going west]
[(1) Will going west]
[(1) Will going west - failed the can't go through closed doors rule]
Will blunders around going nowhere, as usual.
[asking Will to try going west - succeeded]
```

(The "(1)" lets us know that a new action is starting during the old one, and before the old one finishes: sometimes we go up to three or four deep, though seldom more in practical cases.) We can now rewrite the "unsuccessful attempt" rule like so:

```
Unsuccessful attempt by Will going:
    if the reason the action failed is the can't go through closed doors rule, say "Will looks doubtful and mumbles about doors.";
    otherwise say "Will blunders around going nowhere, as usual."
```

The value "reason the action failed" is set to whichever checking rule threw out the action which Will tried. The names of these rules try to be self-explanatory - at any rate, those with gnomic names are not useful for this sort of thing, and can be ignored - and can be found out either using ACTIONS or by consulting the Actions index.

Finally, note that "unsuccessful attempt" rules apply only when the person in question is being asked to perform the action by somebody else - as in the examples above.

Example

188. **Generation X**
   A person who goes along with the player's instructions, but reluctantly, and will get annoyed after too many repetitions of the same kind of unsuccessful command.
   RB 7.14 Obedient Characters

§12.6 Spontaneous actions by other people

The player's actions happen not only when he types a command, but can also happen spontaneously as a result of a "try" phrase.

```
try going west
try asking Will to try going west
```

The latter might, of course, result in Will trying going west: or it might not - that depends on the persuasion rules. But as the author, we have the ultimate powers of persuasion, and can make Will act in any way we like, without asking:

```
try Will going west
```

Nobody in the simulated world requested this: it is an impulse felt by Will alone, so that - from the player's point of view - Will is acting spontaneously. The player need not be anywhere nearby, and may never know what happened. Recall that when actions work their way down through the flow-chart, they are stopped before reaching the "report" stage - when the player is told about them - if they are running "silently". This is also where Inform stops an action which is not witnessed by the player.

To repeat a point in the previous section: "unsuccessful attempt" rules do not apply to actions which the author has caused to happen, using "try". When such actions fail, they invoke no special set of rules. Indeed, when "try" causes somebody other than the player to try an action, nothing will be printed to report back on success or failure. If Will can't go west, that's his problem.

Note that the text "try Will going west" involves the actor's name immediately placed next to the action he is to try, which in a very few cases might cause ambiguities. If the actor's name contains a participle like "going" - say, if Will's full name turned out to be Mr Will Going - then we would have to write out the action name in full, using "trying" to clarify matters:

```
try Will Going trying going west
```

Examples

189. **IQ Test**
    Introducing Ogg, a person who will unlock and open a container when the player tells him to get something inside.
    RB 7.15 Goal-Seeking Characters

190. **Boston Cream**
    A fuller implementation of Ogg, giving him a motivation of his own and allowing him to react to the situation created by the player.
    RB 7.15 Goal-Seeking Characters

§12.7 New actions

It is not often that we need to create new actions, but a large work of interactive fiction with no novelty actions is a flavourless dish. Here we shall create an action for photographing things.

```
The Ruins is a room. "You find this clearing in the rainforest oddly familiar." The camera is in the Ruins. "Your elephantine camera hangs from a convenient branch."

Photographing is an action applying to one visible thing and requiring light.
```

In theory that text is already sufficient to make the new action, but what we have so far is rudimentary to say the least. The two qualifications give Inform the useful information that we cannot photograph in the dark, and that we need to be photographing something - not, as in the case of waiting or taking inventory, acting without reference to any particular thing external to ourselves.

The word "visible" here tells Inform that we do not need to be able to touch the thing in question: a line of sight is good enough. These two stipulations were necessary because the default arrangement is that any object must be in touching range, and that most actions can be performed in darkness. (Also, note that if you invent an action which needs to apply to directions like "north" or "south", you need to make this apply to visible things, because the object used inside Inform to represent the idea of "north" can be seen but not touched. So for understanding purposes, "visible thing" is understood as meaning any visible thing or direction: it's more general than "thing", not more specific.)

Occasionally, when writing general rules about actions, it can be useful to find out what the current action's requirements are: the following conditions do what they suggest.

if action requires a touchable noun:

This condition is true if the action being processed is one whose (first) noun is an object which needs to be touchable by the actor. For example, it's true for "taking", but false for "examining".

if action requires a touchable second noun:

This condition is true if the action being processed is one whose second noun is an object which needs to be touchable by the actor. For example, it's true for "putting the brick in the sack", but false for "throwing the brick at the window".

if action requires a carried noun:

This condition is true if the action being processed is one whose (first) noun is an object which needs to be carried by the actor. For example, it's true for "dropping", but false for "taking".

if action requires a carried second noun:

This condition is true if the action being processed is one whose second noun is an object which needs to be carried by the actor.

if action requires light:

This condition is true if the action being processed is one which can only be performed if the actor has light to see by. For example, it's true for "examining", but false for "dropping".

As further examples, here we create "blinking" and "scraping X with Y". Note the use of "it" to indicate that the name of an object should go here.

```
Blinking is an action applying to nothing. Scraping it with is an action applying to two things.
```

The photographing action now exists, but with two provisos: (a) it never happens, because Inform does not know what commands by the player should cause it, and (b) even if it were to happen, nothing would follow, because Inform does not know what to do. (There are no check, carry out or report rules yet.)

The first problem is easily overcome:

```
Understand "photograph [something]" as photographing.
```

We will return to the whole subject of parsing, as this process of understanding the player's commands is called, later. But this gives the gist of it.

See Also

Understand for the full story.

Examples

191. Red Cross
A DIAGNOSE command which allows the player to check on the health of someone.
RB 7.5 Combat and Death

192. Frizz
Liquid flows within containers and soaks objects that are not waterproof; any contact with a wet object can dampen our gloves.
RB 10.2 Liquids

193. 3 AM
A shake command which agitates soda and makes items thump around in boxes.
RB 10.2 Liquids

§12.8 Irregular English verbs

Our three example actions can be recognised in play using the following:

```
Understand "photograph [something]" as photographing.
Understand "blink" as blinking.
Understand "scrape [something] with [something]" as scraping it with.
```

The last of these examples shows why Inform does not risk generating this automatically: English is so full of irregular verbs. Inform could have guessed "blink" and "photograph", but might then have opted for "scrap" instead of "scrape".

Inform does risk automatically generating the past participle of an action. (Many past participles are never needed, so the stakes are lower if Inform gets this wrong.) What usually happens is that the "-ing" is replaced with "-ed", thus photographed, blinked, scraped - but Inform has a dictionary of some 460 irregular exceptions, such as caught, fled, crossbred, taken, woven. So with luck Inform will guess correctly. If not, we can get around this like so:

```
Squicking is an action with past participle squacked, applying to one thing.
```

§12.9 Check, carry out, report

The normal behaviour of an action is specified by its three associated rulebooks - check, carry out and report. In the case of our "photographing" example, these rulebooks will be:

Check photographing. Here, for instance, we need to verify that the player has the camera. If any of our checks fail, we should say why and stop the action. If they succeed, we say nothing.

Carry out photographing. At this stage no further checking is needed (or allowed): the action now definitively takes place. At this point we might, for instance, deduct one from the number of exposures left on the film in the camera, or award the player points for capturing something interesting for posterity. But we should say nothing.

Report photographing. At this stage no further activity is needed (or allowed): whatever effect the action had, it has happened and is now over. All we can do is to say what has taken place.

So far we have not really gone into the business of what rulebooks are, and we don't do so here either - suffice to say that we can now create whatever rules we need:

```
A check photographing rule:
    if the camera is not carried:
        say "You can hardly photograph without a camera, now can you?" instead.
```

In fact, writing "a check photographing rule" is over-formal. We can more simply label our rules like so:

```
Check photographing:
    if we have photographed the noun:
        say "You've already snapped [the noun]." instead.

Report photographing: say "Click!"
```

For the sake of brevity, photography has no interesting consequence (no points to be won, no film to use up), so there are no carry out rules here. Note the way we used the word "instead" once again to stop actions in their tracks.

We can continue to add rules at any point, and a classic thing that happens when testing a new work is that the designer realises there is a case which has not been thought of:

```
Check photographing:
    if the noun is the camera:
        say "That would require some sort of contraption with mirrors." instead.
```

Examples

194. **The Dark Ages Revisited**
An electric light kind of device which becomes lit when switched on and dark when switched off.
RB 3.7 Lighting

195. **Paddington**
A CUT [something] WITH [something] command which acts differently on different types of objects.
RB 10.4 Glass and Other Damage-Prone Substances

196. **Delicious, Delicious Rocks**
Adding a "sanity-check" stage to decide whether an action makes any sense, which occurs before any before rules, implicit taking, or check rules.
RB 9.1 Food

197. **Noisemaking**
Creating a stage after the report stage of an action, during which other characters may observe and react.
RB 7.3 Reactive Characters

§12.10 Action variables

For some complex situations, it can be useful to keep track of a few values throughout the processing of the action. This is not an everyday occurrence: in the Standard Rules, for instance, only two or three out of 90 actions need to do this. But suppose we want to write a more deluxe version of our "photographing" action. This time, rather than having a single thing called the "camera", we will provide a whole range of possible cameras, varying in quality:

```
Photographing is an action applying to one visible thing and requiring light. Understand "photograph [something]" as photographing.

The Studio is a room. Sally is a woman in the Studio. A foam-lined tote bag is in the Studio.

A camera is a kind of thing. A camera has a number called picture quality. The digital SLR camera is a camera in the tote bag. The player carries a camera called the instant one-shot camera. The picture quality of the SLR camera is 10. The picture quality of the one-shot is 2. Definition: a camera is sharp if its picture quality is 5 or more.
```

And we will want the photographing action to have the player use the best-quality camera which comes to hand. We will give the action a variable called the 'camera photographed with', thus:

```
The photographing action has an object called the camera photographed with.
```

Every action's variables must be named differently from those of all other actions, because there are some "before" rules (for instance) which take effect for many different actions, and which might need access to any of their variables. So action variables should be named in a way marking out to which action they belong. The best way to do this is to include the past participle of the action name - just as "camera photographed with" contains the past participle "photographed" of the action "photographing".

This value is created when the action begins, and disappears when the action ends. (If the action should happen a second time before the first time was completed, a second copy of the value is created, leaving the original undisturbed.) When the action begins, the value starts out as something neutral - so if it is a number, it starts out as 0, if a text, it starts out as the blank text "", and so on. Here it is an object, so it starts out as nothing - the value meaning no object at all. But of course we want to give it a value ourselves. We can do that using the "setting action variables" rulebook. For instance:

```
Setting action variables for photographing:
    now the camera photographed with is the sharpest camera which is carried by the actor.
```

The "setting action variables" rulebook is run through before even the before rules, and it has no power to stop or change the action. Its rules should say nothing and do nothing other than to set rulebook variables like this one. Note that it is intended to work for any actor, not only the player: so rather than referring to the player as the performer of the action, we need to write "the actor", as in the example above. (See subsequent sections for more on actors.)

We can now write rules such as:

```
A check photographing rule:
    if the camera photographed with is nothing:
        say "You can hardly photograph without a camera, now can you?" instead.
```

Only rules to do with the photographing action - before, instead, after, check, carry out, or report rules, and so on - are allowed to see the 'camera photographed with' value: it's the private property of the action.

A further elaboration allows us to make rules about photographing neater to write. If we create our variable like so:

```
The photographing action has an object called the camera photographed with (matched as "using").
```

…then we are now allowed to add an optional 'using …' clause onto a description of the action. The clause has to be introduced with a single word: here, it's 'using'. For instance, we could write rules such as

```
Instead of photographing something using the one-shot camera:
    say "But you promised to give this to Sally's nephew."

Check photographing something using the noun:
    say "That would require some sort of contraption with mirrors." instead.

Report photographing something using a sharp camera:
    say "You feel cool and important as the shutter clicks."
```

(This is the method used by the Standard Rules to attach optional clauses such as 'to', 'with' and 'through' to the going action.)

Examples

198. **Removal**
TAKE expanded to give responses such as "You take the book from the shelf." or "You pick up the toy from the ground."
RB 6.8 Taking, Dropping, Inserting and Putting

199. **Further Reasons Why All Poets Are Liars**
The young William Wordsworth, pushing a box about in his room, must struggle to achieve a Romantic point of view.
RB 3.3 Position Within Rooms

200. **The Second Oldest Problem**
Adapting the going action so that something special can happen when going from a dark room to another dark room.
RB 6.9 Going, Pushing Things in Directions

201. **Puff of Orange Smoke**
A system in which every character has a body, which is left behind when the person dies; attempts to do something to the body are redirected to the person while the person is alive.
RB 7.5 Combat and Death

202. **Croft**
Adding special reporting and handling for objects dropped when the player is on a supporter, and special entering rules for moving from one supporter to another.
RB 6.8 Taking, Dropping, Inserting and Putting

§12.11 Making actions work for other people

The "photographing" action now works very nicely when the player does it. But not when others try. Suppose that neither the player, nor Clark Gable, is holding the camera:

```
>photograph clark
You can hardly photograph without a camera, now can you?

>clark, photograph me
>
```

An uncanny silence. What has happened is that the rules written so far are all implicitly restricted to the player only. This is because when we write -

```
Check photographing:
    if the camera is not carried:
        say "You can hardly photograph without a camera, now can you?" instead.
```

the action is "photographing", not "Clark photographing". In the next few sections we shall see how to make the rules work nicely for everybody. This is a little bit harder, so it should be noted right away that in many projects there is no need. In a story which has no other characters who succumb to persuasion, for instance, only the player will ever try the action.

Examples

203. **The Man of Steel**
An escaping action which means "go to any room you can reach from here", and is only useful to non-player characters.
RB 7.15 Goal-Seeking Characters

204. **Trying Taking Manhattan**
Replacing the inventory reporting rule with another which does something slightly different.
RB 6.7 Inventory

205. **Under Contract**
Creating a person who accepts most instructions and reacts correctly when a request leads implicitly to inappropriate behavior.
RB 7.14 Obedient Characters

§12.12 Check rules for actions by other people

If we want to impose the restriction about carrying the camera on other people, we need a rule like the following:

```
Check someone photographing: if the person asked does not carry the camera, stop the action.
```

Implicitly, that "someone" excludes the player. Note that we say nothing in this rule, stopping the action without a word: after all, Clark might well be out of sight when trying this. If he is within sight, then we read:

```
>clark, photograph me
Clark Gable is unable to do that.
```

We saw before that Inform's built-in rules all have handy names (the "can't drop what's already dropped rule", and such), and that these are useful when writing better "unable to…" messages. So for a deluxe version, we end up with:

```
Check someone trying photographing (this is the other people can't photograph without the camera rule): if the person asked does not carry the camera, stop the action.
```

And now, with ACTIONS on, we find that:

```
>clark, photograph me
[asking Clark Gable to try photographing yourself]
[(1) Clark Gable photographing yourself]
[(1) Clark Gable photographing yourself - failed the other people can't photograph without the camera rule]
Clark Gable is unable to do that.
[asking Clark Gable to try photographing yourself - succeeded]
```

which means that we could have, say,

```
Unsuccessful attempt by Clark photographing:
	if the reason the action failed is the other people can't photograph without the camera rule, say "Clark is too suave to be embarrassed. 'Frankly, my dear, I don't have a camera.'";
	otherwise say "Clark tries, and fails, to take a photograph."
```

Examples

206. **Get Axe**
Changing the check rules to try automatically leaving a container before attempting to take it. (And arranging things so that other people will do likewise.)
RB 6.10 Entering and Exiting, Sitting and Standing

207. **Barter Barter**
Allowing characters other than the player to give objects to one another, accounting for the possibility that some items may not be desired by the intended recipients.
RB 7.4 Barter and Exchange

§12.13 Report rules for actions by other people

Report rules for the player's actions are easy to write, and for many actions, they are not much harder for other people either:

```
Report photographing: say "Click!"
Report someone photographing: say "Click! [The person asked] takes a snapshot of [the noun]."
```

But once other people are involved, we have to go to some trouble to get all of the possibilities right. Here is a case which did not immediately occur to the author of the "going" action, for instance:

```
>get in cage
You get into the cage.
>clark, get in automobile
Clark Gable gets into the automobile.
>clark, push cage west
Clark Gable goes west in the automobile, pushing the cage in front, and you along too.

The Lot (in the cage)
In the Lot you can see an automobile (in which is Clark Gable).
```

We said before that report rules are skipped if the action is running "silently", or if the action is one that the player does not witness. But that is also a tricky concept. Inform's doctrine is that you witness an action if you can see any of the actor, the noun or the second noun at either the beginning or the end of the action; except that being able to see a backdrop does not count. Thus if Clark Gable, in Beverly Hills, photographs the Hollywood sign then we do not witness this from Sunset Boulevard merely because we, too, can see the Hollywood sign.

While the report rules for actions by the player must actually report something, report rules for other people's actions are under no such obligation. For instance, if Clark unlocks a door from the other side to the player, then this counts as an action that the player witnesses - and after all, it could be argued that the player should hear the key turning in the lock - but in fact the standard rules for reporting locking choose to say nothing.

Examples

208. **The Man of Steel Excuses Himself**
    Elaborating the report rules to be more interesting than "Clark goes west."
    RB 7.15 Goal-Seeking Characters

209. **Fate Steps In**
    Fate entity which attempts to make things happen, by hook or by crook, including taking preliminary actions to set the player up a bit.
    RB 4.6 Plot Management

§12.14 Actions for any actor

In the previous sections, we created a new action by providing one set of rules for the player and another for anybody else who might try to perform it. These rules began with action descriptions in one of the following forms:

```
Instead of taking a container, ...
Instead of P taking a container, ...
```

The first form implies that the player must be performing the action: the second allows for any person matching P to be the action, except that this person must not be the player. That means that all rules seen so far either affect only the player, or only other people.

This is often convenient, but sometimes we need to set up a complicated action which really does work in the same way for every actor - for instance, the built-in Inform actions provided by the Standard Rules aim to do this. We can write such rules thus:

```
Instead of an actor taking a container, ...
```

Here the rule applies to anyone who tries taking a container, player or not. Inside such a rule, the special value 'the actor' is the person performing the action. For instance, the Standard Rules include this one:

```
Carry out an actor wearing (this is the standard wearing rule):
    now the actor wears the noun.
```

§12.15 Out of world actions

The actions seen so far are all impulses causing the protagonist inside the fictional world to do something, or at least try to. But when the player types "quit" or "save", that is not a

request for anything to happen in the fictional world: it is an instruction to the program simulating that world. In fact, just the same, such requests are treated as actions, but of a special category called "out of world" actions. They do not cause time to pass by, so the turn counter does not advance, nor does this command cycle count as a turn at all; and they are altogether exempt from "Before", "Instead" and "After" rules. Only the player is allowed to try them.

We can also create new out-of-world actions. Suppose we want a dialogue like so:

```
>ROOMS
You have been to 1 out of 8 rooms.
```

Here is a complete implementation:

```
Requesting the room tally is an action out of world.
Report requesting the room tally: say "You have been to [number of visited rooms] out of [number of rooms] room[s]."
Understand "rooms" as requesting the room tally.
```

It is important not to use "out of world" actions for anything affecting what goes on in the fictional world, or realism will collapse, and action-processing may also fail to work in the usual way. "Out of world" actions should be reserved for providing commands like ROOMS, which monitor events rather than participate in them.

Examples

210. **Spellbreaker**
P. David Lebling's classic "Spellbreaker" (1986) includes a room where the game cannot be saved: here is an Inform implementation.

RB 11.2 Saving and Undoing

211. **A point for never saving the game**
In some of the late 1970s "cave crawl" adventure games, an elaborate scoring system might still leave the player perplexed as to why an apparently perfect play-through resulted in a score which was still one point short of the supposed maximum. Why only 349 out of 350? The answer varied, but sometimes the last point was earned by never saving the game - in other words by playing it right through with nothing to guard against mistakes (except perhaps UNDO for the last command), and in one long session.

RB 11.2 Saving and Undoing

§12.16 Reaching inside and reaching outside rules

The flow chart back at the start of this chapter shows that, early on in processing an action (between Before and Instead), Inform asks the question "Can we see or touch things?" This is where it enforces the requirements in the action's definition:

```
Photographing is an action applying to one visible thing and requiring light.
Scraping it with is an action applying to two things.
```

Seeing and touching are two different questions, which Inform answers in different ways. We shall see ways to modify or entirely alter what can be seen using the "deciding the scope of something" activity when we get to the Understanding and Activities chapters, and later in this chapter we will change the definition of touchability. What both have in common is that they are complicated questions, affected by the circumstances. We cannot simply declare that the player can touch a given lever, or can see in a given room: we must arrange for there to be no barriers between the player and the lever, or for there to be a light source in the room.

An example of rules applying to given objects is provided by the way that Inform decides whether the player can reach something or not. For instance, suppose the following:

```
The Laboratory is a room. In the Laboratory is a conical flask. The flask is closed and
transparent. In the flask is an antibumping granule.
```

The player will be able to examine the granule but not to take it, as that would require reaching through glass. Suppose the player does type TAKE GRANULE: then Inform looks for potential barriers between the player and the granule, and of course finds the conical flask. If, as in this case, the thing to be touched is on the inside, then Inform asks the "reaching inside" rules for permission. There are two reaching inside rules built in to Inform:

```
can't reach inside rooms rule
can't reach inside closed containers rule
```

and in fact the second of these rules will cause the taking action to fail, because the conical flask is a closed container. (The other rule has to do with a player in one room able to see another room through, say, a telescope - merely having a line of sight doesn't give the ability to reach into the frame.)

Symmetrically, Inform also has "reaching outside" rules, used if the player is inside something and wants to reach an object in the wider room. (From a bed, probably yes; from a cage, probably no.) This ordinarily contains just one rule:

```
can't reach outside closed containers rule
```

Example

212. **Carnivale**
    An alternative to backdrops when we want something to be visible from a distance but only touchable from one room.
    RB 3.4 Continuous Spaces and The Outdoors

§12.17 Visible vs touchable vs carried

To recap, actions are created like so:

```
Photographing is an action applying to one visible thing and requiring light.
Depositing it in is an action applying to two things.
Taking inventory is an action applying to nothing.
```

Actions can involve up to two different things. We can place additional requirements on any of these things by describing them as a "visible thing", "touchable thing" or "carried thing". (If we simply say "thing" or "things", as in the second example, Inform assumes the requirement to be "touchable".) These three conditions are increasingly strong:

- To be "visible", something needs only to be possible to refer to by the player, which in practice means that it must be visible to the player-character. The noun or second noun produced by any action resulting from a command at the keyboard will always satisfy this minimal condition.

- To be "touchable", the player-character must be able to physically touch the thing in question: this normally means that it must be in the same room, and there must be no physical barriers in between.

- To be "carried", the player-character must (directly) carry the thing in question. (But if the player types a command using an action requiring something "carried", like WEAR HAT, the thing in question - the hat - will sometimes be picked up automatically. This is called "implicit taking", and results in text like "(first taking the top hat)" being printed.)

If an action involves two things, they need not have the same requirement as each other:

```
Waving it at is an action applying to one carried thing and one visible thing.
```

Thus to "wave magic wand at banyan tree", the player must be holding the wand, but need only be able to see the tree.

Note one special case. Requirements on touchability are waived in the case of "try" actions applied to people other than the player where the things they would need to touch are doors or backdrops. (This is a compromise to avoid difficulties arising from the ambiguous locations of such items.)

Examples

213. Eddystone
    Creating new commands involving the standard compass directions.
    RB 3.4 Continuous Spaces and The Outdoors

214. Slogar's Revenge
    Creating an amulet of tumblers that can be used to lock and unlock things even when it is worn, overriding the usual requirement that keys be carried.
    RB 6.3 Modifying Existing Commands

§12.18 Changing reachability

The question of what the player can, and cannot, reach to touch is important in interactive fiction. It contains some of the subtlest ideas in the model world, though they often go unnoticed. For instance, if a key is on a shelf which is part of a closed box, can we reach for the key? This comes down to whether the shelf, described only as "part of" the box, is on the inside or the outside: and in fact, because it cannot know which is the case, Inform allows either. So in general it is best to regard "parts" as being exterior parts, but to avoid having parts on containers that might in the course of play be closed up with the player inside.

We can, if we wish, change the principles of what can be touched by writing new reaching inside or reaching outside rules. Returning to the example of the conical flask:

```
A rule for reaching inside the flask: say "Your hand passes through the glass as if it were
not there, chilling you to the bone."; allow access.
```

(Or this could equally be called "a reaching inside rule for the flask".) More generally, we could give the usual flexible description of what the rule applies to:

```
A rule for reaching inside open containers: say "Your hands seem enigmatically too large
for [the container in question]."; deny access.
```

The "container in question" is the one to which the rule is being applied. Note that a reaching inside rule can "deny access" (stopping with failure), or "allow access" (stopping with success), or neither, in which case the decision is left up to any subsequent rules in

the rulebook to make. If none of them decide, access is allowed.

If it seems possible that these rules will be employed by people other than the player, then we need to write them a little more carefully, and in particular we need to ensure that they print nothing for other people. In the first case below, anybody can reach through the glass; in the second case, only the player cannot reach into open containers.

```
A rule for reaching inside the flask:
    if the person reaching is the player, say "Your hand passes through the glass as if it were not there, chilling you to the bone.";
    allow access.

A rule for reaching inside open containers:
    if the person reaching is the player:
        say "Your hands seem enigmatically too large for [the container in question].";
        deny access.
```

The "person reaching" is, as its name suggests, the person trying to reach through the barrier in question.

Examples

215. **Magneto's Revenge**
Kitty Pryde of the X-Men is able to reach through solid objects, so we might implement her with special powers that the player does not have…
RB 10.10 Magic (Breaking the Laws of Physics)

216. **Waterworld**
A backdrop which the player can examine, but cannot interact with in any other way.
RB 3.4 Continuous Spaces and The Outdoors

217. **Dinner is Served**
A window between two locations. When the window is open, the player can reach through into the other location; when it isn't, access is barred.
RB 3.6 Windows

§12.19 Changing visibility

Ordinarily, Inform has a simple model for visibility: it is either fully light or it is fully dark, and certain actions are impossible in the dark, such as examining something.

We first need to remember that darkness affects what actions are even tried, as far as the player's typed commands go. If the player is in a dark room, and there is a screwdriver on

the floor, the command EXAMINE SCREWDRIVER will not try any action: the screwdriver is not "in scope", which means that the parser thinks the player does not have any means of knowing it exists. (The rules for scope can be modified - see the chapter on Activities.) But let's suppose that the player types EXAMINE BOOK, and is holding the book in question. The book is now "in scope", so the action "examining the book" is tried.

Some actions require light to be present, and "examining" is one of those. So Inform consults the visibility rules to see if it can go ahead. By default, there is only one visibility rule, which says "yes" in the light and "no" in darkness. Here, though, we create another one:

```
Visibility rule when in darkness:
    if examining the book:
        say "You have to squint. Still...";
        there is sufficient light;
    there is insufficient light.
```

A visibility rule must always conclude "there is sufficient light", or "there is insufficient light", or else do nothing and leave it to other rules to decide.

It is a possibly unexpected fact that "looking" does not require light, but instead behaves differently in darkness - it prints a pseudo-room-description such as

```
Darkness
It is pitch dark, and you can't see a thing.
```

instead of printing the description of the player's current room. This means that the "looking" action is unaffected by visibility rules. All the same, what "looking" does in the dark can be changed by using the two activities "printing the name of a dark room" and "printing the description of a dark room" (see the Activities chapter for details).

Example

218. **Flashlight**
    Visibility set so that looking under objects produces no result unless the player has a light source to shine there (regardless of the light level of the room).
    RB 6.6 Looking Under and Hiding

§12.20 Stored actions

As we have seen, to describe an action fully takes a complicated little bundle of information - we need to know what is to be done, who will do it, and what it will be done to. There are times when we would like to remember an action and look back on it later

(perhaps many turns later, after many other actions have taken effect) - but this is not easy to do with only the techniques we have seen so far. There are quite a few cases to get right, and it would be easy to not store quite enough of the details.

Fortunately, Inform provides a kind of value called "action" which can do all of this automatically. (In older versions of Inform this was called "stored action", but the word "stored" is now unnecessary, and makes no difference.) As with most other kinds of value, actions can be held in variables, "let" values, properties or table columns. For example:

```
The best idea yet is an action that varies.
```

creates a variable called "the best idea yet" which holds an action.

This will normally be created holding the default value - the player waiting. We really only have two ways to make more interesting actions. One is by typing them out explicitly, like so:

```
now the best idea yet is pushing the button;
```

Here "pushing the button" is a constant of the kind "action", so it goes into happily into "best idea yet" in the same way that a number like 3 could go into a number that varies. The action must be specific in every respect, so "taking something" or "doing something" will not work - "taking something" is really a general description of many possible actions, not an action in its own right.

The other way to produce a useful action is:

current action ⇒ *action*

This phrase produces the action currently being processed as a value - it literally stores the action, and remembers, if necessary, the exact wording of the player's command at the time it was stored - so that even actions arising from commands like LOOK UP X100 IN THE CODE BOOK can be stored faithfully. Examples:

```
let the present whim be the current action;
say "How you would like to be [current action].";
```

This only makes sense if an action is currently going on, so it shouldn't be used in "every turn" rules, for instance.

So much for making actions: now for making use of them. The first obvious idea is to store up an action for several turns and then have it take effect later. That's easily done: just as we can "try" any action written out explicitly, so we can also try a stored one. The phrase to do this has exactly the same wording either way, since it does the same thing either way.

But actions can still be useful even if we never intend to try them. For one thing, we can say them, and this produces a fairly natural description of what the action is:

```
Before doing something in the presence of the bearded psychiatrist: say "'Zo, the subject
vishes to engage in [the current action]. Zis is very interesting.'"
```

will produce text such as:

```
"So, the subject vishes to engage in rubbing the fireman's pole. Zis is very interesting."
```

One of Inform's most convenient features is its ability to test if the action being processed matches vague or complicated descriptions of whole classes of actions. For example,

```
if the best idea yet is taking something, ...
```

works even though "taking something" is not a single action; it's a description which could apply to many different actions (taking a box, taking a ball, and so on). What Inform tests is whether the "best idea yet" value, a single action, fits this description or not. We can be even vaguer:

```
if the best idea yet is doing something to the lever, ...
```

Just occasionally, this can lead to ambiguities. For instance,

```
if the current action is wearing something, ...
```

fails because Inform thinks "wearing" is meant in the sense of the current action having clothes on, so it produces a problem message. To avoid this, simply write:

```
if the current action is trying wearing something, ...
```

which can't be misunderstood. Something else to be aware of is that the terms "actor", "noun" and so on will refer to that action: for instance, in

```
if the best idea yet is taking the noun, ...
```

"noun" here refers to the noun in "best idea yet", not to its meaning outside of this phrase (if indeed it has such a meaning).

When dealing with actions, we sometimes want to know what they are dealing with. We can extract this information using the following phrases:

action name part of (action) ⇒ *action name*

This phrase produces the action name part of an action. Example: suppose the current actor is Algy, who is throwing the brick at Biggles. Then

```
action name part of the current action = throwing it at action
```

noun part of (action) ⇒ *object*

This phrase produces the (first) noun of an action. Example: suppose the current actor is Algy, who is throwing the brick at Biggles. Then

```
noun part of the current action = the brick
```

If the noun is something other than an object, this produces just "nothing", the non-object.

second noun part of (action) ⇒ *object*

This phrase produces the second noun of an action. Example: suppose the current actor is Algy, who is throwing the brick at Biggles. Then

```
second noun part of the current action = Biggles
```

If the second noun is something other than an object (for instance for the command SET DIAL TO 3417 it would be the number 3417), this produces just "nothing", the non-object.

actor part of (action) ⇒ *object*

This phrase produces the person who would be carrying out the action if it were being tried. Example: suppose the current actor is Algy, who is throwing the brick at Biggles. Then

```
actor part of the current action = Algy
```

The following phrase is a convenient shorthand form:

if (action) **involves** (object):

This condition is true if the object appears as any of the actor, the noun or the second noun in the action. Example:

```
if the current action involves Algy
```

would be true for "give revolver to Algy", "Algy trying flying the Sopwith Camel", "examine Algy" and so on, but false for "ask Raymond about secret airfield".

action of (an action) ⇒ *action*

This phrase is now seldom needed. It produces a literally typed action as a value. Example:

```
now the best idea yet is the action of pushing the button;
```

Nowadays in most contexts we can just type "pushing the button" as a value, and that will work fine, so this phrase is retained only to keep old code working.

Examples

219. **Bosch**
    Creating a list of actions that will earn the player points, and using this both to change the score and to give FULL SCORE reports.
    RB 11.4 Scoring

220. **Cactus Will Outlive Us All**
    For every character besides the player, there is an action that will cause that character to wither right up and die.
    RB 6.14 Remembering, Converting and Combining Actions

221. **Actor's Studio**
    A video camera that records actions performed in its presence, and plays them back with time-stamps.
    RB 9.12 Cameras and Recording Devices

222. **Anteaters**
    The player carries a gizmo that is able to record actions performed by the player, then force him to repeat them when the gizmo is dropped. This includes storing actions that apply to topics, as in "look up anteater colonies in the guide".
    RB 6.14 Remembering, Converting and Combining Actions

§12.21 Guidelines on how to write rules about actions

Looking at the action-processing diagram, there seem to be a bewildering number of ways to intervene. For instance, suppose it must be fatal to pick up a land mine. All six of the following rules would do the business:

```
Before taking the land mine: end the story saying "Bang!"
Instead of taking the land mine: end the story saying "Bang!"
Check taking the land mine: end the story saying "Bang!"
Carry out taking the land mine: end the story saying "Bang!"
After taking the land mine: end the story saying "Bang!"
Report taking the land mine: end the story saying "Bang!"
```

So which should we use? Of course, we could decide that it really doesn't matter: what works, works. But it is a good idea to play along with the conventions used by Inform, if only because that will make our rules interact better with each other and with rules by other people which we may someday want to borrow. So this chapter ends by offering a few guidelines. Let us suppose that we have some effect which we want to achieve.

1. Are we just trying to correct the player's typing?

For instance, responding to the command "STEAL GOLDEN EAGLE" with a reply like "To steal something, just try to TAKE it." It is bad style to make a special action for this, which does nothing except to print up this text: better is to use the "Understand … as a mistake" technique, which will come up in the chapter on Understanding.

2. Does the effect apply only to a particular situation, or is it a general phenomenon?

In other words, does the effect apply only to particular people, things or places, or is it a generic rule of play? In the case of the land mine, this is an easy question to answer: it is a unique situation. On the other hand, stopping the player from carrying unduly heavy weights would be a generic rule of play.

Rules like the one saying that photography is only possible if one holds the camera are, by convention, also counted as generic rules of play: they are not really special rules about the camera, but apply to all possible acts of photography anywhere, so are actually generic.

Sometimes we can choose our own answer to this question, and go either way. Suppose we want a certain place to be muddy-floored, affecting things that happen there. One way would be to write exceptional rules applying to that one room. But we could alternatively create a general concept of muddiness ("A room can be clean or muddy. A room is usually clean.") and then regard the new behaviour as being a set of generic rules applying in muddy rooms. We could then, of course, create a second muddy room with much greater ease, or transplant these rules to other works and have muddy rooms in those too.

3. Particular situations: use Instead or After (or sometimes Before).

The next question is: does the effect kick in after the hoped-for action has taken place, or not? In the case of the land mine, to answer that means deciding whether we think the detonator is sensitive to the slightest touch - in which case the explosion would happen at the first touch, and should be in an "Instead" rule - or whether one must actually pick up and disturb the mine - in which case an "After" should be used.

That leaves us a choice of two rulebooks if the effect takes place when the merest impulse towards the action is felt: "Before" and "Instead". Which to use? In cases of doubt, choose "Instead". But if the effect is intended to absolutely suppress all such impulses - for instance, in a silent examination room there must be no talking - then "Before" might be more appropriate. We could imagine that someone about to say something first has a mental impulse to speak, then opens his mouth so that it becomes visible to others that some talking is about to go on, and finally utters words. Here are three possible responses:

"You cannot contemplate breaking this smothering silence." (*Before*)
"The invigilator stares you down through her horn-rimmed glasses." (*Instead*)
"Everyone turns, appalled, as the silence is broken like the surface of a swimming pool by a falling elephant." (*After*)

4. Generic situations: work with action rules (or sometimes Before).

(a) If the effect takes place only when an action is definitely being tried, then we should use one of the action's three rulebooks: check, carry out or report. Check rules should do nothing, and should say nothing either unless they block the action (in which case, they should say why).

Carry out rules must not block the action - it is too late for that - and should not say anything - that hasn't happened yet. (There are a few exceptions: if the action is to look at something, then carrying it out is in a sense the same thing as reporting it. But in all cases of doubt, a carry out rule should say nothing.) Adding a carry out rule to an existing action can make it do something extra.

Report rules must neither block the action nor do anything. Working with new report rules is a way to make more natural-seeming, or more informative, messages appear. For instance, an effect where we want to be able to see through a door when it is first opened ("You open the panelled door, through which is the Board Room.") would be a case for a report rule.

In all cases, it is good style to write check, carry out or report rules in such a way that they could be used in other works too, or in situations that could conceivably have happened in this one (even if in fact it never does). We may one day want to put our new rules into an extension to be used by other people or in other projects, after all.

In this chapter, we have only seen the addition of new rules. We could add a new "check taking" rule, for instance, with the techniques seen so far. But what if the effect we want is not a matter of adding a rule but taking away, or restricting the applicability, of an existing one? In that case, we will need to say that the rule "does nothing" under certain circumstances (see the Rulebooks chapter). The check, carry out and report rules for all of the built-in actions are named, and they appear in the Actions index.

(b) If the effect takes place to divert or supplement an action, before that action actually takes place, then this should be done with a "Before" rule. This is the biggest practical use of "Before" rules: to try other actions, either instead or as well as the one just getting under way.

For example, if we want an automatic mechanism to try opening a container before taking something inside it, that would be a classic case for "Before". Indeed, that is the

only way it could work - "Before" rules have a chance to get in before the touchability conditions are tested.

If we wanted a special "stealing" action for the act of taking another person's possessions, we might want to divert any taking action for such items into our new "stealing" action - that too would be a "Before". This would ensure that any "Instead" rules to do with taking do not apply.

5. Changing the behaviour of out-of-world actions.

Remember that "Instead", "Before" and "After" do not apply: so use "check" rules to forbid certain out-of-world activities, or specify that their rules do nothing (see the Rulebooks chapter).

Finally…

These are only guidelines. The system is designed to be flexible in order to give the author the widest possible range of options, and nobody should feel ashamed of making use of them.

13. Relations

§13.1 Sentence verbs
§13.2 What sentences are made up from
§13.3 What are relations?
§13.4 To carry, to wear, to have
§13.5 Making new relations
§13.6 Making reciprocal relations
§13.7 Relations in groups
§13.8 The built-in verbs and their meanings
§13.9 Defining new assertion verbs
§13.10 Defining new prepositions
§13.11 Indirect relations
§13.12 Relations which express conditions
§13.13 Relations involving values
§13.14 Relations as values in their own right
§13.15 Temporary relations
§13.16 What are relations for?

§13.1 Sentence verbs

Descriptions of things - "open door", "people in the Drawing Room" - have already had a whole chapter to themselves. But descriptions are only half of the story of Inform's highly flexible language for talking about places, things and circumstances: this chapter is the other half, and is about the "sentence". Of course all text is made up of sentences, but Inform has a more specific meaning than that. Consider the following pieces of source text:

```
The mouse is in the teapot.

Every turn when the mouse is in the teapot, say "A tail hangs out of the spout."

Instead of taking the mouse:
    say "The mouse slips from your hand and disappears into the teapot!";
    now the mouse is in the teapot.
```

What these three extracts have in common is the sentence "the mouse is in the teapot". Such a sentence can be used in three different ways: to declare the original state of the world, to ask during play if the world currently has that state, or to change things during play so that it does.

Actually, though, only definite sentences about the present can be used in all three ways. A vague instruction like

```
now Mr Darcy can see the mouse;
```

will fail, because there are so many ways in which Darcy might be able to see the mouse that Inform has no way to know how to arrange matters. And this by contrast is not merely difficult but impossible:

```
now Mr Darcy has never seen the mouse;
```

Which cannot be arranged because the past cannot be changed.

Verbs also turn up inside the more complicated descriptions. For instance,

```
things which are in the teapot
people who can see the mouse
```

are both descriptions, not sentences, but they contain "to be" and "to be able to see" respectively.

This chapter is about the verbs which can be used in sentences and descriptions. Inform involves many other features which use verbs - the action "taking the mouse" and the phrase "end the story" both use forms of verbs (to take and to end) - but this chapter has nothing to do with them: so for the sake of clarity, we will call verbs that occur in sentences "sentence verbs".

§13.2 What sentences are made up from

A sentence consists of two nouns with a verb between them. Usually, the two nouns are descriptions, as in:

```
Mr Collins is in a lighted room.
```

Here "Mr Collins" and "a lighted room" are descriptions. But there are sentences where one or both of the nouns is a value of some other kind. For instance, in

```
if the score is greater than 10, ...
```

the sentence "the score is greater than 10" consists of two number values ("the score" and "10") connected by a verb part ("is greater than").

This chapter is about getting the most out of sentences by defining new verbs to express ideas not already built in to Inform. Before we can define a new sentence verb, however, we must first look at the meanings of verbs: which Inform calls "relations".

Example

223. **Formal syntax of sentences**
A more formal description of the sentence grammar used by Inform for both assertions and conditions.

RB 1.4 Information Only

§13.3 What are relations?

Relations are what sentences express. They are yes/no questions about pairs of things: for example, to say that the coin is in the purse is to say that a particular relation ("being in") is true about a specific pair of things (the coin, the purse). It is neither a fact about the coin nor about the purse, but about the two together.

Inform comes with a number of relations built in, almost all of which have been used in previous chapters already. The following table names some of the more useful ones, giving examples of sentences to bring them about:

```
containment relation - The coin is in the purse.
support relation - The coin is on the table.
incorporation relation - The coin is part of the sculpture.
carrying relation - The coin is carried by Peter.
wearing relation - The jacket is worn by Peter.
possession relation - if Mr Darcy has a rapier...
adjacency relation - The Study is east of the Hallway.
visibility relation - if Darcy can see Elizabeth...
touchability relation - if Darcy can touch Elizabeth...
```

These relation names do not trip off the tongue, but they relatively seldom need to be referred to.

The same meaning can often be expressed by using several different verbs, or using the same verb in several different ways, as in the following examples:

```
The coin is in the purse.
The purse contains the coin.
The coin is contained by the purse.
```

all of which boil down to saying that the coin and purse satisfy the containment relation. Because of that, *relations are not the same as verbs*. To create a new idea, we will need first to create a new relation, and only then can we set up a verb which allows us to talk about that relation.

§13.4 To carry, to wear, to have

Inform has altogether five mutually exclusive ways in which one thing can be physically joined to another one:

```
containment relation - The coin is in the purse.
support relation - The coin is on the table.
incorporation relation - The coin is part of the sculpture.
carrying relation - The coin is carried by Peter.
wearing relation - The jacket is worn by Peter.
```

This is why we cannot have

```
The coin is on the table.
The coin is part of the table.
```

simultaneously, and it is a rare exception to the general rule that having one relation does not affect having another.

But there is also a sixth relation used in Inform for these meanings: the possession relation, which is the meaning of the verb "to have". At first sight this looks the same as the carrying relation, but in fact it is a convenient shorthand for "carrying or wearing", provided for conditions rather than assertions:

```
if Mr Darcy has a wet shirt ...
```

will be true during play if he is either carrying or wearing the shirt.

Still another relation exists which can be tested, but not declared to be true or false: the concealment relation, which is the meaning of the verb "to conceal". So we can ask:

```
if Mr Darcy conceals a fob watch ...
```

Examples

224. **Celadon**
    Using the enclosure relation to let the player drop things which he only indirectly carries.
    RB 6.8 Taking, Dropping, Inserting and Putting

225. **Interrogation**
    A wand which, when waved, reveals the concealed items carried by people the player can see.
    RB 10.10 Magic (Breaking the Laws of Physics)

§13.5 Making new relations

We can create new relations like so:

```
Loving relates various people to one person.
```

Every relation has a name which ends with the word "relation", and in this case the name is "loving relation". While the name is often just two words long, as here, it doesn't have to be:

```
Adept sensitivity relates one person to one vehicle.
```

makes the "adept sensitivity relation". (The limit is 32 words.)

In such a definition, we have to say what kind of thing appears on the left and right of any relation, and also whether "one" or "various" possibilities can exist. In the example

```
Loving relates various people to one person.
```

what we are saying is that only people love; that they only love people; and that each person loves only one other person (at any given moment).

The "various" part comes in because, for instance, we might have:

Verenka *loving relation* Stankevich
Liubov *loving relation* Stankevich

so that various people (Verenka and Liubov, to name but two) love one person (Stankevich). But we are forbidding anyone to love two other people at the same time: Stankevich must decide which of them to love, or pick someone else, or no-one at all.

Similarly, we would not allow

```
Liubov *loving relation* Belinsky
```

It is sometimes convenient to give a name to the other side of a relationship, so to speak. We might imagine:

```
Pet-ownership relates various animals to one person (called the owner).
```

It would then make sense to talk about "the owner of Loulou", and we could have phrases like "now Flaubert is the owner of Loulou" or "if the owner of Loulou is a woman…" and so forth. This, however, would not be allowed:

```
Pet-ownership relates various animals (called the pet) to one person.
```

because "the pet of Flaubert" would be ambiguous: he might have owned dozens.

§13.6 Making reciprocal relations

The relationships described in this chapter so far are by no means always reciprocated. For instance, if a stone is on a table, then it is never true that the table is also on the stone. And the question may not even be meaningful to ask. If Peter wears a jacket, the jacket does not even have the possibility of wearing Peter.

But sometimes we do want a relation which works both ways equally well. These are simple to set up:

```
Meeting relates people to each other.
```

The effect is that various people know various other people, and this is always reciprocated. If Daisy knows Sophie then, automatically, Sophie knows Daisy. This even-handedness is maintained throughout play, so that whatever changes are made it is always true that if A knows B then B knows A.

And similarly for a reciprocal relation between one and another:

```
Marriage relates one person to another.
```

In this case, we can again give a name to the partner under a relation:

```
Marriage relates one person to another (called the spouse).
```

and now, for instance, we may have that the spouse of John is Yoko and the spouse of Yoko is John.

Since many of these examples have involved people, it might be worth mentioning again that any kind can be involved, not just the "person" kind.

Example

226. **Four Cheeses**
    A system of telephones on which the player can call distant persons and have conversations.

RB 9.10 Telephones

§13.7 Relations in groups

Finally, there is a kind of relation which binds even more strongly.

```
Nationality relates people to each other in groups.
```

This is a kind of relation which divides people up: we might wish to have all the Icelandic people related to each other, all the Peruvians to each other, and so on. If there were a Pacific island called Informia with one inhabitant, then that person would be related only to himself. As time goes by, we could imagine people emigrating, and so on, so that these groupings would switch: perhaps everyone would leave Belgium and, for a while, there would be no Belgian nationals at all.

The testing command RELATIONS prints out the current state of all the relations created in the source code. For instance:

```
>relations
Overlooking relates various rooms to various rooms:
    The Pub >=> the Garden
    The Garden >=> the Shrubbery
    The Shrubbery >=> the Sundial Plot
Friendship relates people to each other:
    Mr Wickham <=> Sophie
    Charlotte <=> Sophie
Marriage relates one person to another:
    Mr Wickham == Sophie
```

That can produce a lot of output. To see only a single relation, or to see it at some intermediate point in a calculation, there's also a testing phrase:

show relation (relation of values to values)

This phrase is for testing purposes only. It shows the current state of the named relation, that is, it shows which values relate to which other ones, where it's possible to do this in any sensible way.

But this is a phrase - not a typed command.

Examples

227. **Transmutations**
A machine that turns objects into other, similar objects.
RB 10.10 Magic (Breaking the Laws of Physics)

228. **Otranto**
A kind of rope which can be tied to objects and used to anchor the player or drag items from room to room.
RB 10.6 Ropes

§13.8 The built-in verbs and their meanings

It is all very well to define new relations, but this does nothing if there is no way to assert that they are true, or to ask whether they are true or false. That requires a verb: in fact, a relation is nothing more than what Inform uses as the "meaning" of a verb. The assertion verbs built in to Inform have the following built-in relations as their meanings:

Verb - Relation
to be - equality relation
to have - possession relation
to contain - containment relation
to support - support relation
to carry - carrying relation
to wear - wearing relation
to incorporate - incorporation relation

Two of Inform's built-in relations are expressed using prepositions instead:

Preposition - Relation
to be part of - (reversed) incorporation relation
to be adjacent to - adjacency relation

It would be easy to make verbs for these if we wanted ("to adjoin", say) using the techniques of the next section.

The verb *to be* is grammatically different from any other, and its meaning is too complicated to be fully expressed by any one relation. A great deal of the Inform program is given over to its "meaning", which we are not allowed to change or imitate. The "equality relation" is simple enough, and is the one implied by conditions like

```
if the score is 20, ...
```

but *to be* can have more complicated implications - "if Mr Wickham is hungry" clearly doesn't test whether two quantities are equal. Fortunately the other verbs are much simpler.

There are a few other built-in verbs, as can be seen in the Index, but these are mostly for experts only. For example:

Verb - Relation
to mean - meaning relation
to provide - provision relation

"To mean" can be used to make new verbs, as we'll soon see. Provision is to do with whether something can have a given property: for example, "if R provides the property lighted" tests whether R is able to have this property, not whether it actually has it at the moment.

§13.9 Defining new assertion verbs

Here is an example definition of a new verb:

```
The verb to sport means the wearing relation.
```

Once this is done, we can write the assertion

```
Mr Wickham sports a Tory rosette.
```

which will do the the same thing as

```
Mr Wickham wears a Tory rosette.
```

because both verbs have the same relation as their meaning.

Earlier versions of Inform needed to be told how to make other parts of the verb, but that's rarely true now. Just writing:

```
The verb to sport means the wearing relation.
```

is enough for Inform to understand "he sports", "they sport", "he sported", "it is sported", "he is sporting", "he had sported" and so on. It works with irregular verbs, too; it has a very comprehensive dictionary. But it's legal to spell out the conjugation if need be:

```
The verb to sport (he sports, they sport, he sported, it is sported) implies the knowledge relation.
```

Occasionally it's convenient to have the relation the other way around. For instance:

```
The verb to grace means the reversed wearing relation.
```

With that defined, these two sentences have identical meanings:

```
Mr Wickham sports a Tory rosette.
A Tory rosette graces Mr Wickham.
```

Reversed in this sense means that the things related - the subject and object of the verb - are the other way round.

The Phrasebook index contains all the verbs associated with assertions, in the Verbs section. When we add new verbs to our source, those will appear in the Phrasebook as well.

The verbs above ("to grace", "to sport") are short ones, but we're free to make them longer than that. For example:

```
The verb to cover oneself with means the wearing relation.

Peter is covering himself with a tent-like raincoat.
```

Here we have "to cover oneself with", four words long; the limit is 29.

Examples

229. **Unthinkable Alliances**
    People are to be grouped into alliances. To kiss someone is to join his or her faction, which may make a grand alliance; to strike them is to give notice of quitting, and to become a lone wolf.
    RB 7.16 Social Groups

230. **The Unexamined Life**
    An adaptive hint system that tracks what the player needs to have seen or to possess in order to solve a given puzzle, and doles out suggestions accordingly. Handles changes in the game state with remarkable flexibility, and allows the player to decide how explicit a nudge he wants at any given moment.
    RB 11.3 Helping and Hinting

§13.10 Defining new prepositions

The term preposition is used here, a little loosely, to mean anything which we add to the verb *to be* in order to talk about some relation or other. We have seen many examples already, such as:

```
To be in - The ball is in the box.
To be part of - The lever is part of the slot machine.
```

These are defined just the way verbs are. Compare the following:

```
Suspicion relates various people to one person.

The verb to suspect means the suspicion relation.

The verb to be suspicious of means the suspicion relation.
```

The result of this is that

```
Hercule Poirot suspects Colonel Hotchkiss.
Hercule Poirot is suspicious of Colonel Hotchkiss.
```

are exactly equivalent, and so are these two descriptions:

```
somebody who suspects Colonel Hotchkiss
somebody suspicious of Colonel Hotchkiss
```

While most prepositions are short ("in", "part of", "suspicious of"), they're free to be

longer if need be ("inordinately far away from"): the limit is 30 words, which should be ample.

We can also define verbs as auxiliaries, like so:

```
The verb to be able to approach means the approachability relation.
```

Now we can ask if Poirot "can approach" Hotchkiss, and so on.

Examples

231. **The Abolition of Love**
   A thorough exploration of all the kinds of relations established so far, with the syntax to set and unset them.
   RB 7.16 Social Groups

232. **Swerve left? Swerve right? Or think about it and die?**
   Building a marble chute track in which a dropped marble will automatically roll downhill.
   RB 10.5 Volume, Height, Weight

233. **Beneath the Surface**
   An "underlying" relation which adds to the world model the idea of objects hidden under other objects.
   RB 6.6 Looking Under and Hiding

234. **Bogart**
   Clothing for the player that layers, so that items cannot be taken off in the wrong order, and the player's inventory lists only the clothing that is currently visible.
   RB 9.3 Clothing

§13.11 Indirect relations

We have already seen, in the chapter on Descriptions which is a forerunner of this one, that Inform provides not only "adjacent" as a way of seeing if one room is directly connected to another, but also "the best route from A to B", which allows us to see if any sequence of moves connects them.

Something similar - in fact, simpler - is allowed for any relation between objects. Suppose we would like to go sledging: we can go downhill, but not up. Some quite distant places may be reachable, while others close by may not be, even if lower than us, because they would involve climbing again at some point. The following would implement this:

```
Overlooking relates various rooms to various rooms.

The verb to overlook means the overlooking relation.

The Garden overlooks the Shrubbery. The Folly overlooks the Garden. The Shrubbery overlooks the Sundial Plot. The Old Ice House overlooks the Garden.

After looking:
    say "This wintry vantage point overlooks [the list of rooms overlooked by the location].";
    let the way be the next step via the overlooking relation from the location to the Sundial Plot;
    if the way is a room, say "To sledge downhill to the Sundial, aim for [the way].";
    otherwise say "It is not possible to sledge downhill to the Sundial."
```

Here we're making use of:

next step via (relation of values to values) **from** (object) **to** (object) ⇒ *object*

This phrase tries to find a shortest route between the two given endpoints, using the given relation of objects to determine single steps. Example:

```
next step via the overlooking relation from the Folly to the Chinese Lake
```

The result is the special object value "nothing" if the two endpoints are the same or if no route exists.

number of steps via (relation of values to values) **from** (object) **to** (object) ⇒ *number*

This phrase tries to find the length of a shortest route between the two given endpoints, using the given relation of objects to determine single steps. Example:

```
number of steps via the overlooking relation from the Folly to the Chinese Lake
```

The result is 0 if the two endpoints are the same, or -1 if no route exists.

Another example would be the "six degrees of separation" game, where it is claimed that any two people on Earth are connected by a sequence of up to six acquaintances. In an Inform implementation, we might talk about "the next step via the friendship relation from George Bush to Saddam Hussein", for instance, a phrase likely to evaluate to Donald Rumsfeld, and then

```
the number of steps via the friendship relation from George Bush to Saddam Hussein
```

would be… but that would be telling.

As with route-finding through the map, finding "the next step via" a relation can be slow. For instance, suppose we have dozens of articles of clothing all partially revealing each other, connected by two relations - overlying and underlying. Then "the next step via" these relations allows us to establish what can be worn on top of what else. If we need to calculate this often, and there are enormous wardrobes of clothes to choose from, speed starts to matter.

Once again there is a choice of algorithms: "fast" and "slow", where "fast" needs much more memory. To make route-finding for a given relation "fast", we have to declare it that way:

```
Overlying relates various garments to various garments with fast route-finding.
Overlapping relates various garments to each other with fast route-finding.
```

Otherwise, the "slow" method will be used.

This "with fast route-finding" note can only be added to various-to-various relations. (Although route-finding through various-to-one and one-to-various relations is fully supported, it exploits the relative simplicity of these problems to use a more efficient algorithm than either "fast" or "slow".)

See Also

Adjacent rooms and routes through the map for route-finding through the map rather than a relation.

Example

235. **The Problem of Edith**
A conversation in which the main character tries to build logical connections between what the player is saying now and what went immediately before.
RB 7.11 Character Knowledge and Reasoning

§13.12 Relations which express conditions

One last way to create a new relation and, in many ways, the easiest of all. If we write:

```
Contact relates a thing (called X) to a thing (called Y) when X is part of Y or Y is part of X.
The verb to be joined to means the contact relation.
```

then we would be able to talk about a handle being joined to a door, and a door being joined to a handle, and so on. We are not allowed to declare:

```
The hook is joined to the line.
```

because the question of whether they are joined is not for us to decide: that will be for the condition to determine, whenever we test it. Similarly, we cannot meaningfully write

```
now the hook is joined to the line;
```

(and Inform will not let us) because this relation is not something we can force either way: we can make it come true by other means, maybe, but we cannot simply make it true by saying so. Lastly, this kind of relation is restricted in that we are not allowed to find paths or calculate numbers of steps through it.

So this way to define relations is, on the face of it, just a sort of verbal trick to write conditions in a more attractive way. The more flexible, changeable relations in previous sections have much greater expressive power. All the same, it is nice to be able to write -

```
Nearness relates a room (called A) to a room (called B) when the number of moves from B
to A is less than 3. The verb to be near means the nearness relation.
```

and then to be able to write rules like:

```
Instead of listening when the location is near the Sundial: say "You hear a splashing of
water."
```

As with other relations, there's no reason why we have to use objects. For example:

```
Material is a kind of value. The materials are wood and metal. A thing has a material.

Materiality relates a thing (called X) to a material (called Y) when Y is the material of X.
The verb to be made of means the materiality relation.
```

which enables us to write:

```
if the cube is made of wood, ...
say "The carpenter looks at [the list of things which are made of wood].";
```

And here is a mathematical one:

```
Divisibility relates a number (called N) to a number (called M) when the remainder after dividing M by N is 0. The verb to divide means the divisibility relation. The verb to be a factor of means the divisibility relation.
```

We now find that "2 divides 12", "5 is not a factor of 12" and "12 is divisible by 3" are all true. Again, we are only really gaining a nice form of words, but improving the clarity of the source text is never a bad thing.

Examples

236. **Wainwright Acts**
A technical note about checking the location of door objects when characters other than the player are interacting with them.
RB 3.5 Doors, Staircases, and Bridges

237. **A Humble Wayside Flower**
Relations track the relationships between one character and another. Whenever the player meets a relative of someone he already knows, he receives a brief introduction.
RB 7.1 Getting Acquainted

§13.13 Relations involving values

Although most of the examples in this chapter have involved objects, relations can connect almost any values together. We can create relations in groups, one to various relations, various to one relations, one to one relations, and various to various relations for any combination of kinds. For example:

```
Partnership relates various texts to various texts.

The verb to belong with means the
partnership relation.

"cheese" belongs with "crackers".
"clam" belongs with "chowder".
```

How might we make use of this? Clearly it would be impractical to keep trying:

```
if "caviar" belongs with "aardvarks", ...
if "caviar" belongs with "abacuses", ...
...
```

to find out what "caviar" belongs with. It's still harder to find out if it belongs with anything at all -- in theory we would have to try every possibility, which of course is impossible. Instead we have these phrases:

if (value) **relates to** (name of kind) **by** (relation of values to values):

This condition is true if the value V is such that V relates to something by the given relation. Example: suppose partnership relates various texts to various texts. Then we can test

```
if "chalk" relates to a text by the partnership relation, ...
```

if (name of kind) **relates to** (value) **by** (relation of values to values):

This condition is true if the value V is such that something relates to V by the given relation. Example: suppose partnership relates various texts to various texts. Then we can test

```
if a text relates to "cheese" by the partnership relation, ...
```

If a partner does exist, then we can find it with:

(name of kind) **to which/whom** (value) **relates by** (relation of values to values) ⇒ *value*

or…

(name of kind) **that/which/whom** (value) **relates to by** (relation of values to values) ⇒ *value*

This phrase produces an Y such that the given value V relates to Y by the given relation. Example: suppose partnership relates various texts to various texts. Then we can obtain

```
the text to which "chalk" relates by the partnership relation
```

which might be, say, "cheese". It's a run-time problem to use this if no such Y exists.

(name of kind) **that/which/who relates to** (value) **by** (relation of values to values) ⇒ *value*

This phrase produces an X such that X relates to the given value V by the given relation. Example: suppose partnership relates various texts to various texts. Then we can obtain

```
the text which relates to "cheese" by the partnership relation
```

which might be, say, "chalk". It's a run-time problem to use this if no such X exists.

Of course, there might be many answers to this question, so perhaps these are neater:

list of (name of kind) **that/which/who relate to** (value) **by** (relation of values to values) ⇒ *value*

This phrase produces a list of all the X such that X relates to the given value V by the given relation. Example: suppose partnership relates various texts to various texts. Then we can obtain

```
list of texts which relate to "cheese" by the partnership relation
```

which might be, say, { "chalk", "grapes", "macaroni" }. The answer might be the empty set, but that's not a problem.

list of (name of kind) **to which/whom** (value) **relates by** (relation of values to values) ⇒ *value*

or…

list of (name of kind) **that/which/whom** (value) **relates to by** (relation of values to values) ⇒ *value*

This phrase produces a list of all Y such that the given value V relates to Y by the given relation. Example: suppose partnership relates various texts to various texts. Then we can obtain

```
list of texts to which "chalk" relates by the partnership relation
```

which might be, say, { "cheese", "blackboard", "cliffs" }. The answer might be the empty set, but that's not a problem.

Finally, it's sometimes useful to get at the list of all values which can appear on the left or right hand side of a relation. We need tongue-twister like wording to do it, but:

list of (name of kind) **that/which/whom** (relation of values to values) **relates** ⇒ *value*

This phrase produces a list of all X which relate to anything under the given relation. Example: suppose partnership relates various texts to various texts. Then we can obtain

```
list of texts which the partnership relation relates
```

list of (name of kind) **to which/whom** (relation of values to values) **relates** ⇒ *value*

or…

list of (name of kind) **that/which/whom** (relation of values to values) **relates to** ⇒ *value*

This phrase produces a list of all Y which anything relates to under the given relation. Example: suppose partnership relates various texts to various texts. Then we can obtain

```
list of texts which the partnership relation relates to
```

For efficiency reasons, there are no guarantees about what order these lists have - but

they can of course always be sorted when found.

Examples

238. **Meet Market**
    A case in which relations give characters multiple values of the same kind.
    RB 7.1 Getting Acquainted

239. **For Demonstration Purposes**
    A character who learns new actions by watching the player performing them.
    RB 7.14 Obedient Characters

§13.14 Relations as values in their own right

As we've seen, most relations have names - "containment relation", for instance. These are themselves values in Inform, though there are a few restrictions on how they are used. (Relations can contain a colossal amount of data, so we don't want to have to copy them casually.)

Consider these two examples:

```
Parity relates a number (called N) to a number (called M) when N minus M is even.

Joint magnitude relates a number (called N) to a number (called M) when N plus M is greater than 7.
```

Here "parity relation" and "joint magnitude relation" are both values of the same kind: "relation of numbers to numbers". In general, every relation is a value of kind "relation of K to L", for the appropriate kinds K and L. So the parity relation doesn't have the same kind as the containment relation, for example. Because it often happens that K and L are the same, we can just say "relation of K" in this case, so we could equally say that the kind of the parity relation is "relation of numbers".

This is useful to know when writing phrases like so:

```
To chart (R - a relation of numbers):
    repeat with N running from 1 to 5:
        repeat with M running from 1 to 5:
            if R relates N to M, say "[N] <=> [M] ";
        say "[line break]";
```

and now "chart parity relation" will work nicely, but "chart visibility relation" will be rejected (as it should be, because it relates things, not numbers). In general, if R is any

relation, we can write

```
if R relates X to Y, ...
now R relates X to Y;
now R does not relate X to Y;
```

to test, set and unset a relation R between two values. (Inform checks that the values X and Y have the right kind and produces a problem message if not.)

Several useful adjectives can be applied to relations:

```
"empty" - nothing relates to anything else
"symmetric" - by definition X relates to Y if and only if Y relates to X
"equivalence" - this is a relation "in groups", or an "equivalence relation"
"one-to-one" - it relates one K to one L
"one-to-various" - similarly
"various-to-one" - similarly
"various-to-various" - similarly
```

So for example it's possible to ask

```
if R is a symmetric one-to-one relation of texts, ...
```

With some relations, it's possible to clear them out by writing:

```
now R is empty;
```

and with temporary relations (see the next section), it's even possible to change their valencies (one-to-one vs. one-to-various, etc.) using "now", but only when they are empty. The exceptions where "empty" can't be used are those which can't be changed at all, like the parity relation above, and a few built-in cases such as the support, containment and incorporation relations, where emptying would dissolve the model world in a disastrous way.

Example

240. **Number Study**
     The parity and joint magnitude relations explored.

RB 10.11 Mathematics

§13.15 Temporary relations

So far in this chapter, we've only seen relations which exist permanently during play. The relationships might change - sometimes Red Riding Hood would be in the Woodcutter's Cottage, sometimes not - but the relations themselves were eternal.

In fact, though, we can also create relations to be dynamic data structures, like lists:

let (a name not so far used) **be** (description of relations of values to values)

This phrase creates a new temporary variable, and sets its value to the identity of a newly created and equally temporary relation. These last only for the present block of phrases, which certainly means that they exist only in the current rule. Example:

```
let the password dictionary be a relation of texts;
```

This makes a purely temporary various-to-various relation between texts, which lasts as long as the temporary value "password dictionary" lasts. By default, relations are various-to-various, but we could instead write, say:

```
let the nicknames catalogue be a various-to-one relation of texts;
```

Such a relation exists only in the current phrase, and is destroyed when the phrase finishes, like any other "let". Of course there's no verb whose meaning is this relation, but that's no obstacle, because we can manipulate it using "relates":

```
now the nicknames catalogue relates "Trudy" to "Snake-eyes";
```

(At present such a relation cannot be used outside its own phrase.)

§13.16 What are relations for?

It is easy to say what verbs are for: they are to express relations. But what are relations for?

Inform 7's focus on relations between objects is unusual as an approach to interactive fiction; the concept does not exist in most design systems, or rather, it does but is submerged. Traditional design systems do, after all, have the spatial relations of being inside, on top of, and so on. It could well be said that these are the only relationships that inanimate objects ever have. A stone can be on top of a table, and if so then that expresses their entire association.

This is because the stone, and the table, have no opinions, emotions, knowledge or memory. If the stone is taken away and then put back, nothing has changed. People, on the other hand, tend to remember having met each other before; they like being in some places, but not others; their behaviour depends on who, or what, is nearby. Being conscious, they have internal states, unlike the stone. Relations are a simple but powerful way to express and talk about such connections, and although they have numerous uses in physical contexts too, they are at their most powerful when helping to make the characters of interactive fiction come alive.

Examples

241. Murder on the Orient Express
    A number of sleuths (the player among them) find themselves aboard the Orient Express, where a murder has taken place, and one of them is apparently the culprit. Naturally they do not agree on whom, but there is physical evidence which may change their minds…
    RB 7.11 Character Knowledge and Reasoning

242. What Not To Wear
    A general-purpose clothing system that handles a variety of different clothing items layered in different combinations over different areas of the body.
    RB 9.3 Clothing

243. Mathematical view of relations
    Some notes on relations from a mathematical point of view, provided only to clarify some technicalities for those who are interested.
    RB 1.4 Information Only

244. Graph-theory view of relations
    Some notes on relations from the point of view of graph theory.
    RB 1.4 Information Only

14. Adaptive Text and Responses

§14.1 Tense and narrative viewpoint
§14.2 Adaptive text
§14.3 More on adapting verbs
§14.4 Adapting text about the player
§14.5 Adapting text referring to other things
§14.6 Adapting demonstratives and possessives
§14.7 Can, could, may, might, must, should, would
§14.8 Adapting contractions
§14.9 Verbs as values
§14.10 Responses
§14.11 Changing the text of responses
§14.12 The RESPONSES testing command

§14.1 Tense and narrative viewpoint

A conspicuous difference between interactive fiction and a traditional novel is the point of view from which it's told. Inform usually produces text like:

```
You can see a grey cat in the basket.
```

where a novel would usually write:

```
He saw a grey cat in the basket.
```

Standard interactive fiction (IF) is second person singular, and present tense; most novels are told in the third person singular, and past tense.

But these are just conventions - a few novels, for example, use the so-called present historic ("Napoleon looks up at the sky and sighs. Must Ney always be so doubting?"), and plenty are told in the first person singular ("I always get the shakes before a drop."). Inform allows some of this flexibility, too. The two values:

```
story viewpoint
story tense
```

control the style of the text produced. The story viewpoint has to be one of the values:

```
first person singular
second person singular
third person singular
first person plural
second person plural
third person plural
```

(which are actually the six possible values of a kind called "narrative viewpoint"), while the story tense must be one of:

```
past tense
present tense
future tense
perfect tense
past perfect tense
```

(from a kind called "grammatical tense"). Combining these gives 30 possibilities in all, though only a few are at all commonly used.

It's important to make a very large caveat here: Inform uses these settings in producing the replies ("responses") by the built-in actions, but the only way for all of our own text to have a particular tense or narrative viewpoint is to write it that way. If we write:

```
The Taj Mahal is a room. "You stand and admire the Taj Mahal."

When play begins:
    now the story viewpoint is first person plural;
    now the story tense is past tense.
```

then we're likely to see the following peculiar transcript:

```
Taj Mahal
You stand and admire the Taj Mahal.

>e
We couldn't go that way.
```

That's because the response ("We couldn't go that way") was constructed to follow the settings for viewpoint and tense, but the fixed text of the room description wasn't. In fact there are ways to write the room description so that it would adapt itself automatically, as we'll see, but it takes a fair amount of work. More simply:

```
The Taj Mahal is a room. "I stood and admired the Taj Mahal."

