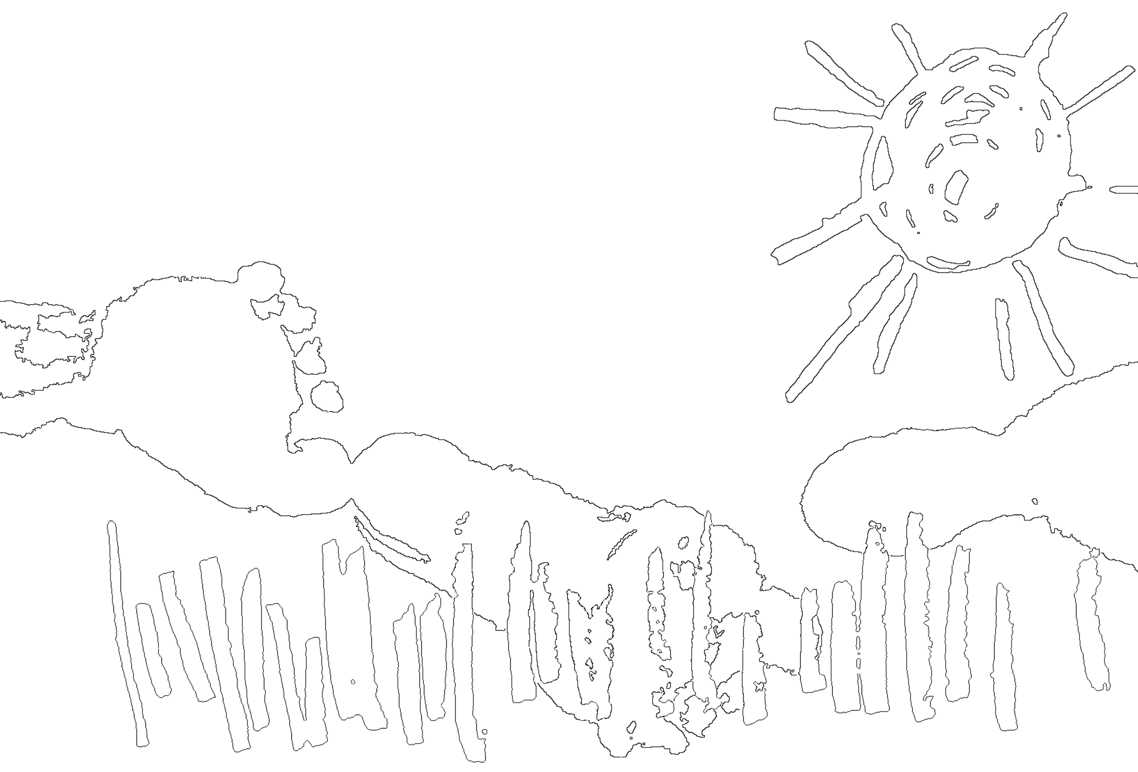
The Inform Beginner's Guide

Roger Firth and Sonja Kesserich



Second Edition: August 2002

With a Foreword by Graham Nelson

The Inform Beginner's Guide

Authors: Roger Firth and Sonja Kesserich
Editor: Dennis G. Jerz
Printed edition managed by: David Cornelson

Cover: *First Steps* (watercolour and crayon on paper, 2002) Harry Firth (2000–)

This book and its associated example games are copyright © Roger Firth and Sonja Kesserich 2002. Their electronic forms may be freely distributed provided that: (a) distributed copies are not substantially different from those archived by the authors, (b) this and other copyright messages are always retained in full, and (c) no profit is involved. Exceptions to these conditions must be negotiated directly with the authors (`roger@firthworks.com` and `polilla@idecnet.com`).

The authors assume no liability for errors and omissions in this book, or for damages or loss of revenue resulting from the use of the information contained herein, or the use of any of the software described herein.

Inform, the program and its source code, its example games and documentation, are copyright © Graham Nelson 1993–2002.

First Edition: April 2002
Second Edition (with minor revisions): August 2002

To be published by The Interactive Fiction Library (IFLibrary.com)
PO Box 3304, St Charles, Illinois 60174
Printed in the United States of America
ISBN 0-9713119-2-7

Contents

Foreword by Graham Nelson 7

About this guide 9
- Scope and approach 10
- Presentation and style 10
- Useful Internet resources 11
- Acknowledgements 12

1 • Just what is interactive fiction? 13

2 • Tools of the trade 17
- Inform on an IBM PC 19
- Inform on an Apple Macintosh 23
- More about the editor 24
- More about the compiler 24
- More about the interpreter 25

3 • Heidi: our first Inform game 27
- Creating a basic source file 27
- Understanding the source file 29
- Defining the game's locations 30
- Joining up the rooms 32
- Adding the bird and the nest 35
- Adding the tree and the branch 37
- Finishing touches 39

4 • Reviewing the basics 41
- Constants and variables 41
- Object definitions 42
- Object relationships – the object tree 44
- Things in quotes 47
- Routines and statements 48

5 • Heidi revisited 51
- Listening to the bird 51
- Entering the cottage 53
- Climbing the tree 55
- Dropping objects from the tree 56
- Is the bird in the nest? 58
- Summing up 58

6 • William Tell: a tale is born 61
- Initial setup 61
- Object classes 64

7 • William Tell: the early years 69
Defining the street 69
Adding some props 71
The player's possessions 73
Moving further along the street 75
Introducing Helga 76

8 • William Tell: in his prime 81
The south side of the square 81
The middle of the square 82
The north side of the square 89

9 • William Tell: the end is nigh 91
The marketplace 91
Gessler the governor 95
Walter and the apple 96
Verbs, verbs, verbs 98

10 • Captain Fate: take 1 105
Fade up on: a nondescript city street 105
A hero is not an ordinary person 112

11 • Captain Fate: take 2 115
A homely atmosphere 115
A door to adore 119

12 • Captain Fate: take 3 127
Too many toilets 127
Don't shoot! I'm only the barman 130

13 • Captain Fate: the final cut 137
Additional catering garnish 137
Toilet or dressing room? 138
And there was light 141
Amazing technicolour dreamcoats 146
It's a wrap 148

14 • Some last lousy points 151
Expressions 151
Internal IDs 152
Statements 152
Directives 154
Objects 155
Routines 157
Reading other people's code 159

15 • Compiling your game 167
Ingredients 167
Compiling *à la carte* 170
Switches 171

16 • Debugging your game 173
Command lists 174
Spill them guts 174
What on earth is going on? 175
Super-powers 177
Infix: the harlot's prerogative 178
No matter what 179

17 • * You have won ***** 181

Appendix A • How to play an IF game 185

Appendix B • "Heidi" story 189
Transcript of play 189
Game source code – original version 190
Game source code – revisited 192

Appendix C • "William Tell" story 195
Transcript of play 195
Game source code 198
Compile-as-you-go 208

Appendix D • "Captain Fate" story 211
Transcript of play 211
Game source code 214
Compile-as-you-go 227

Appendix E • Inform language 229
Literals 229
Names 229
Constants 229
Variables and arrays 229
Expressions and operators 230
Classes and objects 230
Manipulating the object tree 231
Message passing 231
Uncommon and deprecated statements 231
Statements 231
Routines 231
Flow control 232
Loop control 232
Displaying information 232
Verbs and actions 233
Other useful directives 233
Uncommon and deprecated directives 233

Appendix F • Inform library 235
Library objects 235
Library constants 235
User-defined constants 235
Library variables 236
Library routines 236
Object properties 238
Object attributes 241
Optional entry points 242
Group 1 actions 242
Group 2 actions 243
Group 3 actions 243
Fake actions 244

Appendix G • Glossary 245

Index 251

Foreword by Graham Nelson

t would, I think, be immodest to compare myself to Charles Bourbaki (1816–97), French hero of the Crimean War and renowned strategist, a man offered nothing less as a reward than the throne of Greece (he declined). It may be in order, though, to say a few words about his fictitious relative Nicholas, the most dogged, lugubrious, interminably thorough and clotted writer of textbooks ever to state a theorem. Rather the way Hollywood credits movies for which nobody wants the blame to the director "Alan Smithee" (who by now has quite a solid filmography and even gets the occasional cinema festival), so in mathematics many small results are claimed to be the work of Nicholas Bourbaki. Various stories are told of the birth of Bourbaki, under whose name young Parisian mathematicians have clubbed together since 1935 to write surveys of whole fields of algebra. His initials, it may be noted, are NB. Some say "Bourbaki" was an in-joke at the Ecole Normale Supérieure (much as "zork" and "foobar" were at MIT), going right back to a practical joke in 1880 when a pupil successfully impersonated a visiting "General Claude Bourbaki". Folklore also has it that the real general was notorious when on manoeuvres for being able to eat *anything* if need be – stale biscuit, raw turnips, his horse, his horse's hay, his horse's leather nosebag that the hay used to be in – just as Nicholas Bourbaki would have to eat everything there was to eat in the theory of algebra, no matter how tooth-grinding or chewy. To give credit where it's due, Bourbaki's forty volumes are quite useful. Or, actually, they aren't, but it's nice to know they're there.

It was on reading this present book that I realised the melancholy truth: that my own volume on Inform, the *Designer's Manual*, is a Bourbaki. It has to cover every last thing, from Icelandic accents to assembly language to fake actions, not to mention fake fake actions, to grouping together almost-but-not-quite-identical objects such as Scrabble tiles – matters which a dedicated Inform designer might need to look up once in a lifetime, or then again might not. To be sure, the basics do turn up every so often, especially in Chapters II and III, but despite my best intention it is a gentle introduction only if you pick your way through as if on stepping stones. This book, on the other hand, is a follow-as-you-go tutorial, covering the basics thoroughly and a little at a time. Where the *Designer's Manual* tries never to retrace its steps, so for instance there is just one section on locations, the *Beginner's Guide* works its way through three whole games, giving it three chances to visit every subject, reinforcing and showing a little more each time.

I should like to say that my first reaction, when out of the blue the authors sent me advance proofs, was to exclaim with delight at the lucid, uncluttered, sensible approach. Truthfully, however, that was my third reaction, the first being jealousy (it's all right for *you*, you don't have to document how the parser calculates GNA sets for noun clauses) and the second pique (you've cast the *gizbru spell: turn dangerous object into a harmless one* at my book). When it comes

down to it, though, there is no greater compliment any writer can be paid than to have someone else choose to write a book about his work, so I thank Roger and Sonja for their gesture, as well as the fine job they have done.

That is quite enough talking about myself, as Inform belongs to all its users, to the hundreds of serious writers of interactive fiction who find it helpful, and for almost a decade it has been a collective enterprise. Today nobody remembers who suggested what. Its world-modelling rules now resemble a New England patchwork quilt, to which each house in the village contributes one woven square. As you read this book, you might want to bear in mind that such a quilt is never finished, and always has room for one more square from a newly arrived neighbour.

St Anne's College
University of Oxford
April 2002

About this guide

If they asked me, I could write a book;
About the way you TALK, and LISTEN; And LOOK.
– with apologies to Richard Rodgers and Lorenz Hart.

ext adventures, otherwise known collectively as interactive fiction (IF), were highly popular computer games during the 1980s. As technology evolved they faded from the market, unable to compete with increasingly sophisticated graphical games; however, IF was far from dead. The Internet grew, and Usenet discussion forums offered a focal point for fans of the genre. By developing IF programming tools and systems, organising contests and writing tutorials and reviews, these enthusiasts have led a revival responsible for many notable works, including some whose quality arguably surpasses that of the best commercial titles of the 1980s.

Almost everything that you need to begin writing your own text adventures is available, for free, on the Internet. Nowadays it's a hobby, not a business, with a pretty small audience – probably only a few thousand people worldwide are avid consumers of contemporary IF. So, expect fun and satisfaction – but no profit.

While expert programmers may relish the considerable challenge of creating text adventures using a generalised language such as BASIC or C, specialist IF tools have largely solved the fundamental world-building issues. The most common systems are Graham Nelson's Inform – our subject matter – and Mike Roberts' TADS (Text Adventure Development System). New hopefuls arrive each year, but few achieve widespread acceptance; the majority of today's IF (and virtually all the works generally regarded as well written, mature, interesting, innovative, sophisticated, etc.) have been created with either one or the other. In our view, only TADS bears comparison with Inform in popularity, in being able to handle simple and complex stories, and in availability on PCs, Macs, hand-held devices and a wide variety of other computers. But, since you're reading our guide, we'll assume that you've already made a choice, and decided to give Inform a try.

We aim to provide a grounding in Inform basics. When you have learnt a little about it, you'll be able to design simple games for your friends to play and, as you become more accomplished, which you can share via the Internet with enthusiasts worldwide. However, if you simply want to play[1] games written by others – rather than write them yourself – then you don't need to learn Inform, and this guide isn't for you.

1. If you feel confused about IF in general or about this distinction between writing and playing in particular, try glancing ahead at "Just what is interactive fiction?" on page 13 and at "How to play an IF game" on page 185; also, try the Ifaq at http://www.plover.net/~textfire/raiffaq/ifaq/.

Scope and approach

Because this is only an introduction to Inform, many features are treated rather superficially, or ignored altogether. The definitive text is Graham Nelson's *Inform Designer's Manual* (Fourth Edition, July 2001), commonly known as the DM4; you cannot hope to use Inform successfully without having this splendid book by your side. Our guide should be seen merely as a supplement to the DM4, offering step-by-step descriptions of those aspects of Inform which are most important on first acquaintance. In any matter where we seem at odds with what Graham has written, you should assume that he is right and that we are, well, confused.

As a tutorial, this guide is intended to be printed out and then read sequentially; it isn't meant for online usage or designed as a reference manual, though it does provide brief summaries of Inform's language and library. Our approach is to teach you about Inform through the creation of three games: all short, all playable to completion. "Heidi" is just about as simple as an IF game can be, but still manages to introduce a range of important concepts. "William Tell", a retelling of the famous folk tale, is nearly as brief but roams more widely in its use of Inform's capabilities. Finally "Captain Fate" presents a comic-book hero in urgent need of a change. By the end of the guide, we'll have touched on less than half of Inform's capabilities, but we hope we'll have mentioned most of the things that matter when you're starting out to design your first Inform game.

One final point: Inform is a powerful system, often offering several different ways of tackling a particular design requirement. We've tried to present things as simply and consistently as possible, but you shouldn't be surprised to discover other approaches, maybe shorter, maybe more efficient, than those shown here.

Presentation and style

Most of the guide's text appears in this typeface, except where we're using words which are part of the Inform system (like `print`, `Include`, `VerbLib`) or are extracted from one of our games (like `bird`, `nest`, `top_of_tree`). Terms in **bold type** are included in the glossary – Appendix G on page 245. We switch to italic type for a placeholder: for example you should read the Inform statement:

```
print "string";
```

as meaning "display on the player's screen the arbitrary character or characters which are represented here by the placeholder *string*". Examples might include:

```
print "Hello world!";
print "Fourscore and seven years ago our fathers brought forth on this continent
    a new nation, [...] and that government of the people, by the people,
    for the people shall not perish from the earth.";
```

We place the "TYPE" symbol alongside game fragments which you can type in as a part of our working examples. This differentiates them from other code snippets whose only purpose is to illustrate some particular feature.

Useful Internet resources

One of our basic assumptions – along with your burning desire to learn Inform and your ability to work comfortably with the files and folders on your computer – is that you have access to the Internet. This is pretty well essential, since almost everything you need is available only via this medium. In particular, you'll find much helpful material at these locations:

- `http://www.inform-fiction.org/`

  The Inform home page, maintained by Graham Nelson and a small team of helpers. Most important, this is where you can find the *Inform Designer's Manual* in PDF format.

- `http://www.ifarchive.org/`

  The IF Archive, from which you can download almost anything that's free and in the public domain. For a clickable map of Inform-related parts of the Archive, see `http://www.firthworks.com/roger/informfaq/hh.html`.

  NOTE: prior to August 2001, the IF Archive was located elsewhere, at `ftp://ftp.gmd.de/if-archive/`, and references to that location can still be found. *Do not use* the old location: any information still available from there is likely to be out-of-date.

- `http://www.iflibrary.com/`

  Paperback copies of the *Inform Designer's Manual* and this *Inform Beginner's Guide* will at some point be available from David Cornelson's IF Library site.

- `http://www.firthworks.com/roger/`

  Roger Firth's Inform pages, including the Informary (what's new in Inform?), and the Inform Frequently Asked Questions (FAQ) pages.

- `http://www.plover.net/~textfire/raiffaq/`

  A more general list of FAQs about IF authorship, covering both Inform and the other main systems.

- `news:rec.arts.int-fiction`

  The Usenet newsgroup for authors of IF, commonly known by the abbreviation RAIF. Here you'll find discussion on IF technology, criticism and game design issues, and fast, friendly and knowledgeable assistance with your own "how do I…" questions (but please, look in the manual first).

- `news:rec.games.int-fiction`

  The complementary newsgroup for IF *players*, often known as RGIF.

Acknowledgements

Becoming sufficiently conversant with Inform to be able to share it with others is not something done quickly or in isolation. In getting to where we are today, we have been assisted at many times and in many ways by the notably supportive and good-natured people, far too numerous to list by name, who make `rec.arts.int-fiction` such an invaluable IF resource. We are grateful to you all.

In writing this guide, we have received specific help from a number of people (some not even related to us): Harry Firth and Jo Quinn created the cover artwork, while Barney Firth, Megan Firth and Phil Graham assisted us with PC and Macintosh environments. Graham Nelson kindly wrote the Foreword, and delighted us with long and detailed lists of helpful comments and suggestions on two of our drafts; we also greatly appreciate the views of other early readers, including Christine Firth, Jim Fisher, Muffy St. Bernard, Gunther Schmidl, Emily Short and A. Sloe. Inform novice Paul Johnson tested its validity as a tutorial, and painstakingly reassured us that it actually worked. Dennis G. Jerz patiently and skilfully edited the text, making innumerable improvements to our often wayward and inconsistent prose. Further invaluable feedback on the first (beta) edition came from Rosemary Frezza (bug-hunter extraordinaire), Curt Siffert, Pavel Soukenik, Elise Stone and Brent VanFossen. With his usual flair, David Cornelson is supervising the guide's eventual transformation into professionally printed respectability. Thank you: it is impossible to overestimate the value of this freely given support and assistance.

The drop capitals, and their associated poem, are from "A Picture Alphabet", digitised from a collection of public domain woodcuts, circa 1834, by Steven J. Lundeen of emerald city fontwerks.

All credit to the generosity of `http://briefcase.yahoo.com/` for making international file-sharing such a breeze.

Finally, of course, we owe an enormous debt of gratitude to Graham Nelson for devising it all, thereby giving us the opportunity – first independently and later in enjoyable collaboration – of using, and eventually of presenting, the Inform text adventure development system.

Roger Firth
Reading, England

Sonja Kesserich
Madrid, Spain

August 2002

1 • Just what is interactive fiction?

A was an archer, who shot at a frog;
B was a butcher, who had a great dog.

efore we start learning to use the Inform system, it's probably sensible to consider briefly how IF, which has many narrative elements, differs from regular storytelling. Before we do *that*, though, let's look at an example of a familiar folk tale.

"There was once a man called Wilhelm Tell, from high in the Swiss Alps near the town of Altdorf. A hunter and a guide, a proud mountaineer, he lived by his skills in tracking and archery. It happened one day that Wilhelm visited the town to buy provisions, and he took his son Walter with him.

The region was at the time governed by Hermann Gessler (a vain and petty man appointed as vogt by the Austrian emperor), who attempted a show of power over his subjects by placing his hat on a pole in the town square, for everyone to salute. Reluctant citizens were "encouraged" by a troop of the vogt's soldiers, who made sure that their bows were sufficiently respectful.

Wilhelm knew of the hat, and of the humiliating exercise in obeisance. So far he had managed to avoid the town's square, sure that – given his open dislike for the vogt – his refusal to bend the knee would cause trouble. Today, however, he needed to pass near the pole to reach Johansson's tannery.

If Wilhelm had hoped for a lucky break, we'll never know. The square was filled with market-day crowds; the soldiers were especially keen in their salute-enforcing duties, challenging everyone with loud shouts and the occasional coarse expletive. Wilhelm threw a protective arm over his son's shoulder and walked determinedly without looking at the pole or the guards.

A soldier called to him; Wilhelm took no notice. Other guards focused their attention on the archer. "Salute the vogt's hat," he was told. A tense silence followed. Wilhelm tried to keep going, but by now he was surrounded. The men knew of him; one counselled Wilhelm to give a cursory nod towards the hat and be done. Everybody in the vicinity was watching, so the disrespect could not be ignored. There was a long pause. Wilhelm refused.

Word was sent to Gessler, who rushed to the square with reinforcements. The little man was delighted at the chance of making an example of the trouble-maker. He mockingly recounted the many skills of Herr Tell, speculating that such mastery maybe accounted for the pride that prevented acknowledgement of the Emperor's authority. The vogt understood all that, and would give him a fair chance. If Wilhelm were able to shoot true at an apple from fifty paces, Gessler would be inclined to show mercy; however, to make things interesting, the apple was to be balanced on Walter's head.

> Everything was set up. Wilhelm selected and mounted an arrow and slowly raised the bow, conscious of the motionless and brave stance that Walter was displaying. He pulled, feeling the tension mount on the string and his fingers, through his hands, into his arms. He'd made much more difficult shots in the past, at fleeting deer, at soaring birds; but this was the life of his son... he could not fail, would not fail.
>
> Wilhelm let loose. The arrow flew straight and true, violently pinning the apple to the tree behind the boy. The crowd exploded in a roar of relief and admiration, and Gessler, disappointed, had no option but to let them go.
>
> Years later, Wilhelm led an uprising against the vogt... but that's another story."

And now an extract from the same tale, this time in the form of a tiny text adventure game. If you're new to interaction with text adventures you'll find some general instructions in "How to play an IF game" on page 185, and you can see a complete transcript of the game in the "William Tell" story on page 195:

```
A street in Altdorf
The narrow street runs north towards the town square. Local folk are pouring
into the town through the gate to the south, shouting greetings, offering
produce for sale, exchanging news, enquiring with exaggerated disbelief about
the prices of the goods displayed by merchants whose stalls make progress even
more difficult.

"Stay close to me, son," you say, "or you'll get lost among all these people."

>GO NORTH

Further along the street
People are still pushing and shoving their way from the southern gate towards
the town square, just a little further north. You recognise the owner of a fruit
and vegetable stall.

Helga pauses from sorting potatoes to give you a cheery wave.

"Hello, Wilhelm, it's a fine day for trade! Is this young Walter? My, how he's
grown. Here's an apple for him -- tell him to mind that scabby part, but the
rest's good enough. How's Frau Tell? Give her my best wishes."

>INVENTORY
You are carrying:
  an apple
  a quiver (being worn)
    three arrows
  a bow

>TALK TO HELGA
You warmly thank Helga for the apple.

>GIVE THE APPLE TO WALTER
"Thank you, Papa."
```

```
>NORTH

South side of the square
The narrow street to the south has opened onto the town square, and resumes at
the far side of this cobbled meeting place. To continue along the street towards
your destination -- Johansson's tannery -- you must walk north across the
square, in the middle of which you see Gessler's hat set on that loathsome pole.
If you go on, there's no way you can avoid passing it. Imperial soldiers jostle
rudely through the throng, pushing, kicking and swearing loudly.
...
```

Some of the more obvious differences are highlighted by these questions:

- **Who is the protagonist?**

  Our example of narrative prose is written in the third person; it refers to the hero as “Wilhelm” and “he” and “him”, watching and reporting on his activities from afar. In this sample IF game, *you* are the hero, seeing everything through Wilhelm’s eyes.

- **What happens next?**

  The regular narrative is intended to be read once, straight through from beginning to end. Unless you didn’t pay attention the first time, or you’re planning to critique the story, there’s generally no need to go back and read a sentence twice; if you do, you’ll find exactly the same text. The author leads the way and sets the pace; you, as the reader, just go along for the ride.

  In IF, that’s usually much less true. The author has created a landscape and populated it with characters, but you choose how and when to explore it. The game evolves, at least superficially, under your control; perhaps you explore the street first and then the square, perhaps the other way round. There usually are multiple paths to be found and followed – and you can be pretty certain that you won’t discover them all, at least on first acquaintance.

- **How does it all turn out?**

  You can tell when you’ve come to the end of a regular narrative – you read the last sentence, and you know there’s no more. In IF, it’s clear enough when you reach *an* end; what’s much less apparent is whether that’s the only conclusion. In the transcript from the example game, you win by shooting the apple from Walter’s head. But what if you miss? What if you hit him by mistake? Or fire instead at the hated vogt? Or even stand the tale on its head by bowing obsequiously to the governor’s hat and then going about your business? All of these are possible ways in which the game could come to an end. The phrase “what if” is the key to writing successfully, and should always be in the forefront of an IF designer’s mind.

- **Where did Helga come from?**

  You’ll notice that Helga and her stall don’t appear in the regular narrative; she’s a distraction from the tale’s momentum. But in the IF game, she fulfils

a number of useful functions: mentioning the names “Wilhelm”, “Walter” and “Frau Tell” (so that you know who the tale’s about), introducing the all-important apple in a natural manner and, above all, providing an opportunity for the “I” in IF – some interactivity. Without that – the chance to interact with the tale’s environment – the game is little different from a conventional piece of fiction.

- **That item looks interesting; can you tell me more about it?**

  In the regular narrative, what you see is what you get; if you want to know more about alpine life in the fourteenth century, you’ll need to consult another source. IF, on the other hand, offers at least the possibility of delving deeper, of investigating in greater detail an item which has been casually mentioned. For example, you could have explored Helga’s stall:

  ```
  " ... How's Frau Tell? Give her my best wishes."

  >EXAMINE THE STALL
  It's really only a small table, with a big heap of potatoes, some carrots
  and turnips, and a few apples.

  >EXAMINE THE CARROTS
  Fine locally grown produce.
  ```

  You see those descriptions only if you seek them; nothing you find there is unexpected, and if you don’t examine the stall, you’ve not missed anything important. Nevertheless, you’ve enhanced the illusion that you’re visiting a real place. Such details would rapidly grow tedious if the stall and its contents were described in full each time that you pass them.

- **How do I work this thing?**

  Whereas the presence of Helga is an elaboration of the folk tale, the shooting of the arrow (it’s in the transcript in “William Tell” story on page 195, not in the extract above) illustrates the opposite principle: simplification. The tale builds dramatic tension by describing each step as Wilhelm prepares to shoot the apple. That’s OK; he’s been an archer all his life, and knows how to do it. You, on the other hand, probably know little about archery, and shouldn’t be expected to guess at the process and vocabulary. Let’s hope you know that you need to shoot at the apple – and that’s all it takes. The game explains what was involved, but doesn’t force you through each mundane step.

Of course, all of these are generalisations, not universal truths; you could find fine works of IF which contradict each observation. However, for our purposes as beginners in the craft of IF design, they represent useful distinctions between IF and conventional fiction.

We’ll come back to the “William Tell” tale in a later chapter, but before then we’ll work through an even simpler example. And before either of those, we need to download the necessary files which will enable us to write Inform games.

2 • Tools of the trade

C was a captain, all covered with lace;
D was a drunkard, and had a red face.

onventional – static – fiction can be written using nothing more than pencil and paper, or typewriter, or word-processor; however, the requirements for producing IF are a little more extensive, and the creative process slightly more complex.

- For static fiction, you first write the text, and then you check it by reading what you've written.
- For IF, you still have to write all of the text, but you also have to establish what text gets displayed when. Once you have written the necessary Inform instructions, you use a **compiler** program to convert them into a playable format. The resulting information is played by an **interpreter** program, which permits you to interact with your developing world.

With static fiction What You Write Is What You Read, but with IF the format in which you initially write the game doesn't bear much resemblance to the text which the interpreter ultimately displays. For example, the "William Tell" game, in the form that we wrote it, starts like this:

```
!============================================================================
Constant Story "William Tell";
Constant Headline
            "^A simple Inform example
             ^by Roger Firth and Sonja Kesserich.^";

Include "Parser";
Include "VerbLib";

!============================================================================
! Object classes

Class   Room
  has   light;
...
```

You will never need to look at it in the form produced by the compiler:

```
050000012C6C2C2D1EF6010A0C4416900010303230313031004253FEA90C0000
000000000000000000000000000168F0000000000000102000000000362E3231
...
```

but, as you'll notice from the full transcript in "William Tell" story on page 195, the player will see the following:

```
The place: Altdorf, in the Swiss canton of Uri. The year is 1307, at which time
Switzerland is under rule by the Emperor Albert of Habsburg. His local governor
-- the vogt -- is the bullying Hermann Gessler, who has placed his hat atop a
wooden pole in the centre of the town square; everybody who passes through the
square must bow to this hated symbol of imperial might.
...
```

Clearly, there's more to writing IF than just laying down the words in the right order. Fortunately, we can make one immediate simplification: the translated form produced by the Inform compiler – those cryptic numbers and letters held in what's known as the **story file** – is designed to be read by the interpreter program. The story file is an example of a "binary" file, containing data intended for use only by a computer program. Forget all that unreadable gibberish.

So that leaves just the first form – the one starting "`Constant Story`" – which represents the tale written as a piece of IF. That's the **source file** (so called because it contains the game in its original, source, form) which you create on your computer. The source file is a "text" (or "ASCII") file containing words and phrases which can be read – admittedly after a little tuition, which is what this guide is all about – by humans.

How do you create that source file? Using a third software program: an **editor**. However, unlike the compiler and interpreter, this program isn't dedicated to the Inform system – or even to IF. An editor is an entirely general tool for creating and modifying text files; you've probably already got a basic one on your computer (an IBM PC running Windows comes with NotePad, while an Apple Macintosh has SimpleText or TextEdit), or you can download a better one from the Internet. An editor is like a word-processing program such as MS Word, only much less complex; no fancy formatting features, no bold or italics or font control, no embedded graphics; it simply enables you to type lines of text, which is exactly what's needed to create an IF game.

If you look at the game source on the previous page, or in the "William Tell" story on page 195, you'll notice `Include "Parser";` and `Include "VerbLib";` a few lines down from the top of the file. These are instructions to the Inform compiler to "include" – that is, to merge in the contents – of files called `Parser.h` and `VerbLib.h`. These are not files which you have to create; they're standard **library files**, part of the Inform system. All that you have to do is remember to `Include` them in every game that you write. Until you've a fair understanding of how Inform works, you've no need to worry about what they contain (though you can look if you want to: they're readable text files, just like the ones this guide will teach you to write).

So, we've now introduced all of the bits and pieces which you need in order to write an Inform adventure game:

- a text **editor** program which can create and modify the **source file** containing the descriptions and definitions of your game. Although it's not

recommended, you can even use a word-processing program to do this, but you have to remember to save your game in Text File format;

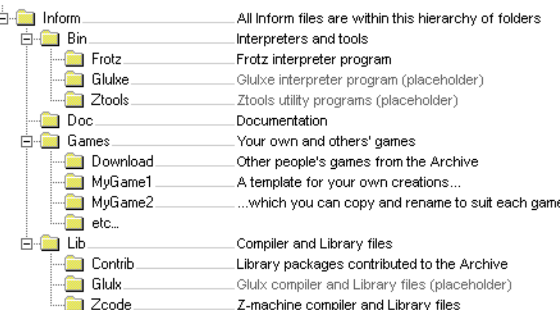
- some Inform **library files** which you `Include` in your own game source file in order to provide the **model world** – a basic game environment and lots of useful standard definitions;
- the Inform **compiler** program, which reads your source file (and the library files) and translates your descriptions and definitions into another format – the **story file** – intended only for...
- an Inform **interpreter** program, which is what players of your game use. A player doesn't require the source file, library files or compiler program, just the interpreter and the game in compiled format (which, because it's a binary file not meaningful to human eyes, neatly discourages players from cheating).