When play begins:
    now the story viewpoint is first person plural;
    now the story tense is past tense.
```

In short, tense and viewpoint switching is neat, but it isn't magic.

If we want to write text which will work in whatever the current tense is, the following turn out to be useful little conveniences:

say "[here]"

Produces "here" if the story tense is the present tense, and "there" otherwise.

say "[now]"

Produces "now" if the story tense is the present tense, and "then" otherwise.

§14.2 Adaptive text

Paying attention to the tense and viewpoint is one reason why text might need to adapt. Another is that it might need to adapt according to whether nouns are singular or plural, or whether it talks about the player or some third party. For example, the following rule isn't ideal:

```
Instead of taking: say "[The noun] is pinned down by Dr Zarkov's force field."
```

Most of the time it's fine ("The V-ray is pinned down by Dr Zarkov's force field"), but then:

```
> GET ME
You is pinned down by Dr Zarkov's force field.
> GET CONDENSERS
The condensers is pinned down by Dr Zarkov's force field.
```

Which is a little unfortunate. But the correction is very easy:

```
Instead of taking: say "[The noun] [are] pinned down by Dr Zarkov's force field."
```

The result is much better: "The V-ray is pinned down…"; "You are…"; "The condensers

are…". In fact, it's also convenient because it adapts to the story viewpoint and story tense: "The condensers will be pinned down…"; "He was pinned down…".

How does Inform do this? The answer is not that "[are]" is a specially-written text substitution. In fact Inform can do this with any verb that it has a definition of. For example,

```
"[The noun] [carry] too much static charge."
```

would also adapt itself - "The V-ray carries too much static charge", and so on. There aren't many verbs built in to Inform, but "[have]" and "[carry]" and "[wear]" and "[can]" may be useful, and "[can see]" and "[can touch]". Negative forms like "[are not]" are also available:

```
"[The noun] [cannot touch] the ionizer terminal."
```

might produce "The V-ray will not be able to touch the ionizer terminal.", for example.

As these examples hint, the verb adapts itself to the most recently printed object name. All of this only works if the previous object's name is printed from a substitution. So:

```
"[The condensers] [are] working."
```

will work -- correctly forming "The condensers are working.", "The condensers will be working." or "The condensers were working.", according to the story tense -- but

```
"The condensers [are] working."
```

probably won't work. Inform doesn't have any way to understand the raw text outside of the text substitution marks "[" and "]", and it doesn't recognise "The condensers" as being something's name.

Something else to be careful with is the use of lists. If we write this:

```
"[The condensers] and [the V-ray] [are] smashed by Voltan's birdmen."
```

then Inform is likely to print:

```
The condensers and the V-ray is smashed by Voltan's birdmen.
```

because it looks at the most recently named object - the V-ray, singular - to decide

whether to use "is" or "are". On the other hand, Inform gets this right:

```
"[The list of things on the bench] [are] smashed by Voltan's birdmen."
```

Because Inform constructs the list itself, it's able to appreciate that the things listed are jointly the subject of the verb, and it uses that information to decide on "is" or "are". So:

```
The condensers and the V-ray are smashed by Voltan's birdmen.
The Atomic Furnace shovel is smashed by Voltan's birdmen.
```

§14.3 More on adapting verbs

If we need an adaptive message with a verb which doesn't belong to Inform's built-in set, all we need do is define it. In the previous chapter we defined verbs by giving them meanings, but in fact that's optional. For example:

```
To retrofit is a verb.
```

defines a verb without telling Inform what it means. Inform will throw a Problem message if we try to write text like:

```
Flash retrofits the meteor beam.
```

because, after all, it doesn't know what "retrofit" means. But it does still know how to print it, so this works:

```
"[The actor] [retrofit] the Mecha-Mole."
```

which might come out as "Dale retrofits the Mecha-Mole", or "Barin's archers retrofitted the Mecha-Mole", and so on.

This is especially neat for writing a single response to an action which works regardless of who the actor was. For example, the Standard Rules include:

```
say "[The actor] [put] [the noun] on [the second noun]."
```

And this can make either:

```
You put the revolver on the table.
General Lee puts the revolver on the table.
```

Examples

245. **Fun with Participles**
    Creating dynamic room descriptions that contain sentences such as "Clark is here, wasting time" or "Clark is here, looking around" depending on Clark's idle activity.
    RB 2.1 Varying What Is Written

246. **Variety**
    Suppose we want all of our action responses to display some randomized variety. We could do this by laboriously rewriting all of the response texts, but this example demonstrates an alternative.
    RB 2.1 Varying What Is Written

247. **Variety 2**
    This builds on the Variety example to add responses such as "You are now carrying the fedora" that describe relations that result from a given verb, as alternate responses.
    RB 2.1 Varying What Is Written

248. **Narrative Register**
    Suppose we want all of our action responses to vary depending on some alterable quality of the narrator, so that sometimes they're slangy, sometimes pompous or archaic.
    RB 2.1 Varying What Is Written

§14.4 Adapting text about the player

In second-person-singular IF, the player is always "you". Many messages look like so:

```
"You have twenty minutes remaining."
```

where the subject, or the object, of the sentence is "you". But what if we want to have this text adapt itself to different narrative viewpoints?

The solution is to use the following:

```
"[We]" or "[we]"
"[Us]" or "[us]"
"[Our]" or "[our]"
"[Ours]" or "[ours]"
"[Ourselves]" or "[ourselves]"
```

The capitalised and uncapitalised versions are identical except, of course, that the initial

letter of the resulting text is upper case in one but not the other. As examples of these:

```
"[We] [carry] the Queen's warrant."
"The birds drop pebbles on [us]. Right on [our] heads!"
"[Ours] are the burdens of office, which [we] take on [ourselves]."
```

Notice that all five of these forms are differently worded, in English. That's the reason why we use the plural to write them - the traditional second person plural forms would be "you", "you", "your", "yours" and "yourself", so we wouldn't know if "[you]" was supposed to be the subject or the object of the verb. So the convention with all of these adaptive forms is that we use "we" and its variations. (That's also why the verbs are written in the plural - "[carry]", not "[carries]".)

§14.5 Adapting text referring to other things

The family in the previous section - "[we]", "[us]", "[our]", "[ours]", "[ourselves]" - always referred to the player. But we also sometimes want to refer to other things without naming them. For example, how should we adapt this?

```
> EXAMINE TREE
It has no clear outline in this misty netherworld.
```

We can easily make the verb adapt - change the "has" to "[have]" - but the trick here is to make the "It" adapt to cases where what's examined is plural, or animate. What we want is:

```
Instead of examining in the Netherworld:
    say "[regarding the noun][They] [have] no clear outline in this misty netherworld."
```

For example, this produces:

```
> EXAMINE ME
You have no clear outline in this misty netherworld.
> EXAMINE MARK
He has no clear outline in this misty netherworld.
> EXAMINE DRUMS
They have no clear outline in this misty netherworld.
```

Note that we have to say "[regarding the noun]", not just start in with "[They]", because nothing has been named so far in the sentence - so Inform doesn't know what object it refers to. "[regarding the noun]" prints nothing, and simply tells the printing part of Inform that the subject has changed.

This isn't always needed:

```
"[We] [have] a look at [the noun], but [they] [are] just too big."
```

works fine, because printing "[the noun]" changes the subject to that, and then "[they]" agrees with it automatically. The text might come out, for example, as:

```
I had a look at Peter Rabbit, but he was just too big.
You have a look at the chessmen, but they are just too big.
We have a look at ourselves, but we are just too big.
```

We have a family of five text substitutions here, matching those in the previous section:

```
"[They]" or "[they]"
"[Them]" or "[them]"
"[Their]" or "[their]"
"[Theirs]" or "[theirs]"
"[Themselves]" or "[themselves]"
```

There's also the peculiar impersonal non-object for English sentences like "It is raining" or "There are books":

```
"[It]" or "[it]"
"[There]" or "[there]"
```

These look pointless - but consider the two texts

```
"[We] [take] [the noun]. It [rain] harder."
"[We] [take] [the noun]. [It] [rain] harder."
```

The first one risks printing "We took the scissors. It rain harder.", because it makes "[rain]" agree with "scissors", which are plural. But the second text makes "[rain]" agree with "[it]". And, as a convenience:

```
"[It's]" or "[it's]"
"[There's]" or "[there's]"
```

do the obvious thing using the current story tense.

Finally, we occasionally want to agree with a number:

```
"Honestly, [dud count][regarding the dud count] of these [are] broken."
```

§14.6 Adapting demonstratives and possessives

Consider the following message: how might we make this adaptive?

```
> MEASURE TOP SHELF
You really are not tall enough to reach that.
```

The verbal part is easy enough, but "that" needs a new feature.

```
"[We] really [are not] tall enough to reach [regarding the noun][those]."
```

This could then adapt to, say,

```
> MEASURE JAM TARTS
He really was not tall enough to reach those.
```

Notice that it's "[regarding the noun][those]", not just "[those]". If we wrote "[those]", Inform would make it agree with the player, who was printed earlier in the sentence by the "[We]".

Lastly, how about:

```
> PUT TEAPOT IN MOUSEHOLE
The teapot's height is just too great.
```

This time we want:

```
"[regarding the noun][Possessive] height [are] just too great."
```

which might adapt to, say,

```
Our height is just too great.
Alice's height will be just too great.
```

Actually, "[regarding …]" can be used for a description of possibly many items, too. For example:

Every turn when the player carries something:

```
say "Every possession is a worry. I wonder if [regarding things carried by the player][they] still [look] okay in your pocket?"
```

So if the player carries just a single coin, say, this automatically becomes:

```
Every possession is a worry. I wonder if it still looks okay in your pocket?
```

but if the player carries a pair of scissors (a single plural-named item) or a coin and an iPhone, it becomes:

```
Every possession is a worry. I wonder if they still look okay in your pocket?
```

Once again these text substitutions are available in capitalised and uncapitalised forms:

```
"[Those]" or "[those]"
"[Possessive]" or "[possessive]"
```

In fact "[Those]" and "[those]" do subtly different things, besides the capital letter, because "[Those]" expects to be the subject of the sentence and "[those]" the object, and this makes a difference if the noun in question is a person. If the noun is an odious person called Tilly then

```
"[regarding the noun][Those] is unacceptable."
"You've never liked [regarding the noun][those]."
```

would come out as "She is unacceptable" - so "[Those]" becomes "She" - but "You've never liked her" - so "[those]" becomes "her". If we need these in different cases, we can explicitly ask for that:

```
"[those in the nominative]"
"[Those in the accusative]"
```

Example

249. **Olfactory Settings**
   Some adaptive text for smelling the flowers, or indeed, anything else.
   RB 2.1 Varying What Is Written

§14.7 Can, could, may, might, must, should, would

English uses so-called "modal verbs" to change a sentence so that it talks about

something only possibly happening. For example, the sentence "Fred goes to school" can be modified to "Fred must go to school", "Fred should go to school" or even "Fred might go to school".

Inform supports the use of modal verbs in text substitutions. For example,

```
"[Fred] [might go] to school."
```

would in the present tense come out as "Fred might go to school.", but could alternatively be "Fred might have gone to school." As this example shows, all that's needed is to take a verb we'll call V - this case, "go" - and we can write any of these:

```
"[can V]" or "[cannot V]" or "[can't V]"
"[could V]" or "[could not V]" or "[couldn't V]"
"[may V]" or "[may not V]" or "[mayn't V]"
"[might V]" or "[might not V]" or "[mightn't V]"
"[must V]" or "[must not V]" or "[mustn't V]"
"[should V]" or "[should not V]" or "[shouldn't V]"
"[would V]" or "[would not V]" or "[wouldn't V]"
```

That helps us to handle informal usages like this one:

```
"You can't go that way."
```

To make this message adaptive, we write:

```
"[We] [can't go] that way."
```

which can adapt in surprising ways -- "They won't be able to go that way.", for example.

Note that the verb V has to be one that Inform knows. But that's easy:

```
To discombobulate is a verb.
```

and then

```
"[Fred] [might not discombobulate] so easily."
```

could produce "Fred might not have discombobulated so easily", for example.

§14.8 Adapting contractions

Contractions usually take the form of part of a word being missed out and replaced by an apostrophe. We've already seen "[can't]", "[couldn't]", "[mayn't]", "[mightn't]", "[mustn't]", "[shouldn't]" and "[wouldn't]", for example. But Inform supports other contractions, too, as follows.

The English verbs "to be" and "to have" are unique in having contracted forms, which we can write "['re]" and "['ve]", like this:

```
"[We]['ve] got rhythm. [We]['re] cool."
```

which might produce, say, "I've got rhythm. I'm cool.", or "He'll have rhythm. He'll be cool.", or "You had got rhythm. You were cool." (The contractions don't appear in the past tense; but the spacing fixes itself automatically.)

The Standard Rules often use a special text substitution for responses like this one:

```
"[They're] hardly portable."
```

This is exactly like "[Those]['re] hardly portable" except that if the plural is needed, Inform prints "They're hardly portable" rather than the correct, but not quite idiomatic, "Those're hardly portable". (If we wrote "[They]['re] ...", that would get the plural form right, but then the singular would be "It's hardly portable" not "That's hardly portable".)

Only a few English verbs have contracted negative forms, beyond those already mentioned. Inform knows these informal forms:

```
"[aren't]"
"[don't]"
"[haven't]"
"[won't]"
```

For example,

```
Instead of taking something:
    say "[The noun] [are] pinned down by Dr Zarkov's force field. [They] [aren't] free to move. [They] [can't] move. [They] [won't] move. [They] [haven't] a chance to move. Anyhow, [they] [don't] move."
```

can produce variations like these:

```
The condensers are pinned down by Dr Zarkov's force field. They aren't free to move. They can't move. They won't move. They haven't a chance to move. Anyhow, they don't move.

You were pinned down by Dr Zarkov's force field. You weren't free to move. You couldn't move. You wouldn't move. You hadn't a chance to move. Anyhow, you didn't move.
```

§14.9 Verbs as values

Each verb known to Inform is actually a value of the kind "verb". To refer to a verb as a value, we have to put the word "verb" in front, as in these examples:

```
the verb contain, the verb might, the verb provoke
```

all of which appear in the Standard Rules.

Two adjectives are provided for use with verbs: "modal" (or "non-modal") to pick out verbs like might, could, should, and so on; and "meaningful" (or "meaningless") to pick out verbs which have a defined meaning as an Inform relation. For example, in the Standard Rules, the verb contain is meaningful, the verb might is modal, and the verb provoke is meaningless.

If V has a meaning as a relation of objects, then "meaning of V" produces that relation. For example,

```
showme the meaning of the verb contain;
showme the meaning of the verb provoke;
```

produces:

```
"meaning of the verb contain" = relation of objects: containment relation
"meaning of the verb provoke" = relation of objects: equality relation
```

As this demonstrates, if a verb has no meaning, or its meaning doesn't relate to objects, we get just the equality relation.

In fact, Inform even defines a verb "to mean": it's meaningful, and its meaning is the meaning relation. Thus:

```
if the verb mean means the meaning relation...
```

is true. More usefully, we can search our vocabulary like this:

```
the list of verbs meaning the containment relation
```

which, unless any non-Standard Rules definitions have been added, produces:

```
list of verbs: {verb contain}
```

Note that the meaning relation can't be changed at run-time: it is not clear what it would even mean to do something like -

```
now the verb contain means the wearing relation;
```

with the story already started, so this will produce a problem message.

say "[adapt (verb)**]"**

Adapts the given verb to the current story tense and story viewpoint. For example, "you [adapt the verb provoke]" might produce "you provoke".

say "[adapt (verb) **from** (narrative viewpoint)**]"**

Adapts the given verb to the current story tense but the given viewpoint. For example, "he [adapt the verb provoke from the third person singular]" might produce "he provokes".

say "[adapt (verb) **in** (grammatical tense)**]"**

Adapts the given verb to the given tense but the current story viewpoint. For example, "you [adapt the verb provoke in the past tense]" might produce "you provoked".

say "[adapt (verb) **in** (grammatical tense) **from** (narrative viewpoint)**]"**

Adapts the given verb to the given tense and viewpoint. For example, "we [adapt the verb provoke in the future tense from the first person plural]" might produce "we will provoke".

say "[negate (verb)**]"**

Adapts the given verb to the current story tense and story viewpoint, giving it a negative sense. For example, "you [negate the verb provoke]" might produce "you do not provoke".

say "[negate (verb) **from** (narrative viewpoint)**]"**

Adapts the given verb to the current story tense but the given viewpoint, giving it a negative sense. For example, "he [negate the verb provoke from the third person singular]" might produce "he does not provoke".

say "[negate (verb) **in** (grammatical tense)**]"**

Adapts the given verb to the given tense but the current story viewpoint, giving it a negative sense. For example, "you [negate the verb provoke in the past tense]" might produce "you did not provoke".

say "[negate (verb) **in** (grammatical tense) **from** (narrative viewpoint)**]"**

Adapts the given verb to the given tense and viewpoint, giving it a negative sense. For example, "we [negate the verb provoke in the future tense from the first person plural]" might produce "we will not provoke".

Note that the verb doesn't have to be named explicitly for use by the adapt or negate phrases, so for example:

```
To decide which text is the rendering of (V - verb) (this is my rendering):
    decide on "[negate V in the past perfect tense]".
When play begins:
    showme my rendering applied to the list of meaningful verbs.
```

produces:

```
"my rendering applied to the list of meaningful verbs" = list of texts: {"had not had", "had not related", "had not meant", "had not provided", "had not contained", "had not supported", "had not incorporated", "had not enclosed", "had not carried", "had not held", "had not worn", "had not been able to see", "had not been able to touch", "had not concealed", "had not unlocked"}
```

Lastly, we can get at three other useful parts of a verb, too. These aren't adaptive, of course: a verb only has one infinitive form.

say "[infinitive of (verb)**]"**

Produces the infinitive of the given verb. Note that this is without a "to": for example, "[infinitive of the verb carry]" is "carry", not "to carry".

say "[past participle of (verb)**]"**

Produces the past participle of the given verb. For example, "[past participle of the verb carry]" is "carried". Warning: because modal verbs like "should" or "might" are defective in English, this will produce odd results on them - "shoulded" and "mighted", for example.

say "[present participle of (verb)**]"**

Produces the present participle of the given verb. For example, "[present participle of the verb carry]" is "carrying". Warning: because modal verbs like "should" or "might" are defective in English, this will produce odd results on them - "shoulding" and "mighting", for example.

Examples

250. **History Lab**
We create phrases such as "the box we took" and "the newspaper Clark looked at" based on what has already happened in the story.
RB 2.1 Varying What Is Written

251. **Relevant Relations**
An example of how to create room descriptions that acknowledge particular relations using their assigned verbs, rather than by the heavily special-cased code used by the standard library.
RB 2.1 Varying What Is Written

§14.10 Responses

Most of the text which the player sees is drawn from the source, but mixed in with this are messages apparently added by Inform itself - usually in the form of short sentences saying that something has been done, or that something can't be done. Such pieces of text are called "responses", because they are almost always replies to commands. For example:

```
> EAST
You can't go that way.

> JUMP
You jump on the spot.
```

Responses like this, which don't appear anywhere in the source text, come from one of the extensions being used; most often from the Standard Rules, the "extension" which is automatically included in every project. The SR contain many small rules, and almost all of these are capable of producing one or two standard responses. These are labelled with the rule's name and then a bracketed letter - (A), (B), (C), … as needed so that every response has its own unique name. There's nothing very mysterious about how this is done. For example, here is a rule with one response:

```
Carry out taking inventory (this is the print empty inventory rule):
    if the first thing held by the player is nothing,
        say "[We] [are] carrying nothing." (A) instead.
```

which makes the familiar text "You are carrying nothing." a response named:

```
print empty inventory rule response (A)
```

These names are actually values, belonging to the kind "response". Because of that, if we try this:

```
say "Hmm: [print empty inventory rule response (A)]"
```

Inform will produce

```
Hmm: print empty inventory rule response (A)
```

since we gave Inform a value to print, and that's just what it then did. As an alternative:

say "[text of (response)**]"**

This text substitution writes out the current text of the given response.

Thus,

```
say "Hmm: [text of print empty inventory rule response (A)]"
```

produces

```
Hmm: You are carrying nothing.
```

§14.11 Changing the text of responses

These responses are named so that they can be changed. Most IF authors dislike one or two of the existing responses, and some would like to change almost all of them to give the text a different style; and extensions for IF in languages other than English change literally every response, of course.

It's very easy to change responses:

```
The print empty inventory rule response (A) is "Your hands are, like, totally empty. Lame."
```

and we can even do this dynamically during play:

```
now the print empty inventory rule response (A) is "Your hands ...";
```

just as if we were setting a variable.

Example

252. **Responsive**
    Altering the standard inventory text for when the player is carrying nothing.
    RB 2.1 Varying What Is Written

§14.12 The RESPONSES testing command

In practice we can't change these responses unless we know what they're called. One way to find out is just to read through the extensions we're using, but that's a laborious process. A more practical answer is to type:

```
> RESPONSES
```

which replies by listing the sets of responses currently available; for example, it says that RESPONSES 1 is the set of responses for the Standard Rules. We can then type exactly that:

```
> RESPONSES 1
Standard Rules:
    block vaguely going rule response (A): "You'll have to say which compass direction to go in."
    print the final prompt rule response (A): "> [run paragraph on]"
    ...
```

and so on. This lists all of the responses, rule by rule, along with their current texts.

15. Numbers and Equations

§15.1 How do we measure things?
§15.2 Numbers and real numbers
§15.3 Real number conversions
§15.4 Printing real numbers
§15.5 Arithmetic
§15.6 Powers and logarithms
§15.7 Trigonometry
§15.8 Units
§15.9 Multiple notations
§15.10 Scaling and equivalents
§15.11 Named notations
§15.12 Making the verb "to weigh"
§15.13 The Metric Units extension
§15.14 Notations including more than one number
§15.15 The parts of a number specification
§15.16 Understanding specified numbers
§15.17 Totals
§15.18 Equations
§15.19 Arithmetic with units
§15.20 Multiplication of units

§15.1 How do we measure things?

In a poem, or in a novel, exact scientific measurements are not the point. So a writer who wants to set up ways to describe the sky at different times might go for something like this:

```
The sky can be cadmium, mackerel, overcast or cornflower.
```

And nobody is interested in the sun angle, the percentage of cloud cover, or any of the other numbers behind all of this. Similarly, if we walk into a familiar office which has been disturbed, we might well say "Look! The filing cabinet is in the middle of the floor." We are not likely to exclaim "Look! The filing cabinet is 1.2m from the east wall and 2.1m from the north wall."

But some writers of interactive fiction do like to make use of physical realism. For instance, it's easier to forbid a bulky object being taken through a narrow doorway if there is a way to measure and compare sizes.

Most computer programs write numbers in the same way, whatever they're used for. But human beings don't. If someone says "How far is Duluth?", we're more likely to say "100 miles" than just "100". This is a useful feature of natural language, because it means we always know how to translate that number into reality - it's 100 miles, not 100 km, or 100 inches; and it's definitely a distance, not 100 apples or 100 kilograms.

Inform lets us use plain numbers if we want to, but it also allows us to create numerical kinds of value:

```
A distance is a kind of value. 5 miles specifies a distance.
```

That kind of definition, and the consequences, will be the subject of this chapter. But we will first look a little harder at the two numerical kinds of value we get for free: "number" and "real number".

§15.2 Numbers and real numbers

Inform uses two different kinds of numerical quantity: "number" and "real number". Neither is better than the other: they're different approaches, each good for a different purpose.

What Inform calls a "number" is a whole number, positive, negative or zero. The range of numbers we can hold is not unlimited - if the format Setting for a project is the Z-machine, then we have:

```
-32768, -32767, ..., -3, -2, -1, 0, 1, 2, 3, ..., 32767
```

and if it is set to Glulx, then we have:

```
-2147483648, -2147483647, ..., -3, -2, -1, 0, 1, 2, 3, ..., 2147483647
```

Numbers from zero to twelve may be written out, but larger ones must be written as numerals. So "twelve" or "12", but "13" only.

If we're using Glulx, Inform also has "real numbers" such as

```
2.1718, 4.0, -1633.9
```

which are not restricted to whole numbers, but which are stored only approximately: only about six to nine decimal digits can be relied on. For example,

```
showme 1.2345654321;
showme 1.2345667890;
```

produces

```
real number: 1.23457
real number: 1.23457
```

because these two numbers are so close together that Inform can't tell them apart. But we do also get the ability to represent enormously large or small quantities, and to help with that, Inform can read and write "scientific notation". For example,

```
let Avogadro's number be 6.022141 x 10^23;
```

is equivalent to typing

```
let Avogadro's number be 602214100000000000000000.0;
```

The "x 10^23" part tells Inform that the decimal point belongs 23 places to the left of where it's written. (In scientific papers, the 23 would be printed as a superscript -- it's 10 to the power 23 -- but that's not convenient to type in to the source text, so we use the "^" symbol to indicate superscript.) The range we can hold is roughly:

```
1.18 x 10^−38 to 3.4 x 10^38
```

It's hard to convey just how enormously different these two numbers are: if we used them to measure widths in meters, one would be a hundred trillion trillion times smaller than an atom, the other a billion times larger than the entire visible universe. Scientific notation is the ultimate adjustable spanner.

Inform also allows the two most famous real numbers in mathematics to be given by their names:

```
pi
e
```

which are close to 3.14159265 and 2.7182818 respectively. (Lower case letters must be used: these can't be written "Pi" or "E". Euler's constant gamma, always in the bronze medal position, will have to be written out longhand as 0.5772156649.)

Most computer programming languages traditionally write floating-point numbers using

the E notation, like so:

```
6.022141E+23;
```

Inform will follow suit if the use option "Use engineering notation." is set, but by default it isn't.

Example

253. **Alias**
A telephone with phone numbers of the standard American seven-digit length.
RB 9.10 Telephones

§15.3 Real number conversions

This section notes down some technicalities about real numbers which need to be put down in writing somewhere, but won't affect most people most of the time.

Inform allows us to use numbers whenever real numbers are expected, and converts them automatically. For example,

```
cosine of 2
```

is read as if it were

```
cosine of 2.0
```

and produces -0.41615 either way. This conversion goes from exactness to approximation, so we may lose a little accuracy: real numbers measure to an accuracy of about 1 part in 16000000, so they'll have trouble telling the difference between 16000000 and 16000001. But this is unlikely to matter, since real numbers are used only for approximate calculations anyway.

The ordinary arithmetic operations work on both numbers and real numbers, so the meaning of "N plus M" depends on the kinds of N and M. In general the rule is that if either is a real number then the other one is automatically converted, and real arithmetic is used. So:

```
3 divided by 2 = 1
3 divided by 2.0 = 1.5
3.0 divided by 2 = 1.5
3.0 divided by 2.0 = 1.5
```

In general we can't do the reverse, that is, we can't silently use a real number where a number is expected. For example,

```
word number 1.6 in "The Great Wall of China"
```

makes no sense. But we can explicitly convert them:

(real number) **to the nearest whole number** ⇒ *number*

This phrase performs signed addition on the given values, whose kinds must agree, and produces the result. Examples:

```
1.4 to the nearest whole number = 1
1.6 to the nearest whole number = 2
-1.6 to the nearest whole number = -2
```

We probably ought to bear in mind that the limited range of "number" means that the nearest whole number might not be all that near. For example:

```
6 x 10^23 to the nearest whole number = 2147483647
```

because 2147483647 is the highest value a "number" can have.

Finally, real number can also store two interesting not-really-number sorts of value. First, we have

```
plus infinity, minus infinity
```

which are used to keep track of what happens when we divide by really small quantities. It's mathematically impossible to divide by 0, but this can be hard to avoid when we're using real numbers, because they're only approximately stored - so it's not always possible to say whether they're exactly 0 or not. So in real number arithmetic,

```
showme 1.0 divided by 0.0;
```

doesn't throw a run-time problem the way that

```
showme 1 divided by 0;
```

does. Instead, it produces "plus infinity". Infinity behaves roughly the way we might expect - for example, "2 divided by plus infinity" produces 0.0 - but once it comes into a calculation the result probably lies on some extreme and won't be very useful. Amusingly, the following is correct Inform syntax:

```
plus infinity to the nearest whole number
```

and evaluates of course to 2147483647. We can use the adjectives "infinite" and "finite" to talk about these numbers: plus infinity and minus infinity are infinite, everything else is finite.

The same problem occurs for calculations like square roots. It's impossible to take the square root of a negative number, but we don't want to throw a run-time problem, because approximation means we can't always guarantee to stay the right side of 0. So for a few calculations like this, Inform generates what's called a "nonexistent" real number. We can use the adjectives nonexistent or existent to talk about this. Every number mentioned on this page so far is "existent", including the infinities. The only way to get a nonexistent number is to carry out an impossible mathematical operation such as

```
logarithm of -10
```

(The design of "real number" here follows well established trade-offs for scientific computing. Inform follows the IEEE-754 binary32 standard for floating-point arithmetic, so Inform's "real number" behaves very like the "float" type in C, C++, Java and similar programming languages. A "nonexistent" number is what's often called a NaN - a Not-a-Number.)

§15.4 Printing real numbers

say "[(real number) to (number) decimal places]"

This text substitution writes out the number to the given number of decimal places. Examples:

```
"The semicircle is roughly [pi to 3 decimal places] paces around."
```

produces "The semicircle is roughly 3.142 paces around." The number of places can only usefully be from 1 to 8. Note that, for example, "[1.235 x 10^-7 to 3 decimal places]" produces 0.0; "[1.235678 x 10^8 to 3 decimal places]" produces "1.236 x 10^8".

say "[(real number) in decimal notation]"

This text substitution writes out the number in decimal form, that is, avoiding "x 10^n" even for very large or very small quantities. For example,

```
"[1.23457 x 10^8 in decimal notation]"
```

produces 123457000.0 rather than 1.23457 x 10^8. This can look pretty extreme: for example, "[1.8983 x 10^27 in decimal notation]", the mass of the planet Jupiter in kilograms, produces 1898296960000000000000000000.0.

say "[(real number) to (number) decimal places in decimal notation]"

This text substitution writes out the number in decimal form, but rounding to the accuracy given.

say "[(real number) **in scientific notation**]**"**

This text substitution writes out the number in scientific form, that is, using "x 10^n" even for easy-to-judge quantities. For example,

```
"[the reciprocal of 137 in scientific notation]"
```

produces 7.29927 x 10^-3 rather than 0.0073. This can look odd: for example, "[pi in scientific notation]" comes out as 3.14159 x 10^0 rather than 3.14159.

say "[(real number) **to** (number) **decimal places in scientific notation**]**"**

This text substitution writes out the number in scientific form, but rounding to the accuracy given.

§15.5 Arithmetic

We are allowed to perform about the same operations on numbers as are provided by a simple office calculator, starting with addition, subtraction, multiplication and division. We can use the traditional typewriter symbols for these, +, -, * and /, or can spell them out in words as "plus", "minus", "times" (or "multiplied by"), and "divided by". Definitively:

(arithmetic value) + (arithmetic value) ⇒ *value*

or...

(arithmetic value) **plus** (arithmetic value) ⇒ *value*

This phrase performs signed addition on the given values, whose kinds must agree, and produces the result. Examples:

```
200 + 1 = 201
10:04 AM + two minutes = 10:06 AM
```

(arithmetic value) - (arithmetic value) ⇒ *value*

or…

(arithmetic value) **minus** (arithmetic value) ⇒ *value*

This phrase performs signed subtraction on the given values, whose kinds must agree, and produces the result. Examples:

```
200 - 1 = 199
10:04 AM - two minutes = 10:02 AM
```

(arithmetic value) * (arithmetic value) ⇒ *value*

or…

(arithmetic value) **times** (arithmetic value) ⇒ *value*

or…

(arithmetic value) **multiplied by** (arithmetic value) ⇒ *value*

This phrase performs signed multiplication on the given values, whose kinds must be dimensionally compatible, and produces the result. Examples:

```
201 times 3 = 603
two minutes times 4 = eight minutes
```

(arithmetic value) / (arithmetic value) ⇒ *value*

or…

(arithmetic value) **divided by** (arithmetic value) ⇒ *value*

This phrase performs signed division on the given values, whose kinds must be dimensionally compatible, and produces the result. Examples:

```
201 divided by 3 = 67
202 divided by 3 = 67
202.0 divided by 3 = 67.33334
twenty minutes divided by 4 = five minutes
twenty minutes divided by five minutes = 4
```

Division rounds whole-number values down to the nearest whole number. An attempt to divide a number by 0 will cause a run-time problem message; but an attempt to divide a real number by 0 will instead produce plus infinity or minus infinity.

remainder after dividing (arithmetic value) **by** (arithmetic value) ⇒ *value*

This phrase performs signed division on the given values, whose kinds must be dimensionally compatible, and then produces the remainder. Examples:

```
remainder after dividing 201 by 5 = 1
remainder after dividing twenty minutes by 7 = six minutes
```

It is mathematically impossible to divide by 0, so any attempt to find the remainder after dividing a number by 0 will cause a run-time problem message. For a real number this won't arise and the remainder will usually be 0.0.

The verbal and symbolic forms of these phrases are equivalent:

```
the score + 10
the score plus 10
```

It's probably better style to spell them out in full when writing text, and keep the symbols for writing equations, as we'll see later on in the chapter. (If we do use the symbols, then spaces around them are obligatory: to Inform, they are words which just happen to be spelt with symbols instead of letters.)

Arithmetic often produces fussily exact answers which seem inappropriate in a conversation. Nobody says "Steeple Barton is 7.655 miles down the road", but "Steeple Barton is eight miles down the road" sounds perfectly normal. In order to make that sort of report easier to make, Inform provides another arithmetic operation, one that's not found in most computer programming languages:

(arithmetic value) **to the nearest** (arithmetic value) ⇒ *value*

This phrase rounds the given value off, rounding upward in boundary cases. Examples:

```
201 to the nearest 5 = 200
205 to the nearest 10 = 210
10:27 AM to the nearest five minutes = 10:25 AM
```

Inform has very few mathematical functions built in as phrases, because these aren't very often needed in story-telling. But it does provide these:

square root of (arithmetic value) ⇒ *value*

This phrase produces an approximate square root, to the nearest integer, of the given value, which must be of a kind which has square roots. Example:

```
square root of 16 = 4
```

Trying to take the square root of a negative number will cause a run-time problem, because then we can't even nearly solve it.

(Warning: this is slow to compute if the Z-machine setting is used. For best performance, use Glulx.)

real square root of (arithmetic value) ⇒ *value*

This phrase produces a square root, as accurately as a real number can hold it, of the given value, which must be of a kind which has square roots. Example:

```
real square root of 2 = 1.41421
```

The real square root of a negative number is nonexistent.

cube root of (arithmetic value) ⇒ *value*

This phrase produces an approximate cube root, to the nearest integer, of the given value, which must be of a kind which has cube roots. Example:

```
cube root of 27 = 3
cube root of -27 = -3
```

(Warning: this is not very accurate if the Z-machine setting is used. For best performance, use Glulx.)

We can compare numbers using either the traditional computer-programming symbols, or using words:

```
if the score is less than 10
if the score < 10
```

and similarly for "greater than", "at least" and "at most", with the symbols ">", ">=" and "<=". But we are not allowed the equals sign: for that we need only use "is" -

```
if the score is 10
```

§15.6 Powers and logarithms

If the last section provided a basic office calculator, this section and the next provide the more exotic rows of buttons found on a scientific calculator. All of these are done using real number arithmetic. To start with some dull ones, here are two ways to round off numbers:

ceiling of (real number) ⇒ *real number*

Produces the smallest integer value greater than or equal to the one given. Examples:

```
ceiling of pi = 4.0
ceiling of -16.315 = -16.0
```

(Note that the result is still a real number; it simply has no fractional part any more.)

floor of (real number) ⇒ *real number*

Produces the largest integer value less than or equal to the one given. Examples:

```
floor of pi = 3.0
floor of -16.315 = -17.0
```

(Note that the result is still a real number; it simply has no fractional part any more.)

Two more easy functions:

absolute value of (real number) ⇒ *real number*

Removes the sign from a value, leaving positive numbers alone but making negative ones positive. Examples:

```
absolute value of 62.1 = 62.1
absolute value of 0 = 0.0
absolute value of -62.1 = 62.1
absolute value of minus infinity = plus infinity
```

reciprocal of (real number) ⇒ *real number*

Calculates 1/x, that is, divides up 1 into this many pieces. Examples:

```
reciprocal of -2 = -0.5
reciprocal of 0.1 = 10.0
reciprocal of 7 = 0.14286
reciprocal of plus infinity = 0.0
```

Now for taking powers. In general we have:

(real number) **to the power** (real number) ⇒ *real number*

Computes x to the power y. Examples:

```
2 to the power 4 = 16.0
100 to the power 0.5 = 10.0
7 to the power -1 = 0.14286
pi to the power 0 = 1.0
```

In the words of the Glulx specification document (section 2.12), "the special cases are breathtaking": if you need to know exactly what, say, "minus infinity to the power Y" will do for different cases of Y, refer to the details of the "pow" opcode.

To compute square roots, it's more efficient to use "real square root of X" function than "X to the power 0.5", though both work. To obtain the Nth root of X, we might use:

```
X to the power (reciprocal of N)
```

being careful to use "reciprocal of N" rather than "1 divided by N" to make sure we're using real and not integer arithmetic.

Similarly, the following is more efficient than "e to the power ...", but equivalent to it:

exponential of (real number) ⇒ *real number*

Computes e to the given power, where e is the base of natural logarithms. Examples:

```
exponential of 0 = 1.0
exponential of 1 = e = 2.7182818
exponential of -10 = 4.53999 x 10^-5
exponential of 10 = 22026.46484
exponential of logarithm of 7.12 = 7.12
```

The reverse of taking powers is taking logarithms.

logarithm to base (number) **of** (real number) ⇒ *real number*

Finds what power the base would have to be raised to in order to get this value. Examples:

```
logarithm to base 10 of 1000000 = 6.0
logarithm to base 10 of 350 = 2.54407
logarithm to base 2 of 256 = 8.0
```

Logarithms of zero or negative numbers are nonexistent. Note that "logarithm to base 10 of ..." is what most calculators call simply "log", but Inform doesn't: it uses "log" for natural logarithms.

natural/-- logarithm of (real number) ⇒ *real number*

Finds what power e would have to be raised to in order to get this value. Examples:

```
logarithm of e = 1.0
logarithm of 1 = 0.0
logarithm of 1000 = 6.90776
logarithm of exponential of 7.12 = 7.12
```

Logarithms of zero or negative numbers are nonexistent. This is the function which most calculators label as "ln", for "log natural", but in mathematical and scientific papers it's more often written "log", and Inform follows that convention.

§15.7 Trigonometry

We have twelve functions left to cover, though they are all closely related.

(real number) **degrees** ⇒ *real number*

Inform measures angles in radians, a convention in which the angle for a half circle is pi, and a right angle is pi divided by 2. This is better from a mathematical point of view, but in practice most people think about angles using degrees, where 180 degrees is a half-circle and a right angle is 90 degrees. This phrase helps with that by converting from degrees to radians: in other words, it multiplies by 0.0174532925, since that's roughly 1/180th of pi. Examples:

```
sine of 90 degrees = 0.0
cosine of 60 degrees = 0.5
```

sine of (real number) ⇒ *real number*

The length of the upright of a right-angled triangle with this angle and a hypotenuse of length 1, where angle is measured in radians. Examples:

```
sine of 0 = 0
sine of 45 degrees = 0.70711
sine of (pi divided by 4) = 0.70711
sine of (pi divided by 2) = 1.0
sine of pi = 0
```

cosine of (real number) ⇒ *real number*

The length of the base of a right-angled triangle with this angle and a hypotenuse of length 1, where angle is measured in radians. Examples:

```
cosine of 0 = 1.0
cosine of 45 degrees = 0.70711
cosine of (pi divided by 4) = 0.70711
cosine of (pi divided by 2) = 0.0
cosine of pi = -1.0
```

tangent of (real number) ⇒ *real number*

The ratio of the upright length to the base length in a right-angled triangle with this angle and a hypotenuse of length 1, where angle is measured in radians. Examples:

```
tangent of 0 = 0.0
tangent of 45 degrees = 1.0
tangent of (pi divided by 4) = 1.0
tangent of (pi divided by 2) = plus infinity
```

arcsine of (real number) ⇒ *real number*

The inverse of the sine function.

arccosine of (real number) ⇒ *real number*

The inverse of the cosine function.

arctangent of (real number) ⇒ *real number*

The inverse of the tangent function.

hyperbolic sine of (real number) ⇒ *real number*

The hyperbolic sine function, often written "sinh" but pronounced "shine".

hyperbolic cosine of (real number) ⇒ *real number*

The hyperbolic cosine function, often written "cosh".

hyperbolic tangent of (real number) ⇒ *real number*

The hyperbolic tangent function, often written "tanh".

hyperbolic arcsine of (real number) ⇒ *real number*

The inverse of the hyperbolic sine function.

hyperbolic arccosine of (real number) ⇒ *real number*

The inverse of the hyperbolic cosine function.

hyperbolic arctangent of (real number) ⇒ *real number*

The inverse of the hyperbolic tangent function.

§15.8 Units

Suppose we want to talk about how tall people are. We could just create a "number" property, like this:

```
A person has a number called height.
```

But then we would have to write lines like "Isabella has height 68", which nobody would naturally say. What we want is to be able to write "Isabella is 5 foot 8." Perhaps the computer will need to store that measurement as the number 68 in some register or other, but we don't want to know about that.

"5 foot 8" is a complicated notation in a way - it involves both feet and inches - so let's start with a simpler example:

```
A weight is a kind of value. 10kg specifies a weight.
```

This is a little different to the kinds of value seen so far, which were all created like so:

```
A colour is a kind of value. The colours are red, green and blue.
```

We can't mix the two styles: a new kind of value will either be numerical at heart ("10kg") or verbal at heart ("blue").

The effect of "10kg specifies a weight" is to tell Inform that this is the notation for writing a constant "weight". So, for instance,

```
The maximum load is a weight that varies. The maximum load is 8000kg.
if the maximum load is greater than 8000kg, ...
```

Inform is then careful not to allow weights to be mixed up with other numerical values. For instance, it won't allow "if the maximum load is 400", because 400 is a number, not a weight.

More or less anything we can do with numbers, we can now do with weights. For instance, we can write:

```
The Weighbridge is a room. "A sign declares that the maximum load is [maximum load]."
```

…which will produce the text "A sign declares that the maximum load is 8000kg."

Numerical kinds of value are sometimes called "units", because one of their main uses is to allow us to write quantities using scientific units such as kilograms. But they have other uses too. We have a great deal of freedom in creating notations like "10kg", or "4 foot 10" - the main thing is that new notations must not already mean a value. So "10 specifies a weight" will not be allowed, because 10 specifies a number already.

By default we can only write whole-number values. As we've seen, Inform can handle both integer (whole-number) and real arithmetic, and they each have their advantages. The default here is to use whole numbers, so

```
10 kg specifies a weight.
```

will store only whole numbers of kilograms (unless clever scaling tricks are used: see the next section). That may be fine, but if we need to handle a wider range of weights, or do scientific calculations that need to be more accurate, this is better:

```
1.0 kg specifies a weight.
```

Here Inform can see from the ".0" in the prototype number that real numbers will be involved. (It needs to be ".0" not, say, ".5" because that could be read as a different sort of notation.) We can still write "8000kg", but we can now also write "1.9885 x 10^30 kg" (the

mass of the Sun) or "9.109383 x 10^−31 kg" (the mass of an electron). On the other hand, any calculations we do will be limited in accuracy to about 6 to 9 decimal places, exactly as for real numbers.

By default we can only write positive values when whole numbers are used. Sometimes it is unnatural to write negative values, and so Inform will issue a Problem message if this is tried - for instance, Inform would not allow us to write a weight of -4 kg. (This doesn't mean that arithmetic on units is forbidden to get a negative result: we may want to work out the difference between two weights. Inform's Problem message is simply to try to prevent the accidental writing of incorrect values.) If we do want the ability to write negative values in the source text, we signal that in the notation itself:

```
-10 kg specifies a weight.
```

That alerts Inform that both positive and negative values for this unit make sense.

If we set up a spread of multiple notations (see the next section) then this is automatically enabled, because then we're clearly dealing with proper physics, where negative values are common; and similarly if we use real numbers (as above).

Examples

254. **rBGH**
    The player character's height is selected randomly at the start of play.
    RB 5.1 The Human Body

255. **Lethal Concentration 1**
    A poisonous gas that spreads from room to room, incapacitating or killing the player when it reaches sufficient levels.
    RB 10.1 Gases

256. **Wonderland**
    Hiking Mount Rainier, with attention to which locations are higher and which lower than the present location.
    RB 6.9 Going, Pushing Things in Directions

257. **Lethal Concentration 2**
    Poisonous gas again, only this time it sinks.
    RB 10.1 Gases

§15.9 Multiple notations

Going back to our weight example:

```
A weight is a kind of value. 10kg specifies a weight.
```

The notation here is a single word, even if it contains digits as well as letters - "10kg". But it doesn't have to be one word. These would have worked, too:

```
10kg net specifies a weight.
10 kg specifies a weight.
```

In fact, we are allowed to have all three at once, as alternatives:

```
A weight is a kind of value. 10kg specifies a weight. 10kg net specifies a weight. 10 kg specifies a weight.
```

If we often have to deal with large weights, it becomes a little cumbersome to keep on writing something like "80000kg". An engineer would write "80 tonnes" for this. Similarly, we wouldn't like road maps to use light years, or speed limit signs to use furlongs per fortnight. So it's sometimes useful to provide a spread of different notations, at different scale factors, for the same kind of value. Here's one way of setting up the tonne, that is, the metric ton:

```
1 tonne specifies a weight scaled up by 1000.
```

This really is an alternative way to write the same thing: for instance, Inform will allow "25kg plus 3 tonne", the result being "3.025 tonne".

That's all very well, but a value like "3 tonne" reads a little oddly, even if it's correct in theory. Outside of scientific journals with old-school copy editing, most people would write "3 tonnes", not "3 tonne". Here's a better try:

```
1 tonne (singular) specifies a weight scaled up by 1000.
2 tonnes (plural) specifies a weight scaled up by 1000.
```

Now Inform will not only recognise both forms, but also use the right one when printing back.

§15.10 Scaling and equivalents

As we've seen, there are two ways to store values like lengths or weights: as whole numbers, or as real numbers. If we prefer to use whole numbers, or if real numbers aren't

available (for example if we're using the Z-machine setting), then we might run into an awkward problem: when we write

```
1 kg specifies a weight.
```

we make this correspond to the whole number "1", and that means Inform can never handle weights smaller than 1 kg.

But as we've seen, we can provide differently scaled notations for the same unit:

```
A length is a kind of value. 1m specifies a length.
1km specifies a length scaled up by 1000.
```

And this allows us to write "0.45km" instead of "450m", if we want to, both having the same effect. "0.45km" doesn't make a real number, despite the decimal point - it's simply another way to write "450m", stored internally as the whole number 450.

Just as we can scale up, so we can also scale down:

```
1cm specifies a length scaled down by 100.
```

Now we have a spread of three notations, so "3cm", "0.03m" and "0.00003km" all mean the same thing. But something quite interesting happened at the same time: Inform realised that we want to know lengths to a greater accuracy than just a whole number of meters.

If we're using whole numbers, and we want to resolve down to very small values, that reduces the size of the largest value we can have. For instance, with the Glulx format setting, writing just

```
A length is a kind of value. 1m specifies a length.
```

gives us a range of 1m up to 2147483647m, which is plenty - it's about six times the distance from the Earth to the Moon. Going down to centimeters:

```
A length is a kind of value. 1m specifies a length. 1cm specifies a length scaled down by
100.
```

gives us instead 1cm up to 21474836.47m, which is still enough to represent any possible distance on the Earth's surface. For instance, London to Sydney is about 17000000m.

Left to itself, Inform chooses the scaling for a unit so that it can represent exactly 1 of the

smallest notation - so in our example Inform resolves down to 0.01m, not 1m, in order that it can represent 1cm accurately. But we can also fix the scaling ourselves:

```
A length is a kind of value. 1m specifies a length scaled at 10000.
```

Notice "scaled at", not "scaled down" or "scaled up" - this is now the first notation for length, so there's no existing notation which it could scale up or down. Anyway, now the range is 0.0001m, the width of a human hair, up to 214748.3647m, which is about 130 miles. (The Kinds index automatically keeps track of the range of values represented exactly.) The "scaled at" feature is meaningless if we're using real numbers, so it throws a Problem message.

Finally, for a really deluxe kind of value, we can also provide "equivalent" notations. The idea here is that we might want both miles and kilometers to work, even though they aren't direct scalings of each other. We can only do this approximately, but:

```
1 mile specifies a length equivalent to 1609m.
```

Equivalent notations are never normally used in printing values back (but see the next section) - we wouldn't want Inform to print a sequence of values such as "1.6km", "1.65km", "1.056 miles", … in an effort to be helpful.

§15.11 Named notations

When it has a variety of notations to choose from, Inform will normally use the neatest one given the size of the value it is printing. Suppose we've set up "weight", with three notations:

```
A weight is a kind of value. 10kg specifies a weight.
1 tonne (singular) specifies a weight scaled up by 1000.
2 tonnes (plural) specifies a weight scaled up by 1000.
```

Inform will then print back values like so:

```
45kg -> "45kg"
1000kg -> "1 tonne"
2500kg -> "2.5 tonnes"
80000kg -> "80 tonnes"
```

Note the way Inform goes into decimal places in order to talk about 2500kg in terms of tonnes rather than kilograms - it is minimising the integer part of the unit, but trying to keep it non-zero. So Inform prefers "45kg" to "0.045 tonnes".

Although Inform's habit of choosing the best notation available is usually just what we want, we sometimes want to make the choice ourselves. For instance, if we were printing out a table of different weights, we might want to give all of them in kilograms, whatever their size. In that case we can, if we want, give names to our different notations:

```
1 tonne (singular, in tonnes) specifies a weight scaled up by 1000.
2 tonnes (plural, in tonnes) specifies a weight scaled up by 1000.
```

Now we could write, for instance:

```
"The weighbridge warns you not to exceed [the maximum load in tonnes]."
```

And the figure will always use tonnes now, even if Inform would normally think it odd: "The weighbridge warns you not to exceed 0.001 tonnes." But it will still correctly use "tonne" or "tonnes" as appropriate - what has changed is that instead of choosing from all of the weight notations, Inform now chooses from the notations labelled as "in tonnes".

§15.12 Making the verb "to weigh"

So now we can invent notations for weight. We could, for instance, write:

```
Weight is a kind of value. 1kg specifies a weight. Every thing has a weight.
```

And that allows us to write:

```
The lead pig is in the Salt Mine. The weight of the lead pig is 45kg.
```

But nobody would say it that way: they'd say "The lead pig weighs 45kg." So what we really need to complete our setup is a verb "to weigh".

We have already created new verbs, but none of those methods are quite convenient for this. We want to relate something tangible (the lead pig) to something intangible (45kg), and there's no convenient relation to express this; if we set it up as a condition, we'd get something we couldn't assert, only test. Instead, we'll do something different this time:

```
The verb to weigh means the weight property.
```

Previous definitions like this ended "means the … relation", rather than "means the … property", but the idea is the same. The meaning of "X weighs Y" is that the weight property of X is equal to Y. So we can now write:

```
A thing usually weighs 1kg. The lead pig weighs 45kg.
something weighing 20kg
if three things weigh 5kg, ...
```

And as we saw in the chapter on Descriptions, we can also set up adjectives, comparatives and superlatives:

```
Definition: A thing is heavy if its weight is 20kg or more.
```

which creates "heavy", "heavier" and "heaviest".

Examples

258. **Dimensions**
This example draws together the previous snippets into a working implementation of the weighbridge.
RB 10.5 Volume, Height, Weight

259. **Lead Cuts Paper**
To give every container a breaking strain, that is, a maximum weight of contents which it can bear - so that to put the lead pig into a paper bag invites disaster.
RB 10.5 Volume, Height, Weight

§15.13 The Metric Units extension

To sum all of this up, what started out as a simple business of setting a notation for lengths becomes something quite elaborate when we try to match the actual notations used by scientists and engineers. It's all optional, of course, but as we want more and more of this, we might find ourselves with a spread of notations like this:

```
1mm ... 1cm ... 1m ... 1km
```

In addition we might want equivalents for the inch, the yard and the mile; and verbal forms like the meter and the millimeter, and then alternate spellings like the kilometre; and then both singular and plural forms. And that's just length - what about density, area, pressure, velocity and a dozen other physical quantities? After a while these declarations start to look as vastly fussy as a box of presentation cutlery.

Fortunately the whole set is indeed available in a presentation box, and at no extra charge.

(a) The built-in extension "Metric Units by Graham Nelson" sets up a whole range of

scientific units, with all the notations we are likely to want. Real numbers are used throughout, so large and small-scale calculations can be carried out quite accurately. Like the other built-in extensions, it has its own documentation and examples.

(b) The built-in extension "Approximate Metric Units by Graham Nelson" does the same but using whole numbers, scaled about right for human situations. This won't be much use for extensive calculations, and won't be as accurate, but it will work reasonably well if real arithmetic isn't available.

§15.14 Notations including more than one number

We've seen quite enough scientific notation for the time being. There are plenty of other notations used in natural language, for everyday concepts, where people don't use a tidy spread of powers of 10. Instead they use mixtures, with some sort of punctuation or text to divide them. For instance, the running time of a piece of music is easier to follow in minutes and seconds than in seconds alone: old-fashioned LP sleeves used to quote running times in the form 4'33.

```
A running time is a kind of value. 3'59 specifies a running time.
```

The choice of "3" here makes no difference, much as the choice of "10" in the weight examples was arbitrary. But the "59" is significant. Numbers after the first one are expected to range from 0 up to the value we quote - so in this case, the number of seconds can be anything from 0 to 59. Or, for instance:

```
A height is a kind of value. 5 foot 11 specifies a height.
```

A specification can contain up to eight numbers like this, but once again we might need to worry about the maximum value which can be stored. For instance, using the 3'59 notation, we can only go up to 546'07 (if we're using the Z-machine format setting) - a little over 9 hours, so the new Tori Amos album will not be a problem, but some of the more punishing German operas might break the bank.

In notations like this, only the first-appearing number part is allowed to be negative, and then only when declared with a minus sign:

```
A secret sign is a kind of value. -2x17 specifies a secret sign with parts mystery and
enigma.
```

Here, the mystery can be negative, but not the enigma.

Notations must not contain double-quotation marks because, even though people did

once use these to denote minutes of arc, they would simply confuse programs like Inform's user interface which have to keep track of what is quoted text and what is not. But other punctuation marks are fine *provided they occur between two digits*. For instance, in

```
A monetary value is a kind of value. $1.99 specifies a monetary value.
```

the full stop between the 1 and the 99 is not interpreted as a division of two sentences; and similarly for colons in examples such as

```
An aspect ratio is a kind of value. 16:9 specifies an aspect ratio.
```

§15.15 The parts of a number specification

We often need to break up a number specification into its pieces. For instance, suppose we want to know the dollars part of $1.99? We can do this by naming the parts:

```
A monetary value is a kind of value. $1.99 specifies a monetary value with parts dollars and cents.
```

We can now find the relevant parts like so. Suppose that "sum" is a monetary value. Then:

```
dollars part of sum
cents part of sum
```

are both numbers, so for instance we can

```
say "Looks like around [dollars part of sum in words] dollar[s]."
```

We can also go the other way:

```
monetary value with dollars part 4 cents part 72
```

produces the monetary value $4.72. (Note the lack of commas or "and"s, and that the parts have to be given in the right order.) This is really intended to be useful when we manipulate such values in unusual ways:

```
An aspect ratio is a kind of value. 16:20 specifies an aspect ratio with parts width and height.

To decide which aspect ratio is the wider version of (AR - an aspect ratio):
    let W be the width part of AR multiplied by 2;
    let H be the height part of AR;
    let the wider ratio be the aspect ratio with width part W height part H;
    decide on the wider ratio.
```

Declaring the parts of a number specification individually also enables us to tack one or more options onto any of the parts:

```
A monetary value is a kind of value. $1.99 specifies a monetary value with parts dollars and cents (optional, preamble optional).
```

This declares that the "cents" part is optional - it will be 0 if not specified - and that if omitted, the non-numeric "preamble" before it should also be omitted. Thus "$3" is now valid and equivalent to "$3.00": indeed it will be the preferred form when Inform prints out a monetary value which is an exact number of dollars. If we had said that "cents" was optional, but not said that the preamble was optional, then "$3." would have been the form - which is less satisfactory.

There is only one other option: "without leading zeros", as in the following.

```
An aspect ratio is a kind of value. 16:20 specifies an aspect ratio with parts width and height (without leading zeros).
```

This ensures that when the ratio 4:3 is printed, it will be printed as "4:3" and not "4:03" as would otherwise happen.

Example

260. Zqlran Era 8
    Creating an alternative system of time for our game, using new units.

RB 4.1 The Passage Of Time

§15.16 Understanding specified numbers

It may be worth noting in passing that number specifications, like all other kinds of value, can be understood in typed commands. (See the chapter on Understanding for more on what can go in such square brackets.) For instance:

```
"America Stands Tall"

The Oval Office is a room. Josh and Toby are men in the Oval. A height is a kind of value. 5 foot 11 specifies a height. A person has a height. Josh is 5 foot 8. Toby is 5 foot 10.

Height guessing is an action applying to one thing and one height. Understand "guess [someone] is [height]" as height guessing.

Check height guessing: if the noun is not a person, say "You can only guess the height of people." instead. Carry out height guessing: if the height of the noun is the height understood, say "Spot on!"; if the height of the noun is greater than the height understood, say "No, [the noun] is taller than that."; if the height of the noun is less than the height understood, say "No, [the noun] is shorter than that."

Test me with "guess josh is 6 foot 3 / guess josh is 5 foot 9 / guess josh is 5 foot 3 / guess josh is 5 foot 8".
```

Example

261. **Snip**
   A string which can be cut into arbitrary lengths, and then tied back together.

RB 10.6 Ropes

§15.17 Totals

This chapter began by mentioning arithmetic, and then went on a long diversion to create scientific units, everyday weights and measures, and other notational conveniences. Putting all of that together, it's time now to calculate something with all of these numerical quantities.

Suppose we invent the idea of weight, and give everything a weight of its own. Most items will have a nominal weight of 1kg, but people will be heavier. Going on actuarial tables, we might say:

```
A weight is a kind of value. 10kg specifies a weight. Everything has a weight. A thing usually has weight 1kg. A man usually has weight 80kg. A woman usually has weight 67kg.

Definition: A thing is light if its weight is 3kg or less.
Definition: A thing is heavy if its weight is 10kg or more.
```

and this provides us with "lighter", "lightest", "heavier" and "heaviest" as before. Now we could say "if Peter is heavier than Paul", or even "if Peter is heavier than 75kg", and so forth. We need one more tool:

total (arithmetic values valued property) **of** (description of values) ⇒ *value*

This phrase produces the total of some property held by all of the values matching the description. A problem message is produced if the values in question can't have that property ("the total carrying capacity of scenes"), or if it holds a kind of value which can't meaningfully be added up ("the total description of open doors"). Example:

```
total carrying capacity of people in the Deep Pool
```

That gives us everything we need for a working balance platform:

```
The balance platform is a supporter in the Weighbridge. "The balance platform is currently weighing [the list of things on the platform]. The scale alongside reads: [total weight of things on the platform]."
```

Note that this only works because we said that "everything has a weight": otherwise it would make no sense to add up the weights of things.

This enables us to get the average weight of a group of things, too:

```
the total weight of things on the platform divided by the number of things on the platform
```

But we should be careful that this does not accidentally divide by zero, which it will if the platform has nothing on it! As well as the average, we could find the maximum and minimum weights:

```
the weight of the heaviest thing on the platform
the weight of the lightest thing on the platform
```

We should remember that "the heaviest thing on the platform" may be ambiguous, because there may be several equally heavy things there. That means

```
if the lead pig is the heaviest thing on the platform
```

will only reliably work if there is no possibility of a tie. A safer bet is:

```
if the lead pig is the weight of the heaviest thing on the platform
```

Example

262. **Nickel and Dimed**
    A more intricate system of money, this time keeping track of the individual denominations of coins and bills, specifying what gets spent at each transaction, and calculating appropriate change.

RB 9.4 Money

§15.18 Equations

Forming totals is all very interesting in its way, but it's book-keeping rather than physics. As a glance at any school science textbook shows, the way to apply physics is to work out an unknown quantity - say, the time taken for a dropped ball to hit the ground - by combining known quantities into an equation - the height it is dropped from, and the strength of gravity.

It's a convention centuries old now that textbooks and research papers never describe these equations in running text. Even for simple formulae, we like to write "F=ma", not "let the force be the mass times the acceleration". And the standard way to print this is to break off and display an equation, not to squeeze it into the text as if it were ordinary verbiage. Just as Inform's Tables imitate those in printed books (see the next chapter), so its Equations do.

In this section, we'll use a combination of three equations to work out how soon and how hard an object pushed off a table will hit the floor. First, we'll include Metric Units, to define all of the kinds of value and notations we need.

```
Include Metric Units by Graham Nelson.
```

Now we'll give everything a mass (Metric Units likes to talk about mass instead of weight, but on Earth it's the same thing) and also set up a typical strength for gravity - it's a little less at the poles, a little more at the equator, but this is the conventional approximate value to use.

```
The acceleration due to gravity is an acceleration that varies. The acceleration due to gravity is usually 9.807 m/ss. A thing has a mass. The mass of a thing is usually 10g.
```

To a Renaissance scientist, typically living in a walled European town, a cannon ball was a familiar thing, and it often featured in imaginary experiments:

```
Laboratory is a room. The cannon ball is in the Laboratory. "A cannon ball perches delicately on a lab bench." The mass of the cannon ball is 2kg.
```

And now we're ready for the three equations. These will all have names, but we could just as easily have numbered them, calling them (say) "Equation 1", "Equation 2" and "Equation 3".

```
Equation - Newton's Second Law
    F=ma
where F is a force, m is a mass, a is an acceleration.

Equation - Principle of Conservation of Energy
    mgh = mv^2/2
where m is a mass, h is a length, v is a velocity, and g is the acceleration due to gravity.

Equation - Galilean Equation for a Falling Body
    v = gt
where g is the acceleration due to gravity, v is a velocity, and t is an elapsed time.
```

An equation has to take the form of one formula equals another, where each formula is made up from symbols defined afterwards. The symbols can be defined as definite values (as "g" is defined in the Galilean Equation), or just by telling Inform their kinds of value (as "v" and "t" are defined).

Equations are read using standard mathematical conventions. So "x + yz" means that we multiply y and z, then add that to x; "ab/cd" divides the product of a and b by the product of c and d. Multiplication signs can be omitted, just as science books normally do (though we can always write them if we want to, using the asterisk *, as usual in computing). The need for brackets is minimised, with any luck, but we can use them if we need to: "x(y+ab)" is legal, for instance.

One difference between Inform's conventions and mathematical ones, though, is that Inform generally ignores upper-versus-lower-case when reading variable names, so it wouldn't be a good idea to write "F = gMm/r^2" and expect "M" and "m" to be different from each other.

Here is the calculation:

```
Instead of pushing the cannon ball:
    let the falling body be the cannon ball;
    let m be the mass of the falling body;
    let h be 1.2m;
    let F be given by Newton's Second Law where a is the acceleration due to gravity;
    let v be given by the Principle of Conservation of Energy;
    let t be given by the Galilean Equation for a Falling Body;
    say "You push [the falling body] off the bench, at a height of [h], and, subject to a downward force of [F], it falls. [t to the nearest 0.01s] later, this mass of [m] hits the floor at [v].";
    now the falling body is in the location.
```

And the result is:

```
You push the cannon ball off the bench, at a height of 1.2m, and, subject to a downward force of 19.614N, it falls. 0.49s later, this mass of 2.0kg hits the floor at 4.85147 m/s.
```

Not all that fast-moving - it's only about 10 mph, ten times slower than one fired by a Renaissance cannon - but half a second wouldn't give you long to get your foot out of the way.

How was that done? The crucial lines are the ones in the form "let X be given by E...", which is a new form of "let".

let (a name not so far used) **be given by** (equation name)

or...

let (a temporary named value) **be given by** (equation name)

This phrase creates a new temporary variable, starting it with the value found by solving the given equation. The variable lasts only for the present block of phrases, which certainly means that it lasts only for the current rule. Example:

```
let F be given by Newton's Second Law where a is the acceleration due to gravity;
```

There is also a more compact syntax, giving the equation explicitly:

```
let KE be given by KE = mv^2/2 where KE is an energy;
```

When we solve with "let", then, all of the other symbols should either already have values (because they exist as "let" values already made) or else be specified in the line. For

instance,

```
let F be given by Newton's Second Law where a is the acceleration due to gravity;
```

is allowed because "F" is one of the symbols in "F = ma"; of the other two symbols, we have a "let" variable called "m" already - it's the mass of the cannon ball - and we declare exactly what "a" is.

The next calculation is more interesting:

```
let v be given by the Principle of Conservation of Energy;
```

Since the equation here is "mgh = mv^2/2", Inform has to do some algebra to work out "v" in terms of the other unknowns - it's the square root of 2gh, but we don't need to work that out. Inform can't always solve implicit equations - for instance, it can't deduce "m" from this equation - but it's correct on all the easy cases which occur in basic physics, and that enables us to write equations in their most natural form, which is easier to read and understand.

The advantage of setting out an equation formally is that it can be used in many places - we could use Newton's Second Law again for something quite different, for example. But it's a little cumbersome for something simple which we only need once, so this is neater:

```
let KE be given by KE = mv^2/2 where KE is an energy;
```

Here the equation is written out explicitly instead of being named, but otherwise everything works in the same way.

Equations can also contain many of our standard functions, which are written for this purpose with their standard mathematical abbreviations. For example:

```
let x be given by sin x = 1 where x is a real number;
```

works out x as pi divided by 4, which is to say, 90 degrees. The Phrasebook entries on the mathematical functions give their abbreviations, but here they all are as a list:

```
abs, root, ceiling, floor, int, log, exp, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh,
arcsinh, arccosh, arctanh
```

As an example, here's the definition of arcsinh given in the Standard Rules:

```
To decide which real number is the hyperbolic arcsine of (R - a real number):
    let x be given by x = log(R + root(R^2 + 1)) where x is a real number;
    decide on x.
```

Something to be a little cautious of: brackets are used in equations to group terms together, and do not mean function application, as they would in a C-like programming language. For example, "sin(1+x)/2" takes the sine of "(1+x)/2": if we want to halve the sine of "1+x", we have to write "(sin(1+x))/2".

Example

263. **Widget Enterprises**
Allowing the player to set a price for a widget on sale, then determining the resulting sales based on consumer demand, and the resulting profit and loss.

RB 9.4 Money

§15.19 Arithmetic with units

The example equations in the previous section carried out quite a lot of arithmetic, but they may have given the impression that Inform always allows arithmetic - which is not true.

This is actually a good thing, because it keeps us from error. For instance, Inform will not allow:

```
Equation - Newton's Totally Bogus Law
    F = m^2
where F is a force, m is a mass.
```

because whatever you get when you square a mass, you don't get a force - in the same way that a length times another length makes an area, not another length. Physicists call this "dimensional analysis", and it often provides clues about which equations are right. Just after the Second World War, someone correctly worked out the explosive power of an atomic bomb without any classified information simply by guessing what values would appear in the formula, and then finding the simplest equation they could appear in.

In general, Inform will not allow numerical kinds of value to be multiplied or divided by each other (or square or cube rooted) unless we give it instructions that this would make sense.

Of course, there's plenty we can still do without any need for such instructions. For instance, going back to weight,

```
The Weighbridge is a room. "A sign declares that the maximum load is [100kg multiplied by 3]."
```

…will produce the text "A sign declares that the maximum load is 300kg." Here Inform knows that it makes sense to multiply a weight by 3, and that the result will be a weight. Similarly, Inform allows us to add and subtract weights, and several different forms of division are allowed:

```
The blackboard is in the Weighbridge. "A blackboard propped against one wall reads: '122 / 10 is [122 divided by 10] remainder [remainder after dividing 122 by 10]; but 122kg / 10kg is [122kg divided by 10kg] remainder [remainder after dividing 122kg by 10kg]; and 122kg / 10 is [122kg divided by 10] remainder [remainder after dividing 122kg by 10].'"
```

When we visit the Weighbridge, we find:

```
A blackboard propped against one wall reads: "122 / 10 is 12 remainder 2; but 122kg / 10kg is 12 remainder 2kg; and 122kg / 10 is 12kg remainder 2kg."
```

Whereas we are not allowed to divide 122 by 10kg: that would make no sense, since 122 is a number and not made up of kilograms. Inform will produce a problem message if we try. Similarly, Inform won't normally allow us to multiply two weights together - but see the next section.

Examples

264. **Frozen Assets**
A treatment of money which keeps track of how much the player has on him, and a BUY command which lets him go shopping.
RB 9.4 Money

265. **Money for Nothing**
An OFFER price FOR command, allowing the player to bargain with a flexible seller.
RB 9.4 Money

266. **Lemonade**
Containers for liquid which keep track of how much liquid they are holding and of what kind, and allow quantities to be moved from one container to another.
RB 10.2 Liquids

267. **Savannah**
Using the liquid implementation demonstrated in Lemonade for putting out fires.
RB 10.2 Liquids

§15.20 Multiplication of units

To recap, then, it is forbidden to multiply 122kg and 10kg, not because it could never make sense (a scientist might occasionally multiply two weights) but because the result is - what? Not a number, and not a weight any more. But we are allowed to tell Inform what the result ought to be, and once we have done so, the multiplication will be allowed:

```
A length is a kind of value. 10m specifies a length. An area is a kind of value. 10 sq m specifies an area.

A length times a length specifies an area.

The balance platform is in the Weighbridge. "The balance platform is 10m by 8m, giving it an area of [10m multiplied by 8m]."
```

which will turn up as:

```
The balance platform is 10m by 8m, giving it an area of 80 sq m.
```

And having told Inform that lengths multiply to area, we could also divide an area by a length to get a length: no further instructions would be needed.

The built-in "Metric Units" extension includes all of the standard ways that physical

quantities are multiplied, and a good way to see these is to try out one of the Metric Units examples and look at the Kinds index, which includes a table showing how all of this works.

Examples

268. **Depth**
   Receptacles that calculate internal volume and the amount of room available, and cannot be overfilled.
   RB 10.5 Volume, Height, Weight

269. **Fabrication**
   A system of assembling clothing from a pattern and materials; both the pattern and the different fabrics have associated prices.
   RB 9.4 Money

270. **The Speed of Thought**
   Describing scientifically-measured objects in units more familiar to the casual audience.
   RB 10.5 Volume, Height, Weight

16. Tables

§16.1 Laying out tables
§16.2 Looking up entries
§16.3 Corresponding entries
§16.4 Changing entries
§16.5 Choosing rows
§16.6 Repeating through tables
§16.7 Blank entries
§16.8 Blank columns
§16.9 Blank rows
§16.10 Adding and removing rows
§16.11 Sorting
§16.12 Listed in...
§16.13 Topic columns
§16.14 Another scoring example
§16.15 Varying which table to look at
§16.16 Defining things with tables
§16.17 Defining values with tables
§16.18 Table continuations
§16.19 Table amendments

§16.1 Laying out tables

When printed books need to display detailed information in a systematic way, they break off from running text and print a table instead. Inform does the same. Here is a typical example:

Table 2.1 - Selected Elements

Element	Symbol	Atomic number	Atomic weight
"Hydrogen"	"H"	1	1
"Iron"	"Fe"	26	56
"Zinc"	"Zn"	30	65
"Uranium"	"U"	92	238

After the two titling lines, each line represents one row in the table, and entries on a line must be separated by at least one tab character. A table must occupy a single whole paragraph, with no skipped lines or missing entries.

The top line is a title, the first word of which must be the word 'Table'. We can then either give a table number (this need not actually be a number: Table C2, or some such, would be fine), or give a name, or both - as in this case. The possible titling formats are:

```
Table 2.3
Table of Population Statistics
Table 2.3 - Population Statistics
```

In the last example we could call the table either "Table 2.3" or "Table of Population Statistics".

Each column then has a name, and the contents must all be the same kind of value. In the elements table the "Symbol" column contains only text, for instance, and the "Atomic weight" column contains only numbers. Any kinds of value will do, so long as all the entries in the column are mutually compatible. (For instance, mixing rooms and things in a single column would be fine, as these can be reconciled, but mixing numbers and rooms would not.)

§16.2 Looking up entries

The simplest way to access the information inside tables is to ask explicitly for it, specifying the row number, the column name and what table is to be consulted. So, given our example table

Table 2.1 - Selected Elements

Element	Symbol	Atomic number	Atomic weight
"Hydrogen"	"H"	1	1
"Iron"	"Fe"	26	56
"Zinc"	"Zn"	30	65
"Uranium"	"U"	92	238

we can write the following description:

```
symbol in row 3 of the Table of Selected Elements
```

to produce the value "Zn". Or the following will run off some chemical data:

```
repeat with N running from 1 to the number of rows in the Table of Selected Elements:
	say "The atomic weight of [element in row N of the Table of Selected Elements] is [atomic weight in row N of the Table of Selected Elements]."
```

The result of which will be:

```
The atomic weight of Hydrogen is 1.
The atomic weight of Iron is 56.
The atomic weight of Zinc is 65.
The atomic weight of Uranium is 238.
```

Note that the first row in a table is row number 1, and that the last can be found with the phrase:

number of rows in/from (table name) ⇒ *number*

This phrase produces the number of rows (including any blank rows) in the given table. Example:

```
number of rows in the Table of Selected Elements
```

§16.3 Corresponding entries

Continuing our example of the elements:

Table 2.1 - Selected Elements

Element	Symbol	Atomic number	Atomic weight
"Hydrogen"	"H"	1	1
"Iron"	"Fe"	26	56
"Zinc"	"Zn"	30	65
"Uranium"	"U"	92	238

If we want to know the atomic number of Uranium, say, it seems artificial to have to talk about the particular row number where the information happens to be. So we are also allowed to cross-reference, like so:

```
the atomic number corresponding to a symbol of "Fe" in the Table of Selected Elements
```

This results in 26, and similarly

```
the symbol corresponding to an atomic number of 26 in the Table of Selected Elements
```

results in "Fe". But we have to be careful:

```
the element corresponding to an atomic number of 27 in the Table of Selected Elements
```

This is not allowed (it produces an error at run-time), because there is no row with atomic number 27 in this rather limited table. We can check this in advance with the condition:

```
if there is an element corresponding to an atomic number of 27 in the Table of Selected Elements ...
```

Or more simply:

```
if there is an atomic number of 27 in the Table of Selected Elements ...
```

The condition "if there is…" can be used with any reference to a table entry: for instance, "if there is a symbol in row 5 of the Table of Selected Elements" would be false, because there are only four rows.

Example

271. **Dubai**
   An elevator which connects any of 27 floors in a luxury hotel.
   RB 8.2 Ships, Trains and Elevators

§16.4 Changing entries

Here is another rather definitive, immutable-looking table:

Table 4 - Recent Monarchs

Name	Accession	Family
"Anne"	1702	Stuart
"George I"	1714	Hanover
"George II"	1720	Hanover
"George III"	1760	Hanover
"George IV"	1820	Hanover
"William IV"	1830	Hanover
"Victoria"	1837	Hanover
"Edward VII"	1901	Saxe-Coburg-Gotha
"George V"	1910	Windsor
"Edward VIII"	1936	Windsor
"George VI"	1936	Windsor
"Elizabeth II"	1952	Windsor

But table entries can be changed as freely as variables: that is, any value can be entered so long as it has the right kind. We cannot put a dynasty into the "Name" column, or text in the "Accession" column. The phrase needed is "now … is …", just as it is for properties or variables:

```
Dynasty is a kind of value. The dynasties are Stuart, Hanover, Saxe-Coburg-Gotha and Windsor.

The Table Office is a room. The Succession is in the Table Office. "The Succession, a ponderous list of English monarchs, takes pride of place."

Instead of examining the Succession:
    say "The Succession List runs as follows...";
    repeat with N running from 1 to the number of rows in the Table of Recent Monarchs:
        say "[accession in row N of Table 4]: [name in row N of Table 4] ([family in row N of Table 4])."

Instead of attacking the Succession:
    now the family corresponding to an accession of 1720 in the Table of Recent Monarchs is Stuart;
    now the name in row 4 of the Table of Recent Monarchs is "Graham I";
    now the name in row 5 of the Table of Recent Monarchs is "Trixibelle IV";
    say "You deface the English succession, making suitable amendments with a quill pen. Considering it is supposed to be mightier than the sword the effect is a little disappointing."

Test me with "examine succession / attack it / examine it".
```

Once we start changing tables, it sometimes becomes useful to check what they contain.

showme the contents of (table name)

This phrase prints a crude but sometimes useful display on screen of the current contents of the named table. It's intended for authors to see when testing, not for players of the finished version to see.

say "[current table row]"

This text substitution produces a crude but sometimes useful listing of the entries in the currently chosen table row.

say "[row (number) **in/from table** (table name)**]"**

This text substitution produces a crude but sometimes useful listing of the entries in the specified row.

say "[(column name) **in/from table** (table name)**]"**

This text substitution produces a crude but sometimes useful listing of the entries in the specified column.

§16.5 Choosing rows

The following would be one way to print out a list of recent Kings and Queens:

```
To list the succession:
say "The Succession List runs as follows...";
repeat with N running from 1 to the number of rows in the Table of Recent Monarchs:
    say "[accession in row N of the Table of Recent Monarchs]: [name in row N of the Table of Recent Monarchs] ([family in row N of the Table of Recent Monarchs])."
```

This works, but is repetitive. We often want to work on a single row for a while, either to change things or think about the contents, and it is tiresome to keep specifying the row over and over again. The following shorthand provides some relief:

choose a/the/-- row (number) **in/from** (table name)

This phrase selects the row with the given number. Row numbers in a table start from 1, so

```
choose row 1 from the Table of Recent Monarchs
```

selects the top row.

That allows us to improve the loop:

```
To list the succession:
    say "The Succession List runs as follows...";
    repeat with N running from 1 to the number of rows in the Table of Recent Monarchs:
        choose row N in the Table of Recent Monarchs;
        say "[accession entry]: [name entry] ([family entry]).";
```

Actually, as we'll see in the next section, this kind of loop is needed so often that there's a shorthand wording for it.

Note that since "accession" is a column name, "accession entry" means the entry in that

column of the currently chosen row. This notation can only be used if a "choose" has certainly already happened, and it is a good idea to make that choice somewhere close by in the source code (and certainly in the same rule or phrase definition) for the sake of avoiding errors. We can also choose rows by specifying something about them, like so:

choose a/the/-- row with (table column) **of** (value) **in/from** (table name)

This phrase selects the first row, working down from the top of the given table, in which the given column has the given value. Example:

```
choose row with a name of "Victoria" in the Table of Recent Monarchs;
```

A run-time problem message is produced if the value isn't found anywhere in that column.

Sometimes it will happen that a column's name clashes with the name of something else: for instance, if we call a column "apples" but we also have a kind called "apple", so that the word "apples" could mean either some fruit or the column. Inform will generally prefer the former meaning as more likely. In case of such trouble, we can simply refer to "the apples column" rather than just "the apples": for instance, "choose row with an apples column of…" rather than "choose row with an apples of…"

We can also choose a row quite at random:

choose a/the/-- random row in/from (table name)

This phrase makes a uniformly random choice of non-blank rows in the given table. Note that although a table always has at least one row, it can't be guaranteed that it always has a non-blank row, so it's possible for this to fail: if it does, a real-time problem message is thrown.

§16.6 Repeating through tables

We very often want to run through a table doing something to, or with, each row in turn, so a special loop is provided for this. Rather than having to write all this out:

```
To list the succession:
    say "The Succession List runs as follows...";
    repeat with N running from 1 to the number of rows in the Table of Recent Monarchs:
        choose row N in the Table of Recent Monarchs;
        say "[accession entry]: [name entry] ([family entry])."
```

We can simply use this instead:

repeat through (table name):

This phrase causes the block of phrases following it to be repeated once for each row in the given table, choosing each row in turn, from top to bottom. Blank rows are skipped. Example:

```
To list the succession:
    say "The Succession List runs as follows...";
    repeat through the Table of Recent Monarchs:
        say "[accession entry]: [name entry] ([family entry])."
```

Note that there is no loop variable here, unlike in other forms of "repeat", because it's the choice of row which keeps track of how far we have got.

We can alternatively go backwards:

repeat through (table name) **in reverse order:**

This phrase causes the block of phrases following it to be repeated once for each row in the given table, choosing each row in turn, from bottom to top. Blank rows are skipped.

More often we want a sequence which is neither forwards nor backwards, but which depends on the actual values in the table.

repeat through (table name) **in** (table column) **order:**

This phrase causes the block of phrases following it to be repeated once for each row in the given table, choosing each row in turn, in order of the values in the given column. Blank rows are skipped. Example:

```
repeat through the Table of Recent Monarchs in name order: ...
repeat through the Table of Recent Monarchs in accession order: ...
```

work through the same table in rather different orders. The sequence is lower to higher (small numbers to high numbers, A to Z, and so on); insert "reverse" after "in" to reverse this.

repeat through (table name) **in reverse** (table column) **order:**

This phrase causes the block of phrases following it to be repeated once for each row in the given table, choosing each row in turn, in order of the values in the given column. Blank rows are skipped. Example:

```
repeat through the Table of Recent Monarchs in reverse name order: ...
repeat through the Table of Recent Monarchs in reverse accession order: ...
```

work through the same table in rather different orders. The sequence is higher to lower (high numbers to small numbers, Z to A, and so on); delete the "reverse" after "in" to reverse this.

In a loop like this, the data is not searched very efficiently, which is fine for modest-sized tables like the examples in this chapter, but might be a problem for much larger tables: see the later section on sorting.

These definitions mentioned blankness several times, and that's the topic to cover in the next section.

See Also

Sorting for reordering a table to put it into increasing or decreasing order of the entries in any column.

Example

272. **Port Royal 4**
   A cell window through which the player can see people who were in Port Royal in the current year of game-time.

RB 3.6 Windows

§16.7 Blank entries

We are allowed to leave certain entries blank (perhaps to be filled in later, perhaps not) by writing "--" instead of the relevant value:

Table 2.1 - Selected Elements

Element	Symbol	Atomic number	Atomic weight
"Hydrogen"	"H"	1	1
"Iron"	"Fe"	--	56
"Zinc"	--	30	65
"Uranium"	"U"	92	238

In effect, blank entries don't exist. "--" is not a value, but only a hole where a value might be. It can be useful to check for this:

if there is (a table entry):

This condition is true if the entry referred to exists, that is, that is, the space for it in the table is not blank. Examples:

```
if there is a symbol corresponding to an atomic number of 30 in the Table of Selected Elements ...
if there is an atomic number in row 2 of the Table of Selected Elements ...
```

if there is no (a table entry):

This condition is true if the entry referred to does not exist, that is, the space for it in the table is blank. Examples:

```
if there is no symbol corresponding to an atomic number of 30 in the Table of
Selected Elements ...
if there is no atomic number in row 2 of the Table of Selected Elements ...
```

§16.8 Blank columns

An entire column of blank entries "--" is problematic:

Table 2 - Selected Elements

Element	Symbol	Atomic number	Atomic weight
"Hydrogen"	"H"	1	--
"Iron"	"Fe"	26	--
"Zinc"	"Zn"	30	--
"Uranium"	"U"	92	--

Inform is unable to work out what kind of value should go into the "atomic weight" column here, since it has no examples to guess from. We can get around this by writing in the name of a kind of value:

Table 2 - Selected Elements

Element	Symbol	Atomic number	Atomic weight
"Hydrogen"	"H"	1	a number
"Iron"	"Fe"	26	--
"Zinc"	"Zn"	30	--
"Uranium"	"U"	92	--

That top entry in the "atomic weight" column is also blank, but now Inform knows that anything put into the column in future will be a number.

If there are many rows, and perhaps several blank columns, it would become very tedious to have to keep typing out "--". So this is optional *at the end of a row*: it remains compulsory for a blank value appearing in between two values which aren't blank. This is

the general idea:

Table 2 - Selected Elements

Element	Symbol	Atomic number	Density	Specific gravity
"Hydrogen"	"H"	1	a number	a number
"Iron"	"Fe"	26		
"Zinc"	"Zn"	30		
"Uranium"	"U"	92		

§16.9 Blank rows

There is no difficulty about entirely blank rows: or rather, the only difficulty is once again that they are boring to type out. We can avoid the necessity by appending "with ... blank rows" at the foot of the table:

Table 2 - Selected Elements

Element	Symbol	Atomic number	Atomic weight
"Hydrogen"	"H"	1	a number
"Iron"	"Fe"	26	--
"Zinc"	"Zn"	30	--
"Uranium"	"U"	92	--

with 3 blank rows

(These words cannot be placed in between rows, but only at the bottom.) And indeed the table can start out completely empty:

Table 3 - Undiscovered Periodic Table

Element (text)	Symbol (text)	Atomic number (a number)	Atomic weight (a number)

with 92 blank rows

Blank rows are useful because they enable us to add new data to a table. In effect, they are invisible when not used. A repeat loop like

```
repeat through Table 3:
    ...
```

automatically skips blank rows, so it would initially do nothing at all. Similarly, choosing

a "random" row will never choose a blank one.

A convenient way to test if a table contains non-blank rows is to use the built-in adjectives "empty" and "non-empty". So:

```
if the Undiscovered Periodic Table is empty, ...
```

tests whether all of its rows are blank; if even one cell contains a value then the table is "non-empty".

Example

273. **If It Hadn't Been For...**
A sound recording device that records the noises made by player and non-player actions, then plays them back on demand.
RB 9.12 Cameras and Recording Devices

§16.10 Adding and removing rows

Writing in new rows is simple, once we can find space for them:

choose a/the/-- blank row in/from (table name)

This phrase chooses a row in the given table which is currently blank under every column. A run-time problem message is issued if no rows are blank. Example:

```
choose a blank row in Table 3;
now element entry is "Fluorine";
now symbol entry is "F";
now atomic number entry is 9;
now atomic weight entry is 19;
```

To avoid problem messages, it can be important to worry about free space. To that end we can not only find the number of rows (as we have already seen) but also the number currently blank and not blank:

number of blank rows in/from (table name) ⇒ *number*

This phrase produces the number of rows in the given table which are entirely blank (that is, blank under every column).

number of filled rows in/from (table name) ⇒ *number*

This phrase produces the number of rows in the given table which are not entirely blank (that is, at least one column has a value in this row).

"Filled" here really means "non-blank": a row can be filled in this sense even if only one of its values exists. Since every row is either blank or filled, it must be true that:

```
the number of blank rows in Table 3
the number of filled rows in Table 3
```

add up to "the number of rows in Table 3".

We've seen that blank entries can be filled with values using "now":

```
now symbol entry is "F";
```

But the same method can't be used to put blanks back, since a blank is not a value. Instead:

blank out (a table entry)

This phrase replaces the entry referred to with a blank, erasing any value previously stored there. Example:

```
choose row 1 in the Table of Fish Habitats;
blank out the salinity entry;
```

These more destructive phrases need a steady hand:

blank out the whole row

This phrase replaces the currently chosen row with blanks, erasing any value previously stored under any of the columns. Example:

```
choose row 1 in the Table of Fish Habitats;
blank out the whole row;
```

blank out the whole (table column) **column in** (table)

This phrase replaces the currently chosen column with blanks, erasing any value previously stored in any of the rows. Example:

```
blank out the whole salinity column in the Table of Fish Habitats;
```

blank out the whole of (table)

This phrase replaces every row of the currently chosen table with blanks, erasing any value previously stored anywhere in it. Example:

```
blank out the whole of the Table of Fish Habitats;
```

This is only really useful when a Table is being used to hold working space for some calculation or other.

Example

274. **Odyssey**
A person who follows a path predetermined and stored in a table, and who can be delayed if the player tries to interact with her.
RB 7.13 Traveling Characters

§16.11 Sorting

The three ways to sort a table correspond loosely to the three different orders in which tables can be repeated through. First:

sort (table name) **in random order**

This phrase rearranges the rows of the given table so that the non-blank rows occur at the top, in a uniformly random order, and any blank rows at the bottom. Example:

```
sort the Table of Recent Monarchs in random order;
```

Secondly:

sort (table name) **in** (table column) **order**

This phrase rearranges the rows of the given table so that the non-blank rows occur at the top, so that the given column has ascending order, and any blank rows at the bottom. Example:

```
sort the Table of Recent Monarchs in accession order;
```

Ascending order means 1 up to 10, say, or A up to Z, with blank values coming last.

sort (table name) **in reverse** (table column) **order**

This phrase rearranges the rows of the given table so that the non-blank rows occur at the top, so that the given column has descending order, and any blank rows at the bottom. Example:

```
sort the Table of Recent Monarchs in reverse name order;
```

Descending order means 10 down to 1, say, or Z down to A, with blank values coming last.

How sorting is done depends on the contents of the column being sorted on. If it holds numbers then numerical order is used, with 2 coming before 7, and so on. (And similarly for real numbers, though the existence of infinities makes this more interesting.) If times are sorted then they are sorted from midnight to midnight, following the "is greater than" relation, not with 4 AM as the zero point, as with "is after".

If text is sorted then alphabetical order is used, though this doesn't always come out the way you might expect, because upper case and lower case letters are treated as different: A-Z come before a-z, and accented letters such as é come after the regular alphabet. (What's happening here is that Inform is sorting on raw character values, not performing the full Unicode collation algorithm, which would be too slow at run-time.)

Note that blank values will always be placed below non-blank ones, and entirely blank rows last of all. This is true even if we use "reverse".

The method of sorting is "stable", that is, if two rows have the same value then they will stay the same way round in the sorted table, rather than being swapped over. For example, if we sort this into reverse index order:

Index	Comment
1	"Originally row 1"
2	"Originally row 2"
2	"Originally row 3"
3	"Originally row 4"

then we get

Index	Comment
3	"Originally row 4"
2	"Originally row 2"
2	"Originally row 3"
1	"Originally row 1"

As a result note that repeating through this sorted table goes through the original rows in order 4, 2, 3, 1; whereas repeating through the original table in reverse order goes through in order 4, 3, 2, 1. (This is all to explain the word "loosely" in the opening sentence of this section.)

Example

275. **Jokers Wild**
A deck of cards which can be shuffled and dealt from.
RB 9.5 Dice and Playing Cards

§16.12 Listed in...

Tables are especially useful for combining a run of basically similar rules in a simple and concise way. The "listed in" condition, as in

```
if the newfound object is an item listed in the Table of Treasures...
```

looks through a given table (here "table of treasures"), in a given column ("item"), to see if a given value is present ("the newfound object"). If this is successful, the row where it was found is automatically chosen; but if not, note that any existing row selection will be lost, so make use of the row only if the test succeeds.

We can similarly use "… listed in …" in a description used when specifying an action. Thus:

```
After taking an item listed in the Table of Treasures:
    if there is no time entry:
        now the time entry is the time of day;
        increase the score by the value entry;
    say "Taken!"
```

This assumes a table in the following shape:

Table of Treasures

Item	Value	Time
brooch	5	a time
tiara	8	--
coronet	10	--

In effect the table has allowed us to combine three very similar rules into one. The time column records the first time at which the item has been picked up, which starts out blank since at the start of play it has never been picked up. This enables us to award the appropriate number of points on the first occasion only.

Example

276. **Noisy Cricket**
    Implementing liquids that can be mixed, and the components automatically recognized as matching one recipe or another.

RB 10.2 Liquids

§16.13 Topic columns

When double-quoted matter appears in a column of a table, Inform will normally treat that as text for printing out. The exception is when the column is called "topic", where it is treated as text for comparing against what the player has typed. There is really only one operation allowed with topic columns, the "…listed in…" construction, but fortunately it is the one most often needed.

Let us suppose that the Sybil has a penchant for telling passers-by which is the Greek muse for what. We might write:

```
After asking the Sybil about a topic listed in the Table of Sybil's Replies, say "The Sybil declaims for a while, the gist being that the muse in question looks after [muse entry]."
```

We can then provide a simple table giving her responses:

Table of Sybil's Replies

Topic	Muse
"calliope"	"epic poetry"
"clio"	"history"
"erato"	"love poetry"
"euterpe"	"music"
"melpomene"	"tragedy"
"polyhymnia"	"sacred poetry"
"terpsichore"	"dancing"
"thalia"	"comedy"
"urania"	"astronomy"
"monica"	"tidiness"
"phoebe"	"massage"
"rachel"	"oval hair-cuts"

Topics can use the full range of abilities of the "understanding" system which Inform uses to parse text, and which will be the subject of a later chapter. For now, note that the Sybil's topics might equally include "flora/eve" (matching the single word "flora" or the single word "eve"), or something more elaborate such as:

```
"Bridget" or "Bridge" or "Bridget Jones"
```

See Also

Understand for the system Inform uses to parse text.

Examples

277. **Merlin**
A REMEMBER command which accepts any text and looks up a response in a table of recollections.
RB 5.4 Background

278. **Questionable Revolutions**
An expansion on the previous idea, only this time we store information and let characters answer depending on their expertise in a given area.
RB 7.11 Character Knowledge and Reasoning

279. **The Queen of Sheba**
Allowing the player to use question words, and using that information to modify the response given by the other character.
RB 7.11 Character Knowledge and Reasoning

§16.14 Another scoring example

```
To record (T - text) as achieved:
    choose row with a citation of T in the Table of Tasks Achieved;
    if there is no time entry:
        now time entry is the time of day;
        increase the score by the points entry.
```

The phrase above expects to see a table like this one:

Table of Tasks Achieved

Points	Citation	Time
1	"pride"	a time
3	"anger"	
2	"avarice"	
4	"envy"	
1	"lust"	
2	"gluttony"	
3	"sloth"	

The middle column records the tasks to be achieved, the first column records the points on offer for each: the final column, initially blank, will store the times at which the tasks

are first achieved.

```
Before eating, record "gluttony" as achieved.
```

The first time we record "gluttony" as achieved, 2 points will be awarded and the time will be logged in the Table, but on all subsequent occasions nothing will happen. So the combination of the phrase and the Table will look after a scoring system based on achieving specific goals (probably not the seven deadly sins, of course). We can, if we choose, use the same system to display a log of recent accomplishments:

```
repeat through the Table of Tasks Achieved in reverse time order:
    say "[time entry]: [citation entry] ([points entry])."
```

Example

280. **Goat-Cheese and Sage Chicken**
Implementing a FULL SCORE command which lists more information than the regular SCORE command, adding times and rankings, as an extension of the example given in this chapter.

RB 11.4 Scoring

§16.15 Varying which table to look at

So far, we have always used fixed table names when referring to tables: for instance in source like "sort the Table of Recent Monarchs in accession order", we refer to the "Table of Recent Monarchs", a definite and explicitly named table.

With a little care, however, we are allowed to have variables which themselves hold the names of tables. This opens up the possibility of more elaborate ways of storing and interconnecting information in table form, but is probably best avoided until it becomes necessary.

For example, suppose we have two different tables with the same basic structure:

Table 1 - Nifty Opening Plays in US Scrabble

word	score
"muzjiks"	128

Table 2 - Nifty Opening Plays in UK Scrabble

word	score
"quartzy"	126
"squeezy"	126

We could then record which one of these tables to use in a variable:

```
The lexicon is a table name that varies. The lexicon is Table 1.
```

Note that for this purpose, the kind of value is a special kind called "table name", not "table". (The word "table" already has too many meanings and we must be careful to avoid ambiguities here.) We could make use of this as follows, for instance:

```
To flip tables:
    say "You exchange dictionaries, lexically crossing the Atlantic. ";
    if the lexicon is Table 1, now the lexicon is Table 2;
    otherwise now the lexicon is Table 1;
    choose a random row in the lexicon;
    say "Did you know that according to [the lexicon], [word entry] scores [score entry]?"
```

which produces text such as

```
You exchange dictionaries, lexically crossing the Atlantic. Did you know that according to Table 1 - Nifty Opening Plays in US Scrabble, muzjiks scores 128?
```

Example

281. **Farewell**
People who respond to conversational gambits, summarize what they said before if asked again, and provide recap of conversation that is past.

RB 7.8 Saying Complicated Things

§16.16 Defining things with tables

Suppose we need to create a collection of items which differ in their properties, but are basically part of a larger pattern. For instance, here we set up what we need to make a collection of coloured shirts:

```
A jersey is a kind of thing. A jersey is wearable. A jersey has a number called year established. A jersey has a text called citation. The description of a jersey is "Since [year established], the Tour de France has awarded this jersey to the [citation]."
```

Now we have the pattern, but making the actual shirts is tedious and repetitive:

```
The yellow jersey is a jersey. The year established of the yellow jersey is 1919. The citation of the yellow jersey is "race leader". The polkadot jersey...
```

And so on. Instead, we can use a table to abbreviate all of this:

```
"Tour des Maillots"

The Staging Area is a room. A jersey is a kind of thing. A jersey is wearable. Some jerseys in the Staging Area are defined by the Table of Honorary Jerseys. The description of a jersey is "Since [year established], the Tour de France has awarded this jersey to the [citation]."

Table of Honorary Jerseys
```

jersey	year established	citation
a yellow jersey	1919	"race leader"
a polkadot jersey	1933	"King of the Mountains"
a green jersey	1953	"highest point scorer on sprints"
a white jersey	1975	"best cyclist aged 25 or less"

The first column provides names for the new things to be created. Subsequent columns provide property values. Note that we did not need to say that jerseys have a number called "year established" because Inform is able to infer this from the column heading and the presence of numbers in the column; similarly for "citation". Lastly, note that if any entry is blank (written "--") then that particular property is simply not set for that particular item.

Note that Inform reads articles such as "the" or "a" in the first column just as it would when something is created with any other sentence.

It's even possible to define kinds this way, though it's rare to need to create many kinds at once. (See the worked example "Reliques of Tolti-Aph" at the Inform website. There's no special syntax needed: rather than saying "Some jerseys are defined by…" we would say "Some kinds of jersey are defined by…")

Examples

282. **Sweeney**
A conversation where each topic may have multiple questions and answers associated with it, and where a given exchange can lead to new additions to the list.
RB 7.8 Saying Complicated Things

283. **Introduction to Juggling**
Assortment of equipment defined with price and description, in a table.
RB 9.4 Money

§16.17 Defining values with tables

Just as we can define many similar things (or kinds) using a table, we can also define a whole run of new values. Again, this avoids unnatural prose like

```
The chemical elements are Hydrogen, Helium, Lithium, ..., and Ununquadium.
```

We can give these new values properties, too. For example:

```
Solar distance is a kind of value. 1000 AU specifies a solar distance. Planet is a kind of value. The planets are defined by the Table of Outer Planets.

Table of Outer Planets
```

planet	semimajor axis
Jupiter	5 AU
Saturn	10 AU
Uranus	19 AU
Neptune	30 AU
Pluto	39 AU

creates five values of the kind "planet", but it also makes a property called "semimajor axis" which belongs only to these five values. Thus:

```
say "Pluto orbits at [semimajor axis of Pluto]."
```

produces "Pluto orbits at 39 AU." We can both use and change this value:

```
Praying is an action applying to nothing. Understand "pray" as praying.
Instead of praying:
    now the semimajor axis of Pluto is 1 AU;
    say "Your prayers are answered, and the Almighty moves Pluto in closer to the fire."
```

Similar properties would be made for each column of the table after the first (there can be any number of properties, including none). Because the values are created first, before the rest of the table is gone through, we can even use "planet" as one of the values of properties:

Table of Outer Planets

planet	semimajor axis	centre of government
Jupiter	5 AU	Jupiter
Saturn	10 AU	Saturn
Uranus	19 AU	Saturn
Neptune	30 AU	Pluto
Pluto	39 AU	Pluto

All of this is intended to be closely parallel to defining a whole run of things, such as the coloured jerseys, using a table, but there are two important restrictions: firstly, when a kind of value is defined by table, the table must contain all of its possible values; and secondly, the column names (after the first) cannot coincide with names of any properties held by any other value (or thing, for that matter). So it is a good idea to give the columns very specific names ("centre of government") rather than vague names which might cause clashes elsewhere ("owner").

Two technical footnotes. In a table used to define a kind of value, blank entries are not left blank: they are filled in with suitable default values. For instance, if the semimajor axis column had been all "--"s except for listing Neptune at "30 AU", say, Inform would deduce that the column was meant to hold a value of kind "solar distance", and would set the solar distances for all of the other planets to be "0 AU". It does this to ensure that "solar distance of P" exists for any planet P.

The second technical note is that we must not sort such a table, because it is used during play to store the properties, and if it were to get rearranged then so would the properties be - with probably disastrous results.

§16.18 Table continuations

A table is an arrangement for putting information together concisely in a single place, so

it might seem odd that we sometimes need to divide it up: but once in a while, we do. Suppose we have:

Table of Outer Planets

planet	semimajor axis
Jupiter	5 AU
Saturn	10 AU
Uranus	19 AU
Neptune	30 AU
Pluto	39 AU

But then someone in Chile with a telescope the size of God's own teacup notices something a long, long way out, and the newspapers get terribly excited. We can write an addendum:

Table of Outer Planets (continued)

planet	semimajor axis
Orcus	39 AU
Quaoar	43 AU
Xena	68 AU
Sedna	524 AU

This may seem unnecessary - why not simply add extra rows to the original table? - but it allows us to split the table between different parts of the source text, if we want to, or to continue a table which exists only in an extension. (Thus if we were using an extension which involved the planets, and had a table like this one, we would be able to add new planets without changing the extension.)

The name for the continuation must be identical to the original. The continuation has no existence in its own right: Inform simply splices the two (or more) pieces together, exactly as if the table were all in one piece at the place where it first occurred. Thus the above creates only one table, the "Table of Outer Planets", with nine rows. Each column in the continuation must exist in the original, but not every column need be given: those omitted are filled with blank entries. The columns need not be in the same order. Both original and continuations are allowed to quote a number of blank rows: if so, the combined total is used.

At time of writing the International Astronomical Union has not yet consented to name

2003 UB313 after Xena, the Warrior Princess, but this is surely only a bureaucratic delay. (Footnote: on 24 August 2006 it was demoted to dwarf planet status, like the luckless Pluto, and on 13 September renamed Eris; though its moon's official name, Dysnomia, is an ingenious double-meaning to do with the name of Xena's actress, Lucy Lawless.)

Example

284. **Food Network Interactive**
     Using a menu system from an extension, but adding our own material to it for this game.

RB 11.3 Helping and Hinting

§16.19 Table amendments

Tables can have amendments as well as continuations. The arrangement is much the same: a supplementary table supplies new rows for the original table. But instead of adding the new rows at the end of the original, as a continuation would, an amendment replaces matching rows in the original. (So the original stays the same size.)

The amendment table must have exactly the columns of the original and in the same order. Moreover, each row in the amended table must match exactly one row in the original. For instance:

Table of Plans

moment	outcome
10 AM	"takeover of Mars"
11:30 AM	"canals reflooded"
11:45 AM	"chocolate bar production doubled"

Table of Plans (amended)

moment	outcome
11:45 AM	"volcanic cave production doubled"

creates a three-row Table of Plans, with reference to the chocolate bars struck out.

Amendment rows may be given in any order. The process of matching a row begins at the left-most column: Inform tries to see if any single row in the original table has a matching entry. If none does, a Problem is issued. If more than one do, Inform then looks at the second column, and so on. For instance:

```
Enthusiasm is a kind of value. The enthusiasms are pumped, wired and languid.

Table of Mental States
```

feeling	extent	consequence
pumped	1	"you feel able to run for your life"
pumped	2	"you feel able to run for President"
wired	1	"you feel able to run"
languid	1	"you feel"

```
Table of Mental States (amended)
```

feeling	extent	consequence
pumped	2	"you feel able to run for the Nebraska State Legislature"

Here the amendment is made to the second row of the original table. The value in the leftmost column, "pumped", matches two rows in the original, so Inform moves on to the next column, reads "2", and finds that only one row in the original still qualifies - so that is the one replaced.

For the present, at least, the columns used for matching may only contain: numbers, times, objects, action names, activities, figure names, sound names, truth states and any new kinds of value or units which have been declared.

Example

285. **Trieste**
Table amendment to adjust HELP commands provided for the player.
RB 11.3 Helping and Hinting

17. Understanding

§17.1 Understand
§17.2 New commands for old grammar
§17.3 Overriding existing commands
§17.4 Standard tokens of grammar
§17.5 The text token
§17.6 Actions applying to kinds of value
§17.7 Understanding any, understanding rooms
§17.8 Understanding names
§17.9 Understanding kinds of value
§17.10 Commands consisting only of nouns
§17.11 Understanding values
§17.12 This/that
§17.13 New tokens
§17.14 Tokens can produce values
§17.15 Understanding things by their properties
§17.16 Understanding things by their relations
§17.17 Context: understanding when
§17.18 Changing the meaning of pronouns
§17.19 Does the player mean...
§17.20 Multiple action processing
§17.21 Understanding mistakes
§17.22 Precedence

§17.1 Understand

During play, the computer and the player alternate in writing messages to each other: in the player's case, these are short instructions, usually saying what to do next. A wide range of such "commands" are automatically understood, but these only apply to the standard built-in actions. (This wide range is conveniently tabulated in the Commands part of the Actions index.) If we want the player to be able to command new actions, then we need to specify what is to be understood as what. For this, we supply special sentences starting with the word "Understand".

Suppose we return to the earlier example of a newly created action:

```
Photographing is an action applying to one visible thing and requiring light.
```

We then supply lines of grammar (as they are called) for Inform to recognise, like so:

```
Understand "photograph [someone]" as photographing.
Understand "photograph [an open door]" as photographing.
```

As usual, the square brackets indicate something which stands for text, rather than text to be taken verbatim. "[someone]" needs to be the name of anything of the kind "person", for instance (though as usual that person will need to be in sight of the player for the name to be accepted). The first word - in these examples "photograph" - must be something definite, not a substitution like this.

For obvious reasons, this pattern of words needs to match the expectations of the action. Photographing applies to "one visible thing" - the "visible" just means it does not need to be touched, only seen - so neither of these would be allowable:

```
Understand "photograph" as photographing.
Understand "photograph [someone] standing next to [something]" as photographing.
```

The first is probably bad because it supplies no things at all, the second is certainly because it supplies two: what we want, of course, is just the one. (The reason the first is only probably bad is that it's possible to tell Inform how to choose the object if the player doesn't: see the "supplying a missing noun" activity.)

Examples

286. **Indirection**
    Renaming the directions of the compass so that "white" corresponds to north, "red" to east, "yellow" to south, and "black" to west.
    RB 3.2 Map

287. **XYZZY**
    Basics of adding a new command reviewed, for the case of the simple magic word XYZZY.
    RB 6.13 Magic Words

288. **Xylan**
    Creating a new command that does require an object to be named; and some comments about the choice of vocabulary, in general.
    RB 10.2 Liquids

§17.2 New commands for old grammar

In the photography example, we are providing entirely new grammar for an action not ordinarily built in to Inform. But we often want simply to provide alternative grammar for

existing actions, or even to put new interpretations on commands that Inform already recognises. For instance:

```
Understand "deposit [something] in [an open container]" as inserting it into.
```

The inserting action is built in to Inform, but the command "deposit" is not, so this is created as new. It is occasionally useful to put a twist on this:

```
Understand "fill [an open container] with [something]" as inserting it into (with nouns
reversed).
```

The clause "(with nouns reversed)" tells Inform to exchange the two nouns parsed, which is necessary because the inserting action expects the noun to be the item and the second noun to be the container, not vice versa.

The following example:

```
Understand "access [something]" as opening.
```

might look as if it makes "access" behave just like "open" when the player types it, but that's not so: "open" can also be used in constructions like "open the door with the brass key", in which case it is understood as the unlocking action. We could add another line to make "access" behave this way too, but if what we really want is to make "access" behave just like "open", it's easier simply to say so:

```
Understand the command "access" as "open".
```

This is very useful when adding a new command which needs synonyms:

```
Understand the commands "snap" and "picture" as "photograph".
```

We can check the current stock of commands by looking at the table in the Actions index: for instance, before making "snap" synonymous with "photograph", it might be wise to check that it is not already defined as a command for breaking something.

Examples

289. **Alpaca Farm**
A generic USE action which behaves sensibly with a range of different objects.
RB 6.17 Clarification and Correction

290. **Anchorite**
By default, Inform understands GET OFF, GET UP, or GET OUT when the player is sitting or standing on an enterable object. We might also want to add GET DOWN and DOWN as exit commands, though.
RB 6.10 Entering and Exiting, Sitting and Standing

291. **Cloak of Darkness**
Implementation of "Cloak of Darkness", a simple example game that for years has been used to demonstrate the features of IF languages.
RB 3.7 Lighting

§17.3 Overriding existing commands

Suppose we are devising specialist commands for a game of whist, and we want "discard" as one of them. Looking at the table of commands in the Action index, we find that, inconveniently enough, "discard" already has a meaning: it is synonymous with "drop", and while that might be sensible most of the time, it is perfectly wrong now. We need a way to free up "discard" for our own use. We can do that by:

```
Understand the command "discard" as something new.
```

This cuts it loose, so to speak, and ready to be given new meanings. If we check the Actions index again, we find no mention of "discard" - it is now a blank slate - but "drop" is still exactly as it was. We could now say something like:

```
Understand "discard [something]" as discarding.
```

(If we had declared that "drop" was something new, the whole thing would have happened in reverse, with "discard" retaining all of the original grammar. Inform does not distinguish between a command and its synonym.)

The "… as something new" sentence works even for a command which did not exist anyway, for instance with:

```
Understand the command "zylqix" as something new.
```

Of course this does nothing - but it is intentional that it generates no problem messages: it means that the sentence can be used to force a command to be fresh and untouched by previous definitions, which might be useful when working with extensions by other people.

It is also possible to clear out all the commands leading to a given action:

```
Understand nothing as taking.
```

The commands "take" and "get" will still exist, but now they'll only have their other senses (for taking off clothes, for getting out of boxes).

Examples

292. **The Trouble with Printing**
    Making a READ command, distinct from EXAMINE, for legible objects.
    RB 9.6 Reading Matter

293. **Lanista 2**
    Randomized combat in which the damage done depends on what weapons the characters are wielding, and in which an ATTACK IT WITH action is created to replace regular attacking. Also folds a new DIAGNOSE command into the system.
    RB 7.5 Combat and Death

§17.4 Standard tokens of grammar

We have already seen "[something]" and "[someone]", which are standard examples of "tokens of grammar" - patterns matched by suitable named things. There are several other standard tokens, provided not so much from necessity but to allow the story parser to be more graceful and responsive. "[someone]" matches the same possibilities as "[a person]" would, but the parser handles it a little better in cases of failure. These special tokens are best explained by looking at some of the examples in the standard grammar, which can be browsed in the Index of any story.

```
Understand "wear [something preferably held]" as wearing.
```

Here we expect that the named item will be one that is held by the player, and the parser will use this to resolve ambiguities between names of things carried and not carried. (If the action is one which positively requires that its noun be something carried, a command matching this token against something not carried will generate an automatic attempt to take it.)

```
Understand "take [things]" as taking.
Understand "drop [things preferably held]" as dropping.
```

"[things]" is like "[something]" but allows a list of items, or a vague plural like "all", to be typed. The result will be a sequence of actions, one for each item thus described. "[things preferably held]" is the analogous token for "[something preferably held]".

```
Understand "take [things inside] from [something]" as removing.
```

"[things inside]" matches only what is inside the second-named thing, and ensures that (for instance) the command "take all from box" does not also try to take the box.

```
Understand "put [other things] in/inside/into [something]" as inserting it into.
```

Similarly, "[other things]" will allow anything except the second-named thing. (Like "[things inside]" it is really only needed for handling containers.)

Finally there is "[any things]", which should be used only with care. This is like "[things]" but with no restriction at all on where the item comes from: it might be invisible, or from a different room, or out of play altogether. If we use this, we had better remember that it would match ALL, with quite extravagant consequences.

Examples

294. **Shawn's Bad Day**
Allowing the player to EXAMINE ALL.
RB 6.15 Actions on Multiple Objects

295. **The Left Hand of Autumn**
The possibility of using a [things] token opens up some interesting complications, because we may want actions on multiple items to be reported differently from actions on just one. Here we look at how to make a multiple examination command that describes groups in special ways.
RB 6.5 Examining

§17.5 The text token

Most actions involve items: taking a vase, perhaps. As we shall see, they might also involve values, or a mixture of the two: turning a dial to 17 would involve both a thing (the dial) and a number (17). A few of Inform's built-in actions, however, can act on any text at all. For instance, asking the Sybil about the Persian army would involve a thing (the Sybil) and some text ("Persian army"). Inform does not try to understand

automatically what that text might mean, or to relate it to any items, places or values it knows about: instead, Inform leaves that to the specific story to work out for itself, since the answer is bound to depend on the context. (In the chapter on Tables, we saw ways to compile tables of responses to particular topics of conversation.)

The token for "accept any text here" is just "[text]". For instance, if we create an action with:

```
Getting help about is an action applying to one topic.
```

We can then provide grammar for this action like so:

```
Understand "help on [text]" as getting help about.
```

When text like this is successfully matched, it is placed in a value called "the topic understood". (The term "topic" is used traditionally, really: most of the times one needs this feature, it's for a topic of conversation, or a topic being looked up in a book.)

The fact that "[text]" can match anything means that it's difficult to tell which version of a command was intended if they disagree only from a "[text]" onwards. For example, given:

```
Yelling specifically is an action applying to one topic. Understand "yell [text]" as yelling
specifically. Understand "yell [text] at/to [someone]" as answering it that (with nouns
reversed).
```

…Inform will in fact try the second possibility first, as being the more specific, but the result may freeze out the first possibility altogether due to autocompletion of commands.

Examples

296. **Ish.**
A (very) simple HELP command, using tokens to accept and interpret the player's text whatever it might be.

RB 11.3 Helping and Hinting

297. **Nameless**
ASKing someone about an object rather than about a topic.

RB 7.7 Saying Simple Things

§17.6 Actions applying to kinds of value

Almost all actions apply to things: the player picks them up, pushes them, looks at them

and so on. We only occasionally need to recognise other kinds of value, but when we do, we can. For instance:

```
Adjusting it to is an action applying to one thing and one number.
Understand "adjust [something] to [a number]" as adjusting it to.
```

The substitution "[a number]" matches any number (actually any whole number that is not too large) typed by the player. Inform checks the various kinds being used to make sure that everything matches, so, for instance, this would be disallowed:

```
Understand "adjust [something] to [something]" as adjusting it to.
```

Examples

298. **Safety**
A safe whose dial can be turned with SPIN SAFE TO 1131, and which will open only with the correct combination.
RB 9.2 Bags, Bottles, Boxes and Safes

299. **Tom's Midnight Garden**
A clock kind that can be set to any time using "the time understood"; may be turned on and off; and will advance itself only when running. Time on the face is also reported differently depending on whether the clock is analog or digital.
RB 9.11 Clocks and Scientific Instruments

300. **Ibid.**
A system which allows the author to assign footnotes to descriptions, and permits the player to retrieve them again by number, using "the number understood". Footnotes will automatically number themselves, according to the order in which the player discovers them.
RB 12.3 Footnotes

§17.7 Understanding any, understanding rooms

Ordinarily, if we write

```
Understand "manipulate [something]".
```

then the "[something]" will only match what is within reach or sight: this is the concept of "scope", which is what prevents a player from spookily acting on objects from a distance. The parser itself prevents the manipulation rules from ever being invoked on such distant

items, which is as it should be.

Sometimes, though, we positively want to allow this possibility. If we use the special word "any", as in

```
Understand "manipulate [any door]".
```

then any door, anywhere in the model world, can be allowed in the player's command. (Of course, the manipulation rules may not do what the player hopes: all that has happened is that the command is now possible to type.) The "any" can be followed by any description of items or rooms, and the latter opens up new possibilities, since rooms are ordinarily never allowed to be named in the player's commands.

For example, the following gives the player the ability to walk between rooms without giving explicit directions of movement.

```
Going by name is an action applying to one thing.

Carry out going by name: say "You walk to [the noun]."; move the player to the noun.

Understand "go to [any adjacent visited room]" as going by name.
```

(This is really only a sketch: in a finished work, "go to" would produce helpful errors if non-adjacent but visited rooms were named, and we might also worry about rules applying to movement, because the method above will circumvent them.)

As might be expected, "[anything]" means the same as "[any thing]"; "[anybody]" and "[anyone]" mean the same as "[any person]"; and "[anywhere]" means the same as "[any room]".

Examples

301. **One of Those Mornings**
    A FIND command that allows the player to find a lost object anywhere
    RB 5.4 Background

302. **Actaeon**
    A FOLLOW command allowing the player to pursue a person who has just left the room.
    RB 7.13 Traveling Characters

§17.8 Understanding names

So far in this chapter, Understand sentences have been used to give names to actions, but

they can also be used to name objects - in particular, things and rooms.

This normally happens automatically. For instance, writing

```
The St Bernard is an animal in the Monastery Cages.
```

makes ST BERNARD refer to the dog, and MONASTERY CAGES refer to the room. But sometimes, as here, that isn't really enough. Why shouldn't the player type EXAMINE DOG? One way to allow this is to write:

```
Understand "dog" as the St Bernard.
```

Matters become more complicated when the player wants to refer to more than one object at once. When a kind is created, and the source text constructs multiple duplicate items of that kind, Inform generates a plural of the kind's name in order to understand commands referring to these multiples. For instance, given…

```
The Lake is a room. A duck is a kind of animal. Four ducks are in
    the Lake.
```

…the player can type TAKE DUCKS to try to pick up all four.

Once again the automatic behaviour can be enhanced:

```
Understand "birds" and "ruddy ducks" as the plural of duck.
```

Now TAKE BIRDS and TAKE DUCKS are equivalent. Plurals can even, strange as it may seem, be given for single things:

```
The magpie is in the Lake. Understand "birds" as the plural of the magpie.
```

And now TAKE BIRDS tries to take all four ducks and the magpie too.

§17.9 Understanding kinds of value

In many cases, if K is the name of a kind of value, then Inform automatically makes an Understand token called "[K]" which matches only values of K. An example is "[number]", which matches text like 203 or SEVEN. There is a chart of the kinds of value in the Kinds index for a project, showing which ones can be understood in this way.

In particular, any newly created kind of value can always be understood. We make good use of that in the example story "Studious":

```
"Studious"

The Studio is a room. "The unreal world of the photographic studio, full of fake furniture, cantilevered stands and silver-white shades like miniature parachutes." The lumpy black camera is in the Studio. "A lumpy black camera hangs from a tripod."

The rake-thin model is a woman in the Studio. "A rake-thin model, exquisitely bored and boringly exquisite, angles herself indolently."

Limb is a kind of value. The limbs are left leg, left arm, right leg and right arm.

Detailing is an action applying to one limb and one visible thing, requiring light. Check detailing: if the camera is not carried then say "You can hardly photograph without a camera, now can you?" instead. Report detailing: say "Click! You take a detail photograph of the [limb understood] of [the second noun]."

Understand "photograph [limb] of [a person]" as detailing.

Test me with "get camera / photograph left leg of model".
```

Note the way we can refer to the limb mentioned by the player as the "limb understood". Similarly, we could talk about the "number understood" if the value parsed had been a number, and so on.

One of the built-in kinds of value is worth special note: time. A time can hold either a specific time of day, such as 10:23 PM, or a duration of something, such as 21 minutes. The "[a time]" token matches times of day, such as 10:15 AM or MIDNIGHT. But 10 MINUTES wouldn't be recognised by "[a time]" since it isn't a specific moment in the day. To get around this, an alternative version called "[a time period]" is available. So:

```
Understand "wait for [a time period]" as ...
```

would match WAIT FOR AN HOUR or WAIT FOR TWO HOURS 12 MINUTES.

Examples

303. **Pages**
    A book with pages that can be read by number (as in "read page 3 in…") and which accepts relative page references as well (such as "read the last page of…", "read the next page", and so on).
    RB 9.6 Reading Matter

304. **Down in Oodville**
    Offering the player a choice of numbered options at certain times, without otherwise interfering with his ability to give regular commands.
    RB 6.18 Alternatives To Standard Parsing

305. **Straw Into Gold**
    Creating a Rumpelstiltskin character who is always referred to as "dwarf", "guy", "dude", or "man" -- depending on which the player last used -- until the first time the player refers to him as "Rumpelstiltskin".
    RB 2.1 Varying What Is Written

§17.10 Commands consisting only of nouns

In every example so far, and in almost all practical cases, the first word in a command which results in an action will be something fixed: a verb, in fact. When we write

```
Understand "photograph [something]" as photographing.
```

we are saying that the first word of such a command will always be "photograph". Occasionally, though, we would like to understand a noun as a command, perhaps in a situation where the command is obvious. If we say:

```
Understand "[something]" as examining.
```

then the command "examine" will be implicit when the player types a bare noun:

```
A red box and a blue ball are here.
> BALL
The blue ball is plaited from many small leather patches.
```

so that the command "ball" has resulted in the action "examining the blue ball".

This is a feature which should be used sparingly, since it could easily lead to confusion if not carefully explained to the player. By default, it is not used at all.

It also has what may be a serious limitation: verbless commands like this work only when typed by the player as actions to follow - they do not work as instructions for other people. So for instance SVEN, BALL would not ask Sven to try examining the ball - instead it would generate the action "answering ball to Sven". (This is because the Inform parser decides whether PERSON, SOME TEXT is a request or just conversation by looking at the first word after the comma to see if it's a command.)

Examples

306. **Misadventure**
A going by name command which does respect movement rules, and accepts names of rooms as commands.
RB 6.9 Going, Pushing Things in Directions

307. **Safari Guide**
The same functionality, but making the player continue to move until he reaches his destination or a barrier, handling all openable doors on the way.
RB 6.9 Going, Pushing Things in Directions

§17.11 Understanding values

"Understand" can be used to supply new ways to talk about both things and other values. For instance, if we create:

```
A brass lantern is in the Building.
```

then it can be called "brass", or "lantern", but not "lamp": Inform does not really know what these words mean, and has no grasp of synonyms. We can arrange for "lamp" to work as well like so:

```
Understand "lamp" as the lantern.
Understand "old lamp" as the lantern.
```

With care, we can do the same trick for entire kinds of thing at once. It is not ordinarily the case that a thing can be called by the name of its kind: if we put a woman called April into a room, then she can usually be called "April", but not "woman". (The exception is when we do not specify any name for her - in that case, Inform will give up and call her just "woman".) So there is not usually any form of words which can refer to anything of a given kind. If we should want this, we have to say so explicitly:

```
Understand "machine" as a device.
```

Device is a kind, so now the word "machine" can be used to refer to any device: if there are two in the same place, the result might play out like so:

```
>switch machine on
Which do you mean, the bale twiner or the grain thresher?
>twiner
You watch absorbed as a perfect cube of hay is trussed up like a parcel.
```

Similarly, we might conceivably want to allow new ways to recognise values - in this case, a number:

```
Understand "eleventy-one" as 111.
```

When making complicated names, we need to watch out for the possibility of writing a definition which will cause Inform to go around in circles (something which will show up as a "Too many activities at once" run-time problem). For instance,

```
Understand "[thing] substitute" as the placebo.
```

will fail because Inform, working left to right, needs to look for every possible object name before it can progress: one possibility is the placebo itself: to check that, it needs to look for every possible object name: and so on, never finishing. A definition like this one very likely matches too much in any case (would we really want to accept PLACEBO SUBSTITUTE or CIGARETTE SUBSTITUTE SUBSTITUTE SUBSTITUTE here, as the definition implies?).

Examples

308. **Palette**
An artist's workshop in which the canvas can be painted in any colour, and where painterly names for pigments ("cerulean") are accepted alongside everyday ones ("blue").
RB 9.7 Painting and Labeling Devices

309. **Baritone, Bass**
Letting the player pick a gender (or perhaps other characteristics) before starting play.
RB 5.2 Traits Determined By the Player

§17.12 This/that

We have already seen "or" used in "Understand" sentences:

```
Understand "scarlet" or "crimson" as red.
```

In general, any number of alternative forms can be given which are to be understood as the same thing (in this case the colour red). When the alternatives are in any way complicated, "or" should always be used, but a shorthand form is allowed for simple cases where it is only a matter of a single word having several possibilities:

```
Understand "reach underneath/under/beneath [something]" as looking under.
```

This is shorthand for:

```
Understand "reach underneath [something]" or "reach under [something]" or "reach beneath [something]" as looking under.
```

Which in turn is shorthand for:

```
Understand "reach underneath [something]" as looking under. Understand "reach under [something]" as looking under. Understand "reach beneath [something]" as looking under.
```

It's possible also to make that second word optional:

```
Understand "reach underneath/under/beneath/-- [something]" as looking under.
```

because "--" is read by Inform as "no word at all". If "--" is an option, it can only be given once and at the end of the list of possibilities.

To recapitulate: the slash "/" can only be used between single, literal words, and is best for the wayward prepositions of English ("in/into/inside", and so forth). For anything more complex, always use "or".

§17.13 New tokens

We have now made good use of square-bracketed tokens, such as "[something]", in a variety of "Understand…" sentences. It is sometimes convenient to create new tokens of our own, to match whatever grammar we choose: this enables complicated knots of grammar to be used in many different "Understand…" sentences without having to write it all out each time.

For instance, here are new tokens: one for each of two groups of alternative prepositions.

```
Understand "beneath/under/by/near/beside/alongside/against" or "next to" or "in front of" as "[beside]".

Understand "on/in/inside" or "on top of" as "[within]".
```

Again, note that the slash indicates a choice between words only, not between entire phrases. For instance, if we write:

```
Understand "red bird/robin" as "[robin]".
```

then the two alternative forms are "red bird" and "red robin", not "red bird" and "robin". By contrast,

```
Understand "red bird" or "robin" as "[robin]".
```

will understand either "red bird" or "robin" but not "red robin". If we want to capture all three forms, we might define

```
Understand "red bird/robin" or "robin" as "[robin]".
```

Example

310. **Lies**
Commands to allow the player to lie down in three different ways.
RB 6.10 Entering and Exiting, Sitting and Standing

§17.14 Tokens can produce values

The examples just seen were tokens which simply matched specific words typed by the player, but newly created tokens can also produce values:

```
Colour is a kind of value. The colours are red, green and blue. Understand "colour [a colour]" or "[a colour] shade" as "[tint]".
```

Here the "[tint]" token matches, for instance, "colour red" and "blue shade", which would result in the values red and blue, respectively.

Tokens are not allowed to produce more than one value, and if several patterns are given to define them then those patterns have to be compatible. That means the following is disallowed, since it might work out to a colour, or to an object, leaving Inform unable to judge whether an action can safely be applied to the result.

```
Understand "colour [a colour]" or "[something]" as "[tint]".
```

§17.15 Understanding things by their properties

Items are ordinarily understood only by their original given names. For instance, if we have:

```
In the Herb Garden is a china pot.
```

then the player could refer to this as "pot", "china pot" or "china". We can embellish this by adding extra forms:

```
Understand "chinese pot" or "chinese vase" as the china pot.
```

But suppose the pot changes its nature in the course of play? If we have:

```
The china pot can be unbroken or broken. The china pot is unbroken.

After dropping the china pot:
    say "Crack!";
    now the china pot is broken;
    now the printed name of the pot is "broken pot".
```

So now the player would reasonably expect to call it "broken pot", a wording which would have been rejected before. We can achieve this by writing:

```
Understand the unbroken property as describing the pot.
```

which allows "unbroken" or "broken" to describe the pot, depending on its state. And, since the player might well use a different adjective but with the same idea in mind, we can even add:

```
Understand "shattered" or "cracked" or "smashed" as broken. Understand "pristine" as unbroken.
```

This is something of a toy example, but the feature looks rather more useful when there are more pots than just one:

```
"Terracotta"

A flowerpot is a kind of thing. A flowerpot can be unbroken or broken. Understand the broken property as describing a flowerpot.

After dropping an unbroken flowerpot:
    say "Crack!";
    now the noun is broken;
    now the printed name of the noun is "broken flowerpot";
    now the printed plural name of the noun is "broken flowerpots".

The Herb Garden is a room. In the Herb Garden are ten unbroken flowerpots.
```

We then have the dialogue:

```
Herb Garden
You can see ten flowerpots here.
>get two flowerpots
flowerpot: Taken.
flowerpot: Taken.
>drop all
flowerpot: Crack!
flowerpot: Crack!
>look
Herb Garden
You can see two broken flowerpots and eight flowerpots here.
>get an unbroken flowerpot
Taken.
```

and so on and so forth.

There are in fact two slightly different forms of this kind of sentence:

```
Understand the broken property as describing a flowerpot.
Understand the broken property as referring to a flowerpot.
```

The only difference is that in the "describing" case, the property's name alone can mean the thing in question - so "take unbroken" will work; whereas, in the "referring to", the property's name can only be used as an adjective preceding the name of thing itself - so "take unbroken flowerpot" will work but "take unbroken" will not.

Examples

311. Aspect
Understanding aspect ratios (a unit) in the names of televisions.
RB 9.9 Televisions and Radios

312. Hymenaeus
Understanding "flaming torch" and "extinguished torch" to refer to torches when lit and unlit.
RB 3.7 Lighting

313. Channel 1
Understanding channels (a number) in the names of televisions.
RB 9.9 Televisions and Radios

314. Terracottissima
The flowerpots once again, but this time arranged so that after the first breakage all undamaged pots are said to be "unbroken", to distinguish them from the others.
RB 10.4 Glass and Other Damage-Prone Substances

315. Peers
The peers of the English realm come in six flavours - Baron, Viscount, Earl, Marquess, Duke and Prince - and must always be addressed properly. While a peerage is for life, it may at the royal pleasure be promoted.
RB 7.1 Getting Acquainted

316. Channel 2
Understanding channels (a number) in the names of televisions, with more sophisticated parsing of the change channel action.
RB 9.9 Televisions and Radios

317. Terracottissima Maxima
Flowerpots with textual names that might change during play.
RB 2.3 Using the Player's Input

318. Tilt 1
A deck of cards with fully implemented individual cards, which can be separately drawn and discarded, and referred to by name.
RB 9.5 Dice and Playing Cards

§17.16 Understanding things by their relations

Sometimes it makes sense for the name of something to involve the names of other things to which it is related. For instance, if we say TAKE THE BOTTLE OF WINE, we mean that the bottle currently contains wine - if it were the very same bottle containing water, we would call it something else.

For names which must involve related names, a special form of token is provided. For instance, we could say:

```
A box is a kind of container. Understand "box of [something related by containment]" as a box.

The Toyshop is a room. The red box is a box in the Toyshop. Some crayons are in the red box.
```

and now TAKE BOX OF CRAYONS will work, because CRAYONS matches against "[something related by containment]" for the red box - or it does for as long as the crayons are there. We can have similar matches against relations of all kinds, but have to name the relation explicitly. (See the examples at the end of this section for plenty of cases.)

We can also reverse the sense. If we write:

```
A box is a kind of container. Understand "box in [something related by reversed containment]" as a box.

The Toyshop is a room. The crate and the hammock are in the Toyshop. In the crate is a box. In the hammock is a box.
```

then TAKE THE BOX IN THE HAMMOCK will work: here, the relation goes the other way, because the box is being contained by the other-named item, rather than doing the containing.

Examples

319. **Cinco**
A taco shell that can be referred to (when it contains things) in terms of its contents.
RB 9.2 Bags, Bottles, Boxes and Safes

320. **Puncak Jaya**
When a character is not visible, responding to such commands as EXAMINE PETER and PETER, HELLO with a short note that the person in question is no longer visible.
RB 5.5 Memory and Knowledge

321. **Whither?**
A door whose description says where it leads; and which automatically understands references such as "the west door" and "the east door" depending on which direction it leads from the location.
RB 3.5 Doors, Staircases, and Bridges

322. **Claims Adjustment**
An instant camera that spits out photographs of anything the player chooses to take a picture of.
RB 9.12 Cameras and Recording Devices

§17.17 Context: understanding when

We have now seen several different forms of "Understand" sentence: for instance,

```
Understand the colour property as describing a building block.
Understand "mix [colour] paint" as mixing paint.
Understand "rouge" as red.
Understand "curious girl" as Alice.
```

Any of these may optionally have a condition tacked on: for instance,

```
Understand "mix [colour] paint" as mixing paint when the location is the Workshop.
Understand "rouge" as red when the make-up set is visible.
```

In principle, "when …" can take in any condition at all. In practice a little care should be exercised not to do anything too slow, or which might have side-effects. (For instance, referring the decision to a phrase which then printed text up would be a bad idea.) Moreover, we must remember that the "noun" and "second noun" are not known yet, nor do we know what the action will be. So we cannot safely say "when the noun is the fir cone", for instance, or refer to things like "the number understood". (We aren't done

understanding yet.) If we want more sophisticated handling of such cases, we need to write checking rules and so on in the usual way.

Contexts can be useful to make sense of things having different names depending on who is being spoken to, as here:

```
Understand "your" as a thing when the item described is held by the person asked.
```

With this rule in place FRODO, GIVE ME YOUR RING means that Frodo will know which ring is meant, even if there are a couple of dozen other rings present.

If the name of something has to change completely, perhaps because the player's understanding of events has changed completely, then Inform's standard way of handling names can be a nuisance. When an item or room is created, Inform automatically makes its name understood as referring to it (in fact, it makes each individual word in that name understood). For instance,

```
The Wabe is a room. The blue peacock and the sundial are in the Wabe.
```

means that the player can type EXAMINE BLUE PEACOCK or PUSH SUNDIAL or SHOWME WABE or TAKE BLUE, and so on. This is almost always a good thing, and here there's no problem, because peacocks and sundials are not usually disguised. But here is a case where a disguise is needed:

```
The secret document is a privately-named thing in the drawer.
The printed name of the secret document is "[if the secret document is handled]secret document[otherwise]dusty paper".
Understand "dusty" and "paper" as the secret document.
Understand "secret" and "document" as the secret document when the secret document is handled.
After taking the secret document for the first time: say "Heavens! It is the secret document!"
```

As this demonstrates, the either/or property "privately-named" makes Inform create a thing or room which starts out with no automatic understandings at all. The name it happens to have in the source text is ignored. If we simply write:

```
The ungraspable concept is a privately-named thing in the Dining Room.
```

then nothing the player can type will ever refer to it; though he will see it, and even be able to pick it up by typing TAKE ALL.

The reverse property is "publicly-named", which all things and rooms are by default.

Inform has four built-in kinds of object (room, thing, direction and region), and all of those have this either/or property. When we create new kinds, they're normally kinds of those four fundamental ones, so they pick up the same behaviour. But if we create a new kind of object outside of these four, that won't be true unless we make it so:

```
A concept is a kind of object. A concept can be privately-named or publicly-named. A concept is usually publicly-named.
```

(Privately-named is a property which only affects how Inform creates the object, and it can't usefully be given or taken away during play. "Understand … when …" is the way to change names during play.)

Examples

323. Quiz Show
In this example by Mike Tarbert, the player can occasionally be quizzed on random data from a table; the potential answers will only be understood if a question has just been asked.
RB 2.2 Varying What Is Read

324. Bibliophilia
A bookshelf with a number of books, where the player's command to examine something will be interpreted as an attempt to look up titles if the bookshelf is present, but otherwise given the usual response.
RB 9.6 Reading Matter

§17.18 Changing the meaning of pronouns

The pronouns IT, HIM, HER and THEM are constantly adjusted during play, to save the player time when typing commands. If the player types EXAMINE NECKLACE on one turn, it's sufficient to type TAKE IT on the next, and IT will be understood as meaning whatever NECKLACE meant last turn.

All of that happens automatically, but once in a while the result can be unfortunate. Suppose that when the player examines the necklace, a security system automatically drugs her unconscious, and she wakes up in a cell, hours later, and is told that the cell is bare except for a key on the floor. If she types TAKE IT, she clearly doesn't mean IT to mean the necklace any more; she means the key. Inform's parser can't make guesses like this, so the following phrase can be used to help it.

set pronouns from (object)

This phrase adjusts the meaning of pronouns like IT, HIM, HER and THEM in the command parser as if the object mentioned has become the subject of conversation. Example: the combination of

```
set pronouns from the key;
set pronouns from Bunny;
```

might change IT to mean the silver key and HIM to mean Harry "Bunny" Manders, while leaving HER and THEM unaltered.

Example

325. **Pot of Petunias**
Responding sensibly to a pot of petunias falling from the sky.
RB 2.2 Varying What Is Read

§17.19 Does the player mean...

When the player types an ambiguous reference, we need to work out what is meant. Consider the following source text:

```
The Champs du Mars is a room. The great Eiffel Tower is here. "The great Tower stands high over you." The souvenir model Eiffel Tower is here. "Comparatively tiny is the souvenir version."
```

Now suppose the player types GET TOWER. The response will be:

```
Which do you mean, the great Eiffel Tower or the souvenir model Eiffel Tower?
```

Which is a silly question, exposing our work of IF as something artificial. It's obvious to the author of the source text, and to the player, that the souvenir must be what is meant: but this is not obvious to the computer program running the story. Works of IF gain a subtle feeling of quality from being able to understand ambiguous references of the kind above, and Inform provides us with a way to do this by giving the parser clues in the form of "Does the player mean…" rules. For instance, if we add:

```
Does the player mean taking the great Eiffel Tower: it is very unlikely.
```

then the response to GET TOWER will now be:

```
(the souvenir model Eiffel Tower)
Taken.
```

"Does the player mean…" rules look at the actions which are possible interpretations of what the player typed, and grade them according to how likely they seem. (Note that these rules are only ever used to handle ambiguities: if the player unambiguously types GET GREAT EIFFEL TOWER, that will be the action. And the rules are only used where they are able to make a decision: if there are still multiple equally plausible meanings, the parser will ask about all possibilities, not just the most likely ones.) Rules in this rulebook can either decide nothing, or come up with one of the following verdicts:

```
it is very likely
it is likely
it is possible
it is unlikely
it is very unlikely
```

If there are no "does the player mean" rules, or the rules make no decision on a given possible action, it will be ranked as "it is possible".

We may use these rules to affect all sorts of interaction with a specific object or kind of object, as in

```
Does the player mean doing something with the cursed dagger of Thog: it is very unlikely.
Does the player mean doing something with the cursed dagger of Thog when the player
is hypnotized: it is likely.
```

…and so on.

Notice that we can also make rules about actions that apply to two objects, so for instance:

```
Does the player mean throwing the can of shoe polish at the shoe polish vending
machine: it is likely.
```

which nicely clarifies THROW POLISH AT POLISH, but does not comment on the likelihood of throwing the can at other things or of throwing other things at the vending machine. Moreover, the (suspected) identity of the first item will be known when the rule is consulted; thus

```
Does the player mean tying the noun to the noun: it is very unlikely.
```

will tell Inform to prefer not to tie something to itself if other interpretations are available.

But there is a caveat. There are some cases where this mechanism will not in fact help Inform to choose its way out of an ambiguous command, because of the way it parses one noun at a time. It usually needs to understand the first noun before it will even try to make sense of the second. So a rule like:

```
Does the player mean throwing the can of shoe polish at the tree: it is likely.
```

may not work if the player types THROW POLISH AT TREE and POLISH is ambiguous, because when the parser is trying to understand POLISH, it hasn't yet seen to the end of the command and realised that the second noun will be the tree; so the second noun is unset and the rule won't match.

As a caveat to the caveat, the "inserting it into", "removing it from" and "putting it on" actions have this slightly back to front. These are parsed using the (little-used) "[other things]" or "[things inside]" tokens, and the Inform parser tries to detect the second noun before the first one, since the identity of the first has to depend on the second. So for instance if the situation contains "an oak tree" and also "an oak chest", we could write:

```
Does the player mean inserting into the oak chest:
    it is very likely.
```

which would successfully make PUT COIN IN OAK mean the chest, not the tree. (Note the way we write "inserting into" without saying anything about what's being inserted, not even that it's "something".)

Example

326. **Masochism Deli**
Multiple potatoes, with rules to make the player drop the hot potato first and pick it up last.

RB 10.9 Heat

§17.20 Multiple action processing

When the player types a command like DROP ALL, this is (usually) a request to carry out more than one action. After the command parser has decided what constitutes "ALL" (a process which can be influenced using the "deciding whether all includes" activity), it forms up a list and then runs through it, starting an action for each in turn. The result

usually looks something like this:

```
>GET ALL
foxglove: Taken.
snake's head fritillary: Taken.
```

However, by adding rules to the rulebook:

```
multiple action processing rules
```

we can take a look at the actions intended, and rearrange or indeed change them before they take effect. To do that, we have to deal with a special list of objects. For two technical reasons this isn't stored as a "list of objects that varies" - first because it needs to exist even in low-memory situations where we can't afford full list-processing, and second because there are times when changing it might be hazardous. Instead, two phrases are provided to read the list and to write it back:

multiple object list ⇒ *list of objects*

This phrase produces the current multiple object list as a value. The list will be the collection of objects found to match a plural noun like ALL in the most recent command typed by the player. If there is no multiple object, say if the command was TAKE PEAR, the list will be empty: it won't be a list of size 1.

alter the multiple object list to (list of objects)

This phrase sets the multiple object list to the given value. The list is ordinarily the collection of objects found to match a plural noun like ALL in the most recent command typed by the player, but using this phrase at the right moment (before the "generate action rule" in the turn sequence rules takes effect).

Examples

327. **The Best Till Last**
    Reordering multiple objects for dramatic effect.
    RB 6.15 Actions on Multiple Objects

328. **Western Art History 305**
    Allowing EXAMINE to see multiple objects with a single command.
    RB 6.15 Actions on Multiple Objects

§17.21 Understanding mistakes

When inspiration strikes the player, he can usually be relied upon to make a good-faith effort to communicate the new idea: he will guess the right command. If he guesses wrongly, the mistake is probably the author's, because a good author will try to anticipate all possible wordings and make all of them work.

Nevertheless it is sometimes good practice to nudge the player towards the right wording - particularly if the player has the right idea but is not explicit enough: for instance, typing TALK TO JUDGE when we really want to know what is to be said (JUDGE, GUILTY); or if the player tries something like PLAY CHESS rather than MOVE PAWN TO KING 4. Similarly, if we make a casual reference such as "In your childhood days, you loved sliding in stocking feet across this hallway", a player might type SLIDE IN STOCKING FEET: a nice idea, and which deserves a nice response, even though it asks to do something beyond the scope of the story.

Inform provides a simple mechanism for recognising a command but at the same time recognising that *it does not properly specify an action*. Such commands are called "mistakes", for the sake of a memorable term, but the player has not really behaved badly, and should be helped rather than reproved. For instance:

```
Understand "act" as a mistake.
```

While that works - the command to "act" is indeed rejected - it is not very good, because no very helpful message is brought up. The following is much better:

```
Understand "act" as a mistake ("To join the actors, you have to adopt a role in the play! Try PLAY HAMLET or similar.").
```

Or we could once again insist on a given context:

```
Understand "act" as a mistake ("To join the actors, you have to adopt a role in the play! Try PLAY HAMLET or similar.") when the location is the Garden Theatre.
```

That still has the drawback that the command "act hamlet" will not be recognised: so the final version we want is probably

```
Understand "act [text]" as a mistake ("To join the actors, you have to adopt a role in the play! Try PLAY HAMLET or similar.") when the location is the Garden Theatre.
```

since the "[text]" part will soak up any words the player types (or none), meaning that any command at all whose first word is "act" will be matched.

We need to be careful to avoid circular things like this:

```
Understand "[text]" as a mistake ("'[the topic understood]' is something I really wish you wouldn't say.") when the topic understood is a topic listed in table 1.
```

This doesn't work because the topic understood isn't set until the line has been understood, but Inform checks the "when…" condition before it tries to understand the line. Indeed, even this:

```
Understand "[text]" as a mistake ("'[the topic understood]' is something I really wish you wouldn't say.").
```

is unsafe (quite apart from being unwise!) - again, "topic understood" doesn't exist for a mistake, because in a mistake, nothing is understood.

The following is often useful during beta-testing of a new work, though we would not want it in the final published edition. Many authors like to ask their testers not to try anything in particular, simply to play naturally: but to record the transcript of the session, and email it back to the author. The following command is a device to allow the tester to type a comment in to the transcript:

```
Understand "* [text]" as a mistake ("Noted.").
```

For instance, the tester might type "* DIDN'T WE SAY DARCY WAS TALL?", to which the story would reply "Noted." - and the author can search for such comments when receiving the transcript.

If we are careful, we can make the reply depend on what was typed in the mistaken command:

```
Understand "steal [something]" as a mistake ("Just TAKE [the noun] and leave without paying: that's stealing in my book.").
```

The care comes in because Inform applies much less checking to mistakes than to other actions, and odd errors will result if we try to refer to (say) "the second noun" in a command which did not have a second noun.

It's probably wise to take particular care if using "as a mistake" with any command which might include the mistake among what the player calls ALL: for example, if "take [sydney harbour bridge]" is understood as a mistake, then TAKE ALL will may result in this, even though the player doesn't intend any such thing.

Addendum

"act [text]" won't match "any word (or none)": it fails to match when it's none. You would need Understand statements.

Examples

329. **Query**
Catching all questions that begin with WHO, WHAT, WHERE, and similar question words, and responding with the instruction to use commands, instead.
RB 11.3 Helping and Hinting

330. **The Gorge at George**
If the player tries to TALK TO a character, suggest alternative modes of conversation.
RB 7.6 Getting Started with Conversation

331. **Hot Glass Looks Like Cold Glass**
Responding to references to a property that the player isn't yet allowed to mention (or when not to use "understand as a mistake").
RB 10.9 Heat

§17.22 Precedence

When several different lines of grammar are supplied to meet the same circumstances, it makes a big difference what order they are tried in. For instance, suppose we have:

```
Understand "photograph [a door]" as photographing.