All of those, apart from the editor, can be downloaded for free from the IF Archive. One approach is to fetch them individually, following the guidance on Graham's page: visit `http://www.inform-fiction.org/` and look for the "Software" section. However, if you're using a PC or a Mac, you'll find it easier to download a complete package containing everything that you need to get started.

Inform on an IBM PC

Follow these steps:

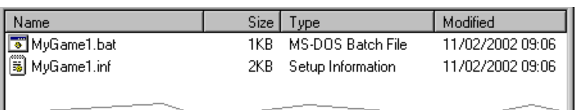
1. Download `http://www.firthworks.com/roger/downloads/inform_pc_env.zip` to a temporary location on your PC.
2. Use a tool like WinZip to unzip the downloaded file, giving you a new `Inform` folder. Move this folder (and its contents) to a suitable location on your PC – a good place would be `C:\My Documents\Inform`, but you could also use `C:\Inform` or `C:\Program Files\Inform`. You should now have this set of folders:



In order to make the download small and fast, these folders include just enough to get you started as an Inform designer – the compiler and

interpreter programs, the library files, the `Ruins.inf` example file from the *Inform Designer's Manual*, and a template for your own first game. A few other folders are included as placeholders where you could later download additional components, if you wanted them. As soon as possible, you should download the *Inform Designer's Manual* into the `Inform\Doc` folder – it's an essential document to have, and has been omitted from this download only because of its 3MB size.

3. To verify that the downloaded files work properly, use Windows Explorer to display the contents of the `Inform\Games\MyGame1` folder: you will see the two files `MyGame1.bat` and `MyGame1.inf`:



Name	Size	Type	Modified
MyGame1.bat	1KB	MS-DOS Batch File	11/02/2002 09:06
MyGame1.inf	2KB	Setup Information	11/02/2002 09:06

`MyGame1.inf` is a tiny skeleton game in Inform source format. By convention, all Inform source files have an extension of `.inf`; Windows has an inbuilt definition for `.inf` files, and so shows its Type as "Setup Information", but this doesn't seem to matter. If you double-click the file, it should open in NotePad so that you can see how it's written, though it probably won't mean much – yet.

4. `MyGame1.bat` is an MS-DOS batch file (an old kind of text-only computer program, from the days before point-and-click interfaces) which runs the Inform compiler. Double-click it; a DOS window opens as the game compiles, and you'll see this:

```
C:\My Documents\Inform\Games\MyGame1>..\..\Lib\Zcode\Infrmw32 MyGame1 -S
+include_path=.\,..\..\Lib\Zcode,..\..\Lib\Contrib | more

PC/Win32 Inform 6.21 (30th April 1999)

C:\My Documents\Inform\Games\MyGame1>pause "at end of compilation"
Press any key to continue . . .
```

Press the space bar, then close the DOS window.

NOTE: on Windows NT, 2000 and XP, the DOS window closes of its own accord when you press the space bar.

5. A story file `MyGame1.z5` has appeared in the folder; this is the compiled game, which you can play using an interpreter:

Name	Size	Type	Modified
MyGame1.bat	1KB	MS-DOS Batch File	11/02/2002 09:06
MyGame1.inf	2KB	Setup Information	11/02/2002 09:06
MyGame1.z5	78KB	Z5 File	19/02/2002 13:25

The extension of `.z5` signifies that the story file contains a Z-Machine game in Version 5 (today's standard) format.

6. Use Windows Explorer to display the contents of the `Inform\Bin\Frotz` folder, and double-click `Frotz.exe`; the interpreter presents an `Open a Z-code Game` dialog box.
7. Browse to display the `Inform\Games\MyGame1` folder, and select `MyGame1.z5`. Click `Open`. The game starts running in the Windows Frotz 2002 window.
8. When you tire of "playing" the game – which won't take long – you can type the QUIT command, you can select `File > Exit`, or you can simply close the Frotz window.
9. Using the same techniques, you can compile and play `Ruins.inf`, which is held in the `Inform\Games\Download` folder. RUINS is the game used as an example throughout the *Inform Designer's Manual.*

Setting file associations

The business of first starting the interpreter, and then locating the story file that you want to play, is clumsy and inconvenient. You'll probably find it easier to automatically associate story files whose extension is `.z5` with the Frotz interpreter.

1. Double-click `MyGame1.z5`; Windows asks you to select the program which is to open it:
   - type `Z-machine game` as the "`Description for...`"
   - click to select "`Always use this program...`"
   - click `Other...`
2. Browse to display the `Inform\Bin\Frotz` folder, and select `Frotz.exe`. Click `Open`. From now on, you'll be able to play a game simply by double-clicking its `.z5` story file.

Changing the Windows icon

If the Windows icon that's displayed alongside `MyGame1.z5` doesn't look right, you can change it.

1. In Windows Explorer, select `View > Options...` and click `File Types`:
   - select the game file type in the list, which is in order either of application (Frotz) or of extension (Z5)
   - click `Edit...`
2. In the Edit File Type dialog, click `Change Icon`.

3. In the `Change Icon` dialog, ensure that the file name is `Inform\Bin\Frotz\Frotz.exe`, and select one of the displayed icons. Click `OK` to close all the dialogs.

   The files in the folder should now look like this:

Name	Size	Type	Modified
MyGame1.bat	1KB	MS-DOS Batch File	11/02/2002 09:06
MyGame1.inf	2KB	Setup Information	11/02/2002 09:06
MyGame1.z5	78KB	Z-machine game	19/02/2002 13:25

Compiling using a batch file

You can view – and of course change – the contents of `MyGame1.bat`, the batch file which you double-click to run the compiler, using any text editor. You'll see two lines, something like this (the first chunk is all on one long line, with a space between the `-S` and the `+include_path`):

```
..\..\Lib\Zcode\Infrmw32 MyGame1 -S
            +include_path=.\,..\..\Lib\Zcode,..\..\Lib\Contrib | more
pause "at end of compilation"
```

These long strings of text are command lines – a powerful interface method predating the icons and menus that most computer users know. You won't need to master the command line interface in order to start using Inform, but this section will tell you what these particular command lines are doing. There are five parts to the first line:

1. `Infrmw32` refers to the compiler program, and `..\..\Lib\Zcode` is the name of the folder which contains it (addressed relative to *this* folder, the one which holds the source file).

2. `MyGame1` is the name of the Inform source file; you don't need to mention its extension of `.inf` if you don't want to.

3. `-S` is a compiler **switch**, a way of controlling detailed aspects of how the compiler operates. This particular switch, one of many, is turning on **Strict mode**, which makes the game less likely to misbehave when being played.

   NOTE: actually, the `-S` is redundant, since Strict mode is already on by default. We include it here as a reminder that (a) to turn Strict mode *off*, you change this setting to `-~S`, and (b) alphabetic case matters here: `-s` causes a display of compiler statistics (and `-~s` does nothing at all).

4. `+include_path=.\,..\..\Lib\Zcode,..\..\Lib\Contrib` tells the compiler where to look for files like `Parser` and `VerbLib` which you've Included. Three locations are suggested: this folder, which holds the source file (`.\`); the folder holding the standard library files (`..\..\Lib\Zcode`); the folder holding useful bits and pieces contributed by the Inform community (`..\..\Lib\Contrib`). The three locations are searched in that order.

5. `| more` causes the compiler to pause if it finds more mistakes than it can tell you about on a single screen, rather than have them scroll off the top of the MS-DOS window. Press the space bar to continue the compilation.

The second line – `pause "at end of compilation"` – just prevents the window from closing before you can read its contents, as it otherwise would on Windows NT, 2000 and XP.

You'll need to have a new batch file like this to match each new source file which you create. The only item which will differ in the new file is the name of the Inform source file – `MyGame1` in this example. You must change this to match the name of the new source file; everything else can stay the same in each `.bat` file that you create.

Getting a better editor

Although NotePad is adequate when you're getting started, you'll find life much easier if you obtain a more powerful editor program. We recommend TextPad, available as shareware from `http://www.textpad.com/`; in addition, there are some detailed instructions at `http://www.onyxring.com/informguide.aspx?article=14` on how to improve the way that TextPad works with Inform. The biggest single improvement, the one that will make game development dramatically simpler, is being able to compile your source file *from within* the editor. No need to save the file, switch to another window and double-click the batch file (and indeed, no further need for the batch file itself): just press a key while editing the file – and it compiles there and then. You can also run the interpreter with similar ease. The convenience of doing this far outweighs the small amount of time needed to obtain and configure TextPad.

Inform on an Apple Macintosh

Follow these steps:

1. Download `http://www.firthworks.com/roger/downloads/inform_mac_env.sit` to a temporary location on your Mac.

2. Use a tool like StuffIt Expander to unpack the downloaded file, giving you a new `Inform` folder. Move this folder (and its contents) to a suitable location on your Mac.

   In order to make the download small and fast, these folders include just enough to get you started as an Inform designer – the compiler and interpreter programs, the library files, the `Ruins.inf` example file from the *Inform Designer's Manual*, and a template for your own first game. A few other folders are included as placeholders where you could later download additional components, if you wanted them. As soon as possible, you should download the *Inform Designer's Manual* into the `Inform:Doc` folder – it's an

essential document to have, and has been omitted from this download only because of its 3MB size.

Getting a better editor

Although SimpleText (or OS X's TextEdit) is adequate when you're getting started, you'll find life much easier if you obtain a more powerful editor program. We recommend BBEdit Lite, available without charge from `http://www.barebones.com/products/bbedit_lite.html`.

More about the editor

As well as the ones that we recommend, other good text editors are listed at `http://www.firthworks.com/roger/editors/`. One feature that's well worth looking out for is "hotkey compilation" – being able to run the compiler from *within* the editor. Another is "syntax colouring", where the editor understands enough of Inform's syntax rules to colour-code your source file; for example: red for brackets, braces and parentheses `[ ] { }` and `( )`, blue for reserved words like `Object` and `print`, green for items in quotes like `'...'` and `"..."`, and so on. Syntax colouring is of great assistance in getting your source file correct and thus avoiding silly compilation errors.

More about the compiler

The Inform compiler is a powerful but undramatic software tool; it does an awful lot of work, but it does it all at once, without stopping to ask you any questions. Its input is a readable text source file; the output is a story file, also sometimes known as a **Z-code file** (because it contains the game translated into code for the Z-Machine, which we describe in the next section).

If you're lucky, the compiler will translate your source file into Z-code; perhaps surprisingly, it doesn't display any form of "success" message when it succeeds. Often, however, it fails, because of mistakes which you've made when writing the game. Inform defines a set of rules – a capital letter here, a comma there, these words only in a certain order, those words spelled just so – about which the compiler is extremely fussy. If you accidentally break the rules, the compiler complains: it refuses to write a Z-code file. *Do not worry about this*: the rules are easy to learn, but just as easy to break, and all Inform designers inadvertently do so on a regular basis. There's some additional information about dealing with these mistakes, and about controlling how the compiler behaves, in "Compiling your game" on page 167.

More about the interpreter

One of the big advantages of the way Inform works is that a compiled game – the Z-code story file – is portable between different computers. That's not just from one PC to another: exactly the same story file will run on a PC, a Mac, an Amiga, UNIX workstations, IBM mainframes, PalmOS hand-helds, and on dozens of other past, present and future computers. The magic that makes this happen is the interpreter program, a software tool which pretends to be a simple computer called a **Z-Machine**. The Z-Machine is an imaginary (or "virtual") computer, but its design has been very carefully specified, so that an expert programmer can quite easily build one. And that's exactly what has happened: a Macintosh guru has built an Inform interpreter which runs on Apple Macs, a UNIX wizard has built one for UNIX workstations, and so on. Sometimes, you even get a choice; for popular machines like the PC and the Mac there are several interpreters available. And the wonderful thing is: each of those interpreters, on each of those computers, is able to play every Inform game that's ever been written *and*, as a surprise bonus, all of the classic 1980s Infocom games like "Zork" and "The Hitchhiker's Guide to the Galaxy" as well!

(Actually, that last sentence is a slight exaggeration; a few games are very large, or have pictures included within them, and not all interpreters can handle this. However, with that small pinch of salt, it's pretty accurate.)

That's enough waffling: let's get started! It's time to begin designing our first game.

3 • Heidi: our first Inform game

E was an esquire, with pride on his brow;
F was a farmer, and followed the plough.

Each of the three games in this guide is created step by step; you'll get most benefit (especially to begin with) if you take an active part, typing in the source code on your computer. Our first game, described in this chapter and the two which follow, tells this sentimental little story:

> "Heidi lives in a tiny cottage deep in the forest. One sunny day, standing before the cottage, she hears the frenzied tweeting of a baby bird; its nest has fallen from the tall tree in the clearing! Heidi puts the bird into the nest, and then climbs the tree to place the nest back on its branch."

It's a very simple tale, but even so we'll cover quite a lot of ground before we have a finished Inform game. We'll get there in stages, first making a very rough approximation of the story, and then successively refining the details until it's good enough for an initial attempt (there's time later for more advanced stuff).

Creating a basic source file

The first task is to create an Inform source file template. Every game that we design will start out like this. Follow these steps:

1. Create an `Inform\Games\Heidi` folder (maybe by copying `Inform\Games\MyGame1`).
2. In that folder, use your text editor to create this source file `Heidi.inf`:

TYPE

```
!============================================================================
Constant Story "Heidi";
Constant Headline
            "^A simple Inform example
             ^by Roger Firth and Sonja Kesserich.^";

Include "Parser";
Include "VerbLib";

!============================================================================
! The game objects

!============================================================================
! Entry point routines

[ Initialise; ];

!============================================================================
! Standard and extended grammar

Include "Grammar";

!============================================================================
```

Soon, we'll explain what this means. For now, just type it all in, paying particular attention to those seven semicolons, and ensuring that the double quotes `"..."` always come in pairs. The lines beginning with exclamation marks, on the other hand, are purely decorative; they just make the file's structure a little easier to understand.

Ensure the file is named `Heidi.inf`, rather than `Heidi.txt` or `Heidi.inf.txt`.

Remember that, throughout this guide, we place the "TYPE" symbol alongside pieces of code that we recommend you to type into your own game files as you read through the examples (which, conversely, means that you *don't* need to type the unmarked pieces of code). You'll learn Inform more quickly by trying it for yourself, rather than just taking our word for how things work.

3. In the same folder, use your text editor to create the compilation support file `Heidi.bat` (on a PC; remember that the stuff before `pause` is one long line):

```
..\..\Lib\Zcode\Infrmw32 Heidi -S
             +include_path=.\,..\..\Lib\Zcode,..\..\Lib\Contrib | more
pause "at end of compilation"
```

   or `Heidi.icl` (on a Macintosh):

```
-S
+source_path=":::Games:Heidi"
+code_path=":::Games:Heidi"
+include_path=":::Games:Heidi,,::Contrib"
compile Heidi.inf
```

   Type in the file from scratch, or copy and edit `MyGame1.bat` (or `MyGame1.icl`). At this point, you should have a `Heidi` folder containing two files: `Heidi.inf` and either `Heidi.bat` or `Heidi.icl`.

4. Compile the source file `Heidi.inf`. (Refer back to "Inform on an IBM PC" on page 19 for guidance; on a Mac, run Inform-Z, click `Compile`, and select `Heidi.icl` in the dialog box.) If the compilation works, a story file `Heidi.z5` appears in the folder. If the compilation *doesn't* work, you've probably made a typing mistake; check everything until you find it.

5. You can run the story file in your Inform interpreter; you should see this (except that the Serial number will be different – it's based on the date):

```
Heidi
A simple Inform example
by Roger Firth and Sonja Kesserich.
Release 1 / Serial number 020827 / Inform v6.21 Library 6/10 SD

Darkness
It is pitch dark, and you can't see a thing.

>
```

When you get that far, your template source file is correct. Let's explain what it contains.

Understanding the source file

Although we've got a certain amount of freedom of expression, source files tend to conform to a standard overall structure: these lines at the start, that material next, those pieces coming at the end, and so on. What we're doing here is mapping out a structure that suits us, giving ourselves a clear framework onto which the elements of the game can be fitted. Having a clear (albeit sparse) map at the start will help us to keep things organised as the game evolves.

We can infer half a dozen Inform rules just by looking at the source file.

- When the compiler comes across an exclamation mark, it ignores the rest of the line. If the `!` is at the start of a line, the whole line is ignored; if the `!` is halfway along a line, the compiler takes note of the first half, and then ignores the exclamation mark and everything after it on that line. We call material following an exclamation mark, not seen by anybody else, a **comment**; it's often a remark that we write to remind ourselves of how something works or why we tackled a problem in a particular way. There's nothing special about those equals signs: they just produce clear lines across the page.

  It's always a good idea to comment code as you write it, for later it will help you to understand what was going on at a particular spot. Although it all seems clear in your head when you first write it, in a few months you may suspect that a totally alien mind must have produced that senseless gibberish.

  By the way, the compiler *doesn't* give special treatment to exclamation marks in quoted text: `!` within quotes `"..."` is treated as a normal character. On this line, the first `!` is part of the sequence (or **string**) of characters to be displayed:

  ```
  print "Hello world!";       ! <- is the start of this comment
  ```

- The compiler ignores blank lines, and treats lots of space like a single space (except when the spaces are part of a character string). So, these two rules tell us that we *could* have typed the source file like this:

  ```
  Constant Story "Heidi";
  Constant Headline
  "^A simple Inform example^by Roger Firth and Sonja Kesserich.^";
  Include "Parser";Include "VerbLib";
  [ Initialise; ];
  Include "Grammar";
  ```

  We didn't type it that way because, though shorter, it's much harder to read. When designing a game, you'll spend a lot of time studying what you've typed, so it's worthwhile taking a bit of care to make it as readable as possible.

- Every game needs the **constant** definitions for `Story` (the game's name) and `Headline` (typically, information on the game's theme, copyright, authorship and so on). These two **string** values, along with a release number and date, and details of the compiler, compose the **banner** which is displayed at the start of each game.

- Every game needs the three lines which `Include` the standard library files – that is, they merge those files' contents into your source file:

  ```
  Include "Parser";
  Include "VerbLib";
  ...
  Include "Grammar";
  ```

  They always have to be in this order, with `Parser` and `VerbLib` near the start of the file, and `Grammar` near the end.

- Every game needs to define an `Initialise` routine (note the British spelling):

  ```
  [ Initialise; ];
  ```

  The **routine** that we've defined here doesn't do anything useful, but it still needs to be present. Later, we'll come back to `Initialise` and explain what a routine is and why we need this one.

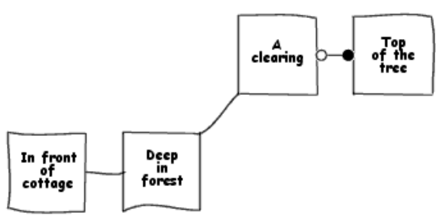
- You'll notice that each of the items mentioned in the previous three rules ends with a semicolon. Inform is very fussy about its punctuation, and gets really upset if you forget a terminating semicolon. In fact, the compiler just keeps reading your source file until it finds one; that's why we were able to take three lines to define the `Headline` constant

  ```
  Constant Headline
              "^A simple Inform example
               ^by Roger Firth and Sonja Kesserich.^";
  ```

Just to repeat what we said earlier: every game that you design will start out from a basic source file like this (in fact, it might be sensible to keep a copy of this template file in a safe place, as a starting point for future games). Think of this stuff as the basic preparation which you'll quickly come to take for granted, much as a landscape artist always begins by sizing the canvas before starting to paint. So, now that we've taken a quick tour of Inform's general needs, we can start thinking about what this particular game requires.

Defining the game's locations

A good starting point in any game is to think about the locations which are involved: this sketch map shows the four that we'll use:



In IF, we talk about each of these locations as a **room**, even though in this example none of them has four walls. So let's use Inform to define those rooms. Here's a first attempt:

```
Object  "In front of a cottage"
  with  description
            "You stand outside a cottage. The forest stretches east.",
  has   light;

Object  "Deep in the forest"
  with  description
            "Through the dense foliage, you glimpse a building to the west.
             A track heads to the northeast.",
  has   light;

Object  "A forest clearing"
  with  description
            "A tall sycamore stands in the middle of this clearing.
             The path winds southwest through the trees.",
  has   light;

Object  "At the top of the tree"
  with  description "You cling precariously to the trunk.",
  has   light;
```

Again, we can infer some general principles from these four examples:

- A room definition starts with the word `Object` and ends, about four lines later, with a semicolon. Each of the components that appears in your game – not only the rooms, but also the people, the things that you see and touch, intangibles like a sound, a smell, a gust of wind – is defined in this way; think of an "object" simply as the general term for the myriad thingies which together comprise the model world which your game inhabits.
- The phrase in double quotes following the word `Object` is the name that the interpreter uses to provide the player character with a list of the objects around her: where she is, what she can see, what she's holding, and so on.

  NOTE: we're using the word "player" to mean both the person who is playing the game, and the principal protagonist (often known as the player character) within the game itself. Since the latter – Heidi – is female, we'll refer to the player as "she" while discussing this game.
- A keyword `with` follows, which simply tells the compiler what to expect next.
- The word `description`, introducing another piece of text which gives more detail about the object: in the case of a room, it's the appearance of the surrounding environment when the player character is in that room. The textual description is given in double quotes, and is followed by a comma.
- Near the end, the keyword `has` appears, which again tells the compiler to expect a certain kind of information.
- The word `light` says that this object is a source of illumination, and that therefore the player character can see what's happening here. There has to

be at least one light source in every room (unless you want the player to be told that "It's pitch dark and you can't see a thing"); most commonly, that light source is the room itself.

A smidgeon of background may help set this into context (there's more in the next chapter). An object can have both **properties** (introduced by the keyword `with`) and **attributes** (written after the word `has`). A property has both a name (like `description`) and a value (like the character string `"You stand outside a cottage. The forest stretches east."`); an attribute has merely a name.

In a little while, when you play this game, you'll observe that it starts like this:

```
In front of a cottage
You stand outside a cottage. The forest stretches east.
```

And here you can see how the room's name (`In front of a cottage`) and description (`You stand outside a cottage. The forest stretches east.`) are used.

Joining up the rooms

We said that this was a first attempt at defining the rooms; it's fine as far as it goes, but a few bits of information are missing. If you look at the game's sketch map, you can see how the rooms are intended to be connected; from "Deep in the forest", for example, the player character should be able to move west towards the cottage, or northeast to the clearing. Now, although our descriptions mention or imply these available routes, we also need to explicitly add them to the room definitions in a form that the game itself can make sense of. Like this:

```
Object  before_cottage "In front of a cottage"
  with  description
            "You stand outside a cottage. The forest stretches east.",
        e_to forest,
  has   light;

Object  forest "Deep in the forest"
  with  description
            "Through the dense foliage, you glimpse a building to the west.
             A track heads to the northeast.",
        w_to before_cottage,
        ne_to clearing,
  has   light;

Object  clearing "A forest clearing"
  with  description
            "A tall sycamore stands in the middle of this clearing.
             The path winds southwest through the trees.",
        sw_to forest,
        u_to top_of_tree,
  has   light;

Object  top_of_tree "At the top of the tree"
  with  description "You cling precariously to the trunk.",
        d_to clearing,
  has   light;
```

We've made two changes to the room objects.

- First, between the word `Object` and the object's name in double quotes, we've inserted a different type of name: a private, internal identification, never seen by the player; one that we can use *within* the source file when one object needs to refer to another object. For example, the first room is identified as `before_cottage`, and the second as `forest`.

  Unlike the external name contained in double quotes, the internal identifier has to be a single word – that is, without spaces. To aid readability, we often use an underscore character to act as sort of pseudo-space: `before_cottage` is a bit clearer than `beforecottage`.

- Second, we've added lines after the object descriptions which use those internal identifiers to show how the rooms are connected; one line for each connection. The `before_cottage` object has this additional line:

  ```
  e_to forest,
  ```

  This means that a player standing in front of the cottage can type GO EAST (or EAST, or just E), and the game will transport her to the room whose internal identification is `forest`. If she tries to move in any other direction from this room, she'll be told "You can't go that way".

  What we've just defined is a *one-way* easterly connection: `before_cottage`→`forest`. The `forest` object has two additional lines:

  ```
  w_to before_cottage,
  ne_to clearing,
  ```

  The first line defines a westerly connection `forest`→`before_cottage` (thus enabling the player character to return to the cottage), and the second defines a connection `forest`→`clearing` which heads off to the northeast.

  Inform provides for eight "horizontal" connections (`n_to`, `ne_to`, `e_to`, `se_to`, `s_to`, `sw_to`, `w_to`, `nw_to`) two "vertical" ones (`u_to`, `d_to`) and two specials `in_to`, and `out_to`. You'll see some of these used for the remaining inter-room connections.

There's one last detail to attend to before we can test what we've done. You'll recollect that our story begins with Heidi standing in front of her cottage. We need to tell the interpreter that `before_cottage` is the room where the game starts, and we do this in the `Initialise` routine:

```
[ Initialise; location = before_cottage; ];
```

`location` is a **variable**, part of the library, which tells the interpreter in which room the player character currently is. Here, we're saying that, at the start of the game, the player character is in the `before_cottage` room.

Now we can add what we've done to the `Heidi.inf` source file template. At this stage, you should study the four room definitions, comparing them with the

sketch map until you're comfortable that you understand how to create simple rooms and define the connections between them.

```
!============================================================================
Constant Story "Heidi";
Constant Headline
            "^A simple Inform example
             ^by Roger Firth and Sonja Kesserich.^";

Include "Parser";
Include "VerbLib";

!============================================================================
! The game objects

Object  before_cottage "In front of a cottage"
  with  description
            "You stand outside a cottage. The forest stretches east.",
        e_to forest,
  has   light;

Object  forest "Deep in the forest"
  with  description
            "Through the dense foliage, you glimpse a building to the west.
             A track heads to the northeast.",
        w_to before_cottage,
        ne_to clearing,
  has   light;

Object  clearing "A forest clearing"
  with  description
            "A tall sycamore stands in the middle of this clearing.
             The path winds southwest through the trees.",
        sw_to forest,
        u_to top_of_tree,
  has   light;

Object  top_of_tree "At the top of the tree"
  with  description "You cling precariously to the trunk.",
        d_to clearing,
  has   light;

!============================================================================
! Entry point routines

[ Initialise; location = before_cottage; ];

!============================================================================
! Standard and extended grammar

Include "Grammar";

!============================================================================
```

Type this in, as always taking great care with the punctuation – watch those commas and semicolons. Compile it, and fix any mistakes which the compiler reports. You can then play the game in its current state. Admittedly, you can't do

very much, but you should be able to move freely among the four rooms that you've defined.

> NOTE: in order to minimise the amount of typing that you have to do, the descriptive text in this game has been kept as short as possible. In a real game, you would typically provide more interesting descriptions than these.

Adding the bird and the nest

Given what we said earlier, you won't be surprised to hear that both the bird and its nest are Inform objects. We'll start their definitions like this:

```
Object  bird "baby bird"
  with  description "Too young to fly, the nestling tweets helplessly.",
  has   ;

Object  nest "bird's nest"
  with  description "The nest is carefully woven of twigs and moss.",
  has   ;
```

You can see that these definitions have exactly the same format as the rooms we defined previously: a one-word internal identifier (`bird`, `nest`), and a word or phrase naming the object for the player's benefit (`baby bird`, `bird's nest`). They both have some descriptive detail: for a room this is printed when the player first enters, or when she types LOOK; for other objects it's printed when she EXAMINEs that object. What they *don't* have are connections (`e_to`, `w_to`, etc. apply only to rooms) or `light` (it's not necessary – the rooms ensure that light is available).

When the game is running, the player will want to refer to these two objects, saying for instance EXAMINE THE BABY BIRD or PICK UP THE NEST. For this to work reliably, we need to specify the word (or words) which relate to each object. Our aim here is flexibility: providing a choice of relevant vocabulary so that the player can use whatever term seems appropriate to her, with a good chance of it being understood. We add a line to each definition:

```
Object  bird "baby bird"
  with  description "Too young to fly, the nestling tweets helplessly.",
        name 'baby' 'bird' 'nestling',
  has   ;

Object  nest "bird's nest"
  with  description "The nest is carefully woven of twigs and moss.",
        name 'bird^s' 'nest' 'twigs' 'moss',
  has   ;
```

The `name` introduces a list in single quotes '...'. We call each of those quoted things a **dictionary word**, and we do mean "word", not "phrase" (`'baby' 'bird'` rather than `'baby bird'`); you can't uses spaces, commas or periods *in* dictionary words, though there's a space *between* each one, and the whole list ends with a comma. The idea is that the interpreter decides which object a player is talking about by matching what she types against the full set of all dictionary words. If the player

mentions BIRD, or BABY BIRD, or NESTLING, it's the `baby bird` that she means; if she mentions NEST, BIRD'S NEST or MOSS, it's the `bird's nest`. And if she types NEST BABY or BIRD TWIGS, the interpreter will politely say that it doesn't understand what on earth she's talking about.

> NOTE: you'll notice the use of `'bird^s'` to define the dictionary word BIRD'S; this oddity is necessary because the compiler expects the single quotes in the list always to come in pairs – one at the start of the dictionary word, and one at the end. If we had typed `'bird's'` then the compiler would find the opening quote, the four letters `b i r` and `d`, and what looks like the closing quote. So far so good; it's read the word BIRD and now expects a space before the next opening quote... but instead finds `s'` which makes no sense. In cases like this we must use the circumflex `^` to *represent* the apostrophe, and the compiler then treats `bird's` as a dictionary word.

You may be wondering why we need a list of `name` words for the bird and its nest, yet we didn't when we defined the rooms? It's because the player can't interact with a room in the same way as with other objects; for example, she doesn't need to say EXAMINE THE FOREST – just being there and typing LOOK is sufficient.

The bird's definition is complete, but there's an additional complexity with the nest: we need to be able to put the bird into it. We do this by labelling the nest as a `container` – able to hold other objects – so that the player can type PUT (or INSERT) BIRD IN (or INTO) NEST. Furthermore, we label it as `open`; this prevents the interpreter from asking us to open it before putting in the bird.

```
Object  nest "bird's nest"
  with  description "The nest is carefully woven of twigs and moss.",
        name 'bird^s' 'nest' 'twigs' 'moss',
  has   container open;
```

Both objects are now defined, and we can incorporate them into the game. To do this, we need to choose the locations where the player will find them. Let's say that the bird is found in the forest, while the nest is in the clearing. This is how we set this up:

TYPE

```
Object  bird "baby bird" forest
  with  description "Too young to fly, the nestling tweets helplessly.",
        name 'baby' 'bird' 'nestling',
  has   ;

Object  nest "bird's nest" clearing
  with  description "The nest is carefully woven of twigs and moss.",
        name 'bird^s' 'nest' 'twigs' 'moss',
  has   container open;
```

Read that first line as: "Here's the definition of an object which is identified within this file as `bird`, which is known to the player as `baby bird`, and which is initially located inside the object identified within this file as `forest`."

Where in the source file do these new objects fit? Well, anywhere really, but you'll find it convenient to insert them following the rooms where they're found.

This means adding the bird just after the forest, and the nest just after the clearing. Here's the middle piece of the source file:

```
!============================================================================
! The game objects

Object  before_cottage "In front of a cottage"
  with  description
            "You stand outside a cottage. The forest stretches east.",
        e_to forest,
  has   light;

Object  forest "Deep in the forest"
  with  description
            "Through the dense foliage, you glimpse a building to the west.
             A track heads to the northeast.",
        w_to before_cottage,
        ne_to clearing,
  has   light;

Object  bird "baby bird" forest
  with  description "Too young to fly, the nestling tweets helplessly.",
        name 'baby' 'bird' 'nestling',
  has   ;

Object  clearing "A forest clearing"
  with  description
            "A tall sycamore stands in the middle of this clearing.
             The path winds southwest through the trees.",
        sw_to forest,
        u_to top_of_tree,
  has   light;

Object  nest "bird's nest" clearing
  with  description "The nest is carefully woven of twigs and moss.",
        name 'bird^s' 'nest' 'twigs' 'moss',
  has   container open;

Object  top_of_tree "At the top of the tree"
  with  description "You cling precariously to the trunk.",
        d_to clearing,
  has   light;

!============================================================================
```

Make those changes, recompile the game, play it and you'll see this:

```
Deep in the forest
Through the dense foliage, you glimpse a building to the west. A track heads
to the northeast.

You can see a baby bird here.

>
```

Adding the tree and the branch

The description of the clearing mentions a tall sycamore tree, up which the player character supposedly "climbs". We'd better define it:

TYPE

```
Object  tree "tall sycamore tree" clearing
  with  description
            "Standing proud in the middle of the clearing,
             the stout tree looks easy to climb.",
        name 'tall' 'sycamore' 'tree' 'stout' 'proud',
  has   scenery;
```

Everything there should be familiar, apart from that `scenery` at the end. We've already mentioned the tree in the description of the forest clearing, so we don't want the interpreter adding "You can see a tall sycamore tree here" afterwards, as it does for the bird and the nest. By labelling the tree as `scenery` we suppress that, and also prevent it from being picked up by the player character.

One final object: the branch at the top of the tree. Again, not many surprises in this definition:

TYPE

```
Object  branch "wide firm bough" top_of_tree
  with  description "It's flat enough to support a small object.",
        name 'wide' 'firm' 'flat' 'bough' 'branch',
  has   static supporter;
```

The only new things are those two labels. `static` is similar to `scenery`: it prevents the branch from being picked up by the player character, but *doesn't* suppress mention of it when describing the setting. And `supporter` is rather like the `container` that we used for the nest, except that this time the player character can put other objects *onto* the branch. (In passing, we'll mention that an object can't normally be both a `container` *and* a `supporter`.) And so here are our objects again:

```
!============================================================================
! The game objects

Object  before_cottage "In front of a cottage"
  with  description
            "You stand outside a cottage. The forest stretches east.",
        e_to forest,
  has   light;

Object  forest "Deep in the forest"
  with  description
            "Through the dense foliage, you glimpse a building to the west.
             A track heads to the northeast.",
        w_to before_cottage,
        ne_to clearing,
  has   light;

Object  bird "baby bird" forest
  with  description "Too young to fly, the nestling tweets helplessly.",
        name 'baby' 'bird' 'nestling',
  has   ;

Object  clearing "A forest clearing"
  with  description
            "A tall sycamore stands in the middle of this clearing.
             The path winds southwest through the trees.",
        sw_to forest,
        u_to top_of_tree,
  has   light;
```

```
Object  nest "bird's nest" clearing
  with  description "The nest is carefully woven of twigs and moss.",
        name 'bird^s' 'nest' 'twigs' 'moss',
  has   container open;

Object  tree "tall sycamore tree" clearing
  with  description
            "Standing proud in the middle of the clearing,
             the stout tree looks easy to climb.",
        name 'tall' 'sycamore' 'tree' 'stout' 'proud',
  has   scenery;

Object  top_of_tree "At the top of the tree"
  with  description "You cling precariously to the trunk.",
        d_to clearing,
  has   light;

Object  branch "wide firm bough" top_of_tree
  with  description "It's flat enough to support a small object.",
        name 'wide' 'firm' 'flat' 'bough' 'branch',
  has   static supporter;

!============================================================================
```

Once again, make the changes, recompile, and investigate what you can do in your model world.

Finishing touches

Our first pass at the game is nearly done; just two more changes to describe. The first is easy: Heidi wouldn't be able to climb the tree carrying the bird and the nest separately: we want the player character to put the bird into the nest first. One easy way to enforce this is by adding a line near the top of the file:

```
!============================================================================
Constant Story "Heidi";
Constant Headline
            "^A simple Inform example
             ^by Roger Firth and Sonja Kesserich.^";

Constant MAX_CARRIED 1;
```

The value of `MAX_CARRIED` limits the number of objects that the player character can be holding at any one time; by setting it to 1, we're saying that she can carry the bird or the nest, but not both. However, the limit ignores the contents of `container` or `supporter` objects, so the nest with the bird inside it is still counted as one object.

The other change is slightly more complex and more important: there's currently no way to "win" the game! The goal is for the player character to put the bird in the nest, take the nest to the top of the tree, and place it on the branch; when that happens, the game should be over. This is one way of making it happen:

TYPE

```
Object  branch "wide firm bough" top_of_tree
  with  description "It's flat enough to support a small object.",
        name 'wide' 'firm' 'flat' 'bough' 'branch',
        each_turn [; if (nest in branch) deadflag = 2; ],
  has   static supporter;
```

NOTE: here's an explanation of what's going on. If you find this difficult to grasp, don't worry. It's the hardest bit so far, and it introduces several new concepts all at once. Later in the guide, we'll explain those concepts more clearly, so you can just skip this bit if you want.

The variable `deadflag`, part of the library, is normally 0. If you set its value to 2, the interpreter notices and ends the game with "You have won". The statement:

```
if (nest in branch) deadflag = 2;
```

should be read as: "Test whether the `nest` is currently in the `branch` (if the branch is a `container`) or on it (if the branch is a `supporter`); if it is, set the value of `deadflag` to 2; if it isn't, do nothing." The surrounding part:

```
each_turn [; ... ],
```

should be read as: "At the end of each turn (when the player is in the same room as the branch), do whatever is written inside the square brackets". So, putting that all together:

- At the end of each turn (after the player has typed something and pressed the Enter key, and the interpreter has done whatever was requested) the interpreter checks whether the player and the `branch` are in the same room. If not, nothing happens. If they're together, it looks to see where the `nest` is. Initially it's in the `clearing`, so nothing happens.
- Also at the end of each turn, the interpreter checks the value of `deadflag`. Usually it's 0, so nothing happens.
- Finally the player character puts the `nest` on the `branch`. "Aha!" says the interpreter (to itself, of course), and sets the value of `deadflag` to 2.
- Immediately afterwards, (another part of) the interpreter checks and finds that the value of `deadflag` has changed to 2, which means that the game is successfully completed; so, it says to the player, "you've won!"

That's as far as we'll take this example for now. Make those final changes, recompile, and test what you've achieved. You'll probably find a few things that could be done better – even on a simple game like this there's considerable scope for improvement – so we'll revisit Heidi in her forest shortly. First, though, we'll recap what we've learnt so far.

4 • Reviewing the basics

G was a gamester, who had but ill-luck;
H was a hunter, and hunted a buck.

oing through the design of our first game in the previous chapter has introduced all sorts of Inform concepts, often without giving you much detail about what's been happening. So let's review some of what we've learnt so far, in a slightly more organised fashion. We'll talk about "Constants and variables" on page 41, "Object definitions" on page 42, "Object relationships – the object tree" on page 44, "Things in quotes" on page 47, and "Routines and statements" on page 48.

Constants and variables

Superficially similar, constants and variables are actually very different beasts.

Constants

A **constant** is a name to which a value is given once and once only; you can't later use that name to stand for a different value. Think of it as a stone tablet on which you carve a number: a carving can't be undone, so that you see the same number every time you look at the stone.

So far, we've seen a `Constant` being set up with its value as a string of characters:

```
Constant Story "Heidi";
```

and as a number:

```
Constant MAX_CARRIED 1;
```

Those two examples represent the most common ways in which constants are used in Inform.

Variables

A **variable** is a name to which a value is given, but that value can be changed to a different one at any time. Think of it as a blackboard on which you mark a number in chalk: whenever you need to, just wipe the board and write up a new number.

We haven't set up any variables of our own yet, though we've used a couple which the library created like this:

```
Global location;
Global deadflag;
```

The value of a **global variable** created in this way is initially 0, but you can change it at any time. For example, we used the statement:

```
location = before_cottage;
```

to reset the value of the `location` variable to the `before_cottage` object, and we wrote:

```
if (nest in branch) deadflag = 2;
```

to reset the value of the `deadflag` variable to 2.

Later, we'll talk about the **local variable** (see "Routines" on page 157) and about using object properties as variables (see "Objects" on page 155).

Object definitions

The most important information you should have gleaned from the previous chapter is that your entire game is defined as a series of objects. Each room is an object, each item that the player sees and touches is an object; indeed the player herself is also an object (one that's automatically defined by the library).

The general model of an **object** definition looks like this:

```
Object   obj_id  "external_name"  parent_obj_id

  with   property  value ,
         property  value ,
         ...
         property  value ,

  has    attribute  attribute  ... attribute
  ;
```

The definition starts with the word `Object` and ends with a semicolon; in between are three major blocks of information:

- immediately after the word `Object` is the header information;
- the word `with` introduces the object's **properties**;
- the word `has` introduces the object's **attributes**.

Object headers

An object header comprises up to three items, all optional:

- An internal *obj_id* by which other objects refer to this object. It's a single word (though it can contain digits and underscores) of up to thirty-two characters, and it must be unique within the game. You can omit the *obj_id* if this object isn't referred to by any other objects.

  For example: `bird`, `tree`, `top_of_tree`.

- An *external_name*, in double quotes, which is what the interpreter uses when referring to the object. It can be one or more words, and need not be unique

(for instance, you might have several `"Somewhere in the desert"` rooms). Although not mandatory, it's best to give *every* object an *`external_name`*.

For example: `"baby bird"`, `"tall sycamore tree"`, `"At the top of the tree"`.

- The internal *`obj_id`* of another object which is the initial location of this object (its "parent" – see the next section) at the start of the game. This is omitted from objects which have no initial parent; it's *always* omitted from a room.

  For example: the definition of the `bird` starts like this, specifying that at the start of the game, it can be found in the `forest` room (though later the player character will pick it up and move it around):

  ```
  Object  bird "baby bird" forest
  ...
  ```

  The `tree` starts like this; the only real difference is that, because the player character can't move a `scenery` object, it's always going to be in the `clearing`:

  ```
  Object  tree "tall sycamore tree" clearing
  ...
  ```

  NOTE: there's an alternative method for defining an object's initial location, using "arrows" rather than the parent's internal *`obj_id`*. For example, the definition of the bird could have started like this:

  ```
  Object  -> bird "baby bird"
  ...
  ```

  We don't use the arrows method in this guide, though we do describe how it works in "Setting up the object tree" on page 163.

Object properties

An object's property definitions are introduced by the `with` keyword. An object can have any number of properties, and they can be defined in any order. Each definition has two parts: a name, and a value; there's a space between the two parts, and a comma at the end.

Think of each property as a variable which is specifically associated with that object. The variable's initial setting is the supplied value; if necessary, it can be reset to other values during play (though in fact most property values don't change in this way).

Here are examples of the properties that we've come across so far:

```
description "The nest is carefully woven of twigs and moss.",
e_to forest,
name 'baby' 'bird' 'nestling',
each_turn [; if (nest in branch) deadflag = 2; ],
```

By happy coincidence, those examples also demonstrate most of the different types of value which can be assigned to a property. The value associated with the `description` property in this particular example is a string of characters in double

quotes; the value associated with this `e_to` property is the internal identity of an object; the `name` property is a bit unusual – its value is a list of dictionary words, each in single quotes; the `each_turn` property has a value which is an **embedded routine** (see "Embedded routines" on page 50). The only other type of value which is commonly found is a simple number; for example:

```
capacity 10,
```

In all, the library defines around forty-eight standard properties – like `name` and `each_turn` – which you can associate with your objects; there's a complete list in "Object properties" on page 238. And in "William Tell: in his prime" on page 81 we show you how to invent your own property variables.

Object attributes

An object's attribute list is introduced by the `has` keyword. An object can have any number of attributes, and they can be listed in any order, with a space between each.

As with properties, you can think of each attribute as a variable which is specifically associated with that object. However, an attribute is a much more limited form of variable, since it can have only two possible states: present, and absent (also known as set/clear, on/off, or true/false; incidentally, a two-state variable like this is often called a **flag**). Initially, an attribute is either present (if you mention its name in the list) or absent (otherwise); if necessary, its state can change during play (and this is relatively common). We often say that a certain object currently *has* a certain attribute, or that conversely it *hasn't* got it.

The attributes that we've come across so far are:

```
container light open scenery static supporter
```

Each of those answers a question: Is this object a container? Does it provide light? and so on. If the attribute is present then the answer is Yes; if the attribute isn't present, the answer is No.

In all, the library defines around thirty standard attributes which you can associate with your objects; there's a complete list in "Object attributes" on page 241.

Object relationships – the object tree

Not only is your game composed entirely of objects, but also Inform takes great care to keep track of the relationships between those objects. By "relationship" we don't mean that Walter is Wilhelm's son, while Helga and Wilhelm are just good friends; it's a much more comprehensive exercise in recording exactly where each object is located, relative to the other objects in the game.

Despite what we just said, Inform relationships *are* managed in terms of **parent** and **child** objects, though in a much broader sense than Wilhelm and Walter.

When the player character is in a particular room – for example the forest – we can say that:

- the forest object is *the* parent of the player object, or alternatively
- the player object is *a* child of the forest object.

Also, if the player is carrying an object – for example the nest – we say that:

- the player object is *the* parent of the nest object, or that
- the nest object is *a* child of the player object.

Note the emphasis there: an object has exactly *one* parent (or no parent at all), but can have *any number* of child objects (including none).

For an example of an object having more than one child, think about the way we defined the nest and tree objects:

```
Object  nest "bird's nest" clearing
...

Object  tree "tall sycamore tree" clearing
...
```

We used the third of the header items to say that the clearing was the parent of the nest, and also that the clearing was the parent of the tree; that is, both nest and tree are child objects of the clearing.

> NOTE: a "room" isn't anything magical; it's just an object which *never* has a parent, and which *may* from time to time have the player object as a child.

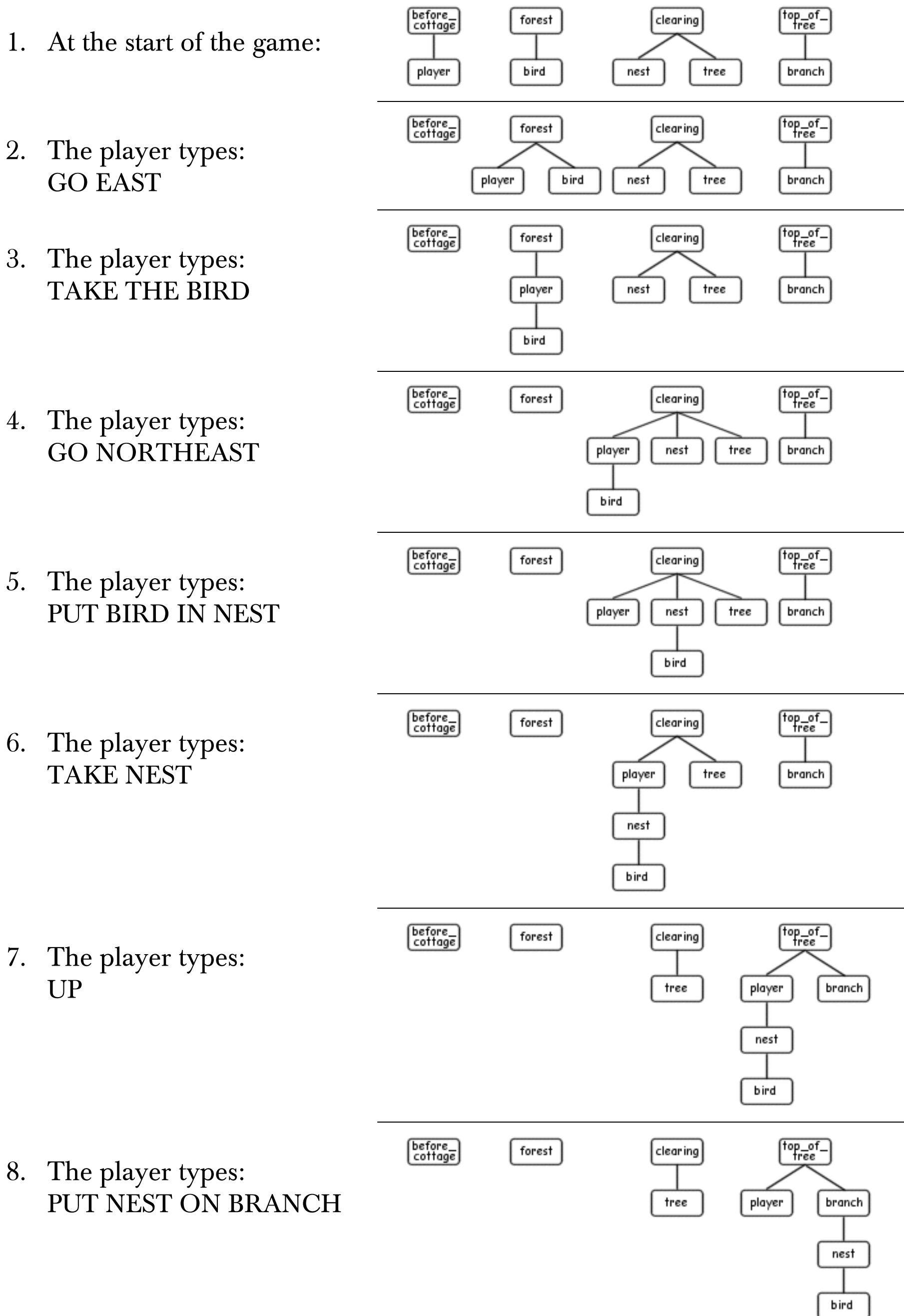
When we defined the bird, we placed it in the forest, like so:

```
Object  bird "baby bird" forest
...
```

We didn't place any other objects in that room, so at the start of the game the forest was the parent of the bird (and the bird was the only child of the forest). But what happens when the player character, initially in the `before_cottage` room, goes EAST to the forest? Answer: the player's parent is now the forest, and the forest has two children – the bird *and* the player. This is a key principle of the way Inform manages its objects: the parent–child relationships between objects change continuously, often dramatically, as the game progresses.

Another example of this: suppose the player character picks up the bird. This causes another change in the relationships. The bird is now a child of the player (and *not* of the forest), and the player is both a parent (of the bird) and a child (of the forest).

In this diagram, we show how the object relationships change during the course of the game. The straight lines represent parent–child relationships, with the parent object at the top of the line, and the child object at the bottom.



In this short example, we've taken a lot of time and space to spell out exactly how the objects relationship patterns – generally known as the **object tree** – appear

at each stage. Normally you wouldn't bother with this much detail (a) because the interpreter does most of the work for you, and (b) because in a real game there are usually too many objects for you to keep track of. What's important is that you understand the basic principles: at any moment in time an object either has no parent (which probably means either that it's a room, or that it's floating in hyperspace and not currently part of the game) or exactly one parent – the object that it's "in" or "on" or "a part of". However, there's no restriction on the number of children that an object can have.

There's a practical use for these relationships, covered in detail further on. As a designer, you can refer to the current parent or children of any given object with the `parent`, `child` and `children` routines, and this is one feature that you will be using frequently. There are also other routines associated with the object tree, to help you keep track of the objects or move them around. We'll see them one by one in the next chapters. For a quick summary, see "Objects" on page 155.

Things in quotes

Inform makes careful distinction between double and single quotes.

Double quotes

Double quotes `"..."` surround a **string** – a letter, a word, a paragraph, or almost any number of characters – which you want the interpreter to display while the game is being played. You can use the tilde `~` to represent a double quote inside the string, and the circumflex `^` to represent a **newline** (line break) character. Upper-case and lower-case letters are treated as different.

A long string can be split over several lines; Inform transforms each line break (and any spaces around it) into a single space (extra spaces *not* at a line break are preserved, though). These two strings are equivalent:

```
"This is a     string of characters."

"This
  is
    a     string
              of characters."
```

When the interpreter displays a long character string – for example, while describing a feature-packed room – it employs automatic word-wrapping to fit the text to the player's screen. This is where you might insert `^` characters to force line breaks to appear, thus presenting the text as a series of paragraphs.

So far, we've seen strings used as the value of a `Constant`:

```
Constant Headline
             "^A simple Inform example
              ^by Roger Firth and Sonja Kesserich.^";
```

which could equally have been defined thus:

```
Constant Headline
            "^A simple Inform example^by Roger Firth and Sonja Kesserich.^";
```

and as the value of an object `description` property:

```
description "Too young to fly, the nestling tweets helplessly.",
```

Later, you'll find that they're also very common in `print` statements.

Single quotes

Single quotes `'...'` surround a **dictionary word**. This has to be a single word – no spaces – and generally contains only letters (and occasionally numbers and hyphens), though you can use `^` to represent an apostrophe inside the word. Upper-case and lower-case letters are treated as identical; also, the interpreter normally looks only at the first nine characters of each word that the player types.

When the player types a command, the interpreter divides what was typed into individual words, which it then looks up in the dictionary. If it finds all the words, and they seem to represent a sensible course of action, that's what happens next.

So far, we've seen dictionary words used as the values of an object `name` property:

```
name 'bird^s' 'nest' 'twigs' 'moss',
```

and indeed that's just about the only place where they commonly occur.

You'll save yourself a lot of confusion by remembering the distinction: Double quotes for Output, Single quotes for Input (DOSI).

Routines and statements

A routine is a collection of statements, which are performed (or we often say "are executed") at run-time by the interpreter. There are two types of routine, and about two dozen types of statement (there's a complete list in "Statements" on page 152; see also "Inform language" on page 229).

Statements

A **statement** is an instruction telling the interpreter to perform a particular task – to "do something" – while the game is being played. A real game usually has lots and lots of statements, but so far we've encountered only a few. We saw:

```
location = before_cottage;
```

which is an example of an **assignment** statement, so-called because the equals sign `=` assigns a new value (the internal ID of our `before_cottage` room) to a variable (the global variable `location` which is part of the library). Later we saw:

```
if (nest in branch) deadflag = 2;
```

which is actually *two* statements: an assignment, preceded by an `if` statement:

```
if (nest in branch) ...
```

The `if` statement tests a particular condition; if the condition is true, the interpreter executes whatever statement comes next; if it isn't true, the interpreter ignores the next statement. In this example, the interpreter is testing whether the `nest` object is "in" or "on" (which we now know means "is a child of") the `branch` object. For most of the game, that condition is not true, and so the interpreter ignores the following statement. Eventually, when the condition becomes true, the interpreter executes that statement: it performs an assignment:

```
deadflag = 2;
```

which changes the value of the library variable `deadflag` from its current value to 2. Incidentally, `if` statements are often written on two lines, with the "controlled" statement indented. This makes it easier to read, but doesn't change the way that it works:

```
if (nest in branch)
    deadflag = 2;
```

The thing that's being controlled by the `if` statement doesn't have to be an assignment; it can be any kind of statement. In fact, you can have lots of statements, not just one, controlled by an `if` statement. We'll talk about these other possibilities later. For now, just remember that the only place where you'll find statements are within standalone routines and embedded routines.

Standalone routines

A **standalone routine** is a series of statements, collected together and given a name. When the routine is "called" – by its given name – those statements are executed. Here's the one that we've defined:

```
[ Initialise; location = before_cottage; ];
```

Because it's such a tiny routine, we placed it all on a single line. Let's rewrite it to use several lines (as with the `if` statement, this improves the readability, but doesn't affect how it works);

```
[ Initialise;
    location = before_cottage;
];
```

The `[ Initialise;` is the start of the routine, and defines the name by which it can be "called". The `];` is the end of the routine. In between are the statements – sometimes known as the body of the routine – which are executed when the routine is called. And how is that done? By a statement like this:

```
Initialise();
```

That single statement, the routine's name followed by opening and closing parentheses, is all that it takes to call a routine. When it comes across a line like this, the interpreter executes the statements – in this example there's only one, but there may be ten, twenty, even a hundred of them – in the body of the

routine. Having done that, the interpreter resumes what it was doing, on the line following the `Initialise();` call.

> NOTE: you may have noticed that, although we've defined a routine named `Initialise`, we've never actually called it. Don't worry – the routine *is* called, by the Inform library, right at the start of a game.

Embedded routines

An **embedded routine** is much like a standalone routine, though it doesn't have a name and doesn't end in a semicolon. This is the one that we defined:

```
[; if (nest in branch) deadflag = 2; ]
```

except that we didn't write it in isolation like that: instead, we defined it to be the value of an object property:

```
each_turn [; if (nest in branch) deadflag = 2; ],
```

which would have worked just the same if we'd written it like this:

```
each_turn [;
    if (nest in branch)
        deadflag = 2;
],
```

All embedded routines are defined in this manner: as the value of an object property. That's where they're embedded – inside an object. The introductory characters `[;` maybe look a little odd, but it's really only the same syntax as for a standalone routine, only without a name between the `[` and `;`.

For calling an embedded routine, thus causing the statements it contains to be executed, the method that we described for a standalone routine won't work. An embedded routine has no name, and needs none; it's *automatically* called by the library at appropriate moments, which are determined by the role of the property for which it is the value. In our example, that's at the end of every turn in which the player character is in the same room as the branch. Later, we'll see other examples of embedded routines, each designed to perform a task which is appropriate for the property whose value it is; we'll also see that it *is* possible to call an embedded routine yourself, using an `obj_id.property()` syntax – in this example, we could call the routine by writing `branch.each_turn()`.

There's more about these topics in "Routines and arguments" on page 59, "Routines" on page 157 and in "The marketplace" on page 91.

That ends our review of the ground covered in our first game. We'll have more to say about most of this later, but we're trying not to overload you with facts at this early stage. What we'd like you to do is to look back at the source of the game, and ensure that you can recognise all the elements which this chapter has described. Then, we'll move on to fix a few of the game's more important defects.

5 • Heidi revisited

I was an innkeeper, who loved to carouse;
J was a joiner, and built up a house.

In even the simplest story, there's bound to be scope for the player to attempt activities that you hadn't anticipated. Sometimes there may be alternative ways of approaching a problem: if you can't be sure which approach the player will take, you really ought to allow for all possibilities. Sometimes the objects you create and the descriptions you provide may suggest to the player that doing such-and-such should be possible, and, within reason, you ought to allow for that also. The basic game design is easy: what takes the time, and makes a game large and complex, is taking care of all the *other* things that the player may think of trying.

Here, we try to illustrate what this means by addressing a few of the more glaring deficiencies in our first game.

Listening to the bird

Here's a fragment of the game being played:

```
Deep in the forest
Through the dense foliage, you glimpse a building to the west. A track heads
to the northeast.

You can see a baby bird here.

>EXAMINE THE BIRD
Too young to fly, the nestling tweets helplessly.

>LISTEN TO BIRD
You hear nothing unexpected.

>
```

That's not too smart, is it? Our description specifically calls the player's attention to the sound of the bird – and then she finds out that we've got nothing special to say about its helpless tweeting.

The library has a stock of actions and responses for each of the game's defined verbs, so it can handle most of the player's input with a default, standard behaviour instead of remaining impertinently silent or saying that it doesn't understand what the player intends. "You hear nothing unexpected" is the library's standard LISTEN response, good enough after LISTEN TO NEST or LISTEN TO TREE, but fairly inappropriate here; we really need to substitute a more relevant response after LISTEN TO BIRD. Here's how we do it:

TYPE

```
Object  bird "baby bird" forest
  with  description "Too young to fly, the nestling tweets helplessly.",
        name 'baby' 'bird' 'nestling',
        before [; Listen:
            print "It sounds scared and in need of assistance.^";
            return true;
        ],
  has   ;
```

We'll go through this a step at a time:

1. We've added a new `before` property to our `bird` object. The interpreter looks at the property *before* attempting to perform any action which is directed specifically at this object:

   ```
   before [; ... ],
   ```

2. The value of the property is an embedded routine, containing a label and two statements:

   ```
   Listen:
   print "It sounds scared and in need of assistance.^";
   return true;
   ```

3. The label is the name of an action, in this case `Listen`. What we're telling the interpreter is: if the action that you're about to perform on the bird is a `Listen`, execute these statements first; if it's any other action, carry on as normal. So, if the player types EXAMINE BIRD, PICK UP BIRD, PUT BIRD IN NEST, HIT BIRD or FONDLE BIRD, then she'll get the standard response. If she types LISTEN TO BIRD, then our two statements get executed before anything else happens. We call this "trapping" or "intercepting" the action of Listening to the bird.

4. The two statements that we execute are, first:

   ```
   print "It sounds scared and in need of assistance.^";
   ```

   which causes the interpreter to display the string given in double quotes; remember that a `^` character in a string appears as a newline. Second, we execute:

   ```
   return true;
   ```

   which tells the interpreter that it doesn't need to do anything else, because we've handled the `Listen` action ourselves. And the game now behaves like this – perfect:

   ```
   >LISTEN TO BIRD
   It sounds scared and in need of assistance.

   >
   ```

The use of the `return true` statement probably needs a bit more explanation. An object's `before` property traps an action aimed at that object right at the start, before the interpreter has started to do anything. That's the point at which the statements in the embedded routine are executed. If the last of those statements

is `return true` then the interpreter assumes that the action has been dealt with by those statements, and so there's nothing left to do: no action, no message; nothing. On the other hand, if the last of the statements is `return false` then the interpreter carries on to perform the default action as though it hadn't been intercepted. Sometimes that's what you want it to do, but not here: if instead we'd written this:

```
Object  bird "baby bird" forest
  with  description "Too young to fly, the nestling tweets helplessly.",
        name 'baby' 'bird' 'nestling',
        before [; Listen:
            print "It sounds scared and in need of assistance.^";
            return false;
        ],
  has   ;
```

then the interpreter would first have displayed our string, and then carried on with its normal response, which is to display the standard message:

```
>LISTEN TO BIRD
It sounds scared and in need of assistance.
You hear nothing unexpected.

>
```

The technique that we've used here – intercepting an action aimed at a particular object in order to do something appropriate for that object – is one that we'll use again and again.

Entering the cottage

At the start of the game the player character stands "outside a cottage", which might lead her to believe that she can go inside:

```
In front of a cottage
You stand outside a cottage. The forest stretches east.

>IN
You can't go that way.

>
```

Again, that isn't perhaps the most appropriate response, but it's easy to change:

TYPE

```
Object  before_cottage "In front of a cottage"
  with  description
            "You stand outside a cottage. The forest stretches east.",
        e_to forest,
        in_to "It's such a lovely day -- much too nice to go inside.",
        cant_go "The only path lies to the east.",
  has   light;
```

The `in_to` property would normally link to another room, in the same way as the `e_to` property contain the internal ID of the `forest` object. However, if instead you set its value to be a string, the interpreter displays that string when the player tries

the IN direction. Other – unspecified – directions like NORTH and UP still elicit the standard “You can’t go that way” response, but we can change that too, by supplying a `cant_go` property whose value is a suitable string. We then get this friendlier behaviour:

```
In front of a cottage
You stand outside a cottage. The forest stretches east.

>IN
It's such a lovely day -- much too nice to go inside.

>NORTH
The only path lies to the east.

>EAST

Deep in the forest
...
```

There’s another issue here; since we haven’t actually implemented an object to represent the cottage, a perfectly reasonable EXAMINE COTTAGE command receives the obviously nonsensical reply “You can’t see any such thing”. That’s easy to fix; we can add a new `cottage` object, making it a piece of `scenery` just like the `tree`:

```
Object  cottage "tiny cottage" before_cottage
  with  description "It's small and simple, but you're very happy here.",
        name 'tiny' 'cottage' 'home' 'house' 'hut' 'shed' 'hovel',
  has   scenery;
```

This solves the problem, but promptly gives us another unreasonable response:

```
In front of a cottage
You stand outside a cottage. The forest stretches east.

>ENTER COTTAGE
That’s not something you can enter.

>
```

The situation here is similar to our LISTEN TO BIRD problem, and the solution we adopt is similar as well:

TYPE

```
Object  cottage "tiny cottage" before_cottage
  with  description "It's small and simple, but you're very happy here.",
        name 'tiny' 'cottage' 'home' 'house' 'hut' 'shed' 'hovel',
        before [; Enter:
            print_ret "It's such a lovely day -- much too nice to go inside.";
        ],
  has   scenery;
```

We use a `before` property to intercept the `Enter` action applied to the `cottage` object, so that we can display a more appropriate message. This time, however, we’ve done it using one statement rather than two. It turns out that the sequence “`print` a string which ends with a newline character, and then `return true`” is so

frequently needed that there's a special statement which does it all. That is, this single statement (where you'll note that the string *doesn't* need to end in ^):

```
print_ret "It's such a lovely day -- much too nice to go inside.";
```

works exactly the same as this pair of statements:

```
print "It's such a lovely day -- much too nice to go inside.^";
return true;
```

We could have used the shorter form when handling LISTEN TO BIRD, and we *will* use it from now on.