Understand "photograph [an open door]" as photographing.
```

The second line is more specific than the first, so Inform takes these grammar lines the other way around: it checks for "open door" before it checks for "door". That didn't matter here, since both lines came out with the same result (the action of photographing), but it matters very much in the next example:

```
Understand "employ [a door]" as opening.
Understand "employ [an open door]" as entering.
```

More subtle is a line already seen:

```
Understand "on/in/inside" or "on top of" as "[within]".
```

Here Inform puts "on top of" before "on/in/inside", since otherwise only the "on" of "on top of" will be recognised.

Mistakes always take precedence over non-mistakes: this is intended to make sure that

```
Understand "take umbrage" as a mistake ("Nobody takes umbrage in this story, mister.").
```

will take precedence over

```
Understand "take [something]" as taking.
```

even if there is, in fact, a character called Mr Nimbus Umbrage so that the command could conceivably make sense.

Finally, there are a few grammars where the number of values produced is different in different lines. For example, the Standard Rules include these among the possible "put" commands:

```
Understand "put [something preferably held] on" as wearing.
Understand "put [other things] on/onto [something]" as putting it on.
```

One produces a single object, the other produces two. Inform gives precedence to the first of these, that is, it tries the one with fewer values first. This is important when reading commands like "PUT MARCH ON WASHINGTON SHIRT ON", and also prevents bogus auto-completions, in which PUT HAT ON might wrongly be auto-completed as if it were PUT HAT ON THE TABLE.

Examples

332. Some Assembly Required
    Building different styles of shirt from component sleeves and collars.

    RB 9.3 Clothing

333. Lakeside Living
    Similar to "Lemonade", but with bodies of liquid that can never be depleted, and some adjustments to the "fill" command so that it will automatically attempt to fill from a large liquid source if possible.

    RB 10.2 Liquids

18. Activities

§18.1 What are activities?
§18.2 How activities work
§18.3 Rules applied to activities
§18.4 While clauses
§18.5 New activities
§18.6 Activity variables
§18.7 Beginning and ending activities manually
§18.8 Introduction to the list of built-in activities
§18.9 Deciding the concealed possessions of something
§18.10 Printing the name of something
§18.11 Printing the plural name of something
§18.12 Printing a number of something
§18.13 Listing contents of something
§18.14 Grouping together something
§18.15 Issuing the response text of something
§18.16 Printing room description details of something
§18.17 Printing inventory details of something
§18.18 Printing a refusal to act in the dark
§18.19 Printing the announcement of darkness
§18.20 Printing the announcement of light
§18.21 Printing the name of a dark room
§18.22 Printing the description of a dark room
§18.23 Constructing the status line
§18.24 Writing a paragraph about
§18.25 Listing nondescript items of something
§18.26 Printing the locale description of something
§18.27 Choosing notable locale objects for something
§18.28 Printing a locale paragraph about
§18.29 Deciding the scope of something
§18.30 Clarifying the parser's choice of something
§18.31 Asking which do you mean
§18.32 Supplying a missing noun/second noun
§18.33 Reading a command
§18.34 Implicitly taking something
§18.35 Printing a parser error
§18.36 Deciding whether all includes
§18.37 Printing the banner text
§18.38 Printing the player's obituary

§18.39 Amusing a victorious player
§18.40 Starting the virtual machine

§18.1 What are activities?

It is poor form to define with negatives, but the first thing to say about activities is that they are *not* actions. This needs saying because Inform often seems to treat them as if they are, by allowing us to write rules like so:

```
Before printing the name of a woman, say "Ms ".
```

With this rule in place, someone called "Daphne" will always be described as "Ms Daphne", and so on. The language looks as if we were imposing a rule on an action called "printing the name of", but there is no such action: instead, it is an "activity". To spell out the difference:

```
An action is a simulated task for the fictional protagonist.
An activity is a real task for the computer program doing the simulation.
```

Activities allow us to influence or change some of the standard habits of Inform, using rules as flexible and powerful as those applicable to actions, though activities are in several ways simpler and easier.

Example

334. **Ant-Sensitive Sunglasses**
What are activities good for? Controlling output when we want the same action to be able to produce very flexible text depending on the state of the world -- in this case, making highly variable room description and object description text.

RB 3.1 Room Descriptions

§18.2 How activities work

All activities start, continue for a while and then finish: however, no activity ever runs on for more than a single turn. Several activities can be going on at the same time. For instance, suppose the following is printed as part of the description of a grocery:

```
You can see a banana, an apple and a star-fruit here.
```

At the moment when Inform prints "apple", two activities are under way: "listing contents of the Grocery", and "printing the name of the apple". The sequence of events was in fact:

```
say "You can see "
start listing contents of the Grocery
    say "a "
    start printing the name of the banana
        say "banana"
    finish printing the name of the banana
    say ", an "
    start printing the name of the apple
        say "apple"
    finish printing the name of the apple
    say " and a "
    start printing the name of the star-fruit
        say "star-fruit"
    finish printing the name of the star-fruit
finish listing contents of the Grocery
say " here."
```

The golden rule is: if activity B starts during activity A, it must also finish during activity A.

If we ever need to find out, we can always test:

```
if the printing the name activity is going on, ...
if the printing the name activity is not going on, ...
```

but as we shall see, it's usually simpler to attach "while printing the name" provisos to rules.

§18.3 Rules applied to activities

The activity "printing the name of something" is the process of printing up the name of something on screen: ordinarily, this means saying the text in its "printed name" property.

As with actions, rules can be attached to activities which change or augment what would normally happen. In fact the situation is simpler, because (unlike an action) an activity almost always finishes, so we almost always do reach its "after" stage. There are also only three rulebooks attached to an activity, as compared with the six affecting an action.

The three rulebooks for printing the name are called "before printing the name", "for printing the name" and "after printing the name", and this is the general pattern. What happens is:

```
1. All "before printing the name of" rules are considered;
2. The most specific, applicable "rule for printing the name of" is considered;
3. All "after printing the name of" rules are considered.
```

Whereas an action's later stages never take place if an early stage ends unexpectedly, an activity always goes through all three of its stages. Invoking the word "instead" in a before rule for an action will terminate not only the before rules but the whole action: the same thing for an activity will only terminate the before rules, and the for and after rules will take place as usual.

The actual task is usually carried out by one single rule tucked into the back of the "for…" rulebook: it is the rule for printing the name of whatever is concerned, hence the name. Inform's standard activities are all of this pattern: they start out with no "before" or "after" rules, and just one "for" rule.

Why the part about an activity only "almost always" finishing? One reason is that the story might end during it; but another is that it's possible, though uncommon, to abandon an activity partway. Very few of the activities supplied with Inform ever do this, and those that do are noted in the sections which follow.

§18.4 While clauses

Rules applied to actions can become baroque ("after going through a door in the presence of an animal when -" and so on and so forth), but activities are again simpler: they only have one possible clause attached, which is called "while". For instance, the following would provide a fairly sledgehammer hint that the sack should not lightly be thrown away:

```
The sack is a player's holdall. The sack is carried. Rule for printing the name of the sack
while the sack is not carried: say "your abandoned sack".
```

Any condition can be given after the "while", and we can also specify that another activity has to be going on. Thus:

```
Rule for printing the name of the lemon sherbet while listing contents: say "curious sort
of lemon sherbet sweet".
```

This nicely distinguishes between contexts where it's appropriate to be more verbose, and where it isn't. Thus:

```
You can see a teaspoon and a curious sort of lemon sherbet sweet here.
> TAKE ALL
teaspoon: Taken.
lemon sherbet: Taken.
```

§18.5 New activities

Activities are all about influencing the standard mechanisms which Inform uses, so it might at first seem that there is no need to create new activities: but on further reflection, quite a lot of the writing of interactive fiction involves creating new and systematic ways to do things, and as soon as we have a general rule, we will want to have exceptions. Inform therefore allows us to create our own activities, giving us ways to influence the operation of our own mechanisms.

There are two kinds of activity: those which relate to a specific value (usually an object but not necessarily), and those which do not. Here are some examples of activities being created:

```
Assaying is an activity.
Analysing something is an activity.
Announcing something is an activity on numbers.
```

Inform looks for the clue "something" (or "of something") after the activity's name to see if it will work on a value: so analysing and announcing will do, but assaying won't. If we don't specify a kind, Inform assumes the value will be an object, as if we had written:

```
Analysing something is an activity on objects.
```

As always in Inform, the names of activities are themselves values.

```
"assaying activity" has kind activity on nothing
"analysing activity" has kind activity on objects
"announcing activity" has kind activity on numbers
```

Creating an activity is like creating an action: it automatically makes new rulebooks - "before analysing", "for analysing" and "after analysing" - but they start out empty, so the activity does nothing yet. Just as it does for rulebooks, Inform defines the adjectives "empty" and "non-empty" for activities to test this state:

```
if the analysing activity is empty, ...
```

will be true only when all three of its rulebooks are empty.

A newly created activity never happens unless we take steps to make it do so. We can make an activity happen at any time by writing phrases like so:

carry out the (activity) **activity**

This phrase carries out the given activity, which must be one not applying to any value. Example:

```
carry out the assaying activity;
```

carry out the (activity on values) **activity with** (value)

This phrase carries out the given activity, which must apply to a kind of value matching the one supplied. Example:

```
carry out the analysing activity with the pitchblende;
carry out the announcing activity with the score;
```

To make the activity do something useful, we need to put a rule into its "for" rulebook:

```
Rule for announcing a number (called N): say "Ladies and gentlemen, [N]."

The last for assaying rule:
    say "Professionally, you cast an eye around mineral deposits nearby, noticing [list of rocks in the location]."
```

"The last" is a technicality about rulebooks (see the next chapter) which, put briefly, guarantees that this rule comes last among all possible "for assaying" rules. This is good form because the whole point of an activity is to make it easy for further rules to interfere - so we deliberately hang back to last place, giving precedence to anybody else who wants it.

The "for" rulebook is one where rules stop the activity, by default, when they take effect - in the same way that the "instead" rules stop actions by default. If this causes problems, we can use:

continue the activity

This phrase should be used only in rules in activity rulebooks. It causes the current rule to end, but without result, so that the activity continues rather than stopping as a result of the rule. This is useful for rulebooks (like the "for" rulebook of an activity) where the default is that a rule does stop the activity.

Activities are more useful than they first appear. Every new one provides a context which other activities can observe. We could, for instance, define

```
Rule for printing the name of a rock while assaying: ...
```

so that during assays more technical names are used.

Examples

335. **AARP-Gnosis**
An Encyclopedia set which treats volumes in the same place as a single object, but can also be split up.
RB 9.6 Reading Matter

336. **Aftershock**
Modifying the rules for examining a device so that all devices have some specific behavior when switched on, which is described at various times.
RB 9.9 Televisions and Radios

337. **Crusoe**
Adding a "printing the description of something" activity.
RB 6.5 Examining

§18.6 Activity variables

Just as actions can have variables, which are created when the action starts and disappear when it finishes, so activities can also have variables. They are visible to the rules for that activity, and nowhere else. (If the activity should happen a second time within its first run, that second occurrence gets its own copy of the variable, leaving the original untouched.)

Typically it will be useful to set a variable to some default value at the "before" stage, calculate some interesting value for it in the "for" stage, and make use of the outcome during the "after" stage. For instance:

```
Analysing something is an activity. The analysing activity has a text called first impression. Instead of examining something (called the sample), carry out the analysing activity with the sample.

Before analysing: now the first impression is "unremarkable".

Rule for analysing someone: now the first impression is "living tissue".

After analysing something (called the sample):
    say "Your professional opinion of [the sample] is that it is
        [first impression]."
```

§18.7 Beginning and ending activities manually

If we have declared a new activity, like "analysing", the normal way to make it happen would be to write

```
carry out the analysing activity with the pitchblende;
```

which goes through the whole machinery of rules - before, for, after - and then resumes, the activity having started, taken place and come to an end.

But there are times when it is not convenient to write a suitable "for ..." rule, or where we need more control, and do not wish to hand the whole business over to a single phrase. For such times we are allowed to write:

begin the (activity) **activity**

This phrase causes the named activity to become active, and runs its "before" rulebook. The activity must be one which applies to nothing. Example:

```
begin the assaying activity;
```

In all cases a matching "end the ... activity" or else "abandon the ... activity" phrase must be reached.

begin the (activity on values) **activity with** (value)

This phrase causes the named activity to become active, and runs its "before" rulebook. The activity must be one which applies to a value of a matching kind. Example:

```
begin the analysing activity with the pitchblende;
```

In all cases a matching "end the … activity with …" or else "abandon the … activity with…" phrase must be reached.

And when we are done:

end the (activity) **activity**

This phrase runs the "after" rulebook of the activity and then causes it to become inactive. The activity must be one which applies to nothing. Example:

```
end the assaying activity;
```

This must only happen to match an earlier "begin the … activity" phrase.

end the (activity on values) **activity with** (value)

This phrase runs the "after" rulebook of the activity and then causes it to become inactive. The activity must be one which applies to a value of a matching kind. Example:

```
end the analysing activity with the pitchblende;
```

This must only happen to match an earlier "begin the … activity with…" phrase.

So the usual structure is like so:

```
begin the analysing activity with the pitchblende;
...
end the analysing activity with the pitchblende;
```

This time the activity is ongoing throughout as many phrases as we care to write between

the "begin" and "end". The before rules are considered at the time of the "begin ..." phrase; the after rules at the "end ...".

What, then, of the "for" rules? In the above setup, they would simply be ignored. But we can make them effectual thus

```
begin the analysing activity with the pitchblende;
...
if handling the analysing activity with the pitchblende:
    ...
...
end the analysing activity with the pitchblende;
```

We place the activity's normal behaviour inside the "if"; the condition, "if handling...", is true only if no rule has intervened. This means that we (or other authors using our activity) can create their own for rules to substitute here. If we elsewhere write

```
Rule for handling the analysing activity with the pitchblende when the player is not sober:
    say "You can't seem to focus."
```

that rule will intervene and take the place of whatever we have placed inside the condition.

if handling (activity) **activity:**

This should be used only where the given activity has been started with "begin ..." and will be finished with "end ...". It runs the "for" rules for the activity, and then comes out true if none of those for rules intervened in the handling of that activity. (The activity must be one which doesn't apply to any value.)

if handling (activity on values) **activity with** (value)**:**

This should be used only where the given activity has been started with "begin ..." and will be finished with "end ...". It runs the "for" rules for the activity, and then comes out true if none of those for rules intervened in the handling of that activity. (The given value must be the one it is being applied to.)

It is also legal to force an early end to an activity with:

abandon the (activity) **activity**

This phrase ends an activity at once (without consulting any further rulebooks, including its "after" rulebook). It can only be used with an activity which has had its "begin" but not yet its "end" phrase; it is a drastic remedy best taken only if it is clear that circumstances have changed so that the activity now seems inappropriate. It must not be used during one of the rules for the activity: it can only be used between the begin and for stages, or between the for and end stages.

```
abandon the assaying activity;
```

abandon the (activity on values) **activity with** (value)

This phrase ends an activity at once (without consulting any further rulebooks, including its "after" rulebook). It can only be used with an activity which has had its "begin" but not yet its "end" phrase; it is a drastic remedy best taken only if it is clear that circumstances have changed so that the activity now seems inappropriate. It must not be used during one of the rules for the activity: it can only be used between the begin and for stages, or between the for and end stages.

```
abandon the analysing activity with the pitchblende;
```

We need to follow three golden rules: all activities must end, they must never last longer than a turn, and if activity B starts during activity A then it must also finish during activity A. We must also be careful to make sure that if an activity applies to something, then it begins and ends with the same something (the pitchblende, in the above example).

§18.8 Introduction to the list of built-in activities

Activities tend to be about process, rather than outcome. Many of the things Inform does - printing up lists of items, reading commands from the keyboard, and so on - are done as activities, because that way the process can be nudged a little. Too many works of interactive fiction betray their mechanical nature by making it visible that the general machinery being used does not quite seem natural for this or that situation. Activities enable us to add the many graceful touches which avoid that: which contribute nothing to a work, and also everything.

The rest of this chapter covers every activity built in to Inform, with one section for each.

It is intended primarily for reference, but may be worth skimming through at a first reading, to give a sense of the possibilities.

§18.9 Deciding the concealed possessions of something

1. When it happens. Frequently - whenever Inform needs to check whether something is visible or not. Nothing should be printed, and the activity needs to run quickly, so it should not (for instance) calculate best routes through complicated maps before getting an answer.

2. The default behaviour. There is no concealment. The ordinary rules still apply, though: the contents of a closed opaque container are invisible because there is a barrier in the way which cannot be seen through, even though nobody is "concealing" anything.

3. Examples. To repeat a number of brief examples given at the end of Chapter 3, where this activity made an early appearance:

```
Rule for deciding the concealed possessions of the Cloaked Villain: if the particular
possession is the sable cloak, no; otherwise yes.

The coin is in the Roman Villa. The face and inscription are parts of the coin. Rule for
deciding the concealed possessions of the coin: if the coin is carried, no; otherwise yes.
```

The value "particular possession" is the one whose concealment is in question, of course. We can ignore this if someone is invariably secretive:

```
Rule for deciding the concealed possessions of the furtive ghost: yes.
```

In general a rule for deciding the concealed possessions of something will decide "yes" if finishes without making a decision, but it's better style to write such a rule in such a way that it always makes a decision.

Example

338. **Hays Code**
Clark Gable in a pin-striped suit and a pink thong.

RB 9.3 Clothing

§18.10 Printing the name of something

1. When it happens. Whenever the name of a thing or room is printed, either as part of text visible to the player, or sometimes internally in order to determine something about that name.

2. The default behaviour. For items other than the current player, the "printed name" property is printed out; but for the current player, "you" or "yourself" is printed. (That doesn't necessarily mean that the "printed name" of the player is never used. Suppose there are two people, Alice and Bob, and the narrative switches between them: when Alice is the player, she appears as "yourself" but Bob is "Bob"; but when Bob is the player, he is "yourself" and Alice is "Alice".)

3. Examples. (a) A pen which is described differently in inventories:

```
Rule for printing the name of the pen while taking inventory: say "useful pen".
```

"Taking inventory" is a condition which is true if that's the current action and not otherwise, so the effect is that the pen is called "a useful pen" only in inventory listings. "While looking" is a similarly useful one.

(b) Italicising the names of novels:

```
A novel is a kind of thing. Dr Zhivago and Persuasion are novels. Before printing the name of a novel, say "[italic type]". After printing the name of a novel, say "[roman type]".
```

(c) Telling the time:

```
After printing the name of the wrist watch while taking inventory: say " (time: [the time of day])".
```

(d) Merging containers with their contents:

```
Rule for printing the name of the bottle while not inserting or removing:
    if the bottle contains sand, say "bottle of sand";
    otherwise say "empty bottle";
    omit contents in listing.
```

This example makes use of a special phrase:

omit contents in listing

This phrase changes the form of an inventory listing, room description, etc., so that it will simply list "a bottle of sand" or "an empty bottle", rather than "a bottle (in which is sand)" or "a bottle (which is empty)". It should be used only when the listing is imminent, and does not have permanent effect.

The clause about not inserting or removing is to prevent messages like "You put the sand in the bottle of sand.", where it's confusing to refer to the bottle as anything other than "the bottle".

Examples

339. **Shipping Trunk**
A box of baking soda whose name changes to "completely ineffective baking soda" when it is in a container with something that smells funny.
RB 9.2 Bags, Bottles, Boxes and Safes

340. **Trachypachidae Maturin 1803**
Bottles with removable stoppers: when the stopper is in the bottle, the bottle is functionally closed, but the stopper can also be removed and used elsewhere. Descriptions of the bottle reflect its state intelligently.
RB 9.2 Bags, Bottles, Boxes and Safes

341. **Chronic Hinting Syndrome**
Using name-printing rules to keep track of whether the player knows about objects, and also to highlight things he might want to follow up.
RB 7.11 Character Knowledge and Reasoning

§18.11 Printing the plural name of something

1. When it happens. Only when a group of identical items is present in the same place, and are being described jointly with text like "You can see five gold rings here." The activity happens after "five" and before "here." (See the activity "printing a number of something" if the whole phrase needs to be altered.)

2. The default behaviour. The plural name - in this case "gold rings" - is printed out.

3. Examples. (a) Suppose we want to emphasise how nice it is to have more than one gold ring:

```
Rule for printing the plural name of a gold ring: say "gleaming gold rings".
```

(b) If the number needs changing as well, it's necessary to use the "printing a number of something" activity instead.

Example

342. **Hudsucker Industries**
Letters which are described differently as a group, depending on whether the player has read none, some, or all of them, and on whether they are alike or unlike.
RB 9.2 Bags, Bottles, Boxes and Safes

§18.12 Printing a number of something

1. When it happens. Only when a group of identical items is present in the same place, and are being described jointly with text like "You can see five gold rings here." The activity prints the "five gold rings" part. The variable "listing group size" contains the number, which in this example would be 5, and is always at least 2.

2. The default behaviour. The number of items is printed, in words ("five") and then the "printing the plural name" activity is run ("gold rings").

3. Examples. (a) Using this activity is for perfectionists, because the normal behaviour is almost always fine. Still:

```
Rule for printing a number of blocks when the listing group size is 3: say "all three blocks".
```

(b) Or perhaps:

```
Rule for printing a number of ants: say "altogether [listing group size in words] ants".
```

(c) If the only part needing variation is the plural name, it's simpler and tidier to use the "printing the plural name of something" activity instead.

Example

343. **Prolegomena**
Replacing precise numbers with "some" or other quantifiers when too many objects are clustered together for the player to count at a glance.
RB 2.1 Varying What Is Written

§18.13 Listing contents of something

1. When it happens. When taking inventory, the list is produced by the activity "listing contents of yourself"; when looking, a list of items which do not deserve their own paragraphs is produced by "listing contents of" the location.

And when it doesn't happen. (a) If the Storage Room contains a sideboard and an open

shoe box, then "listing contents of the Storage Room" is used to produce the part of the room description mentioning sideboard and box. But if the box in turn contains a pair of brogues, then "listing contents of the shoe box" is not used to say that part. So this works:

```
Rule for printing the name of the brogues while listing contents of a room: ...
```

But this won't affect room descriptions:

```
Rule for printing the name of the brogues while listing contents of the shoe box: ...
```

(b) The activity also doesn't happen when, for instance, "[a list of animals]" is printed, because that isn't a list of the contents of any room or location.

2. The default behaviour. The list is printed out.

3. Examples. (a) We have already seen that it can be elegant to elaborate on a description in the context of a list. Here we add "discarded" to a sweet wrapper which is found on the ground.

```
Rule for printing the name of the wrapper while listing contents of a room: say "discarded sweet wrapper".
```

(b) Lists can be considerably shortened and tidied up if similar items are grouped together. We do this by specifying what should be grouped together before listing contents, using the special phrase "group … together":

```
Utensil is a kind of thing. The knife, the fork and the spoon are utensils. Before listing contents: group utensils together as "utensils".
```

The result will be, say, "two utensils (knife and spoon)", if both are found in the same place.

(c) We can less obtrusively group items together like so:

```
Before listing contents while taking inventory: group utensils together.
```

Three special phrases exist for this kind of list organisation:

group (description of objects) **together**

This phrase causes the objects described to be listed together in a single item as part of an inventory or room description. The effect is temporary, and the phrase should only be used when this list is imminent. Example:

```
Utensil is a kind of thing. The knife, the fork and the spoon are utensils. Before listing contents: group utensils together.
```

This might produce the list item "fork and spoon".

group (description of objects) **together giving articles**

This phrase causes the objects described to be listed together in a single item as part of an inventory or room description, but giving each individual item its indefinite article. The effect is temporary, and the phrase should only be used when this list is imminent. Example:

```
Utensil is a kind of thing. The knife, the fork and the spoon are utensils. Before listing contents: group utensils together giving articles.
```

This might produce the list item "a fork and a spoon".

group (description of objects) **together as** (text)

This phrase causes the objects described to be listed together in a single item as part of an inventory or room description, summarised with the given text. The effect is temporary, and the phrase should only be used when this list is imminent. Example:

```
Utensil is a kind of thing. The knife, the fork and the spoon are utensils. Before listing contents: group utensils together as "utensils".
```

This might produce the list item "two utensils (fork and spoon)".

Example

344. **Unpeeled**
Calling an onion "a single yellow onion" when (and only when) it is being listed as the sole content of a room or container.

RB 9.2 Bags, Bottles, Boxes and Safes

§18.14 Grouping together something

1. When it happens. Only while listing contents, and only when a collection of items to be grouped together is reached. This in turn happens only if a "before listing contents" rule has chosen it (see previous section). The first item in the group is the one to which the activity formally applies.

The variable "listing group size" usually gives the number of items grouped together in this way, but care is needed if the "something" covers multiple groups, because then this will all be working on a list of groups, not of items. For example, in a "grouping together things" rule, where "things" is such a broad description that it can apply to multiple kinds of thing all appearing in the list, the list-maker is likely to set "listing group size" to the number of groups. If the list contains five men and six women, for example, "listing group size" might be 2 rather than 11.

2. The default behaviour. The items grouped together are printed in an English phrase, such as "egg, chicken and farmer". In particular, they are not split onto separate lines even if the rest of the list is. (See previous section.)

3. Examples. (a) Here are Scrabble pieces which are described as "the tile W from a Scrabble set" or similar outside of lists, but which, when they turn up together in lists, are rolled together into "the tiles A, B and D from a Scrabble set".

```
A Scrabble piece is a kind of thing. The X, the Y and the Z are Scrabble pieces.

Before listing contents: group Scrabble pieces together.

Before printing the name of a Scrabble piece while not grouping together, say "tile ". After printing the name of a Scrabble piece while not grouping together, say " from a Scrabble set".

Before grouping together Scrabble pieces, say "the tiles ". After grouping together Scrabble pieces, say " from a Scrabble set".
```

(b) Maybe we only want an abbreviated form when there are five or more tiles in one place:

```
A Scrabble piece is a kind of thing. The X, the W, the F, the Y and the Z are Scrabble pieces in the Lounge.

Before listing contents: group Scrabble pieces together.

Before grouping together Scrabble pieces when the listing group size is greater than 4:
    say "some [listing group size in words] tiles (".
After grouping together Scrabble pieces when the listing group size is greater than 4:
    say ") from a Scrabble set".
```

(c) We can throw out all pretence at listing and say whatever we like, in fact:

```
Before listing contents while taking inventory: group utensils together. Rule for grouping together utensils: say "the usual utensils".
```

§18.15 Issuing the response text of something

1. When it happens. When Inform prints a text marked with a response letter (A), (B), (C), …, in a rule making use of them. For example, in this rule:

```
Carry out taking inventory (this is the print empty inventory rule):
    if the first thing held by the player is nothing,
        say "[We] [are] carrying nothing." (A) instead.
```

Or, less directly,

```
let R be the print empty inventory rule response (A);
say "To be frank: [text of R].";
```

2. The default behaviour. To print the current textual value of the response, making any substitutions in the ordinary way.

3. Examples. This activity is not the best way to amend responses or make them dynamic; the whole idea of responses is that they can be changed just as if they were text variables. This activity should be used only if it's important to amend blocks of responses in some systematic way.

(a) With that said, some interesting effects can be achieved. This is a way to see which responses are being printed, for example:

```
Before issuing the response text of a response (called R): say "[R]: ".
```

whence:

```
>WAIT
standard report waiting rule response (A): Time passes.
```

(b) And this intercepts the activity in order to re-run it in each of the six viewpoints. (Note the way a variable is used to prevent the rule from applying to all of those re-runs as well.)

```
The response inhibition is initially false.

Rule for issuing the response text of a response (called R) when the response inhibition is false:
    now the response inhibition is true;
    let the current viewpoint be the story viewpoint;
    repeat with P running through narrative viewpoints:
        now the story viewpoint is P;
        say "[P]: [text of R][command clarification break]";
    now the story viewpoint is the current viewpoint;
    now the response inhibition is false.
```

With that in place,

```
>EAST
first person singular: I can't go that way.
second person singular: You can't go that way.
third person singular: He can't go that way.
first person plural: We can't go that way.
second person plural: You can't go that way.
third person plural: They can't go that way.
```

Example

345. **Wesponses**
Parser messages that are delivered with a speech impediment.
RB 2.1 Varying What Is Written

§18.16 Printing room description details of something

1. When it happens. When an item is listed in the miscellaneous collection of items present in a room (the ones which do not deserve their own paragraphs): this is normally the last paragraph of a room description.

2. The default behaviour. A bracketed piece of extra information is added for certain items such as containers:

```
You can also see Po and a cage (empty) here.
```

The " (empty)" (note initial space) was added by this activity. (Note that this activity is not responsible for describing further items visible because of the item in question: that is, it does not print the text such as "(in which is a notepad)" which would appear if there were contents. If we want to remove such text, we should use "omit contents in listing": see the activity "for printing the name of something".)

3. Examples. (a) To get rid of such addenda entirely, try:

```
Rule for printing room description details: stop.
```

(b) To add a new form of addendum:

```
Rule for printing room description details of a person:
    say " (at last, someone to talk to)" instead.
```

If both examples (a) and (b) are in place at once, we might now read:

```
You can also see Po (at last, someone to talk to) and a cage here.
```

Example

346. **Rules of Attraction**
A magnet which picks up nearby metal objects, and describes itself appropriately in room descriptions and inventory listings, but otherwise goes by its ordinary name.
RB 10.7 Electricity and Magnetism

§18.17 Printing inventory details of something

1. When it happens. When an item is listed in an inventory of items carried by the player.

2. The default behaviour. A bracketed piece of extra information is added for certain items such as containers:

```
a flaming branch (providing light)
```

The " (providing light)" (note initial space) was added by this activity.

3. Examples. (a) To get rid of such addenda entirely, try:

```
Rule for printing inventory details: stop.
```

(b) To add a new form of addendum:

```
Rule for printing inventory details of something edible:
    say " (yummy!)[run paragraph on]".
```

§18.18 Printing a refusal to act in the dark

1. When it happens. When an action which requires light is tried, and the visibility rules decide that not enough light is present.

2. The default behaviour. To print "It is pitch dark, and you can't see a thing."

3. Examples. (a) This might do for some twilit, penumbral room:

```
Rule for printing a refusal to act in the dark: if we are examining something, say "It's not totally dark here, perhaps, but certainly too dim for close-up examination of anything." instead.
```

Example

347. **Zorn of Zorna**
Light levels vary depending on the number of candles the player has lit, and this determines whether or not he is able to examine detailed objects successfully.
RB 3.7 Lighting

§18.19 Printing the announcement of darkness

1. When it happens. Inform frequently calculates to see if the player is in light or darkness: this activity happens on the change from light to darkness.

2. The default behaviour. To print "It is now pitch dark in here!".

3. Examples. (a) The most obvious use is to change the text:

```
Rule for printing the announcement of darkness: say "Ooh-er! It's now very nearly pitch dark in here." instead.
```

(b) But we could also use this activity for sneakier purposes, silently moving things around:

```
Before printing the announcement of darkness: now all of the gremlins are in the kitchen.
```

(c) A special description for occasions when the player has climbed into a container and

shut it (so that the darkness is the result of his own actions, rather than some external circumstance):

```
Rule for printing the announcement of darkness when closing a container which contains the player:
    say "Congratulations: now you can't see a thing." instead.
```

§18.20 Printing the announcement of light

1. When it happens. Inform frequently calculates to see if the player is in light or darkness: this activity happens on the change from darkness to light.

2. The default behaviour. To try the looking action, which usually prints up a room description.

3. Examples. (a) Perhaps the player is initially too disoriented to look around in any coherent way:

```
Rule for printing the announcement of light in the Dazzling Temple: say "You are almost blinded by the suffusion of white light, and have spots before your eyes." instead.
```

§18.21 Printing the name of a dark room

1. When it happens. When looking in darkness, or writing the (default) status line in darkness.

2. The default behaviour. To print "Darkness".

3. Examples. (a) One might modify the darkness with some adjective:

```
Before printing the name of a dark room, say "Near ".
```

(Note that this activity does not come in different forms for different dark rooms: the wording is fixed at "printing the name of a dark room", and we are not allowed to substitute particular dark rooms or assign a "(called …)" onto the mention of the dark room.)

§18.22 Printing the description of a dark room

1. When it happens. When looking in darkness.

2. The default behaviour. To print "It is pitch dark, and you can't see a thing."

3. Examples. (a) A simple variation of wording:

```
Rule for printing the description of a dark room: say "Your eyes can barely make anything out." instead.
```

(b) More stylishly,

```
Rule for printing the description of a dark room: try listening instead.
```

which produces, for instance,

```
Darkness
You hear nothing unexpected.
```

(Note that this activity does not come in different forms for different dark rooms: the wording is fixed at "printing the description of a dark room", and we are not allowed to substitute particular dark rooms or assign a "(called …)" onto the mention of the dark room.)

Examples

348. **Hohmann Transfer**
Changing the way dark rooms are described to avoid the standard Inform phrasing.
RB 3.7 Lighting

349. **Four Stars 1**
An elaboration of the idea that when light is absent, the player should be given a description of what he can smell and hear, instead.
RB 3.7 Lighting

§18.23 Constructing the status line

1. When it happens. Just before input is accepted from the keyboard, Inform constructs a "status line" at the top of the window which is normally displayed in reverse colours (white on black instead of black on white, say).

2. The default behaviour. Makes the status line up out of two pieces, the "left hand status line" and the "right hand status line". Since these can freely be changed, note that the status line is already very customisable without using rules applied to this activity.

3. Examples. (a) The most useful thing about this activity is that it allows us to vary descriptions in the status line. This is especially helpful to abbreviate unduly long room names, which might not otherwise fit:

```
The Temple Of A Thousand Mightily Peeved Deities is a room. Rule for printing the name
of the Temple while constructing the status line: say "Temple".
```

(b) Again, it's usually not necessary to apply activity rules to this, but occasionally amusing effects are possible if we do:

```
The blindfold is wearable and carried. Rule for constructing the status line while the
blindfold is worn: do nothing.
```

Examples

350. **Ways Out**
A status line that lists the available exits from the current location.
RB 12.2 The Status Line

351. **Guided Tour**
A status line that lists the available exits from the current location, changing the names of these exits depending on whether the room has been visited or not.
RB 12.2 The Status Line

§18.24 Writing a paragraph about

1. When it happens. Just *before* writing a paragraph about some item in a room description.

2. The default behaviour. Is to do nothing. However, if a rule is supplied which prints something up, then this replaces the paragraph which would otherwise have been printed. Moreover, any items whose names are said in the course of this rule - for instance, by being listed - are then excluded from the remainder of the room description, because they are considered as having been described sufficiently already.

Warning: because we often want a "for" rule for this activity to make some calculation and then possibly choose to do nothing (see the example "Otranto"), Inform suppresses the usual paragraph not when a "for" rule took effect but when it detected a paragraph having been printed. This can get confused if a text substitution affecting paragraph breaks, say "[line break]", is within the final "say" of a "for writing a paragraph about" rule.

3. Examples. (a) This is a neat way to wrap several things together into the same paragraph:

```
Rule for writing a paragraph about Mr Wickham:
    say "Mr Wickham looks speculatively at [list of women in the location]."
```

because now "Mr Wickham looks speculatively at Velma and Daphne" will now prevent the appearance of the subsequent text "You can also see Velma and Daphne."

Inform keeps track of which objects have already been named with an either/or property called "mentioned", which it assigns whenever the name of an object has been automatically printed. So in this case, Velma and Daphne are now mentioned. Note "automatically printed", though: if the text printed had just been "Mr Wickham looks speculatively at Velma and Daphne", rather than the text-substitution list used above, then Inform would not know that Velma and Daphne have been described.

If we ever need to override this - say, we want to list all the women but make sure that Velma gets another paragraph anyway - we could change Velma to unmentioned again after the listing.

Examples

352. **Reflections**
    Emphasizing the reflective quality of shiny objects whenever they are described in the presence of the torch.
    RB 3.7 Lighting

353. **Emma**
    Social dynamics in which groups of people form and circulate during a party.
    RB 7.16 Social Groups

354. **Air Conditioning is Standard**
    Uses "writing a paragraph about" to make person and object descriptions that vary considerably depending on what else is going on in the room, including some randomized NPC interactions with objects or with each other.
    RB 4.3 Event Scheduling

§18.25 Listing nondescript items of something

1. When it happens. This activity prints up the also-ran paragraph at the end of a room description. These are nondescript items because they don't merit paragraphs of their own: if, as sometimes happens, there are none in the room, then no such paragraph is printed and this activity does not happen. (So to add a further paragraph to a room description, a simpler "after looking" rule should be used, not an "after listing nondescript items" rule.)

2. The default behaviour. The paragraph ordinarily reads as "You can also see a cask and a clock." or similar. Before the activity begins, those objects which are nondescript - in this case the cask and the clock - are given the property of being "marked for listing".

If it turns out that nothing is marked for listing, because of before rules like the one in the example below, then nothing is printed and **the activity is abandoned**, so that the rules for and after are never reached.

3. Examples. (a) Promoting something out of the nondescript category, by unmarking it.

```
Before listing nondescript items:
    if the watch is marked for listing:
        say "The watch catches your eye.";
        now the watch is not marked for listing.
```

(b) Changing the normal phrasing of the paragraph. Note that we can also change the listing style; the one below is the default.

```
Rule for listing nondescript items of the Distressingly Messy Room:
    say "Strewn carelessly on the floor";
    list the contents of the Distressingly Messy Room, as a sentence,
        tersely, listing marked items only, prefacing with is/are,
        including contents and giving brief inventory information;
    say "."
```

Examples

355. **Rip Van Winkle**
    A simple way to allow objects in certain places to be described in the room description body text rather than in paragraphs following the room description.
    RB 3.1 Room Descriptions

356. **Happy Hour**
    Listing visible characters as a group, then giving some followup details in the same paragraph about specific ones.
    RB 7.16 Social Groups

357. **The Eye of the Idol**
    A systematic way to allow objects in certain places to be described in the room description body text rather than in paragraphs following the room description, and to control whether supporters list their contents or not.
    RB 3.1 Room Descriptions

§18.26 Printing the locale description of something

1. When it happens. A "locale description" is Inform jargon for the part of a room description which catalogues the visible items in the room. When looking, Inform will normally print the description of the room itself, followed by a locale description for the room. But if the player is in a cage in the room, there will be two locale descriptions: one for the room, then another for the cage. This activity is used to write the locale description for a single domain, and the "something" can be either a room, an enterable container, or an enterable supporter.

2. The default behaviour. Is quite complicated, and is written up in full in the typeset form of the Standard Rules downloadable from the Inform website. Briefly, though: we first run the "choosing notable locale objects" activity to find out what ought to be mentioned here. That assembles a list of things to mention, sorted into priority order. Items with priority 1 go first, then those with priority 2, and so on. The "printing a locale paragraph" activity is run for each, and in practice that usually hands the job over to "writing a paragraph about". Sometimes a paragraph will indeed be written, but not always. Sometimes there is nothing interesting to say, and an item is left until a final, single paragraph which gathers up the leftovers ("You can also see a scarlet fish, a harmonium and a kite here."), the printing of which is done by the "listing nondescript items of" activity. As soon as any item picks up the either/or property "mentioned", by having its name printed, it is struck out so that it will not appear subsequently, whatever its priority.

3. Examples. As general advice: if the effect wanted can be got using "writing a paragraph about" and "listing nondescript items of" alone, use those; if it's necessary to meddle further, use "choosing notable locale objects" and "printing a locale paragraph" to alter the normal processes; use the all-powerful "printing the locale description" activity only when the whole process needs to be altered, not the item-by-item workings.

(a) In the Very Misty Moorlands, nothing on the ground can ordinarily be seen through the swirling mist, so the locale description is suppressed entirely:

```
Rule for printing the locale description of the Very Misty Moorlands:
    say "Mist coils around your feet, thick as a blanket. You cannot even see the ground you walk upon." instead.

Report taking something in the Very Misty Moorlands:
    say "You grope blindly in the mist and pick up [the noun]." instead.
```

(b) Here we take the chance to insert an additional paragraph into the locale description. This does relate to an item which might be described later, but where the player doesn't know that:

```
The Horological Workshop is a room. The marble table is fixed in place in the Workshop.

The parcel is a closed opaque container on the marble table. The alarm clock is a device in the parcel. The alarm clock is switched on.

Before printing the locale description of a room (called the locale):
	if the locale encloses the alarm clock and the alarm clock is switched on, say "A faint ticking noise can be heard."
```

Example

358. **Priority Lab**
    A debugging rule useful for checking the priorities of objects about to be listed.
    RB 3.1 Room Descriptions

§18.27 Choosing notable locale objects for something

1. When it happens. See "printing the locale description". This activity is expected to decide which items ought to be mentioned in a locale description for a given room, enterable container or enterable supporter, and to give each item a priority, which is a number ranging upwards from 1 (which is the top priority). The lower the priority number, the earlier the mention, or at least, the earlier the opportunity to be mentioned: it's up to other activities whether to give it a paragraph of its own or not. This activity only makes something a candidate, and decides what order the candidates will be tried in.

2. The default behaviour. By default, this activity contains only the "standard notable locale objects rule". This chooses exactly those items directly contained by the locale, assigning all of them priority 5. Note that this includes scenery, and other probably unwanted items - those will be excluded later.

3. Examples. (a) In the Misty Moorlands, only large items on the ground are visible through the mist:

```
A thing can be large or small. A thing is usually small. The stepladder is a large thing in the Misty Moorlands.

Rule for choosing notable locale objects for the Misty Moorlands:
	repeat with item running through large things in the Misty Moorlands:
		set the locale priority of the item to 5.

Report taking a small thing in the Misty Moorlands:
	say "You grope blindly in the mist and pick up [the noun]." instead.
```

Note the special phrase

```
set the locale priority of the item to 5;
```

which should be used only in rules for locale activities. It makes the given item a candidate and sets its priority. (Setting the priority to 0 forces an item not to be a candidate, and can thus undo the effect of previous rules.)

It's best to avoid situations where an item has a locale priority which is higher than that of something it is on top of, or inside, since this can result in an oddly-worded description.

Examples

359. **Low Light**
An object that is only visible and manipulable when a bright light fixture is on.
RB 3.1 Room Descriptions

360. **Casino Banale**
Creating room descriptions and object descriptions that change as the player learns new facts and pieces things together.
RB 5.5 Memory and Knowledge

§18.28 Printing a locale paragraph about

1. When it happens. See "printing the locale description". By this point, the locale description process has identified a number of items as candidates to be described, and worked out a priority order. This activity is then called for each candidate in turn, starting with the highest priority items and working downwards. It can either print some text or not, and can either mark the item as "mentioned" or not: if it does, then the item won't appear subsequently in the locale description. If the activity does nothing, the item becomes "nondescript" and falls through into the final "You can also see…" paragraph, unless another rule mentions it in the mean time.

2. The default behaviour. Is provided by a sequence of seven rules:

(1) The "don't mention player's supporter in room descriptions rule" excludes anything the player is directly or indirectly standing on or, less frequently, in. The header of the room description has probably already said something like "Boudoir (on the four-poster bed)", so the player can't be unaware of this item.

(2) The "don't mention scenery in room descriptions rule" excludes scenery.

(3) The "don't mention undescribed items in room descriptions rule" excludes the player object. (It's redundant to say "You can also see yourself here.") At present nothing else in I7 is "undescribed" in this sense.

(4) The "set pronouns from items in room descriptions rule" adjusts the meaning of pronouns like IT and HER to pick up items mentioned. Thus if a room description ends "Mme Tourmalet glares at you.", then HER would be adjusted to mean Mme Tourmalet.

(5) The "offer items to writing a paragraph about rule" gives the "writing a paragraph about" activity a chance to intervene. We detect whether it does intervene or not by looking to see if it has printed any text.

(6) The "use initial appearance in room descriptions rule" prints the "initial appearance" property of an item which has never been handled as a paragraph, if it has one.

(7) The "describe what's on scenery supporters in room descriptions rule" is somewhat controversial. It prints text such as "On the mantelpiece is a piece of chalk." for items which, like the mantelpiece, are scenery mentioned - we assume - in the main room description. (It is assumed that scenery supporters make their contents more prominently visible than scenery containers, which we do not announce the contents of.)

3. Examples. If all that's required is to supply an interesting paragraph of room description about something then it's always better to use the "writing a paragraph about" activity, not this one. This activity should only be used when the mechanism itself needs to be adjusted.

(a) The following excludes doors from room descriptions:

```
For printing a locale paragraph about a door (called the item)
    (this is the don't mention doors in room descriptions rule):
    set the locale priority of the item to 0;
    continue the activity.
```

(It's usually a good idea to "continue the activity" at the end of rules for this activity, since usually they all need to take effect for a happy outcome to the process. Here it doesn't really matter, since we were trying to stop anything from happening about the door, but it doesn't do any harm either.)

(b) Here's how to abolish what may be the most contentious rule in the whole Standard Rules:

```
The describe what's on scenery supporters in room descriptions rule is not listed in any
rulebook.
```

Examples

361. **Kiwi**
   Creating a raised supporter kind whose contents the player can't see or take from the ground.
   RB 8.4 Furniture

362. **Copper River**
   Manipulating room descriptions so that only interesting items are mentioned, while objects that are present but not currently useful to the player are ignored.
   RB 3.1 Room Descriptions

§18.29 Deciding the scope of something

1. When it happens. "Scope" is a term of art in interactive fiction programming: it means the collection of things which can be interacted with at any given moment, which depends on who you are and where you are. Commands typed by the player will only be allowed to go forward into actions if the things they refer to are "in scope". Inform also needs to determine scope at other times, too: for instance, when deciding whether a rule conditional on being "in the presence of" something is valid. It is a bad idea to say anything during this activity.

2. The default behaviour. Is complicated: see the Inform Designer's Manual, 4th edition, page 227. Briefly, the scope for someone consists of everything in the same place as them, unless it is dark.

3. Examples. (a) We very rarely want to forbid the player to refer to things close at hand, but often want to allow references to distant ones. For instance, a mirage of something which is not present at all:

```
After deciding the scope of the player while the location is the Shrine:
    place the holy grail in scope.
```

Two different phrases enable us to place unusual items in scope:

place (object) **in scope**

This phrase should only be used in rules for the "deciding the scope of..." activity. It places the given object in scope, making it accessible to the player's commands, regardless of where it is in the model world. Examples:

```
place the distant volcano in scope;
place the lacquered box in scope, but not its contents;
```

Ordinarily if something is placed in scope, then so are its parts and (in the case of a supporter or a transparent or open container) its contents; using the "but not its contents" option we can place just the box itself in scope.

place the/-- contents of (object) **in scope**

This phrase should only be used in rules for the "deciding the scope of..." activity. It places the things inside or on top of the given object in scope, making them accessible to the player's commands, but it does nothing to place the object itself in scope. (It might of course be in scope anyway, and if it is then this phrase won't remove it.) Example:

```
place the contents of the lacquered box in scope;
place the contents of the Marbled Steps in scope;
```

Note that the object in question can be a room, as in this second example.

(b) Another useful device is to be able to see, but not touch, another room:

```
The Cloakroom is a room. "This is just a cloakroom, but through a vague, misty mirror-window you can make out the Beyond." After looking in the Cloakroom, say "In the mirror you can see [list of things in the Beyond]."

After deciding the scope of the player while the location is the Cloakroom: place the Beyond in scope.

The Beyond is a room. Johnny Depp is a man in the Beyond.
```

(This must, however, also be a mirage, as at time of writing Mr Depp is alive and as well as can be expected following the reviews of "Charlie and the Chocolate Factory".) Note that "place the Ballroom in scope" doesn't just allow the player to talk about the dancers, the chamber musicians and so forth, also allows, say, "EXAMINE BALLROOM". To get one but

not the other, use "place the contents of the Ballroom in scope" or "place the Ballroom in scope, but not its contents".

(c) In darkness, the scope of someone is ordinarily restricted to his or her possessions (and body), but we can override that:

```
After deciding the scope of the player while in darkness: place the location in scope.
```

4. A note about actions. This activity takes place during the process of understanding the player's command, when the action that will take place is not fully known. So if the player types "TAKE SHOEBOX", this activity would happen when SHOEBOX is being examined for meaning. Inform knows the action it would be taking if the current line of command grammar were to be accepted, but it does not yet know to what objects that command would be applied. That means attaching a proviso like "… while taking a container" to a rule for this activity will cause the rule to have no effect - whereas "… while taking" would be fine.

Examples

363. **Peeled**
   Two different approaches to adjusting what the player can interact with, compared.
   RB 3.7 Lighting

364. **Four Stars 2**
   Using "deciding the scope" to change the content of lists such as "the list of audible things which can be touched by the player".
   RB 3.8 Sounds

365. **Ginger Beer**
   A portable magic telescope which allows the player to view items in another room of his choice.
   RB 9.11 Clocks and Scientific Instruments

366. **Rock Garden**
   A simple open landscape where the player can see between rooms and will automatically move to touch things in distant rooms.
   RB 3.4 Continuous Spaces and The Outdoors

367. **Stately Gardens**
   An open landscape where the player can see landmarks in nearby areas, with somewhat more complex room descriptions than the previous example, and in which we also account for size differences between things seen at a distance.
   RB 3.4 Continuous Spaces and The Outdoors

§18.30 Clarifying the parser's choice of something

1. When it happens. When the player has typed an ambiguous noun reference, and Inform has made a decision about what was meant, and it matters what this decision is. (If the decision is between three identical gold coins, say, then it doesn't matter, and this activity does not take place.) There are a couple of limitations on this: the activity applies only to the first noun, and only if it's an object. So for a command like SELECT BLUE, where BLUE is a noun referring to a colour value, say, this activity isn't used. But the simple case where the activity does play a part is nevertheless very useful.

2. The default behaviour. Text in brackets such as "(the laminated mahogany box)" is printed, on its own line.

3. Examples. (a) In the following, asking to TAKE TOWER results in the parser choosing the souvenir model (because of the "does the player mean…" rule making the alternative unlikely), and then explaining itself by saying "(The little one, obviously.)" instead of "(the

souvenir model Eiffel Tower)".

```
The Champs du Mars is a room. The great Eiffel Tower is here. "The great Tower stands high over you." The souvenir model Eiffel Tower is here. "Comparatively tiny is the souvenir version." The great Eiffel Tower is fixed in place. Does the player mean taking the great Eiffel Tower: it is very unlikely.

Rule for clarifying the parser's choice of the model tower: say "(The little one, obviously.)"
```

4. A note about actions. This activity takes place during the process of understanding the player's command, when the action that will take place is not fully known. So if the player types "TAKE SHOEBOX", this activity would happen when SHOEBOX is being examined for meaning. Inform knows that the action will be taking, but nothing else. That means attaching a proviso like "… while taking a container" to a rule for this activity will cause the rule to have no effect - whereas "… while taking" would be fine.

§18.31 Asking which do you mean

1. When it happens. When the player has typed an ambiguous noun reference, and Inform has not been able to decide what was meant.

2. The default behaviour. A question such as "Which do you mean, the laminated mahogany box or the boom box?" is printed. (This activity shapes the question: it is not responsible for parsing the answer. It would be very mysterious to write rules for this activity such that nothing is printed, because the player would then have no idea what to type.)

3. Examples. The question is harder to print than may first appear, since one must not simply list the options, but also take into account collections of plural objects ("Which do you mean, the gold-tipped pen or a gold coin?"). It is probably better not to try to rewrite this.

(a) But we can place notes before or after: here is a verbose explanation for beginners to IF.

```
Before asking which do you mean: say "Okay, so I'm going to have to ask a question now: you've typed something ambiguous, and I don't know which noun you're referring to."

After asking which do you mean: say "(Just type a word or two to give me more information.)"
```

(b) We can also use this activity as a context for other activities. For instance:

```
The Champs du Mars is a room. The great Eiffel Tower is here. "The great Tower stands high over you." The souvenir model Eiffel Tower is here. "Comparatively tiny is the souvenir version." The great Eiffel Tower is fixed in place. Understand "actual" as the great Tower.

Rule for printing the name of the great Tower while asking which do you mean: say "actual Tower". Rule for printing the name of the souvenir tower while asking which do you mean: say "souvenir".
```

causes TAKE TOWER (for instance) to produce a nice tidy question in reply: "Which do you mean, the actual Tower or the souvenir?"

4. A note about actions. This activity takes place during the process of understanding the player's command, when the action that will take place is not fully known. So if the player types "TAKE SHOEBOX", this activity would happen when SHOEBOX is being examined for meaning. Inform knows that the action will be taking, but nothing else. That means attaching a proviso like "… while taking a container" to a rule for this activity will cause the rule to have no effect - whereas "… while taking" would be fine.

Examples

368. **Apples**
Prompting the player on how to disambiguate otherwise similar objects.
RB 6.17 Clarification and Correction

369. **Originals**
Allowing the player to create models of anything in the game world; parsing the name "model [thing]" or even just "[thing]" to refer to these newly-created models; asking "which do you mean, the model [thing] or the actual [thing]" when there is ambiguity.
RB 9.12 Cameras and Recording Devices

370. **Walls and Noses**
Responding to "EXAMINE WALL" with "In which direction?", and to "EXAMINE NOSE" with "Whose nose do you mean, Frederica's, Betty's, Wilma's or your own?"
RB 6.17 Clarification and Correction

§18.32 Supplying a missing noun/second noun

1. When it happens. (Two different activities here, but identical except for applying to different nouns.) This happens when an Understand sentence fails to supply a noun for an action which requires one. For example, in the sentence 'Understand "seize" as taking.' - the "taking" action is incompletely specified, because it requires a noun, and

there's no noun in the command to be understood.

Note that this is not what happens if it's the player who fails to supply the noun. That is, suppose the player types a half-finished command like TAKE, which can't be matched against (for example) 'Understand "take [things]" as taking.' because the player didn't name any thing(s). Typically a story will reply to such a command with a question asking for clarification, but sometimes it makes guesses about what was meant. The "supplying a missing noun" activity plays no part in this guesswork, and can't influence it: that's the task of the "does the player mean" rulebook.

Suppose we do have the first of these cases, then. "Supplying a missing noun" takes place to remedy the problem. It can either:

(i) Set a noun, printing text like "(presumably the black bag)" if it wants, in which case the action goes forward, though it is still subject to the full rules on accessibility exactly as any other action would be; or

(ii) Make no choice, in which case no action takes place and the player's command is rejected. If the activity printed nothing, Inform will produce a generic reply to the player that "You must supply a noun.".

2. The default behaviour. In the default grammar for Inform, only three such half-finished actions are ever Understood. One is "going" with no direction, for which this activity simply prints a refusal. The other two are the two undirected senses, "smelling" and "listening". In each case, the "supplying a missing noun" activity sets the noun to the current location: so, for instance, typing the bare command "listen" might generate the action "listening to the Shoreline".

3. Examples. (a) This is the definition Inform uses to make "listen" work as outlined above:

```
Rule for supplying a missing noun while listening (this is the ambient sound rule):
    now the noun is the location.
```

(b) It can be elegant to allow second nouns to be dropped with habitual actions, or where the choice is obvious:

```
Understand "unlock [something]" as unlocking it with.

Rule for supplying a missing second noun while unlocking:
    if the skeleton key is carried, now the second noun is the skeleton key;
    otherwise say "You will have to specify what to unlock [the noun] with."
```

Note that, in order for our activity to succeed, we do need to supply a grammar line allowing the player to try "unlocking it with" using only one noun. Otherwise, the command "unlock something" will still produce the question "What do you want to unlock the door with?"

Examples

371. **Latin Lessons**
 Supplying missing nouns and second nouns for other characters besides the player.
 RB 7.14 Obedient Characters

372. **Minimal Movement**
 Supplying a default direction for "go", so that "leave", "go", etc., are always interpreted as "out".
 RB 6.9 Going, Pushing Things in Directions

§18.33 Reading a command

1. When it happens. When reading a command from the keyboard.

2. The default behaviour. Print the prompt text; wait for the player to type something and press return. Reject an entirely blank line, and treat a command beginning "oops" as a correction to the previous one. This is a fairly complicated business, so it is probably best not to change the "for" rules for this activity: "before", and especially "after", are another matter. (Note, however, that if Inform does reject a blank line and ask for another then this all happens inside the "for" rules: no "after" occurs after the blank line, nor does a "before" happen before the second attempt by the player. It is all a single round of the activity, not two.)

3. Examples. (a) To lead absolute beginners in gently:

```
Before reading a command while the turn count is 1, say "(This is your chance to say what the protagonist should do next. After the '>', try typing 'take inventory'.)"
```

(b) The following responds politely but firmly if the player tries to type "please look", say, instead of just "look":

```
After reading a command:
    if the player's command includes "please":
        say "Please do not say please.";
        reject the player's command.
```

To explain. Fragments of what the player has typed are called snippets: "the player's

command" is the entire thing. We can test if a snippet matches a given pattern like so:

if (snippet) **matches** (topic):

This condition is true if the given snippet exactly matches the specification. Example:

```
if the player's command matches "room [number]", ...
```

will be true if the command is ROOM 101, but not if it's EXPLORE ROOM 7.

if (snippet) **does not match** (topic):

This condition is true if the given snippet does not exactly match the specification.

if (snippet) **includes** (topic):

This condition is true if the given snippet includes words matching the specification, either at the beginning, in the middle, or at the end. Example:

```
if the player's command includes "room [number]", ...
```

will be true if the command is ROOM 101, EXPLORE ROOM 7, or ROOM 22 AHOY, but not if it's VISIT ROOM GAMMA 7.

if (snippet) **does not include** (topic):

This condition is true if the given snippet does not include any run of words which matches the specification.

Lastly, we took drastic action with another new phrase:

reject the player's command

This phrase should be used only in rules for the "reading a command" activity. It tells Inform not to bother analysing the text further, but to go back to the keyboard. (No time passes; no turn elapses; nothing happens in the simulated world.)

(c) An improved version takes commands like "please drop the coin" and strips "please" from them, but then allows them to proceed normally:

```
After reading a command:
    if the player's command includes "please":
        say "(Quelle politesse! But no need to say please.)";
        cut the matched text.
```

"Matched text" is a snippet containing the words which matched against the pattern in the most recent "includes" condition, so in this case it contains just the single word "please". Two phrases allow snippets to be altered:

replace (snippet) **with** (text)

This phrase should be used only in "after" rules for the "reading a command" activity; it replaces the snippet of command, usually the "matched text" found immediately before, with the given text. Example:

```
if the player's command includes "room [number]":
    replace the matched text with "office".
```

cut (snippet)

This phrase should be used only in "after" rules for the "reading a command" activity; it removes the snippet of command. Example:

```
if the player's command includes "or else":
    cut the matched text.
```

Note that "replace" and "cut" can only be used in "after reading a command" rules: not when an action has been chosen and has gone ahead into its rulebooks. Once the

"reading a command" activity has finished, the command is final.

(d) To make the word "grab" an abbreviation for "take all":

```
After reading a command:
    if the player's command matches "grab", replace the player's command with "take all".
```

("Snippet" is actually a kind of value, so we could say "Ah, you typed '[the player's command]'!" or some such if we liked. But in practice only three snippets are likely to be useful: the two mentioned above, "player's command" and "matched text", and the "topic understood", used when matching the "[text]" token in command grammar.)

(e) Finally, we can make still more detailed alterations to the text of the command using the techniques presented in the Advanced Text chapter. For instance:

change the text of the player's command to (text)

This phrase should be used only in "after" rules for the "reading a command" activity; it replaces the current command text entirely. Example:

```
After reading a command:
    let T be "[the player's command]";
    replace the regular expression "\p" in T with "";
    change the text of the player's command to T.
```

This converts the player's command to text, which is then manipulated by searching for any punctuation mark and replacing it with blank text (that is, deleted), and then put back again as the new command.

Examples

373. **Cloves**
    Accepting adverbs anywhere in a command, registering what the player typed but then cutting them out before interpreting the command.
    RB 6.18 Alternatives To Standard Parsing

374. **Fragment of a Greek Tragedy**
    Responding to the player's input based on keywords only, and overriding the original parser entirely.
    RB 6.18 Alternatives To Standard Parsing

375. **North by Northwest**
    Creating additional compass directions between those that already exist (for instance, NNW) -- and dealing with an awkwardness that arises when the player tries to type "north-northwest". The example demonstrates a way around the nine-character limit on parsed words.
    RB 2.2 Varying What Is Read

376. **Complimentary Peanuts**
    A character who responds to keywords in the player's instructions and remarks, even if there are other words included.
    RB 7.8 Saying Complicated Things

§18.34 Implicitly taking something

1. When it happens. When an action is tried which requires the actor (normally the player, of course) to be carrying something, but which is not in fact carried by that person. For instance, if the player types WEAR OVERCOAT in reference to a Moroccan overcoat currently draped over a chair.

2. The default behaviour. To print text such as "(first taking the Moroccan overcoat)" and then silently try taking the object in question (the overcoat). If the take succeeds, the silence means that nothing else is printed: if it fails, it will say why.

No matter what rules are written for this activity, it is impossible to use it to allow the action to go ahead even without the item. The activity allows us to change how, or if, an implicit take will happen, but not to change the consequences of failure. (To do that, we would need to say that "The carrying requirements rule does nothing", but this kind of unstitching of the action machinery needs to be done with caution.)

3. Examples. (a) Forbidding implicit takes for certain dangerous items. (This seems especially fair if taking such items might cause death: the player will not wish to be killed

on the strength only of our guess as to what he might be intending to do.)

```
Rule for implicitly taking the curare:
    say "Ordinarily you'd pick up the curare in order to be able to do that, but this seems like a good moment for caution." instead.
```

(b) Changing the way the implicit action is reported for the player:

```
Rule for implicitly taking something (called target):
    try silently taking the target;
    if the player carries the target, say "You appropriate [the target] first, of course. [run paragraph on]"
```

(c) Combining implicit takes when the noun and second noun must both be carried:

```
Rule for implicitly taking the noun when the second noun is a thing and the second noun is not carried by the player:
    try silently taking the noun;
    try silently taking the second noun;
    say "(first taking both [the noun] and [the second noun])[line break]"
```

(d) Making another character reply amusingly:

```
Rule for implicitly taking something which is carried by the player when the person asked is Clark:
    say "'I don't see how I'm supposed to do that when you're holding [the noun],' remarks Clark sulkily." instead.
```

(e) Causing implicit takes which wouldn't otherwise happen. Suppose we have a photographing action, and there are very small flowers which can't conveniently be snapped without being first picked. We then want an implicit take to occur, even though we wouldn't want this for other sorts of photography. So:

```
Check an actor photographing a flower:
    if the actor is not carrying the noun:
        carry out the implicitly taking activity with the noun;
        if the actor is not carrying the noun, stop the action.
```

Note that if the activity doesn't succeed in taking the item, it's expected to print some text explaining this, which is why we don't need to say anything further.

Examples

377. **The Big Sainsbury's**
Making implicit takes add a minute to the clock, just as though the player had typed TAKE THING explicitly.
RB 4.1 The Passage Of Time

378. **Pizza Prince**
Providing a pizza buffet from which the player can take as many pieces as he wants.
RB 10.3 Dispensers and Supplies of Small Objects

379. **Lollipop Guild**
Overriding the rules to allow the player to show something to another character without first taking it.
RB 6.8 Taking, Dropping, Inserting and Putting

§18.35 Printing a parser error

1. When it happens. The parser is the part of the run-time software, included in all works produced by Inform, which tries to match the player's command against the grammar provided by the work. When it is unable to make a valid match, the parser prints an error to the player: for instance,

```
> BIFURCATE TREE
That's not a verb I recognise.
```

There are more than twenty possible messages. The one which the parser wants to say is stored in the variable "latest parser error", which has the convenient kind "command parser error". This has the following possible values:

```
didn't understand error
only understood as far as error
didn't understand that number error
can't see any such thing error
said too little error
aren't holding that error
can't use multiple objects error
can only use multiple objects error
not sure what it refers to error
excepted something not included error
can only do that to something animate error
not a verb I recognise error
not something you need to refer to error
can't see it at the moment error
didn't understand the way that finished error
not enough of those available error
nothing to do error
referred to a determination of scope error
I beg your pardon error
noun did not make sense in that context error
can't again the addressee error
comma can't begin error
can't see whom to talk to error
can't talk to inanimate things error
didn't understand addressee's last name error
```

2. The default behaviour. Prints the message in question.

3. Examples. (a) Perhaps for newcomers:

```
After printing a parser error:
    say "If you are new to interactive fiction, you may like to try typing HELP."
```

(b) Or to give the parser a certain amount of character:

```
Rule for printing a parser error when the latest parser error is the I beg your pardon error:
    say "What's that? Speak up, speak up." instead.