Climbing the tree

In the clearing, holding the nest and looking at the tree, the player is meant to type UP. Just as likely, though, she'll try CLIMB TREE (which currently gives the completely misleading response "I don't think much is to be achieved by that"). Yet another opportunity to use a `before` property – they really are very useful – but now with a difference.

TYPE

```
Object  tree "tall sycamore tree" clearing
  with  description
            "Standing proud in the middle of the clearing,
             the stout tree looks easy to climb.",
        name 'tall' 'sycamore' 'tree' 'stout' 'proud',
        before [; Climb:
            PlayerTo(top_of_tree);
            return true;
        ],
  has   scenery;
```

This time, when we intercept the `Climb` action applied to the `tree` object, it's not in order to display a better message; it's because we want to move the player character to another room, just as if she'd typed UP. Relocating the player character is actually quite a complex business, but fortunately all of that complexity is hidden: there's a standard **library routine** to do the job, not one that we've written, but one that's provided as part of the Inform system.

You'll remember that, when we first mentioned routines (see "Standalone routines" on page 49), we used the example of `Initialise()` and said that "the routine's name followed by opening and closing parentheses is all that it takes to call a routine". That was true for `Initialise()`, but not quite the whole story. To move the player character, we've got to specify where we want her to go, and we do that by supplying the internal ID of the destination room within the opening and closing parentheses. That is, instead of just `PlayerTo()` we call `PlayerTo(top_of_tree)`, and we describe `top_of_tree` as the routine's **argument**.

Although we've moved the player character to another room, we're still in the middle of the intercepted `Climb` action. As previously, we need to tell the interpreter that we've dealt with the action, and so we don't want the standard rejection message to be displayed. The `return true` statement does that, as usual.

Dropping objects from the tree

In a normal room like the `forest` or the `clearing`, the player can DROP something she's carrying and it'll effectively fall to the ground at her feet. Simple, convenient, predictable – except when the player is at the top of the tree. Should she DROP something from up there, having it land nearby might seem a bit improbable; much more likely that it would fall to the clearing below.

It looks like we might want to intercept the `Drop` action, but not quite in the way we've been doing up until now. For one thing, we don't want to complicate the definitions of the `bird` and the `nest` and any other objects we may introduce: much better to find a general solution that will work for all objects. And second, we need to recognise that not all objects are droppable; the player can't, for example, DROP THE BRANCH.

The best approach to the second problem is to intercept the `Drop` action *after* it has occurred, rather than beforehand. That way, we let the library take care of objects which aren't being held or which can't be dropped, and only become involved once a `Drop` has been successful. And the best approach to the first problem is to do this particular interception not on an object-by-object basis, as we have been doing so far, but instead for every `Drop` which takes place in our troublesome `top_of_tree` room. This is what we have to write:

TYPE

```
Object  top_of_tree "At the top of the tree"
  with  description "You cling precariously to the trunk.",
        d_to clearing,
        after [; Drop:
            move noun to clearing;
            return false;
        ],
  has   light;
```

Let's again take it a step at a time:

1. We've added a new `after` property to our `top_of_tree` object. The interpreter looks at the property *subsequent to* performing any action in this room:

   ```
   after [; ... ],
   ```

2. The value of the property is an embedded routine, containing a label and two statements:

   ```
   Drop:
   move noun to clearing;
   return false;
   ```

3. The label is the name of an action, in this case `Drop`. What we're telling the interpreter is: if the action that you've just performed here is a `Drop`, execute these statements before telling the player what you've done; if it's any other action, carry on as normal.

4. The two statements that we execute are first:

   ```
   move noun to clearing;
   ```

which takes the object which has just been moved from the `player` object to the `top_of_tree` object (by the successful `Drop` action) and moves it again so that its parent becomes the `clearing` object. That `noun` is a library variable that always contains the internal ID of the object which is the target of the current action. If the player types DROP NEST, `noun` contains the internal ID of the `nest` object; if she types DROP NESTLING then `noun` contains the internal ID of the `bird` object. Second, we execute:

```
return false;
```

which tells the interpreter that it should now let the player know what's happened. Here's the result of all this:

```
At the top of the tree
You cling precariously to the trunk.

You can see a wide firm bough here.

>DROP NEST
Dropped.

>LOOK

At the top of the tree
You cling precariously to the trunk.

You can see a wide firm bough here.

>DOWN

A forest clearing
A tall sycamore stands in the middle of this clearing. The path winds
southwest through the trees.

You can see a bird's nest (in which is a baby bird) here.

>
```

Of course, you might think that the standard message "Dropped" is slightly unhelpful in these non-standard circumstances. If you prefer to hint at what's just happened, you could use this alternative solution:

```
Object  top_of_tree "At the top of the tree"
  with  description "You cling precariously to the trunk.",
        d_to clearing,
        after [; Drop:
            move noun to clearing;
            print_ret "Dropped... to the ground far below.";
        ],
  has   light;
```

The `print_ret` statement does two things for us: displays a more informative message, and returns `true` to tell the interpreter that there's no need to let the player know what's happened – we've handled that ourselves.

Is the bird in the nest?

The game ends when the player character puts the nest onto the branch. Our assumption here is that the bird is inside the nest, but this might not be so; the player may have first taken up the bird and then gone back for the nest, or vice versa. It would be better not to end the game until we'd checked for the bird actually being in the nest; fortunately, that's easy to do:

TYPE

```
Object  branch "wide firm bough" top_of_tree
  with  description "It's flat enough to support a small object.",
        name 'wide' 'firm' 'flat' 'bough' 'branch',
        each_turn [; if (bird in nest && nest in branch) deadflag = 2; ],
  has   static supporter;
```

The extended `if` statement:

```
if (bird in nest && nest in branch) deadflag = 2;
```

should now be read as: "Test whether the `bird` is currently in (or on) the `nest`, *and* whether the `nest` is currently on (or in) the `branch`; if both parts are true, set the value of `deadflag` to 2; otherwise, do nothing".

Summing up

You should by now have some appreciation of the need not only to handle the obvious actions which were at the forefront of your mind when designing the game, but also as many as you can of the other possible ways that a player may choose to interact with the objects presented to her. Some of those ways will be highly intelligent, some downright dumb; in either case you should try to ensure that the game's response is at least sensible, even when you're telling the player "sorry, you can't do that".

The new topics that we've encountered here include these:

Object properties

Objects can have a `before` property – if there is one, the interpreter looks at it *before* performing an action which in some way involves that object. Similarly, you can provide an `after` property, which the interpreter looks at *after* performing an action but before telling the player what's happened. Both `before` and `after` properties can be used not only with tangible objects like the `bird`, `cottage` and `tree` (when they intercept actions aimed at that particular object) but also with rooms (when they intercept actions aimed at any object in that room).

The value of each `before` and `after` property is an embedded routine. If such a routine ends with `return false`, the interpreter then carries on with the next stage of the action which has been intercepted; if it ends with `return true`, the interpreter does nothing further for that action. By combining these possibilities, you can supplement the work done by a standard action with statements of your own, or you can replace a standard action completely.

Previously, we've seen connection properties used with the internal ID of the room to which they lead. In this chapter, we showed that the value could also be a string (explaining why movement in that direction isn't possible). Here are examples of both, and also of the `cant_go` property which provides just such an explanation for *all* connections that aren't explicitly listed:

```
e_to forest,
in_to "It's such a lovely day -- much too nice to go inside.",
cant_go "The only path lies to the east.",
```

Routines and arguments

The library includes a number of useful routines, available to perform certain common tasks if you require them; there's a list in "Library routines" on page 236. We used the `PlayerTo` routine, which moves the player character from her current room to another one – not necessarily adjacent to the first room.

When calling `PlayerTo`, we had to tell the library which room is the destination. We did this by supplying that room's internal ID within parentheses, thus:

```
PlayerTo(clearing);
```

A value given in parentheses like that is called an **argument** of the routine. In fact, a routine can have more than one argument; if so, they're separated by commas. For example, to move the player character to a room *without* displaying that room's description, we could have supplied a second argument:

```
PlayerTo(clearing,1);
```

In this example, the effect of the `1` is to prevent the description being displayed.

Statements

We encountered several new statements:

```
return true;
return false;
```

We used these at the end of embedded routines to control what the interpreter did next.

```
print "string";
print_ret "string";
```

The `print` statement simply displays the string of characters represented here by *`string`*. The `print_ret` statement also does that, then outputs a newline character, and finally executes a `return true;`

```
move obj_id to parent_obj_id;
```

The `move` statement rearranges the object tree, by making the first *`obj_id`* a child of the *`parent_obj_id`*.

```
if (condition && condition ) ...
```

We extended the simple `if` statement that we met before. The `&&` (to be read as "and") is an operator commonly used when testing for more than one condition at the same time. It means "if this condition is true *and* this condition is also true *and* ..." There's also a `||` operator, to be read as "or".

NOTE: in addition, there are `&` and `|` operators, but they do a rather different job and are much less common. Take care not to get them confused.

Actions

We've talked a lot about intercepting actions like `Listen`, `Enter`, `Climb` and `Drop`. An **action** is a generalised representation of something to be done, determined by the verb which the player types. For example, the verbs HEAR and LISTEN are ways of saying much the same thing, and so both result in the same action: `Listen`. Similarly, verbs like ENTER, GET INTO, SIT ON and WALK INSIDE all lead to an action of `Enter`, CLIMB and SCALE lead to `Climb`, and DISCARD, DROP, PUT DOWN and THROW all lead to `Drop`. This makes life much easier for the designer; although Inform defines quite a lot of actions, there are many fewer than there are ways of expressing those same actions using English verbs.

Each action is represented internally by a number, and the value of the current action is stored in a library variable called, erm, `action`. Two more variables are also useful here: `noun` holds the internal ID of the object which is the focus of the action, and `second` holds the internal ID of the secondary object (if there is one). Here are some examples of these:

Player types	`action`	`noun`	`second`
LISTEN	`Listen`	`nothing`	`nothing`
LISTEN TO THE BIRD	`Listen`	`bird`	`nothing`
PICK UP THE BIRD	`Take`	`bird`	`nothing`
PUT BIRD IN NEST	`Insert`	`bird`	`nest`
DROP THE NEST	`Drop`	`nest`	`nothing`
PUT NEST ON BRANCH	`PutOn`	`nest`	`branch`

The value `nothing` is a built-in constant (like `true` and `false`) which means, well, there isn't any object to refer to. There's a list of standard library actions in "Group 1 actions" on page 242, "Group 2 actions" on page 243 and "Group 3 actions" on page 243.

We've now reached the end of our first game. In these three chapters we've shown you the basic principles on which almost all games are based, and introduced you to many of the components that you'll need when creating more interesting IF. We suggest that you take one last look at the source code (see "Heidi" story on page 189), and then move on to the next stage.

6 • William Tell: a tale is born

K was King William, once governed the land;
L was a lady, who had a white hand.

eeping up the momentum, this chapter (and the three which follow) works steadily through the design of the "William Tell" game that we encountered right at the start of this guide. Many of the principles are the same as the ones we explained when designing Heidi and her forest, so we'll not linger on what should be familiar ground. "William Tell" is a slightly longer and more complex game, so we'll move as swiftly as possible to examine the features which are new.

Initial setup

Our starting point is much the same as last time. Here's a basic `Tell.inf`:

```
!============================================================================
Constant Story "William Tell";
Constant Headline
            "^A simple Inform example
             ^by Roger Firth and Sonja Kesserich.^";
Release 2;  Serial "020827";    ! for keeping track of public releases

Constant MAX_SCORE = 4;

Include "Parser";
Include "VerbLib";

!============================================================================
! Object classes

!============================================================================
! The game objects

!============================================================================
! The player's possessions

!============================================================================
! Entry point routines

[ Initialise;
    location = street;
    lookmode = 2;           ! like the VERBOSE command
    move bow to player;
    move quiver to player; give quiver worn;
    player.description =
        "You wear the traditional clothing of a Swiss mountaineer.";
    print_ret "^^
        The place: Altdorf, in the Swiss canton of Uri. The year is 1307,
        at which time Switzerland is under rule by the Emperor Albert of
        Habsburg. His local governor -- the vogt -- is the bullying
        Hermann Gessler, who has placed his hat atop a wooden pole in
```

```
        the centre of the town square; everybody who passes through the
        square must bow to this hated symbol of imperial might.
        ^^
        You have come from your cottage high in the mountains,
        accompanied by your younger son, to purchase provisions. You are
        a proud and independent man, a hunter and guide, renowned both
        for your skill as an archer and, perhaps unwisely (for his soldiers
        are everywhere), for failing to hide your dislike of the vogt.
        ^^
        It's market-day: the town is packed with people from the
        surrounding villages and settlements.^";
];

!============================================================================
! Standard and extended grammar

Include "Grammar";

!============================================================================
```

You'll see that we've marked a couple of extra divisions in the file, to help organise the stuff we'll add later, but the overall structure is identical to our first game. Let's quickly point out some extra bits and pieces:

- If you look at a game's banner, you'll see two pieces of information: "Release" and "Serial number".

  ```
  William Tell
  A simple Inform example
  by Roger Firth and Sonja Kesserich.
  Release 2 / Serial number 020827 / Inform v6.21 Library 6/10 SD
  ```

  These two fields are automatically written by the compiler, which sets by default Release to 1 and the Serial Number to today's date. However, we can explicitly override this behaviour using `Release` and `Serial`, to keep track of different versions of our game. Typically, we will publish several updates of our games over time, each version fixing problems which were found in the previous release. If somebody else reports a problem with a game, we'd like to know exactly which version they were using; so, rather than take the default values, we set our own. When it's time to release a new version, all we have to do is comment out the previous lines and add another below them:

  ```
  !Release 1; Serial "020128";    ! First beta-test release
  !Release 2; Serial "020217";    ! Second beta-test release
  Release 3;  Serial "020315";    ! IF Library competition entry
  ```

- We'll be implementing a simple system of awarding points when the player gets something right, so we define top marks:

  ```
  Constant MAX_SCORE = 4;
  ```

- The `Initialise` routine that we wrote last time contained only one statement, to set the player's initial `location`. We do that here as well, but we also do some other stuff.

- The first thing is to assign 2 to the library variable `lookmode`. Inform's default mode for displaying room descriptions is BRIEF (a description is displayed only when a room is visited for the first time) and, by changing this variable's value, we set it to VERBOSE (descriptions are displayed on *every* visit). Doing this is largely a matter of personal preference, and in any case it's nothing more than a convenience; it just saves having to remember to type VERBOSE each time that we test the game.

- At the start of the game, we want Wilhelm to be equipped with his bow and quiver of arrows. The recommended way of making this happen is to perform the necessary object tree rearrangement with a couple of `move` statements in the `Initialise` routine:

```
move bow to player;
move quiver to player;
```

  and indeed this is the clearest way to place objects in the player's inventory at the beginning of any game.

  > NOTE: wait! you say. In the previous chapter, to make an object the child of another object all we needed to do was to define the child object with the internal identification of the parent object at the end of the header:
  >
  > ```
  > Object bird "baby bird" forest
  > ```
  >
  > Why not do that with the player? Because the object which represents the player is defined by the library (rather than as part of our game), and actually has an internal ID of `selfobj`; `player` is a variable whose value is that identifier. Rather than worry all about this, it's easier to use the `move` statements.

  There's one other task associated with the quiver; it's an article of clothing which Wilhelm is "wearing", a state denoted by the attribute `worn`. Normally the interpreter would apply this automatically, while handling a command like WEAR QUIVER, but since we've moved the quiver ourselves, we also need to set the quiver's `worn` attribute. The `give` statement does the job:

```
give quiver worn;
```

  (To clear the attribute, by the way, you'd use the statement `give quiver ~worn` – read that as "give the quiver not-worn"; Inform often uses ~ to mean "not".)

- If the player types EXAMINE ME, the interpreter displays the `description` property of the `player` object. The default value is "As good-looking as ever", a bit of a cliché in the world of Inform games. It's easy to change, though, once you realise that, since the properties of an object are variables, you can assign new values to them just as you'd assign new values to `location` and `lookmode`. The only problem is getting the syntax right; you can't say just:

```
description = "You wear the traditional clothing of a Swiss mountaineer.";
```

because there are dozens of objects in the game, each with its own `description` property; you need to be a little more explicit. Here's what to type:

```
player.description =
    "You wear the traditional clothing of a Swiss mountaineer.";
```

- Finally, the `Initialise` routine ends with a lengthy `print_ret` statement. Since the interpreter calls `Initialise` right at the start of the game, that's the point at which this material is displayed, so that it acts as a scene-setting preamble before the game gets under way. In fact, everything you want set or done at the very beginning of the game, should go into the `Initialise` routine.

The game won't compile in this state, because it contains references to objects which we haven't yet defined. In any case, we don't intend to build up the game in layers as we did last time, but rather to talk about it in logically related chunks. To see (and if you wish, to type) the complete source, go to "William Tell" story on page 195.

Object classes

Remember how we defined the rooms in "Heidi"? Our first attempt started like this:

```
Object  "In front of a cottage"
  with  description
            "You stand outside a cottage. The forest stretches east.",
  has   light;

Object  "Deep in the forest"
  with  description
            "Through the dense foliage, you glimpse a building to the west.
             A track heads to the northeast.",
  has   light;
...
```

and we explained that just about *every* room needs that `light` attribute, or else the player would be literally in the dark. It's a bit of a nuisance having to specify that same attribute each time; what would be neater would be to say that *all* rooms are illuminated. So we can write this:

```
Class   Room
  has   light;

Room    "In front of a cottage"
  with  description
            "You stand outside a cottage. The forest stretches east.",
  has   ;

Room    "Deep in the forest"
  with  description
            "Through the dense foliage, you glimpse a building to the west.
             A track heads to the northeast.",
  has   ;
...
```

We've done four things:

1. We've said that some of the objects in our game are going to be defined by the specialised word `Room` rather than the general-purpose word `Object`. In effect, we've taught Inform a new word specially for defining objects, which we can now use as though it had been part of the language all along.
2. We've furthermore said that every object which we define using `Room` is automatically going to have the `light` attribute.
3. We've changed the way in which we define the four room objects, by starting them with our specialised word `Room`. The remainder of the definition for these objects – the header information, the block of properties, the block of attributes and the final semicolon – remains the same; except that:
4. We don't need to explicitly include the `light` attribute each time; every `Room` object has it automatically.

A **class** is a family of closely related objects, all of which behave in the same way. Any properties defined for the class, and any attributes defined for the class, are automatically given to objects which you specify as belonging to that class; this process of acquisition just by being a member of a class is called **inheritance**. In our example, we've defined a `Room` class with a `light` attribute, and then we've specified four objects each of which is a member of that class, and each of which gets given a `light` attribute as a result of that membership.

Why have we gone to this trouble? Three main reasons:

- By moving the common bits of the definitions from the individual objects to the class definition which they share, those object definitions become shorter and simpler. Even if we had a hundred rooms, we'd still need to specify `has light` only once.
- By creating a specialised word to identify our class of objects, we make our source file easier to read. Rather than absolutely everything being an anonymous `Object`, we can now immediately recognise that some are `Room` objects (and others belong to the different classes that we'll create soon).
- By collecting the common definitions into one place, we make it much easier to make widespread modifications in future. If we need to make some change to the definition of all our rooms, we just modify the `Room` class, and all of the class members inherit the change.

For these reasons, the use of classes is an incredibly powerful technique, easier than it may look, and very well worth mastering. From now on, we'll be defining object classes whenever it makes sense (which is generally when two or more objects are meant to behave in exactly the same way).

You may be wondering: suppose I want to define a room which for some reason *doesn't* have `light`; can I still use the `Room` class? Sure you can:

```
Room    cellar "Gloomy cellar"
  with  description "Your torch shows only cobwebby brick walls.",
  has   ~light;
```

This illustrates another nice feature of inheritance: the object definition can override the class definition. The class says `has light`, but the object itself says `has ~light` (read that as "has no light") and the object wins. The cellar is dark, and the player will need a torch to see what's in it.

In fact, for any object both the block of properties and the block of attributes are optional and can be omitted if there's nothing to be specified. Now that the `light` attribute is being provided automatically and there aren't any other attributes to set, the word `has` can be left out. Here's the class again:

TYPE

```
Class   Room
  has   light;
```

and here is how we could have used it in "Heidi":

```
Room    "In front of a cottage"
  with  description
            "You stand outside a cottage. The forest stretches east.";

Room    "Deep in the forest"
  with  description
            "Through the dense foliage, you glimpse a building to the west.
             A track heads to the northeast.";
...
```

You'll notice that, if an object has no block of attributes, the semicolon which terminates its definition simply moves to the end of its last property.

A class for props

We use the `Room` class in "William Tell", and a few other classes besides. Here's a `Prop` class (that's "Prop" in the sense of a theatrical property rather than a supportive device), useful for scenic items whose only role is to sit waiting in the background on the off-chance that the player might think to EXAMINE them:

TYPE

```
Class   Prop
  with  before [;
            Examine: return false;
            default:
                print_ret "You don't need to worry about ", (the) self, ".";
        ],
  has   scenery;
```

You'll see that all objects of this class inherit the `scenery` attribute, which means they're excluded from room descriptions. Of greater interest, there's a `before` property; one that's more complex than our previous efforts. You'll remember that the first `before` we met looked like this:

```
before [; Listen:
    print "It sounds scared and in need of assistance.^";
    return true;
],
```

The role of that original `before` was to intercept `Listen` actions, while leaving all others well alone. The role of the `before` in the `Prop class` is broader: to intercept (a) `Examine` actions, and (b) all the rest. If the action is `Examine`, then the `return false` statement means that the action carries on. If the action is `default` – none of those explicitly listed, which in this instance means *every* action apart from `Examine` – then the `print_ret` statement is executed, after which the interpreter does nothing further. So, a `Prop` object can be EXAMINEd, but any other action addressed to it results in a "no need to worry" message.

That message is also more involved than anything we've so far displayed. The statement which produces it is:

```
print_ret "You don't need to worry about ", (the) self, ".";
```

which you should read as doing this:

1. display the string "You don't need to worry about ",
2. display a definite article (usually "the") followed by a space and the external name of the object concerned,
3. display a period, and
4. display a newline and return true in the usual way for a `print_ret` statement.

The interesting things that this statement demonstrates are:

- The `print` and `print_ret` statements aren't restricted to displaying a single piece of information: they can display a list of items which are separated by commas. The statement still ends with a semicolon in the usual way.
- As well as displaying strings, you can also display the names of objects: given the `nest` object from our first game, `(the) nest` would display "the bird's nest", `(The) nest` would display "The bird's nest", `(a) nest` would display "a bird's nest" and `(name) nest` would display just "bird's nest". This use of a word in parentheses, telling the interpreter how to display the following object's internal ID, is called a **print rule**.
- There's a library variable `self` which always contains the internal ID of the current object, and is really convenient when using a `Class`. By using this variable in our `print_ret` statement, we ensure that the message contains the name of the appropriate object.

Let's see an example of this in action; here's a `Prop` object from "William Tell":

```
Prop    "south gate" street
  with  name 'south' 'southern' 'wooden' 'gate',
        description "The large wooden gate in the town walls is wide open.",
        ...
```

If players type EXAMINE GATE, they'll see "The large wooden gate..."; if they type CLOSE GATE then the gate's `before` property will step in and display "You

don't need to worry about the south gate", neatly picking up the name of the object from the `self` variable.

The reason for doing all this, rather than just creating a simple scenery object like Heidi's `tree` and `cottage`, is to support EXAMINE for increased realism, while clearly hinting to players that trying other verbs would be a waste of time.

A class for furniture

The last class for now – we'll talk about the `Arrow` and `NPC` classes in the next chapter – is for furniture-like objects. If you label an object with the `static` attribute, an attempt to TAKE it results in "That's fixed in place" – acceptable in the case of Heidi's branch object (which is indeed supposed to be part of the tree), less so for items which are simply large and heavy. This `Furniture` class might sometimes be more appropriate:

TYPE

```
Class   Furniture
  with  before [;
            Take,Pull,Push,PushDir:
                print_ret (The) self, " is too heavy for that.";
        ],
  has   static supporter;
```

Its structure is similar to that of our `Prop` class: some appropriate attributes, and a `before` property to trap actions directed at it. Again, we display a message which is "personalised" for the object concerned by using a `(The) self` print rule. This time we're intercepting four actions; we *could* have written the property like this:

```
before [;
    Take: print_ret (The) self, " is too heavy for that.";
    Pull: print_ret (The) self, " is too heavy for that.";
    Push: print_ret (The) self, " is too heavy for that.";
    PushDir: print_ret (The) self, " is too heavy for that.";
],
```

but since we're giving exactly the same response each time, it's better to put all of those actions into one list, separated by commas. `PushDir`, if you were wondering, is the action triggered by a command like PUSH THE TABLE NORTH.

Incidentally, another bonus of defining classes like these is that you can probably reuse them in your next game.

Now that most of our class definitions are in place, we can get on with defining some real rooms and objects. First, though, if you're typing in the "William Tell" game as you read through the guide, you'd probably like to check that what you've entered so far is correct; "Compile-as-you-go" on page 208 explains how to compile the game in its current – incomplete – state.

7 • William Tell: the early years

M was a miser, and hoarded up gold;
N was a nobleman, gallant and bold.

oving along swiftly, we'll define the first two rooms and populate them with assorted townspeople and street furniture, we'll equip Wilhelm with his trusty bow and quiver of arrows, and we'll introduce Helga the friendly stallholder.

Defining the street

This is the `street` room, the location where the game starts:

TYPE

```
Room     street "A street in Altdorf"
  with   description [;
             print "The narrow street runs north towards the town square.
                 Local folk are pouring into the town through the gate to the
                 south, shouting greetings, offering produce for sale,
                 exchanging news, enquiring with exaggerated disbelief about
                 the prices of the goods displayed by merchants whose stalls
                 make progress even more difficult.^";
             if (self hasnt visited)
                 print "^~Stay close to me, son,~ you say,
                     ~or you'll get lost among all these people.~^";
         ],
         n_to below_square,
         s_to
             "The crowd, pressing north towards the square,
              makes that impossible.";
```

We're using our new `Room` class, so there's no need for a `light` attribute. The `n_to` and `s_to` properties, whose values are an internal ID and a string respectively, are techniques we've used before. The only innovation is that the `description` property has an embedded routine as its value.

The first thing in that routine is a `print` statement, displaying details of the street surroundings. If that was all that we wanted to do, we could have supplied those details by making the `description` value a string; that is, these two examples behave identically:

```
description [;
    print "The narrow street runs north towards the town square.
        Local folk are pouring into the town through the gate to the
        south, shouting greetings, offering produce for sale,
        exchanging news, enquiring with exaggerated disbelief about
        the prices of the goods displayed by merchants whose stalls
        make progress even more difficult.^";
],
```

```
description
    "The narrow street runs north towards the town square.
     Local folk are pouring into the town through the gate to the
     south, shouting greetings, offering produce for sale,
     exchanging news, enquiring with exaggerated disbelief about
     the prices of the goods displayed by merchants whose stalls
     make progress even more difficult.",
```

However, that *isn't* all that we want to do. Having presented the basic description, we're going to display that little line of dialogue, where Wilhelm tells his son to be careful. And we want to do that only once, the very first time that the street's description is displayed. If the player types LOOK a few times, or moves north and then returns south to the street, we're happy to see the surroundings described – but we don't want that dialogue again. This is the pair of statements that makes it happen:

```
if (self hasnt visited)
    print "^~Stay close to me, son,~ you say,
        ~or you'll get lost among all these people.~^";
```

The line of dialogue is produced by the `print` statement, the `print` statement is controlled by the `if` statement, and the `if` statement is performing the test `self hasnt visited`. In detail:

- `visited` is an attribute, but not one that you'd normally give to an object yourself. It's automatically applied to a room object by the interpreter, but only after that room has been visited for the first time by the player.
- `hasnt` (and `has`) are available for testing whether a given attribute is currently set for a given object. *`X`* `has` *`Y`* is true if object *`X`* currently has attribute *`Y`*, false if it doesn't. To make the test in reverse, *`X`* `hasnt` *`Y`* is true if object *`X`* currently does not have attribute *`Y`*, false if it does.
- `self`, which we met in the previous chapter, is that useful variable which, within an object, always refers to that object. Since we're using it in the middle of the `street` object, that's what it refers to.

So, putting it all together, `self hasnt visited` is true (and therefore the `print` statement is executed) only while the `street` object has *not* got a `visited` attribute. Because the interpreter automatically gives rooms a `visited` attribute as soon as the player has been there once, this test will be true only for one turn. Therefore, the line of dialogue will be displayed only once: the first time the player visits the street, at the very start of the game.

Although the primary importance of `self` is within class definitions, it can also be convenient to use it simply within an object. Why didn't we just write this?

```
if (street hasnt visited)
    print "^~Stay close to me, son,~ you say,
        ~or you'll get lost among all these people.~^";
```

It's true that the effect is identical, but there are a couple of good reasons for using `self`. One: it's an aid to understanding your code days or weeks after writing it.

If you read the line `if (street hasnt visited)`, you need to think for a moment about which object is being tested; oh, it's this one. When you read `if (self hasnt visited)`, you immediately *know* which object we're talking about.

Another reason is auto-plagiarism. Many times you'll find that a chunk of code is useful in different situations (say, you want to repeat the mechanics of the street description in another room). Rather than writing everything from scratch, you'll typically use copy-and-paste to repeat the routine, and then all you have to do is compose the appropriate descriptive strings for the new room. If you've used `self`, the line `if (self hasnt visited)` is still good; if you've written instead `if (street hasnt visited)`, you'll have to change that as well. Worse, if you *forget* to change it, the game will still work – but not in the way you'd intended, and the resulting bug will be quite difficult to track down.

Adding some props

The street's description mentions various items – the gate, the people, etc. – which ought to exist within the game (albeit only in minimal form) to sustain the illusion of hustle and bustle. Our `Prop` class is ideal for this:

TYPE

```
Prop    "south gate" street
  with  name 'south' 'southern' 'wooden' 'gate',
        description "The large wooden gate in the town walls is wide open.";

Prop    "assorted stalls"
  with  name 'assorted' 'stalls',
        description "Food, clothing, mountain gear; the usual stuff.",
        found_in street below_square,
  has   pluralname;

Prop    "merchants"
  with  name 'merchant' 'merchants' 'trader' 'traders',
        description
            "A few crooks, but mostly decent traders touting their wares
             with raucous overstatement.",
        found_in street below_square,
  has   animate pluralname;

Prop    "local people"
  with  name 'people' 'folk' 'local' 'crowd',
        description "Mountain folk, just like yourself.",
        found_in [; return true; ],
  has   animate pluralname;
```

NOTE: because these objects are not referenced by other objects, we haven't bothered to given them internal *`obj_ids`* (though we could have; it wouldn't make any difference). However, we *have* provided *`external_names`*, because these are used by the `Prop` class's `print_ret ... (the) self` statement.

You'll see a couple of new attributes: `animate` marks an object as being "alive", while `pluralname` specifies that its external name is plural rather than singular. The interpreter uses these attributes to ensure that messages about such objects are grammatical and appropriate (for example, it will now refer to "some merchants"

rather than "a merchants"). Because the library handles so many situations automatically, it's hard to be sure exactly what messages players may trigger; the best approach is to play safe and always give an object the relevant set of attributes, even when, as here, they probably won't be needed.

You'll also see a new `found_in` property, which specifies the rooms where this object is to appear. The stalls, for example, can be EXAMINEd both in the street and below the square, so we *could* have created a `Prop` object in each room:

```
Prop    "assorted stalls" street
  with  name 'assorted' 'stalls',
        description "Food, clothing, mountain gear; the usual stuff.",
  has   pluralname;

Prop    "assorted stalls" below_square
  with  name 'assorted' 'stalls',
        description "Food, clothing, mountain gear; the usual stuff.",
  has   pluralname;
```

but `found_in` does the same job more neatly – there's only one object, but it appears in both the `street` and `below_square` rooms while the player's there.

The local people are even more ubiquitous. In this case the `found_in` value is an embedded routine rather than a list of rooms; such a routine would generally test the value of the current location and `return true` if it wants to be present here, or `false` if not. Since we'd like the local people *always* to be present, in every room, we `return true` without bothering to examine `location`. It's as though we'd written any of these, but simpler and less error prone:

```
Prop    "local people"
  with  name 'people' 'folk' 'local' 'crowd',
        description "Mountain folk, just like yourself.",
        found_in street below_square south_square mid_square north_square
            marketplace,
  has   animate pluralname;

Prop    "local people"
  with  name 'people' 'folk' 'local' 'crowd',
        description "Mountain folk, just like yourself.",
        found_in [;
            if (location == street       || location == below_square ||
                location == south_square || location == mid_square ||
                location == north_square || location == marketplace)
                return true;
            return false;
        ],
  has   animate pluralname;

Prop    "local people"
  with  name 'people' 'folk' 'local' 'crowd',
        description "Mountain folk, just like yourself.",
        found_in [;
            if (location == street or below_square or south_square or
                mid_square or north_square or marketplace) return true;
            return false;
        ],
  has   animate pluralname;
```

In the second example, you'll see the `||` operator, to be read as "or", which we mentioned near the end of "Heidi"; it combines the various `location == some_room` comparisons so that the `if` statement is true if *any* of those individual tests is true. And in the third example we introduce the `or` keyword, which is a more succinct way of achieving exactly the same result.

The player's possessions

Since our `Initialise` routine has already mentioned them, we might as well define Wilhelm's bow and arrows:

TYPE

```
Object  bow "bow"
  with  name 'bow',
        description "Your trusty yew bow, strung with flax.",
        before [;
            Drop,Give: print_ret "You're never without your trusty bow.";
        ],
  has   clothing;

Object  quiver "quiver"
  with  name 'quiver',
        description
            "Made of goatskin, it usually hangs over your left shoulder.",
        before [;
            Drop,Give:
                print_ret "But it was a present from Hedwig, your wife.";
        ],
  has   container open clothing;
```

Both of these are straightforward objects, with the `Drop` and `Give` actions being intercepted to ensure that Wilhelm is never without them. The `clothing` attribute makes its first appearance, marking both the quiver and the bow as capable of being worn (as the result of a WEAR BOW command, for instance); you'll remember that our `Initialise` routine goes on to add a `worn` attribute to the quiver.

An empty quiver is pretty useless, so here's the class used to define Wilhelm's stock of arrows. This class has some unusual features:

TYPE

```
Class   Arrow
  with  name 'arrow' 'arrows//p',
        article "an",
        plural "arrows",
        description "Just like all your other arrows -- sharp and true.",
        before [;
            Drop: print_ret "Much too dangerous to leave lying around.";
        ];
```

The classes we've created so far – `Room`, `Prop` and `Furniture` – are intended for objects which behave the same but are otherwise clearly separate. For example, a table, a bed and a wardrobe would generally have their own individual characteristics – a name, a description, maybe some specialised properties – while still inheriting the general behaviour of `Furniture` objects. The arrows aren't

like this: not only do they behave the same, but also they are indistinguishable one from another. We're trying for this effect:

```
>INVENTORY
You are carrying:
  a quiver (being worn)
    three arrows
  a bow

>
```

where the interpreter lumps together our stock of three arrows, rather than listing them individually in this clumsy fashion:

```
>INVENTORY
You are carrying:
  a quiver (being worn)
    an arrow
    an arrow
    an arrow
  a bow

>
```

The interpreter will do this for us if our objects are "indistinguishable", best achieved by making them members of a class which includes both `name` and `plural` properties. We define the actual arrows very simply, like this:

```
Arrow   "arrow" quiver;
Arrow   "arrow" quiver;
Arrow   "arrow" quiver;
```

and you can see that we provide only two pieces of information for each `Arrow` object: an external name in double quotes ("arrow" in each case) which the interpreter uses when referring to the object, and an initial location (in the quiver). That's all: no block of properties, no set of attributes, and no internal identifier, because we never need to refer to the individual `Arrow` objects within the game.

The `name` property of the class definition has an odd-looking dictionary word:

```
name 'arrow' 'arrows//p',
```

The word `'arrow'` refers to a single arrow. So also would the word `'arrows'`, unless we specifically tell the interpreter that it's a plural reference. That `//p` marks `'arrows'` as being a potential reference to more than one object at once, thus enabling players to type TAKE ARROWS and thereby pick up as many arrows as happened to be available (without it, TAKE ARROWS would have picked up one at random).

There are two other properties not seen previously:

```
article "an",
plural "arrows",
```

The `article` property lets you define the object's indefinite article – usually something like "a", "an" or "some" – instead of letting the library assign one automatically. It's a belt-and-braces (OK, belt-and-suspenders) precaution: because "arrow" starts with a vowel, we need to display "an arrow" not "a arrow". Most interpreters automatically get this right, but just to be on the safe side, we explicitly define the appropriate word. And the `plural` property defines the word to be used when lumping several of these objects together, as in the "three arrows" inventory listing. The interpreter can't just automatically slap an "s" on the end; the plural of "slice of cake", for example, isn't "slice of cakes".

Moving further along the street

As Wilhelm moves north towards the square, he comes to this room:

TYPE

```
Room    below_square "Further along the street"
  with  description
            "People are still pushing and shoving their way from the southern
             gate towards the town square, just a little further north.
             You recognise the owner of a fruit and vegetable stall.",
        n_to south_square,
        s_to street;
```

No surprises there, nor in most of the supporting scenery objects.

TYPE

```
Furniture   stall "fruit and vegetable stall" below_square
  with  name 'fruit' 'veg' 'vegetable' 'stall' 'table',
        description
            "It's really only a small table, with a big heap of potatoes,
             some carrots and turnips, and a few apples.",
        before [; Search: <<Examine self>>; ],
  has   scenery;

Prop    "potatoes" below_square
  with  name 'potato' 'potatoes' 'spuds',
        description
            "Must be a particularly early variety... by some 300 years!",
  has   pluralname;

Prop    "fruit and vegetables" below_square
  with  name 'carrot' 'carrots' 'turnip' 'turnips' 'apples' 'vegetables',
        description "Fine locally grown produce.",
  has   pluralname;
```

The only new thing here is the `before` property of the fruit'n'veg stall. The stall's description – lots of items on a table – may suggest to players that they can SEARCH through the produce, maybe finding a lucky beetroot or something else interesting. No such luck – and we might as well trap the attempt.

Having intercepted a `Search` action, our plan is to respond with the stall's description, as though the player has typed EXAMINE THE STALL. There isn't an easy way for us to stealthily slide those literal words into the interpreter, but we *can* simulate the effect which they'd cause: an action of `Examine` applied to the object `stall`. This rather cryptic statement does the job:

```
<Examine stall>;
```

Having diverted the `Search` action into an `Examine` action, we must tell the interpreter that it doesn't need to do anything else, because we've handled the action ourselves. We've done that before – using `return true` – and so a first stab at the `before` action looks like this:

```
before [; Search: <Examine stall>; return true; ],
```

The two-statement sequence `<...>; return true` is so common that there's a single statement shortcut: `<<...>>`. Also, for exactly the same reason as before, our code is clearer if we use `self` instead of `stall`. So this is how the property finally stands:

```
before [; Search: <<Examine self>>; ],
```

A couple of final observations before we leave this topic. The example here is of an action (`Examine`) applied to an object (`self`, though `stall` or `noun` would also work at this point). You can also use the `<...>` and `<<...>>` statements for actions which affect no objects:

```
<<Look>>;
```

(representing the command LOOK), or which affect two. For example, the command PUT THE BIRD IN THE NEST can be simulated with this statement:

```
<<Insert bird nest>>;
```

Introducing Helga

One of the trickiest aspects of designing a good game is to provide satisfying interaction with other characters. It's hard enough to code inanimate objects which provoke appropriate responses to whatever actions the player character (PC) might attempt. That all gets much worse once those "other objects" are living creatures – non-player characters (**NPC**s) – with, supposedly, minds of their own. A good NPC might move around independently, perform actions with a purpose, initiate conversations, respond to what you say and do (and even to what you *don't* say or do); it can be a real nightmare.

But not here: we've kept our three NPCs – Helga, Walter and the vogt – as simple as possible. Nevertheless, we can establish some fundamental principles; here's the class upon which we base our NPCs:

```
Class   NPC
  with  life [;
            Answer,Ask,Order,Tell:
                print_ret "Just use T[ALK] [TO ", (the) self, "].";
        ],
  has   animate;
```

The most important thing here is the `animate` attribute – that's what defines an object as an NPC, and causes the interpreter to treat it a little differently – for example, TAKE HELGA results in "I don't suppose Helga would care for that".

The animate attribute also brings into play nine extra actions which can be applied only to animate objects: Answer, Ask, Order and Tell are all associated with speech, and Attack, Kiss, Show, ThrowAt and WakeOther are associated with non-verbal interaction. Additionally, a new life property – very similar to before – can be defined to intercept them. Here we use it to trap speech-related commands such as ASK HELGA ABOUT APPLE and TELL WALTER ABOUT BABIES, telling players that in this game we've implemented only a simpler TALK verb (which we describe in "Verbs, verbs, verbs" on page 98).

Based on the NPC class we've created, here's Helga:

TYPE

```
NPC     stallholder "Helga" below_square
  with  name 'stallholder' 'greengrocer' 'monger' 'shopkeeper' 'merchant'
            'owner' 'Helga' 'dress' 'scarf' 'headscarf',
        description
            "Helga is a plump, cheerful woman,
             concealed beneath a shapeless dress and a spotted headscarf.",
        initial [;
            print "Helga pauses from sorting potatoes
                to give you a cheery wave.^";
            if (location hasnt visited) {
                move apple to player;
                print "^~Hello, Wilhelm, it's a fine day for trade! Is this
                    young Walter? My, how he's grown. Here's an apple for him
                    -- tell him to mind that scabby part, but the rest's good
                    enough. How's Frau Tell? Give her my best wishes.~^";
            }
        ],
        times_spoken_to 0,       ! for counting the conversation topics
        life [;
            Kiss: print_ret "~Ooh, you saucy thing!~";
            Talk:
                self.times_spoken_to = self.times_spoken_to + 1;
                switch (self.times_spoken_to) {
                    1:  score = score + 1;
                        print_ret "You warmly thank Helga for the apple.";
                    2:  score = score + 1;
                        print_ret "~See you again soon.~";
                    default: return false;
                }
        ],
  has   female proper;
```

The new attributes are female – because we want the interpreter to refer to Helga with the appropriate pronouns – and proper. The latter signifies that this object's external name is a proper noun, and so references to it should not be preceded by "a" or "the": you wouldn't want to display "You can see a Helga here" or "I don't suppose the Helga would care for that". You may notice the library variable score being incremented. This variable holds the number of points that the player has scored; when it changes like this, the interpreter tells the player that "Your score has just gone up by one point".

There are also life and times_spoken_to properties (which we'll talk about in "William Tell: the end is nigh" on page 91) and an initial property.

`initial` is used when the interpreter is describing a room and listing the objects you can see there. If we *didn't* define it, you'd get this:

```
Further along the street
People are still pushing and shoving their way from the southern gate towards
the town square, just a little further north. You recognise the owner of a fruit
and vegetable stall.

You can see Helga here.

>
```

but we want to introduce Helga in a more interactive manner, and that's what the `initial` property is for: it replaces the standard "You can see *object* here" with a tailored message of your own design. The value of an `initial` property can be either a string which is to be displayed or, as here, an embedded routine. This one is pretty similar to the `description` property that we defined for the street: something that's *always* printed (Helga pauses...) and something that's printed only on the first occasion ("Hello, Wilhelm, it's a fine day... "):

```
Further along the street
People are still pushing and shoving their way from the southern gate towards
the town square, just a little further north. You recognise the owner of a fruit
and vegetable stall.

Helga pauses from sorting potatoes to give you a cheery wave.

"Hello, Wilhelm, it's a fine day for trade! Is this young Walter? My, how he's
grown. Here's an apple for him -- tell him to mind that scabby part, but the
rest's good enough. How's Frau Tell? Give her my best wishes."

>
```

But it's not quite the same as the street's description routine. First, we need a slightly different `if` test: `self hasnt visited` works fine for a room object, but this routine is part of an object *in* a room; instead we could use either `below_square hasnt visited` or (better) `location hasnt visited` – since `location` is the library variable that refers to the room where the player currently is. And second, some curly braces `{...}` have appeared: why?

On Wilhelm's first visit to this room, we need to do two things:

- ensure that Wilhelm is in possession of an apple, because that's mentioned when we...
- display Helga's cheery greeting.

The `move` statement does the first of those, and the `print` statement does the second. And both statements need to be controlled by the `if` statement.

So far, we've used an `if` statement twice, in both cases to control a single following statement.

```
if (nest in branch) deadflag = 2;

if (self hasnt visited)
    print "^~Stay close to me, son,~ you say,
        ~or you'll get lost among all these people.~^";
```

That's what an `if` does – it controls whether the following statement is executed or not. So how can we control two statements at once? Well, we *could* write two `if` statements:

```
if (location hasnt visited)
    move apple to player;
if (location hasnt visited)
    print "^~Hello, Wilhelm, it's a fine day for trade! Is this
        young Walter? My, how he's grown. Here's an apple for him
        -- tell him to mind that scabby part, but the rest's good
        enough. How's Frau Tell? Give her my best wishes.~^";
```

but that's unbearably clumsy; instead, we use the braces to group the `move` and `print` statement into a **statement block** (sometimes known as a code block) which counts as a single statement for the purposes of control by the `if` statement.

```
if (location hasnt visited) {
    move apple to player;
    print "^~Hello, Wilhelm, it's a fine day for trade! Is this
        young Walter? My, how he's grown. Here's an apple for him
        -- tell him to mind that scabby part, but the rest's good
        enough. How's Frau Tell? Give her my best wishes.~^";
}
```

A statement block can contain one, two, ten, a hundred statements; it doesn't matter – they're all treated as one unit by `if` (and by `objectloop`, which we meet later, and by `do`, `for` and `while`, all of them loop statements that we don't encounter in this guide).

> NOTE: the exact positioning of the braces is a matter of personal choice. We use this style:

```
if (condition) {
    statement;
    statement;
    ...
}
```

> but other designers have their own preferences, including:

```
if (condition) {
    statement;
    statement;
    ...
    }
```

```
if (condition)
{   statement;
    statement;
    ...
}

if (condition)
    {
    statement;
    statement;
    ...
    }
```

Although we've not yet needed to use it, now would probably be a good time to mention the `else` extension to the `if` statement. Sometimes we want to execute one statement block if a certain condition is true, and a different statement block if it's not true. Again, we *could* write two `if` statements:

```
if (location has visited) {
    statement;
    statement;
    ...
}
if (location hasnt visited) {
    statement;
    statement;
    ...
};
```

but that's hardly an elegant approach; an `else` clause does the job more neatly:

```
if (location has visited) {
    statement;
    statement;
    ...
}
else {
    statement;
    statement;
    ...
};
```

We've done a lot of scene-setting, but the real action is still to come. Next, it's time to define the town square, and create a confrontation between Wilhelm and the vogt's soldiers. (But first, see again "Compile-as-you-go" on page 208 if you're typing in the game as you read through the guide.)

8 • William Tell: in his prime

O was an oyster girl, and went about town;
P was a parson, and wore a black gown.

ur game's action nears its climax in the town's central square. In this chapter we define the square's constituent rooms and deal with Wilhelm's approach to the hat on the pole – does he salute it, or does he remain proudly defiant?

The south side of the square

The town square, notionally one enormous open space, is represented by three rooms. Here's the south side:

```
Room    south_square "South side of the square"
  with  description
            "The narrow street to the south has opened onto the town square,
             and resumes at the far side of this cobbled meeting place.
             To continue along the street towards your destination --
             Johansson's tannery -- you must walk north across the square,
             in the middle of which you see Gessler's hat set on that
             loathsome pole. If you go on, there's no way you can avoid
             passing it. Imperial soldiers jostle rudely through the throng,
             pushing, kicking and swearing loudly.",
        n_to mid_square,
        s_to below_square;

Prop    "pole"
  with  name 'wooden' 'pole',
        description "You're too far away to see any detail.",
        found_in south_square north_square;

Prop    "hat"
  with  name 'hat',
        description "You're too far away to see any detail.",
        found_in south_square north_square;

Prop    "Gessler's soldiers"
  with  name 'soldier' 'soldiers',
        description "They're uncouth, violent men, not from around here.",
        before [;
            FireAt: print_ret "You're outnumbered many times.";
            Talk:   print_ret "Such scum are beneath your contempt.";
        ],
        found_in south_square mid_square north_square marketplace,
  has   animate pluralname proper;
```

It's all pretty standard stuff: just a `Room` and three `Props`. The "real" pole object is located in the `mid_square` room, which means that players can't EXAMINE it from this room (technically, it's "not in scope"). However, since we're pretending that Wilhelm can see the whole of the square from where he's standing, we need to

provide dummy pole and hat objects, `found_in` both this room and the north side of the square, even if they're "too far away" for a detailed description.

The obnoxious soldiers are also implemented very sketchily; they need to be there, but they don't do much. Their most interesting characteristic is probably that they trap two actions – `FireAt` and `Talk` – which are *not* part of the library, but instead new actions that we've defined specially for this game. We'll talk about those actions in "Verbs, verbs, verbs" on page 98, at which time the role of this `before` property will make more sense.

The middle of the square

The activities here are pivotal to the game's plot. Wilhelm has arrived from the south side of the square, and now encounters the pole with the hat on top. He can do three things:

1. Return south. That's allowed, but all it does is waste a little time – there's nothing else to usefully do south of here.
2. Salute the pole, and then proceed to the north. That's allowed, though it rather subverts the folk story.
3. Attempt to proceed northwards without saluting the pole. Twice, a soldier will prevent this, and issue a verbal warning. On the third attempt, patience runs out, and Wilhelm is hauled off to perform his party piece.

So, there are two actions that we need to look out for: `Salute` (trapped by the pole), and `Go` (which can be trapped by the room itself). `Go` is a standard library action. `Salute` is one that we've devised; let's deal with it first. Here's a first cut of the room:

```
Room    mid_square "Middle of the square"
  with  description
            "There is less of a crush in the middle of the square; most
             people prefer to keep as far away as possible from the pole
             which towers here, topped with that absurd ceremonial hat. A
             group of soldiers stands nearby, watching everyone who passes.",
        n_to north_square,
        s_to south_square;
```

and the pole:

TYPE

```
Furniture   pole "wooden pole" mid_square
  with  name 'wooden' 'pole' 'pine' 'hat' 'black' 'red' 'brim' 'feathers',
        description
            "The pole, the trunk of a small pine some few inches in diameter,
             stands about nine or ten feet high. Set carefully on top is
             Gessler's ludicrous black and red leather hat, with a widely
             curving brim and a cluster of dyed goose feathers.",
        has_been_saluted false,
        before [;
            Salute:
                self.has_been_saluted = true;
                print_ret "You salute the hat on the pole. ^^
                    ~Why, thank you, sir,~ sneers the soldier.";
        ],
  has   scenery;
```

The room will need some more work in a minute, but the pole object is complete (note that we've simplified matters slightly by making one object represent both the pole and the hat which it supports). It mentions a property which we've not met before: `has_been_saluted`. What a remarkable coincidence: the library provides a property with a name that's exactly right for our game; surely not?

No, of course not. `has_been_saluted` isn't a standard library property; it's one that we've just invented. Notice how easily we did it – we simply included the line:

```
has_been_saluted false,
```

in the object definition and voilà, we've added our own home-made property, and initialised it to `false`. To switch the state of the property, we can simply write:

```
pole.has_been_saluted = true;
...
pole.has_been_saluted = false;
```

or just (within the `pole` object):

```
self.has_been_saluted = true;
...
self.has_been_saluted = false;
```

We could also test, if necessary, how the property currently fares:

```
if (pole.has_been_saluted == true) ...
```

Notice that we use `==` (that's two equals signs) to test for "is equal to"; don't confuse this usage with `=` (a single equals sign) which assigns a value to a variable. Compare these examples:

Correct	**Incorrect**
`score = 10;` assigns the value 10 to `score`.	`score == 10;` does nothing; `score` is unchanged.
`if (score == 10) ...` executes the next statement only if the value of `score` is 10.	`if (score = 10) ...` assigns 10 to `score`, then *always* executes the next statement – because `score = 10` evaluates to 10, which is treated as `true`, so the test is *always* true.

Defining a new property variable which, instead of applying to every object in the game (as do the standard library properties), is specific only to a class of objects or even – as here – to a single object, is a common and powerful technique. In this game, we need a `true/false` variable to show whether Wilhelm has saluted the pole or not: the clearest way is to create one as part of the pole. So, when the pole object traps the `Salute` action, we do two things: use a `self.has_been_saluted = true` statement to record the fact, and then use a `print_ret` statement to tell players that the salute was "gratefully" received.

> NOTE: creating new property variables like this – at the drop of a hat, as it were – is the recommended approach, but it isn't the only possibility. We briefly mention some alternative approaches in "Reading other people's code" on page 159.

Back to the `mid_square` room. We've said that we need to detect Wilhelm trying to leave this room, which we can do by trapping the `Go` action in a `before` property. Let's sketch the coding we'll need:

```
before [; Go:
    if (noun == s_obj)        { Wilhelm is trying to move south }
    if (noun == n_obj)        { Wilhelm is trying to move north }
];
```

We can easily trap the `Go` action, but which direction is he moving? Well, it turns out that the interpreter turns a command of GO SOUTH (or just SOUTH) into an action of `Go` applied to an object `s_obj`. This object is defined by the library; so why isn't it called just "`south`"? Well, because we already have another kind of south, the property `s_to` used to say what lies in a southerly direction when defining a room. To avoid confusing them, `s_to` means "south to" and `s_obj` means "south when the player types it as the object of a verb".

The identity of the object which is the target of the current action is stored in the `noun` variable, so we can write the statement `if (noun == s_obj)` to test whether the contents of the `noun` variable are equal to the ID of the `s_obj` object – and, if so, Wilhelm is trying to move south. Another similar statement tests whether he's trying to move north, and that's all that we're interested in; we can let other movements take care of themselves.

The words *`Wilhelm is trying to move south`* aren't part of our game; they're just a temporary reminder that, if we need to execute any statements in this situation, here's the place to put them. Actually, that's the simpler case; it's when *`Wilhelm is trying to move north`* that the fun starts. We need to behave in one of two ways, depending on whether or not he's saluted the pole. But we *know* when he's done that; the pole's `has_been_saluted` property tells us. So we can expand our sketch like this:

```
before [; Go:
    if (noun == s_obj)       { Wilhelm is trying to move south [1] }
    if (noun == n_obj)       { Wilhelm is trying to move north...
        if (pole.has_been_saluted == true)
                             { ...and he's saluted the pole [2] }
        else                 { ...but he hasn't saluted the pole [3] }
    }
];
```

Here we have one `if` statement nested inside another. And there's more: the inner `if` has an `else` clause, meaning that we can execute one statement block when the test `if (pole.has_been_saluted == true)` is true, and an alternative block when the test isn't true. Read that again carefully, checking how the braces `{...}` pair up; it's quite complex, and you need to understand what's going on. One important point to remember is that, unless you insert braces to change this, an `else` clause always pairs with the most recent `if`. Compare these two examples:

```
if (condition1) {
    if (condition2) { here when condition1 is true and condition2 is true }
    else            { here when condition1 is true and condition2 is false }
}

if (condition1) {
    if (condition2) { here when condition1 is true and condition2 is true }
}
else                { here when condition1 is false }
```

In the first example, the `else` pairs with the most recent `if (condition2)`, whereas in the second example the revised positioning of the braces causes the `else` to pair with the earlier `if (condition1)`.

> NOTE: we've used indentation as a visual guide to how the `if` and `else` are related. Be careful, though; the compiler matches an `else` to its `if` purely on the basis of logical grouping, regardless of how you've laid out the code.

Back to the `before` property. You should be able to see that the cases marked `[1]`, `[2]` and `[3]` correspond to the three possible courses of action we listed at the start of this section. Let's write the code for those, one at a time.

Case 1: Returning south

First, *`Wilhelm is trying to move south`*; not very much to this:

```
warnings_count 0,       ! for counting the soldier's warnings
before [; Go:
    if (noun == s_obj) {
        self.warnings_count = 0;
        pole.has_been_saluted = false;
    }
    if (noun == n_obj) {
        if (pole.has_been_saluted == true)
                  { moving north...and he's saluted the pole }
        else      { moving north...but he hasn't saluted the pole }
    }
];
```

Wilhelm might wander into the middle of the square, take one look at the pole and promptly return south. Or, he might make one or two (but not three) attempts to move north first, and then head south. *Or*, he might be really perverse, salute the pole and only then head south. In all of these cases, we take him back to square one, as though he'd received no soldier's warnings (irrespective of how many he'd actually had) and as though the pole had not been saluted (irrespective of whether it was or not). In effect, we're pretending that the soldier has such a short memory, he'll completely forget Wilhelm if our hero should move away from the pole.

To do all this, we've added a new property and two statements. The property is `warnings_count`, and its value will count how many times Wilhelm has tried to go north without saluting the pole: 0 initially, 1 after his first warning, 2 after his second warning, 3 when the soldier's patience finally runs out. The property `warnings_count` isn't a standard library property; like the pole's `has_been_saluted` property, it's one that we've created to meet a specific need.

Our first statement is `self.warnings_count = 0`, which resets the value of the `warnings_count` property of the current object – the `mid_square` room – to 0. The second statement is `pole.has_been_saluted = false`, which signifies that the pole has not be saluted. That's it: the soldier's memory is erased, and Wilhelm's actions are forgotten.

Case 2: Moving north after saluting

Wilhelm is *moving north...and he's saluted the pole*; another easy one:

```
warnings_count 0,        ! for counting the soldier's warnings
before [; Go:
    if (noun == s_obj) {
        self.warnings_count = 0;
        pole.has_been_saluted = false;
    }
    if (noun == n_obj) {
        if (pole.has_been_saluted == true) {
            print "^~Be sure to have a nice day.~^";
            return false;
        }
        else                  { moving north...but he hasn't saluted the pole }
    }
];
```

All that we need do is print a sarcastic goodbye from the soldier, and then `return false`. You'll remember that doing so tells the interpreter to continue handling the action, which in this case is an attempt to move north. Since this is a permitted connection, Wilhelm thus ends up in the `north_square` room, defined shortly.

Case 3: Moving north before saluting

So that just leaves the final case: `moving north...but he hasn't saluted the pole`. This one has more to it than the others, because we need the "three strikes and you're out" coding. Let's sketch a little more:

```
warnings_count 0,       ! for counting the soldier's warnings
before [; Go:
    if (noun == s_obj) {
        self.warnings_count = 0;
        pole.has_been_saluted = false;
    }
    if (noun == n_obj) {
        if (pole.has_been_saluted == true) {
            print "^~Be sure to have a nice day.~^";
            return false;
        }
        else {
            self.warnings_count = self.warnings_count + 1;
            switch (self.warnings_count) {
                1:       First attempt at moving north
                2:       Second attempt at moving north
                default: Final attempt at moving north
            }
        }
    }
];
```

First of all, we need to count how many times he's tried to move north. `self.warnings_count` is the variable containing his current tally, so we add 1 to whatever value it contains: `self.warnings_count = self.warnings_count + 1`. Then, determined by the value of the variable, we must decide what action to take: first attempt, second attempt, or final confrontation. We *could* have used three separate `if` statements:

```
if (self.warnings_count == 1)     { First attempt at moving north }
if (self.warnings_count == 2)     { Second attempt at moving north }
if (self.warnings_count == 3)     { Final attempt at moving north }
```

or a couple of nested `if` statements:

```
if (self.warnings_count == 1)     { First attempt at moving north }
else {
    if (self.warnings_count == 2) { Second attempt at moving north }
    else                          { Final attempt at moving north }
}
```

but for a series of tests all involving the same variable, a `switch` statement is usually a clearer way of achieving the same effect. The generic syntax for a `switch` statement is:

```
switch (expression) {
    value1: whatever happens when the expression evaluates to value1
    value2: whatever happens when the expression evaluates to value2
    ...
    valueN: whatever happens when the expression evaluates to valueN
    default: whatever happens when the expression evaluates to something else
}
```

This means that, according to the current value of an expression, we can get different outcomes. Remember that the `expression` may be a `Global` or local variable, an object's property, one of the variables defined in the library, or any other expression capable of having more than one value. You could write `switch (x)` if `x` is a defined variable, or even, for instance, `switch (x+y)` if both `x` and `y` are defined variables. Those `whatever happens when...` are collections of statements which implement the desired effect for a particular value of the switched variable.

Although a `switch` statement `switch (expression) { ... }` needs that one pair of braces, it doesn't need braces around each of the individual "cases", no matter how many statements each of them includes. As it happens, case 1 and case 2 contain only a single `print_ret` statement each, so we'll move swiftly past them to the third, more interesting, case – when `self.warnings_count` is 3. Again, we *could* have written this:

```
switch (self.warnings_count) {
    1:   First attempt at moving north
    2:   Second attempt at moving north
    3:   Final attempt at moving north
}
```

but using the word `default` – meaning "any value not already catered for" – is better design practice; it's less likely to produce misleading results if for some unforeseen reason the value of `self.warnings_count` isn't the 1, 2 or 3 you'd anticipated. Here's the remainder of the code (with some of the printed text omitted):

```
self.warnings_count = self.warnings_count + 1;
switch (self.warnings_count) {
    1: print_ret "...";
    2: print_ret "...";
    default:
        print "^~OK, ";
        style underline; print "Herr"; style roman;
        print " Tell, now you're in real trouble. I asked you
            ...
            old lime tree growing in the marketplace.^";
        move apple to son;
        PlayerTo(marketplace);
        return true;
} ;
```

The first part is really just displaying a lot of text, made slightly messier because we're adding emphasis to the word "Herr" by using underlining (which actually comes out as *italic type* on most interpreters). Then, we make sure that Walter has the apple (just in case we didn't give it to him earlier in the game), relocate to the final room using `PlayerTo(marketplace)`, and finally `return true` to tell the interpreter that we've handled this part of the `Go` action ourselves.