Rule for printing a parser error:
    say "That's a rum thing to say, and no mistake." instead.
```

(c) This can be helpful for seeing what's going on:

```
Rule for printing a parser error:
    say "The [latest parser error] happened.";
    continue the activity.
```

Examples

380. **WXPQ**
    Creating a more sensible parser error than "that noun did not make sense in this context".
    RB 6.17 Clarification and Correction

381. **Xot**
    Storing an invalid command to be repeated as text later in the game.
    RB 2.3 Using the Player's Input

§18.36 Deciding whether all includes

1. When it happens. When parsing a command such as "take all", where the player uses "all" to signify everything within reach.

2. The default behaviour. The actual method used is complicated, as "all" is not as simple as it seems - "take all" would not include the player's own body, for instance, or the crescent moon. The point of this activity is to allow the normal method to be changed for given objects, or given kinds of object.

3. Examples. (a) Removing scenery from "all" (but see (4) below):

```
Rule for deciding whether all includes scenery: it does not.
```

The phrases "it does" and "it does not" make a decision.

(b) Ensuring that a given thing, which might otherwise be excluded, is included:

```
Rule for deciding whether all includes the oval roof: it does.
```

4. The Standard Rules already uses this. Note that the Standard Rules already stocks this activity with several rules:

```
exclude scenery from take all rule
exclude people from take all rule
exclude fixed in place things from take all rule
```

5. A note about actions. This activity takes place during the process of understanding the player's command, when the action that will take place is not fully known. So if the player types "TAKE SHOEBOX", this activity would happen when SHOEBOX is being examined for meaning. Inform knows that the action will be taking, but nothing else. That means attaching a proviso like "... while taking a container" to a rule for this activity will cause the rule to have no effect - whereas "... while taking" would be fine.

§18.37 Printing the banner text

1. When it happens. The banner is the bibliographic masthead text, which typically looks something like this:

```
Relations
An Interactive Fiction by Emily Short
Release 1 / Serial number 050630 / Inform 7 build 2U98 (I6/v6.30 lib 6/10N) SD
```

(The serial and build numbers are those applying when the story file was last made: these ones are from the mid-2000s.) The banner is printed at the start of play, and when the player types "version" at the command line, and when say "[banner text]" occurs.

say "[the/-- banner text]"

This text substitution expands to the banner text giving bibliographic details of the current story file, rather like the opening credits of a movie, or the title page of a book.

2. The default behaviour. Prints the text above, giving the title, the headline, the author, the release number, the date of compilation (that's the serial number: YYMMDD), and version numbers of the Inform components used to put the story together.

3. Examples. (a) Adding a line to the banner:

```
After printing the banner text, say "DRM authentication code:
13S-451-2034u75y65u%%a1248."
```

(b) Simplifying the banner:

```
Rule for printing the banner text: say "Welcome." instead.
```

Example

382. **Bikini Atoll**
   Delaying the banner for later.
   RB 11.1 Start-Up Features

§18.38 Printing the player's obituary

1. When it happens. The obituary is the text "*** You have died ***" or similar, sometimes followed by a final score, if the appropriate use option ("Use scoring.") is in force.

2. The default behaviour. Printing the aforementioned text, then the final score, and reducing the status line to a largely blank state.

3. Examples. Here's one way to add to the verdict of history:

```
After printing the player's obituary: say "And you visited [number of visited rooms] place[s]."
```

Examples

383. **Battle of Ridgefield**
   Completely replacing the endgame text and stopping the game without giving the player a chance to restart or restore.
   RB 11.6 Ending The Story

384. **Finality**
   Not mentioning UNDO in the final set of options.
   RB 11.6 Ending The Story

385. **Jamaica 1688**
   Adding a feature to the final question after victory, so that the player can choose to reveal notes about items in the game.
   RB 11.6 Ending The Story

§18.39 Amusing a victorious player

1. When it happens. When the player chooses "AMUSING" from the short menu of choices after a story has been won. Traditionally, this is where the author gets to point out quirky by-ways of the story, or make some final acknowledgements, or simply salute the player's perseverance. Note that the AMUSING option is only offered when the story has ended finally, and that it is only offered if there is at least one rule present in the "for amusing a victorious player" rulebook.

2. The default behaviour. None. The "for amusing a victorious player" rulebook is empty by default, and no amusement is available.

3. Examples. The format would be like so:

```
Rule for amusing a victorious player: say "Hmm. You're easily amused."
```

Example

386. **Xerxes**
Offering the player a menu of things to read after winning the game.
RB 11.6 Ending The Story

§18.40 Starting the virtual machine

1. When it happens. This activity is provided solely as a "hook" for any low-level tasks which need to be performed when the virtual computer which runs Inform story files is starting up. This happens much earlier than "when play begins" rules, and should be used only as a last resort.

It should be remembered that Inform can produce story files for several different virtual computers. On some of these, it will not be safe to print any text during this activity, as the windows which would display such text do not yet exist.

2. The default behaviour. None.

3. Examples. No detailed examples will be given here, but the activity might be used (for instance) to set styles for the Glulx windows shortly to be brought into existence.

Example

387. **Blankness**
Emptying the status line during the first screen of the game.
RB 12.2 The Status Line

19. Rulebooks

§19.1 On rules
§19.2 Named rules and rulebooks
§19.3 New rules
§19.4 Listing rules explicitly
§19.5 Changing the behaviour of rules
§19.6 Sorting and indexing of rules
§19.7 The preamble of a rule
§19.8 New rulebooks
§19.9 Basis of a rulebook
§19.10 Rulebook variables
§19.11 Success and failure
§19.12 Named outcomes
§19.13 Rulebooks producing values
§19.14 Abide by
§19.15 Two rulebooks used internally
§19.16 The Laws for Sorting Rulebooks

§19.1 On rules

When we open the casing and look inside the machinery of Inform, what we see are rules and rulebooks. We seldom need to know how this machinery works, but every once in a while we want to replace components, or even install new mechanisms of our own. And as we shall see, creating new rulebooks can be a neat way to tackle complicated simulations full of exceptions and special cases.

So far we have seen many rules, and the term "rulebook" has frequently but vaguely been used. Here is a summary of the rulebooks seen so far:

```
before
instead
after
check taking, carry out taking, report taking
```
and three similar rulebooks for each of the 90 or so actions
```
persuasion
unsuccessful attempt
reaching inside
reaching outside
visibility
does the player mean

when play begins
when play ends
every turn
when Confrontation Scene begins
when Confrontation Scene ends
```
and two similar rulebooks for each scene we create, if any
```
before printing the name of
for printing the name of
after printing the name of
```
and three similar rulebooks for each of the 20 or so activities

Which makes around 340 rulebooks before we even start to write. All the same, not everything in Inform belongs to a rulebook - timed events, for example, are rules which normally live outside of rulebooks; and other constructions, such as newly-created phrases, or definitions, may look vaguely like rules, but they aren't. So the following are not rulebooks:

```
At 11:10 PM: ...
To dislodge the shelf: ...
Definition: ...
```

§19.2 Named rules and rulebooks

Most of the rules built into Inform have names. For instance, a rule called "the advance time rule" is the one which increments the number of turns and advances the clock, values which are usually not visible, but are ticking away behind the scenes.

A rulebook is a list of rules to be followed in sequence until one of them makes a decision. For instance, when actions get to the "instead" stage, each "instead" rule is tried until one of them chooses to do something. If the source text contains the rules

```
Instead of taking something: say "You have no particular need just now."
Instead of taking a fish: say "It's all slimy."
```

and a command to TAKE something is tried, then only one of these rules will have any effect. The "instead" rulebook contains:

```
Rule (1) to be applied if the action matches "taking a fish"
Rule (2) to be applied if the action matches "taking something"
```

Inside their rulebook, the rules are not listed in the order of definition in the source text. Rule (1) comes before rule (2) because it applies in more specific circumstances. This is the main idea: a rulebook gathers together rules about making some decision, or taking some action, and sorts them in order to give the more specific rules first choice about whether they want to intervene.

Whereas only some rules are named (the two "instead" rules above have no name, for instance), every rulebook has a name. For convenience, the following forms of rule and rulebook name are synonymous:

```
advance time = the advance time rule

the instead rules = instead rulebook = instead
```

The names of built-in rules have been chosen as descriptively as possible: the "can't go through closed doors rule", for instance. Names for rules tend to be verbose, but this is a situation where clarity is very much better than brevity.

Examples

388. **Nine AM Appointment**
A WAIT [number] MINUTES command which advances through an arbitrary number of turns.

RB 6.11 Waiting, Sleeping

389. **Delayed Gratification**
A WAIT UNTIL [time] command which advances until the game clock reaches the correct hour.

RB 6.11 Waiting, Sleeping

§19.3 New rules

Stretching a point seasonally, we might write:

```
Every turn, say "The summer breeze shakes the apple-blossom."
```

This rule is nameless. It needs no name because it will never need to be referred to: by identifying it as an every turn rule we have already said enough to lodge it in the "every turn" rulebook. In fact, though, it is easy to create a named rule:

```
This is the blossom shaking rule: say "The summer breeze shakes the apple-blossom."
```

The name of a rule must always end with the word "rule", for clarity's sake. (The phrasing "This is the … rule" is used because "The … rule" would be open to misinterpretation.)

Previously we had a rule which had no name, but belonged to a rulebook: now we have the opposite, because although the "blossom shaking rule" has a name, it has not been placed in any rulebook. That means it will probably never be applied, unless we give specific instructions for that.

Alternatively, it is possible to both name and place a rule in a single sentence:

```
Every turn (this is the alternative blossom rule): say "The summer breeze shakes the
apple-blossom."
```

Now the "alternative blossom rule" is a named rule in the "every turn" rulebook.

Examples

390. **The Crane's Leg 2**
A description text generated based on the propensities of the player-character, following different rulebooks for different characters.
RB 5.6 Viewpoint

391. **Stone**
A soup to which the player can add ingredients, which will have different effects when the player eats.
RB 9.1 Food

392. **Bribery**
A GIVE command that gets rid of Inform's default refusal message in favor of something a bit more sophisticated.
RB 7.4 Barter and Exchange

§19.4 Listing rules explicitly

If rules can manage perfectly well without, why bother to have names for rules?

The answer is that although Inform contains an elaborate mechanism for placing rules into the correct rulebook at the correct position, and this happens automatically, Inform will sometimes get it wrong. It will use a rule we do not want, or place them in an order which does not suit us. To put this right, we can give explicit instructions which take precedence over Inform's normal practice. This is done with the "to list" verb, as in the following examples.

1. The simplest usage is to place a named rule, which currently has no home, in any rulebook of our choice. (This looks redundant, but just occasionally we want the same rule to appear in two different rulebooks.)

```
The blossom rule is listed in the every turn rules.
```

A rule can appear in more than one rulebook, but within any single rulebook it can only appear once.

2. We can also specify that the rule needs to appear before, or after, some other named rule in the same rulebook:

```
The collapsing bridge rule is listed before the moving doorways rule in the instead rules.
```

Instead of being placed in specificity order in the whole "instead" rulebook, the "collapsing bridge" rule would now be placed in specificity order only in the first half of the "instead" rulebook - the rules from the start up to (but not including) the "moving doorways" rule. To reiterate: that doesn't necessarily mean it will be immediately before the "moving doorways" rule; it will be placed according to Inform's usual sorting rules within that range.

"Listed" sentences are obeyed by Inform in sequence, so if later ones issue instructions contradicting earlier ones, it's the later ones which win out. Thus if we say "A is listed before B in X" and then "B is listed before A in X", the result is that B comes before A.

3. We can specify that a rule needs to appear first or last in a given rulebook:

```
The collapsing bridge rule is listed first in the instead rules.
```

Again, if we make several such instructions about the same rulebook then the most recent one wins: "A is listed first in X. B is listed first in X. C is listed first in X." causes rulebook X to begin C, B, A.

4. We can also substitute one rule for another:

```
My darkness rule is listed instead of the can't act in the dark rule in the visibility rules.
```

If rule A is listed instead of rule B in rulebook X, and A was already a rule in rulebook X, then A will move from its previous position to occupy the place where B was, and B will disappear. (In particular rule A will not be duplicated, which would break the principle that no rule occurs twice in the same rulebook.)

5. And we can strike down existing rules, either specifically or in all their applications:

```
The can't act in the dark rule is not listed in the visibility rules.
The can't remove from people rule is not listed in any rulebook.
```

This does not actually destroy the rules in question: they could still, for instance, be put into another rulebook, or even be applied explicitly, as we shall see. But unless we take deliberate action to the contrary, un-listing a rule amounts to abolishing it forever. This is a little drastic, and more subtle effects can be seen in the next section.

Examples

393. **Saint Eligius**
Adding a first look rule that comments on locations when we visit them for the first time, inserting text after objects are listed but before any "every turn" rules might occur.
RB 6.9 Going, Pushing Things in Directions

394. **Uptempo**
Adjust time advancement so the game clock moves fifteen minutes each turn.
RB 4.1 The Passage Of Time

395. **Verbosity 2**
Making rooms give full descriptions each time we enter, even if we have visited before, and disallowing player use of BRIEF and SUPERBRIEF.
RB 6.4 Looking

396. **Slouching**
A system of postures allowing the player and other characters to sit, stand, or lie down explicitly or implicitly on a variety of enterable supporters or containers, or in location.
RB 5.1 The Human Body

397. **Swigmore U.**
Adding a new kind of supporter called a perch, where everything dropped lands on the floor.
RB 8.4 Furniture

§19.5 Changing the behaviour of rules

Here is another way to abolish an already-existing rule:

```
The print final score rule does nothing.
```

The rule continues to be listed in any rulebook it would normally be listed in: but now it doesn't do anything. More usefully, we can attach a condition:

```
The print final score rule does nothing if the score is 0.
```

or:

```
The print final score rule does nothing unless the score is 100.
```

We can also substitute a rule of our own:

```
This is the print fancy final score rule:
    say "Oh my, you scored a mammoth [score]!"

The print fancy final score rule substitutes for the print final score rule.
```

and once again a condition can be applied:

```
The print fancy final score rule substitutes for the print final score rule when the score is greater than 100.
```

Example

398. **Access All Areas**
     The Pointy Hat of Liminal Transgression allows its wearer to walk clean through closed doors.

RB 10.10 Magic (Breaking the Laws of Physics)

§19.6 Sorting and indexing of rules

The Rules page of the index for a project offers a view of the rulebooks and their contents, with two major exceptions: built-in rules for specific actions are left to the Actions page, and any rules for scene endings or beginnings are left to the Scenes page.

As we have seen, we need to know the name of a rule before we can change its rulebook listing or alter its applicability. The Rules and Actions index pages show the names of the built-in rules, which are not worth memorising. (Typing can be saved by using the paste-into-source buttons, or by selecting a rule's name and copying and pasting it by hand.)

In the Rules index, each rulebook is named and then followed by a list of the rules within it, one on each line: if nothing follows, then the rulebook is currently empty. The rules are given in order, and icons are used which indicate which rules are more specific than which others. Hovering the mouse over such an icon should bring up a "tooltip" which explains Inform's reasoning.

As this suggests, Inform performs its automatic sorting using a precise collection of Laws (the term "rules" would be ambiguous here, so we call these guidelines Laws instead), and the tooltip shows which Law was applied. It is bad style to write source text which absolutely depends on detailed points of these Laws, but they are documented at the end of this chapter for those who do wish to see the full details.

§19.7 The preamble of a rule

In general, a rule looks like this:

```
preamble : list of one or more phrases divided by semicolons
```

though in a few common cases (where the preamble begins with Before, After, Instead of, Every turn, or When, and there is only one phrase in the list) the colon can be replaced with a comma. Three kinds of declaration are special, and these we can tell apart by the first word:

```
To ... - a new phrase: see the chapter on Phrases
At ... - something due to happen at a given time: see Time
Definition: ... - a new adjective: see Descriptions
```

All other declarations (that is, starting with any other word) create rules fit for going into rulebooks. The preamble can either just be a name, which is required to end with the word "rule", or it can give circumstances and have no name, or it can do both:

```
This is the ...name of rule...
...circumstances...
...circumstances... (this is the ...name of rule...)
```

The circumstances should be a sequence of the following ingredients, each of which is optional except the name of the rulebook:

first *or* last
followed by ...rulebook name...
followed by about *or* for *or* of *or* on *or* rule
followed by ...what to apply to...
followed by while *or* when ...condition...
followed by during ...a scene...

The word "first" or "last", if present, is significant: it tells Inform exactly where the new rule should be placed into its rulebook, and so overrides the normal practice of placing the rule according to how specific it is.

On the other hand, the use of any of the following:

```
for
of
rule about
rule for
rule on
```

is purely to make the text easier to read: Inform does not make any direct use of these words (except perhaps that it may help to avoid ambiguities by separating the rulebook name from what is being applied to). Thus in the rule

```
Instead of kissing Clark: ...
```

the word "instead" is the rulebook's name, while "of" is technically optional. "Instead rule about kissing Clark: …" would work just as well.

In this whole list of possible ingredients, only the rulebook name is compulsory. We could define a rule called simply "Instead: …" if we wanted - though its universal applicability would make it pretty disruptive, with every action stopped in its tracks.

§19.8 New rulebooks

Creating a new rulebook is also straightforward, as we see in the following modest example story:

```
"Appraisal"

The Passage is east of the Tomb. The green-eyed idol is in the Tomb. A Speak-Your-Progress machine is in the Passage.

Appraisal rules is a rulebook.

An appraisal rule: say "Click... whirr... the score is [the score in words] points."

An appraisal rule:
    if we have taken the idol, say "Most importantly of all, the idol has been found."

Instead of switching on the machine, follow the appraisal rules.
```

The creation of the rulebook is all very well, but without the final sentence it would never be used. The crucial new phrase here is:

follow (rule)

This phrase causes the rule to be obeyed immediately (rather than simply at predetermined times such as when a particular action is being tried, or at the end of every turn, and such). Example:

```
follow the advance time rule;
follow the appraisal rulebook;
```

Like "number" or "text", "rule" and "rulebook" are kinds of value built into Inform: "the blossom rule" is a value whose kind is "rule", whereas "the every turn rules" is a value whose kind is "rulebook". In fact, Inform considers a rulebook to be a special case of a rule, so that whenever a rule is required it is legal to name a rulebook instead, but not vice versa. The "follow" phrase here…

```
Instead of switching on the machine, follow the appraisal rules.
```

…expects to be applied to a value of kind "rule"; "the appraisal rules" is in fact a rulebook, but since that counts as a rule the phrase makes sense to Inform. To follow a rulebook means to run through all its rules in turn, stopping when one rule reaches an outcome; to follow a single rule means just that one, of course.

When created, a rulebook starts out with no rules in it - in this example, of course, we quickly defined a couple of rules to go into it. But it's often the case in Inform that a rulebook exists without ever being stocked up, especially if the rulebook is for some obscure purpose never needed. The built-in adjectives "empty" and "non-empty", applied to a rulebook, test whether any rule is present or not.

Examples

399. **Solitude**
   Novice mode that prefaces every prompt with a list of possible commands the player could try, and highlights every important word used, to alert players to interactive items in the scenery.

   RB 11.3 Helping and Hinting

400. **In Fire or in Flood**
   A BURN command; flammable objects which light other items in their vicinity and can burn for different periods of time; the possibility of having parts or contents of a flaming item which survive being burnt.

   RB 10.8 Fire

401. **Patient Zero**
   People who wander around the map performing various errands, and in the process spread a disease which only the player can eradicate.

   RB 7.13 Traveling Characters

§19.9 Basis of a rulebook

Every rulebook works on a value supplied to it, though it doesn't always look that way. The kind of the value is called its "basis"; for example, if a rulebook works on a number, it's called a "number based rulebook". Most of the rulebooks seen up to now have been action based rulebooks:

```
Instead of eating the cake: ...
```

"Instead" is an action based rulebook, and the action it works on is the one currently being processed. Besides before, after and instead, other action based rulebooks include the check, carry out, and report rules; general rulebooks such as every turn rules, the visibility rules, the turn sequence rules; and rules specially for dealing with the actions of other characters, such as the persuasion and unsuccessful attempt rules. But we have also seen object based rulebooks:

```
Rule for reaching inside the flask: ...
```

"Reaching inside" is an object based rulebook, and here we're giving it a rule which applies if the object is the flask. Inform would reject something like:

```
Rule for reaching inside 100: ...
```

because 100 has the wrong kind to fit - it's a number, not an object. There are many object based rulebooks, because most activities built-in to Inform act on objects. For example, the "printing the name of" activity has three rulebooks attached to it: before printing the name of, for printing the name of, after printing the name of. All of these are object based rulebooks.

Finally, we've also seen scene based rulebooks (which is how rules like "when a recurring scene ends" worked, in the Scenes chapter).

If a rulebook is declared like so:

```
Marvellous reasoning is a rulebook.
```

then it is an action based rulebook. If we want something different, we must write something like this:

```
Grading is a number based rulebook.
Grading 5: say "It's five. What can I say?" instead.
Grading an odd number (called N): say "There's something odd about [N]." instead.
Grading a number (called N): say "Just [N]." instead.

When play begins:
repeat with N running from 1 to 10:
    say "Grading [N]: ";
    follow the grading rulebook for N.
```

which produces:

```
Grading 1: There's something odd about 1.
Grading 2: Just 2.
Grading 3: There's something odd about 3.
Grading 4: Just 4.
Grading 5: It's five. What can I say?
Grading 6: Just 6.
Grading 7: There's something odd about 7.
Grading 8: Just 8.
Grading 9: There's something odd about 9.
Grading 10: Just 10.
```

Here we needed a variation on "follow" which supplies the value to apply to:

follow (values based rule producing values) **for** (value)

This phrase causes the rule to be obeyed immediately (rather than simply at predetermined times such as when a particular action is being tried, or at the end of every turn, and such), and applies it to the value given, which must be of a matching kind. Example:

```
follow the reaching inside rulebook for the electrified cage;
```

And here is an example based on objects:

```
The flotation rules are an object based rulebook.
A flotation rule for the cork: rule succeeds.
A flotation rule for an inflated thing: rule succeeds.
A flotation rule: rule fails.
```

And we might use the flotation rules in a circumstance like this:

```
After inserting something into the well:
    follow the flotation rules for the noun;
    if the rule succeeded:
        say "[The noun] bobs on the surface.";
    otherwise:
        now the noun is nowhere;
        say "[The noun] sinks out of sight."
```

Example

402. **Flotation**
Objects that can sink or float in a well, depending on their own properties and the state of the surrounding environment.

RB 10.2 Liquids

§19.10 Rulebook variables

When a rulebook is intended to perform some complicated task or calculation, it is sometimes useful for earlier rules to be able to leave information which will help later ones.

For instance, suppose we want a rulebook which is intended to print out the player's current aptitude. We will suppose that this is a number from 0 upwards: the higher, the

apter. The player gets bonus aptitude marks for achievements, but marks deducted for accidents, and so on. Moreover, we want to design this system so that it's easy to add further rules. The natural solution is to have a number which varies (or 'variable') acting as the running aptitude total: it should start at 0 and be altered up or down by subsequent rules. First, we should make the rulebook, and then add a variable:

```
Aptitude is a rulebook. The aptitude rulebook has a number called the aptitude mark.
```

The new value 'aptitude mark' is shared by the rules of the rulebook: nobody else can see it. It is created at the start of the rulebook being followed, and destroyed at the end. (If the rulebook should be followed a second time inside of itself, a new copy is created which does not disturb the old one.) So, in this case, 'aptitude mark' is started as 0 (since it is a number) each time the aptitude rules run. We can then write whatever rules we please to modify it:

```
An aptitude rule:
    if in darkness:
        decrease the aptitude mark by 3.

An aptitude rule:
    if we have taken the idol:
        increase the aptitude mark by 10.
```

And we had better do something with the result:

```
The last aptitude rule: say "Your aptitude rating is [aptitude mark]."
```

A rulebook can have any number of variables like this. They behave much like "let" values except that they last for a whole rulebook, not an individual rule or To phrase definition. (Well, strictly speaking they are accessible not just to the rules which belong to the rulebook, but also to any rules which previously belonged to the rulebook but were kicked out by means of an explicit rule-listing sentence. This is good because otherwise they will suddenly cause problem messages when unlisted.)

§19.11 Success and failure

Though we have blurred over this point so far, each rule must ordinarily end with one of three outcomes: success, failure and neither ("no outcome").

When a rulebook is followed, what happens is that each of its rules is followed in turn **until one of them ends in success or failure** - if ever: it is possible that each rule is tried and each ends with no outcome, so that the rulebook simply runs out of rules to try.

For some rulebooks, these are not useful ideas: "every turn" rules, for instance, by default never produce an outcome, which is why the "every turn" rulebook normally runs through all its rules at the end of each turn. (Use of the phrases below can change that, so it's best not to use them in "every turn" rules.) But for other rulebooks, such as "check taking", it's important that a rule which fails will stop the whole rulebook. For instance, we might find that the "can't take yourself rule" produces no outcome (because we aren't trying to do that), and then likewise the "can't take other people rule" (ditto) but that the "can't take component parts rule" prints up a complaint, and fails the action: the rulebook stops, and never goes on to (for instance) the "can't take scenery rule". This is good, because an impossible action often fails for several reasons at once, and we only want to print up one objection, not a whole list.

To follow the working of this mechanism, we need to be able to predict the outcome of any given rule. Sometimes this is easy to spot. For instance, in a rule which works on actions:

continue the action; *means* "end this rule with no outcome"
stop the action; *means* "end this rule in failure"
... instead; *means* "end this rule in failure"

("Success" and "failure" are technical terms here: they do not mean that the player has or hasn't got what he wanted.) This is why the rule:

```
Before taking something: say "The sentry won't let you!" instead.
```

ends in failure, and therefore stops the "before" rulebook. Another easy-to-spot case is when a rule makes use of the explicit phrases:

rule succeeds

This causes the current rule to end immediately, with its outcome considered to be a success. The rulebook being worked through also ends, and is also a success.

rule fails

This causes the current rule to end immediately, with its outcome considered to be a failure. The rulebook being worked through also ends, and is also a failure.

make no decision

This causes the current rule to end immediately, but with no outcome. That means the rulebook being worked through will continue to run on, beginning with the next rule.

But what happens if a rule simply doesn't say whether it succeeds, fails or has no outcome? In that case **it depends on the rulebook**. For almost all rulebooks, a rule which doesn't make a choice has no outcome, as in the following example:

```
Before taking something: say "The sentry looks at you anxiously!"
```

This rule, if it takes effect, ends with no outcome - so the action continues. But other rulebooks have a different convention: the most important is "instead", where a rule making no explicit choice is deemed to end in failure. For instance:

```
Instead of taking something: say "The sentry prods you with his rifle!"
```

This rule, if it takes effect, ends in failure and therefore stops the action.

We call this the **default outcome** of a rulebook. The default outcome of "before" (and of almost all rulebooks, in fact) is no outcome; the default outcome of "instead" is failure; the default outcome of "after" is success. The few exceptional cases with default outcome success or failure are marked as such in the Rules index.

When we create a rulebook, it will default to "no outcome". But we can specify otherwise with sentences like so:

```
The cosmic analysis rules are a rulebook. The cosmic analysis rules have default failure.
```

Finally, note that the default outcome for a rulebook is really the default outcome for any rule in that rulebook: if no rules in the rulebook ever apply, for instance if there aren't any and the rulebook is empty, then the rulebook ends with no outcome at all.

We can test the latest outcome like so:

if rule succeeded:

This condition is true if the most recently followed rule or rulebook ended in success. Example:

```
follow the hypothetical clever rule;
if rule succeeded:
    ...
```

if rule failed:

This condition is true if the most recently followed rule or rulebook ended in failure. Example:

```
follow the hypothetical clever rule;
if rule failed:
    ...
```

Note that this is not the opposite of "rule succeeded", because there's a third possibility: that it ended with no outcome.

Example

403. **Kyoto**
Expanding the effects of the THROW something AT something command so that objects do make contact with one another.
RB 10.4 Glass and Other Damage-Prone Substances

§19.12 Named outcomes

We have seen that the terms "success" and "failure" can be misleading - after all, it might be a good thing for a particular rulebook to "fail". At any rate, these are vague terms, and we don't want to have to remember the conventions used by every rulebook. This is why certain rulebooks have explicitly named outcomes instead. For instance, the "visibility" rules are allowed to have the outcomes:

```
there is sufficient light;
there is insufficient light;
```

These look like phrases, but are in fact named outcomes which can only be used in visibility rules. (They would make no sense elsewhere, and Inform will not allow their use if they are clearly out of context.) Such named outcomes are listed in the Rules index.

There can be any number of named outcomes. For instance, the Standard Rules define:

```
The does the player mean rules are a rulebook. The does the player mean rules have outcomes it is very likely, it is likely, it is possible, it is unlikely and it is very unlikely.
```

which makes five possible outcomes. Five outcomes seems to contradict the principle that there are only three possible outcomes for a rule: in fact, though, the five are counted as five different forms of "success", and any of them will cause a "does the player mean" rule to succeed. If we do not want this, we can instead specify explicitly how the named outcomes correspond to success, failure or "no outcome":

```
Visibility rules have outcomes there is sufficient light (failure) and there is insufficient light (success).
```

Again, see the Rules index for examples.

The same named outcome can be used for more than one rulebook, and can have different meanings in the context of different rulebooks - "good news" could be defined as success in one rulebook and failure in another, for instance. (This means that rulebook creators need not worry about name clashes and is an important difference in behaviour between rulebook outcomes and kinds of value.) We can even name a specific named outcome as the default outcome for rules in this rulebook:

```
Audibility rules have outcomes high background noise (failure), low background noise (success - the default) and absolute silence (success).
```

After a rulebook using named outcomes has run, we can test which outcome occurred by using the phrase:

outcome of the rulebook ⇒ *rulebook outcome*

This phrase produces the (named) outcome of the phrase most recently followed. Example:

```
follow the audibility rules;
if the outcome of the rulebook is the absolute silence outcome:
    say "You could hear a pin drop in here."
```

Each named outcome is a value if followed by the word "outcome", which is how "absolute silence" has become "the absolute silence outcome". Named outcomes can be said, so we could use the text substitution "[outcome of the rulebook]", for instance. A final caveat: it is perfectly legal to create a named outcome which means "no outcome", but if so then this will never be "the outcome of the rulebook" because "no outcome" is not an outcome.

Example

404. **Being Peter**
A set of rules determining the attitude a character will take when asked about certain topics.

RB 7.10 Character Emotion

§19.13 Rulebooks producing values

We have now seen two ways to write the outcome of a rule: as simple success or failure, with more or less explicit phrases like:

```
rule succeeds;
rule fails;
continue the action;
stop the action;
```

and by using a named outcome for the current rulebook as if it were a phrase, as in:

```
low background noise;
```

There is still a third way: we can stop a rule and at the same time produce a value. This isn't needed very often - none of the built-in rulebooks in the Standard Rules produces a value.

As we've seen, every rulebook has one kind of value as its basis, and it also has another kind of value for it to produce. If we call these K and L, then we have altogether four ways to write down the kind of a rulebook:

```
rulebook
K based rulebook
rulebook producing L
K based rulebook producing L
```

If we don't mention K, Inform assumes the rulebook is action based. If we don't mention L, Inform assumes L is "nothing", that is, Inform assumes no value is ever produced. Thus

```
Drum summons rules is a rulebook.
```

is equivalent to

```
Drum summons rules is an action based rulebook producing nothing.
```

But let's now look at a rulebook which does produce something.

```
The cat behavior rules is a rulebook producing an object.
```

This rulebook works out which thing the cat will destroy next. We might have rules like this one:

```
Cat behavior when Austin can see the ball of wool:
    rule succeeds with result the ball of wool.
```

The value is produced only when a rule succeeds, using this phrase:

rule succeeds with result (value)

This phrase can only be used in a rule which produces a value, and the value given must be of the right kind. It causes the current rule to finish immediately, to succeed, and to produce the value given.

How are we to use the cat behavior rulebook? If we write:

```
follow cat behavior
```

then the rulebook runs just as any other rulebook would, but the value produced is lost at the end, which defeats the point. Instead, we might write:

```
Every turn:
    let the destroyed object be the object produced by the cat behavior rules;
    if the destroyed object is not nothing:
        say "Austin pounces on [the destroyed object] in a flurry.";
        now the destroyed object is nowhere.
```

The key phrase here is

```
object produced by the cat behavior rules
```

which accesses the value this rulebook produces. In general, we write:

(name of kind) **produced by** (rule producing values) ⇒ *value*

This phrase is used to follow the named rule, and to collect the resulting value.

(name of kind) **produced by** (values based rule producing values) **for** (value) ⇒ *value*

This phrase is used to follow the named rule based on the value given, and to collect the resulting value.

Examples

405. **Feline Behavior**
    A cat which reacts to whatever items it has handy, returning the result of a rulebook for further processing.
    RB 8.3 Animals

406. **Tilt 2**
    A deck of cards with fully implemented individual cards; when the player has a full poker hand, the inventory listing describes the resulting hand accordingly.
    RB 9.5 Dice and Playing Cards

§19.14 Abide by

It often happens that one rule needs to invoke another one. Most of the time, the best way to do this is with "follow":

```
follow the magical mystery tour rule;
```

More often, though, we want not only to invoke another rule, but also to be guided by its advice. For this, we use the otherwise identical phrase:

abide by (rule)

This phrase applies the given rule, and makes its result the result of the present rule. If the rule being abided by succeeds or fails then the original rule also stops, at once and without going on to any further instructions. Example:

```
The omnibus rule:
    abide by the first rule;
    abide by the second rule;
    abide by the third rule;
    abide by the fourth rule.
```

This duplicates the effect of a rulebook of four rules: the "omnibus rule" tries each in turn, and stops as soon as any of them stop.

abide by (values based rule producing values) **for** (value)

This phrase applies the given rule to the given value, and makes its result the result of the present rule. If the rule being abided by succeeds or fails then the original rule also stops, at once and without going on to any further instructions.

Abide might be used in examples like this one:

```
A thing can be fragile or robust.
This is the can't handle fragile things roughly rule: if the noun is fragile, say "[The noun] is too fragile for such rough handling." instead.
A check dropping rule: abide by the can't handle fragile things roughly rule. A check throwing it at rule: abide by the can't handle fragile things roughly rule.
```

Had we used "follow" instead of "abide by", then in the event of the player typing "drop angel" the text "The glass angel is too fragile for such rough handling" would be printed, which is correct - but then the action would continue as though no difficulty had occurred, which is definitely not correct.

Finally, we can "anonymously abide":

anonymously abide by (rule)

or…

anonymously abide by (values based rule producing values) **for** (value)

This phrase applies the given rule, and makes its result the result of the present rule. If the rule being abided by succeeds or fails then the original rule also stops, at once and without going on to any further instructions. However, the rule deemed to have decided the outcome is the one abided by, not the one doing the abiding.

This is only useful in complicated situations where one rulebook uses another which… and so on. Its effect is exactly the same as "abide", except that the rule deemed to have decided the outcome is the one abided by, not the one doing the abiding. It thus allows a rule or rulebook to act purely as a middle-man, never getting the blame or the credit for what happens. The rule which made the decision is often not very relevant anyway, but it's used as the source of the value "reason the action failed" (see the Advanced Actions chapter).

§19.15 Two rulebooks used internally

Rulebooks handle almost all of the important tasks which an Inform work of IF must carry out in order to keep play going. We have seen them used in clarifying the player's command, supplying missing ingredients, processing the action to see what should happen, responding, and so on: by this point in the documentation, it must look as if Inform uses rulebooks for everything.

This is nearly true. There is not actually a super-rulebook controlling everything. (Such a super-rulebook would need to repeat itself and never finish, something a rulebook is not allowed to do.) Instead, what happens during play looks like so:

```
1. Following the "when play begins" rulebook.
2. Repeating:
    2(a). Reading and parsing a command into an action;
    2(b). Following the "action processing" rulebook;
    2(c). Following the "turn sequence" rulebook.
until the story has finished.
3. Following the "when play ends" rulebook.
```

The command parser occasionally calls on the services of activity rulebooks to help it, but otherwise gets on with its job in ways that we do not "see" as Inform 7 users. The rest

of what happens involves rulebooks, and in particular two important beneath-the-surface rulebooks: action processing and turn sequence.

The **action processing rules** are used whenever an action must be tried, by whoever tries it. This usually happens in response to player commands, but not always: it might happen because of a "try…", and it can certainly interrupt an existing action.

The **turn sequence rules** are used at the end of each turn, and include housekeeping as well as timekeeping. They consult the "every turn" rulebook, and advance the time of day, among other useful tasks.

In general, we should only modify the operation of these two crucial rulebooks as a last resort. Play can evidently fall to pieces if they cease to work normally.

Examples

407. **Electrified**
Adding a rule before the basic accessibility rule that will prevent the player from touching electrified objects under the wrong circumstances.
RB 10.7 Electricity and Magnetism

408. **Timeless**
A set of actions which do not take any game time at all.
RB 4.1 The Passage Of Time

409. **Endurance**
Giving different actions a range of durations using a time allotment rulebook.
RB 4.1 The Passage Of Time

410. **Escape from the Seraglio**
Replacing the usual response to TAKE ALL so that instead of output such as "grapes: Taken. orange: Taken.", Inform produces variable responses in place of "grapes:".
RB 6.15 Actions on Multiple Objects

§19.16 The Laws for Sorting Rulebooks

Large works created by Inform are heaped high with rules, most of them instead rules, but with a leavening of befores and afters as well. What will happen if these conflict with each other? For instance:

```
Instead of opening a container, say "Your mother-in-law looks on with such evident disappointment that you withdraw your hand again."

Instead of opening an open container, say "Your daughter tuts in theatrical exasperation at your, like, lameness."
```

Such clashes are resolved by sorting the rulebooks in order of specificity: thus your daughter gets in before your mother-in-law, because although both have rules hanging on the "opening" action, "an open container" is more specific than "a container". The full set of Laws used by Inform to sort rulebooks is quite elaborate. As we've seen, practical consequences can be investigated using the Rules index; and in most cases, the results are either natural (as above) or irrelevant (because the two rules being compared could not both activate at the same time anyway); but the full set of Laws is laid out below, for reference. It is probably a bad idea to write source text which absolutely relies on non-obvious rule sorting conventions, just the same, because this will make the source text harder to read and understand.

Sorting is done by comparing rules in pairs to decide which is more specific. We shall call these rules X and Y. The Laws are tried in sequence; the first Law to distinguish X and Y gets to decide which is more specific. If no Law is able to decide, X and Y go into the rulebook in order of their appearance in the source text - that is, whichever is defined first appears earlier in the rulebook and therefore takes priority.

Law I - Number of aspects constrained. For action-based rulebooks, rules are scored from 0 to 6 according to whether they constrain any of: (i) the exotic "going" clauses (pushing, by and through), (ii) the location of the action (in, from and to), (iii) the things directly involved (actor, noun, second noun, "nowhere" in the case of "going"), (iv) the presence of others (in the presence of…), (v) the time at which the action occurs (when, or "for the nth time" or "for the nth turn"), and/or (vi) the scene the action occurs in (during). For value based rulebooks, rules are scored from 0 to 3 according to whether they constrain: (i) the value parameter, (ii) the scene in which the rulebook is followed (when, during), and/or (iii) any condition which must hold or activities going on at the same time (when/while). A higher score is more specific than a lower one.

Law II - When/while requirement. A rule with a when/while restriction beats one without.

Law III - Action requirement. A rule with a more specific action requirement beats one with a more general action requirement. (Or similarly, for value based rulebooks, a rule with a more specific parameter requirement beats a more general one.) Details are given below.

Law IV - Scene requirement. A rule with a scene restriction ("during") beats one without.

Details of Law III now follow:

Law III.1 - Object To Which Rule Applies. For value based rulebooks only: the more specific value requirement wins.

Law III.2.1 - Action/Where/Going In Exotic Ways. A more specific combination of "…pushing…", "… by …", and "… through …" clauses in a "going" action beats a less specific. (Placing conditions on all three of these clauses beats placing conditions on any two, which beats any one, which beats none at all.) In cases where X and Y each place, let's say, two such conditions, they are considered in the order "…pushing…", "…by…" and then "…through…" until one wins. (The idea here is that pushing something from room to room is rarer than travelling in a vehicle, which in turn is rarer than going through a door. The rarer action goes first, as more specific.)

Law III.2.2 - Action/Where/Room Where Action Takes Place. A more specific combination of conditions on the room in which the action starts, and in which it ends, beats a less specific. For all actions other than "going", there is no combination to be considered, and what we do is to look at the specificity of the "… in …" clause. (So "Before looking in the Taj Mahal" beats "Before looking".)

For "going" actions, there are strictly speaking three possible room clauses: "… in …", "… from …" and "… to …". However, "… in …" and "… from …" cannot both be present, so that in practice a "going" rule constraining two rooms beats a "going" rule constraining only one.

If both the room gone from (the "…in…" or "…from…" room, whichever is given) and the room gone to (the "… to…" room) are constrained, then the constraints are looked at in the order from-room followed by to-room, since an action which goes to room Z could start in many different places and thus is likely to be more general.

Giving a place as a specific room beats giving only the name of a region; if region R is entirely within region S, then a rule applying in R beats a rule applying in S. (Note that regions can only overlap if one is contained in the other, so this does not lead to ambiguity.)

Law III.2.3 - Action/Where/In The Presence Of. A more specific "…in the presence of…" clause beats a less specific one. (This is again a constraint on where the action can take place, but it's now a potentially a constraint which could be passed in many different places at different times, so it's the most likely to be achieved and therefore the last to be considered of the Laws on Where.)

Law III.3.1 - Action/What/Second Thing Acted On. A more specific constraint on the second noun beats a less specific. Thus "putting something in the wooden box" beats

"putting something in a container".

Law III.3.2 - Action/What/Thing Acted On. A more specific constraint on the first noun beats a less specific. Thus "taking a container which is on a supporter" beats "taking a container".

In the case of "going" actions, the first noun is a direction. The special constraint "going nowhere" (which means: a direction in which the actor's location has no map connection) is considered more general than any other constraint placed on the first noun, but more specific than having no constraint at all. Thus "Instead of going north" beats "Instead of going nowhere" which beats "Instead of going".

Law III.3.3 - Action/What/Actor Performing Action. A more specific constraint on the actor beats a less specific.

Law III.4.1 - Action/How/What Happens. A more specific set of actions beats a less specific. For instance, "taking" beats "taking or dropping" beats "doing something other than looking" beats "doing something". A named kind of action (such as "behaving badly") is more specific than "doing something", but considered less specific than any explicitly spelled out list of actions.

Law III.5.1 - Action/When/Duration. An action with a constraint on its history ("for the fifth time", say, or "for the fifth turn") beats one without. If both rules place constraints on history, then the one occurring on the smaller number of possible turns wins (thus "for the third to seventh time" - 5 possible turns of applicability - beats "for less than the tenth turn" - 9 possible turns).

Law III.5.2 - Action/When/Circumstances. A more specific condition under "…when…" beats a less specific one. These conditions could potentially be complex: Inform judges how specific they are by counting the clauses found in them. The more clauses, the more specific the condition, it is assumed.

Law III.6.1 - Action/Name/Is This Named. A rule with a name ("the apple blossom rule", say) beats a rule without.

20. Advanced Text

§20.1 Changing texts
§20.2 Memory limitations
§20.3 Characters, words, punctuated words, unpunctuated words, lines, paragraphs
§20.4 Upper and lower case letters
§20.5 Matching and exactly matching
§20.6 Regular expression matching
§20.7 Making new text with text substitutions
§20.8 Replacements
§20.9 Summary of regular expression notation

§20.1 Changing texts

So far, we have dealt with text as something which comes in little packets: we have printed it out, read it in from the keyboard, and compared it with other text. But we have never tried to open the packets and get at the contents, letter by letter, or to make any alterations, or look for certain combinations of letters. These tricks are surprisingly seldom needed - a surprise, that is, given that everything Inform does is textual - but they are in fact open to us. For example:

```
if character number 1 in "[time of day]" is "1", ...
```

will be true at, for example, 11:30 PM and 1:22 AM, but not at 3:15 PM. What happens here is that Inform expands the time of day into a text, say "11:30 PM", then extracts the first character, say "1", and tests it.

Until 2012, Inform had two kinds of text - plain "text", and "indexed text" - but there's now only "text", which has all of the abilities of both.

§20.2 Memory limitations

Inform creates "story files" for very small virtual computers (capable of running on phones, for instance) where memory is tight. If we create a number variable and keep on adding 1 to it, the value simply gets bigger. But if we make some text and keep on adding a letter "x" to it, the text takes up more and more space, growing into longer and longer runs of "x"s until there is no more space to hold it.

The following warnings are rather like the tiny print about side-effects on medicine bottles: that is, we mostly ignore them, and if the drugs should kill us, well, at least we have the consolation of knowing we were warned. There are basically three limitations on text:

(1) An amount of memory has to be set aside for text (and other flexible-sized data), and Inform guesses the amount needed. Story files using the Glulx format (see the Settings panel) are able to increase this as necessary in play, so there's no problem if the guess was wrong. But Z-machine story files are stuck with whatever amount of memory was initially chosen.

That choice can be increased with a use option, like so:

```
Use dynamic memory allocation of at least 16384.
```

Inform raises its estimate of the amount needed to ensure that this amount is always at least its own guess, and also at least any amount declared like this. (And then it rounds up to the nearest power of 2, as it happens.) The default value of "dynamic memory allocation" is 8192. In practice, this use option isn't needed much, though, because any story needing large amounts of dynamic memory will likely be on Glulx in any case.

(2) Text has a maximum length. This maximum is normally 1000 characters, which ought to be plenty, but can be raised by sentences such as:

```
Use maximum text length of at least 2000.
```

What happens if this is broken, that is, if we try to use text overrunning this length? The Z-machine may simply crash, so if there is any chance that any single text may grow unpredictably large, Glulx should always be used. On Glulx, overrunning text is truncated safely, except that under Glulx 3.1.0 or better the story file will try to use dynamic memory allocation to expand the limit as needed to avoid truncation. (Testing shows that text is slow to manipulate once it grows beyond about 20,000 characters in length, but this is not really surprising.)

(3) Under the Z-machine, text may only contain characters from the so-called "ZSCII" character set - standard numbers, letters, punctuation marks and the commonest West European accented letters. Anything more exotic is likely to be flattened into a question mark "?". Under Glulx, any character can be used.

All of this makes the Z-machine sound very inferior, for text purposes. But note that Z can handle all of the examples in this chapter perfectly happily.

§20.3 Characters, words, punctuated words, unpunctuated words, lines, paragraphs

Inform can get at the contents of text in a variety of ways. The lowest-level is by character - a character is a letter, digit, punctuation symbol, space or other letter-form. (We use the

term "character" rather than "letter" because otherwise we would have to call "5" a letter, and so on.) Characters number upwards from 1: character number 1, to repeat that, starts the text. We can get the Nth character with:

character number (number) **in** (text) ⇒ *text*

This phrase produces the Nth character from the text, counting from 1. Characters include letters, digits, punctuation symbols, spaces or other letter-forms. Example:

```
character number 8 in "numberless projects of social reform"
```

produces "e". If the index is less than 1 or more than the length of the text, the result is an empty text, "".

The maximum character number varies with the current length of the text, and can be evaluated as:

number of characters in (text) ⇒ *number*

This phrase produces the number of characters from the text. Characters include letters, digits, punctuation symbols, spaces or other letter-forms. Examples:

```
number of characters in "War and Peace"
number of characters in ""
```

produce 13 and 0 respectively.

We can also use the adjective "empty":

```
if the description of the location is empty, ...
```

The empty text, "", is the only one with 0 characters.

We can also extract the contents by word, again numbered from 1. Thus:

word number (number) **in** (text) ⇒ *text*

This phrase produces the Nth word from the text, counting from 1. Words for this purpose are what's left after breaking the text up at punctuation or spacing (spaces, line breaks, paragraph breaks) and then removing that punctuation or spacing. Example:

```
word number 3 in "ice-hot, don't you think?"
```

produces "don't". If the index is less than 1 or more than the number of words in the text, the result is an empty text, "".

number of words in (text) ⇒ *number*

This phrase produces the number of words from the text. Words for this purpose are what's left after breaking the text up at punctuation or spacing (spaces, line breaks, paragraph breaks) and then removing that punctuation or spacing. Example:

```
number of words in "ice-hot, don't you think?"
```

produces 5.

Note that the contraction apostrophe in "don't" doesn't count as punctuation. Because this is not always quite what we want, Inform offers two variations:

punctuated word number (number) **in** (text) ⇒ *text*

This phrase produces the Nth word from the text, counting from 1. Words for this purpose are what's left after breaking the text up at punctuation or spacing (spaces, line breaks, paragraph breaks) and then removing the spacing, but leaving the punctuation as independent words. Example:

```
punctuated word number 2 in "ice-hot, don't you think?"
```

produces "-". The punctuated words here are "ice", "-", "hot", ",", "don't", "you", "think", "?". If two or more punctuation marks are adjacent, they are counted as different words, except for runs of dashes or periods: thus ",," has two punctuated words, but "--" and "..." have only one each. If the index is less than 1 or more than the number of punctuated words in the text, the result is an empty text, "".

number of punctuated words in (text) ⇒ *number*

This phrase produces the number of words from the text. Words for this purpose are what's left after breaking the text up at punctuation or spacing (spaces, line breaks, paragraph breaks) and then removing the spacing, but leaving the punctuation as independent words. Example:

```
number of punctuated words in "ice-hot, don't you think?"
```

produces 8; see if you can find them all.

unpunctuated word number (number) **in** (text) ⇒ *text*

This phrase produces the Nth word from the text, counting from 1. Words for this purpose are what's left after breaking the text up at spacing (spaces, line breaks, paragraph breaks) but including all punctuation as if it were part of the spelling of the words it joins to. Example:

```
unpunctuated word number 1 in "ice-hot, don't you think?"
```

produces "ice-hot,". The unpunctuated words in "ice-hot, don't you think?" are "ice-hot,", "don't", "you", "think?". If the index is less than 1 or more than the number of punctuated words in the text, the result is an empty text, "".

number of unpunctuated words in (text) ⇒ *number*

This phrase produces the number of words from the text. Words for this purpose are what's left after breaking the text up at spacing (spaces, line breaks, paragraph breaks) but including all punctuation as if it were part of the spelling of the words it joins to. Example:

```
number of unpunctuated words in "ice-hot, don't you think?"
```

produces just 4.

Finally, on the larger scale still, we also have:

line number (number) **in** (text) ⇒ *text*

This phrase produces the Nth line from the text, counting from 1. Unless explicit use is made of line-breaking, lines and paragraphs will be the same - it doesn't refer to lines as visible on screen, because we have no way of knowing what size screen the player might have.

number of lines in (text) ⇒ *number*

This phrase produces the number of lines in the text. Unless explicit use is made of line-breaking, lines and paragraphs will be the same - it doesn't refer to lines as visible on screen, because we have no way of knowing what size screen the player might have. Example: the number of lines in

```
"Sensational news just in![paragraph break]The Martians have invaded
Miranda.[line break](One of the moons of Uranus, that is.)"
```

is 3.

paragraph number (number) **in** (text) ⇒ *text*

This phrase produces the Nth paragraph from the text, counting from 1.

number of paragraphs in (text) ⇒ *number*

This phrase produces the number of paragraphs in the text. Example: the number of paragraphs in

```
"Sensational news just in![paragraph break]The Martians have invaded
Miranda.[line break](One of the moons of Uranus, that is.)"
```

is 2.

(Attempting to make large enough texts to have a serious paragraph count is slightly risky if there is not much memory to play with, as on the Z-machine. But the facilities do exist.)

§20.4 Upper and lower case letters

In most European languages the same letters can appear in two forms: as capitals, like "X", mainly used to mark a name or the start of a sentence; or in their ordinary less prominent form, like "x". These forms are called upper and lower case because, historically, typesetters kept lead castings of letters in two wooden cases, one above the other on the workbench. Lower case letters were in the lower box closer to hand, being more often needed.

Human languages are complicated. Not every lower case letter has an upper case partner: ordinal markers in Hispanic languages don't, for instance, and the German "ß" is never used in upper case. Sometimes two different lower case letters have the same upper case form: "ς" and "σ", two versions of the Greek sigma, both capitalise to "Σ". Inform follows the international Unicode standard in coping with all this.

We can test whether text is in either case like so:

if (text) **is in lower case:**

This condition is true if every character in the text is a lower case letter. Examples: this is true for "wax", but false for "wax seal" or "eZ mOnEy".

if (text) **is in upper case:**

This condition is true if every character in the text is in upper case. Examples: this is true for "BEESWAX", but false for "ROOM 101".

We can change the casing of text using:

(text) **in lower case** ⇒ *text*

This phrase produces a new version of the given text, but with all upper case letters reduced to lower case. Example: "a ticket to Tromsø via Østfold" becomes

```
"a ticket to tromsø via østfold"
```

(text) **in upper case** ⇒ *text*

This phrase produces a new version of the given text, but with all upper case letters reduced to lower case. Example: "a ticket to Tromsø via Østfold" becomes

```
"A TICKET TO TROMSØ VIA ØSTFOLD"
```

(text) **in title case** ⇒ *text*

This phrase produces a new version of the given text, but with casing of words changed to title casing: this capitalises the first letter of each word, and lowers the rest. Example: "a ticket to Tromsø via Østfold" becomes

```
"A Ticket To Tromsø Via Østfold"
```

(text) **in sentence case** ⇒ *text*

This phrase produces a new version of the given text, but with casing of words changed to sentence casing: this capitalises the first letter of each sentence and reduces the rest to lower case. Example: "a ticket to Tromsø via Østfold" becomes

```
"A ticket to tromsø via østfold"
```

Accents are preserved in case changes. So (if we are using Glulx and have Unicode available) title case can turn Aristophanes' discomfortingly lower-case lines

```
ἐξ οὗ γὰρ ἡμᾶς προὔδοσαν μιλήσιοι,
οὐκ εἶδον οὐδ᾽ ὄλισβον ὀκτωδάκτυλον,
ὃς ἦν ἂν ἡμῖν σκυτίνη "πικουρία
```

by raising them proudly up like so:

```
Ἐξ Οὗ Γὰρ Ἡμᾶς Προὔδοσαν Μιλήσιοι,
Οὐκ Εἶδον Οὐδ᾽ Ὄλισβον Ὀκτωδάκτυλον,
Ὃς Ἦν Ἂν Ἡμῖν Σκυτίνη "Πικουρία.
```

Title and sentence casing can only be approximate if done by computer. Inform looks at the letters, but is blind to the words and sentences they make up. (Note the way sentence casing did not realise "Tromsø" and "Østfold" were proper nouns.) If asked to put the name "MCKAY" into title casing, Inform will opt for "Mckay", not recognising this as the Scottish patronymic surname "McKay". Given "baym dnieper", the title of David Bergelson's great Yiddish novel of 1932, it will opt for "BAYM DNIEPER": but properly speaking Yiddish does not have upper case lettering at all, though nowadays it is sometimes printed as if it did. And conventions are very variable about which words should be capitalised in titles: English publishers mostly agree that connectives, articles

and prepositions should be in lower case, but in France almost anything goes, with Académie Française rules giving way to avant-garde book design. In short, we cannot rely on Inform's title casing to produce a result which a human reader will always think perfect.

This discussion has all been about how Inform prints, not about how it reads commands from the keyboard, because the latter is done case-insensitively. The virtual machines for which Inform creates programs normally flatten all command input to lower case, and in any case Understand comparison ignores casing. Thus

```
Understand "mckay" as the Highland Piper.
```

means that "examine McKay", "examine MCKAY", "examine mckay", and so forth are all equivalent. The text of the player's command probably doesn't preserve the original casing typed in any event.

One more caution, though it will affect hardly anyone. For projects using the Z-machine, only a restricted character set is available in texts: for more, we must use Glulx. A mad anomaly of ZSCII, the Z-machine character set, is that it contains the lower case letter "ÿ" but not its upper case form "Ÿ", so that

```
"ÿ" in upper case
```

produces "Ÿ" in Glulx but "ÿ" in the Z-machine. This will come as a blow to Queensrÿche fans, but in all other respects any result on the Z-machine should agree with its counterpart on Glulx.

Examples

411. **Capital City**
To arrange that the location information normally given on the left-hand side of the status line appears in block capitals.
RB 12.2 The Status Line

412. **Rocket Man**
Using case changes on any text produced by a "to say…" phrase.
RB 2.1 Varying What Is Written

§20.5 Matching and exactly matching

Up to now, we have only been able to judge two texts by seeing if they are equal, but we can now ask more subtle questions.

if (text) **matches the text** (text):

This condition is true if the second text occurs anywhere inside the first. Examples:

```
if "[score]" matches the text "3", ...
```

tests whether the digit 3 occurs anywhere in the score, as printed out; and

```
if the printed name of the location matches the text "the", ...
```

tests to see whether "the" can be found anywhere in the current room's name. Note that the location "Smotheringly Hot Jungle" would pass this test - it's there if you look. On the other hand, "The Orangery" would not, because "The" does not match against "the". We can get around this in a variety of ways, one of which is to tell Inform to be insensitive to the case (upper or lower) of letters:

```
if the printed name of the location matches the text "the", case insensitively: ...
```

if (text) **exactly matches the text** (text):

This condition is true if the second text matches the first, starting at the beginning and finishing at the end. This appears to be the same as testing if one is equal to the other, but that's not quite true: for example,

```
if "[score]" exactly matches the text "[best score]", ...
```

is true if the score and best score currently print out as the same text, which will be true if they are currently equal as numbers; but

```
if "[score]" is "[best score]", ...
```

is never true - these are different texts, even if they sometimes look the same.

In the next section we shall see that "matches" and "exactly matches" can do much more than the simple text matching demonstrated above.

We can also see how many times something matches:

number of times (text) **matches the text** (text) ⇒ *number*

This produces the number of times the second text occurs within the first. The matches are not allowed to overlap. Example:

```
number of times "pell-mell sally" matches the text "ll" = 3
number of times "xyzzy" matches the text "Z" = 0
number of times "xyzzy" matches the text "Z", case insensitively = 2
number of times "aaaaaaaa" matches the text "aaaa" = 2
```

There's no "number of times WHATEVER exactly matches the text FIND" phrase since this is by definition going to have to be 0 or 1.

§20.6 Regular expression matching

When playing around with text, we tend to get into longer and trickier wrangles of matching - we find that we want to look not for simple text like "gold", but for "gold" used only as a separate word, or for a date in YYYY-MM-DD format, or for a seemingly endless range of other possibilities. What we need is not just for Inform to provide a highly flexible matching program, but also a good notation in which to describe what we want.

Fortunately, such a notation already exists. This is the "regular expression" notation, named for a 1950s mathematical model by the logician Stephen Kleene, applied to computing in the late 60s by Ken Thompson, borrowed almost at once by the early Unix tools of the 70s, and developed further by Henry Spencer in the 80s and Philip Hazel in the 90s. The glue holding the Internet together - the Apache web-server, the scripting languages Perl and Python, and so forth - makes indispensable use of regular expressions.

As might be expected from the previous section, we simply have to describe the FIND text as "regular expression" rather than "text" and then the same facilities are available:

if (text) **matches the regular expression** (text):

This condition is true if any contiguous part of the text can be matched against the given regular expression. Examples:

```
if "taramasalata" matches the regular expression "a.*l", ...
```

is true, since this looks for a part of "taramasalata" which begins with "a", continues with any number of characters, and finishes with "l"; so it matches "aramasal". (Not "asal", because it gets the makes the leftmost match it can.) The option "case insensitively" causes lower and upper case letters to be treated as equivalent.

if (text) **exactly matches the regular expression** (text):

This condition is true if the whole text (starting from the beginning and finishing at the end) can be matched against the given regular expression. The option "case insensitively" causes lower and upper case letters to be treated as equivalent.

And once again:

number of times (text) **matches the regular expression** (text) ⇒ *number*

This produces the number of times that contiguous pieces of the text can be matched against the regular expression, without allowing them to overlap.

Since a regular expression can match quite a variety of possibilities (for instance "b\w+t" could match "boast", "boat", "bonnet" and so on), it's sometimes useful to find what the match actually was:

text matching regular expression ⇒ *text*

This phrase is only meaningful immediately after a successful match of a regular expression against text, and it produces the text which matched. Example:

```
if "taramasalata" matches the regular expression "m.*l":
    say "[text matching regular expression].";
```

says "masal."

Perhaps fairly, perhaps not, regular expressions have a reputation for being inscrutable. The basic idea is that although alphanumeric characters (letters, numbers and spaces) mean just what they look like, punctuation characters are commands with sometimes dramatic effects. Thus:

```
if WHATEVER matches the regular expression "fish", ...
if WHATEVER matches the regular expression "f.*h", ...
```

behave very differently. The first is just like matching the text "fish", but the second matches on any sequence of characters starting with an "f" and ending with an "h". This is not at all obvious at first sight: reading regular expressions is a skill which must be learned, like reading a musical score. A really complex regular expression can look like a soup of punctuation and even an expert will blink for a few minutes before telling you what it does - but a beginner can pick up the basics very quickly. Newcomers might like to try out and become comfortable with the features a few at a time, reading down the following list.

1. Golden rule. Don't try to remember all the characters with weird effects. Instead, if you actually mean any symbol other than a letter, digit or space to be taken literally, place a backslash "\" in front of it. For instance, matching the regular expression

```
"\*A\* of the Galactic Patrol"
```

is the same as matching the text "*A* of the Galactic Patrol", because the asterisks are robbed of their normal powers. This includes backslash itself: "\\" means a literal backslash. (Don't backslash letters or digits - that turns out to have a meaning all its own, but anyway, there is never any need.)

2. Alternatives. The vertical stroke "|" - not a letter I or L, nor the digit 1 - divides alternatives. Thus

```
"the fish|fowl|crawling thing"
```

is the same as saying match "the fish", or "fowl", or "crawling thing".

3. Dividing with brackets. Round brackets "(" and ")" group parts of the expression together.

```
"the (fish|fowl|crawling thing) in question"
```

is the same as saying match "the fish in question", or "the fowl in question", or "the crawling thing in question". Note that the "|" ranges outwards only as far as the group it is in.

4. Any character. The period "." means any single character. So

```
"a...z"
```

matches on any sequence of five characters so long as the first is "a" and the last is "z".

5. Character alternatives. The angle brackets "<" and ">" are a more concise way of specifying alternatives for a single character. Thus

```
"b<aeiou>b"
```

matches on "bab", "beb", "bib", "bob" or "bub", but not "baob" or "beeb" - any single character within the angle brackets is accepted. Beginning the range with "^" means "any single character so long as it is not one of these": thus

```
"b<^aeiou>b"
```

matches on "blb" but not "bab", "beb", etc., nor on "blob" or "bb". Because long runs like this can be a little tiresome, we are also allowed to use "-" to indicate whole ranges. Thus

```
"b<a-z>b"
```

matches a "b", then any lower case English letter, then another "b".

In traditional regular expression language, square brackets rather than angle brackets are used for character ranges. In fact Inform does understand this notation if there are actual square brackets "[" and "]" in the pattern text, but in practice this would be tiresome to achieve, since Inform uses those to achieve text substitutions. So Inform allows "b<a-

z>b" rather than making us type something like

```
"b[bracket]a-z[close bracket]b"
```

to create the text "b[a-z]b".

6. Popular character ranges. The range "<0-9>", matching any decimal digit, is needed so often that it has an abbreviation: "\d". Thus

```
"\d\d\d\d-\d\d-\d\d"
```

matches, say, "2006-12-03". Similarly, "\s" means "any spacing character" - a space, tab or line break. "\p" is a punctuation character, in the same sense used for word division in the previous section: it actually matches any of

```
. , ! ? - / " : ; ( ) [ ] { }
```

"\w" means "any character appearing in a word", and Inform defines it as anything not matching "\s" or "\p".

"\l" and "\u" match lower and upper case letters, respectively. These are much stronger than "<a-z>" and "<A-Z>", since they use the complete definition in the Unicode 4.0.0 standard, so that letter-forms from all languages are catered for: for example "δ" matches "\l" and "Δ" matches "\u".

The reverse of these is achieved by capitalising the letter. So "\D" means "anything not a digit", "\P" means "anything not punctuation", "\W" means "anything not a word character", "\L" means "anything not a lower case letter" and so on.

7. Positional restrictions. The notation "^" does not match anything, as such, but instead requires that we be positioned at the start of the text. Thus

```
"^fish"
```

matches only "fish" at the start of the text, not occurring anywhere later on. Similarly, "$" requires that the position be the end of the text. So

```
"fish$"
```

matches only if the last four characters are "fish". Matching "^fish$" is the same thing as what Inform calls exactly matching "fish".

Another useful notation is "\b", which matches a word boundary: that is, it matches no actual text, but requires the position to be a junction between a word character and a non-word character (a "\w" and a "\W") or vice versa. Thus

```
"\bfish\b"
```

matches "fish" in "some fish" and also "some fish, please!", but not in "shellfish". (The regular expression "\w*fish\b" catches all words ending in "fish", as we will see below.) As usual, the capitalised version "\B" negates this, and means "not at a word boundary".

8. **Line break and tab**. The notations "\n" and "\t" are used for a line break ("n" for "new line") and tab, respectively. Tabs normally do not occur in Inform strings, but can do when reading from files. It makes no sense to reverse these, so "\N" and "\T" produce errors.

9. **Repetition**. Placing a number in braces "{" and "}" after something says that it should be repeated that many times. Thus

```
"ax{25}"
```

matches only on "axxxxxxxxxxxxxxxxxxxxxxxxx". More usefully, perhaps, we can specify a range of the number of repetitions:

```
"ax{2,6}"
```

matches only on "axx", "axxx", "axxxx", "axxxxx", "axxxxxx". And we can leave the top end open: "ax{2,}" means "a" followed by at least two "x"s.

Note that the braces attach only to most recent thing - so "ax{2}" means "a" followed by two of "x" - but, as always, we can use grouping brackets to change that. So "(ax){2,}" matches "axax", "axaxax", "axaxaxax",…

(It's probably best not to use Inform to try to match the human genome against "<acgt>{3000000000}", but one of the most important practical uses of regular expression matching in science is in treating DNA as a string of nucleotides represented by the letters "a", "c", "g", "t", and looking for patterns.)

10. Popular repetitions. Three cases are so often needed that they have standard short forms:

"{0,1}", which means 0 or 1 repetition of something - in other words, doesn't so much repeat it as make it optional - is written "?". Thus "ax?y" matches only on "ay" or "axy".

"{0,}", which means 0 or more repetitions - in other words, any number at all - is written "*". Thus "ax*y" matches on "ay", "axy", "axxy", "axxxy", ... and the omnivorous ".*" - which means "anything, any number of times" - matches absolutely every text. (Perhaps unexpectedly, replacing ".*" in a text with "X" will produce "XX", not "X", because the ".*" first matches the text, then matches the empty gap at the end. To match the entire text just once, try "^.*$".)

"{1,}", which means 1 or more repetitions, is written "+". So "\d+" matches any run of digits, for instance.

11. Greedy vs lazy. Once we allow things to repeat an unknown number of times, we run into an ambiguity. Sure, "\d+" matches the text "16339b". But does it look only as far as the "1", then reason that it now has one or more digits in a row, and stop? Or does it run onward devouring digits until it can do so no longer, so matching the "16339" part? These two strategies are called "lazy" and "greedy" respectively.

Do we care? Well, the strategy used makes no difference to whether there is a match, but it does affect what part of the text is matched, and the number of matches there are. Unless we mark for it, all repetitions are greedy. Usually this is good, but it means that, for instance,

```
"-.+-"
```

applied to "-alpha- -beta- -gamma-" will match the whole text, because ".+" picks up all of "alpha- -beta- -gamma". To get around this, we can mark any of the repetition operators as lazy by adding a question mark "?". Thus:

```
"-.+?-"
```

applied to "-alpha- -beta- -gamma-" matches three times, producing "-alpha-" then "-beta-" then "-gamma-".

A logical but sometimes confusing consequence is that a doubled question mark "??" means "repeat 0 or 1 times, but prefer 0 matches to 1 if both are possibilities": whereas a single question mark "?", being greedy, means "repeat 0 or 1 times, but prefer 1 match to 0 if both are possibilities".

12. Numbered groups. We have already seen that round brackets are useful to clump together parts of the regular expression - to choose within them, or repeat them. In fact, Inform numbers these from 1 upwards as they are used from left to right, and we can subsequently refer back to their contents with the notation "\1", "\2", ... After a successful match, we can find the results of these subexpressions with:

text matching subexpression (number) ⇒ *text*

This phrase is only meaningful immediately after a successful match of a regular expression against text, and it produces the text which matched. The number must be from 1 to 9, and must correspond to one of the bracketed groups in the expression just matched. Example: after

```
if "taramasalata" matches the regular expression "a(r.*l)a(.)":
```

the "text matching regular expression" is "aramasalat", the "text matching subexpression 1" is "ramasal", and "text matching subexpression 2" is "t".

For instance:

```
"(\w)\w*\1"
```

matches any run of two or more word-characters, subject to the restriction that the last one has to be the same as the first - so it matches "xerox" but not "alphabet". When Inform matches this against "xerox", first it matches the initial "x" against the group "(\w)". It then matches "\w*" ("any number of word-characters") against "ero", so that the "*" runs up to 3 repetitions. It then matches "\1" against the final "x", because "\1" requires it to match against whatever last matched in sub-expression 1 - which was an "x".

Numbered groups allow wicked tricks in matching, it's true, but really come into their own when it comes to replacing - as we shall see.

13. Switching case sensitivity on and off. The special notations "(?i)" and "(?-i)" switch sensitivity to upper vs. lower case off and on, mid-expression. Thus "a(?i)bcd(?-i)e" matches "abcde", "aBcDe", etc., but not "Abcde" or "abcdE".

14. Groups with special meanings. This is the last of the special syntaxes: but it's a doozy. A round-bracketed group can be marked to behave in a special way by following the open bracket by a symbol with a special meaning. Groups like this have no number and are not counted as part of \1, \2, and so forth - they are intended not to gather up material but to have some effect of their own.

```
"(# ...)"
```

Is a comment, that is, causes the group to do nothing and match against anything.

```
"(?= ...)"
```

Is a lookahead: it is a form of positional requirement, like "\b" or "^", but one which requires that the text ahead of us matches whatever is in the brackets. (It doesn't consume that text - only checks to see that it's there.) For instance "\w+(?=;)" matches a word followed by a semicolon, but does not match the semicolon itself.

```
"(?! ...)"
```

Is the same but negated: it requires that the text ahead of us does not match the material given. For instance, "a+(?!z)" matches any run of "a"s not followed by a "z".

```
"(?<= ...)" and "(?<! ...)"
```

Are the same but looking behind (hence the "<"), not forward. These are restricted to cases where Inform can determine that the material to be matched has a definite known width. For instance, "(?<!shell)fish" matches any "fish" not occurring in "shellfish".

```
"(> ...)"
```

Is a possessive, that is, causes the material to be matched and, once matched, never lets go. No matter what subsequently turns out to be convenient, it will never change its match. For instance, "\d+8" matches against "768" because Inform realises that "\d+" cannot be allowed to eat the "8" if there is to be a match, and stops it. But "(>\d+)8" does not match against "768" because now the "\d+", which initially eats "768", is possessive and refuses to give up the "8" once taken.

```
"(?(1)...)" and "(?(1)...|...)"
```

Are conditionals. These require us to match the material given if \1 has successfully matched already; in the second version, the material after the "|" must be matched if \1 has not successfully matched yet. And the same for 2, 3, …, 9, of course.

Finally, conditionals can also use lookaheads or lookbehinds as their conditions. So for instance:

```
"(?(?=\d)\d\d\d\d|AY-\d\d\d\d)"
```

means if you start with a digit, match four digits; otherwise match "AY-" followed by four digits. There are easier ways to do this, of course, but the really juicy uses of conditionals

are only borderline legible and make poor examples - perhaps this is telling us something.

Examples

413. **Alpha**
Creating a beta-testing command that matches any line starting with punctuation.
RB 13.1 Testing

414. **About Inform's regular expression support**
Some footnotes on Inform's regular expressions, and how they compare to those of other programming languages.
RB 1.4 Information Only

§20.7 Making new text with text substitutions

Substitutions are most often used just for printing, like so:

```
say "The clock reads [time of day].";
```

But they can also produce text which can be stored up or used in other ways. For example, defining

```
To decide what text is (T - text) doubled:
    decide on "[T][T]".
```

makes

```
let the Gerard Kenny reference be "NewYork" doubled;
```

set this temporary variable to "NewYorkNewYork".

There is, however, a subtlety here. A text with a substitution in it, like:

```
"The clock reads [time of day]."
```

is always waiting to be substituted, that is, to become something like:

```
"The clock reads 11:12 AM."
```

If all we do with text is to print it, there's nothing to worry about. But if we're storing it up, especially for multiple turns, there are ambiguities. For example, suppose we're changing

the look of the black status line bar at the top of the text window:

```
now the left hand status line is "[time of day]";
```

Just copying "[time of day]" to the "left hand status line" variable doesn't make it substitute - which is just as well, or the top of the screen would perpetually show "9:00 AM".

On the other hand, looking back at the phrase example:

```
To decide what text is (T - text) doubled:
    decide on "[T][T]".
```

"[T][T]" is substituted immediately it's formed. That's also a good thing, because "T" loses its meaning the moment the phrase finishes, which would make "[T][T]" meaningless anywhere else.

What's going on here is this: Inform substitutes text immediately if it contains references to a temporary value such as "T", and otherwise only if it needs to access the contents. This is why "[time of day]" isn't substituted until we need to print it out (or, say, access the third character): "time of day" is a value which always exists, not a temporary one.

Another case where that might be important is if we want to set a text to an elaborated version of itself. For example, suppose there is a variable (not a temporary one) called "the accumulated tally", and consider this:

```
now the accumulated tally is "[the accumulated tally]X";
```

The intention of the writer here was to add an "X" each time this happens. But the result is a hang, because what it actually means is that accumulated tally can only be printed if the accumulated tally is printed first… an infinite regress. The safe way to do this would be:

```
now the accumulated tally is the substituted form of "[the accumulated tally]X";
```

Using the adjectives "substituted" and "unsubstituted", it's always possible to test whether a given text is in either state, should this ever be useful. For example,

```
now the left hand status line is "[time of day]";
if the left hand status line is unsubstituted, say "Yes!";
```

will say "Yes!": the LHSL is like a bomb waiting to go off. Speaking of which:

```
The player is holding a temporal bomb.

When play begins:
    now the left hand status line is "Clock reads: [time of day]".