And so, at long last, here's the complete code for the `mid_square`, the most complicated object in the whole game:

```
Room    mid_square "Middle of the square"
  with  description
            "There is less of a crush in the middle of the square; most
             people prefer to keep as far away as possible from the pole
             which towers here, topped with that absurd ceremonial hat. A
             group of soldiers stands nearby, watching everyone who passes.",
        n_to north_square,
        s_to south_square,
        warnings_count 0,       ! for counting the soldier's warnings
        before [; Go:
            if (noun == s_obj) {
                self.warnings_count = 0;
                pole.has_been_saluted = false;
            }
            if (noun == n_obj) {
                if (pole.has_been_saluted == true) {
                    print "^~Be sure to have a nice day.~^";
                    return false;
                }   ! end of (pole has_been_saluted)
                else {
                    self.warnings_count = self.warnings_count + 1;
                    switch (self.warnings_count) {
                    1:  print_ret "A soldier bars your way. ^^
                            ~Oi, you, lofty; forgot yer manners, didn't you?
                            How's about a nice salute for the vogt's hat?~";
                    2:  print_ret "^~I know you, Tell, yer a troublemaker,
                            ain't you? Well, we don't want no bovver here,
                            so just be a good boy and salute the friggin'
                            hat. Do it now: I ain't gonna ask you again...~";
                    default:
                        print "^~OK, ";
                        style underline; print "Herr"; style roman;
                        print " Tell, now you're in real trouble. I asked you
                            nice, but you was too proud and too stupid. I
                            think it's time that the vogt had a little word
                            with you.~
                            ^^
                            And with that the soldiers seize you and Walter
                            and, while the sergeant hurries off to fetch
                            Gessler, the rest drag you roughly towards the
                            old lime tree growing in the marketplace.^";
                        move apple to son;
                        PlayerTo(marketplace);
                        return true;
                }   ! end of switch
                }   ! end of (pole has_NOT_been_saluted)
            }   ! end of (noun == n_obj)
        ];
```

The north side of the square

The only way to get here is by saluting the pole and then moving north; not very likely, but good game design is about predicting the unpredictable.

TYPE

```
Room    north_square "North side of the square"
  with  description
            "A narrow street leads north from the cobbled square. In its
             centre, a little way south, you catch a last glimpse of the pole
             and hat.",
        n_to [;
            deadflag = 3;
            print_ret "With Walter at your side, you leave the square by the
                north street, heading for Johansson's tannery.";
        ],
        s_to "You hardly feel like going through all that again.";
```

There's one new feature in this room: the value of the `n_to` property is a routine, which the interpreter runs when Wilhelm tries to exit the square northwards. All that the routine does is set the value of the library variable `deadflag` to 3, print a confirmation message, and `return true`, thus ending the action.

At this point, the interpreter notices that `deadflag` is no longer zero, and terminates the game. In fact, the interpreter checks `deadflag` at the end of *every* turn; these are the values that it's expecting to find:

- 0 – this is the normal state; the game continues.
- 1 – the game is over. The interpreter displays "You have died".
- 2 – the game is over. The interpreter displays "You have won".
- any other value – the game is over, but there aren't any appropriate messages built into the library. Instead, the interpreter looks for an **entry point** routine called `DeathMessage` – which we must provide – where we can define our own tailored "end messages".

In this game, we never set `deadflag` to 1, but we do use values of 2 and 3. So we'd better define a `DeathMessage` routine to tell players what they've done:

TYPE

```
[ DeathMessage; print "You have screwed up a favourite folk story"; ];
```

Our game has only one customised ending, so the simple `DeathMessage` routine we've written is sufficient for our purposes. Were you to conceive multiple endings for a game, you could specify suitable messages by checking for the current value of the `deadflag` variable:

```
[ DeathMessage;
    if (deadflag == 3) print "You leave Scarlett O'Hara for good";
    if (deadflag == 4) print "You crush Scarlett with a passionate embrace";
    if (deadflag == 5) print "You've managed to divorce Scarlett";
    ...
];
```

Of course, you must assign the appropriate value to `deadflag` at the point when the game arrives at each of those possible endings.

We've nearly finished. In the concluding chapter of this game, we'll talk about the fateful shooting of the arrow.

9 • William Tell: the end is nigh

Q was a queen, who wore a silk slip;
R was a robber, and wanted a whip.

uite a few objects still remain undefined, so we'll talk about them first. Then, we'll explain how to make additions to Inform's standard repertoire of verbs, and how to define the actions which those verbs trigger.

The marketplace

The marketplace room is unremarkable, and the tree growing there has only one feature of interest:

```
Room    marketplace "Marketplace near the square"
  with  description
            "Altdorf's marketplace, close by the town square, has been hastily
             cleared of stalls. A troop of soldiers has pushed back the crowd
             to leave a clear space in front of the lime tree, which has been
             growing here for as long as anybody can remember. Usually it
             provides shade for the old men of the town, who gather below to
             gossip, watch the girls, and play cards. Today, though, it
             stands alone... apart, that is, from Walter, who has been lashed
             to the trunk. About forty yards away, you are restrained by two
             of the vogt's men.",
        cant_go "What? And leave your son tied up here?";

Object  tree "lime tree" marketplace
  with  name 'lime' 'tree',
        description "It's just a large tree.",
        before [; FireAt:
            if (BowOrArrow(second)) {
                deadflag = 3;
                print_ret "Your hand shakes a little, and your arrow flies
                    high, hitting the trunk a few inches above Walter's
                    head.";
            }
            return true;
        ],
  has   scenery;
```

The tree's before property is intercepting a FireAt action, which we'll define in a few moments. This action is the result of a command like SHOOT AT TREE WITH BOW – we could simulate it with the statement <<FireAt tree bow>> – and it needs extra care to ensure that the second object is a feasible weapon. To deal with silly commands like SHOOT AT TREE WITH HELGA, we must test that second is the bow, one of the arrows, or nothing (from just SHOOT AT TREE). Since this is quite a complex test, and one that we'll be making in several places, it's sensible to write a routine to do the job. Which we'll do shortly – but first, a general introduction to working with routines.

A **standalone routine**, like the familiar routines embedded as the value of a property such as `before` or `each_turn`, is simply a collection of statements to be executed. The major differences are in content, in timing, and in the default return value:

- Whereas an embedded routine has to contain statements which do something appropriate for that associated property variable, a standalone routine can contain statements which do anything you wish. You have total freedom as to what the routine actually does and what value it returns.
- An embedded routine is called when the interpreter is dealing with that property of that object; you provide the routine, but you don't directly control when it's called. A standalone routine, however, is completely under your control; it runs only when you explicitly call it.
- If an embedded routine executes all of its statements and reaches the final `];` without encountering some form of `return` statement, it returns the value `false`. In the same circumstances, a standalone routine returns the value `true`. There's a good reason for this difference – it turns out to be the natural default behaviour more often than not – but it can sometimes baffle newcomers. To avoid confusion, we've always included explicit `return` statements in our routines.

What this generally boils down to is: *if* you have a collection of statements which perform some specific task *and* you need to execute those same statements in more than one place in your game, *then* it often makes sense to turn those statements into a standalone routine. The advantages are: you write the statements only once, so any subsequent changes are easier to make; also, your game becomes simpler and easier to read. We'll look at some simple examples.

At the start of the previous chapter, we defined these objects:

```
Prop    "pole"
  with  name 'wooden' 'pole',
        description "You're too far away to see any detail.",
        found_in south_square north_square;

Prop    "hat"
  with  name 'hat',
        description "You're too far away to see any detail.",
        found_in south_square north_square;
```

The `descriptions` are identical: perhaps we could display them using a routine?

```
[ TooFarAway; print_ret "You're too far away to see any detail."; ];

Prop    "pole"
  with  name 'wooden' 'pole',
        description [; TooFarAway(); ],
        found_in south_square north_square;
```

```
Prop    "hat"
  with  name 'hat',
        description [; TooFarAway(); ],
        found_in south_square north_square;
```

This isn't a very realistic approach – there are more elegant ways of avoiding typing the same string twice – but it works, and it illustrates how we can define a routine to do something useful, and then call it wherever we need to.

Here's another simple example showing how, by returning a value, a routine can report back to the piece of code which called it. We've once or twice used the test `if (self has visited) ...`; we could create a routine which performs that same check and then returns `true` or `false` to indicate what it discovered:

```
[ BeenHereBefore;
    if (self has visited) return true;
    else                  return false;
];
```

Then, we'd rewrite our test as `if (BeenHereBefore() == true) ...`; no shorter or quicker, but maybe more descriptive of what's going on.

One more example of using routines. As well as testing `if (self has visited) ...` we've also tested `if (location has visited) ...` a few times, so we *could* write another routine to perform that check:

```
[ BeenThereBefore;
    if (location has visited) return true;
    else                      return false;
];
```

However, the two routines are very similar; the only difference is the name of the variable – `self` or `location` – which is being checked. A better approach might be to rework our `BeenHereBefore` routine so that it does both jobs, but we somehow need to tell it which variable's value is to be checked. That's easy: we design the routine so that it expects an **argument**:

```
[ BeenToBefore this_room;
    if (this_room has visited) return true;
    else                       return false;
];
```

Notice that the argument's name is one that we've invented to be descriptive of its content; it doesn't matter if we define it as "`x`", "`this_room`" or "`hubba_hubba`". Whatever its name, the argument acts as a placeholder for a value (here, one of the variables `self` or `location`) which we must supply when calling the routine:

```
if (BeenToBefore(self) == true) ...

if (BeenToBefore(location) == true) ...
```

In the first line, we supply `self` as the routine's argument. The routine doesn't care where the argument came from; it just sees a value which it knows as `this_room`, and which it then uses to test for the `visited` attribute. On the second line we supply `location` as the argument, but the routine just sees another value

in its `this_room` variable. `this_room` is called a **local variable** of the `BeenToBefore` routine, one that must be set to a suitable value each time that the routine is called. In this example routine, the value needs to be a room object; we could also check an explicit named room:

```
if (BeenToBefore(mid_square) == true) ...
```

Remember that all routines return *something* sooner or later, either because you explicitly write a `return`, `rtrue` or `rfalse` statement, or because execution reaches the `]` marking the routine's end (in which case the default STEF rule applies: Standalone routines return True, Embedded routines return False). We gave this example of an embedded routine in "Adding some props" on page 71. The `return false` statement is redundant: we could remove it without affecting the routine's behaviour:

```
found_in [;
    if (location == street or below_square or south_square or
        mid_square or north_square or marketplace) return true;
    return false;
],
```

After all that introduction, finally back to the `FireAt` action. We want to check on the characteristics of an object, possibly then displaying a message. We don't know exactly *which* object is to be checked, so we need to write our routine in a generalised way, capable of checking *any* object which we choose; that is, we'll supply the object to be checked as an argument. Here's the routine:

TYPE

```
[ BowOrArrow o;
    if (o == bow or nothing || o ofclass Arrow) return true;
    print "That's an unlikely weapon, isn't it?^";
    return false;
];
```

The routine is designed to inspect any object which is passed to it as its argument `o`; that is, we could call the routine like this:

```
BowOrArrow(stallholder)
BowOrArrow(tree)
BowOrArrow(bow)
```

Given the `bow` object, or any object which we defined as class `Arrow`, it will silently `return true` to signify agreement that this object can be fired. However, given an object like Helga, the apple or the tree, it will print a message and `return false` to signify that this object is not a suitable weapon. The test that we make is:

```
if (o == bow or nothing || o ofclass Arrow) ...
```

which is merely a slightly shorter way of saying this:

```
if (o == bow || o == nothing || o ofclass Arrow) ...
```

The result is that we ask three questions: Is `o` the `bow` object? *Or* is it `nothing`? *Or*, using the `ofclass` test, is it any object which is a member of the `Arrow` class?

What this means is that the value returned by the call `BowOrArrow(bow)` is `true`, while the value returned by the call `BowOrArrow(tree)` is `false`. Or, more generally, the value returned by the call `BowOrArrow(second)` will be either `true` or `false`, depending on the characteristics of the object defined by the value of the variable `second`. So, we can write this set of statements in an object's `before` property:

```
if (BowOrArrow(second)) {
    This object deals with having an arrow fired at it
}
return true;
```

and the effect is either

- `second` is a weapon: `BowOrArrow` displays nothing and returns a value of `true`, the `if` statement reacts to that value and executes the following statements to produce an appropriate response to the fast-approaching arrow; or
- `second` *isn't* a weapon: `BowOrArrow` displays a standard "don't be silly" message and returns a value of `false`, the `if` statement reacts to that value and ignores the following statements. Then
- in both cases, the `return true` statement terminates the object's interception of the `FireAt` action.

That bit was rather complex, but the rest of the `FireAt` action is straightforward. Once the tree has determined that it's being shot at by something sensible, it can just set `deadflag` to 3 – the "You have screwed up" ending, display a message, and be done.

Gessler the governor

There's nothing in Gessler's definition that we haven't already encountered:

TYPE

```
NPC     governor "governor" marketplace
  with  name 'governor' 'vogt' 'Hermann' 'Gessler',
        description
            "Short, stout but with a thin, mean face, Gessler relishes the
             power he holds over the local community.",
        initial [;
            print "Gessler is watching from a safe distance,
                a sneer on his face.^";
            if (location hasnt visited)
                print_ret "^~It appears that you need to be taught a lesson,
                    fool. Nobody shall pass through the square without paying
                    homage to His Imperial Highness Albert; nobody, hear me?
                    I could have you beheaded for treason, but I'm going to
                    be lenient. If you should be so foolish again, you can
                    expect no mercy, but this time, I'll let you go free...
                    just as soon as you demonstrate your archery skills by
                    hitting this apple from where you stand. That shouldn't
                    prove too difficult; here, sergeant, catch. Balance it on
                    the little bastard's head.~";
        ],
```

```
          life [;
              Talk: print_ret "You cannot bring yourself to speak to him.";
          ],
          before [; FireAt:
              if (BowOrArrow(second)) {
                  deadflag = 3;
                  print_ret "Before the startled soldiers can react, you turn
                      and fire at Gessler; your arrow pierces his heart,
                      and he dies messily. A gasp, and then a cheer,
                      goes up from the crowd.";
              }
              return true;
          ],
    has   male;
```

Like most NPCs, Gessler has a `life` property which deals with actions applicable only to `animate` objects. This one responds merely to `Talk` (as in TALK TO THE GOVERNOR).

Walter and the apple

Since he's been with you throughout, it's really about time we defined Walter:

TYPE

```
  NPC     son "your son"
    with  name 'son' 'your' 'boy' 'lad' 'Walter',
          description [;
              if (location == marketplace)
                  print_ret "He stares at you, trying to appear brave and
                      remain still. His arms are pulled back and tied behind
                      the trunk, and the apple nestles amid his blond hair.";
              else
                  print_ret "A quiet, blond lad of eight summers, he's fast
                      learning the ways of mountain folk.";
          ],
          life [;
              Give:
                  score = score + 1;
                  move noun to self;
                  print_ret "~Thank you, Papa.~";
              Talk:
                  if (location == marketplace)
                      print_ret "~Stay calm, my son, and trust in God.~";
                  else
                      print_ret "You point out a few interesting sights.";
          ],
          before [;
              Examine,Listen,Salute,Talk: return false;
              FireAt:
                  if (location == marketplace) {
                      if (BowOrArrow(second)) {
                          deadflag = 3;
                          print_ret "Oops! Surely you didn't mean to do that?";
                      }
                      return true;
                  }
```

```
                else
                    return false;
            default:
                if (location == marketplace)
                    print_ret "Your guards won't permit it.";
                else
                    return false;
        ],
        found_in [; return true; ],
  has   male proper scenery transparent;
```

His attributes are `male` (he's your son, after all), `proper` (so the interpreter doesn't mention "the your son"), `scenery` (so he's not listed in every room description), and `transparent` (because you see right through him). No, that's wrong: a `transparent` object isn't made of glass; it's one whose possessions are visible to you. We've done that because we'd still like to be able to EXAMINE APPLE even when Walter is carrying it. Without the `transparent` attribute, it would be as though the apple was in his pocket or otherwise out of sight; the interpreter would reply "You can't see any such thing".

Walter has a `found_in` property which automatically moves him to the player's location on each turn. We can get away with this because in such a short and simple game, he does indeed follow you everywhere. In a more realistic model world, NPCs often move around independently, but we don't need such complexity here.

Several of Walter's properties test whether `(location == marketplace)`; that is, is the player (and hence Walter) currently in that room? The events in the marketplace are such that specialised responses are more appropriate there than our standard ones.

Walter's `life` property responds to `Give` (as in GIVE APPLE TO WALTER) and `Talk` (as in TALK TO YOUR SON); during `Give`, we increment the library variable `score`, thus rewarding the player's generous good nature. His `before` property is perhaps a little confusing. It's saying:

1. The `Examine`, `Listen`, `Salute` and `Talk` actions are always available (a `Talk` action then gets passed to Walter's `life` property).
2. The `FireAt` action is permitted in the `marketplace`, albeit with unfortunate results. Elsewhere, it triggers the standard `FireAt` response of "Pretty dangerous, don't you think?"
3. All other actions are prevented in the `marketplace`, and allowed to run their standard course (thanks to the `return false`) elsewhere.

The apple's moment of glory has arrived! Its `before` property responds to the `FireAt` action by setting `deadflag` to 2. When that happens, the game is over; the player has won.

TYPE

```
Object  apple "apple"
  with  name 'apple',
        description [;
            if (location == marketplace)
                print_ret "At this distance you can barely see it.";
            else
                print_ret "The apple is blotchy green and brown.";
        ],
        before [;
            Drop: print_ret "An apple is worth quite a bit --
                better hang on to it.";
            Eat: print_ret "Helga intended it for Walter...";
            FireAt:
                if (location == marketplace) {
                    if (BowOrArrow(second)) {
                        score = score + 1;
                        deadflag = 2;
                        print_ret "Slowly and steadily, you place an arrow in
                            the bow, draw back the string, and take aim with
                            more care than ever in your life. Holding your
                            breath, unblinking, fearful, you release the
                            arrow. It flies across the square towards your
                            son, and drives the apple against the trunk of
                            the tree. The crowd erupts with joy;
                            Gessler looks distinctly disappointed.";
                    }
                    return true;
                }
                else
                    return false;
        ];
```

And with that, we've defined all of the objects. In doing so, we've added a whole load of new nouns and adjectives to the game's dictionary, but no verbs. That's the final task.

Verbs, verbs, verbs

The Inform library delivers a standard set of nearly a hundred actions which players can perform; around twenty of those are "meta-actions" (like SAVE and QUIT) aimed at the interpreter itself, and the remainder operate within the model world. Having such a large starting set is a great blessing; it means that many of the actions which players might attempt are already catered for, either by the interpreter doing something useful, or by explaining why it's unable to. Nevertheless, most games find the need to define additional actions, and "William Tell" is no exception. We'll be adding four actions of our own: `Untie`, `Salute` (see page 100), `FireAt` (see page 102) and `Talk` (see page 103).

Untie

It's not the most useful action, but it *is* the simplest. In the marketplace, when Walter is lashed to the tree, it's possible that players might be tempted to try to UNTIE WALTER; unlikely, but as we've said before, anticipating the

improbable is part of the craft of IF. For this, and for all new actions, two things are required. We need a grammar definition, spelling out the structure of the English sentences which we're prepared to accept:

TYPE
```
Verb 'untie' 'unfasten' 'unfix' 'free' 'release'
    * noun                          -> Untie;
```

and we need a routine to handle the action in the default situation (where the action *isn't* intercepted by an object's `before` property).

TYPE
```
[ UntieSub; print_ret "You really shouldn't try that."; ];
```

The grammar is less complex than it perhaps at first appears:

1. The English verbs UNTIE, UNFASTEN, UNFIX, FREE and RELEASE are synonymous.
2. The asterisk `*` indicates the start of a pattern defining what word(s) might follow the verb.
3. In this example, there's only one pattern: the "`noun`" token represents an object which is currently in scope – in the same room as the player.
4. The `->` indicates an action to be triggered.
5. If players type something that matches the pattern – one of those five verbs followed by an object in scope – the interpreter triggers an `Untie` action, which by default is handled by a routine having the same name as the action, with `Sub` appended. In this example, that's the `UntieSub` routine.
6. The grammar is laid out this way just to make it easier to read. All those spaces aren't important; we could equally have typed:

   ```
   Verb 'untie' 'unfasten' 'unfix' 'free' 'release' * noun -> Untie;
   ```

We can illustrate how this works in the Altdorf street:

```
A street in Altdorf
The narrow street runs north towards the town square. Local folk are pouring
into the town through the gate to the south, shouting greetings, offering
produce for sale, exchanging news, enquiring with exaggerated disbelief about
the prices of the goods displayed by merchants whose stalls make progress even
more difficult.

"Stay close to me, son," you say, "or you'll get lost among all these people."

>UNTIE
What do you want to untie?

>UNFASTEN THE DOG
You can't see any such thing.

>UNTIE THE PEOPLE
You don't need to worry about the local people.

>UNFIX YOUR SON
You really shouldn't try that.
```

The illustration shows four attempted usages of the new action. In the first, the player omits to mention an object; the interpreter knows (from that `noun` in the grammar which implies that the action needs a direct object) that something is missing, so it issues a helpful prompt. In the second, the player mentions an object that isn't in scope (in fact, there's no dog anywhere in the game, but the interpreter isn't about to give *that* away to the player). In the third, the object is in scope, but its `before` property intercepts the `Untie` action (and indeed, since this object is of the class `Prop`, *all* actions apart from `Examine`) to display a customised rejection message. Finally, the fourth usage refers to an object which *doesn't* intercept the action, so the interpreter calls the default action handler – `UntieSub` – which displays a general-purpose refusal to perform the action.

The principles presented here are those that you should generally employ: write a generic action handler which either refuses to do anything (see, for example SQUASH or HIT), or performs the action without affecting the state of the model world (see, for example, JUMP or WAVE); then, intercept that non-action (generally using a `before` property) for those objects which might make a legitimate target for the action, and instead provide a more specific response, either performing or rejecting the action.

In the case of `Untie`, there are no objects which can be untied in this game, so we always generate a refusal of some sort.

Salute

The next action is `Salute`, provided in case Wilhelm chooses to defer to the hat on the pole. Here's the default action handler:

TYPE
```
[ SaluteSub;
    if (noun has animate) print_ret (The) noun, " acknowledges you.";
    print_ret (The) noun, " takes no notice.";
];
```

You'll notice that this is slightly more intelligent than our `Untie` handler, since it produces different responses depending on whether the object being saluted – stored in the `noun` variable – is `animate` or not. But it's basically doing the same job. And here's the grammar:

TYPE
```
Verb 'bow' 'nod' 'kowtow' 'genuflect'
    * 'at'/'to'/'towards' noun      -> Salute;

Verb 'salute' 'greet' 'acknowledge'
    * noun                          -> Salute;
```

This grammar says that:

1. The English verbs BOW, NOD, KOWTOW, GENUFLECT, SALUTE, GREET and ACKNOWLEDGE are synonymous.

2. The first four (but not the last three) can then be followed by any of the prepositions AT, TO or TOWARDS: words in apostrophes `'...'` are matched literally, with the slash `/` separating alternatives.

3. After that comes the name of an object which is currently in scope – in the same room as the player.
4. If players type something that matches one of those patterns, the interpreter triggers a `Salute` action, which by default is dealt with by the `SaluteSub` routine.

So, we're allowing BOW AT HAT and KOWTOW TOWARDS HAT, but not simply NOD HAT. We're allowing SALUTE HAT but not GREET TO HAT. It's not perfect, but it's a fair attempt at defining some new verbs to handle salutation.

But suppose that we think of still other ways in which players might attempt this (remember, they don't know which verbs we've defined; they're just stabbing in the dark, trying out things that seem as though they ought to work). How about PAY HOMAGE TO HAT, or maybe WAVE AT HAT? They sound pretty reasonable, don't they? Except that, if we'd written:

```
Verb 'bow' 'nod' 'kowtow' 'genuflect' 'wave'
    * 'at'/'to'/'towards' noun      -> Salute;
```

we'd have caused a compilation error: two different verb definitions refer to "wave". `Grammar.h`, one of the library files whose contents a beginner might find useful to study, contains these lines:

```
Verb 'give' 'pay' 'offer' 'feed'
    * held 'to' creature          -> Give
    * creature held               -> Give reverse
    * 'over' held 'to' creature   -> Give;

Verb 'wave'
    *                             -> WaveHands
    * noun                        -> Wave;
```

The problem is that the verbs PAY and WAVE are already defined by the library, and Inform's rule is that a verb can appear in only one `Verb` definition. The wrong solution: edit `Grammar.h` to *physically* add lines to the existing definitions (it's almost never a good idea to make changes to the standard library files). The right solution: use `Extend` to *logically* add those lines. If we write this in our source file:

TYPE

```
Extend 'give'
    * 'homage' 'to' noun          -> Salute;

Extend 'wave'
    * 'at' noun                   -> Salute;
```

then the effect is exactly as if we'd edited `Grammar.h` to read like this:

```
Verb 'give' 'pay' 'offer' 'feed'
    * held 'to' creature          -> Give
    * creature held               -> Give reverse
    * 'over' held 'to' creature   -> Give
    * 'homage' 'to' noun          -> Salute;

Verb 'wave'
    *                             -> WaveHands
    * noun                        -> Wave
    * 'at' noun                   -> Salute;
```

and now players can PAY (or GIVE, or OFFER) HOMAGE to any object. (Because GIVE, PAY, OFFER and FEED are defined as synonyms, players can also FEED HOMAGE, but it's unlikely that anybody will notice this minor aberration; players are usually too busy trying to figure out *logical* possibilities.)

FireAt

As usual, we'll show you the default handler for this action:

TYPE

```
[ FireAtSub;
    if (noun == nothing)
        print_ret "What, just fire off an arrow at random?";
    if (BowOrArrow(second))
        print_ret "Pretty dangerous, don't you think?";
];
```

followed by the grammar:

TYPE

```
Verb 'fire' 'shoot' 'aim'
    *                               -> FireAt
    * noun                          -> FireAt
    * 'at' noun                     -> FireAt
    * 'at' noun 'with' noun         -> FireAt
    * noun 'with' noun              -> FireAt
    * noun 'at' noun                -> FireAt reverse;
```

This is the most complex grammar that we'll write, and the first one offering several different options for the words which follow the initial verb. The first line of grammar:

```
    *                               -> FireAt
```

is going to let us type FIRE (or SHOOT, or AIM) by itself. The second line:

```
    * noun                          -> FireAt
```

supports FIRE BOW or FIRE ARROW (or something less sensible like FIRE TREE). The third line:

```
    * 'at' noun                     -> FireAt
```

accepts FIRE AT APPLE, FIRE AT TREE, and so on. Note that there's only one semicolon in all of the grammar, right at the very end.

The first two statements in `FireAtSub` deal with the first line of grammar: FIRE (or SHOOT, or AIM) by itself. If the player types just that, both `noun` and `second` will contain `nothing`, so we reject the attempt with the "at random?" message. Otherwise, we've got at least a `noun` value, and possibly a `second` value also, so we make our standard check that `second` is something that can be fired, and then reject the attempt with the "Pretty dangerous" message.

There are a couple of reasons why you might find this grammar a bit tricky. The first is that on some lines the word `noun` appears twice: you need to remember that in this context `noun` is a parsing token which matches any single object visible to the player. Thus, the line:

```
    * 'at' noun 'with' noun         -> FireAt
```

is matching FIRE AT `some_visible_target` WITH `some_visible_weapon`; perhaps confusingly, the value of the target object is then stored in variable `noun`, and the value of the weapon object in variable `second`.

The second difficulty may be the final grammar line. Whereas on the preceding lines, the first `noun` matches a target object and the second `noun`, if present, matches a weapon object, that final line matches FIRE `some_visible_weapon` AT `some_visible_target` – the two objects are mentioned in the wrong sequence. If we did nothing, our `FireAtSub` would get pretty confused at this point, but we can swap the two objects back into the expected order by adding that `reverse` keyword at the end of the line, and then `FireAtSub` will work the same in all cases.

Talk

The final action that we define – `Talk` – provides a simple system of canned conversation, a low-key replacement for the standard `Answer`, `Ask` and `Tell` actions. The default `TalkSub` handler is closely based on `TellSub` (defined in library file `verblibm.h`, should you be curious), and does three things:

1. Deals with TALK TO ME or TALK TO MYSELF.
2. Checks (a) whether the creature being talked to has a `life` property, (b) whether that property is prepared to process a `Talk` action, and (c) if the `Talk` processing returns `true`. If all three checks succeed then `TalkSub` need do nothing more; if one or more of them fails then `TalkSub` simply...
3. Displays a general “nothing to say” refusal to talk.

TYPE

```
[ TalkSub;
    if (noun == player) print_ret "Nothing you hear surprises you.";
    if (RunLife(noun,##Talk) ~= false) return;
    print_ret "At the moment, you can't think of anything to say.";
];
```

NOTE: that second condition `(RunLife(noun,##Talk) ~= false)` is a bit of a stumbling block, since it uses `RunLife` – an undocumented internal library routine – to offer the `Talk` action to the NPC’s `life` property. We’ve decided to use it in exactly the same way as the `Tell` action does, without worrying too much about how it works (though it looks as though `RunLife` returns some `true` value if the `life` property has intercepted the action, `false` if it hasn’t).

The grammar is straightforward; notice the use of `'t//'` to define T as a synonym for TALK:

TYPE

```
Verb 'talk' 't//' 'converse' 'chat' 'gossip'
    * 'to'/'with' creature          -> Talk
    * creature                      -> Talk;
```

Here’s the simplest `Talk` handler that we’ve seen – it’s from Gessler the governor. Any attempt to TALK TO GESSLER will provoke “You cannot bring yourself to speak to him”.

```
life [;
    Talk: print_ret "You cannot bring yourself to speak to him.";
],
```

Walter's `Talk` handler is only slightly more involved:

```
life [;
    Talk:
        if (location == marketplace)
            print_ret "~Stay calm, my son, and trust in God.~";
    print_ret "You point out a few interesting sights.";
],
```

And Helga's is the most sophisticated (though that isn't saying much):

```
times_spoken_to 0,       ! for counting the conversation topics
life [; Talk:
        self.times_spoken_to = self.times_spoken_to + 1;
        switch (self.times_spoken_to) {
            1:  score = score + 1;
                print_ret "You warmly thank Helga for the apple.";
            2:  score = score + 1;
                print_ret "~See you again soon.~";
            default: return false;
        }
],
```

This handler uses Helga's `times_spoken_to` property – not a library property, it's one that we invented, like the `mid_square.warnings_count` and `pole.has_been_saluted` properties – to keep track of what's been said, permitting two snatches of conversation (and awarding some points) before falling back on the embarrassing silences implied by "You can't think of anything to say".

That's the end of our little fable; you'll find a transcript and the full source in "William Tell" story on page 195. And now, it's time to meet – Captain Fate!

10 • Captain Fate: take 1

S was a sailor, and spent all he got;
T was a tinker, and mended a pot.

Simple though they are, our two games have covered most of the basic functionality of Inform, providing enough solid ground underfoot for you to start creating simple stories. Even if some of what you've encountered doesn't make sense yet, you should be able to browse a game's source code and form a general understanding of what is going on.

We'll now design a third game, to show you a few additional features and give you some more sample code to analyse. In "Heidi" we tried to make progress step by step, explaining every bit of code that went into the game as we laid the objects sequentially; in "William Tell" you'll have noticed that we took a few necessary explanatory detours, as the concepts grew more interesting and complicated. Here we'll organise the information in logical didactic chunks, defining some of the objects minimally at first and then adding complexity as need arises. Again, this means that you won't be able to compile for testing purposes after the addition of every code snippet, so, if you're typing in the game as you read, you'll need to check the advice in "Compile-as-you-go" on page 227.

A lot of what goes into this game we have already seen; you may deduce from this that the game design business is fairly repetitious and that most games are, when you reach the programming bottom line, another remake of the same old theme. Well, yes and no: you've got a camera and have seen some short home videos in the making, but it's a long way from here to Casablanca. To stick with the analogy, we'll now construct the opening sequence of an indie B-movie, a tribute to the style of super-hero made famous by a childhood of comic books:

> "Impersonating mild mannered John Covarth, assistant help boy at an insignificant drugstore, you suddenly STOP when your acute hearing deciphers a stray radio call from the POLICE. There's some MADMAN attacking the population in Granary Park! You must change into your Captain FATE costume fast...!"

which won't be so easy to do. In this short example, players will win when they manage to change into their super-hero costume and fly away to meet the foe. The confrontation will – perhaps – take place in some other game, where we can but hope that Captain Fate will vanquish the forces of evil, thanks to his mysterious (and here unspecified) superpowers.

Fade up on: a nondescript city street

The game starts with meek John Covarth walking down the street. We set up the game as usual:

TYPE

```
!============================================================================
Constant Story "Captain Fate";
Constant Headline
            "^A simple Inform example
             ^by Roger Firth and Sonja Kesserich.^";
Release 2;  Serial "020827";    ! for keeping track of public releases

Constant MANUAL_PRONOUNS;
Constant MAX_SCORE     2;
Constant OBJECT_SCORE  1;
Constant ROOM_SCORE    1;

Include "Parser";
Include "VerbLib";

!============================================================================
! Object classes

Class   Room
  with  description "UNDER CONSTRUCTION",
  has   light;

Class   Appliance
  with  before [; Take,Pull,Push,PushDir:
            "Even though your SCULPTED adamantine muscles are up to the task,
             you don't favour property damage.";
        ],
  has   scenery;

!============================================================================
! The game objects

Room    street "On the street"
  with  name 'city' 'buildings' 'skyscrapers' 'shops' 'apartments' 'cars',
        description [;
            "On one side -- which your HEIGHTENED sense of direction
             indicates is NORTH -- there's an open cafe now serving
             lunch. To the south, you can see a phone booth.";

!============================================================================
! The player's possessions

!============================================================================
! Entry point routines

[ Initialise;
    location = street;
    lookmode = 2;
    "^^Impersonating mild mannered John Covarth, assistant help boy at an
     insignificant drugstore, you suddenly STOP when your acute hearing
     deciphers a stray radio call from the POLICE. There's some MADMAN
     attacking the population in Granary Park! You must change into your
     Captain FATE costume fast...!^^";
];
!============================================================================
! Standard and extended grammar

Include "Grammar";

!============================================================================
```

Almost everything is familiar, apart from a few details:

```
Constant MANUAL_PRONOUNS;
Constant MAX_SCORE     2;
Constant OBJECT_SCORE  1;
Constant ROOM_SCORE    1;
```

By default, Inform uses a system of automatic pronouns: as the player character moves into a room, the library assigns pronouns like IT and HIM to likely objects (if you play "Heidi" or "William Tell" and type PRONOUNS, you can see how the settings change). There is another option. If we declare the `MANUAL_PRONOUNS` constant, we force the library to assign pronouns to objects only as the player mentions them (that is, IT will be unassigned until the player types, say, EXAMINE TREE, at which point, IT becomes the TREE). The behaviour of pronoun assignment is a matter of personal taste; no system is objectively perfect.

Apart from the constant `MAX_SCORE` that we have seen in "William Tell", which defines the maximum number of points to be scored, we now see two more constants: `OBJECT_SCORE` and `ROOM_SCORE`. There are several scoring systems predefined in Inform. In "William Tell" we've seen how you can manually add (or subtract) points by changing the value of the variable `score`. Another approach is to award points to players on the first occasion that they (a) enter a particular room, or (b) pick up a particular object. To define that a room or object is indeed "particular", all you have to do is give it the attribute `scored`; the library take cares of the rest. The predefined scores are five points for finally reached rooms and four points for wondrous acquisition of objects. With the constants `OBJECT_SCORE` and `ROOM_SCORE` we can change those defaults; for the sake of example, we've decided to modestly award one point for each. By the way, the use of an equals sign `=` is optional with `Constant`; these two lines have identical effect:

```
Constant ROOM_SCORE    1;

Constant ROOM_SCORE  = 1;
```

Another difference has to do with a special short-hand method that Inform provides for displaying strings of text. Until now, we have shown you:

```
print "And now for something completely different...^"; return true;
...
print_ret "And now for something completely different...";
```

Both lines do the same thing: they display the quoted string, output a newline character, and return true. As you have seen in the previous example games, this happens quite a lot, so there is a yet shorter way of achieving the same result:

```
"And now for something completely different...";
```

That is, *in a routine* (where the compiler is expecting to find a collection of statements each terminated by a semicolon), a string in double quotes by itself, without the need for any explicit keywords, works exactly as if there were a `print_ret` in front of it. Remember that this way of displaying text implies a `return true` at the end (which therefore exits from the routine immediately). This

detail becomes important if we *don't* want to return true after the string has been displayed on the screen – we should use the explicit `print` statement instead.

You'll notice that – unusually for a room – our `street` object has a `name` property:

```
Room    street "On the street"
  with  name 'city' 'buildings' 'skyscrapers' 'shops' 'apartments' 'cars',
        ...
```

Rooms aren't normally referenced by name, so this may seem odd. In fact, we're illustrating a feature of Inform: the ability to define dictionary words as "known but irrelevant" in this location. If the player types EXAMINE CITY here, the interpreter will reply "That's not something you need to refer to in order to SAVE the day", rather than the misleading "You can' t see any such thing". We mostly prefer to deal with such scenic words using classes like `Prop` and `Furniture`, but sometimes a room's `name` property is a quick and convenient solution.

In this game, we provide a class named `Appliance` to take care of furniture and unmovable objects. You'll notice that the starting room we have defined has no connections yet. The description mentions a phone booth and a café, so we might want to code those. While the café will be a normal room, it would seem logical that the phone booth is actually a big box on the sidewalk; therefore we define a `container` set in the street, which players may enter.

TYPE

```
Appliance booth "phone booth" street
  with  name 'old' 'red' 'picturesque' 'phone' 'booth' 'cabin'
            'telephone' 'box',
        description
            "It's one of the old picturesque models, a red cabin with room
             for one caller.",
        before [;
            Open:  "The booth is already open.";
            Close: "There's no way to close this booth.";
        ],
        after [; Enter:
            "With implausible celerity, you dive inside the phone booth.";
        ],
  has   enterable container open;
```

What's interesting are the attributes at the end of the definition. You'll recall from Heidi's `nest` object that a `container` is an object capable of having other objects placed in it. If we make something `enterable`, players count as one of those objects, so that they may squeeze inside. Finally, `containers` are, by default, supposed to be closed. You can make them `openable` if you wish players to be able to OPEN and CLOSE them at will, but this doesn't seem appropriate behaviour for a public cabin – it would become tedious to have to type OPEN BOOTH and CLOSE BOOTH when these actions provide nothing special – so we add instead the attribute `open` (as we did with the nest), telling the interpreter that the container is open from the word go. Players aren't aware of our design, of course; they may indeed try to OPEN and CLOSE the booth, so we trap those actions in a `before` property which just tells them that these are not relevant options. The

`after` property gives a customised message to override the library's default for commands like ENTER BOOTH or GO INSIDE BOOTH.

Since in the street's description we have told players that the phone booth is to the south, they might also try typing SOUTH. We must intercept this attempt and redirect it (while we're at it, we add a connection to the as-yet-undefined café room and a default message for the movement which is not allowed):

```
Room    street "On the street"
  with  name 'city' 'buildings' 'skyscrapers' 'shops' 'apartments' 'cars',
        description [;
            "On one side -- which your HEIGHTENED sense of direction
             indicates is NORTH -- there's an open cafe now serving
             lunch. To the south, you can see a phone booth.",
        n_to cafe,
        s_to [; <<Enter booth>>; ],
        cant_go
            "No time now for exploring! You'll move much faster in your
             Captain FATE costume.";
```

That takes care of entering the booth. But what about leaving it? Players may type EXIT or OUT while they are inside an enterable container and the interpreter will oblige but, again, they might type NORTH. This is a problem, since we are actually in the street (albeit inside the booth) and to the north we have the café. We may provide for this condition in the room's `before` property:

```
before [; Go:
    if (player in booth && noun == n_obj)  <<Exit booth>>;
],
```

Since we are outdoors and the booth provides a shelter, it's not impossible that a player may try just IN, which is a perfectly valid connection. However, that would be an ambiguous command, for it could also refer to the café, so we express our bafflement and force the player to try something else:

```
n_to cafe,
s_to [; <<Enter booth>>; ],
in_to "But which way?",
```

Now everything seems to be fine, except for a tiny detail. We've said that, while in the booth, the player character's location is still the `street` room, regardless of being inside a `container`; if players chanced to type LOOK, they'd get:

```
On the street (in the phone booth)
On one side -- which your HEIGHTENED sense of direction indicates is NORTH --
there's an open cafe now serving lunch. To the south, you can see a phone booth.
```

Hardly an adequate description while *inside* the booth. There are several ways to fix the problem, depending on the result you wish to achieve. The library provides a property called `inside_description` which you can utilise with enterable containers. It works pretty much like the normal `description` property, but it gets printed only when the player is inside the container. The library makes use of this property in a very clever way, because for every LOOK action it checks whether we can see outside the container: if the container has the `transparent` attribute set,

or if it happens to be `open`, the library displays the normal `description` of the room first and then the `inside_description` of the container. If the library decides we can't see outside the container, only the `inside_description` is displayed. Take for instance the following (simplified) example:

```
Room    stage "On stage"
  with  description
            "The stage is filled with David Copperfield's
             magical contraptions.",
        ...

Object  magic_box "magic box" stage
  with  description
            "A big elongated box decorated with silver stars, where
             scantily clad ladies make a disappearing act.",
        inside_description
            "The inside panels of the magic box are covered with black
             velvet. There is a tiny switch by your right foot.",
        ...
  has   container openable enterable light;
```

Now, the player would be able to OPEN BOX and ENTER BOX. A player who tried a LOOK would get:

```
On stage (in the magic box)
The stage is filled with David Copperfield's magical contraptions.

The inside panels of the magic box are covered with black velvet. There is a
tiny switch by your right foot.
```

If now the player closes the box and LOOKs:

```
On stage (in the magic box)
The inside panels of the magic box are covered with black velvet. There is a
tiny switch by your right foot.
```

In our case, however, we don't wish the description of the street to be displayed at all (even if a caller is supposedly able to see the street while inside a booth). The problem is that we have made the booth an `open` container, so the street's description would be displayed every time. There is another solution. We can make the `description` property of the `street` room a bit more complex, and change its value: instead of a string, we write an embedded routine. Here's the (almost) finished room:

TYPE

```
Room    street "On the street"
  with  name 'city' 'buildings' 'skyscrapers' 'shops' 'apartments' 'cars',
        description [;
            if (player in booth)
                "From this VANTAGE point, you are rewarded with a broad view
                 of the sidewalk and the entrance to Benny's cafe.";
            else
                "On one side -- which your HEIGHTENED sense of direction
                 indicates is NORTH -- there's an open cafe now serving
                 lunch. To the south, you can see a phone booth.";
        ],
        before [; Go:
            if (player in booth && noun == n_obj) <<Exit booth>>;
        ],
```

```
        n_to cafe,
        s_to [; <<Enter booth>>; ],
        in_to "But which way?",
        cant_go
            "No time now for exploring! You'll move much faster in your
             Captain FATE costume.";
```

The description while inside the booth mentions the sidewalk, which might invite the player to EXAMINE it. No problem:

```
Appliance "sidewalk" street
  with  name 'sidewalk' 'pavement' 'street',
        article "the",
        description
            "You make a quick surveillance of the sidewalk and discover much
             to your surprise that it looks JUST like any other sidewalk in
             the CITY!";
```

Unfortunately, both descriptions also mention the café, which will be a room and therefore not, from the outside, an examinable object. The player may enter it and will get whatever description we code as the result of a LOOK action (which will have to do with the way the café looks from the *inside*); but while we are on the street we need something else to describe it:

```
Appliance outside_of_cafe "Benny's cafe" street
  with  name 'benny^s' 'cafe' 'entrance',
        description
            "The town's favourite for a quick snack, Benny's cafe has a 50's
             ROCKETSHIP look.",
        before [; Enter:
            print "With an impressive mixture of hurry and nonchalance
                you step into the open cafe.^";
            PlayerTo(cafe);
            return true;
        ],
  has   enterable proper;
```

NOTE: although the text of our guide calls Benny's establishment a "café" – note the acute "e" – the game itself simplifies this to "cafe". We do this for clarity, not because Inform doesn't support accented characters. The *Inform Designer's Manual* explains in detail how to display these characters in "§1.11 *How text is printed*" and provides the whole Z-Machine character set in Table 2. In our case, we could have displayed this:

```
The town's favourite for a quick snack, Benny's café has a 50's ROCKETSHIP look.
```

by defining the `description` property as any of these:

```
description
    "The town's favourite for a quick snack, Benny's caf@'e has a 50's
     ROCKETSHIP look.",

description
    "The town's favourite for a quick snack, Benny's caf@@170 has a 50's
     ROCKETSHIP look.",
```

```
description
    "The town's favourite for a quick snack, Benny's caf@{E9} has a 50's
     ROCKETSHIP look.",
```

However, all three forms are harder to read than the vanilla "cafe", so we've opted for the simple life.

Unlike the sidewalk object, we offer more than a mere description. Since the player may try ENTER CAFE as a reasonable way of access – which would have confused the interpreter immensely – we take the opportunity of making this object also `enterable`, and we cheat a little. The attribute `enterable` has permitted the verb ENTER to be applied to this object, but this is not a `container`; we want the player to be sent into the *real* café room instead. The `before` property handles this, intercepting the action, displaying a message and teleporting the player into the café. We `return true` to inform the interpreter that we have taken care of the `Enter` action ourselves, so it can stop right there.

As a final detail, note that we now have two ways of going into the café: the `n_to` property of the `street` room and the `Enter` action of the `outside_of_cafe` object. A perfectionist might point out that it would be neater to handle the actual movement of the player in just one place of our code, because this helps clarity. To achieve this, we redirect the street's `n_to` property thus:

TYPE

```
n_to [; <<Enter outside_of_cafe>>; ],
```

You may think that this is unnecessary madness, but a word to the wise: in a large game, you want action handling going on just in one place when possible, because it will help you to keep track of where things are a-happening if something goes *ploof* (as, believe us, it will; see "Debugging your game" on page 173). You don't need to be a perfectionist, just cautious.

A booth in this kind of situation is an open invitation for the player to step inside and try to change into Captain Fate's costume. We won't let this happen – the player isn't Clark Kent, after all; later we'll explain how we forbid this action – and that will force the player to go inside the café, looking for a discreet place to disrobe; but first, let's freeze John Covarth outside Benny's and reflect about a fundamental truth.

A hero is not an ordinary person

Which is to say, normal actions won't be the same for him.

As you have probably inferred from the previous chapters, some of the library's standard defined actions are less important than others in making the game advance towards one of its conclusions. The library defines PRAY and SING, for instance, which are of little consequence in a normal gaming situation; each displays an all-purpose message, sufficiently non-committal, and that's it. Of course, if your game includes a magic portal that will reveal itself only if the player lets rip with a snatch of Wagner, you may intercept the `Sing` action in a

`before` property and alter its default, pretty useless behaviour. If not, it's "Your singing is abominable" for you.

All actions, useful or not, have a stock of messages associated with them (the messages are held in the `english.h` library file and listed in Appendix 4 of the *Inform Designer's Manual*). We have already seen one way of altering the player character's description – "As good looking as ever" – in "William Tell", but the other defaults may also be redefined to suit your tastes and circumstantial needs.

John Covarth, aka Captain Fate, could happily settle for most of these default messages, but we deem it worthwhile to give him some customised responses. If nothing else, this adds to the general atmosphere, a nicety that many players regard as important. For this mission, we make use of the `LibraryMessages` object.

TYPE

```
Include "Parser";

Object  LibraryMessages         ! must be defined between Parser and VerbLib
  with  before [;
            Buy:    "Petty commerce has rarely interested you.";
            Dig:    "Your keen senses detect NOTHING underground worth your
                     immediate attention.";
            Pray:   "You won't need to bother almighty DIVINITIES to save
                     the day.";
            Sing:   "Alas! That is not one of your many superpowers.";
            Sleep:  "A hero is ALWAYS on the watch.";
            Sorry:  "Captain FATE has no time for apologies, only for
                     ACTION.";
            Strong: "An unlikely vocabulary for a HERO like you.";
            Swim:   "You quickly turn all your ATTENTION towards locating a
                     suitable place to EXERCISE your superior strokes,
                     but alas! you find none.";
            Miscellany:
                if (lm_n == 19)
                    if (clothes has worn)
                        "In your secret identity's outfit, you manage most
                         efficaciously to look like a two-cent loser, a
                         good-for-nothing wimp.";
                    else
                        "Now that you are wearing your costume, you project
                         the image of power UNBOUND, of ballooned,
                         multicoloured MUSCLE, of DASHING yet MODEST chic.";
                if (lm_n == 38)
                    "That's not a verb you need to SUCCESSFULLY save the
                     day.";
                if (lm_n == 39)
                    "That's not something you need to refer to in order to
                     SAVE the day.";
        ];

Include "VerbLib";
```

If you provide it, the `LibraryMessages` object must be defined *between* the inclusion of `Parser` and `VerbLib` (it won't work otherwise and you'll get a compiler error). The object contains a single property – `before` – which intercepts display of the default messages that you want to change. An attempt to SING, for example, will now result in "Alas! That is not one of your many superpowers" being displayed.

In addition to such verb-specific responses, the library defines other messages not directly associated with an action, like the default response when a verb is unrecognised, or if you refer to an object which is not in scope, or indeed many other things. Most of these messages can be accessed through the `Miscellany` entry, which has a numbered list of responses. The variable `lm_n` holds the current value of the number of the message to be displayed, so you can change the default with a test like this:

```
if (lm_n == 39)
    "That's not something you need to refer to in order to SAVE the day.";
```

where 39 is the number for the standard message "That's not something you need to refer to in the course of this game" – displayed when the player mentions a noun which is listed in a room's name property, as we did for the `street`.

Not surprisingly, the default message for self-examination: "As good looking as ever" is a `Miscellany` entry – it's number 19 – so we can change it through the `LibraryMessages` object instead of, as before, assigning a new string to the `player.description` property. In our game, the description of the player character has two states: with street clothes as John Covarth and with the super-hero outfit as Captain Fate; hence the `if (clothes has worn)` clause.

This discussion of changing our hero's appearance shows that there are different ways of achieving the same result, which is a common situation while designing a game. Problems may be approached from different angles; why use one technique and not another? Usually, the context tips the balance in favour of one solution, though it might happen that you opt for the not-so-hot approach for some overriding reason. Don't feel discouraged; choices like this become more common (and easier) as your experience grows.

> NOTE: going back to our example, an alternative approach would be to set the variable `player.description` in the `Initialise` routine (as we did with "William Tell") to the "ordinary clothes" string, and then later change it as the need arises. It is a variable, after all, and you can alter its value with another statement like `player.description =` *`whatever new look`* anywhere in your code. This alternative solution might be better if we intended changing the description of the player many times through the game. Since we plan to have only two states, the `LibraryMessages` approach will do just fine.

A final warning: as we explained when extending the standard verb grammars, you *could* edit the appropriate library file and change all the default messages, but that wouldn't be a sound practice, because your library file will probably not be right for the next game. Use of the `LibraryMessages` object is strongly advised.

If you're typing in the game, you'll probably want to read the brief section on "Compile-as-you-go" on page 227 prior to performing a test compile. Once everything's correct, it's time that our hero entered that enticing café.

11 • Captain Fate: take 2

U was a usurer, a miserable elf;
V was a vintner, who drank all himself.

iewed from the inside, Benny's café is warm and welcoming, and packed with lunchtime customers. We'll try to conjure up some appropriate images, but the main focus of the room isn't the decor: it's the door leading to the toilet – and, perhaps, privacy?

A homely atmosphere

Benny's café is populated with customers enjoying their lunch, so it won't be a good place to change identities. However, the toilet to the north looks promising, though Benny has strict rules about its use and the door seems to be locked.

> CULTURAL NOTE: not for the first time, this guide betrays its origins. In European countries the word "toilet" often refers not only to the white porcelain artefact, but also to the room in which it can be found (also, a "bathroom" is for taking a bath, a "restroom" for taking a rest). Bear with us on this; the dual usage becomes important a little later on.

We define the café room in simple form:

```
Room    cafe "Inside Benny's cafe"
  with  description
        "Benny's offers the FINEST selection of pastries and
         sandwiches. Customers clog the counter, where Benny himself
         manages to serve, cook and charge without missing a step. At
         the north side of the cafe you can see a red door connecting
         with the toilet.",
        s_to street,
        n_to toilet_door;
```

We'll elaborate on the last line (`n_to toilet_door`) later, when we define the door object which lies between the café and the toilet.

We've mentioned a counter:

```
Appliance counter "counter" cafe
  with  name 'counter' 'bar',
        article "the",
        description
            "The counter is made of an astonishing ALLOY of metals, CRUMB- &
             SPILL-RESISTANT and EASY to clean. Customers enjoy their snacks
             with UTTER tranquility, safe in the notion that the counter can
             take it all.",
  has   supporter;
```

And some customers. These are treated as NPCs, reacting to our hero's performance.

TYPE

```
Object  customers "customers" cafe
  with  name 'customers' 'people' 'customer' 'men' 'women',
        description [;
            if (costume has worn)
                "Most seem to be concentrating on their food, but some do
                 look at you quite blatantly. Must be the MIND-BEFUDDLING
                 colours of your costume.";
            else
                "A group of HELPLESS and UNSUSPECTING mortals, the kind
                 Captain FATE swore to DEFEND the day his parents choked on a
                 DEVIOUS slice of RASPBERRY PIE.";
        ],
        life [;
            Ask,Tell,Answer:
                if (costume has worn)
                    "People seem to MISTRUST the look of your FABULOUS
                     costume.";
                else
                    "As John Covarth, you attract LESS interest than Benny's
                     food.";
            Kiss:
                "There's no telling what sorts of MUTANT bacteria these
                 STRANGERS may be carrying around.";
            Attack:
                "Mindless massacre of civilians is the qualification for
                 VILLAINS. You are SUPPOSED to protect the likes of these
                 people.";
        ],
        orders [;
            "These people don't appear to be of the cooperative sort.";
        ],
        number_of_comments 0,           ! for counting the customer comments
        daemon [;
            if (location == cafe && random(2) == 1) {
                self.number_of_comments = self.number_of_comments + 1;
                switch (self.number_of_comments) {
                    1:  "^~Didn't know there was a circus in town,~ comments
                         one customer to another. ~Seems like the clowns have
                         the day off.~";
                    2:  "^~These fashion designers don't know what to do to
                         show off,~ snorts a fat gentleman, looking your way.
                         Those within earshot try to conceal their smiles.";
                    3:  "^~Must be carnival again,~ says a man to his wife,
                         who giggles, stealing a peek at you.
                         ~Time sure flies.~";
                    4:  "^~Bad thing about big towns~, comments someone to
                         his table companion, ~is you get the damnedest bugs
                         coming out from toilets.~";
                    5:  "^~I sure WISH I could go to work in my pyjamas,~
                         says a girl in an office suit to some colleagues.
                         ~It looks SO comfortable.~";
                    default: StopDaemon(self);
                }
            }
        ],
   has  scenery animate pluralname;
```

Let's go step by step. Our hero enters the café dressed as John Covarth, but will eventually manage to change clothes in the toilet, and he'll have to cross back

through the café to reach the street and win the game. The customers' `description` takes into consideration which outfit the player character is wearing.

In "William Tell" we've seen a brief manifestation of the `life` property, but here we'll extend it a little. As we explained, `life` lets you intercept those actions particular to animate objects. Here we trap `Attack` and `Kiss` to offer some customised messages for these actions when applied to the customers. Also, we avoid conversation by intercepting `Ask`, `Tell` and `Answer` in order just to produce a message which depends on the player character's attire.

One other feature of `animate` objects is the possibility of giving them orders: BILL, SHAKE THE SPEAR or ANNIE, GET YOUR GUN. These actions are dealt with in the `orders` property and, as with the `life` property, the embedded routine can become quite complex if you want your NPCs to behave in an interesting way. In this case, we don't need the customers to perform tasks for us, so instead we provide a simple rejection message, just in case the player tries to order people around.

Which leaves us with the `daemon` bit. A daemon is a property normally used to perform some timed or repetitive action without the need of the player's direct interaction; for example, machines which work by themselves, animals that move on their own, or people going about their business. More powerfully, a daemon may take notice of the player's decisions at a particular moment, allowing for some interactive behaviour; this is, however, an advanced feature that we won't use in this example. A daemon gets a chance of doing something at the end of every turn, typically to (or with) the object to which it's associated. In our example, the daemon triggers some sneers and nasty comments from the customers once our hero comes out of the toilet dressed in Captain Fate's costume.

To code a daemon, you need to do three things:

1. First, define a `daemon` property in the object's body; the value of the property is always an embedded routine.
2. However, daemons do nothing until you activate them. This is easily achieved with the call `StartDaemon(`*obj_id*`)`, which may happen anywhere (if you want some object's daemon to be active from the beginning of the game, you can make the call in your `Initialise` routine).
3. Once the daemon has finished its mission (if ever) you may stop it with the call `StopDaemon(`*obj_id*`)`.

How does our particular daemon work? We want the customers to make snarky remarks once they see the costumed Captain, but not on a completely predictable basis.

```
if (random(2) == 1) ...
```

`random` is an Inform routine used to generate random numbers or to choose randomly between given choices; in the form `random(`*`expression`*`)` it returns a random number between 1 and *`expression`* inclusive. So our condition is actually stating: if a random choice between 1 and 2 happens to be 1 then perform some action. Remember that a daemon is run once at the end of every turn, so the condition is trying to squeeze a comment from a customer roughly once every other turn.

Next, we proceed as we have already seen in "William Tell", with a `switch` statement to order the comments in a controlled sequence by cunning use of a customised local property, `number_of_comments`. We have written just five messages (could have been one or a hundred) and then we reach the `default` case, which is a good place to stop the daemon, since we have no more customers' witticisms to display.

Ah, but when does the daemon *start* functioning? Well, as soon as our protagonist comes out of the toilet dressed in his multicoloured super-hero pyjamas. Since we want to minimise the possible game states, we'll make some general rules to avoid trouble: (a) players will be able to change only in the toilet; (b) we won't let players change back into street clothes; and (c) once players manage to step into the street thus dressed, the game is won. So, we can safely assume that if players enter the café in their Captain's outfit, they'll be coming from the toilet. As a consequence of all this, we can change the café room `description` to:

TYPE

```
Room    cafe "Inside Benny's cafe"
  with  description [;
            print "Benny's offers the FINEST selection of pastries and
                sandwiches. Customers clog the counter, where Benny himself
                manages to serve, cook and charge without missing a step. At
                the north side of the cafe you can see a red door connecting
                with the toilet.";
            if (costume has worn && self.first_time_out == false) {
                self.first_time_out = true;
                StartDaemon(customers);
                print "^^Nearby customers glance at your costume with open
                    curiosity.";
            }
            new_line;
        ],
        first_time_out false,           ! Captain Fate's first appearance?
        ...
```

This routine always displays the first string "Benny's offers the finest..." and then checks whether the player character is wearing the costume, in which case it starts the daemon of the `customers` object and displays another message: "^^Nearby customers...". The use of the local `first_time_out` property ensures that the condition is true only once, so the statement block attached to it runs also once. Finally, we find the `new_line` statement. This just outputs a carriage return at the end of our routine – in fact, it works exactly the same as `print "^"`; since we have two `print` statements and one of them will be displayed only once, the carriage return tidies the text layout for both situations:

```
Inside Benny's cafe
Benny's offers the FINEST selection of pastries and sandwiches. Customers clog
the counter, where Benny himself manages to serve, cook and charge without
missing a step. At the north side of the cafe you can see a red door connecting
with the toilet.                                  <- NEW_LINE PRINTS HERE...

>

Inside Benny's cafe
Benny's offers the FINEST selection of pastries and sandwiches. Customers clog
the counter, where Benny himself manages to serve, cook and charge without
missing a step. At the north side of the cafe you can see a red door connecting
with the toilet.

Nearby customers glance at your costume with open curiosity.  <- ...AND HERE

>
```

We've finished with the customers in the café. Now, we have the toilet to the north which, for reasons of gameplay *and* decency, is protected by a door.

A door to adore

Door objects require some specific properties and attributes. Let's first code a simple door:

```
Object  toilet_door "toilet door" cafe
  with  name 'red' 'toilet' 'door',
        description
            "A red door with the unequivocal black man-woman
             silhouettes marking the entrance to hygienic facilities.
             There is a scribbled note stuck on its surface.",
        door_dir n_to,
        door_to toilet,
        with_key toilet_key,
  has   scenery door openable lockable locked;
```

We find this door in the café. We must specify the direction in which the door leads and, as we have mentioned in the café's description, that would be to the north. That's what the `door_dir` property is for, and in this case it takes the value of the north direction property `n_to`. Then we must tell Inform the identity of the room to be found behind the door, hence the `door_to` property, which takes the value of the toilet room – to be defined later. Remember the café's connection to the north, `n_to toilet_door`? Thanks to it, Inform will know that the door is in the way, and thanks to the `door_to` property, what lies beyond.

Doors *must* have the attribute `door`, but beyond that we have a stock of options to help us define exactly what kind of door we are dealing with. As for containers, doors can be `openable` (which activates the verbs OPEN and CLOSE so that they can be applied to this object) and, since by default they are closed, you can give them the attribute `open` if you wish otherwise. Additionally, doors can be `lockable` (which sets up the LOCK/UNLOCK verbs) and you can make them `locked` to override their default unlocked status. The verbs LOCK and UNLOCK are

expecting some kind of key object to operate the door. This must be defined using the `with_key` property, whose value should be the internal ID of the key; in our example, the soon-to-be-defined `toilet_key`. If you don't supply this property, players won't be able to lock or unlock the door.

This simple door definition has one problem, namely, that it exists only in the café room. If you wish the door to be present also from the toilet side, you can either (a) define another door to be found in the `toilet` room, or (b) make this one a two-sided door.

Solution (a) seems superficially straightforward, but then you have the problem of keeping the states of the two doors – open/closed, locked/unlocked – in synch. In this scenario, where you can access the toilet only through this door, that wouldn't be too complicated, since you could leave the door object in the café room opened all the time, regardless of what players do with the door object in the toilet room and vice versa – they are never going to see them at the same time. In general terms, though, such inconsistencies lead to problems; solution (a) is best ignored for most purposes.

Solution (b) is better, since you have only one door object to deal with and its possible states affect both sides. However, the coding gets a little bit complicated and you'll have to define routines for most properties:

TYPE

```
Object  toilet_door "toilet door"
  with  name 'red' 'toilet' 'door',
        description [;
            if (location == cafe)
                "A red door with the unequivocal black man-woman silhouettes
                 marking the entrance to hygienic facilities. There is a
                 scribbled note stuck on its surface.";
            else
                "A red door with no OUTSTANDING features.";
        ],
        found_in cafe toilet,
        door_dir [;
            if (location == cafe) return n_to;
            else                  return s_to;
        ],
        door_to [;
            if (location == cafe) return toilet;
            else                  return cafe;
        ],
        with_key toilet_key,
  has   scenery door openable lockable locked;
```

First of all, the door now needs a `found_in` property, since it's going to be located both in the café and the toilet. The `description` checks which side of the door we are looking at – testing the current value of the variable `location`, which holds the room the player is in – because we have a scribbled note stuck on one side, but not on the other. And the `door_dir` and `door_to` properties must use the same trick, because we travel north from the café into the toilet, but south from the toilet into the café.