After dropping the temporal bomb:
    now the left hand status line is the substituted form of the left hand status line;
    say "Time itself is now broken. Well done."
```

This is making use of:

substituted form of (text) ⇒ *text*

This takes a text and makes substitution occur immediately. For example,

```
substituted form of "time of death, [time of day]"
```

produces something like "time of death, 9:15 AM" rather than "time of death, [time of day]". It's entirely legal to apply this to text which never had any substitutions in, so

```
substituted form of "balloon"
```

produces "balloon".

Note that there's no analogous phrase for "unsubstituted form of...", because once text has substituted, there's no way to go back.

Examples

415. **Identity Theft**
Allowing the player to enter a name to be used for the player character during the game.
RB 5.2 Traits Determined By the Player

416. **Mirror, Mirror**
The sorcerer's mirror can, when held up high, form an impression of its surroundings which it then preserves.
RB 9.12 Cameras and Recording Devices

417. **The Cow Exonerated**
Creating a class of matches that burn for a time and then go out, with elegant reporting when several matches go out at once.
RB 10.8 Fire

§20.8 Replacements

Suppose V is a text which varies - perhaps a property of something, or a variable defined everywhere, or a temporary "let"-named value. How do we change its contents? The easiest way is simply to assign text to it. Thus:

```
let V be "It is now [the time of the day in words]."
```

And, for instance,

```
let V be "[V]!"
```

adds an exclamation mark at the end of V.

Otherwise, it is more useful (also a little faster) to modify V by changing its characters, words and so on. Thus:

replace character number (number) **in** (text) **with** (text)

This phrase acts on the named text by placing the given text in place of the Nth character, counting from 1. Example:

```
let V be "mope";
replace character number 3 in V with "lecul";
say V;
```

says "molecule".

replace word number (number) **in** (text) **with** (text)

This phrase acts on the named text by placing the given text in place of the Nth word, counting from 1, and dividing words at spacing or punctuation. Example:

```
let V be "Does the well run dry?";
replace word number 3 in V with "jogger";
say V;
```

says "Does the jogger run dry?".

replace punctuated word number (number) **in** (text) **with** (text)

This phrase acts on the named text by placing the given text in place of the Nth word, counting from 1, and dividing words at spacing, counting punctuation runs as words in their own right. Example:

```
let V be "Frankly, yes, I agree.";
replace punctuated word number 2 in V with ":";
say V;
```

says "Frankly: yes, I agree.".

replace unpunctuated word number (number) **in** (text) **with** (text)

This phrase acts on the named text by placing the given text in place of the Nth word, counting from 1, and dividing words at spacing, counting punctuation as part of a word just as if it were lettering. Example:

```
let V be "Frankly, yes, I agree.";
replace unpunctuated word number 2 in V with "of course";
say V;
```

says "Frankly, of course I agree.".

replace line number (number) **in** (text) **with** (text)

This phrase acts on the named text by placing the given text in place of the Nth line, counting from 1. Lines are divided by paragraph or line breaks.

replace paragraph number (number) **in** (text) **with** (text)

This phrase acts on the named text by placing the given text in place of the Nth paragraph, counting from 1.

Last, but not least, we can replace text wherever it occurs:

replace the text (text) **in** (text) **with** (text)

This phrase acts on the named text by searching and replacing, as many non-overlapping times as possible. Example:

```
replace the text "a" in V with "z"
```

changes every lower-case "a" to "z": the same thing done with the "case insensitively" option would change each "a" or "A" to "z".

All very well for letters, but it can be unfortunate to try

```
replace the text "Bob" in V with "Robert"
```

if V happens to contain, say "The Olympic Bobsleigh Team": it would become "The Olympic Robertsleigh Team". What we want, of course, is for Bob to become Robert only when it's a whole word. We can get that with:

replace the word (text) **in** (text) **with** (text)

This phrase acts on the named text by searching and replacing, as many non-overlapping times as possible, where the search text must occur as a whole word. Example:

```
replace the word "Bob" in V with "Robert"
```

changes "Bob got on the Bobsleigh" to "Robert got on the Bobsleigh".

replace the punctuated word (text) **in** (text) **with** (text)

This phrase acts on the named text by searching and replacing, as many non-overlapping times as possible, where the search text must occur as a whole word or run of punctuation.

But these are all just special cases of the grand-daddy of all replacement phrases:

replace the regular expression (text) **in** (text) **with** (text)

This phrase acts on the named text by matching the regular expression and replacing anything which fits it, as many non-overlapping times as possible. Example:

```
replace the regular expression "\d+" in V with "..."
```

changes "The Battle of Waterloo, 1815, rivalled Trafalgar, 1805" to "The Battle of Waterloo, …, rivalled Trafalgar, …". The "case insensitively" causes lower and upper case letters to be treated as if the same letter.

When replacing a regular expression, the replacement text also has a few special meanings (though, thankfully, many fewer than for the expression itself). Once again "\n" and "\t" can be used for line break and tab characters, and "\\" must be used for an actual backslash. But, very usefully, "\1" to "\9" expand as the contents of groups numbered 1 to 9, and "\0" to the exact text matched. So:

```
replace the regular expression "\d+" in V with "roughly \0"
```

adds the word "roughly" in front of any run of digits in V, because \0 becomes in turn whichever run of digits matched. And

```
replace the regular expression "(\w+) (.*)" in V with "\2, \1"
```

performs the transformation "Frank Booth" to "Booth, Frank".

Finally, prefixing the number by "l" or "u" forces the text it represents into lower or upper case, respectively. For instance:

```
replace the regular expression "\b(\w)(\w*)" in X with "\u1\l2";
```

changes the casing of X to "title casing", where each individual word is capitalised. (This is a little slow on large texts, since so many matches and replacements are made: it's more efficient to use the official phrases for changing case.)

Examples

418. **Blackout**
Filtering the names of rooms printed while in darkness.
RB 2.1 Varying What Is Written

419. **Fido**
A dog the player can name and un-name at will.
RB 8.3 Animals

420. **Igpay Atinlay**
A pig Latin filter for the player's commands.
RB 2.3 Using the Player's Input

421. **Mr. Burns' Repast**
Letting the player guess types for an unidentifiable fish.
RB 2.3 Using the Player's Input

422. **Northstar**
Making Inform understand ASK JOSH TO TAKE INVENTORY as JOSH, TAKE INVENTORY. This requires us to use a regular expression on the player's command, replacing some of the content.
RB 7.14 Obedient Characters

423. **Cave-troll**
Determining that the command the player typed is invalid, editing it, and re-examining it to see whether it now reads correctly.
RB 6.17 Clarification and Correction

§20.9 Summary of regular expression notation

MATCHING

Positional restrictions

^	Matches (accepting no text) only at the start of the text
$	Matches (accepting no text) only at the end of the text
\b	Word boundary: matches at either end of text or between a \w and a \W
\B	Matches anywhere where \b does not match

Backslashed character classes

\char	If char is other than a-z, A-Z, 0-9 or space, matches that literal char
\\	For example, this matches literal backslash "\"
\n	Matches literal line break character
\t	Matches literal tab character (but use this only with external files)

\d	Matches any single digit
\l	Matches any lower case letter (by Unicode 4.0.0 definition)
\p	Matches any single punctuation mark: . , ! ? - / " : ; ( ) [ ] { } \sMatches any single spacing character (space, line break, tab) \uMatches any upper case letter (by Unicode 4.0.0 definition) \wMatches any single word character (neither \p nor \s) \DMatches any single non-digit \LMatches any non-lower-case-letter \PMatches any single non-punctuation-mark \SMatches any single non-spacing-character \UMatches any non-upper-case-letter \WMatches any single non-word-character (i.e., matches either \p or \s)

Other character classes

.	Matches any single character
<...>	Character range: matches any single character inside
<^...>	Negated character range: matches any single character not inside

Inside a character range

e-h	Any character in the run "e" to "h" inclusive (and so on for other runs)
>...	Starting with ">" means that a literal close angle bracket is included
\	Backslash has the same meaning as for backslashed character classes: see above

Structural

<table>
<tr><td>|</td><td>Divides alternatives: "fish|fowl" matches either</td></tr>
<tr><td>(?i)</td><td>Always matches: switches to case-insensitive matching from here on</td></tr>
<tr><td>(?-i)</td><td>Always matches: switches to case-sensitive matching from here on</td></tr>
</table>

Repetitions

...?	Matches "..." either 0 or 1 times, i.e., makes "..." optional
...*	Matches "..." 0 or more times: e.g. "\s*" matches an optional run of space
...+	Matches "..." 1 or more times: e.g. "x+" matches any run of "x"s
...{6}	Matches "..." exactly 6 times (similarly for other numbers, of course)
...{2,5}	Matches "..." between 2 and 5 times
...{3,}	Matches "..." 3 or more times
....?	"?" after any repetition makes it "lazy", matching as few repeats as it can

Numbered subexpressions

(...)	Groups part of the expression together: matches if the interior matches
\1	Matches the contents of the 1st subexpression reading left to right
\2	Matches the contents of the 2nd, and so on up to "\9" (but no further)

Unnumbered subexpressions

(# ...)	Comment: always matches, and the contents are ignored
(?= ...)	Lookahead: matches if the text ahead matches "...", but doesn't consume it
(?! ...)	Negated lookahead: matches if lookahead fails
(?<= ...)	Lookbehind: matches if the text behind matches "...", but doesn't consume it
(?<! ...)	Negated lookbehind: matches if lookbehind fails
(> ...)	Possessive: tries to match "..." and if it succeeds, never backtracks on this
(?(1)...)	Conditional: if \1 has matched by now, require that "..." be matched
(?(1)...|...)	Conditional: ditto, but if \1 has not matched, require the second part
(?(?=...)...|...)	Conditional with lookahead as its condition for which to match
(?(?<=...)...|...)	Conditional with lookbehind as its condition for which to match

IN REPLACEMENT TEXT

\char	If char is other than a-z, A-Z, 0-9 or space, expands to that literal char
\\	In particular, "\\" expands to a literal backslash "\"
\n	Expands to a line break character
\t	Expands to a tab character (but use this only with external files)
\0	Expands to the full text matched
\1	Expands to whatever the 1st bracketed subexpression matched
\2	Expands to whatever the 2nd matched, and so on up to "\9" (but no further)
\l0	Expands to \0 converted to lower case (and so on for "\l1" to "\l9")
\u0	Expands to \0 converted to upper case (and so on for "\u1" to "\u9")

21. Lists

§21.1 Lists and entries
§21.2 Constant lists
§21.3 Saying lists of values
§21.4 Testing and iterating over lists
§21.5 Building lists
§21.6 Lists of objects
§21.7 Lists of values matching a description
§21.8 Sorting, reversing and rotating lists
§21.9 Accessing entries in a list
§21.10 Lengthening or shortening a list
§21.11 Variations: arrays, logs, queues, stacks, sets, sieves and rings

§21.1 Lists and entries

Many sections in this book begin by introducing a new kind of value. Reading through in order, the possibilities mount up: numbers, times, texts, and so on. (See the Kinds page of the Index for a convenient list of the options.) This section is a little different: rather than showing a single new kind of value, it shows how to make a new kind out of any existing one.

If K is any kind of value, then "list of K" is also a kind of value. For instance, we could write:

```
let L be a list of numbers;
```

and this would create a new "let" variable, called L, whose kind of value is "list of numbers". On the other hand, we are not allowed to write:

```
let L be a list;
```

because "list" by itself is not a kind of value. (Inform always needs to know what kinds the values entered in a list are going to have.)

Lists are like flexible-length table columns, but that probably makes them sound more mysterious than they really are. A list is simply a sequence of values, called its "entries", numbered from 1 upwards. The number of entries is called its "length". If we try

```
let L be a list of numbers;
say "L has [the number of entries in L] entries.";
```

then we find

```
L has 0 entries.
```

This is because all lists start out empty when created: that is, they initially have 0 entries. Inform has two built-in adjectives "empty" and "non-empty" which can apply to lists, and they mean just what they ought to mean: a list is empty if its length is 0, and otherwise non-empty.

We can add entries very easily:

```
add 2 to L; add 3 to L; add 5 to L;
```

We can now, for instance, try saying the list:

```
say "L is now [L].";
```

with the result

```
L is now 2, 3 and 5.
```

Note that only numbers can be added to L: if we try

```
add "clock" to L;
```

Inform will produce a problem message, because L has kind "list of numbers", whereas "clock" is text. In this way, Inform ensures that a list always contains values of the same kind throughout. So it's not possible to construct a list whose entries are:

```
2, "fish", 4 and the Entire Game
```

Such a list would be very hazardous to deal with, in any case. If what we need is a combination of different kinds of values, tables are a better option.

Finally, note that since "list of numbers" is a kind of value in its own right, so is "list of lists of numbers", and so on - though such lists are trickier to deal with, they are sometimes handy.

§21.2 Constant lists

It is convenient to have a concise way to write down a constant list. Just as we could write

"231", say, or "7:01 AM" to refer to particular number and time constants, so we can write list constants:

```
let L be {1, 2, 3, 4};
```

Inform recognises that "{1, 2, 3, 4}" is a list because of the braces, and looks at the entries inside, sees that they are numbers, and deduces that it is a constant whose kind of value is "list of numbers". L is then a temporary list variable and we can add to it, remove things, and so on as we please - {1, 2, 3, 4} is merely its initial value.

When constructing lists, it is worth noting that Inform requires spaces after the commas (which seems a little harsh, but is necessary because otherwise many sensible literal specifications for units would be impossible - anyway, the reason isn't important here). So

```
let L be {1,2,3,4};
```

would produce problem messages. But Inform does not require spaces round its braces.

We call this way of writing a list "brace notation". In mathematics, braces are usually used for sets, and properly speaking these are sequences not sets - so that "{1, 2, 3, 4}" is different from "{4, 3, 2, 1}" - but it is still a familiar notation. Similarly,

```
let L be {"apple", "pear", "loganberry"};
```

makes L a list of texts; and

```
The marshmallow, the firework and the stink bomb are in the Scout Hut. The list of
prohibited items is a list of objects that varies. The list of prohibited items is {the firework,
the stink bomb}.
```

makes a global variable ("list of prohibited items") with kind of value "list of objects", and whose initial value is to contain two things: the firework and the stink bomb. More exotically, if we need to make lists of lists:

```
let L be {{1, 2}, {6, 7, 8}};
```

gives L the kind of value "list of lists of numbers", with (initially) two entries: the list {1, 2} (a list of numbers), then the list {6, 7, 8} (ditto).

Constant lists are convenient, too, when a column in a table needs to contain lists:

```
The duck, the orange, the cider, the cinnamon and the orange are in the Kitchen.

Table of Requirements
```

recipe	ingredients
"duck à l'orange"	{the duck, the orange}
"spiced cider"	{the cider, the cinnamon, the orange}

A special word about the constant list "{ }". This means the list with no entries - the empty list. If we try to create a new "let" variable M with

```
let M be { };
```

then Inform will produce a problem message, because it cannot tell what sort of list M will be: a list of numbers, or texts, or times, or…? On the other hand, writing

```
now M is { };
```

is fine if M already exists, and then does the obvious thing - empties M. Similarly, a table column in which every entry is "{ }" produces a problem message unless the heading for that column spells out the kind of value stored within it: for instance, "ingredients (list of texts)".

All of this is a notation for constant lists only, not some sort of gluing-things-together operation. So this, for instance:

```
let L be {100, the turn count};
```

is not allowed, even though "the turn count" is a number: because it is a number that varies, the braces do not contain constants, and therefore this is not a list constant.

§21.3 Saying lists of values

Any list L can be said, provided that its contents can be said. For example:

```
let L1 be {2, 3, 5, 7, 11};
say L1;
```

produces the text "2, 3, 5, 7 and 11" - unless we have "Use serial comma." set, in which case a comma appears after the 7. We also have the option of using the more formal notation:

say "[(list of values) in brace notation]"

This text substitution produces the list in the form of "{", then a comma-separated list, and then "}", which looks less like an English sentence but more mathematical. Example:

```
"[list of people in brace notation]"
```

might produce "{ yourself, Mr Darcy, Flashman }".

If we say a list of lists, then the individual entry lists are always printed in brace notation: the ordinary sentence way would be incomprehensible.

Of course, the values in L1 are written out in number form because L1 is a list of numbers: we could alternatively try

```
let L2 be {the piano, the music stand};
say L2;
```

which produces "piano and music stand". Lists of objects can be said in two additional ways:

say "[(list of objects) with definite articles]"

This text substitution writes out the list in sentence form, adding the appropriate definite articles. Example:

```
let L be {the piano, the music stand};
say "[L with definite articles]";
```

says "the piano and the music stand".

say "[(list of objects) **with indefinite articles**]"

This text substitution writes out the list in sentence form, adding the appropriate indefinite articles. Example:

```
let L be {the piano, the music stand};
say "[L with definite articles]";
```

says "a piano and a music stand".

Example

424. **Oyster Wide Shut**
Replacing Inform's default printing of properties such as "(closed)", "(open and providing light)", etc., with our own, more flexible variation.

RB 6.7 Inventory

§21.4 Testing and iterating over lists

If L is a list, we can interrogate it to see whether it does or does not contain (at least one instance of) any compatible value V:

if (value) **is listed in** (list of values):

This condition is true if the given value, which must be of a compatible kind, is one of those in the list. For instance, if L is our list of the numbers 2, 3, 5, 7 and 11 then 5 is listed in it but 6 is not.

if (value) **is not listed in** (list of values):

This condition is true if the given value, which must be of a compatible kind, is not one of those in the list.

We can also repeat running through a list (just as we can with table rows). Thus:

repeat with (a name not so far used) **running through** (list of values):

This phrase causes the block of phrases following it to be repeated once for each item in the given list, storing that value in the named variable. (The variable exists only temporarily, within the repetition.) Example:

```
let L be {2, 3, 5, 7, 11, 13, 17, 19};
repeat with prime running through L:
    ...
```

If the list is empty, nothing happens: the "..." phrase(s) are never tried.

In the next sections, we shall see that it is possible to change, reorder and resize lists. But it's important never to change a list that's being repeated through. The following:

```
let L1 be {1, 2, 3, 4};
repeat with n running through L1:
    remove n from L1;
```

leaves L1 containing {2, 4}, since the removals from the list cause it to shuffle back even while we repeat through it - a bad, bad idea.

§21.5 Building lists

We have already seen "add... to...". This in fact comes in two forms:

add (value) **to** (list of values)

This phrase adds the given value to the end of the list. Example:

```
let L be {60, 168};
add 360 to L;
```

results in L being {60, 168, 360}. Note that the value is added even if it already occurs somewhere in L; this can be avoided with "if absent". So:

```
add 168 to L, if absent;
```

would do nothing - it is already there.

add (list of values) **to** (list of values)

This phrase adds the first list to the end of the second. Example:

```
let L be {2, 3, 5, 7};
add {11, 13, 17, 19} to L;
```

results in L being {2, 3, 5, 7, 11, 13, 17, 19}.

If we don't want to add new entries at the end, we can instead say where they should go:

add (value) **at entry** (number) **in/from** (list of values)

This phrase adds the given value so that it becomes the entry with that index number in the list. Example:

```
let L be {1, 2, 3, 4, 8, 24};
add 12 at entry 6 in L;
```

sets L to {1, 2, 3, 4, 8, 12, 24}. If there are N entries in L, then we can add at any of entries 1 up to N+1: adding at entry N+1 means adding at the end. The phrase option "if absent" makes the phrase do nothing if the value already exists anywhere in L.

add (list of values) **at entry** (number) **in/from** (list of values)

This phrase adds the first list to the second so that it begins at the given position. Example:

```
let L be {1, 2, 3, 4};
add {4, 8, 12} at entry 3 in L;
```

results in L being {1, 2, 4, 8, 12, 3, 4}.

A list is allowed to contain duplicates, and the order matters. For instance:

```
let L be {2, 2, 3};
```

makes L into "2, 2 and 3". This is a different list to the one made by:

```
let M be {2, 3, 2};
```

even though L and M have the same values, repeated the same number of times - for two lists to be equal, they must have the same kind of entry, the same number of entries, and the same entries in each position.

We can also strike out values:

remove (value) **in/from** (list of values)

This phrase removes every instance of the given value from the list. Example:

```
let L be {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
remove 1 from L;
```

results in L being {3, 4, 5, 9, 2, 6, 5, 3}. Ordinarily "remove 7 from L" would produce a run-time problem, since L does not contain the value 7, but using the "if present" option lets us off this: the phrase then does nothing if L does not contain the value to be removed.

remove (list of values) **in/from** (list of values)

This phrase removes every instance of any value in the first list from the second. Example:

```
let L be {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
remove {0, 2, 4, 6, 8} from L;
```

results in L being {3, 1, 1, 5, 9, 5, 3}. If both lists are large, this can be a slow process, and we might do better by sorting them and trying a more sophisticated method. But this is convenient for anything reasonable-sized.

Again, we can also remove from specific positions:

remove entry (number) **in/from** (list of values)

This phrase removes the entry at the given position, counting from 1 as the first entry. (Once it is removed, the other entries shuffle down.) Example:

```
let L be {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
remove entry 3 from L;
```

results in L being {3, 1, 1, 5, 9, 2, 6, 5, 3}.

remove entries (number) **to** (number) **in/from** (list of values)

This phrase removes the entries at the given range of positions, counting from 1 as the first entry. (Once they are removed, the other entries shuffle down.) Example:

```
let L be {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
remove entries 3 to 6 from L;
```

results in L being {3, 1, 2, 6, 5, 3}.

Example

425. **Robo 1**
A robot which watches and records the player's actions, then tries to repeat them back in the same order when he is switched into play-back mode.
RB 7.12 Characters Following a Script

§21.6 Lists of objects

Lists can be made of values of any kind (including other lists), but lists of objects are especially useful. We could always make these "by hand":

```
let L be {the pot plant, the foxglove};
```

But it is usually easier and clearer to use descriptions.

list of (description of values) ⇒ *value*

This phrase produces the list of all values matching the given description. Inform will issue a problem message if the result would be an infinite list, or one which is impractical to test: for instance "list of even numbers" is not feasible.

While that works nicely for many kinds of value ("list of recurring scenes", say), it's particularly useful for objects:

```
let L be the list of open containers;
add the list of open doors to L;
```

means that L now contains the open containers (if any) followed by the open doors (if any). Or, for example:

```
let L be the list of things;
remove the list of backdrops from L;
```

makes a list of all non-backdrops.

As mentioned above, lists of objects can be said in two additional ways:

```
"[L with definite articles]"
"[L with indefinite articles]"
```

And as mentioned below, they can be sorted in property value order:

```
sort L in P order;
sort L in reverse P order;
```

where P is any value property. In all other respects, lists of objects are no different to other lists.

Examples

426. **What Makes You Tick**
Building a fishing pole from several component parts that the player might put together in any order.
RB 9.8 Simple Machines

427. **Formicidae**
Manipulating the order in which items are handled after TAKE ALL.
RB 6.15 Actions on Multiple Objects

§21.7 Lists of values matching a description

The useful "list of …" syntax can also be used to produce lists of the values matching a description, too. Thus:

```
let L be the list of non-recurring scenes;
let C be the list of colours;
```

There is little to say here except for the usual warning that some kinds of value have a range which is too large to make this possible. For instance, Inform could not sensibly represent:

```
let N be the list of even numbers;
```

It would just be too large to hold. In general, if we can repeat through, or find the number of, values matching a description, then we can also use "list of" to bring them all together. See the chart of kinds of value in the Kinds index for a project for which kinds of value allow this.

§21.8 Sorting, reversing and rotating lists

Any list L can be reversed:

reverse (list of values)

This phrase puts the list in reverse order. The old entry 1 becomes the new last entry, and so on: reversing an empty list or a list containing only one entry leaves it unchanged. Example:

```
let L be {11, 12, 14, 15, 16, 17};
reverse L;
```

results in L being {17, 16, 15, 14, 12, 11}.

And any list can similarly be sorted:

sort (list of values)

This phrase puts the list into ascending order. Example:

```
let L be {6 PM, 11:13 AM, 4:21 PM, 9:01 AM};
sort L;
```

results in L being {9:01 AM, 11:13 AM, 4:21 PM, 6 PM}.

sort (list of values) **in reverse order**

This phrase puts the list into descending order. Example:

```
let L be {6 PM, 11:13 AM, 4:21 PM, 9:01 AM};
sort L in reverse order;
```

results in L being {6 PM, 4:21 PM, 11:13 AM, 9:01 AM}.

sort (list of values) **in random order**

This phrase puts the list into a uniformly random order, shuffling it as if it were a pack of cards. Example:

```
let L be {1, 2, 3, 4, 5, 6};
sort L in random order;
```

might result in L being {3, 1, 5, 6, 4, 2}. Or any of 719 other arrangements, including being left as it was.

Lists of objects can also be sorted in property value order. For instance,

sort (list of objects) **in** (property) **order**

This phrase puts the list into ascending order of the values of the given property for the items in the list; this is only allowed if all of those values do have the property in question. Example:

```
let L be the list of people;
sort L in carrying capacity order;
```

would arrange people with weaklings first, titans last.

sort (list of objects) **in reverse** (property) **order**

This phrase puts the list into descending order of the values of the given property for the items in the list; this is only allowed if all of those values do have the property in question. Example:

```
let L be the list of people;
sort L in reverse carrying capacity order;
```

would arrange people with titans first, weaklings last.

Rotating a list means moving all of its entries along by one place, and then moving the one on the end back to the start. For instance, if L is {1, 2, 3, 4}, then

rotate (list of values)

This phrase shuffles the entries of the list forwards (to the right) by one place, so that the 1st becomes 2nd, the 2nd becomes 3rd, and so on until the last, which becomes the new first entry. Example:

```
let L be { "cow", "heifer", "bullock" };
rotate L;
```

results in L being { "bullock", "cow", "heifer" }.

rotate (list of values) **backwards**

This phrase shuffles the entries of the list backwards (to the left) by one place, so that the 3rd becomes 2nd, the 2nd becomes 1st, and so on; the previous 1st entry becomes the new last entry. Example:

```
let L be { "cow", "heifer", "bullock" };
rotate L backwards;
```

results in L being { "heifer", "bullock", "cow"}. (This achieves the same effect as "reverse L; rotate L; reverse L;" but is a little faster, and a lot less effort to read.)

§21.9 Accessing entries in a list

The length of a list can change as values are added or removed, and can in principle be any number from 0 upwards. A list with 0 entries is empty. We can find the length with:

number of entries in/of/from (list of values) ⇒ *number*

This phrase produces the number of positions in the list. Example:

```
the number of entries in {1, 1, 1, 3, 1}
```

is 5, even though there are only two genuinely different items in the list.

If the length is N then the entries are numbered from 1 (the front) to N (the back). These entries can be accessed directly by their numbers. For instance,

```
entry 2 of L
```

refers to the second entry of L: it can be used as a value, or changed, just as if it were a named variable. For instance, we could write:

```
now entry 7 of L is "Spain";
say "The rain in [entry 7 of L] stays mainly in the plain.";
```

which would (untruthfully) print "The rain in Spain stays mainly in the plain", but only if L had an entry 7 to make use of: if L were a list of 5 entries, say, then a run-time problem results. (And if L cannot hold text, a problem message means that we never get as far as run-time.) Because entries number from 1, this is always incorrect:

```
entry 0 of L
```

and if L is currently empty, then there is no entry which can be accessed, so that any use of "entry … of L" would produce a run-time problem. There are programming languages in the world where accessing entry 100 in a 7-entry list automatically extends it to be 100 entries long: Inform is not one of them. But see the next section for how to change list lengths explicitly.

Example

428. **Robo 2**
A robot which watches and records the player's actions, then tries to repeat them back in the same order when he is switched into play-back mode.
RB 7.12 Characters Following a Script

§21.10 Lengthening or shortening a list

We can explicitly change the length of a list like so:

change (list of values) **to have** (number) **entries/entry**

This phrase alters the given list so that it now has exactly the number of entries given. Example:

```
change L to have 21 entries;
```

If L previously had more than 21 entries, they are thrown away (and lost forever); if L previously had fewer, then new entries are created, using the default value for whatever kind of value L holds. So extending a list of numbers will pad it out with 0s, but extending a list of texts will pad it out with the empty text "", and so on.

We can also write the equivalent phrases:

truncate (list of values) **to** (number) **entries/entry**

This phrase alters the given list so that it now has no more than the number of entries given. Example:

```
truncate L to 8 entries;
```

shortens L to length 8 if it is currently longer than that, trimming entries from the end, but would (for instance) leave a list of length 3 unchanged. Note that

```
truncate L to 0 entries;
```

empties it to { }, the list with nothing in.

truncate (list of values) **to the first** (number) **entries/entry**

This phrase alters the given list so that it now consists only of the initial part of the list with the given length. Example:

```
truncate L to the first 4 entries;
```

turns {1, 3, 5, 7, 9, 11} to {1, 3, 5, 7}.

truncate (list of values) **to the last** (number) **entries/entry**

This phrase alters the given list so that it now consists only of the final part of the list with the given length. Example:

```
truncate L to the last 4 entries;
```

turns {1, 3, 5, 7, 9, 11} to {5, 7, 9, 11}.

But we don't have to truncate: we can also -

extend (list of values) **to** (number) **entries/entry**

This phrase pads out the list with default values as needed so that it now has at least the given length. (If the list is already at least that length, nothing is done.) Example:

```
extend L to 80 entries;
```

lengthens L to length 80 if it is currently shorter than that.

For example,

```
To check sorting (N - a number):
    let L be a list of numbers;
    extend L to N entries;
    repeat with X running from 1 to N:
        now entry X of L is X;
    say "L unrandomised is [L].";
    sort L in random order;
    say "L randomised is [L].";
    sort L;
    say "L in ascending order is [L]."
```

builds a list of N numbers (initially all 0), fills it with the numbers 1, 2, 3, ..., N, then randomly reorders them, then sorts them back again, recovering the original order. The text produced by "check sorting 10" depends partly on chance but might for instance be:

```
L unrandomised is 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10.
L randomised is 6, 2, 9, 3, 10, 1, 7, 4, 8 and 5.
L in ascending order is 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10.
```

As with text in the previous chapter, a project which needs really long lists should use the Glulx virtual machine - "check sorting 10000", for instance, would break the default memory environment on the Z-machine, which is very tight, but works fine (if not very rapidly) on Glulx.

Examples

429. **Leopard-skin**
    A maze that the player can escape if he performs an exact sequence of actions.
    RB 6.14 Remembering, Converting and Combining Actions

430. **The Facts Were These**
    Creating a variant GIVE action that lets the player give multiple objects simultaneously with commands like GIVE ALL TO ATTENDANT or GIVE THREE DOLLARS TO ATTENDANT or GIVE PIE AND HAT TO ATTENDANT. The attendant accepts the gifts only if their total combined value matches some minimum amount.
    RB 6.15 Actions on Multiple Objects

§21.11 Variations: arrays, logs, queues, stacks, sets, sieves and rings

Lists are highly adaptable, and many other collection-like constructions can be made using them. This section introduces no new material, but simply suggests some of the variations which are possible.

1. The traditional computing term **array** means a list of values accessed by their entry numbers, often used in mathematical computations. The difference between an array and a list is mostly one of attitude, but usually arrays are fixed in length whereas lists can expand or contract.

2. A **log** is a list which records the most recently arrived values, but does not allow itself to grow indefinitely. In the following, which remembers the seven most recently taken items, new values arrive at the end while old ones eventually disappear from the front:

```
The most-recently-taken list is a list of objects that varies.
Carry out taking something (called the item):
    truncate the most-recently-taken list to the last 6 entries;
    add the item to the most-recently-taken list.
After taking:
    say "Taken. (So, your recent acquisitions: [most-recently-taken list].)"
```

Note that the most-recently-taken list begins play as the empty list, grows as the first few items are taken, but then stabilises at length 7 thereafter. If we need to remember recent history, but only *recent* history, then a log is better than a list which can grow indefinitely, because there is no risk of speed reduction or memory exhaustion in a very long story.

3. A **queue** is a list of values which are waiting for attention. New values join at the back, while those being dealt with are removed from the front (whereupon the whole queue moves up one). An empty queue means that nobody is waiting for attention: but there is, in principle, no upper limit to the size of a queue, as anyone who has tried to make a couchette reservation at Roma Termini will know.

Queues typically form when two independent processes are at work, but going at different or variable speeds. An empty queue looks just like any other list:

```
The queue is a list of objects that varies.
```

(Invariably people, in what follows, but we'll make it a "list of objects" to allow for other possibilities too.) Once we identify a "new customer", we can join him to the queue thus:

```
add the new customer to the queue;
```

The process of serving the customers needs to make sure there is actually somebody waiting in the queue before it does anything:

```
Every turn when the number of entries in the queue is not 0:
    let the next customer be entry 1 of the queue;
    say "[The next customer] is served and leaves.";
    remove entry 1 from the queue.
```

Of course queues can also be constructed which empty from other positions, rather than the front: or we could make what computer scientists sometimes call a **deque**, a "double-ended queue" where new values arrive at both ends.

4. A **stack** is like a queue except that values arrive at, and are removed from, the same end. Stacks are slightly faster if the active end is the back rather than the front, though this will only be noticeable if they grow quite large.

To put a value V onto a stack S (which is known as "pushing") is simple:

```
add V to S;
```

And to remove a value from the top of the stack (which is known as "pulling"):

```
let N be the number of entries in S;
let V be entry N of S;
remove entry N from S;
```

Note that the middle line, accessing entry N, will fail if N = 0, that is, if the stack is empty: Inform's list routines will produce a run-time problem message.

Stacks are useful if some long-term process is constantly being interrupted by newer and more urgent demands, but they can also be used in planning. If a character has a long-term goal, which needs various short-term goals to be achieved along the way, then a stack can represent the goals currently being pursued. The top of the stack represents what the character is trying to achieve now. If the character realises that it needs to achieve something else first, we put that new goal onto the top of the stack, and it becomes the new current goal. When the character completes a task, it can be removed, and we can go back to trying to finish whatever is now on top. When the stack is empty, the character has achieved the original goal.

5. Notoriously, **set** has 464 distinct meanings in the Oxford English Dictionary, making it the single most ambiguous word in the language. Here we mean not the home of a badger or the Egyptian god of the desert, but the mathematical sense: a collection of values (sometimes called "elements") without duplicates, and which is normally written in brace notation and in some natural order for the reader's convenience.

The trick here is to maintain the principle that, at all times, our list is sorted in order and contains no duplicates. To provide an example, we start with two sets of numbers:

```
let S be {2, 4, 8, 16, 32, 64};
let T be {2, 4, 6, 10};
```

Here we add an element to T:

```
add 8 to T, if absent; sort T;
```

The "if absent" clause ensures that no duplicate can occur, and by sorting T afterwards, we maintain the principle that a set must remain in order - so T is now {2, 4, 6, 8, 10}, not {2, 4, 6, 10, 8}. (Inform's sorting algorithm is fast on nearly-sorted lists, so frequent sorting is not as inefficient as it might look.)

We next take the union of T and S, that is, the set containing everything which is in either or both:

```
let U be S; add T to U, if absent; sort U;
```

This makes U = {2, 4, 6, 8, 10, 16, 32, 64}, and once again no duplicates occur and we preserve the sorting. The intersection of T and S, the set of elements in both of them, is a little trickier:

```
let I be T;
repeat with the element running through T:
    if the element is not listed in S, remove the element from I.
```

(Faster methods could be devised which exploit the sortedness of T and S, but are not worth it for shortish lists.) This produces I = {2, 4, 8}. Lastly, we can form the set difference, consisting of those elements which are in S but not in T:

```
let D be S; remove T from D, if present;
```

Here, as with intersection, since all we do is to strike out unwanted elements, the surviving ones remain in order and there is no need to sort when we are finished. This produces D = {16, 32, 64}.

6. A **sieve** is used to make a complicated choice where there are many constraints, by ruling out impossible cases to see what is left. The term derives from the kitchen utensil (for sieving fine grains of flour), but via the name of the "sieve of Eratosthenes", an ancient Greek method for determining the prime numbers.

Using a sieve is much like using a set, and the difference is mainly one of outlook - we are interested in what does not belong, rather than what does.

7. A **ring** is not so much a row of values, more a circle, with the last and first entries thought of as adjacent. One position is usually thought of as special, and is the place where new items are added: this may as well be entry 1. For instance, to add "new item" to the ring:

```
add the item at entry 1 in the ring;
```

To set "item" to the frontmost value and extract it from the ring:

```
let the item be entry 1 of the ring;
remove entry 1 from the ring;
```

And we can rotate the ring in either direction, making a different entry the new entry 1

and therefore the new frontmost value:

```
rotate the ring;
rotate the ring backwards;
```

A last note to conclude the chapter on lists. Lists, like almost all other values in Inform, can be passed to phrases as parameters. However, note that they are genuine values, not what some programming languages call "references" or "pointers". So the following:

```
To mess with (L - a list of numbers):
    add 7 to L, if absent.
```

does nothing, in practice. If given a list, it adds 7 to the list, but then throws it away again, so the longer list is never seen; it's exactly like

```
To mess with (N - a number):
    now N is 3.
```

which can never affect anything other than its own temporary value "N", which expires almost immediately in any case.

If we want a phrase which changes a list in a useful way and gives it back to us, we need a phrase which both takes in and gives back:

```
To decide which list of numbers is the extended (L - a list of numbers):
    add 7 to L, if absent;
    decide on L.
```

And then, for example -

```
the extended { 2, 4, 6 };
```

produces:

```
{ 2, 4, 6, 7 }
```

Examples

431. **Circle of Misery**
Retrieving items from an airport luggage carousel is such fun, how can we resist simulating it, using a list as a ring buffer?
RB 8.4 Furniture

432. **Eyes, Fingers, Toes**
A safe with a multi-number combination, meant to be dialed over multiple turns, is implemented using a log of the last three numbers dialed. The log can then be compared to the safe's correct combination.
RB 9.2 Bags, Bottles, Boxes and Safes

433. **The Fibonacci Sequence**
The modest Leonardo Fibonacci of Pisa will be only too happy to construct his sequence on request, using an array.
RB 10.11 Mathematics

434. **I Didn't Come All The Way From Great Portland Street**
In this fiendishly difficult puzzle, which may perhaps owe some inspiration to a certain BBC Radio panel game (1967-), a list is used as a set of actions to help enforce the rule that the player must keep going for ten turns without hesitation, repetition, or deviating from the subject on the card.
RB 6.14 Remembering, Converting and Combining Actions

435. **Lugubrious Pete's Delicatessen**
In this evocation of supermarket deli counter life, a list is used as a queue to keep track of who is waiting to be served.
RB 7.16 Social Groups

436. **Sieve of Eratosthenes**
The haughty Eratosthenes of Cyrene will nevertheless consent to sieve prime numbers on request.
RB 10.11 Mathematics

437. **Your Mother Doesn't Work Here**
Your hard-working mother uses a list as a stack: urgent tasks are added to the end of the list, interrupting longer-term plans.
RB 7.12 Characters Following a Script

22. Advanced Phrases

§22.1 A review of kinds
§22.2 Descriptions as values
§22.3 Phrases as values
§22.4 Default values for phrase kinds
§22.5 Map, filter and reduce
§22.6 Generic phrases
§22.7 Kind variables
§22.8 Matching the names of kinds
§22.9 In what order?
§22.10 Ambiguities

§22.1 A review of kinds

Most of the time, what's created in an Inform source text will have a name which can be used as a value - sometimes openly so, sometimes not. In this book, we haven't gone out of our way to make that point, because there was no real need to do so. It's possible to make heavy use of rulebooks and write large-scale Inform projects without ever needing to use a rulebook's name as a value in its own right, for example. But if we want to create sophisticated extensions to Inform, or to use modern techniques such as functional and generic programming, we need to be fluent in the language of kinds.

Inform's language of kinds has four ingredients: base kinds, constructions, kind variables and kinds of kinds.

1. Base kinds. Inform provides the following base kinds for values:

```
object, number, real number, time, truth state, text, snippet, Unicode character, action,
scene, table name, equation name, use option, action name, figure name, sound name,
external file, rulebook outcome, parser error
```

together with a few others, such as "response" and "verb", to do with linguistic features.

And Inform allows us to create new base kinds either by making more specialised kinds of object:

```
A geographical layout is a kind of object.
A marmoset is a kind of animal.
```

Or by making new enumerations or arithmetical kinds:

```
Distance is a kind of value. 10km specifies a distance.
Colour is a kind of value. Red, green and blue are colours.
```

2. Constructions. These are ways to make new kinds from existing ones. The construction most often used is "list of…". For any kind K, there is a kind called list of K. So the range of possible kinds in Inform is actually infinite, because:

```
number
list of numbers
list of lists of numbers
list of lists of lists of numbers
...
```

are all different from each other. Inform has nine constructions, as follows:

```
list of K
description of K
relation of K to L
K based rule producing L
K based rulebook producing L
activity on K
phrase K -> L
K valued property
K valued table column
```

Some of these have appeared in previous chapters, but in abbreviated form. For example, "rulebook" abbreviates "action based rulebook producing nothing", and "either/or property" is a synonym for "truth state valued property". The kinds of descriptions and phrases haven't been covered yet, but are coming up in the sections following.

These constructions can of course be combined:

```
phrase (relation of numbers to colours, number) -> list of colours
```

Brackets can be used to clarify matters:

```
phrase (phrase number -> (phrase number -> number)) -> nothing
```

Nothing will make that a simple idea, but it's unambiguous and can be puzzled out with practice.

3. Variables. In a way, that's everything: there are just base kinds and constructions on

them, and those construct every possible kind in Inform. But the language we use to describe kinds is larger than that, because it allows us to describe many kinds at once, in the same way that Inform reads the word "something" as applying to many objects at once, not as a single object.

Kind variables will be covered later in the chapter, but the idea is that:

```
To hunt for (needle - value of kind K) in (haystack - list of Ks): ...
```

allows us to describe the kinds acceptable in a phrase so that a wide range of possibilities are allowed. The above matches both:

```
hunt for 4 in { 2, 3, 4, 5 };
hunt for "fish" in { "lemon sauce", "fish", "garden peas" };
```

The letter K in the definition stood for any kind; in the first use of "hunt" here, K turned out to be "number", and in the second it was "text". On the other hand Inform would reject:

```
hunt for 4 in { containment relation, support relation };
```

because there is no kind K which can make this match the definition.

There are potentially 26 kind variables, A to Z, though it's customary to use them in the order K, L, M, …, and it's very rare to need more than two at a time.

4. Kinds of kind. Inform understands several names which look as if they are kinds, but actually aren't:

```
value, arithmetic value, enumerated value, sayable value
```

(Again, these are built in to Inform.) They are not kinds because they're just too loose and vague. Instead, they can be used in phrase definitions to match against multiple possibilities:

```
To announce (X - sayable value): say "I declare that [X] has arrived."
```

This makes "announce X" work for any value X which can be said. All the same, "sayable value" is not a kind. It could never be safe for this to be the kind of a variable, because Inform would never know what could be done with the contents (except that it could be printed out).

5. Secret inner workings. There isn't a fifth ingredient, but if there were, it would be a handful of names used in matching some of the core built-in phrases of Inform which have so-called inline I6 definitions. These are not intended for anyone else to use, and are just an internal convenience; they aren't publicly documented and might change without notice. They don't describe kinds at all, because they tell the matcher to look for something else instead. For instance, there's one called "condition", which means "match a condition rather than a value". They appear in red ink in the Phrasebook index.

§22.2 Descriptions as values

In the chapter on Descriptions, we saw that a description can be any source text which describes one or more objects: it might be as simple as "the Corn Market", or as complicated as "open containers which are in dark rooms". More or less the only restriction is that it must be unambiguous as to what counts and what does not: "three containers" is ambiguous as a description because it does not say which three.

We've now seen several interesting tricks with descriptions. In fact, if D is a description, then

```
say "You gaze mournfully at [the list of D].";
let the tally be the number of D;
let the surprise prize be a random D;
repeat with item running through D:
    ...
```

are all standard things to do. These examples make it look as if it must be possible to define phrases which act on descriptions, and in fact it is, because a description can be a value in itself. For example,

```
even numbers
open containers which are in dark rooms
```

are values of kind "description of numbers" and "description of objects" respectively. In general, if K is any kind then "description of K" is also a kind. Here is how we might make use of that:

```
To enumerate (collection - a description of objects):
    repeat with the item running through the collection:
        say "-- [The item]."
```

This makes "enumerate lighted rooms" run off a list of lighted rooms in a textual format different from the standard one produced by "say the list of lighted rooms". Inside the

definition, "collection" can be used wherever a description might be used: here, for instance, we use it as the range for the repeat loop. (That's only possible because the range is limited in size: Inform wouldn't have allowed us to range through, say, all texts.)

Purely as a convenience, we can also write "member of" or "members of" in this context. For instance, in the enumerate definition, it would have been just as good to write "…running through the members of the collection…" in the repeat. (Similarly, we could write "number of members of the collection" or "a random member of the collection", which looks grammatically tidier than writing "number of the collection" or "random of the collection" - though in fact both of these do work.)

Finally, it's sometimes useful in an abstract situation to test

if (value) **matches** (description of values):

This condition is true if the value matches the description; the kinds must be compatible, or Inform will issue a problem message. There is no point using this for cases where the description is given explicitly:

```
if 4 matches even numbers, ...
```

because it is easier to write just:

```
if 4 is an even number, ...
```

So this condition is only useful when the description is stored in some variable, and its identity is not known.

Example

438. **Curare**
A phrase that chooses and names the least-recently selected item from the collection given, allowing the text to cycle semi-randomly through a group of objects.

RB 2.1 Varying What Is Written

§22.3 Phrases as values

Given any two kinds K and L, the kind "phrase K -> L" is now a kind. (This is meant to look like a mathematical function arrow.) For example, the phrase defined by

```
To decide which number is the square of (N - a number): ...
```

has the kind "phrase number -> number". Brackets and commas are used if the phrase combines several values, so

```
To decide which text is (T - text) repeated (N - a number) times: ...
```

has the kind "phrase (text, number) -> text". The word "nothing" is used if there are no values in, or no value out - thus

```
To decide which number is the magic target: ...
```

has kind "phrase nothing -> number", and

```
To dig (eastward - length) by (northward - length): ...
```

has the kind "phrase (length, length) -> nothing".

But how are we to get at these values? The answer is that we need to give a phrase a name in order to do so. For example:

```
To decide what number is double (N - a number) (this is doubling):
    decide on N plus N.
```

This is the same syntax used to name rules, and the idea is the same. If we try "showme doubling", the result is

```
phrase number -> number: doubling
```

The main thing we want to do with a phrase is to apply it. So:

```
showme doubling applied to 2;
```

produces

```
"doubling applied to 2" = number: 4
```

There are versions of "applied to" for phrases applied to 0 to 3 values:

(phrase nothing -> value) **applied** ⇒ *value*

This phrase produces the result of applying the given phrase, which must be one which takes no values itself.

(phrase value -> value) **applied to** (value) ⇒ *value*

This phrase produces the result of applying the given phrase, which must be one which takes one value itself.

(phrase (value, value) -> value) **applied to** (value) **and** (value) ⇒ *value*

This phrase produces the result of applying the given phrase, which must be one which takes two values itself.

(phrase (value, value, value) -> value) **applied to** (value) **and** (value) **and** (value) ⇒ *value*

This phrase produces the result of applying the given phrase, which must be one which takes three values itself.

So for example:

```
F applied
F applied to V
F applied to V and W
F applied to V and W and X
```

For phrases which do not produce a value, we use "apply":

apply (phrase nothing -> nothing)

This phrase causes the given phrase to be applied. It must be one which takes no values itself.

apply (phrase value -> nothing) **to** (value)

This phrase causes the given phrase to be applied. It must be one which takes one value itself.

apply (phrase (value, value) -> nothing) **to** (value) **and** (value)

This phrase causes the given phrase to be applied. It must be one which takes two values itself.

apply (phrase (value, value, value) -> nothing) **to** (value) **and** (value) **and** (value)

This phrase causes the given phrase to be applied. It must be one which takes three values itself.

Thus:

```
apply F;
apply F to V;
apply F to V and W;
apply F to V and W and X;
```

§22.4 Default values for phrase kinds

The default value for "phrase K -> nothing" is a phrase which does nothing. For example, if we write:

```
let S be a phrase numbers -> nothing;
```

then S is created holding the default phrase numbers -> nothing, and if we then try it out with:

```
apply S to 17;
```

we will indeed find that nothing happens.

The default value for "phrase K -> L" is a phrase which, no matter what value of K it applies to, always produces the default value of L. (It's a sort of equivalent of the zero

function in mathematics - indeed that's exactly what it is, if L is "number".) So:

```
let Q be a phrase numbers -> times;
showme Q;
showme Q applied to 4;
showme Q applied to -7;
```

produces:

```
"q" = phrase number -> time: default value of phrase number -> time
"q applied to 4" = time: 9:00 am
"q applied to -7" = time: 9:00 am
```

Here Q is set to the default phrase because we didn't give it any other value; it has the name we might expect ("default value of phrase number -> time") and it works as advertised, producing 9:00 am no matter what number is fed in.

More ambitiously, and supposing that we have a kind called "colour" whose first possible value is "red":

```
let R be a phrase numbers -> (phrase numbers -> colours);
showme R;
showme R applied to 3;
showme (R applied to 3) applied to 2;
```

produces:

```
"r" = phrase number -> (phrase number -> colour): default value of phrase
number -> (phrase number -> colour)
"r applied to 3" = phrase number -> colour: default value of phrase number
-> colour
"( r applied to 3 ) applied to 2" = colour: red
```

§22.5 Map, filter and reduce

When a mass of computations has to be done, the traditional approach is to work through them in a "repeat" loop. One modern alternative, growing in popularity, is to form a list of inputs; then apply the same computation to each input in turn to form a list of results (this is called "mapping"); throw out any bad or unwanted results ("filtering"); and then combine the surviving results into a single composite answer ("reducing", though some programming languages call this "folding" or "accumulation"; it's a much-reinvented idea).

Inform provides all three of these fundamental list-processing operations. There is no special term for a "map", because Inform treats it as another case of "applied to".

(phrase value −> value) **applied to** (list of values) ⇒ *value*

This phrase takes the list, applies the phrase to each entry in the list, and forms a new list of the result. Example:

```
To decide what number is double (N - a number) (this is doubling):
    decide on N plus N.
```

Then "doubling applied to 2" produces 4, by the simpler definition of "applied to", but also:

```
doubling applied to {2, 3, 4}
```

produces the list {4, 6, 8}.

More divertingly, suppose we define:

```
To decide what text is the longhand form of (N - a number)
    (this is spelling out):
    decide on "[N in words]".
To decide what text is the consonant form of (T - text)
    (this is txtng):
    replace the regular expression "<aeiou>" in T with "";
    decide on T.
```

Then we can write a chain of three maps in succession:

```
txtng applied to spelling out applied to doubling applied to {3, 8, 4, 19, 7}
```

to produce the value {"sx", "sxtn", "ght", "thrty-ght", "frtn"}.

Next, filtering. Here we make use of descriptions, in order to say what values will be allowed through the filter. So:

filter to (description of values) **of** (list of values) ⇒ *value*

This phrase produces a new list which is a thinner version of the one given, so that it contains only those values which match the description given. Example:

```
filter to even numbers of {3, 8, 4, 19, 7}
```

produces {8, 4}, with the values 3, 19, and 7 failing to make it through. A sufficiently fine filter may well thin out a list to a single entry, or even no entries at all, but the result is always a list.

To get the full effect of filtering, we probably need to define an adjective or two. For example:

```
Definition: a text (called T) is lengthy if the number of characters in it is greater than 6.
```

We can then write, for example:

```
let L be the filter to lengthy texts of spelling out applied to {15, 2, 20, 29, -4};
showme L;
```

which produces the list {"fifteen", "twenty-nine", "minus four"}.

Lastly, reduction. In order to combine a whole list of values, we need a phrase to combine any two. Here are some samples:

```
To decide what number is the larger of (N - number) and (M - number)
    (this is maximization):
    if N > M, decide on N;
    decide on M.

To decide what text is the concatenation of (X - text) and (Y - text)
    (this is concatenation):
    decide on "[X][Y]".
```

And here are some sample reductions:

```
let X be the maximization reduction of {3, 8, 4, 19, 7};
let Y be the concatenation reduction of txtng applied to spelling out
    applied to doubling applied to {3, 8, 4, 19, 7};
```

sets X to 19, the highest of the values, and Y to the text "sxsxtnghtthrty-ghtfrtn". In each case a list has been reduced to a single value which somehow combines the contents.

(phrase (value, value) −> value) **reduction of** (list of values) ⇒ *value*

This phrase works through the list and accumulates the values in it, using the phrase supplied. Example: if we have

```
To decide what number is the sum of (N - number) and (M - number)
    (this is summing):
    decide on N + M.
```

then the summing reduction of {3, 8, 4, 19, 7} is the number 41, obtained by

```
(((3 + 8) + 4) + 19) + 7
```

so that the summing phrase has been used four times.

Is map/filter/reduce always a good idea? Devotees point out that almost any computation can be thought of in this way, and in systems where the work has to be distributed around multiple processors it can be a very powerful tool. (There are programming languages without loops where it's essentially the only tool.) At its best, it reads very elegantly: one assembles all of the tools needed - definitions of doubling, lengthy, spelling out, concatenation and so on - and then each actual task is expressed in a single line at the end.

On the other hand, there are also times when this is a needlessly complicated disguise for what could more easily be done with a "repeat" loop, and also more efficiently since assembling and dismantling lists in memory does take some overhead time. So these list operations are not a panacea, but it's good to have them available.

§22.6 Generic phrases

The following looks quite innocent:

```
To say (V - value) twice: say "[V]. [V], I say!"
```

It's clear at a glance what this is intended to do, but at a second glance things aren't so straightforward. "Value" is not itself a kind - it's too big and unspecific. For instance, if we were to allow a variable to be just "a value", we could freely set it to 12 one minute and to "dahlias" the next, and such a variable would be dangerous since we would never know

what could safely be done with its contents. A phrase like this one is called "generic", because it's not so much a single, actual phrase as a recipe to make phrases. (Inform automatically works out which kinds we need the phrase for, and creates a version of the phrase for those kinds.)

So "value" is not a kind, but a kind of kind. Inform has several of these:

```
value, arithmetic value, enumerated value, sayable value
```

These act as ways to say "a value of any kind matching this can go here". For example, "value" is a way to say "any kind at all"; "arithmetic value" is any kind which arithmetic can be performed on (any kind with the little calculator icon in the Arithmetic part of the Kinds index); and so on. If we write:

```
To double (V - arithmetic value): say "[V times 2]."
```

the restriction to "arithmetic value" means that although "double 3", "double 6 kg", etc., would be matched, "double the Entire Game" would not - you can't perform arithmetic on scenes. Similarly, it would have been tidier to write:

```
To say (V - sayable value) twice: say "[V]. [V], I say!"
```

because then Inform will make it clearer why "say X twice" won't work if X is one of those rare values which it can't say (an activity, for instance).

The Kinds index shows which kinds match against which of these "kinds of kind". For instance, it shows that "time"

```
Matches: value, arithmetic value, sayable value
```

which means that time is something we can do arithmetic on, and can say.

§22.7 Kind variables

The examples of generic phrases in the previous section were really only toy examples. Suppose we want a phrase which will take any arithmetic value and triple it. We could do something like this:

```
To triple (V - arithmetic value): say "[V times 3]."
```

But this only prints the answer. Suppose we want to be given the value back, instead:

how can we write the phrase? The trouble is that, not knowing the kind of V, we can't say what kind will be produced. We need a way of saying "the same kind comes out as went in". Inform expresses that using kind variables:

```
To decide which K is triple (original - arithmetic value of kind K):
    decide on 3 times the original.
```

Here, K stands for any kind which matches "arithmetic value". Inform supports exactly 26 of these symbols, which are written A to Z, but it's customary to use K and L. (They can be written in the plural if we like: e.g., "list of Ks". But they must always use a capital letter: "list of k" is not allowed.)

Each symbol we use has to be declared in exactly one of the bracketed ingredients for the phrase - here, the declaration is "arithmetic value of kind K". That creates K and says that it has to be arithmetic; if we'd just said "value of kind K", it could have been anything. (Alternatively, we could use any of the kinds of kind in the previous section.)

For a more ambitious example, here is one way to define the mapping operation described earlier in the chapter:

```
To decide what list of L is (function - phrase K -> value of kind L)
    applied to (original list - list of values of kind K):
    let the result be a list of Ls;
    repeat with item running through the original list:
        let the mapped item be the function applied to the item;
        add the mapped item to the result;
    decide on the result.
```

Here we need two symbols to explain the complicated way that the values going in and out have to match up to each other. Note also the way that the temporary variable "result" is created:

```
let the result be a list of Ls;
```

Ordinarily, of course, "L" is not a kind. But within the body of a phrase definition, it means whatever kind L matched against.

When a symbol occurs several times in the same definition, subtle differences can arise according to which appearance is the declaration. These are not quite the same:

```
To hunt for (V - value of kind K) in (L - list of Ks): ...
To hunt for (V - K) in (L - list of values of kind K): ...
```

The difference arises - though very rarely - if V has some different kind compared to the list entries, but which can be used as if it were of that kind. For example,

```
hunt for the player's command in {"take all", "wait"};
```

Here V is a snippet, but L is a list of texts; and a snippet can be used in place of a text, but not vice versa. So this will match the second definition, because K is set to "text", but it won't match the first, where K is set to "snippet".

§22.8 Matching the names of kinds

Sometimes a phrase needs to know what kind it's to work on, but isn't going to be given any particular value of it. For example:

```
To assay (name of kind of value K):
    repeat with item running through Ks:
        say "There's [item].";
    say "But the default is [default value of K].";
```

Note that there's no hyphen, and no name for the bracketed token - it only sets K. We can then have, say:

```
assay colours;
assay vehicles;
```

But "assay texts" would throw a problem message, because we can't repeat through all possible texts. For a different reason,

```
assay open doors;
```

would not be allowed - "open doors" is a description which applies to some doors and not others; it isn't a kind. It would make no sense to talk about "default value of open door", for example.

§22.9 In what order?

Recall the definition:

```
To slam shut (box - an open container): say "With great panache, you slam shut [the box].".
```

Suppose we then try to "slam shut the wall safe" at a time during play when the wall safe

is already closed. An error message will then be printed during play, since there must be a mistake in the design. The combination of checking both when Inform builds the story file and then continuously when the story file is played guarantees that, in all cases, a varying item such as "box" in the definition of "To slam shut (box - open container)" always satisfies the condition laid down.

Instead suppose we also have the following definition:

```
To slam shut (box - a container): say "You are unable to slam shut [the box], which is already closed.".
```

We now have two definitions of "slam shut". Sometimes the box it's applied to will be closed, in which case only the second definition fits, and will be the one used. But if the box is open, both definitions fit. Which will happen? The rule is:

> 1. A narrower condition takes precedence over a broader one;
>
> 2. If neither condition is narrower than the other, then whichever phrase was defined later in the source code is the one taking precedence;
>
> 3. Except that if the phrase is being used in the definition of phrase P, then P is always last in precedence order, so that recursion is always the very last possibility considered. This allows more specific or later definitions to make use of less specific or earlier ones in a natural way.

Rule 1 ensures that a definition involving "open container" takes priority over one which merely involves "container", for instance.

And therefore when the box is open, it's the more specific phrase to do with open containers which is invoked: so, with great panache, the box is slammed shut.

On the other hand, neither of these patterns is narrower than the other:

```
To describe (something - transparent): ...
To describe (something - container): ...
```

Some containers are transparent, some not; some transparent things are containers, some not. Rule 1 therefore does not apply, so it is the later of the two phrases which takes effect.

§22.10 Ambiguities

Another possible ambiguity occurs when a phrase might match two lexically different possibilities using the same words.

```
say the dishcloth;
```

could be construed as a usage of either of these cases:

```
say the (something - a thing)
say (something - a thing)
```

These of course have different effects - one produces the name with a definite article, the other just the name, so the difference is important.

The rule here is that whichever possibility contains the most words, in this case "say the (…)", takes precedence, because it's assumed to be a more specific form of the less wordy version.

23. Figures, Sounds and Files

§23.1 Beyond text
§23.2 How IF views pictures
§23.3 Virtual machines and story file formats
§23.4 Gathering the figures
§23.5 Declaring and previewing the figures
§23.6 Displaying the figures
§23.7 Recorded sounds
§23.8 Declaring and playing back sounds
§23.9 Providing accessibility text
§23.10 Some technicalities about figures and sounds
§23.11 Files
§23.12 Declaring files
§23.13 Writing and reading tables to external files
§23.14 Writing, reading and appending text to files
§23.15 Exchanging files with other programs

§23.1 Beyond text

In this chapter, we explore a number of ways to go beyond the traditional text-only, one-story-file-only model for IF.

These relatively exotic features are more demanding of the interpreter which a player uses than a plain text story file would be. They can only be used if the project is being compiled to the Glulx story file format (see the Settings panel for the project), and even then, the player will need to have a good Glulx interpreter - one which is reasonably up to date and well-written, that is - to be sure of everything working as intended.

§23.2 How IF views pictures

Looking around a bookshop, perhaps half of all the books published have illustrations. The proportion may be lower for novels, but if we count maps or other occasional diagrams, even the fiction section turns out to be surprisingly pictorial. Illustrations do not suit every book, but they are an option we would like to have available.

In the cultural history of IF, graphics in text adventures have sometimes been looked at with suspicion. Mostly this is because attempts in the 1980s were not very successful, because computer graphics were so poor then (by modern standards). It may be that some people also felt that the takeover of computer games by graphical interfaces was the death knell of IF. But pictures are now rendered in superb quality by computers, and the death of IF turned out to be an exaggeration, so it is time to move on.

Whether to have illustrations ought to be an artistic choice, like whether to include a romantic sub-plot or how much of the back story is revealed. But there are practical considerations too. The most successful illustrated books are those whose pictures are well-chosen, have a sense of design to them, and above all are consistent. Consider how much worse off *Winnie the Pooh* would have been if a selection of random teddy-bear drawings had been used, instead of E. H. Shepherd's beautifully conceived world; or a cookery book in which the recipes are all photographed at different distances and light levels. IF writers may want to look for collaborators with a visual eye, just as most novelists do not draw their own illustrations.

Another consideration is that displaying images is more complicated for computers than displaying text. Not all devices can show pictures (consider handheld gadgets) and if they do, they may use different colour ranges or resolutions. So IF with pictures is always just a bit less portable than IF without, and because of that we must next look again at IF story file formats.

§23.3 Virtual machines and story file formats

Back in Chapter 2, we saw that the Settings panel allows any given Inform project to be produced in either of two possible story file formats. Recall that story files are the released IF works: what the player sees. The source text, the Index, and so on are not part of this.

A story file is not like a word-processed document, or a photograph. There are many rival formats for these - for instance, an image on a web page might be in JPEG or PNG format, among many others - but basically they are simple things for the reader to look at, and see everything in one go. An IF story file is more complicated, because the "reader" reacts to it, types in to it, is surprised by it, never quite knows what might happen next.

A story file is in fact a computer program in its own right, but not a program like iTunes or Firefox which runs on a typical home or business computer. Instead it is a program for an imaginary computer, called a "virtual machine" or "VM". This has a design ideally suited to IF, and it would be the perfect IF player's computer if only it actually existed. Because it doesn't, the player instead runs an "interpreter" program like Windows Glulxe or Zoom or Spatterlight - and this one is a program like iTunes or Firefox - and the interpreter acts as a middle-man. It creates a software version of the virtual machine, and then runs the IF story file on that VM. This sounds slow and impractical, but in fact it works well, and is also much safer since programs on the VM are not allowed to touch the real computer - so they cannot at all easily contain viruses or other malware. (In theory a malicious story file might try to exploit a bug in one of the various VM implementations in use, just as malicious image files have been used to attack bugs in web browsers, but this has never in practice happened. Nothing can be absolutely safe, but a story file belongs in the

"mostly harmless" category of files - like images - rather than the "how far do you trust this person?" category - like programs.)

The different formats of story file are programs for different virtual machines. Just as Windows and Mac OS X offer basically similar services to the user but are very different in appearance and their workings, so the different VMs are quite different. Some can display pictures, others not.

§23.4 Gathering the figures

Inform provides basic support for displaying pictures and leaves more exotic effects for Extensions to provide. But either way, for reasons explained in the previous section, **we can only have pictures if the Settings for the project are set to the Glulx story file format**.

Inform calls these pictures "figures", following the usual Inform analogy with books. We will think of our work of IF as being like a mostly textual book which in broken up with illustrations here and there - Figure 1, Figure 2, and so on. These might be used to mark each new chapter of the plot, or each new location: whatever the author would like. So the first thing we need to do is decide when pictures should appear.

The second thing to do is to get hold of the pictures we want to use. These might be photographs, or artwork, or diagrams: anything, really, but we will need them to be in either JPEG or PNG format. Inform does not itself try to be an image editor, or an artwork program - there are many such programs already which do these things much better than Inform could.

The pictures then need to be put in a special place where Inform can reach them. Suppose the Inform project is called Example.inform. Then we need to create a folder alongside it called "Example.materials", and create a further folder inside that called "Figures". The actual images go inside "Figures". So we might then have files like so:

```
Example.inform
Example.materials
    Figures
        Woodlands.png
        Blackberry.jpg
        Red Admiral Butterfly.png
```

The ".materials" folder for an Inform project will turn out to have many other uses in the chapter on Publishing, and will be explained further there.

§23.5 Declaring and previewing the figures

Inside Inform, the source text for a project always tries to avoid talking about filenames -

we need a better way to refer to the individual figures.

We do this by declaring each figure with a sentence like the following examples:

```
Figure of Woodlands is the file "Woodlands.png".
Figure 2 is the file "Red Admiral Butterfly.png".
```

Figure names can consist of any text provided that text starts with the word "Figure". So "Figure 3 - Woodlands", for instance, or even "Figure W" would have been just as good as "Figure of Woodlands". Books tend to number figures, but then, in a book the order in which they appear is known in advance - which might not be true in IF.

The file names must be exactly those used in the Figures folder. We need not declare every image kept there, but those we don't declare - remember Blackberry.jpg? - cannot be displayed.

We can preview the stock of figures by going to the table of figures in the Contents index for a project (once the project has been built, that is, so that its index is up to date). This preview shows thumbnail forms of the pictures, the names, the formats and the image sizes in pixels. A warning triangle is shown for any images in the wrong format, or which are missing from the Figures folder.

§23.6 Displaying the figures

Inform's basic picture support simply allows figures to be shown at particular times. Once seen, they scroll away, just as text does once it has been printed. These pictures are really part of the stream of narrative. (If we would like icons or other images to be permanently present on screen, and divide the screen up in pictorial ways to achieve interesting layouts, we need to use special extensions to access Glulx's more exotic features.)

Displaying a picture is therefore like printing some text. Rather than

```
say "The woodlands stretch from here to the horizon.";
```

we would use:

display (figure name)

This phrase causes the figure to be displayed in a way visible to the player. If the option "one time only" is used, it will have no effect if the figure has been displayed before. Example:

```
display the Figure of Woodlands;
```

Once again, note that the "display" phrase does nothing unless the Settings for the project are set to the Glulx story file format. When a Glulx work is released as a blorb (the default setting for the way releases occur: see the chapter on Publishing), all the images used are automatically included.

§23.7 Recorded sounds

Inform also supports the playing back of recorded sounds, which might be anything from a three-second sound effect for a creaking door to an epic orchestral symphony. **Sound support is very newly added to the system and work is still in progress. In particular, sounds are not played by Inform for OS X (although it does produce valid blorbed Glulx story files), though they should be audible from within the Inform application for Windows.**

Once again, sound effects are supported by Inform 7 only on the Glulx platform, and even then we must be prepared for the fact that not all interpreters will be able to play them back. We must also bear in mind that a sound recording is a large pile of bits, and that adding any kind of sounds will greatly increase the size of the Blorb file for the released Glulx story file.

The sound files provided must have one of two formats: AIFF or Ogg Vorbis. **AIFF** is a traditional format in the recording industry, though it is more familiar to Mac OS X users than Windows users. It is uncompressed, giving what can be excellent audio quality, but at the cost of sometimes enormous file sizes - perhaps as much as 10 MB per minute, though this can be greatly reduced by lowering the sampling frequency, and halved again by dropping from stereo to mono.

Except for very short sound effects, we recommend using **Ogg Vorbis** instead. This is a compressed format whose file sizes will typically be more like 1 MB per minute. Inform uses Ogg Vorbis as the only format safe from licencing and patent disputes. (We would very much have liked to provide MP3 support, but this is no longer legally possible for free software.)

Support for Ogg Vorbis is not built in to either Windows or Mac OS X, and any sound recording you make will probably have to be made first to another format (perhaps AIFF or WAV), and then converted. See **xiph.org/vorbis** for encoding software which can convert from other sound formats to Vorbis.

Lastly, it must be remembered that recording industry bodies are very hostile to established copyright law covering fair use, parody, quotation of insubstantial passages, etc., when it comes to mixing or using commercially released music. They are well-resourced and highly litigious. If you use sound effects not originated by yourself, you do so at your own risk, even if what you do is perfectly legal on any reading of the statutes.

§23.8 Declaring and playing back sounds

Sound effects are accommodated on the same basis as illustrations. The relevant media files need to be placed in a subfolder of the project's ".materials" folder, but this time called Sounds rather than Figures, so for instance:

```
Example.inform
Example.materials
    Figures
        Woodlands.png
        Blackberry.jpg
        Red Admiral Butterfly.png
    Sounds
        Rustling leaves.ogg
```

Again, these must be declared in the source text:

```
Sound of rustling leaves is the file "Rustling leaves.ogg".
```

And they can be played using a special phrase:

play (sound name)

This phrase causes the sound effect to be played. If the option "one time only" is used, it will have no effect if the sound effect has been played before. Example:

```
play the sound of rustling leaves;
```

§23.9 Providing accessibility text

It's conventional for web pages to provide "alt-text" for significant images displayed, so that partially sighted or blind users can get an idea of what is being shown. Inform allows figures to be given these short descriptions like so:

```
Figure 2 is the file "butterfly.jpg" ("A red admiral butterfly.").
```

As we'll see, the same can be done for the cover image:

```
Release along with cover art ("A cathedral at sunset.").
```

And also for sounds:

```
Fugue is the file "Bach.ogg" ("A church organ playing a Bach fugue.").
```

§23.10 Some technicalities about figures and sounds

(i) Names for figures, such as "Figure of Woodlands", are values for a special kind of value called "figure name". This can in turn be used to define variables, properties and phrases:

```
The turn card image is a figure name that varies.

An Old Master is a kind of thing. An Old Master has a figure name called appearance. Figure 1 is the file "Giaconda.jpg". The Mona Lisa is an Old Master. The appearance of the Mona Lisa is Figure 1.

To place (F - a figure name) in the gallery: ...
```

(ii) Similarly, names for sound effects, such as "Sound of rustling leaves", are values for the kind of value "sound name".

(iii) In the released, blorbed-up Glulx file, figures and sound effects are internally given resource ID numbers which count upwards from 2 in order of their declaration. (Figure and sound numbers can thus be intermingled, if their declarations are.) Resource ID number 1 is reserved for the image of the cover art, if there is any. (See the chapter on Publishing.) To obtain these numbers, if we need them, we can use:

Glulx resource ID of (figure name) ⇒ *number*

This phrase produces the ID number used in the eventual Glulx file for the given figure.

Glulx resource ID of (sound name) ⇒ *number*

This phrase produces the ID number used in the eventual Glulx file for the given sound effect.

(iv) Glulx hackers may also like to know that whenever Inform 7 builds a project for Glulx, the Inform 6 code it generates always contains a full copy of John Cater's definitive header file "infglk.h".

§23.11 Files

Once an Inform project is released, it is playable as a "story file", which is in effect a computer program for a specially IF-adapted design of computer. Story files run in what in computing is sometimes called a "sandbox", a kind of safe play area where it can be guaranteed that they cannot do any harm. This is good, because it means a story file can't be infected with viruses or other malware. If the project's Settings panel has the story file format set to the Z-machine, the story file is so thoroughly boxed in that it cannot even see the bigger computer beyond: it lives in a world of its own. But the Glulx format opens the door a crack, allowing the story file to read and write a small number of data files, which live in a single folder on the bigger computer's hard drive.

Why might we want this? Among the reasons are -

```
- to remember what has happened in previous attempts by the player;
- to store the player's preferences;
- in a two-part story, where each part is an independently released story file, to allow Part I to save some information about its ending which Part II could then pick up and make use of;
- to communicate with some external program, such as an Internet service.
```

§23.12 Declaring files

Like figures and sounds, files must be declared before they can be used. For instance:

```
The File of Glaciers is called "ice".
```

This creates a new named constant "File of Glaciers" to refer to the file, throughout the source text. We use this name for it whether or not the actual disc file exists yet: it might be one that will only be created if something unusual happens in play, for instance.

Quoted filenames should contain only letters and digits, should be 23 characters or fewer,

and should begin with a letter. (In particular they can contain no slashes or dots - no subfolders or extensions can be indicated.) The actual filename this translates to will vary from platform to platform, but "ice.glkdata" is typical, stored in some sensible folder.

Every file has an owner - not a person, but the project which normally writes to it. Inform assumes that the current project will be owning any file which it declares - the File of Glaciers, for instance. But we can optionally specify that it is owned by somebody else:

```
The file of Boundaries (owned by another project) is called "milnor".
The file of Spectral Sequences (owned by project
"4122DDA8-A153-46BC-8F57-42220F9D8795") is called "adams".
```

Inform uses ownership to make sure that we do not accidentally read in a file which has nothing to do with us, but merely happens to use the same name. Thus it is an error to read a file whose ownership does not agree with our declaration. Saying that a file is owned by "another project" allows us to read it whatever the owner is (so this can be used for files shared between multiple projects); specifying exactly where it needs to come from allows us to pass information from one project to another. Note that we identify projects using the IFID number - this can be found in the Contents index for a project, or by typing VERSION during play; see the chapter on Publishing for more about IFIDs.

Files are indexed in the Contents index, alongside figures and sound effects.

Two technicalities. First, constants such as "File of Glaciers" are of a kind of value called "external file" (compare "figure name" and "sound name"). Second, Inform's file-handling is provided for the Glulx machine, which in turn uses the Glk interface. This allows for either text or binary files. Inform's higher-level phrases to do with files, described in this chapter, all use text files, and all declared files are text files by default. But we can optionally add the keyword "binary" to declare a binary file, if needed:

```
The binary File of Glaciation Data is called "icedata".
```

§23.13 Writing and reading tables to external files

The main use for files is to store and retrieve data, and the most flexible form of data used by Inform is the Table, so facilities are provided which make it as easy as possible to write and read the contents of a table to files. If so, the file must contain just one single table: so to write multiple tables, we need to write multiple files, one for each.

To save the contents of a table to a file, we use the phrase:

write (external file) **from** (table name)

This phrase causes the entire contents of the given table to be written out to the given file. Note that files must have been declared, and must be referred to by their Inform names, not by textual filenames. Example:

```
write File of Glaciation Data from the Table of Antarctic Reserves
```

Any blank rows in the table are automatically moved to the bottom, and only the non-blank rows are written.

To load a file back into a table,

read (external file) **into** (table name)

This phrase causes the entire contents of the given table to be read in from the given file. Note that files must have been declared, and must be referred to by their Inform names, not by textual filenames. Example:

```
read File of Glaciation Data into the Table of Antarctic Reserves
```

Any rows left spare at the foot of the table are automatically blanked. On the other hand if the file is too large to fit into the table - with too many columns or too many rows - a run-time problem is produced.

We can check if a file already exists using:

if (external file) **exists:**

This condition is true if the file-system used by the player appears to contain a file with the right name. For example, if we declared:

```
The binary File of Glaciation Data is called "icedata".
```

and then tested

```
if the File of Glaciation Data exists, ...
```

then Inform would search for a file called "icedata". (The arrangements for where this might be stored, and its filename extension, vary from platform to platform.)

One unfortunate restriction must be kept in mind. Some of what is stored in tables is solid information whose meaning never changes: the number 342, for instance, means the same to everyone. But other information depends entirely on the current location of certain structures in memory - for instance, a rule is internally referred to by its memory location. This potentially changes each time Go or Replay is clicked, and so it is not safe to pass it from one copy to another, or from one project to another. The only tables which Inform allows us to write into files are those containing "safe" data: numbers, units, times of day and kinds of value with named alternatives. Scenes, rules or rulebooks, in particular, are not allowed.

Examples

439. **Alien Invasion Part 23**
Keeping a preference file that could be loaded by any game in a series.
RB 11.1 Start-Up Features

440. **Labyrinth of Ghosts**
Remembering the fates of all previous explorers of the labyrinth.
RB 11.6 Ending The Story

441. **Rubies**
A scoreboard that keeps track of the ten highest-scoring players from one playthrough to the next, adding the player's name if he has done well enough.
RB 11.4 Scoring

§23.14 Writing, reading and appending text to files

Text can also be saved to a file, and again all file-handling is automatic:

write (text) **to** (external file)

This phrase makes the given text become the entire contents of the named file. Note that files must have been declared, and must be referred to by their Inform names, not by textual filenames. Example:

```
write "Jackdaws love my big sphinx of quartz." to the file of Abecedary Wisdom;
```

append (text) **to** (external file)

This phrase adds the given text to the end of the current contents of the named file (creating it if it does not exist on disc). Note that files must have been declared, and must be referred to by their Inform names, not by textual filenames. Example:

```
append "Jinxed wizards pluck ivy from the big quilt." to the file of Abecedary
Wisdom;
```

The quoted text can, of course, contain substitutions, so can be long and complex if need be.

Text from a file is printed back with the text substitution:

say "[text of (external file)**]"**

This text expands to the contents of the named file. Note that files must have been declared, and must be referred to by their Inform names, not by textual filenames. Example:

```
"[text of the File of Abecedary Wisdom]"
```

To copy one file to another, for instance,

```
write "[text of the file of Abecedary Wisdom]" to the file of Secondary Wisdom;
```

Examples

442. **The Fourth Body**
    Notebooks in which the player can record assorted notes throughout play.
    RB 9.6 Reading Matter

443. **The Fifth Body**
    An expansion on the notebook, allowing the player somewhat more room in which to type his recorded remark.
    RB 9.6 Reading Matter

§23.15 Exchanging files with other programs

Provided we declare the files in the right way, it is easy for one project to read a file created by another project.

But if we want more rapid communication, between two projects which are each playing at the same time, we need to be more careful. What if project A tries to read the file at the same moment that project B is writing it?

To avoid this, we have a concept of files being "ready". A file is ready if it exists, and is completely written, and not in use elsewhere. We have already seen:

```
if the file of Invariants exists...
```

But now we want a stronger condition:

if ready to read (external file):

This condition is true if the file exists and is marked as being ready to read; that is, it is not in a state where another program is currently writing it. Example:

```
if ready to read the file of Invariants, ...
```

A file cannot be ready to read if it does not exist, so this is a stronger condition. If A and B are attempting communication in real time, both running at once, then Project A should check that an external file owned by B is ready before it tries to read it. Files can also be marked as ready or not ready, in effect claiming them, thus:

mark (external file) **as ready to read**

This phrase marks that we have finished writing to the given file, so that any external program is welcome to read it now. Example:

```
mark the file of Invariants as ready to read;
```

mark (external file) **as not ready to read**

This phrase marks that we are about to start writing to the given file, so that any external program should wait until we're finished if it wants to read the file. Example:

```
mark the file of Invariants as not ready to read;
```

Possibilities really begin to open up when project A is our story file, but B is not another story file at all: it is some external program such as a Web service, say. (Of course this is harder to set up, since the player needs to have both A and B running at the same time, but for stories running on an Internet server this can all be made seamless.)

When Inform begins writing a table, or text, to a file, it initially marks the file as not ready: only when the table or text is completely written and the file about to close is the file marked as ready.

In order to write non-story-file programs as B, communicating with story files as A, we need to know the file format used by Inform. An Inform file is currently a Unix text file (with 10 as the line division character), encoded as ASCII Latin-1. (We would like to use Unicode at some point in the future, but the Glk and Glulx layers are still not fully converted to Unicode.) It opens with a single header line in the form:

```
* //IFID// leafname
```

The opening character is an asterisk if the file is currently ready, a hyphen if the file is currently not ready. The IFID between the slashes is the IFID number of the project which last wrote to the file. (Marking "ready" or "not ready" does not count as a write for this purpose.) If an external program wrote the file, it should call itself something which will not clash with any story file's IFID. The leafname is the filename text used inside the story file where the file was declared. For instance:

```
* //4122DDA8-A153-46BC-8F57-42220F9D8795// ice
```

Example

444. Flathead News Network
Using external files, together with a simple Unix script running in the background, to provide live news headlines inside a story file.
RB 12.5 Glulx Multimedia Effects

24. Testing and Debugging

§24.1 Checking against the Index
§24.2 Debugging features to use in source
§24.3 High-level debugging commands
§24.4 Low-level debugging commands
§24.5 Adding new testing verbs and Release for Testing
§24.6 Testing for thoroughness
§24.7 Commands for beta-testers
§24.8 Help from the user community

§24.1 Checking against the Index

Testing a story -- and indeed writing a story so that it is easy to test consistently -- is an art in itself. We should expect that we'll do some preliminary testing, both by running test commands and by playing through the story ourselves, and that we'll then hand on the story to beta-testers who will tell us about faults in the play experience that we haven't been able to see.

Every time Inform builds a new story file, it assembles a vast amount of information about that world, in the form of the Index. Often a visit to the Index is all that's needed to explain a piece of undesired behavior.

Is travel not working as it should? Check the World index and see whether the map shows the rooms arranged the way you thought.

Are objects not showing the behavior you'd expect based on their kind? Check the Kinds index and make sure they've been defined as the kind of thing you expected. For instance, we might find that we've written

```
The red door is west of Foo and east of Bar.
```

but not

```
The red door is a door.
```

A human reader wouldn't make this mistake, but Inform hasn't actually registered the red door as belonging to the door kind, and consequently has treated it as a room instead. All we need to do is add the kind declaration. The Kinds index will make that obvious.

When an error appears in the Index, there is often a link back to the source text that

defined that room or object. If not, there's often at least some information about what rule or phrase might be responsible for it.

§24.2 Debugging features to use in source

The TEST command is an extremely useful way of managing a story and continuing to verify that it does everything we want. We can create new test commands of the form

```
Test me with "up / kill captain eo".
Test eo with "zap eo" holding the ray gun.
Test dinner with "eat bread / eat soup / eat butter" in the Ship Cafeteria.
```

and we are free to have as many of these tests as we would like. Test commands can call other tests, as well, so we might have a test command such as

```
Test megatest with "test me / test eo".
```

A word of warning: if the first command in the test is "again", that will likely repeat the TEST command, sending Inform round in circles forever.

For complicated objects and commands, sometimes it's a good idea to develop the test commands at the same time that we're writing the source code itself. Each time we add a new rule or piece of behavior, we also add to that object's special test command something that will put that new feature to the test. This means that we can keep running the test command as we work and verify that everything is behaving as expected.

Sometimes we need to get a look at what is happening within the source itself. Many of the most annoying bugs come about because we're making some assumptions about what's true in the story world that differ from Inform's assumptions. When that happens, we may need to add something to the source to check that the variables are set to what we think, that certain parts of the source are being reached, and so on.

For instance, suppose we have a phrase like this:

```
To say key score:
    let count be the number of keys which are not carried by the player;
    if count is greater than 2 and the player is timid:
        say "You're still missing a lot of keys, bucko!"
```

Now, we expect this to print something, but perhaps it's not doing so when we had anticipated that it would. At some point when we think the count is greater than 2 and the player is timid, at least one of those things is not true. An easy way to check up on this

is to add a showme line to the source, like so:

```
To say key score:
    let count be the number of keys which are not carried by the player;
    showme count;
    if count is greater than 2 and the player is timid:
        say "You're still missing a lot of keys, bucko!"
```

and this will then check the relevant number and print it to screen when this phrase is called, like so

```
"count" = number: 1
```

In this case, it looks like the count is not high enough to trigger the text, so we can concentrate on working out why that might be. Maybe we didn't correctly define something as a key, for instance.

§24.3 High-level debugging commands

If an object is not responding in the way we expect, it may be that we're wrong about where it is or about some of its current properties or relations. We can find our current location and the things around us by typing

```
>SHOWME
Boudoir - room
    four-poster bed - supporter
    yourself - person
    pillow
```

and similarly we can inquire about the status of a particular object during play by typing SHOWME and the object's name:

```
>SHOWME BAT
bat - thing
location: on the table in Locker Room
singular-named, improper-named; unlit, inedible, portable, patterned
printed name: "bat"
printed plural name: none
indefinite article: none
description: none
initial appearance: none
```

This will work even if we're not in the same location as the object we want shown.

Another common type of problem is one in which we type a command but Inform does not perform the action that we were expecting as a result. In some cases, this is because the command we're typing is actually triggering some other action. An easy way to check on this is to type ACTIONS before issuing the command that is behaving unsatisfactorily. Thus:

```
>ACTIONS
Actions listing on.

>JUMP
[jumping]
You jump on the spot.
[jumping - succeeded]
```

This tells us how Inform interpreted our input and whether the action was successful or failed for some reason. If the command is being understood as a different command than we expected, that may mean that we have made a mistake in our Understand instructions, and need to double-check these.

Sometimes, however, the action is being correctly understood, but the action rules that are firing are producing a result other than we'd like. If we want to see which rules are running, we can type

```
>RULES
Rules tracing now switched on. Type "rules off" to switch it off again, or "rules all" to include even rules which do not apply.

>JUMP
[Rule "announce items from multiple object lists rule" applies.]
[Rule "set pronouns from items from multiple object lists rule" applies.]
[Rule "before stage rule" applies.]
[Rule "instead stage rule" applies.]
[Rule "investigate player's awareness before action rule" applies.]
[Rule "player aware of his own actions rule" applies.]
[Rule "check stage rule" applies.]
[Rule "carry out stage rule" applies.]
[Rule "after stage rule" applies.]
[Rule "investigate player's awareness after action rule" applies.]
[Rule "report stage rule" applies.]
[Rule "report jumping rule" applies.]
You jump on the spot.
[Rule "last specific action-processing rule" applies.]

[Rule "A first turn sequence rule" applies.]
[Rule "every turn stage rule" applies.]
[Rule "A last turn sequence rule" applies.]
[Rule "notify score changes rule" applies.]
>
```

As we can see, RULES produces a lot of output, much of which is probably irrelevant to whatever problem we're tracking down. Nonetheless, knowing exactly which rule is printing undesirable output is helpful, especially if that rule comes out of an extension or some other source that we did not write ourselves: this output has told us that the text we saw came from the "report jumping rule".

To find out more about what is going on in specific rules, we can also turn to the Index tab under Actions and click through to that specific action. From there we will be able to see which rules are included, what responses they're writing, and where they were defined in the source text.

SCENES lists which scenes are currently playing and which are complete. This is valuable if scene-triggered events are not happening when we expect them to.

RANDOM sets the random number generator to a predictable seed. If we include this in a test command, it will guarantee that the subsequent behavior of the story is consistent across multiple playthroughs, which is helpful if we're trying to test something to do with, say, randomly wandering non-player characters.

RELATIONS lists all the relations defined in the story, except for things like support and containment that are part of the world model and are so numerous that the output would be overwhelming.

RESPONSES lists all the named responses established by all the extensions currently included. This can be informative, or it can be a bit overwhelming. Except where responses have been changed at runtime, the same information is available in a different form in the Index on Actions. If we're interested in a particular single response, digging into the actions index is probably the easiest way to find it.

If, however, we want a rapid overview of all the responses provided by a given extension (perhaps an extension we are ourselves writing), the RESPONSES command can be a help.

§24.4 Low-level debugging commands

There are also several debugging commands going back to the early days of interactive fiction, and relating in a simple way to objects and places. These can still come in handy for a quick and dirty resolution of a problem during gameplay, and are as follows.

PURLOIN moves an object to your possession, no matter where it is on the map, like so:

```
>PURLOIN TABLE
[Purloined.]

>I
You are carrying:
a table
```

Note that purloin does not consider the usual rules about whether something can be taken. In this case, we've just moved the table to our inventory even though it is a fixed in place supporter that could not be taken in the normal course of events.

Because purloin works on things that are far away as well as things that are close, it has to do a lot of extra parsing work and may take a moment or two to complete if we try it in a very large story. It is generally more efficient to give the player the relevant object using a testing command, like this:

```
Test me with "drop table" holding the table.
```

Nonetheless, there are occasionally times when we're halfway into a 2000-move story and suddenly realize we implemented a vital object in the wrong room, making the story unsolvable. We could fix the bug, press replay and return to this story state fairly quickly,

but if we don't feel like waiting even that long, PURLOIN will resolve the issue.

ABSTRACT is PURLOIN's less useful cousin, allowing the player to move an object from one place to a specified other place, as in

```
Bar
You can see a table here.

>ABSTRACT KEY TO TABLE
[Abstracted.]

>LOOK
Bar
You can see a table (on which is a key) here.
```

GONEAR transports the player instantly to the vicinity of the named object, so for instance

```
>GONEAR GRAIN
Fertile Plain
You can see some grain here.
```

As a debugging command, this isn't protected in the ways that commands usually are. It's possible to type GONEAR NORTH and produce a run-time error when Inform tries to move the player into the object that represents the compass. Again, except in cases where we're tracing a problem very deep in an already running story, it is usually more practical to write a test command to put the player in the correct situation, as in

```
Test me with "eat grain" in the Fertile Plain.
```

VERIFY checks that the story file is intact rather than damaged, but it is hard to think of an occasion when this would be likely to arise within the Inform application. The command is a holdover from a time when data transfer was much slower and more error-prone, and it was plausible to have a story file of just a few hundred KB corrupted during transmission.

TREE creates a list of object containment. It is similar to SHOWME, but less elegant and thorough.

SCOPE lists the objects that are currently in scope for the player, which is to say, things that could be referred to when we're typing a typical command. Thus:

```
Bar
You can see a table here.

>SCOPE
1: yourself (574631)
2: a table (574759)
```

The following numbers are object IDs for these objects, which can distinguish items with identical names. It is likely that the output of this will not be terribly interesting or different from checking SHOWME, except in cases where the author is deliberately changing the scope to be something other than "the set of things that are visible in the room with the player right now". This usually involves the Deciding the scope of something activity (see the chapter on Activities).

SHOWHEAP shows how many bytes are currently free. This is usually not helpful.

SHOWVERB (verbname) lists the Understand information associated with a particular verb. Similar information, in a vastly more palatable form, is available in Index / Actions / Commands, so the one time SHOWVERB becomes useful is when Inform is considering the understand lines in the wrong order and producing a result we didn't want: SHOWVERB will show us the order in which the lines are being assessed. The challenge will then be to add conditions to the Understand lines to move them into the correct order.

Finally, TRACE (and its more advanced stages TRACE 2, TRACE 3, TRACE 4, and TRACE 5) will reveal things, more things than we ever wanted to know, about the assumptions being made by the parser when it takes in a command. In practice this information is almost never useful to an Inform 7 author.

There is no guarantee that any of these commands will make life better or that they won't crash the story or put it into an unwinnable state. There is also no absolute guarantee that they won't be withdrawn entirely from future versions of Inform. Consider them as Old High Magic, and treat accordingly.

§24.5 Adding new testing verbs and Release for Testing

As we saw in Chapter 2, we can mark some of our source text so that it will not be included in a finished story. This means that we can add special testing commands available to the author but not available to our eventual players. This is a good way to add our own suite of testing verbs to a story beyond the "Test me with…" features already described.

Here are some types of testing verbs that can be useful to add:

Chapter jumps. We might create test commands that took us to a later stage of the story (perhaps doing more setup than "Test me…" alone can handle).

Status information. We might create a test command that would show us status information beyond what's covered in the Standard Rules. For instance, if we had a story that heavily modeled the moods of other characters and we wanted to be able to check those moods at any time, we might add a SHOWMOOD command that would tell us about a character's emotional state.

Puzzle satisfaction lists. Some simulation-rich stories offer puzzles that can be solved in a variety of ways: for instance, a sealed glass box that can be smashed with any object that has been marked with the properties "hard" and "heavy". Later, we might want to be able to check which in-story objects would work as a solution to this puzzle, so we might create a command like

```
Listing hammers is an action out of world applying to nothing.

Understand "list hammers" as listing hammers.

Carry out listing hammers:
    say "These things can break the glass: [line break]";
    repeat with item running through portable hard heavy things:
        say "[item][line break]";
```

so that we can review that there are enough objects available and that the list doesn't include anything it shouldn't. In a small story this kind of thing is pretty easy to keep track of in the author's head. Large stories can contain thousands of objects, however, at which point it becomes valuable to have an automated method of verification.

Just occasionally, we might also want to build a version of a story that will allow beta-testers access to the debugging commands. This is especially relevant for long stories: if we're testing a story with a lot of playtime and the testers have already thoroughly reviewed the first portion of the story, we might want to let them have access to testing commands that fast-forward to later sections.

To do this, we can use the "Release for Testing" feature. Release for testing builds a version of the story that *does* include testing commands and any sections labeled "Not for release".

§24.6 Testing for thoroughness

The presence of actual bugs or defects is not the only thing we want to consider when testing a story. We may also want to check whether we have built the story with a consistent amount of depth.

Are there descriptions for everything the player might look at? If we've implemented special verbs, do they have appropriate reactions for all the different objects? If most objects in a story about restaurant reviewing have a special response to being tasted, for instance, it might be disappointing for the player to encounter late-added objects that don't.

Checking implementation thoroughness can be a laborious process, but there are a few things we can do to automate it. For instance, we might add to a not-for-release section a rule that checks for certain properties:

```
When play begins (this is the run property checks at the start of play rule):
    repeat with item running through things:
        if description of the item is "":
            say "[item] has no description."
```

This will confront us with a reminder of what we still need to fill in every time we start up the story.

There are also some extensions that are designed to assist with this, notably the massive Object Response Tests by Juhana Leinonen. Object Response Tests allows us to try out a long list of commands against any object in the story, so that we can quickly identify ones with nonsensical replies.

§24.7 Commands for beta-testers

Inform includes a command that is especially designed to help beta-testers report flaws: namely, TRANSCRIPT. A tester can type TRANSCRIPT (or just SCRIPT) at the beginning of the story in order to start generating a recording of everything that happens. She can then add her own annotations when something buggy or otherwise notable occurs (for instance by typing a standard symbol, such as *, followed by a note).

When she then sends us the completed transcript, we can look through for these symbols and note the problems the tester found in the context of the rest of the story's behavior. Having information about how she reached that position typically makes it much easier to reproduce the problem than if she gave only a general account of it.

§24.8 Help from the user community

Sometimes we get really stuck on a problem and despite all our best efforts cannot figure out how to solve it.

Fortunately, Inform has a lively and helpful community of users who are often willing to assist other authors. The easiest way to reach these users is to make a post at the

intfiction forum at

```
https://intfiction.org/
```

and in particular to post Inform-related problems under the topic Inform 7 Development. Where possible, it's a good idea to post the example source that is causing trouble, and to make it as short as possible so that prospective helpers will not have to read any more than necessary in order to pinpoint the problem.

The user community is also a good place to find beta-testers who can try out our work and give feedback.

25. Releasing

§25.1 The finished product
§25.2 Bibliographic data
§25.3 Genres
§25.4 The Library Card
§25.5 The Treaty of Babel and the IFID
§25.6 The Release button and the Materials folder
§25.7 The Joy of Feelies
§25.8 Cover art
§25.9 An introductory booklet and postcard
§25.10 A website
§25.11 A playable web page
§25.12 Using Inform with Vorple
§25.13 Website templates
§25.14 Advanced website templates
§25.15 Republishing existing works of IF
§25.16 Walkthrough solutions
§25.17 Releasing the source text
§25.18 Improving the index map
§25.19 Producing an EPS format map
§25.20 Settings in the map-maker
§25.21 Table of map-maker settings
§25.22 Kinds of value accepted by the map-maker
§25.23 Titling and abbreviation
§25.24 Rubrics

§25.1 The finished product

This chapter and the next are about what to do when we have a complete, finished work on our hands.

For almost all of the time when a new work of IF is being written, it lives inside the familiar two-panel spread of the Inform user interface. But that isn't how eventual players will experience it. They will want to play a "story file" in a standard format, and they will do so with a wide range of different interpreters on many different computers or websites, including some -- like mobile phones -- on which Inform itself will not run.

So how does a new work of IF reach players? The simple answer, covered in this chapter, is that clicking the Release button instead of Go causes Inform to output a stand-alone story file. But as we will see, Release can do much more than that: it can attach covers,

include bibliographic data, make websites and much more. Releasing is the process of making all of the material we want to deliver to our eventual players.

But that is only the first step. What do we do with the material when we have it? Printing out a manuscript of a novel is not the same as publishing it. So the next chapter, on Publishing, completes the story.

§25.2 Bibliographic data

Almost all printed books have a title page and a so-called "imprint" page, often its verso, which make up a description of the contents. The title page gives the name of the book and of the author, while an imprint page contains a variety of details about the edition, the printing, and so on. An ISBN number is allocated so that, from the number alone, any book seller or cataloguer can identify exactly this work. Sometimes other cataloguing information is added, such as the Library of Congress classification. This set of information is called "bibliographic data", and without it libraries and booksellers would be at a total loss.

IF has bibliographic data, too. Inform has a number of special named values to hold this - who wrote the work being created, what it is called, what headline it has, what genre it has and what its release number is, and so on.

These can be set as follows:

```
The story title is "Mansfield Perk".
The story author is "Janet Austen".
The story headline is "An Interactive Romance".
The story genre is "Romance".
The release number is 7.
The story description is "In Miss Austen's new interactive novella, Miss Henrietta Pollifax is adopted by the tempestuous landowner Sir Tankerley Mordant, and must make a new life for herself on the rugged moors."
The story creation year is 2005.
```

Most of these are self-explanatory. The "story creation year" is provided so that if we need to revise the work to fix some bugs a year later - by no means an uncommon occurrence - then we can make sure it is correctly identified as still being basically a 2005 work. (Just as a book which has had innumerable revised printings may say "First published 1988" on its imprint page.) The "story description" is a piece of text, analogous to the back cover blurb on a book: it might be two or three paragraphs long, so the example above is rather minimal, but it should not be epic in length.

As we have already seen, a convenient abbreviation provides that if the first sentence of

the source text consists solely of text in quotation marks, then that is considered the title. Thus if the source begins:

```
"Mansfield Perk"
```

then that will be the "story title". Further, we can write

```
"Mansfield Perk" by Janet Austen
```

with the obvious effect: quotation marks around the author's name are optional here, for convenience, but note that we'd better have them in cases like:

```
"Three Men in a Boat" by "Jerome K. Jerome"
```

as otherwise the full stop after the K will end the sentence prematurely.

The text of these bibliographic descriptions cannot normally include text substitutions, since they are written into external descriptions of the story file as part of its "binding". Two exceptions are allowed, though: "[']" makes a literal apostrophe, and can be used if we need to override Inform's normal conventions to do with converting apostrophes at the ends of words to double-quotes. For instance:

```
"Summer of [']69" by Buzz Aldrin
```

The other exception is that the "[unicode ...]" text substitution works, so for example:

```
The story description is "This is a sentence[unicode 8212]with a parenthetical in
dashes[unicode 8212]because 8212 is the Unicode number for an em-dash. But for
example, 'pawn to [unicode black chess bishop]4' draws in a black chess bishop, so it
works with names, too."
```

If the bibliographic named values are not set by the source text, Inform will still need to say something. Unset text and number variables evaluate to "" and 0 respectively, but this would make for a very unhelpful record. So Inform uses the following table instead of any value which is unset:

```
Story title: Untitled
Story author: Anonymous
Story headline: An Interactive Fiction
Story genre: Fiction
Release number: 1
```

§25.3 Genres

The "story genre" is not used in the banner at all, and exists purely to help librarians. If it is at all possible to do so, authors are asked to use one of the following standard categories:

```
Comedy, Erotica, Fairy Tale, Fantasy, Fiction, Historical, Horror, Mystery, Non-Fiction,
Other, Romance, Science Fiction, Surreal
```

These categories are based on those currently used by bookshops, but a few notes may be helpful. "Fiction" is intended for works whose essential purpose is literary, in a way which trumps any subject they happen to have: if Julian Barnes writes a mystery, for instance, a bookshop will shelve it with modern novels rather than in the detective stories section, whereas P. D. James's Adam Dalgliesh mysteries will end up filed with detective fiction even though she has appreciable claims to be an important novelist.

"Comedy" is used rather than "humour" to avoid the clash of spellings with "humor". This genre includes parodies.

"Non-Fiction" would be used for a work of IF which is essentially a presentation, perhaps in a novel interactive format, of true information. A meticulous simulation of the Great Exhibition of 1851, for instance, might qualify.

The distinction between "Surreal" and "Other" is that "Surreal" works contain at least some semblance of narrative, whereas "Other" is intended for works which "abuse" the format to present some entirely different sort of game - Tetris, say, or Minesweeper.

§25.4 The Library Card

Bibliographic data is useful for two reasons. Firstly, it enables the equivalent of a title page to be printed - traditionally called the "banner" - at the start of play; secondly, Inform uses it to generate the equivalent of a library card for the work, which can be used by other programs to help organise, sort and classify interactive fiction. If the card is given to any other program on any other machine (or an Internet-based archive) then, in principle, that system can know about our work of fiction without a human librarian having to get hold of a copy, play it and laboriously copy out the details.

The "library card" is not of course a physical card, but a small "metadata" file which could potentially be transmitted quickly across the Internet. It contains no personal data other than what you choose to put on it, using the sentences documented in this chapter: it does not, for instance, identify your computer or IP address. In any case Inform does not send it anywhere, but merely keeps an up-to-date copy within the project, and includes it when making a release copy of the work. You can always see (a representation of) the

current library card for a project in the Contents index.

Authors are asked to play fair, in return, by writing sensible and useful bibliographic information for any work which is likely to circulate to other people; by being honest (writing under a pseudonym is fine, but not impersonating other people); and by conforming to standard practice.

The Settings panel of each project contains a tick-box called "Bind up into a Blorb archive on release", and by default this is ticked. "Blorb" is a nonsense word from a popular early 1980s work of IF called "Enchanter", where it was the name of a magic spell whose purpose was to "safely protect a small object as though in a strong box". In the late 1990s, the name was borrowed for a standard format for what might be called the wrapping and packaging of IF. A typical Blorb archive produced by Inform contains the "story file" - the actual program for the story - together with its library card and cover art.

Modern IF interpreters such as Zoom for Mac OS X and Unix, and Windows Frotz, can play blorb archives directly, and the authors of Inform hope to make this the normal practice in future. Still, some interpreters cannot read blorbs directly and have to be given the actual story file: so by unchecking the above tick-box, we can insist that Inform creates only that. The disadvantage with this, of course, is that the library card (with all its bibliographic data) and any cover art is lost in the process.

§25.5 The Treaty of Babel and the IFID

During March and April 2006, an agreement was reached between the IF archive and most of the different systems for creating IF - of which Inform is only one - called the Treaty of Babel. While these different systems create computer programs which are quite different internally, the Treaty provides for works of IF to come with bibliographic data which identifies them in a standard way.

Inform is fully compliant with the Treaty. In particular, each new project created by Inform is allocated a unique identification number called its IFID. The IFID is the equivalent for IF of the ISBN of a printed book. Inform copies it onto the "library card" for the benefit of Internet-based libraries which may eventually accession the work. Of course many projects start but never see the light of day, so many possible IFIDs are "wasted": but that hardly matters, as there are plenty more numbers in the world.

The important thing is that

The IFID number must be unique to this one work out of all the IF ever created

Inform will make sure this is true, **unless** we do something to break this ourselves. For

instance, if we take an existing project, copy it as a file, then work divergently on the original and on the copy so that they become two radically different works, they will still each have the same ID. This is a bad thing: if we want to duplicate a project but then turn it into something new, the best way to do that is to create a new project, and to copy and paste the source from the old to the new.

§25.6 The Release button and the Materials folder

Inform's Release button does two things: it makes a stand-alone, public version of the current project - a "story file" - and it gathers up, or creates, whatever material we want to go with it.

The release version of the project can be played by anyone with an "interpreter" - they do not need the Inform application installed on their computers, and they will not be able to see the source text. Released versions differ slightly from the versions playable in the Story panel of Inform, because debugging commands such as ACTIONS are not included with them. (As we've seen, also excluded is any material in the source text under a heading including the words "not for release".) In some cases, if we release along with an interpreter, we can even make the project playable from a web browser, so that the player doesn't need to install any software at all, not even a browser plugin.

The Release button also creates a ".materials" folder for the project, if one doesn't exist already. (On some platforms, the Inform user interface creates it automatically alongside the project.) Inform adopts the following convention:

> The files associated with the project "Whatever.inform" should all be kept in a subfolder called "Whatever.materials" in the same folder that contains the project.

For example, if we have a project filenamed Magician.inform which lives in a folder called "Works in Progress", then files might be arranged like so:

```
Works in Progress
    Magician.inform
    Magician.materials
        Collegio.pdf
        Mating Wyverns.mp3
```

Of course "Magician" might not actually be the title of the project - it might be an abbreviation, or a working title. The name of the .materials folder has to match the name of the .inform file, not the title.

Several advanced features of Inform make use of the materials folder, and the "Release"

button is one of them. It creates a further subfolder called "Release" within the materials folder. This is where it will always place the story file it creates, together with anything released "along with" the story - Inform will not need to put up a dialogue box asking us where to save the story file, because there is already a natural place. For instance, after a successful click on Release, we might then see:

```
Works in Progress
    Magician.inform
    Magician.materials
        Collegio.pdf
        Mating Wyverns.mp3
        Release
            Collegio.pdf
            Magician.zblorb
            Mating Wyverns.mp3
```

where "Magician.zblorb" is the actual story file produced by Inform. Note that Inform has made copies of the files to be released with it - the idea is that the Release subfolder contains only what Inform makes, and everything in the Release subfolder can be thrown away at any time.

This is especially useful if we're releasing along with a website (see below), as then the Release subfolder will be exactly what needs to be uploaded to a server to be shown to the world. Equally, the Release subfolder is what can be zipped up and uploaded to archives or (if small enough) emailed out.

§25.7 The Joy of Feelies

"Feelie" is a slang word, again going back to the early days of IF, for something tactile included with commercially sold copies of IF games. For instance, Infocom's "Wishbringer" was not just a diskette in a pretty box: the box also contained a map, a letter, an envelope, a magic stone (well, a stone) and a booklet. Most of this was purely for fun, and to flesh out background to the story, but there would usually be clues sneaked into the text or artwork as well.

Today's IF is usually not supplied in physical packaging, and not accompanied by physical objects. But authors do sometimes want to include extraneous matter, whether it's a simple read-me file of instructions or a multimedia extravaganza. Inform does not provide facilities to make artwork, movies, soundscapes, booklets, etc.: there are plenty of programs out there to do all of that already.

But Inform does help with the collation and packaging-together. For instance, by placing the following sentence in the source text:

```
Release along with a file of "Collegio magazine" called "Collegio.pdf" and a file of "The mating call of the green wyvern" called "Mating Wyverns.mp3".
```

…we tell Inform that we will also be providing two additional files. Note that in each case we supply a brief description and a filename. The filename should always have a standard file extension for a well-known and thoroughly standardised file format - ".pdf" and ".mp3" are pretty safe: so for instance are ".txt", ".png", ".jpg", ".html". The filename should not include punctuation marks other than the full stop dividing name from extension, and should not exceed 30 characters in length.

It is also possible to supply a feelie which is not a single file, but is a mini-website: that is, a collection of interlinked HTML (and perhaps other) files. The convention here would be:

```
Release along with a file of "Baltrazar's Guide to Magic" called "Guide".
```

The absence of a file extension on the filename "Guide" tells Inform that the feelie in question is a mini-website: it is expected to sit inside a folder called "Guide", with its home page being "Guide/index.html". However, a mini-website like this must be created by hand: Inform does not copy it into place, it only creates links to the place where it ought to be put.

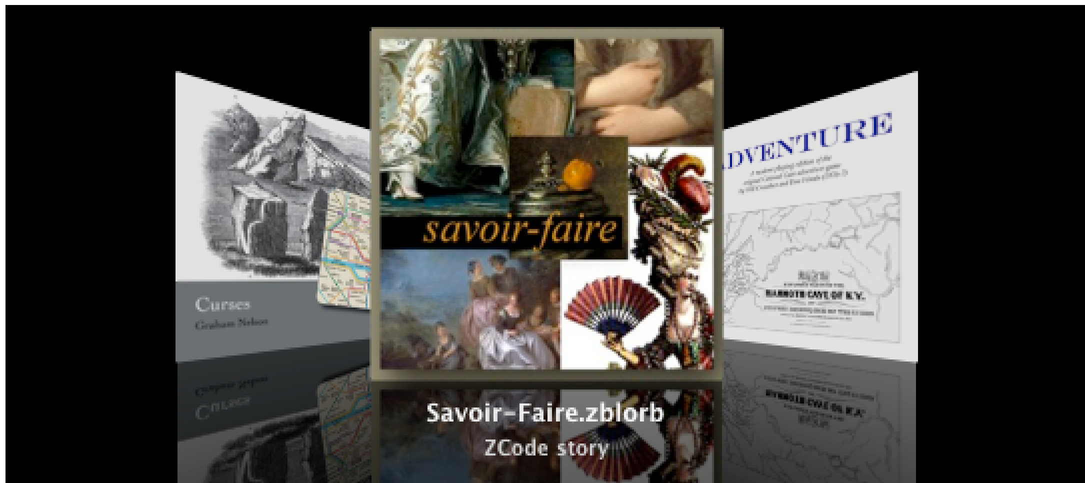
We have seen that Inform takes the story file, which is analogous to the pages of a book, and places it into a Blorb archive, analogous to the binding. These new additional files are not placed in the Blorb, because that would make the Blorb archive rather large (and would hide them from the player, which defeats the purpose). But references to them do appear in the Blorb, so that any interpreter playing the Blorb would be able to tell that there are supposed to be additional files available. Similarly, references are entered onto the library card.

§25.8 Cover art

Accompanying files are not the only things which can be included in a "release along with" sentence: for instance, we could

```
Release along with cover art ("A stone gargoyle"), a file of "Collegio magazine" called "Collegio.pdf" and a file of "The mating call of the green wyvern" called "Mating Wyverns.mp3".
```

Cover art can not only be used to advertise a work of IF, it is also displayed to players by certain interpreters, such as Zoom or Spatterlight for OS X, or Windows Frotz for Windows. It is also used on the IFDB (ifdb.org), and by browsing applications. If Zoom is installed, then on Mac OS X Leopard, the Finder shows cover art directly:



Cover art for a work should be prepared in either JPEG (".jpg") or PNG (".png") format, and we recommend that it should be square, like a music album cover. Programs which notice the cover art for a work of IF are likely to scale this up or down as convenient for their own display purposes, but it would be helpful to provide the original art at 960 by 960 resolution. The cover art must not be smaller than 120 pixels in either dimension.

To provide cover art, we should create an image file called Cover.jpg, or else Cover.png, and place it in the project's .materials folder. For instance, we might have:

```
Works in Progress
    Magician.inform
    Magician.materials
        Collegio.pdf
        Cover.jpg
        Mating Wyverns.mp3
```

(supposing that, as in the previous examples, "Collegio.pdf" and "Mating Wyverns.mp3" are the filenames of two feelies that accompany the release).

The text in brackets after the release instruction…

```
Release along with cover art ("A cathedral at sunset.").
```

…is provided for the benefit of blind or partially sighted users, and should be brief.

§25.9 An introductory booklet and postcard

When IF is aimed particularly at people who have never played IF before, there are certain conventions which it's a good idea to explain, or players will simply not know what to do. It can become a chore writing a clear set of instructions, and then there is the further nuisance of explaining about the need for an interpreter program to play the IF story file.

To alleviate this, Inform can "Release along with an introductory booklet", as for instance in this example:

```
Release along with cover art, the introductory booklet, a file of "Collegio magazine" called "Collegio.pdf" and a file of "The mating call of the green wyvern" called "Mating Wyverns.mp3".
```

The introductory booklet is a standard 8-page PDF file, written and designed by Emily Short, which contains all the basic information needed for a player to get started. It has been written to be as general-purpose as possible, in the hope of being useful for a range of widely different works of IF. There will certainly be works to which it would not be an appropriate supplement, and some authors will certainly prefer to write their own notes for players, but of course it is not compulsory. By making it available as a convenience, the authors of Inform do not intend to say that these are the "official" instructions or that others are not. It is simply intended as a time-saver.

As an alternative, or a supplement, it's also possible to:

```
Release along with an introductory postcard.
```

which supplies a standard postcard about IF (everything new players need to know, at one glance) written and designed by Andrew Plotkin and Lea Albaugh.

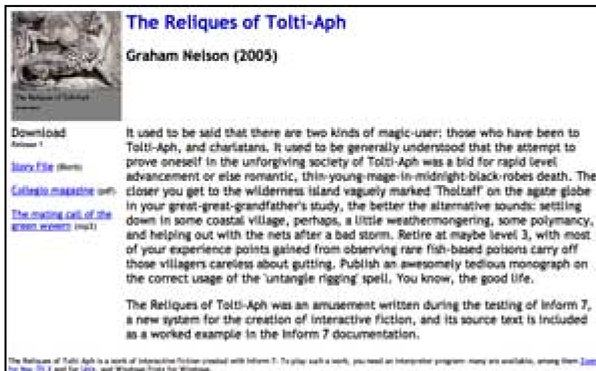
§25.10 A website

Much of the published IF of the last twenty years came with a brief text file describing what it was - a release note. Today it makes more sense to write this as a small web page, which can either be placed online, or simply distributed as part of the release.

Inform is able to manufacture such a website automatically. We request this by writing, for instance,

```
Release along with cover art, a website, a file of "Collegio magazine" called "Collegio.pdf" and a file of "The mating call of the green wyvern" called "Mating Wyverns.mp3".
```

where the list of ingredients now includes "a website". In fact, Inform makes only a single web page, called "index.html", which it places in the materials folder (as set up in the previous section): this then contains suitable links to all the other material, such as the cover art images, if they are also provided. For instance:



After a successful release now, then, we should see:

```
Works in Progress
    Magician.inform
    Magician.materials
        Collegio.pdf
        Cover.jpg
        Mating Wyverns.mp3
        Release
            Collegio.pdf
            Cover.jpg
            index.html
            Magician.zblorb
            Mating Wyverns.mp3
            Small Cover.jpg
```

("Release/Small Cover.jpg" is a form of the cover image intended for display at a smaller size. In earlier versions of Inform, the author had to provide this: there is now no need.)

§25.11 A playable web page

Modern web browsers are now so powerful as computing environments that they almost amount to general-purpose computers in their own right. The websites made in the previous section were passive, and simply displayed information about a story file. But it's also possible to make a more active page - one which can play the story file, right inside the browser, for anybody who visits.

To make such a page, we must:

```
Release along with an interpreter.
```

This automatically releases along with a website as well, since we need the website in order to house the new page, which will be called "play.html". This page will be bundled up with a customised copy of a story file interpreter coded in Javascript - in effect, a

program for a web browser to follow - and a suitably encoded version of the story file. The practical effect should be that anyone visiting the page with any modern browser can just play.

Inform ships with the "Parchment" and "Quixe" interpreters built in. By default Inform will use Parchment if the format (on the project's Settings panel) is set to Z-code, and Quixe if the format is Glulx. In fact, though, Parchment works with either format, and some users prefer using it. If we want to have Parchment even for a Glulx project, we can write:

```
Release along with the "Parchment" interpreter.
```

…and that's just what will happen. In fact, Inform also supports the use of any other interpreter the author wants to try. If we have access to an exotic Javascript-based interpreter called, let's say, "Urbzig", then we can install it by putting it into the "Templates" subfolder of the ".materials" folder for the project:

```
Release along with the "Urbzig" interpreter.
```

§25.12 Using Inform with Vorple

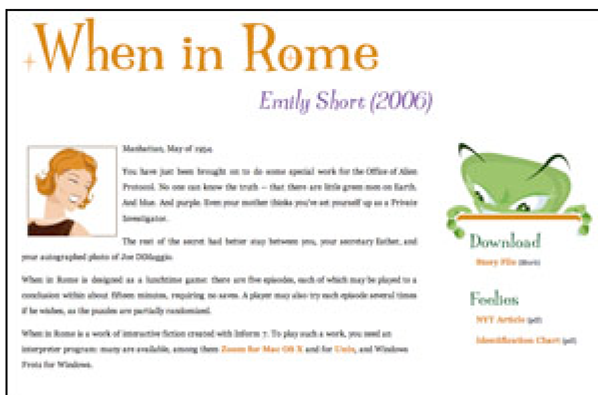
"Vorple" is an innovative system by Juhana Leinonen for allowing web-based Inform stories to make use of web controls and other gadgets. Using Vorple, a story can in principle have an entirely different user interface, and can make much better use of CSS styling, interface to Javascript libraries, and so on.

Vorple has seen rapid development. In its early days it was included as part of the Inform app, but it has now evolved into a dynamic project which is better served by its own website than from here:

```
vorple-if.com
```

§25.13 Website templates

Web pages are very idiosyncratic things and Inform will almost certainly not produce exactly what we want. What it actually does is to take an existing "template" web page, and paste in the relevant information to make the final product. So by starting with a different template, we can end up with an entirely different-looking web page: like this one, for instance -



The template ordinarily used by Inform is called "Standard" and comes built in. (A second built-in template, "Classic", imitates the look used in 2005-08. The word "classic" here is to be understood in the sense of Classic Mac OS, the classic Doctor Who adventure "Time and the Rani", classic Mayan civilisation, and so forth - really pretty awful.)

Any other templates we must make ourselves, giving each one a different name, by convention a single word. In this section, we'll make a new one called "Platinum".

Suppose we write:

```
Release along with cover art, a "Platinum" website, a file of "Collegio magazine" called "Collegio.pdf" and a file of "The mating call of the green wyvern" called "Mating Wyverns.mp3".
```

This is identical to the previous version except for the "Platinum": note the quotation marks. When it needs to find a template, Inform searches the following places in sequence:

```
(a) the "Templates" subfolder of the project's own .materials folder, if this subfolder should exist;
(b) the "Templates" folder in the user's own library - on Mac OS X, this is:
    ~/Library/Inform/Templates
or on Windows:
    My Documents\Inform\Templates
or on Linux:
    /Inform/Templates
(c) the built-in stock of templates, currently only "Standard" and "Classic".
```

What Inform looks for is a folder name matching that of the template - so in our case we need to provide a folder called "Platinum", and put it in either location (a) or (b).

The template folder is expected to contain some combination of the following files:

```
Platinum
    index.html
    source.html
    style.css
    (extras).txt
```

There are two HTML pages here, one for the main front page, the other for pages of displayed source text (if we release along with the source text - see later in the chapter). The CSS file defines styles of text - sizes, fonts, colours, and so on - and positions material on the page. The "(extras).txt" - which is optional, of course - allows additional HTML pages, images, movies and so on to be added.

If any of these is missing, Inform uses the one in "Standard" instead. In practice, this means the easiest way to create a new template is to supply just a new CSS file, which can change the colour, font, type size, and position of more or less everything in the site:

```
Platinum
    style.css
```

We probably want to start from the "Standard" version of "style.css" and edit in a few changes; the easiest way to get a clean copy of "Standard"'s CSS file to work on is to release the project with a "Standard" template, which causes this default "style.css" to appear in the "Release" subfolder of the project's .materials folder. (But it's wise to move the file out of "Release" before starting to edit it - files in "Release" are overwritten by Inform whenever a release is made.)

This is not the place to describe how CSS works. CSS is a more or less universal format today for describing how web pages should look - their style rather than their content. A dazzling variety of possibilities can be seen at the excellent:

```
csszengarden.com
```

but of course there are many, many other textbooks and websites which describe CSS.

§25.14 Advanced website templates

The following describes how Inform uses the extras file and the two HTML pages in a template, and will only be needed if a new template has to make changes so radical that altering the CSS alone won't be enough.

The optional "(extras).txt" file - note brackets - is a text file which contains a list of named extras to throw in. For instance:

```
easter.html
egg.png
```

These named files need to be present in the template folder. Files with the extension ".html" go through the placeholder expansion process just like the index and source pages; all other files are copied verbatim.

HTML templates like "index.html" and "source.html" are fully valid HTML pages in their own right, though they have placeholder text where Inform will substitute the project's bibliographic data (see below). The "<head>" element should include a reference to "style.css", which of course will mean the CSS file given in the template (or the one from "Standard" if no CSS file is given) - for instance,

```
<link rel="stylesheet" href="style.css" type="text/css" media="all" />
```

When it turns the template into the final web page, what Inform does is to replace certain capitalised words in square brackets with the appropriate text:

```
[TITLE] becomes the story title
[AUTHOR] becomes the author's name
[YEAR] becomes the story creation year
[BLURB] becomes the story description
[RELEASE] becomes the release number
[COVER] becomes an image of the cover art (the small 120x120 cover image)
[DOWNLOAD] becomes the download link
[AUXILIARY] becomes the list of feelie-like files, if any
[IFID] becomes the IFID
[STORYFILE] becomes the "leafname" of the story file, e.g., "Bronze.gblorb"
[TEMPLATE] becomes the name of the template used to make the page
[SMALLCOVER] becomes the filename of the cover when used at a smaller size
[BIGCOVER] becomes the filename of the cover when used at full size
[TIMESTAMP] and [DATESTAMP] become the time and date of releasing
```

Everything else is left alone. In source pages, five further placeholders are available:

```
[SOURCE] becomes the portion of the source text on this page
[SOURCELINKS] becomes the navigational links
[SOURCENOTES] becomes the footnote matter at the bottom of the source
[PAGENUMBER] and [PAGEEXTENT] are such that the text "page [PAGENUMBER] of
[PAGEEXTENT]" produces, e.g., "page 2 of 7"
```

Both [SOURCE] and [SOURCENOTES] must exist on the page, and [SOURCENOTES] must

appear after [SOURCE] does in the file. (Of course the CSS in "style.css" might move the copy around on screen, but that's another matter.)

§25.15 Republishing existing works of IF

Some long-time users of Inform will have projects which were originally made using the very different Inform 6 language. Story files produced with Inform 6 do not have any of the extra touches in this chapter: in particular, they have no cover art and no bibliographic data, which makes them rather plain and anonymous to newer Treaty of Babel-equipped programs like Zoom, Spatterlight or Windows Frotz.

To help with this, today's Inform can republish an Inform 6 project by combining an Inform 7 source text which contains only release instructions and bibliographic data with an already-compiled Inform 6 story file. We do this by writing a short source text which contains:

```
Release along with an existing story file.
```

We then place the story file in the ".materials" folder. By default this will be called "Story.z8", but we can alternatively name it:

```
Release along with an existing story file called "Zork1_sg.z5".
```

The Settings panel must be switched to the Z-machine for this to work, since only Z-machine story files are supported this way, not Glulx. And we can now use the Release button to obtain the goods.

An existing story file can take advantage of all of the extra features - cover art, titling, website, feelies and so forth - earlier in this chapter, but not those - walkthrough, source text, map - which are still to come.

The following is a typical example of a source text used solely to bind up an old Inform 6-compiled story file:

```
"Curses" by Graham Nelson

The story genre is "Fantasy".

The story headline is "An Interactive Diversion".

The story creation year is 1993.

The release number is 16.

The story description is "It's become a matter of pride now not to give up. That tourist map of Paris must be up here somewhere in all this clutter, even if it has been five years since your last trip. And it's your own fault. It looks as if your great-grandfather was the last person to tidy up these lofts..."

Release along with cover art and an existing story file.
```

§25.16 Walkthrough solutions

Since the earliest days of IF, players have distributed solutions to well-known stories, to help out other players at their wits' ends. The commonest format for these is a list of commands to type, sometimes with notes in the margin, and such a solution is called a "walkthrough", since it walks a player through the story.

Few authors publish solutions of their own works, but many supply their testers with solutions, especially towards the end of testing, or submit a solution as part of a competition entry. To help with this, Inform can generate such a walkthrough solution automatically:

```
Release along with a solution.
```

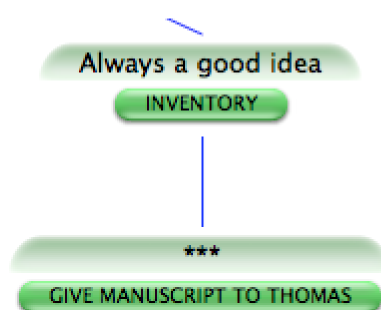
Inform will then place a file called "solution.txt" inside the "Release" folder. The solution might look like so (although probably much longer):

```
Solution to "Memoirs of India" by Graham Nelson

Choice:
INVENTORY -> go to branch (1)
EAST -> go to branch (2)

Branch (1)
DROP MANUSCRIPT
SOUTH

Branch (2)
INVENTORY ... Always a good idea
GIVE MANUSCRIPT TO THOMAS
```

Inform does not, of course, know how to solve IF all by itself, but derives the solution

from the project's Skein. Since the Skein will have been used in testing the story, it will very likely contain a perfect solution - or several different ones, taking the story to a variety of possible endings. In the example above, there are two possible winning lines, which diverge right from the first move. (There can be further divergences: for instance, if branch (2) splits, it will split into branches called (2.1), (2.2), (2.3) and so on.)

But the Skein will also contain plenty of unwanted diversions, so Inform does not rewrite the entire Skein as a solution. Instead, it looks for knots in the Skein which have been annotated. Any knot whose annotation begins "***" (three asterisks) is considered to be a final, winning move. (It is probably a good idea to lock such a knot once it has been annotated thus, too.) We can mark any number of knots "***" since, after all, we can declare any number of lines of play as possible solutions. Inform then constructs the solution out of all lines of play in the Skein which lead to "***" endings, and ignores other threads.



Annotations other than "***" in the Skein are turned automatically into comments in the solution text. For instance, the knot for the INVENTORY command in the second branch above was annotated "Always a good idea", and this was transcribed into the solution. (If an ending knot is annotated with, say, "*** Happy ending!" then the "***" marks it as an ending, and "Happy ending!" is added as an annotation to that ending.)

By default, the solution text is not linked from our webpage, on the assumption that we may want to generate a walkthrough but not immediately advertise it to players. If we wish to change this, we may write instead

```
Release along with a public solution.
```

The terms public and private may also be applied to other elements we are having Inform generate to include on our webpage: see also the notes on private source text, below.

§25.17 Releasing the source text

Most authors will not want to publish the source text alongside the work itself, because this gives away all of its secrets. Inform provides the option mainly for the sake of the

examples published on its own website, where making the source available is the whole point. But anyone is welcome to use the option, of course:

```
Release along with the source text.
```

If Inform is not also generating a website, this produces a plain text file called "source.txt" in the "Release" folder, and there is nothing more to be said.

However, if a website is also being released, the source is also converted to a suite of web pages which are linked to and from the home page. (Each heading with substantive content is placed on its own web page, with the opening page containing a contents list.)

Comments in the source are rendered in grey. As a special feature, any comment which begins with an asterisk is considered a footnote and is printed below the source text, with a link. Thus comments thus:

```
Hercules is a demigod.[* We're using Greek spellings so he ought to be Heracles, but players are so much more familiar with Hercules.]
```

will be printed more like so:

```
Hercules is a demigod.[1]
...
Note
[1]. We're using Greek spellings so he ought to be Heracles, but players are so much more familiar with Hercules.
```

Footnotes are automatically numbered from 1 on each source page.

By default, the source text is linked from our generated webpage, if we are releasing with a webpage. If we wish to change this, we may write instead

```
Release along with the private source text.
```

This will create a text file containing the source for our story, and place this file in our release folder, but not create a link so that the player can find it.

Finally, we can:

```
Release along with the library card.
```

which releases a stand-alone XML file in 'iFiction' format for the bibliographic data on the

story file; this is the same data embedded in the blorb file itself, but having an external copy makes it easier to see what Inform has done, and some external programs can read iFiction data like this.

§25.18 Improving the index map

As we have seen, "Release along with…" allows us to package up a work of IF with all manner of extra materials. But what are these to be? One popular option is to produce a map - sometimes partial, sometimes obfuscated - and supply that with the story: besides, there are some IF competitions where the rules require that the referee is supplied with a map even if the players are not, and failing that, it is sometimes nice to be able to print out a map of a work in progress.

The World map in the Index tab is heavily stylised and cartoonish, intended to be clicked on or moused over, and viewed in a browser: although it is, in fact, possible to print it, the results are not very good. Fortunately, the same underlying map mechanism can be used to output something more useful and very much more customisable, as we shall see.

The map-maker is one of the most complex parts of Inform, even though it actually contributes nothing to the final story file: the problem of how to draw up a "correct" map from the source text is by no means easy to solve. Inform tries, but it often gets things wrong. Its general practice is to place rooms on a square grid (actually a cubic lattice, as it works in three dimensions), but not all conceptual maps fit well onto this, and Inform often annoyingly puts a particular room in the "wrong" place. For instance, suppose Inform puts "Didcot" east of "Abingdon" and this makes the geometry look different to what we had in mind. We can correct with:

```
Index map with Didcot mapped southeast of Abingdon.
```

Note that this says nothing about exits from any room to any other room, and changes the final work of IF not at all: it simply helps Inform to draw the map index. (Instructions like this one are treated as being almost certainly true, but Inform does not quite always obey: it will never allow two rooms to be superimposed at the same grid position, no matter what we have asked in "Index map with…" instructions.) The same trick is useful if we have a situation like so:

```
Inside of Sweeping Sands is Beach Hut Interior.
```

"Beach Hut Interior" is a single room which does not connect to the rest of the map by any of the ten spatial directions, so Inform does not place it on the main map but instead moves it off out of the way in a map of its own. Given that it's just a single room, however,

we might prefer to put into a convenient otherwise empty grid position like so:

```
Index map with Beach Hut Interior mapped west of Sweeping Sands.
```

Finally, note that this trick also ensures that the two locations are mapped on the same level vertically, and can be useful in cases where room A is both north of and above room B: Inform will want A to be higher up than B, but we can insist otherwise.

§25.19 Producing an EPS format map

The "Index map with…" instruction is a much more varied thing than hinted at in the previous section, and its general form is

```
Index map with [instruction] and [instruction] and ... and [instruction].
```

where the instructions can be of four different forms, as follows:

```
[room A] mapped [direction] of [room B]
EPS file
rubric [text] ... and some optional details ...
[setting] of [whatever] set to [value]
```

We have already seen the first of these instructions. The second is short and has a fixed wording:

```
EPS file
```

so can be invoked by typing "Index map with EPS file.", for instance. EPS stands for Encapsulated PostScript, which is a standard file format for line art. EPS files can be edited with sophisticated graphics programs such as Adobe Illustrator, and can be used as illustrations in many word-processors and page layout programs. They can also be converted to PDF by Mac OS X Preview, or used in Linux or Windows with the open-source Evince viewer. We need a line-art format because the map produced will never be exactly what we want: we are probably going to end up hacking it to change the fonts, add some drawings, tidy up the spacing and so on. A really large map will end up using quite a large "canvas", in EPS terms; it may be necessary to shrink it down in order to get it onto an A4 page, or to adjust whatever editing software is used to "custom paper size".

When the map-maker has been given the "EPS file" instruction, it writes an attempt to draw the current project's map in EPS format as a file in the project's ".materials" folder, with the filename "Inform Map.eps".

Note that Inform will over-write any existing file of this name: but that is intentional, because one usually ends up tweaking and rebuilding the project over and over to get the map just so, and it would be tiresome for Inform to produce endless copies "Inform Map 19.eps", etc.

(The reason the EPS file is not placed in the Release subfolder is that it is not going to releasable to the public as it stands: for one thing it will be too raw, and for another, EPS is not a format everyone can read. It is provided as raw materials.)

§25.20 Settings in the map-maker

The map-maker has altogether 35 named settings, and tweaking these can affect the result in ways which vary from the subtle to the grotesque. An important point is that the map-maker deals separately with the three levels in its working: the big picture of the whole map; each of the vertical slices which contain sub-maps; and finally all of the individual rooms. For instance, we might have 67 rooms, arranged on 3 vertical levels, all shown on one big map: Inform will try to show these stacked above each other, with the highest level at the top of the map, then the middle level, then the bottom level.

Moreover, not only does the whole map have its 35 settings, but each level has its own independent collection of those 35 settings, and so does each individual room. So the actual number of variables in our example is 1+3+67 = 71 times 35, which is a lot. The convention is that setting the value of S (some setting, let's say) for something affects not only that thing, but also everything inside it, unless they have their own individual settings for S.

For example: one of the settings is called "room-size", and is the size of the little square boxes representing a room, measured in points. (One point is 1/72 of an inch, so 72 points equals 1 inch: it's a traditional printer's measure.) Suppose we write:

```
Index map with room-size set to 36
    and room-size of level 2 set to 28
    and room-size of the Hall of Kings set to 52.
```

The first instruction sets the value of "room-size" for the whole map (note the lack of an "of…"); the second for level 2 of the map, and the last for a single room only. The result is that the Hall of Kings is drawn as 52x52 point box, all rooms on level 2 are 28x28 (except the Hall of Kings, if it's on level 2), and all others are 36x36, half an inch square.

The setting instruction also allows three other useful forms. A setting "of the first room" applies to the room in which the story begins: we might for instance write

```
Index map with room-outline-thickness of the first room set to 2.
```

which gives this special room a bolder edge to it, since the default value is 1.

We can also apply settings not just to single rooms but to all rooms of a given kind:

```
A rivery room is a kind of room. Index map with room-colour of rivery rooms set to "Navy" and room-name-colour of rivery rooms set to "White".
```

Lastly, we can apply settings to all rooms in a given region:

```
Northern Oxfordshire is a region. Hampton Poyle and Steeple Barton are in Northern Oxfordshire. Index map with room-name-font of Northern Oxfordshire set to "Helvetica-Oblique".
```

(Note that rooms and regions don't have their own individual sets of the 35 settings: what happens is just that instructions like the last one change more than one room at once.)

§25.21 Table of map-maker settings

Note that all map-maker settings have single word names, though many are hyphenated, and that "colour" is always given the English and Canadian spelling, not the American form "color".

font	font (named in double-quotes)
minimum-map-width	integer (measured in points: 72 = 1 inch)
title	text (in double-quotes)
title-size	integer (measured in points)
title-font	font (named in double-quotes)
title-colour	colour (named in double-quotes)
map-outline	on/off
border-size	integer (measured in points)
vertical-spacing	integer (measured in points)
monochrome	on/off
annotation-size	integer (measured in points)
annotation-length	integer (length to abbreviate down to)
annotation-font	font (named in double-quotes)
subtitle	text (in double-quotes)
subtitle-size	integer (measured in points)
subtitle-font	font (named in double-quotes)
subtitle-colour	colour (named in double-quotes)
grid-size	integer (measured in points)
route-stiffness	integer (Bezier spline curve scale factor)
route-thickness	integer (measured in points)
route-colour	colour (named in double-quotes)
room-offset	offset (in percentages of grid-size)
room-size	integer (measured in points)
room-colour	colour (named in double-quotes)
room-name	text (in double-quotes)
room-name-size	integer (measured in points)
room-name-font	font (named in double-quotes)
room-name-colour	colour (named in double-quotes)
room-name-length	integer (length to abbreviate down to)
room-name-offset	offset (in percentages of grid-size)
room-outline	on/off
room-outline-colour	colour (named in double-quotes)

room-outline-thickness	integer (measured in points)
room-shape	shape (named in double-quotes)

§25.22 Kinds of value accepted by the map-maker

Integer values are typed in the usual way: 3, -72, etc.

Text is in double-quotes: "Map of Lower Delta", etc.

Font names are in double-quotes: "Helvetica", etc. Note that Inform makes no effort to look for such fonts: if we give the name of a font we haven't got, the result will probably be that the map's EPS file will be displayed in various applications with Courier (which looks like bad typewriting) substituted. All fonts are by default equal to the global "font" setting (by default equal to "Helvetica"), so changing "font" for the whole map affects everything not explicitly specified as having a different font.

Shape names are in double-quotes with lower case. At present, the only legal shapes are "circle", "square" and "rectangle".

On/off values are written just thus: on, off. No quotation marks.

Offset values are actually pairs, and are written as two numbers (possibly negative numbers) joined by an ampersand, as in the example: "Index map with room-offset of Botley set to 10&-30." Note lack of spaces around the ampersand. This means that Botley's room is displaced from its correct grid position on the EPS map by 10% of the grid size eastwards, and 30% southwards. (The grid size is the distance between one grid position and the next: displacing Botley by -200&0 would move it two whole grid positions westwards.)

The **route-stiffness** setting is used when drawing routes between two rooms. These are drawn as Bezier curves, a standard way to make a smooth curve not only travel from A to B but also from pointing in a given direction at A to ending up pointing in a given direction at B. Thus a Bezier curve may turn a route round so that it leaves A pointing west, but curves around to enter B from the south. (Most routes involve leaving in one direction and arriving in the opposite direction, of course, and in those cases a Bezier curve is just a straight line.)

The stiffness factor for a given room measures how much the curves are allowed to warp around in order to force them to arrive at that room from exactly the right compass bearing. The default is 100. Raising to, say, 250 can force curved paths into freakish zig-zags: whereas lowering to 1, the minimum, may make the route arrive at completely the wrong bearing. (Formally speaking: at each end of the route, a "control point" for the

Bezier curve is made by taking the centre point of the room, then adding the relevant compass bearing's vector, scaled up by the route-stiffness as a percentage of the grid size.)

Colour values are named and in double-quotes. These names are the same as those for the traditional set of web-page-safe colour chips, as follows:

```
"Alice Blue"
"Antique White"
"Aqua"
"Aquamarine"
"Azure"
"Beige"
"Bisque"
"Black"
"Blanched Almond"
"Blue"
"Blue Violet"
"Brown"
"Burly Wood"
"Cadet Blue"
"Chartreuse"
"Chocolate"
"Coral"
"Cornflower Blue"
"Cornsilk"
"Crimson"
"Cyan"
"Dark Blue"
"Dark Cyan"
"Dark Golden Rod"
"Dark Gray"
"Dark Green"
"Dark Khaki"
"Dark Magenta"
"Dark Olive Green"
"Dark Orange"
"Dark Orchid"
"Dark Red"
"Dark Salmon"
"Dark Sea Green"
"Dark Slate Blue"
"Dark Slate Gray"
"Dark Turquoise"
"Dark Violet"
"Deep Pink"
"Deep Sky Blue"
"Dim Gray"
"Dodger Blue"
"Feldspar"
"Fire Brick"
"Floral White"
```

```
"Forest Green"
"Fuchsia"
"Gainsboro"
"Ghost White"
"Gold"
"Golden Rod"
"Gray"
"Green"
"Green Yellow"
"Honey Dew"
"Hot Pink"
"Indian Red"
"Indigo"
"Ivory"
"Khaki"
"Lavender"
"Lavender Blush"
"Lawn Green"
"Lemon Chiffon"
"Light Blue"
"Light Coral"
"Light Cyan"
"Light Golden Rod Yellow"
"Light Grey"
"Light Green"
"Light Pink"
"Light Salmon"
"Light Sea Green"
"Light Sky Blue"
"Light Slate Blue"
"Light Slate Gray"
"Light Steel Blue"
"Light Yellow"
"Lime"
"Lime Green"
"Linen"
"Magenta"
"Maroon"
"Medium Aquamarine"
"Medium Blue"
"Medium Orchid"
"Medium Purple"
"Medium Sea Green"
"Medium Slate Blue"
"Medium Spring Green"
```

```
"Medium Turquoise"
"Medium Violet Red"
"Midnight Blue"
"Mint Cream"
"Misty Rose"
"Moccasin"
"Navajo White"
"Navy"
"Old Lace"
"Olive"
"Olive Drab"
"Orange"
"Orange Red"
"Orchid"
"Pale Golden Rod"
"Pale Green"
"Pale Turquoise"
"Pale Violet Red"
"Papaya Whip"
"Peach Puff"
"Peru"
"Pink"
"Plum"
"Powder Blue"
"Purple"
"Red"
"Rosy Brown"
"Royal Blue"
"Saddle Brown"
"Salmon"
"Sandy Brown"
"Sea Green"
"Sea Shell"
"Sienna"
"Silver"
"Sky Blue"
"Slate Blue"
"Slate Gray"
"Snow"
"Spring Green"
"Steel Blue"
"Tan"
"Teal"
"Thistle"
"Tomato"
```

```
"Turquoise"
"Violet"
"Violet Red"
"Wheat"
"White"
"White Smoke"
"Yellow"
"Yellow Green"
```

§25.23 Titling and abbreviation

The main title of the map is the value of "title" for the whole map, so for instance we might write:

```
Index map with title set to "Oxford and its Environs".
```

The subtitle settings apply to the subtitles used for each of the levels, so for instance

```
Index map with subtitle of level -1 set to "Tunnels and Sewers".
```

Names of individual rooms can be controlled with:

```
Index map with name of Radcliffe Camera set to "Library".
```

(By default, the name of a room is its name in the main IF project, of course.) The smallest writing on the map is normally that used to label unorthodox or unclear exits (in particular, those going from one layer to another): this is what the "annotation" size, font and colour are used for.

For most ways to set up the map, it's a practical necessity to abbreviate names of rooms, or they will spill out all over each other. Inform does this using the "room-name-length" setting. (The "annotation-name-length" is analogous.) For instance, if this setting is 5, then Inform will reduce the text of a name to at most 5 characters. It does this by successively throwing out spaces, lower case vowels, then other lower case letters, punctuation marks and finally upper case letters, always starting at the back of the name and working inwards: the process stops as soon as the name is short enough. For instance, "Reading" is abbreviated to "Redng", "Shangri-La" to "Shn-La" and "Cloud-Cuckoo-Land" to "C-C-L". The result can be a little comical, but is surprisingly unambiguous in practice. Abbreviation can effectively be abolished by raising the "room-name-length" to 128 (the highest permitted level), and note that the setting can be changed for individual rooms, so it is possible to have some room names abbreviated

and others not, or in different degrees.

Examples

445. **Baedeker**
    Creating a floorplan of the cathedral using the locations from previous examples.
    RB 13.2 Publishing

446. **Port Royal 5**
    Port Royal scenario given instructions for an EPS map.
    RB 13.2 Publishing

447. **Bay Leaves and Honey Wine**
    Creating a map of Greece using the locations from previous examples.
    RB 13.2 Publishing

§25.24 Rubrics

Lastly, we can add our own arbitrary text to the map: perhaps to annotate points, perhaps just to add more heading matter (such as the author's name, or the date). Each individual line added - and only single lines can be added, not typeset paragraphs - is called a "rubric". (There can be up to 100 of these.) We can create a rubric like so:

```
Index map with rubric "Here Be Wyverns" size 16 font "Helvetica-Oblique" colour "Thistle" at 150&0 from Cloud-Cuckoo-Land.
```

This gives rather more detailed information than is needed: "size 16" could have been omitted, giving us 12-point type by default, and similarly there is no need to specify a font unless it differs from the main "font" setting for the whole map; and the colour will be black if unspecified. The "at" position does need to be given, though. Note that it is relative to a given room on the map, and that the position specified is that of the centre-point of the text. (If we had written just "at 100&100", say, that would specify a position relative to the bottom left hand corner of the map.) So, for instance:

```
Index map with rubric "trapped door" size 8 at -60&-60 from Longwall.
```

would add a little 8-point-type safety tip for naive map-followers.

Inevitably, the settings in the map-maker will fail to get exactly the effect desired (though they will offer an excellent opportunity to waste entire days). But that is the whole point of producing output in EPS format: Inform aims not to produce final print-ready professional art, but to produce the raw material for making that final work of art. And if

all that's required is a sketch-map, then Inform's output should be good enough quickly and without too much fuss.

26. Publishing

§26.1 Finding a readership
§26.2 Editing and Quality Assurance
§26.3 A Page of Its Own
§26.4 The IF Archive
§26.5 IFDB: The Interactive Fiction Database
§26.6 Competitions, Exhibitions, and Jams
§26.7 Meetups and Conferences
§26.8 A short concluding homily

§26.1 Finding a readership

So the new work of IF is written, and tested, and has all its bibliographic data and a fancy cover illustration lined up. What next?

Releasing and gaining attention for independent games - commercial or otherwise - is a big, complex, and constantly changing field, and other online sources will be able to provide more up-to-date information than we can offer here. However, there are some resources, events, and community spaces specifically for authors of interactive fiction and text adventures in particular.

First, though, a word about terminology.

For many years, the phrase "interactive fiction" referred primarily to parser-based games like the ones Inform produces by default. For those games, there has always been an avid hobbyist community, but few sales, and most parser IF writers have not felt that it would be more trouble than it was worth to charge for their games, because the income would be slight relative to the effort of setting up a storefront.

In recent years, other forms of interactive fiction - those that do not rely on typed input from the player - have experienced a commercial revival. There are a number of commercial game studios that write text-rich, choice-driven stories, especially for a mobile market.

Simultaneously, the communities of interactive fiction readers and players have grown and diversified. Once "the IF community" referred to a specific group of people; now, there are many communities of people who play text-based games, in various formats, with various amounts of overlap.

Although it is not a typical tool for choice-based mobile games, Inform has been used to produce commercial works, both parser-based and not. Users are very welcome to sell

works created by Inform with no royalty or requirement for rights clearance. It's also widely used in education, and as a prototyping tool for other kinds of stories, such as interactive narratives that will ultimately take another (not text-based) form.

§26.2 Editing and Quality Assurance

Authors coming from a literary background may think in terms of editing; people coming from software development and the game industry may think about playtesting and quality assurance.

Whatever the background, it's good practice to have your work checked by other people before you release it. Other players can identify issues from typos to missing hints to thematic incongruities.

Play-testers can often be recruited by placing an ad on **intfiction.org**.

§26.3 A Page of Its Own

One option for sharing your work with the world is to set up a web page and a copy of the story file on a private web host. That host should ideally be as stable as possible, so that the URL is likely to remain fixed for what might be a long period. Freeware stories have a long period of viability relative to commercial games, which means that players may still be hearing about and checking out a story years after its initial release. A stable address helps everyone with links, and makes it easier for search engines to direct people.

Of course creating a web page involves a little design work, but tools are widely available which make this quite easy nowadays. And as we've seen, Inform can automatically generate web pages and whole small mini-sites to put all the information about a story file into a tidy format, even including the ability to play online.

A second approach - instead of or alongside giving the game its own website - is to put it on a distribution platform designed for sharing games.

One of the most accessible is itch.io. While it's a lot of work to put a game on a mobile app store or on Steam, setting up a storefront at the itch.io site takes only a few minutes. Doing so enables an author to list a game for download, set a price for their work or just to accept donations of the player's choosing.

A game on itch.io will still need promotion and other attention if the author hopes to make any significant amount of money, but the barriers to listing something for sale are much lower than they once were. And itch.io can be a viable way to share a game that isn't intended to charge money at all.

At the time of this writing, the itch.io platform lists 15,988 games tagged "interactive

fiction."

§26.4 The IF Archive

Games and interactive works in general tend to become obsolete or unplayable fairly quickly. Many games written for iOS in the mid-2010s, for instance, are already impossible to access.

Because of the portable underlying format, however, games written in Inform are unusually stable and maintainable. Inform projects written in the early 90s can still be played - indeed, can be played on platforms that did not exist when the games were written.

If you're interested in the longevity of your project, you may want to submit the final version to the **IF Archive**.

The Archive is a mirrored, stable collection of thousands of interactive fiction games and programming languages, manuals, fanzines, maps, walkthroughs, and other materials. As such, it's likely to stay around even if a personal website goes off-line; it's also the primary resource for people doing scholarship on interactive fiction (and there are a growing number of these).

The Archive is very much a library, for long-term archiving, rather than a book-store. The catalogue is sober and textual, and there are no visual shop-windows, or posters advertising new titles hot off the press. Newcomers sometimes need practice finding their way around. And the Archive hosts story files (and associated manuals, as appropriate) but not advertising for them - it does not provide web-hosting for authors to set up mini-sites.

Uploading a work to the IF Archive is not too difficult, and can be done in two ways. One way is to use the archive's web form at:

```
https://upload.ifarchive.org/cgi-bin/upload.py
```

The other is to create a new page at the Interactive Fiction Database, at:

```
ifdb.org
```

It's then possible to upload the story file to the IF Archive from IFDB. This is easiest all round, since it allows both IFDB and IF Archive to be updated at once.

In either approach, an author chooses and uploads a file, and accompanies it with a name and email address (so that the archive maintainers can verify the legitimacy of the

work). The "About this file" field is for a line or two explaining what the story is -- its full title and any critical information -- and is used in generating the archive index. This is normally much shorter than the "blurb" described earlier. There's also a field to suggest where in the archive the story should be stored, but this is optional and intended chiefly for people expert in how the archive is filed. The archive maintainers will file a new story file in the obvious directory for its format. For Inform works, that means other Z-Machine - "z-code" - or Glulx story files. The maintainers sometimes place the same story file in multiple places in the Archive, using links.

As with all large libraries, it takes the Archive a little while for new acquisitions to be processed. When this happens, one of the volunteer maintainers will email with the official URL from which anyone can now download the story file.

Committing a story to the Archive is meant to be permanent. While the maintainers will happily replace older versions of stories with new improved releases, they are less eager to remove stories entirely. If that doesn't seem appealing, or if we do not want our story to be treated as freeware with essentially unlimited distribution, the Archive may not be a good choice. But it is deeply valued by the IF community, and has saved many works which could otherwise easily have been lost forever. Many contributions important in the history of IF were made by people who are now not easy to trace, and whose websites are long gone. But their work lives on.

§26.5 IFDB: The Interactive Fiction Database

Once the story file has a home online, and a URL (that is, a web address) at which it can be found, it needs to be registered with IFDB:

```
ifdb.org
```

the Interactive Fiction Database. Just as the IF Archive is a repository for stories themselves, IFDB is a database containing information about them - titles, authors, locations, solutions, reviews, recommendation lists and more.

The name IFDB echoes the Internet Movie Database (IMDB), but in some ways it is also like the iTunes Music Store. For one thing, it's a shop-window for what's new, with cover art to catch the eye. For another, IFDB serves as a portal for players to try games directly in their browser. Promoting IF is all about pulling in impulse players -- people who are passingly interested, but might not try the story if there is any significant work involved in setting it up. This is what IFDB is all about.

IFDB is community-editable, like Wikipedia, though editors are required to create an account and log in first -- this is free, of course. A standard form is provided for creating a

new record (accessible by selecting the option to add a story listing). More or less the same information that appears on Inform's library card in the Contents index needs to be copied over: there's space for the author name, story title, genre, and so on. IFDB will also ask for an IFID, a code identifying the story uniquely. Inform generates one of these automatically for each project, and it, too, is on the Library Card. It can always be found by typing VERSION into the compiled story and looking at the line that says

Identification number: //[some letters and numbers]//

The part between the // marks is the IFID. If there's cover art, that can also be uploaded, and good cover art makes a big difference to shop-window-appeal.

The download link should give the most stable URL available. If you have not yet uploaded your story to the IF Archive, you may do so by selecting the "Upload it to the IF Archive" link instead of pressing the "Add a Link" button. The benefits of submitting your story to the IF Archive in this manner are two-fold. One, IFDB will fill in much of the information required by the IF Archive for you. Two, the link to your story will not appear until the IF Archive maintainers move it to its permanent home in the archive, at which point the download link will be automatically updated and presented on the story page.

If you choose to upload your story file to the IF Archive independent of IFDB, then once the story file is safely up at its permanent home on the IF Archive, that is an ideal address to quote here. Otherwise, the URL of the work's own website is best. (Note that the IFDB entry can always be edited later, if the URL moves.)

Commercial works which aren't available as free downloads can be registered on IFDB just the same, and this is almost certainly a good idea.

Some awards for interactive fiction, such as the annual XYZZY Awards, require a game to have an IFDB entry as an eligibility requirement.

§26.6 Competitions, Exhibitions, and Jams

One very common way to get players for IF is to enter the story into an IF competition. The annual IF Competition, often just called IFComp, is the most prestigious and has the widest field, but the Spring Thing, ParserComp, EctoComp, and other events also catch people's attention. Entering a competition is a path of least effort for authors promoting their new work, because the competition organizer usually takes care of hosting and archiving submitted stories, promoting the competition as a whole, collecting votes, and encouraging players to post reviews. Different contests have different arrangements. The ifwiki usually posts a list of current and upcoming competitions, as well as lists of results for those recently past, on the front page:

```
ifwiki.org
```

Some competitions also have their own websites, at least at the relevant times of year.

All the same, there are many IF works that aren't cut out for competition release. Competitions tend to be best for short or medium-short works, because judges don't necessarily have time to play a lot of long stories at once, and sometimes this is a condition of entry.

It's also good for publicity to win one of the annual **XYZZY Awards**. All interactive fiction stories released in a given year are eligible, as long as they are listed on IFDB.

Meanwhile, itch.io hosts many jams every year. A small handful of these are specifically intended for interactive fiction or parser-based adventures, but there are many other jams that allow entrants to put up any game with an appropriate theme, regardless of its format.

```
itch.io/jams
```

lists the calendar of everything currently upcoming.

Finally, if your project is heavily focused on procedural generation - creating or remixing elements on each playthrough - then it may have a natural home at procjam:

```
procjam.com
```

Procjam is a yearly event to "make something that makes something", and welcomes all kinds of generative projects, whether they are games or not.

§26.7 Meetups and Conferences

There are a number of different local groups that get together to play or discuss interactive fiction, including a number that hold remote meetings. Announcements of some of these can be found at

```
intfiction.org/c/general/events/47
```

Joining these groups may provide a context to discuss work in progress, and many are willing to do a group playthrough of games written by group members.

There are also a range of conferences that accept talks or presentations about interactive fiction, both academic conferences and conferences adjacent to the game industry. While

it is not a complete listing, Emily Short's blog attempts to link upcoming events:

```
emshort.blog
```

§26.8 A short concluding homily

It's natural to want to make a huge splash with a story, but in the IF community, instant widespread adulation for any work is pretty uncommon.

For one thing, players tend to play when they get around to it… which may be weeks, months, or even years after the initial release. Reviews trickle rather than flooding in. Appreciation builds slowly. And sometimes works that placed unspectacularly in a competition, or seemed to be overlooked in the annual XYZZY Awards, gradually come to be regarded as classics because of some pioneering technique.

So it's wise (if difficult) not to judge a story's success entirely by its immediate feedback. Even after its debut, a story can often use a little care and attention if it's to reach all its potential fans -- whether that means building further releases, posting hint files or walkthroughs, developing new websites, or approaching outside reviewers.

27. Extensions

§27.1 The status of extensions
§27.2 The Standard Rules
§27.3 Built-in, installed and project-specific extensions
§27.4 Authorship
§27.5 A simple example extension
§27.6 Version numbering
§27.7 Extensions and story file formats
§27.8 Extensions can include other extensions
§27.9 Extensions can interact with other extensions
§27.10 Extensions in the Index
§27.11 Extension documentation
§27.12 Examples and headings in extension documentation
§27.13 Implications
§27.14 Using Inform 6 within Inform 7
§27.15 Defining phrases in Inform 6
§27.16 Phrases to decide in Inform 6
§27.17 Handling phrase options
§27.18 Making and testing use options
§27.19 Longer extracts of Inform 6 code
§27.20 Primitive Inform 6 declarations of rules
§27.21 Inform 6 objects and classes
§27.22 Inform 6 variables, properties, actions, and attributes
§27.23 Inform 6 Understand tokens
§27.24 Inform 6 adjectives
§27.25 Naming Unicode characters
§27.26 Overriding definitions in kits
§27.27 Translating the language of play
§27.28 Segmented substitutions
§27.29 Invocation labels, counters and storage
§27.30 To say one of

§27.1 The status of extensions

The range of simulation offered by Inform's model world is intentionally limited to a core of basic essentials. We could argue at the margins, and the choice of what's in and what's out is partly traditional, but most people find the model reasonable as far as it goes.

Between 1993 and 2006, quite a range of "library extensions" for Inform's predecessor language (Inform 6) was written. Most of these extensions aimed to fill out the model by

simulating other aspects of life, too: money, clothing, pourable liquids. None of these extensions was official and all of them were: it was a free-for-all, and in several cases different authors wrote rival extensions to model the same basic ideas. The development of Inform 7 was strongly influenced by this history and by the recognition that the base of rules and grammar inside a typical modern story are seldom written by a single author. They combine the standard Inform material with extensions by several third parties, together with anything specific to the story in question.

Inform 7 has a more organised idea of extensions, as we shall see. But anyone is free to write an extension on any terms or for any reason. Writers may wish to use the techniques in this chapter to develop private extensions of their own, used in several projects, or to share them with associates but not more widely.

But most writers of extensions do so to contribute to the Inform community, and for the satisfaction of solving a problem. Inform does not recognise anyone's approach to a particular need as "the official solution" - for instance, although the standard Inform distribution includes a copy of Locksmith by Emily Short, that is not the "official" way to make automatically unlocking doors, and anyone is welcome to try a better one.

However, the Inform project does recognise some extensions as "public". Public extensions are the ones archived on the Inform website for the free use of all Inform writers. Those who wish to contribute an extension as a public one are obliged to follow a number of guidelines, which are mostly stylistic points intended to make the range of extensions easier to work with. Extension writers are asked to join in the spirit of these rules and help make the whole cooperative enterprise work harmoniously. Extensions which do play by these rules are also accepted into the Public Library, which makes them easy for all Inform users everywhere to find and obtain them.

Writers who wish to make their extensions public on the Inform website should also be clear that by doing so, they are donating their work to the community on the basis of the broadest form of Creative Commons license: that is, they retain copyright and the right to be identified as the author (and as we shall see they are automatically credited in any work of IF which uses their extension), but are giving unlimited permission to use, circulate and republish their extensions in any form, even as part of commercial works (should that arise). To publish a public extension is a public-spirited act, done for only the reward of a modest acknowledgement.

If the author of an extension has not made it public, or indicated in some other way that it is free to be used without the need for permission, then it would be both polite and prudent to check with the author before publishing something which incorporates his work.

§27.2 The Standard Rules

When any source text is run through Inform, a secret first line is inserted, which reads:

```
Include the Standard Rules by Graham Nelson.
```

The "Standard Rules" file contains the definitions of the basic kinds, phrases, actions and grammar described in this documentation: for instance, it includes lines like

```
A container is a kind of thing.
```

…without which Inform would be lost. Although including the Standard Rules is compulsory, it is treated internally as if it were any other "extension".

What happens when an "Include" sentence is reached is that the sentence is replaced with the whole text of the file in question, often many paragraphs long.

If the file has already been included, then the sentence is simply ignored. This is so that we can have two extensions, each of which needs the other: if A says to include B, and B says to include A, the result is that including one automatically includes the other, so we always get both which ever we ask for - not that there is a hideous infinite regress.

§27.3 Built-in, installed and project-specific extensions

To recap: Inform builds projects from both the source text typed by the author and from Extensions; one of these, the Standard Rules, is always included; others are added as authors please. About 20 are "built-in" to Inform, meaning that they are stored inside the application and always available. Others must be "installed", and each Inform user will have a folder somewhere on his computer which contains these. Users typically obtain these from the Public Library feature in the Inform application, but can also download them directly from the extension writer's website and then use an Install Extension menu option in the application. Either way, the application then squirrels the file away, and it becomes available to any projects that that user may be working on.

It is also possible to have extensions available to just one project. These must be stored in the Extensions subfolder of the project's ".materials" folder, but otherwise are arranged the same as installed extensions - there's an outer folder for each author's name, and extensions are named with a ".i7x" extension within. For example:

```
Mourning Hypercritical.inform
Mourning Hypercritical.materials
    Extensions
        John Siracusa
            Fixing The Finder.i7x
```

When Inform needs to find an extension, it looks here first, then in the installed area, then in its built-in area. That means that we can make our own revised or hacked version of an extension, put it in the ".materials" area, and then have it take precedence over the installed or built-in one. We could even have our own private version of the Standard Rules here.

(This has a number of possible uses - for example, to provide a convenient test-bed when working on an experimental version of an extension.)

§27.4 Authorship

Extensions are identified by author and by name, so that a given author can produce his or her own range of extensions, and need only ensure that these are named differently from each other. If John Smith and Mary Brown each want to write an extension called "Following People", there is no conflict.

The name of an extension, and of an author, should be written in Sentence Capitalisation: that is, upper case for the first letter in each word. (Inform uses this to minimise problems on machines where filenames are read with case sensitivity.) It is permitted for author names to include upper-case letters within words, as with the "G" in "Jesse McGrew". In general it is best to avoid accented or unusual letters in titles and author names, but the standard ISO Latin-1 characters should be allowed - for instance,

```
Étude Pour La Fênetre by Françoise Gauß begins here.
```

The author name must not start with "The", nor contain the words "by", "and" or "version", or contain punctuation, as in "John X. Doe"; the title similarly, except that "and" is permitted. Name and author's name must each be no more than 50 characters long, including any spaces between words.

Authors are asked to use real names rather than cryptic handles like "ifguy", and to use genteel, plausible pseudonyms like "Emily Short" rather than, say, "Drooling Zombie" or "Team Inform". Authors are also asked to use the same author's name for all their own extensions, and (it should go without saying) not to masquerade as anybody else.

Sometimes authorship is complicated. What if Mary Brown finds some Inform 6 code written by John Smith in the mid-90s, and puts an I7 gloss on it to make an I7 extension,

but then Pierre Dupont translates it into French: who's the author of the result? The rule is that the person making the current, latest version is the author listed in the titling line, so we end up with

```
... by Pierre Dupont begins here.
```

But Mary and John deserve their credits too: see the next section for how to give them.

§27.5 A simple example extension

Extensions are plain text files, and can be created with any text editor. (It is sometimes said that "there is no such thing as plain text", there being so many ways to represent exotic characters: so to be precise, an extension is a text file with the Unicode UTF-8 encoding, either with or without a BOM marker, using any of the possible forms of line-ending (Unix, Windows, Macintosh, or Unicode line divider). This is a detail which will only matter if the extension contains accented letters or other exotica.)

Extensions look very much like passages of Inform source, because except for a special introductory and concluding sentence, and one convention, that is all they are:

```
The Ducking Action by Beatrix Potter begins here.

"An action for ducking one's head."

Ducking is an action applying to nothing. Report ducking: say "You duck!" Understand "duck" as ducking.

The Ducking Action ends here.
```

Not a useful or interesting extension, but those few words add a whole new action and everything needed to make it work. It is Inform's ability to mix up rooms, things, kinds, grammar, phrases and rules, in more or less any order, which makes it possible for extensions to work.

The introductory sentence must be placed as the only content of line 1 of the file, which must not contain comments, and has to be written in exactly the correct form. Inform checks this very carefully when performing its census of installed extensions, on each translation of the text. (In case the extension's title is a plural, we are allowed to write "begin" and "end" instead of "begins" and "ends". For instance, the last line of the standard rules is "The Standard Rules end here.")

The "one convention" mentioned above is that if a double-quoted text is placed immediately after the beginning sentence (and with no intervening comments), then it is taken to be a short description of the extension's content called the "rubric". Hence the

line:

```
"An action for ducking one's head."
```

Providing a rubric is helpful, because it enables Inform to give a meaningful listing even for an as-yet unused and unindexed extension, and because it helps the Inform website to produce better directories. Note the word "short": such text is likely to be truncated if it exceeds 500 characters.

A second double-quoted text can also, optionally, be added in yet a third special starting paragraph. This is to provide additional credits to people who have contributed to this or earlier versions. For instance:

```
The Ducking Action by Beatrix Potter begins here.

"An action for ducking one's head."

"based on original Inform 6 code by Marc Canard"
```

Note the typical style here: it's a phrase rather than a sentence, and neither starts with an upper-case letter nor ends with a full stop. (The additional credit is then used in documentation and also in the VERSION text of any Inform story file using the extension.)

Example

448. **Modern Conveniences**
Exemplifying the kind of source we might use in writing extensions for kitchen and bathroom appliances.

RB 8.5 Kitchen and Bathroom

§27.6 Version numbering

As we have seen, extensions are referred to by name and author, but they can also (optionally) be referred to by version. For instance:

```
Include version 2 of the Ducking Action by Beatrix Potter.

Version 1.2.4 of the Ducking Action by Beatrix Potter begins here.
```

Version numbers should consist of one to three whole numbers divided by dots, with no negative numbers allowed. Thus "5", "3.3" and "2.1.71652" are all valid as version numbers, but "-4" and "3.1.2.5" are not. Any numbers not specified are taken to be 0: thus "3.3" means the same as "3.3.0", and "5" means the same as "5.0.0".

In versions of Inform before 2022, versions of extensions were also allowed to be written in the form "N/YYMMDD", as in this example:

```
Version 6/040426 of the Ducking Action by Beatrix Potter begins here.
```

The material after the slash '/' was expected to be a date, so that 040426 would mean 26 April 2004. In order to preserve compatibility with old extensions, Inform continues to allow this notation, but treats it as equivalent to writing "N.0.YYMMDD", though with any leading 0s trimmed. So the above sentence is equivalent to writing:

```
Version 6.0.40426 of the Ducking Action by Beatrix Potter begins here.
```

Extensions are usually intended to be shared and passed around between Inform users, and good use of version numbering can be a huge help to those users; and it's helpful if we can agree as a community on what good version-numbering is. Because of that, the Inform project tries to use a widely-recognised Internet standard called "semantic version numbering".

For full details see semver.org, but for Inform purposes the following fairly simple rules should be enough. "Semantic" just means that version number changes should communicate something meaningful. So, whenever an extension author puts out a new version of an extension, the extension number should change in a way that signals how drastic the change will be.

In this system, the three possible numbers X.Y.Z are called the "major", "minor" and "patch" numbers. Every time an extension is changed and re-released, even just informally among friends but certainly if posted somewhere on the Internet, X, Y or Z should change. The rules are:

(X) If the extension has changed so much that Inform projects using it will need to be changed in order to keep on working - for example, if a "To…" phrase has been taken out, or the name of a kind changed - then X should be increased. Y and Z then usually go back to 0. This is a "major version".

(Y) If the extension provides new features but doesn't do anything to change the way its existing features are used, then X can stay the same but Y should increase, and Z then usually rolls around to 0. This is a "minor version".

(Z) If the extension has changed only to fix bugs, or make its existing features work more efficiently, or provide better documentation or examples, then X and Y can stay the same but Z should increase. This is a "patch version".

So, for example, a user who currently has version 3.2.7 can update to 3.2.8 without really investigating. That same user can update to 3.3, 3.4, … without any trouble, choosing either to use or ignore whatever new features they are presenting. But the user knows that moving up to version 4 might well require some work - a project using version 3.Y.Z will likely need writing to adopt version 4.

Now let's turn to "Include" sentences. A request like:

```
Include the Ducking Action by Beatrix Potter.
```

will be happy with any version of the extension at all, whether numbered or not; but

```
Include version 2.4 of the Ducking Action by Beatrix Potter.
```

will only accept the extension if its version number is "compatible" with 2.4, which means, if it is 2.4 or later, but still belongs to the same major version, "2". So if we write this inclusion sentence, but the version we have installed is version 3.1, Inform will give a problem message. The fix may well be as simple as changing the inclusion sentence to match - but it may not, because a change in major version number is a signal that things have changed a lot inside the extension (see above).

During play of any story compiled by Inform 7, typing VERSION lists various serial numbers of the pieces of software used to make it. The list concludes with names, authors and version numbers of any extensions used. So every author whose work contributes to a story automatically gets a modest credit within it. The same list can be printed, at the discretion of the designer, using the textual substitution:

say "[the/-- list of extension credits]"

This text substitution expands to one or more lines of text crediting each of the extensions used by the current source text, along with their version numbers and authors. Extensions whose authors have chosen the "use authorial modesty" option are missed out.

If we want our extension to go uncredited - perhaps if it is a low-level enabling sort of thing, for instance - we can place the following sentence inside the definition of the extension:

```
Use authorial modesty.
```

The same sentence placed in the body of a source text causes all extensions by the same author as the main source text to go uncredited. In other words, if Isaac Miggins writes a source text and includes, say, Unlikely Events by Isaac Miggins, then this extension will go uncredited in the VERSION command.

A complete list, undiluted by modesty, can always be obtained using:

say "[the/-- complete list of extension credits]"

This text substitution expands to one or more lines of text crediting each of the extensions used by the current source text, along with their version numbers and authors. Every extension is included, even those whose authors have opted for "use authorial modesty".

§27.7 Extensions and story file formats

Inform compiles to several different story file formats, and in each case uses only a small part of their abilities - especially when it comes to fancy tricks with the keyboard or screen. So people may well want to write extensions which provide access to some of these tricks (like "Basic Screen Effects", included in the standard Inform distribution, but more so). Unfortunately, these tricks are very likely to fail to compile - or fail to work - on some of the possible story file formats, so the resulting extension would probably go wrong (and mysteriously wrong) for users who have chosen a different format.

Inform therefore provides a way for extensions to declare the formats they are compatible with. All that is required is to add a proviso in brackets after the title is declared:

```
Version 2 of Basic Screen Effects (for Z-Machine version 8 only) by Emily Short begins here.
```

Other examples might be "(for Glulx only)", or "(for Z-machine only)". If no such proviso is given, the extension is assumed to be compatible with every story file format.

Extensions are also able to include material which is only used on some story file formats and not others - in principle, this might allow the same facilities to be provided to the author whatever story file format is used, but to achieve these effects differently depending on the current Settings. The convention here is exactly like "not for release": if a heading or subheading in the source text contains a bracketed proviso, then the material under that heading (and under its dependent subheadings) will be ignored if the current story file format does not match. For example:

```
Section 2.3G (for Glulx only)

To reveal the explosion:
    [...the Glulx way...]

Section 2.3Z (for Z-machine only)

To reveal the explosion:
    [...the Z-machine way...]
```

would ensure that "reveal the explosion" works nicely whichever story file format is used.

Example

449. Tilt 3
Displaying the card suits from our deck of cards with red and black colored unicode symbols.
RB 12.1 Typography

§27.8 Extensions can include other extensions

Extensions can themselves contain "Include…" sentences asking for other extensions to be included. An extension might, for example, start like this:

```
Version 1 of Basic Help Menu by Emily Short begins here.

Include Menus by Emily Short.

...
```

A project which asks to include "Basic Help Menu" will then also include "Menus", even though the author might never even realise that. Indeed, the author could also have asked to include "Menus", not realising that "Basic Help Menu" was going to ask for the same thing.

So the same extension is often requested multiple times. This is fine if the version numbers in the requests are compatible, but they might not be. For instance, suppose the main source text asks to include version 2 of extension X, and also to include extension Y. Suppose further that Y contains a request to include version 4 of X. We now have two different requests for X, and they contradict each other - the major version of X cannot be both 2 and 4 at the same time. So Inform will produce a problem message in this case.

But in cases where it is possible for everyone to be satisfied, Inform will try to find a solution. If one extension asks for version 2.3 of X, and another asks just for X, and a third asks for version 2.7.2 of X, then Inform will work out that any version number in the range 2.7.2 up to (but not including) 3 will be fine. If it can in fact find such an extension, it will

then use it. So if the user has version 2.8.17 installed, everything is fine.

If an extension does include other extensions, it is good style to place the "Include…" sentence(s) as early as possible after the introductory sentence, just so that human readers looking at the text of the extension can see these dependencies easily.

§27.9 Extensions can interact with other extensions

When one extension is being used, it's probably only one among several. A really general-purpose extension might want to behave differently depending on which other extensions are also present. This can be achieved using headings which are "for use with" (or "without") other extensions. For instance:

```
Chapter 2a (for use with Locksmith by Emily Short)
```

specifies that everything under this heading (and its subheadings, if any) will be ignored unless the extension Locksmith by Emily Short is included. Conversely,

```
Chapter 2b (for use without Locksmith by Emily Short)
```

will be ignored unless it isn't included. This allows an extension to give two variations on the same material - one if Locksmith is present, the other if not.

Headings can also replace portions of extensions which have been included. For instance:

```
Section 6 - Hacked locking (in place of Section 1 - Regular locking in Locksmith by Emily Short)
```

places the source text under the new heading in the place of the old (which is thrown away). If there should be two or more headings of the same name in the given extension, the first is the one replaced; if two or more headings attempt to replace the same heading in the given extension, the final attempt in source text order is the one which succeeds; and finally, heading dependencies like the above are scanned in a top-down way. Thus, if we have:

```
Chapter 2a (for use with Locksmith by Emily Short)
...
Section 1 - Hacked marbles (in place of Section 4 in Marbles by Peter Wong)
...
```

and we don't include Locksmith, then the replacement of Section 4 of Marbles is not made, because Section 1 - Hacked marbles is subordinate to the Chapter 2a heading

which we've told Inform to ignore.

If the name of the heading to replace contains the word "in", it's a good idea to use quotation marks for clarity:

```
Section - Hacked questions (in place of "Section 4 - Phrase used to ask questions in closed mode" in Questions by Michael Callaghan)
```

§27.10 Extensions in the Index

As soon as a project has successfully been translated, its Index is brought up to date: pages of the index record all the kinds and what they are for, all the phrases which can be used, and so on. Any kind or phrase created in an extension is automatically included. The extension's presence in the project is itself recorded - the Contents index for any project contains a brief list of all extensions used in that project, along with their authors and version numbers.

The Kinds index aims to give the reader a brief note of what each kind is intended for. We can provide for this by writing a sentence like so:

```
The specification of player's holdall is "Represents a container which the player can carry around as a sort of rucksack, into which spare items are automatically stowed away."
```

There is no need to specify the properties which apply: that is all done automatically. "Specification" is a sort of pseudo-property used just for this: we can also give specifications to kinds of value and to actions, and these are similarly used in the Index pages.

Every extension has the right to its own set of headings and subheadings, independently of those used by the main source for the work or by any other extension which may be included. (So if the extension is divided into four sections and finishes on Section D, say, that doesn't mean that Section D will continue outside the extension as the main source of the story runs on.)

Extensions should, of course, be written so that they never produce Problem messages, so at first sight it appears that these headings will never be outwardly visible. In fact, though, Problems do occasionally turn up in extensions, usually when the user has made a mistake, or when two inconsistent extensions are used in the same project. But more importantly, the headings in an extension are used when indexing phrases (and also actions) to group similar phrases together. For instance, the Standard Rules contain the heading:

```
Section SR4/7 - Searching and sorting tables
```

The half-dozen phrases defined in this section of the Standard Rules are then indexed under the subheading "Searching and sorting tables": Inform looks for a hyphen in the heading and then uses any text which follows the hyphen. (If there is no hyphen, the entire heading text is used.)

If an extension contains no headings, its phrases (or actions) are indexed simply as "Miscellaneous".

Finally, any phrase or variable defined immediately under a heading whose name ends in the word "unindexed" will be omitted from the Phrasebook or Contents index respectively. (That won't apply to definitions under subheadings of the heading.) This is intended so that technical apparatus used only inside the extensions can be concealed from the outside user's immediate view. Inform as it is presently constituted does not allow extensions to make fully private definitions, but this feature at least allows them to make unadvertised ones.

§27.11 Extension documentation

A basic mechanism for documenting extensions is built into Inform. For many extensions, this will probably do instead of a manual; for more complex ones, it should still prove a useful supplement to one.

As described in Chapter 2 above, whenever an extension is installed, its documentation is made available to the user. Such text should be written concisely, while giving examples wherever appropriate. Stylistically, it should ideally follow the model of the main Inform documentation: just as an extension expands the standard rules, so its documentation expands this manual. "We need…" is preferred to "You need…", and so on: we're all in this together.

In order to be recognised as documentation, this text should appear at the foot of the extension file, *after* the compulsory end sentence. The first paragraph must have exactly the following form, with a skipped line before and after:

```
---- DOCUMENTATION ----
```

For instance, the "Ducking Action" example might end:

```
...
The Ducking Action ends here.

---- DOCUMENTATION ----

This is a modest extension, with much to be modest about. It allows us to use a new action for ducking, as in ducking the player's head (not as in ducking a witch). Ducking will do nothing unless rules are added:

	Instead of ducking in the Shooting Gallery, say "Too late!"

...
```

We obtain indented code examples by beginning a line with a tab. A double indentation can be got with two tabs in a row, and so forth. (Beware: some text editors, or emailers, flatten tabs into a row of four or perhaps eight spaces each. Inform will not recognise such a line of spaces as a tab.)

Note that text in square brackets should be avoided in the documentation, because that's taken as being comment matter on the extension, and omitted.

Tables should be similarly indented, and should begin with the word "Table ...": the top line is taken to be the name of the table, and subsequent lines are tab-divided columns. Inform will automatically group this into a table, like so:

Table of Exemplariness

stellar object	example
galaxy	"Andromeda Galaxy M31"
star	"Sirius"
planet	"Neptune"
moon	"Enceladus"
dwarf planet	"Ceres"
plutino	"38628 Huya"
cubewano	"Easterbunny"

(Footnote: Since the first appearance of this book, Easterbunny has been renamed Makemake, the creator god in the mythology of the people of Easter Island.)

§27.12 Examples and headings in extension documentation

Extensions with very large amounts of documentation can, if the author chooses, divide the material up using headings and/or subheadings. These must be written as paragraphs exactly like so:

```
Chapter: Avoiding Events

Section: Ducking examinations and tests
```

Inform will then typeset them to stand out, will number them automatically, and will add a table of contents at the top of the page. (For most extensions, the documentation will be short and sweet, and this would just be clutter: headings and subheadings are best used only where the text would otherwise be difficult to read.)

Any extension's documentation can contain Examples, just as the main Inform documentation does: these are automatically labelled A, B, C, … rather than given numbers, to ensure that they do not clash with the numbering used in the built-in chapters. (The labels may be helpful in writing an extension's documentation: we can write, for instance, a note such as "see Example C below".)

Examples must be given last in the documentation, and there can be up to 26 of them, though most extensions will need one example at the most, and some will have none at all. Each example must begin with a paragraph exactly like so:

```
Example: ** We Must Perform a Quirkafleeg - Ducking to avoid arrows as one proceeds east across battlements.
```

Again, there must be a skipped line before and after. The row of asterisks must be *, **, *** or ****, just as in the main documentation, which we should follow on all points of style. The rest of the line contains the title, a hyphen, and then the description. The title should be given with Each Word except Prepositions and Similar Things Capitalized, while the description should look like a sentence, and end with a full stop.

The text of the example follows, of course, and continues until the end of the file, or the next "Example:" line, whichever comes first.

Each example should (normally) contain one single, complete, story, long enough to demonstrate the use of the extension and to have a little flavour to it, but not so long that the reader gets lost. It should have a title, which should match the name of the example (in the case above, "We Must Perform a Quirkafleeg"). It should conclude with a paragraph defining a test:

```
Test me with "east / duck / east / jump / east / duck / east / rescue esmerelda".
```

The idea is that typing one single command, TEST ME, into the resulting story should show off what the extension does.

When an extension contains more than one example, they should be given in order of

asterisk rating, that is, starting with the * examples, then the ** examples, and so on up.

Extension documentation can provide "paste" buttons, much like the examples in this book. For example:

```
Here is a sample -

    *: "Coriander"

    Include Herbs by Charlotte Quirke.

    The Herb Marketing Centre is a room.

If we want to add some content -

    The coriander is a herb. Understand "cilantro" as the coriander.
```

Note that the paste button, denoted "*:", pastes in the text following it, but only as far as the next paragraph of unindented documentation - here, the one beginning "If we…". (But of course, an extension can have multiple paste buttons if desired.)

§27.13 Implications

Extensions often need to define new kinds or properties, which we want to make as helpful as possible for the user. In particular, we want them not to require additional work for the author just to obtain the effect which seems only natural.

For example, consider Inform's built-in "locked" property. If a door is locked, then it cannot be opened, which seems fair enough. But if the player tries to unlock the door, he might then find the following response:

```
That doesn't seem to be something you can unlock.
```

Which does not seem right. In real life, almost all locked items have outwardly exposed locks which it is perfectly sensible to try to unlock, given a key. The problem is that our door has the "locked" property, but not the "lockable" one.

The Standard Rules solve this problem by including the following line:

```
Something locked is usually lockable.
```

This ensures that any door said by the author only to be "locked" will be "lockable" as well, and adds a small but worthwhile touch of realism.

Such a sentence is called an "implication", as it is in the form "Condition A implies Condition B". Note that the two conditions must consist of either/or properties with or

without kinds attached. Thus:

```
A room in the Open Desert is usually lighted.
```

will not work because "a room in the Open Desert" is a more complicated grammatical construction than, say, "lighted" or "a lighted room": it contains a relative clause. Inform can only deal with simple implications.

Inform never overrides certainties with mere implications, and is cautious about allowing them to build overly long chains of argument. This is to prevent the following kind of difficulty:

```
An open door is usually closed. A closed door is usually open.
```

Implications work just the same for values which aren't objects, so:

```
Colour is a kind of value. The colours are red, green and blue.
A colour can be zesty or flat. A colour can be bright or dull.
Red and blue are bright. Blue is flat.

A bright colour is usually zesty.
```

results in red being zesty, but blue and green being flat; blue because the source text explicitly says so (which trumps the "usually"), and green because this isn't a bright colour, so the implication doesn't arise.

Implications have not been mentioned up to now since they are only really needed by extensions, but also because they can be tricky, with unforeseen consequences. We should handle them with care.

§27.14 Using Inform 6 within Inform 7

The current Inform, "Inform 7", had a low-level precursor unsurprisingly called Inform, which ran through versions 1 to 6. What made Inform 6 low-level was that its style of coding was much more like traditional programming: it reads as a simple form of C, or an elaborate form of assembly-language, but with some interactive fiction tweaks.

That language is still used inside today's Inform project as a way to express very low-level operations. What happens to code like that is now very different (it is compiled into Inter, an intermediate-level representation used inside Inform, and no longer by the Inform 6 compiler). But the notation is the same, and the practical effect is that it is as if we are writing I6 code.

The final sections of this chapter show how such I6 code can be mixed directly in with natural-language source text. The remaining pages will therefore make little or no sense to those who do not already know I6 notation, and in any case, such programming is really a last resort - it is always best to write regular source text than to resort to so-called "inclusions" of I6. Ideally, all I6 content would be confined to extensions (and this may be mandated in future releases of Inform), and even writers of extensions are asked to pare down their usage of I6 to the minimum necessary.

The methods for incorporating I6 code into I7 have been designed with this in mind, that is, to encourage people to use I6 in as self-contained a way as possible: in particular to isolate the relatively few functions which need to be written in I6, and to give them natural language expression.

Finally, anyone hacking with I7 for a while is likely to become curious about the Basic Inform or Standard Rules extensions, and to look at the text which sets up the Inform language and world model. These extensions are, of course, no secret, but can be misleading to read. For one thing, they appear to have great freedom to set up the world model as it pleases, but in fact the I7 compiler may well crash unless certain things are done just so in the Standard Rules: they depend on each other.

Moreover, the Basic Inform and Standard Rules extensions use a number of syntaxes which are not documented in this chapter: these are constantly being altered, and it would not be safe to imitate them. Any I6-related syntax which is not documented in this chapter may be removed or changed in effect at any time without warning, for instance in an update of Inform to fix bugs.

§27.15 Defining phrases in Inform 6

The phrases described in this documentation, such as "end the story", are all defined in the Standard Rules, and are for the most part defined not in terms of other I7 phrases but instead reduced to equivalents in I6. For instance:

```
To end the story: (- deadflag=3; story_complete=false; -).
```

The notation "(-" and "-)" indicates that what comes in between is I6 code. The minus sign is supposed to be a mnemonic for the decrease from 7 to 6: later we shall use "(+" and "+)" to go back up the other way, from 6 to 7.

When a phrase is defined as containing only a single command, and that command is defined using I6 - as here - it is compiled in-line. This means that the phrase "end the story" will always be translated as "deadflag=3; story_complete=false;", rather than being translated into a call to a suitable function whose only statement is "deadflag=3;

story_complete=false;".

This is an easy case since the wording never varies. More typical examples would be:

```
To say (something - number): (- print {something}; -).
To sort (T - table name) in (TC - table column) order:
    (- TableSort({T}, {TC}, 1); -).
```

When the braced name of one of the variables in the phrase preamble appears, this is compiled to the corresponding I6 expression at the relevant position in the I6 code. So, for instance,

```
say the capacity of the basket
```

might be compiled to

```
print O17_basket.capacity;
```

because "{something}" is expanded to "capacity of the basket" (I7 code) and then translated to "O17_basket.capacity" (I6 code), which is then spliced into the original I6 definition "print {something};".

Braces "{" are of course significant in I6. A real brace can be obtained by making the character following it a space, and then I7 will not attempt to read it as a request for substitution.

It's also possible for the pair of characters "-)" to occur in I6 code, for example here:

```
for (i=3 : i>0 : i--)
```

and I7 will read the "-)" as terminating the I6; we can get around this with an extra space:

```
for (i=3 : i>0 : i-- )
```

Warning: Inform 6 uses a restricted character set, allowing use of most of the accented characters in ISO Latin-1 (those found in a set called ZSCII) but little beyond that. It's therefore hazardous to use any exotic Unicode characters in an inclusion.

Example

450. **Pink or Blue**
Asking the player to select a gender to begin play.
RB 5.2 Traits Determined By the Player

§27.16 Phrases to decide in Inform 6

There are basically three forms of phrase in I7: phrases which do something, but produce no value or opinion as a result; phrases to decide whether or not something is true; and phrases to decide on a value. We have already seen examples of writing the first form in I6:

```
To say (something - number): (- print {something}; -).
```

Here the I6 form is required to be I6 routine code in void context, that is, it will normally be one or more statements each of which ends in a semicolon (unless there are braced code blocks present). In this case, we have just one I6 statement, ending in a semicolon.

An example of a phrase to decide whether something is true would be:

```
To decide whether in darkness: (- (location==thedark) -).
```

Here the I6 code providing the definition must be a valid I6 condition, and be in round brackets, but there is no semicolon.

Lastly, an example of a phrase to decide on a value:

```
To decide which number is the hours part of (t - time): (- ({t}/60) -).
```

Again, this is a value in I6 as well: no semicolon. It is probably safest to place the value in round brackets.

§27.17 Handling phrase options

The Standard Rules use the Inform list-writer with the following definition, which shows how a much more complicated I6 routine can be given a natural-language expression.

```
To list the contents of (O - an object),
    with newlines,
    indented,
    giving inventory information,
    as a sentence,
    including contents,
    including all contents,
    tersely,
    giving brief inventory information,
    using the definite article,
    listing marked items only,
    prefacing with is/are,
    not listing concealed items,
    suppressing all articles
    and/or with extra indentation:
    (- I7WriteListFrom(child({O}), {phrase options}); -).
```

This can be used by, say:

```
list the contents of O, as a sentence, using the definite article
```

"{phrase options}" is a special substitution: it is a bitmap which assigns the given options one bit each, starting with the least significant bit for the first-mentioned option ("with newlines" above) and going up to the most significant bit for the last ("with extra indentation").

§27.18 Making and testing use options

Use options (see Chapter 2 above) manifest themselves in the I6 code generated by I7 as constants which are either defined, or not. For instance, the "use American dialect" option results in the constant DIALECT_US being defined, a constant which otherwise would not be. Some use options define the constant as a particular value, others simply define it (so that I6 gives this constant the value 0).

New use options can be created as in the following examples, which are found in the Standard Rules:

```
Use American dialect translates as (- Constant DIALECT_US; -).
Use full-length room descriptions translates as (- Constant I7_LOOKMODE = 2; -).
```

Most Inform users will not need to test whether a use option is currently set: after all, they will know whether or not their own story uses American dialect. But an extension does not know what use options apply in the story which is using it. An extension which needs

to print a list, using its own formatting, might want to know whether "use serial comma" is set. Or it might want to speak differently in American dialect.

To test for American dialect, we should ideally not use I6 to look for the constant DIALECT_US using #ifdef: there is no guarantee that this constant will not be renamed at some point. Instead we can perform the test directly in I7:

```
if the American dialect option is active, ...
```

and similarly for all other named use options. The adjectives "active" and "inactive" have the obvious meanings for use options. This means it's possible to describe the current options like so:

```
say "We're currently using: [list of active use options].";
```

The result might be, say,

```
We're currently using: dynamic memory allocation option [8192], maximum text length option [1024], maximum things understood at once option [100], American dialect option and fast route-finding option.
```

This may be useful for testing purposes.

Use options can also allow the writer to raise certain maximum values. If we write an extension which needs some I6 array, say, and therefore has some limitation - for instance a footnotes presenter which can handle at most 100 footnotes before its array space runs out - it would obviously be cleaner to allow this maximum to be raised. We can set this up like so:

```
Use maximum presented footnotes of at least 100 translates as (- Constant MAX_PRESENTED_FOOTNOTES = {N}; -).
```

With such a definition, the number given is the default value, and the I6 source is included whether or not anybody uses the option: the default value being given if nobody does. The text "{N}" is replaced with the value. So the above definition normally results in this being defined:

```
Constant MAX_PRESENTED_FOOTNOTES = 100;
```

but if the user writes

```
Use maximum presented footnotes of at least 350.
```

then instead the I6 inclusion becomes:

```
Constant MAX_PRESENTED_FOOTNOTES = 350;
```

The I6 constant MAX_PRESENTED_FOOTNOTES can then be used as the size of an array, for instance.

Finally, note that it is legal to define the same use option more than once, but only if it has exactly the same meaning each time it is defined. (This is allowed so that multiple extensions all needing the same definition can safely make it, and still be used together.)

§27.19 Longer extracts of Inform 6 code

Whole routines, object and class definitions (or any other directives) can be pasted in wholesale using sentences like so:

```
Include (-
[ ExtraFunction a b; return a*b; ];
-).
```

Such inclusions are pasted into the final compiled code at the end of the file, after the I6 grammar has been declared.

In such extracts, we sometimes need to refer to objects, variables or values which can't be described using I6: or rather, which can be described, but we don't know how. To this end, any text in an inclusion written in "(+" and "+)" parentheses is treated as an I7 value, and compiled accordingly, with all type-checking waived for the occasion. For instance:

```
Include (-
Global my_global = (+ the tartan rucksack +);
-).
```

Here "the tartan rucksack" is translated into "O18_tartan_rucksack", or something similar: the I6 object created to represent the rucksack. Thus the actual line of code produced is

```
Global my_global = O18_tartan_rucksack;
```

The material between "(+" and "+)" is generally treated as a value, and thus compiles to the I6 form of that value. But it could also be a property name, which compiles to the I6

form in question, or a defined adjective, which compiles to the name of the routine to call which tests whether that adjective is true.

Three warnings. The material in "(-" and "-)" is not quite treated as literal. Certain characters cause Inform to react:

1. Beware of accidental "(+" usage - for instance,

```
Include (-
[ MyCleverLoop i; for (++i; i<10; i++) print i; ];
-).
```

looks reasonable, but contains "(+" and "+)". Spaces around the first "++" would have been enough to avoid this one; "+)" is only significant where it follows a "(+".

2. Beware of placing an "@" character in the first column, that is, immediately following a new line. (In template code this marks off paragraph divisions.) So for instance,

```
Include (-
[ Set_Stream ret;
@glk 67 ret;
];
-).
```

is tripped up by the Glulx assembly language opcode "@glk" because this occurs in column 1. Indenting it with a little space or a tab is enough to avoid the problem.

3. Be careful if you're creating an I6 variable holding initialised I7 text. For example,

```
Include (-
Global saved_optional_prompt = (+ "!!>" +);
-).
```

looks as if it will work, but doesn't, for reference-counting reasons we needn't go into; instead you need

```
Include (-
Array sop_storage --> PACKED_TEXT_STORAGE "!!>";
Global saved_optional_prompt = sop_storage;
-).
```

But it's far better to avoid initialising text variables from I6 entirely. The same problems arise with constant lists.

It should also be noted that the I6 syntax recognised inside "Include (- ... -)" is slightly restricted compared to the full range recognised by the stand-alone Inform 6 compiler. In particular:

1. Only new-style "for" loops with colons in the header are allowed, so that "for (i=0: i<10: i++)" is okay but "for (i=0; i<10; i++)" is not. Moreover, "for" loops cannot contain empty clauses.

2. Local variable names are not allowed to be the same as an I6 statement keyword: for example, "style" and "spaces" are not allowed.

3. The (undocumented) Inform 6 function "indirect()" is not supported. But since "indirect(A)" is equivalent to "A()", which does work, this is no real loss. Similarly, the "glk()" function is not supported: function calls to BasicInformKit should be used instead.

4. Conditional compilation cannot be placed around cases in a "switch" statement.

5. Compile-time constant expression evaluation can be used with arithmetic operations, so "Constant FOO = BAR + 1;" is okay, but not with bitwise or logical operations, so "Constant FOO = (BAR | 1);" does not work.

6. Calculated values cannot occur as assembly-language operands.

7. Calculated values can be used for array extents, but need to be put in brackets. For example:

```
Include (-
Array unit_captured_text --> (UNIT_CAPTURE_BUFFER_LEN + 1);
-).
```

Example

451. **Status line with centered text, the hard way**
A status line which has only the name of the location, centered.
RB 12.2 The Status Line

§27.20 Primitive Inform 6 declarations of rules

By writing a sentence like this:

```
The underground rule translates into I6 as "UNDERGROUND_R".
```

we create a new rule, the "underground rule", and also notify Inform that it will have no definition as I7 source text: instead, it will be provided as an I6 routine called

"UNDERGROUND_R". We can define this with an Include like so:

```
Include (-
[ UNDERGROUND_R;
    if (real_location hasnt light) { RulebookSucceeds(); rtrue; }
    rfalse;
];
-).
```

The rule should return false if it wants to make no decision, but call either RulebookSucceeds or RulebookFails and return true if it does. These routines can optionally take an argument: which will be the return value from the rulebook.

Note that UNDERGROUND_R itself has no arguments. In the case of an action based rulebook, the I6 variables noun, second and actor can be referred to, while for a value based rulebook the parameter is stored in the I6 global variable parameter_object (which is not necessarily an object, in spite of the name).

We can put this rule into a rulebook in the same way that any named rule can be:

```
The underground rule is listed in the spot danger rules.
```

§27.21 Inform 6 objects and classes

As might be expected, I7 compiles an I6 class for each kind, and an I6 object for each of its own objects. We can meddle with its compilation process here using a further refinement of Include. For instance, suppose we want the I6 class definition for things to come out containing a property like this:

```
Class K2_thing ...
    with marmalade_jar_size 6,
    ...
```

How to arrange this? One way is to create an ordinary I7 property, like so:

```
A thing has a number called marmalade jar size. The marmalade jar size of a thing is usually 6. The marmalade jar size property translates into I6 as "marmalade_jar_size".
```

(Without that last sentence, the property won't get any familiar name.) But sometimes we need more, and want to actually write new material to go into the definition. This can be done like so:

```
Include (- with before [; Go: return 1; ], -) when defining a vehicle.
```

This glues in a new property to the class compiled to represent the I7 kind "vehicle". (See the DM4 for why. However, since the entire actions machinery is different in the I7 world, note that "after", "react_before" and "react_after" no longer have any effect, and nor does "before" for rooms.)

And similarly:

```
Include (- has my_funny_attribute, -) when defining the hot air balloon.
```

If we need a particular I7 object or kind to end up with a particular I6 name, we can write:

```
The whatsit object translates into I6 as "whatsit".
The thingummy kind translates into I6 as "thingummy_class".
```

WARNING: The "Include (- … -) when defining …" usage still works for the moment (except in projects compiled to C at the command line, where it may fail), but it is deprecated and likely to be removed in later versions of Inform. Avoid it if at all possible.

§27.22 Inform 6 variables, properties, actions, and attributes

I7's variables are usually compiled as entries in an array rather than as I6 variables. However, we can instead tell Inform to use an existing I6 variable (either one that we declare ourselves, or one in the I6 template layer). For example:

```
Room description style is a kind of value. The room description styles are Brief, Verbose and Superbrief.
The current room description style is a room description style that varies.
The current room description style variable translates into I6 as "lookmode".
```

This is a feature provided to help I7 source text to use variables internal to the I6 template code. It can, if really necessary, also be used to give I7 names to entirely new I6-level variables, created like so:

```
Include (- Global my_variable = 0; -).
```

This style of hybrid coding is really not encouraged.

I7's properties are compiled sometimes as I6 properties, sometimes as I6 attributes, sometimes as bits in a bitmap somewhere. However, we can override I7 by telling it that one of its property names is equivalent to an already-existing I6 property or attribute: if

so then I7 will use that name and will not compile any directive to create it. For example:

```
The switched on property translates into I6 as "on".
The initial appearance property translates into I6 as "initial".
```

We do not need to translate "switched off", the opposite to "switched on": I7 will now compile this to "~on".

Lastly, actions can also be translated (though it's usually better to translate their rules instead and invent new I7 actions covering them):

```
The unlocking it with action translates into I6 as "Unlock".
```

§27.23 Inform 6 Understand tokens

The parser which deciphers the player's typed commands is written in I6, and many of the basic tokens of Understand grammar are implemented as "general parsing routines" (GPRs), the specification of which is described fully in the Inform 6 Designer's Manual. I7 translates much of the source text's Understand grammar into GPRs, and once again we can bypass this process and supply an Understand token directly as an I6 GPR. For example:

```
The Understand token squiggle translates into I6 as "SQUIGGLE_TOKEN".
```

We then have to include a routine of that name into I7's output using the "Include" instruction, on which more later.

This creates a token "[squiggle]"; so for instance if the source text contains:

```
Understand "copy [squiggle]" as ...
```

then Inform would parse the command COPY FIGURE EIGHT by calling the SQUIGGLE_TOKEN routine as a GPR with the word marker at 2, that is, at the word FIGURE.

As always, this should be done only where there seems no better way, or where speed is very important. For any fairly simple range of possibilities, it's better to use the techniques in the Understand chapter, or to use unit specifications.

§27.24 Inform 6 adjectives

There are three ways to specify that an adjective is defined at the I6 level. For example:

```
Definition: a number is prime rather than composite if I6 routine
    "PRIMALITY_TEST" says so (it is greater than 1 and is divisible only by itself and 1).
```

Inform now actually tests if a number N is prime by calling PRIMALITY_TEST(N), and it assumes that we have also included such a routine in the output. The routine is expected to return true or false accordingly.

The text in brackets does nothing functional, but is the text used in the Lexicon dictionary part of the Phrasebook index for the user's benefit; it should be a brief definition. Extension authors are asked to provide these little definitions, so that their users won't be confused by blank lexicon entries.

The second way makes a more capable adjective, since it can not only be tested, but also made true or false using "now". For example:

```
Definition: a scene is crucial if I6 routine "SceneCrucial" makes it so
    (it is essential to winning).
```

The difference here is "makes it so", not "says so", and as this implies, the routine has more power. "SceneCrucial" is called with two arguments: SceneCrucial(S, -1) tests whether the scene is crucial or not and returns true or false; SceneCrucial(S, true) must make it true; and SceneCrucial(S, false) must make it false. Another useful difference is that if the kind of value is one which is stored in block form (e.g. for an adjective applying to text), the routine is given a pointer to the block, not a fresh copy.

A third way to define an adjective, which should be used only if speed is exceptionally important, is to provide a "schema" - a sort of I6 macro, like those provided by the C preprocessor. For example:

```
Definition: a rulebook is exciting if I6 condition
    "excitement_array-->(*1)==1" says so (it is really wild).
```

The escape "*1" is expanded to the value on which the adjective is being tested. (This is usually faster than calling a routine, but in case of side-effects, the "*1" should occur only once in the condition, just as with a C macro.) To repeat: if in doubt, use the I6 routine method above.

§27.25 Naming Unicode characters

At one time Inform allowed names to be given to Unicode character values with

sentences like so:

```
anticlockwise open circle arrow translates into Unicode as 8634.
```

These sentences now throw problem messages, and instead Inform allows exactly

those names in the Unicode standard.

§27.26 Overriding definitions in kits

When Go is clicked, Inform translates the I7 source text into a large body of so-called "Inter" code: "Inter" is short for "intermediate". Large as this program is, it could not survive on its own: it needs a large body of pre-compiled code, also written in Inter, to sustain it. This additional material is organised in blocks called "kits". Most Inform users never need to know about kits, but for example, a typical Inform project includes kits called BasicInformKit, WorldModelKit and CommandParserKit.

These kits are compiled from what is (nearly) Inform 6-syntax source code, and for the details of that, see the documentation on the low-level tool "inter". While it's absolutely possible for Inform users to create and use their own kits, that's beyond the scope of this book. But what we will cover here is the ability to include just a little extra Inter code - perhaps only a few functions or constants.

In fact, we have seen the necessary syntax already:

```
Include (- ... -).
```

puts the given material "…" into the project. For example:

```
Include (-
    [ ExtraFunction a b; return a*b; ];
-).
```

adds just a single function called "ExtraFunction".

And this works fine, but if we tried the same trick to create a function called "SquareRoot", for example, then the result would be a problem message - because BasicInformKit also defines a function of the same name. This problem message is useful, because it warns us about accidental name clashes.

But what if the name clash was not an accident at all, and what we actually wanted to give our own definition of "SquareRoot", to be used instead of the one in BasicInformKit? This is also possible:

```
Include (-
[ SquareRoot num;
    "Nobody cares about square roots, son.";
];
-) replacing "SquareRoot".
```

And now whenever square roots are calculated, this snarky text will be printed, and the result will always be rather meaningless (since this I6 routine always returns 1). Unless one is very careful, the result of replacing kit definitions can be absolute chaos.

An important historical note: between about 2010 and 2021, kits did not exist, and instead there were "template files" of Inform 6 code which served roughly then same purpose. These had names like "Relations.i6t" or "Mathematics.i6t" and were internally divided into named subsections; and Inform supported syntax like the following:

```
Include (- ... -) before "Relations.i6t".
Include (- ... -) instead of "Relations.i6t".
Include (- ... -) after "Symmetric One To One Relations" in "Relations.i6t".
```

to allow new material to be placed at oddball positions in the final code. There is now no need to worry about the placement of code - Inform's final code generator manages things so that code-ordering issues do not arise; as a result, the "before" and "after" options are now unnecessary. For now, Inform ignores these usages, and just disregards the "before…" or "after…" parts. But in some later version of Inform they will begin to cause problem messages, so writers of extensions using these syntaxes should now please remove them.

The "instead of" option now cannot work at all, and throws a problem message. The new way to substitute a fresh definition of something built-in is to use the "replacing" notation described above.

With the demise of the "template layer", as it was called, another form of so-called "template hacking" has gone with it - the special notation:

```
Include (- {-segment:MyStuff.i6t} -).
```

to allow a whole extra file of Inform 6 code called "MyStuff.i6t" to be pasted in. The new way to do that is to create a new kit, say MyStuffKit, to hold the material in question. This is not hard to do, but beyond the scope of this book. See the documentation on the low-level Inform tool "inter".

§27.27 Translating the language of play

The "language of play" is the natural language used to communicate with the player at run-time: this is normally English.

That means that it is difficult to write, say, Spanish-language IF using Inform 7, though heroic work by the Spanish IF community has overcome this. Inform 6 provided for translation by isolating its linguistic code in a part of the I6 library called the "language definition file", which was normally "English.h". Translations were gradually made to most major European languages, resulting in alternative language definition files called "French.h", "Italian.h" and so on. Full details on how to write a language definition file were given in the Translations chapter of the DM4, that is, the fourth edition of the Inform 6 Designer's Manual.

In I7 the system is different. We use the template, not a library. Instead of providing a language definition file such as "French.h", a translator should create an extension called something like "French Language by Jacques Mensonge". (The language should be named in English, so "French Language by …", not "Langue français by …") This extension should then contain broadly the same material as an I6 language definition file, but written in a mostly higher-level way. See the extension "English Language by Graham Nelson" supplied with I7, which is included automatically by default.

§27.28 Segmented substitutions

A "segmented" substitution is a syntax where text is placed between two or more different text substitutions. Examples include:

```
"This hotel is [if the player is female]just awful[otherwise]basic[end if]."
"Annie [one of]dances[or]sulks[or]hangs out at Remo's[at random]."
```

To create such syntaxes, it is not enough just to define how each expands into I6 code: for one thing we may need to know about the later terms in order to expand the earlier ones, which is normally impossible, and for another thing, the individual text substitutions mean nothing in isolation. For instance, Inform produces a problem if the following is tried:

```
"The hotel [at random] is on fire."
```

because "[at random]" is only legal when closing a "[one of] …" construction. But if "[at random]" had been defined as just another text substitution, Inform would not have been able to detect such problems.

Inform therefore allows us to mark text substitutions as being any of three special kinds:

beginning, in the middle of, or ending a segmented substitution. There can be any number of alternative forms for each of these three variants. The syntax policed is that

```
(a) Any usage must lie entirely within a single say or piece of text.
(b) It must begin with exactly one of the substitutions marked as "beginning".
(c) It can contain any number, including none, of the substitutions marked as
"continuing" (if there are any).
(d) It must end with exactly one of the substitutions marked as "ending".
```

A simple example:

```
To say emphasis on -- beginning say_emphasis_on: (- style underline; -).
To say emphasis off -- ending say_emphasis_on: (- style roman; -).
```

This creates "[emphasis on]" and "[emphasis off]" such that they can only be used as a pair. The keyword "say_emphasis_on", which must be a valid I6 identifier (and hence a single word), is never seen by the user: it is simply an ID token so that Inform can identify the construction to which these belong. (We recommend that anybody creating such constructions should choose an ID token which consists of the construction's name but with underscores in place of spaces: this means that the namespace for ID tokens will only clash if the primary definitions would have clashed in any case.)

Example

452. **Chanel Version 1**
Making paired italic and boldface tags like those used by HTML for web pages.
RB 12.1 Typography

§27.29 Invocation labels, counters and storage

The process of expanding the I6 code which represents a phrase is called "invocation". As we have seen, when a phrase is defined using a single piece of I6 code, invocation consists of copying out that I6 code, except that tokens in braces "{thus}" are replaced:

```
To say (something - number): (- print {something}; -).
```

Ordinarily the only token names allowed are those matching up with names in the prototype, as here, but we have already seen one special syntax: "{phrase options}", which expands as a bitmap of the options chosen. And in fact the invocation language is larger still, as a skim through the Standard Rules will show. The notes below deliberately cover only some of its features: those which are likely to remain part of the permanent design of Inform, and which are adaptable to many uses. **Please do not use any of the**

undocumented invocation syntaxes: they change frequently, without notice or even mention in the change log.

The first special syntaxes are textual tricks. {-delete} deletes the most recent character in the I6 expansion of the phrase so far. {-erase} erases the I6 expansion of the phrase so far. {-open-brace} and {-close-brace} produce literal "{" and "}" characters.

The following:

```
{-counter:NAME}
{-counter-up:NAME}
{-zero-counter:NAME}
{-counter-makes-array:NAME}
```

create (if one does not already exist) a counter called NAME. This is initially zero, and can be reset back to zero using "{-zero-counter:NAME}", which expands into no text. The token "{-counter:NAME}" expands into the current value of the counter, as a literal decimal number. The token "{-counter-up:NAME}" does the same, but then also increases it by one. Finally, the token "{-counter-makes-array:NAME}" expands to nothing, but tells Inform to create an "-->" array called "I7_ST_NAME" which includes entries from 0 up to the final value of the NAME counter.

This allows each instance in the source text of a given phrase to have both (i) a unique ID number for that invocation, and (ii) its own word of run-time storage, which can allow it to have a state preserved in between times when it is executed. For example:

```
To say once only -- beginning say_once_only:
	(- {-counter-makes-array:say_once_only}if (I7_ST_say_once_only-->{-
counter:say_once_only} == false) {-open-brace} I7_ST_say_once_only-->{-counter-
up:say_once_only} = true; -).
To say end once only -- ending say_once_only:
	(- {-close-brace} -).
```

To complete the tools available for defining a segmented substitution, we need a way for the definition of the head to know about the middle segments and the tail:

When invoking either the head or the tail, {-segment-count} expands to the literal decimal number of pieces of text in between the two, which is always one more than the number of middle segments, since the text comes in between the segments. When invoking any middle segment, {-segment-count} expands to the number of pieces of text so far -- thus it expands to 1 on the first middle segment invoked, 2 on the next, and so on.

Lastly {-final-segment-marker} expands to the I6 identifier which marks the end segment,

or to I6_NULL if the end segment has no marker. The idea of markers is to enable the head's definition to know which of a number of choices has been used for the tail, supposing that this is a construction with a variety of legal endings. For example:

```
To say emphasise -- beginning say_emphasise:
    (- style {-final-segment-marker}; -).
To say with italics -- ending say_emphasise with marker underline:
    (- style roman; -).
To say with fixed space type -- ending say_emphasise with marker fixed:
    (- style roman; -).
```

The markers used for the tails here are "underline" and "fixed", and when the head is invoked, the marker for its tail is expanded into the argument of I6's "style" statement.

The examples above are all to do with segmented substitutions, which is where they are most useful, but most of the syntaxes above work equally well for ordinary "To…" phrase definitions.

§27.30 To say one of

Many of the invocation syntaxes described in the previous section are used in the definition by the Standard Rules of the "[one of] … [or] … [purely at random]" construction, so it makes a good example of how they can be used.

First, this is a segmented substitution with a single possible beginning ("[one of]"), a single possible middle ("[or]") but a choice of many possible endings. Almost everything is compiled by the invocation of the beginning:

```
To say one of -- beginning say_one_of (documented at phs_oneof): (-
	{-counter-makes-array:say_one_of}
	{-counter-makes-array:say_one_flag}
	if (I7_ST_say_one_flag-->{-counter:say_one_flag} == false) {
		I7_ST_say_one_of-->{-counter:say_one_of} = {-final-segment-
marker}(I7_ST_say_one_of-->{-counter:say_one_of},
{-segment-count});
		I7_ST_say_one_flag-->{-counter:say_one_flag} = true;
	}
	if (say__comp == false) I7_ST_say_one_flag-->{-counter:say_one_flag}{-counter-
up:say_one_flag} =
false;
	switch ((I7_ST_say_one_of-->{-counter:say_one_of}{-counter-up:say_one_of})%({-
segment-count}+1)-1)
{-open-brace}
		0: -).
To say or -- continuing say_one_of (documented at phs_or):
	(- @nop; {-segment-count}: -).
To say purely at random -- ending say_one_of with marker I7_SOO_PAR (documented at
phs_purelyrandom):
	(- {-close-brace} -).
```

The 3rd invocation of this (say) might compile the following:

```
I7_ST_say_one_of-->2 = I7_SOO_PAR(I7_ST_say_one_of-->2, 4);
switch((I7_ST_say_one_of-->2)%5 - 1) {
	0: ... first text ...
	1: ... second text ...
	2: ... third text ...
	3: ... fourth text ...
}
```

First, we notified Inform that it needs to allocate an array (I7_ST_say_one_of) providing storage associated with the counter "say_one_of". This we used to count off individual invocations of "[one of]", so that each would have its own word of storage - for the 3rd invocation, I7_ST_say_one_of-->2. We then call a state-changing routine, in this case I7_SOO_PAR, which is allowed to know the previous state and also the number of options available, and which returns the new state. The state is supposed to be the option chosen last time, but that means that there are not 4, but 5 possibilities: 0 for "there was no last time", then 1 to 4 for the possible outcomes. We reduce the state mod 5 to obtain the decision this time, and subtract 1 because it happens to be convenient to make the switch statement run from 0 to 3 rather than 1 to 4. (The reason we reduce the state mod 5 is to allow the state-changer to squirrel away secret information in the upper bits of the

state, if it wants to. Note that subtracting one means that the switch value might be -1, which results in no text being printed: thus if the state-changer chooses 0, it can decide on none of the above.)

In this design, the marker attached to the choice of ending substitution is the name of the I6 state-changer: here is the I7_SOO_PAR routine.

```
[ I7_SOO_PAR oldval count; if (count <= 1) return count; return random(count); ];
```

As it happens, this ignores the old value: after all, it is meant to be purely at random, and nothing could be less pure than taking the last outcome into consideration when choosing the next.

Note that the counter say_one_of is advanced in invocation of the head. It might seem that the tidier design, somehow, would be to advance the counter in the invocation of the tails, but this is not a good idea. In general it is not safe to assume that the counter will have the same value when the tail is invoked that it had when the head was invoked, because segmented say constructions can legally be nested in Inform strings. Because of this, it is best to deal with a counter entirely in a single invocation, either of the beginning or the ending.

Because "[one of] … [or] …" is such a useful construction - switching between alternative forms of text, which writers of IF very often do - the above implementation is intentionally left open for new endings to be added, and the examples below show how easily this can be done.

Examples

453. **Blink**
Making a "by atmosphere" token, allowing us to design our own text variations such as "[one of]normal[or]gloomy[or]scary[by atmosphere]".
RB 2.1 Varying What Is Written

454. **Uncommon Ground**
Making a "by viewpoint" token, allowing us to design our own text variations such as "[show to yourself]quaint[to Lolita]thrilling[to everyone else]squalid[end show]" depending on the identity of the player at the moment.
RB 5.6 Viewpoint