Right now, the game will display "the toilet door" every time it needs to refer to this object. It would be nice if we could somehow get the game to distinguish between "the door to the toilet" and "the door to the cafe", depending on the side we are facing. For this, a `short_name` property is the thing. We have already talked about the external name defined as part of an object's header information:

```
Object  toilet_door "toilet door"
```

That "toilet door" will be the name displayed by the game at run-time to refer to the door. With identical effect, this could also have been coded thus:

```
Object  toilet_door
  with  short_name "toilet door",
```

`short_name` is a property that supplies the external name of an object, either as a string or an embedded routine. Normally, objects retain the same external name throughout the game – and the header information method is perfect in that case – but if it needs to change, it's easy to write a routine as the value of `short_name`:

TYPE

```
Object  toilet_door
  with  name 'red' 'toilet' 'door',
        short_name [;
            if (location == cafe) print "door to the toilet";
            else                  print "door to the cafe";
            return true;
        ],
        description
            ...
```

Notice the `return true` at the end of the routine. You'll recall that the standard rule says "return false to carry on, true to take over and stop normal execution". In the case of `short_name`, "carry on" means "and now display the external name from the header information", which is sometimes handy; for instance, you could write a `short_name` routine to prefix an object's external name with one of a range of adjectives – perhaps a shining/flickering/fading/useless lantern.

> NOTE: what's displayed if there isn't an external name in an object's header? If you've read the section "Compile-as-you-go" on page 208, you'll recall that the interpreter simply uses the internal identifier within parentheses; that is, with no external name and no `short_name` property, we might see:
>
> ```
> You open the (toilet_door).
> ```
>
> And the same principle applies if we were mistakenly to `return false` from this `short_name` routine: we would get, first, the result of our `print` statement, and then the standard rules would display the internal ID:
>
> ```
> You open the door to the toilet(toilet_door).
> ```

Doors can get more complicated than this (no, please, don't throw our guide out of the window). Here comes some optional deluxe coding to make the door object a bit friendlier in game play, so you can skip it if you foresee headaches.

Our door now behaves nicely at run-time. It can be locked and unlocked if the player character has the right key; it can be opened and closed. A sequence of commands to go into the toilet and lock the door behind you would be: UNLOCK DOOR WITH KEY, OPEN DOOR, GO NORTH, CLOSE DOOR, LOCK DOOR WITH KEY. After we are finished, let's go back to the café: UNLOCK DOOR WITH KEY, OPEN DOOR, SOUTH. If the player is of the fastidious kind: CLOSE DOOR, LOCK DOOR WITH KEY. This game features only one door, but if it had three or four of them, players would grow restless (at the very least) if they needed to type so many commands just to go through a door. This is the kind of thing reportedly considered as poor design, because the game is suddenly slowed down to get over a simple action which involves no secrets or surprises. How exciting can the crossing of an ordinary door be, after all?

If a few lines of code can make the life of the player easier, it's worth a shot. Let's provide a few improvements to our toilet door in `before` and `after` properties:

```
before [ ks;
    Open:
        if (self hasnt locked || toilet_key notin player)
            return false;
        ks = keep_silent; keep_silent = true;
        <Unlock self toilet_key>; keep_silent = ks;
        return true;
    Lock:
        if (self hasnt open) return false;
        print "(first closing ", (the) self, ")^";
        ks = keep_silent; keep_silent = true;
        <Close self>; keep_silent = ks;
        return false;
],
after [ ks;
    Unlock:
        if (self has locked) return false;
        print "You unlock ", (the) self, " and open it.^";
        ks = keep_silent; keep_silent = true;
        <Open self>; keep_silent = ks;
        return true;
],
```

The basic idea here is to let the player who holds the key perform just one action to both unlock *and* open the door (and, conversely, to close *and* lock it). The relevant actions are `Unlock` and `Open`, and `Lock` (`Close` is not necessary; if players just close the door we shouldn't assume that they want to lock it as well).

- **Open**: if the door isn't locked or the player doesn't hold the key, keep going with the default `Open` action defined by the library. That leaves a locked door and a player holding the key, so we redirect processing to the `Unlock` action, giving as arguments the door (self) and the toilet key. Since we are using single angle-brackets `<...>`, the action resumes after the unlocking is done (note that the `Unlock` action also takes care of opening the door). Finally, we `return true` to stop the library from trying to open the door by itself.

- **Lock**: if the door is already closed, keep going with the standard library `Lock` action. If not, tell players that we are closing the door for them, redirect the action briefly to actually close it, and then `return false` to let the `Lock` action proceed as before.
- **Unlock**: we place this action in the `after` property, so (let's hope) the `Unlock` action has already happened. If the door is still locked, something went wrong, so we `return false` to display the standard message for an unsuccessful unlocking. Otherwise, the door is now unlocked, so we inform the player that we are opening the door and redirect the action to actually open it, returning `true` to suppress the standard message.

In all processes there is a library variable called `keep_silent`, which can be either `false` (the normal state) or `true`; when `true`, the interpreter does not display the associated message of an action in progress, so we can avoid things like:

```
>OPEN DOOR
You open the door to the toilet.
You unlock the door to the toilet and open it.
```

Although we want to set `keep_silent` to `true` for the duration of our extra processing, we need to reset it afterwards. In a case like this, good design practice is to preserve its initial value (which was probably `false`, but you should avoid risky assumptions); we use a local variable `ks` to remember that initial setting so that we can safely restore it afterwards. You'll remember that a local variable in a standalone routine is declared between the routine's name and the semicolon:

```
[ BeenToBefore this_room;
```

In exactly the same way, a local variable in an embedded routine is declared between the `[` starting marker of the routine and the semicolon:

```
before [ ks;
```

You can declare up to fifteen variables this way – just separated by spaces – which are usable only within the embedded routine. When we assign it thus:

```
ks = keep_silent;
```

we are actually making `ks` equal to whatever value `keep_silent` has (either `true` or `false`; we actually don't care). We then set `keep_silent` to `true`, make the desired silent actions, and we assign:

```
keep_silent = ks;
```

which restores the value originally stored in `ks` to `keep_silent`. The effect is that we manage to leave it as it was before we tampered with it.

Well, that's about everything about doors. Everything? Well, no, not really; any object can grow as complex as your imagination allows, but we'll drop the subject here. If you care to see more sophisticated doors, check Exercises 3 and 4 in the *Inform Designer's Manual*, where an obliging door opens and unlocks by itself if the player simply walks in its direction.

So far, we have the player in front of a locked door leading to the toilet. A dead end? No, the description mentions a scribbled note on its surface. This one should offer no problem:

TYPE

```
Object  "scribbled note" cafe
  with  name 'scribbled' 'note',
        description [;
            if (self.read_once == false) {
                self.read_once = true;
                "You apply your ENHANCED ULTRAFREQUENCY vision to the note
                 and squint in concentration, giving up only when you see the
                 borders of the note begin to blacken under the incredible
                 intensity of your burning stare. You reflect once more how
                 helpful it would've been if you'd ever learnt to read.
                 ^^A kind old lady passes by and explains:
                 ~You have to ask Benny for the key, at the counter.~^^
                 You turn quickly and begin, ~Oh, I KNOW that, but...~^^
                 ~My pleasure, son,~ says the lady, as she exits the cafe.";
            }
            else
                "The scorched undecipherable note holds no SECRETS from
                 you NOW! Ha!";
        ],
        read_once false,                ! has the player read the note once?
        before [; Take:
            "No reason to start collecting UNDECIPHERABLE notes.";
        ],
  has   scenery;
```

Just notice how we change the description after the first time the player examines the note, using the local property `read_once` created just for this purpose. We don't want the player to walk off with the note, so we intercept the `Take` action and display something more in character than the default message for `scenery` objects: "That's hardly portable".

We've talked a lot about the toilet key; it seems about time to code it. Originally, the key is in Benny's possession, and the player will have to ask for it, just as the note explains. Although we'll define Benny in detail throughout the next chapter, here we present a basic definition, largely so that the key has a parent object.

TYPE

```
Object  benny "Benny" cafe
  with  name 'benny',
        description
            "A deceptively FAT man of uncanny agility, Benny entertains his
             customers crushing coconuts against his forehead when the mood
             strikes him.",
  has   scenery animate male proper transparent;
```

TYPE

```
Object  toilet_key "toilet key" benny
  with  name 'toilet' 'key',
        article "the",
        invent [;
            if (clothes has worn) print "the CRUCIAL key";
            else                  print "the used and IRRELEVANT key";
            return true;
        ],
        description
            "Your SUPRA PERCEPTIVE senses detect nothing of consequence
             about the toilet key.",
        before [;
            if (self in benny)
                "You SCAN your surroundings with ENHANCED AWARENESS,
                 but fail to detect any key.";
        ];
```

While Benny has the key, there's logically no way to examine it (or perform any other action involving it), but we want to prevent the interpreter from objecting that "You can't see any such thing". We've made the `toilet_key` a child of the `benny` object, and you can see that Benny's got a `transparent` attribute; this means that the key is in scope, and enables the player to refer to it without the interpreter complaining. Because Benny also has an `animate` attribute, the interpreter would normally intercept a TAKE KEY action with "That seems to belong to Benny"; however, the same wouldn't apply to other commands like TOUCH KEY and TASTE KEY. So, to prevent any interaction with the key while it's in Benny's pockets, we define a `before` property.

```
before [;
    if (self in benny)
        "You SCAN your surroundings with ENHANCED AWARENESS,
         but fail to detect any key.";
];
```

All of the `before` properties that we've so far created have contained one or more labels specifying the actions which they are to intercept; you'll remember that in "William Tell" we introduced the `default` action (see "A class for props" on page 66) to mean "any value not already catered for". Here, though, things are simpler: because we wish to intercept everything we can dispense with the labels altogether; our code will then be executed at the start of *every* action directed at the key. If it's still in Benny's possession, we display a polite refusal; otherwise, the action continues unhindered.

Another small innovation here: the `invent` library property (we didn't make it up) which enables you to control how objects appear in inventory listings, overriding the default. Left to itself, the interpreter simply displays the object's external name, preceded either by a standard article like "a" or "some", or one specifically defined in the object's `article` property. Here we replace "the toilet key" with one of two more helpful descriptions, making it a most valuable object in the eyes of John Covarth, and something to be despised haughtily by Captain Fate once it's of no further use to him.

When we had players in the street, we faced the problem that they might choose to examine the café from the outside. While it's unlikely that they'll try to examine the toilet room from the outside, it takes very little effort to offer a sensible output just in case:

TYPE

```
Object  outside_of_toilet "toilet" cafe
  with  name 'toilet' 'bath' 'rest' 'room' 'bathroom' 'restroom',
        before [;
            Enter:
                if (toilet_door has open) {
                    PlayerTo(toilet);
                    return true;
                }
                else
                    "Your SUPERB deductive mind detects that the DOOR is
                     CLOSED.";
            Examine:
                if (toilet_door has open)
                    "A brilliant thought flashes through your SUPERLATIVE
                     brain: detailed examination of the toilet would be
                     EXTREMELY facilitated if you entered it.";
                else
                    "With a TREMENDOUS effort of will, you summon your
                     unfathomable ASTRAL VISION and project it FORWARD
                     towards the closed door... until you remember that it's
                     Dr Mystere who's the one with mystic powers.";
            Open:  <<Open  toilet_door>>;
            Close: <<Close toilet_door>>;
            Take,Push,Pull: "That would be PART of the building.";
        ],
  has   scenery openable enterable;
```

As with the `outside_of_cafe` object, we intercept an `Enter` action, to teleport players into the toilet room if they type ENTER TOILET (or to display a refusal if the toilet door is closed). Players may try to EXAMINE TOILET; they'll get a different message if the door is open – we invite them to enter it – or if it's closed. OPEN TOILET and CLOSE TOILET inputs are redirected to `Open` and `Close` actions for the toilet door; remember that the double angle-brackets imply a `return true`, so that the action stops there and the interpreter does not attempt to `Open` or `Close` the `outside_of_toilet` object itself after it has dealt with the door.

You're right: the toilet looms large in this game (we blame it on early maternal influences). We've introduced an ambiguity problem with the `outside_of_toilet` object, and we'll need some help in fixing it.

12 • Captain Fate: take 3

W was a watchman, and guarded the door;
X was expensive, and so became poor.

e've given ourselves an interesting challenge by overusing that convenient word "toilet", and here we show you how we resolve the ambiguities that have been introduced. Also, it's time for the eponymous owner of Benny's café to be developed in full.

Too many toilets

If you check the `name` properties of the toilet door, the toilet key and the toilet room, you'll see that the dictionary word `'toilet'` occurs in all of them. There won't be any problems if players mention the words DOOR or KEY, but we reach a strange impasse should they try to perform some action with just the word TOILET. The interpreter has to think fast: is the player talking about the key? About the door? Or about the toilet? Unable to decide, it asks: "Which do you mean, the door to the toilet, the toilet key or the toilet?"

And guess what? Players will never be able to refer to the toilet object (unless they type BATH ROOM or REST ROOM, not an obvious choice since we haven't used those phrases anywhere visible). If the player answers TOILET the parser will still have three objects with that dictionary word as a possible name, so it will ask again, and again – until we give it some dictionary word which is not ambiguous. A human reader would be able to understand that the word TOILET alone refers to the room, but the interpreter won't – unless we help it a little.

We could work around this problem in more than one way, but we'll take this opportunity of demonstrating the use of a third-party library package.

When experienced designers find a problem which is not easily solvable, they may come up with a smart solution and then consider that others could benefit from the effort. The product of this generosity takes the form of a library extension: the solution neatly packaged as a file that other designers can incorporate into their source code. These files can be found in the IF Archive: go to `http://www.ifarchive.org/indexes/if-archive.html` and then select "`.../infocom`", "`.../compilers`", "`.../inform6`", "`.../library`", and "`.../contributions`". All of these files contain Inform code. To use a library extension (also known as a library contribution), you should download it and read the instructions (usually embedded as comments in the file, but occasionally supplied separately) to discover what to do next. Normally, you `Include` it (as we have already done with `Parser`, `VerbLib` and `Grammar`), but often there are rules about where exactly this `Include` should be placed in your source code. It is not unusual to find other suggestions and warnings.

To help us out of the disambiguation problem with the word TOILET, we are going to use Neil Cerutti's extension `pname.h`, which is designed for situations precisely like this. First, we follow the link to the IF archive and download the compressed file `pname.zip`, which contains two more files: `pname.h` and `pname.txt`. We place these files in the folder where we are currently developing our game or, if using the environment we proposed in "Tools of the trade" on page 17, in the `Inform\Lib\Contrib` folder. The text file offers instructions about installation and usage. Here we find a warning:

> This version of pname.h is recommended for use only with version 6/10 of the Inform Library.

That's what we are currently using, so there's no problem. Most extensions aren't this fussy, but `pname.h` fiddles with some routines at the heart of the standard library; these may not be identical in other Inform versions.

The introduction explains what `pname.h` does for you; namely, it lets you avoid using complicated `parse_name` routines to disambiguate the player's input when the same dictionary word refers to more than one item. A `parse_name` routine would have been the solution to our problem before the existence of this file, and it qualifies as an advanced programming topic, difficult to master on a first approach. Fortunately, we don't need to worry. Neil Cerutti explains:

> The `pname.h` package defines a new object property, `pname` (short for phrase name), with a similar look and feel to the standard `name` property: both contain a list of dictionary words. However, in a `pname` property the order of the words is significant, and special operators `'.p'` `'.or'` and `'.x'` enable you to embed some intelligence into the list. In most cases where the standard `name` property isn't enough, you can now just replace it with a `pname` property, rather than write a `parse_name` property routine.

We'll soon see how it works. Let's take a look at the installation instructions:

> To incorporate this package into your program, do three things:
>
> 1. Add four lines near the head of the program (before you include `Parser.h`).
>
> ```
> Replace MakeMatch;
> Replace Identical;
> Replace NounDomain;
> Replace TryGivenObject;
> ```
>
> 2. Include the `pname.h` header just after you include `Parser.h`.
>
> ```
> Include "Parser";
> Include "pname";
> ```
>
> 3. Add `pname` properties to those objects which require phrase recognition.

It seems simple enough. So, following steps one and two, we add those `Replace...` lines before the inclusion of `Parser`, and we include `pname.h` right after it. `Replace` tells the compiler that we're providing replacements for some standard routines.

```
Constant Story "Captain Fate";
Constant Headline
            "^A simple Inform example
             ^by Roger Firth and Sonja Kesserich.^";
Release 2;  Serial "020827";    ! for keeping track of public releases

Constant MANUAL_PRONOUNS;

Replace MakeMatch;              ! required by pname.h
Replace Identical;
Replace NounDomain;
Replace TryGivenObject;

Include "Parser";
Include "pname";
...
```

Now our source code is ready to benefit from the library package. How does it work? We have acquired a new property – `pname` – which can be added to some of our objects, and which works pretty much like a `name` property. In fact, it should be used *instead* of a `name` property where we have a disambiguation problem. Let's change the relevant lines for the toilet door and the toilet key:

```
Object  toilet_door
  with  pname '.x' 'red' '.x' 'toilet' 'door',
        short_name [;
        ...

Object  toilet_key "toilet key" benny
  with  pname '.x' 'toilet' 'key',
        article "the",
        ...
```

while leaving the `outside_of_toilet` unchanged:

```
Object  outside_of_toilet "toilet" cafe
  with  name 'toilet' 'bath' 'rest' 'room' 'bathroom' 'restroom',
        before [;
        ...
```

We are now using a new operator – `'.x '` – in our `pname` word lists. The text file explains

> The first dictionary word to the right of a `'.x'` operator is interpreted as optional.

and this makes the dictionary word `'toilet'` of lesser importance for these objects, so that at run-time players could refer to the DOOR or TOILET DOOR or the KEY or TOILET KEY – but not simply to the TOILET – when referring to either the door or the key. And, by leaving unchanged the `name` property of the `outside_of_toilet` object – where there is also another `'toilet'` entry – the `pname` properties will tell the interpreter to discard the key and the door as possible objects to be considered when players refer just to TOILET. Looking at it in terms of the English language, we've effectively said that "TOILET" is an adjective in the phrases "TOILET DOOR" and "TOILET KEY", but a noun when used on its own to refer to the room.

The `pname.h` package has additional functionality to deal with more complex phrases, but we don't need it in our example game. Feel free, however, to read `pname.txt` and discover what this fine library extension can do for you: it's an easy answer to many a disambiguation headache.

Don't shoot! I'm only the barman

A lot of the action of the game happens around Benny, and his definition needs a little care. Let's explain what we want to happen.

> So the door is locked and the player, after discovering what the note stuck on the toilet door said, will eventually ask Benny for the key. Sadly, Benny allows use of the toilet only to customers, a remark he'll make looking pointedly at the menu board behind him. The player will have to ask for a coffee first, thereby qualifying as a customer in Benny's eyes and thus entitled to make use of the toilet. At last! Rush inside, change into Captain Fate's costume and fly away to save the day!

Except that the player neither paid for the coffee, nor returned the toilet key. Benny will have to stop the player from leaving the café in these circumstances. To prevent unnecessary complication, there will be a coin near the lavatory, enough cash to pay for the coffee. And that about sums it all up; pretty simple to describe – not so simple to code. Remember Benny's basic definition from the previous chapter:

```
Object  benny "Benny" cafe
  with  name 'benny',
        description
            "A deceptively FAT man of uncanny agility, Benny entertains his
             customers crushing coconuts against his forehead when the mood
             strikes him.",
  has   scenery animate male proper transparent;
```

We can now add some complexity, beginning with a `life` property. In generic form:

```
life [;
    Give:             ... code for giving objects to Benny
    Attack:           ... code to deal with player's aggressive moves
    Kiss:             ... code about the player getting tender on Benny
    Ask,Tell,Answer: ... code to handle conversation
],
```

We have seen some of these actions before. We'll take care of the easier ones:

```
Attack:
    if (costume has worn) {
        deadflag = 4;
        print "Before the horror-stricken eyes of the surrounding
            people, you MAGNIFICENTLY jump OVER the counter and
            attack Benny with REMARKABLE, albeit NOT sufficient,
            speed. Benny receives you with a TREACHEROUS
            upper-cut that sends your GRANITE JAW flying through
            the cafe.^^
            ~These guys in pyjamas think they can bully innocent
            folk,~ snorts Benny, as the EERIE hands of DARKNESS
            engulf your vision and you lose consciousness.";
    }
    else
        "That would be an unlikely act for MEEK John Covarth.";
Kiss:   "This is no time for MINDLESS infatuation.";
Ask,Tell,Answer:
    "Benny is too busy for idle chit-chat.";
```

Attacking Benny is not wise. If the player is still dressed as John Covarth, the game displays a message refusing to use violence by reason of staying in character as a worthless wimp. However, if Captain Fate attempts the action, we'll find that there is more to Benny than meets the eye, and the game is lost. Kissing and conversation are disallowed by a couple of tailored responses.

The `Give` action is a bit more complicated, since Benny reacts to certain objects in a special and significant way. Bear in mind that Benny's definition needs to keep track of whether the player has asked for a coffee (thereby becoming a customer and thus worthy of the key), whether the coffee has been paid for, and whether the toilet key has been returned. The solution, yet again (this really is a most useful capability), is more local property variables:

```
Object  benny "Benny" cafe
  with  name 'benny',
        description
            "A deceptively FAT man of uncanny agility, Benny entertains his
             customers crushing coconuts against his forehead when the mood
             strikes him.",
        coffee_asked_for false,          ! has player asked for a coffee?
        coffee_not_paid  false,          ! is Benny waiting to be paid?
        key_not_returned false,          ! is Benny waiting for the key?
        life [;
        ...
```

Now we are ready to tackle the `Give` action of the `life` property, which deals with commands like GIVE THE KEY TO BENNY (in a moment, we'll come to the `Give` action of the `orders` property, which deals with commands like BENNY, GIVE ME THE KEY):

TYPE

```
Give: switch (noun) {
    clothes:
        "You NEED your unpretentious John Covarth clothes.";
    costume:
        "You NEED your stupendous ACID-PROTECTIVE suit.";
    toilet_key:
        self.key_not_returned = false;
        move toilet_key to benny;
        "Benny nods as you ADMIRABLY return his key.";
    coin:
        remove coin;
        self.coffee_not_paid = false;
        "With marvellous ILLUSIONIST gestures, you produce the
         coin from the depths of your BULLET-PROOF costume as if
         it had popped out from Benny's ear! People around you
         clap politely.  Benny accepts the coin and gives it a
         SUSPICIOUS bite. ~Thank you, sir. Come back anytime,~
         he says.";
}
```

The `Give` action in the `life` property holds the variable `noun` as the object offered to the NPC. Remember that we can use the `switch` statement as shorthand for:

```
if (noun == costume) { whatever };
if (noun == clothes) { whatever };
...
```

We won't let players give away their clothes or their costume (yes, an improbable action, but you never know). The toilet key and the coin are successfully transferred. The property `key_not_returned` will be set to `true` when we receive the toilet key from Benny (we have not coded that bit yet), and now, when we give it back, it's reset to `false`. The `move` statement is in charge of the actual transfer of the object from the player's inventory to Benny, and we finally display a confirmation message. With the coin, we find a new statement: `remove`. This extracts the object from the object tree, so that it now has no parent. The effect is to make it disappear from the game (though you are not destroying the object permanently – and indeed you could return it to the object tree using the `move` statement); as far as the player is concerned, there isn't a COIN to be found anywhere. The `coffee_not_paid` property will be set to `true` when Benny serves us the cup of coffee (again, we'll see that in a moment); now we reset it to `false`, which liberates the player from debt. This culminates with the `"..."` print-and-return statement, telling the player that the action was successful.

Why move the key to Benny but remove the coin instead? Once players qualify as customers by ordering a coffee, they will be able to ask for the key and return it as many times as they like, so it seems sensible to keep the key around. The coin, however, will be a one-shot. We won't let players ask for more than one coffee, to prevent their debt from growing ad infinitum – besides, they came in here to change, not to indulge in caffeine. Once the coin is paid, it disappears for good, supposedly into Benny's greedy pockets. No need to worry about it any more.

The `benny` object needs also an `orders` property, just to take care of the player's requests for coffee and the key, and to fend off any other demands. The `Give` action in an `orders` property deals with inputs like ASK BENNY FOR THE KEY or BENNY, GIVE ME THE KEY. The syntax is similar to that of the `life` property:

```
orders [;   ! handles ASK BENNY FOR X and BENNY, GIVE ME XXX
    Give: switch (noun) {
        toilet_key:
            if (toilet_key in player)
                "But you DO have the key already.";
            if (self.coffee_asked_for == true) {
                move toilet_key to player;
                self.key_not_returned = true;
                "Benny tosses the key to the rest rooms on the
                 counter, where you grab it with a dextrous and
                 precise movement of your HYPER-AGILE hand.";
            }
            else
                "~Toilet is only fer customers,~ he grumbles,
                 looking pointedly at a menu board behind him.";
        coffee:
            if (self.coffee_asked_for == true)
                "One coffee should be enough.";
            move coffee to counter;
            self.coffee_asked_for = true;
            self.coffee_not_paid = true;
            "With two gracious steps, Benny places his world-famous
             Cappuccino in front of you.";
        food:
            "Food will take too much time, and you must change NOW.";
        menu:
            "With only the smallest sigh, Benny nods towards the menu
             on the wall behind him.";
        default:
            "~I don't think that's on the menu, sir.~";
    }
],
```

- **Toilet key**: first, we check whether players already have the key or not, and complain if they do, stopping execution thanks to the implicit `return true` of the "`...`" statement. If players don't have the key, we proceed to check whether they've asked for a coffee yet, by testing the `coffee_asked_for` property. If this is `true`, they get the key, which means that they have to return it – the `key_not_returned` property becomes `true` – and we display a suitable message. If this is not `true` (the `else` clause, which pairs up with the nearest `if` statement) Benny refuses to oblige, mentioning for the first time the menu board where players will be able to see a picture of a cup of coffee when they EXAMINE it.

- **Coffee**: we check whether players have already asked for a coffee, by testing the `coffee_asked_for` property, and refuse to serve another one if `true`. If `false`, we place the coffee on the counter, and set the properties `coffee_asked_for` and `coffee_not_paid` to true. The message bit you know about.

- **Food**: we'll provide an object to deal with all of the delicious comestibles to be found in the café, specifically those (such as "pastries and sandwiches") mentioned in our descriptions. Although that object is not yet defined, we code ahead to thwart player's gluttony in case they choose to ask Benny for food.
- **Menu**: our default response – "I don't think that's on the menu, sir" – isn't very appropriate if the player asks for a menu, so we provide a better one.
- **Default**: this takes care of anything else that the player asks Benny for, displaying his curt response.

And before you know it, Benny's object is out of the way; however, don't celebrate too soon. There's still some Benny-related behaviour that, curiously enough, doesn't happen in Benny's object; we're talking about Benny's reaction if the player tries to leave without paying or returning the key. We promised you that Benny would stop the player, and indeed he will. But where?

We must revisit the café room object:

TYPE

```
Room    cafe "Inside Benny's cafe"
  with  description [;
            print "Benny's offers the FINEST selection of pastries and
                sandwiches. Customers clog the counter, where Benny himself
                manages to serve, cook and charge without missing a step. At
                the north side of the cafe you can see a red door connecting
                with the toilet.";
            if (costume has worn && self.first_time_out == false) {
                self.first_time_out = true;
                StartDaemon(customers);
                print "^^Nearby customers glance at your costume with open
                    curiosity.";
            }
            new_line;
        ],
        first_time_out false,           ! Captain Fate's first appearance?
        before [; Go: if (noun ~= s_obj) return false;
            if (benny.coffee_not_paid == true ||
                benny.key_not_returned == true) {
                print "Just as you are stepping into the street, the big hand
                    of Benny falls on your shoulder.";
                if (benny.coffee_not_paid == true &&
                    benny.key_not_returned == true)
                    "^^~Hey! You've got my key and haven't paid for the
                     coffee. Do I look like a chump?~ You apologise as only a
                     HERO knows how to do and return inside.";
                if (benny.coffee_not_paid == true)
                    "^^~Just waidda minute here, Mister,~ he says.
                     ~Sneaking out without paying, are you?~ You quickly
                     mumble an excuse and go back into the cafe. Benny
                     returns to his chores with a mistrusting eye.";
                if (benny.key_not_returned == true)
                    "^^~Just where you think you're going with the toilet
                     key?~ he says. ~You a thief?~ As Benny forces you back
                     into the cafe, you quickly assure him that it was only
                     a STUPEFYING mistake.";
```

```
        }
        if (costume has worn) {
            deadflag = 5;              ! you win!
            "You step onto the sidewalk, where the passing pedestrians
             recognise the rainbow EXTRAVAGANZA of Captain FATE's costume
             and cry your name in awe as you JUMP with sensational
             momentum into the BLUE morning skies!";
            }
    ],
    s_to street,
    n_to toilet_door;
```

Once again, we find that the solution to a design problem is not necessarily unique. Remember what we saw when dealing with the player's description: we could have assigned a new value to the `player.description` variable, but opted to use the `LibraryMessages` object instead. This is a similar case. The code causing Benny to intercept the forgetful player could have been added, perhaps, to a `daemon` property in Benny's definition. However, since the action to be intercepted is always the same one and happens to be a movement action when the player tries to leave the café room, it is also possible to code it by trapping the `Go` action of the room object. Both would have been right, but this is somewhat simpler.

We have added a `before` property to the room object (albeit a longish one), just dealing with the `Go` action. This technique lets you trap the player who is about to exit a room before the movement actually takes place, a good moment to interfere if we want to prevent escape. The first line:

```
if (noun ~= s_obj) return false;
```

is telling the interpreter that we want to tamper only with southwards movement, allowing the interpreter to apply normal rules for the other available directions; the `~=` operator stands for "not equal to". From here on, it's only conditions and more conditions. The player may attempt to leave:

- without paying for the coffee *and* without returning the key,
- having paid for the coffee, but without returning the key,
- having returned the key, but not paid for the coffee, or
- free of sin and accountable for nothing in the eyes of all men (well, in the eye of Benny, at least).

The first three are covered by the test:

```
if (benny.coffee_not_paid == true || benny.key_not_returned == true) ...
```

that is, if either the coffee is not paid for *or* if the key is not returned. When this condition is `false`, it means that both misdemeanours have been avoided and that the player is free to go. However, when this condition is `true`, the hand of Benny falls on the player's shoulder and then the game displays a different message according to which fault or faults the player has committed.

If the player is free to go, *and* is wearing the crime-fighting costume, the game is won. We tell you how that's reported in the next chapter, where we finish off the design.

13 • Captain Fate: the final cut

Y was a youth, that did not love school;
Z was a zany, a poor harmless fool.

ou'll probably be pleased to hear that Captain Fate has almost run his allotted span. There are some minor objects still to be defined – the toilet, our hero's clothes, the all-important costume – but first we need to decorate the café a little more.

Additional catering garnish

We must not forget a couple of tiny details in the café room:

```
Object  food "Benny's snacks" cafe
  with  name 'food' 'pastry' 'pastries' 'sandwich' 'sandwiches' 'snack'
            'snacks' 'doughnut',
        before [; "There is no time for FOOD right now."; ],
  has   scenery proper;

Object  menu "menu" cafe
  with  name 'informative' 'menu' 'board' 'picture' 'writing',
        description
            "The menu board lists Benny's food and drinks, along with their
             prices. Too bad you've never learnt how to read, but luckily
             there is a picture of a big cup of coffee among the
             incomprehensible writing.",
        before [; Take:
            "The board is mounted on the wall behind Benny. Besides, it's
             useless WRITING.";
        ]
  has   scenery;
```

And a not-so-trivial object:

```
Object  coffee "cup of coffee" benny
  with  name 'cup' 'of' 'coffee' 'steaming' 'cappuccino'
            'cappucino' 'capuccino' 'capucino',
        initial "On the counter, the steaming Cappuccino awaits you.",
        description [;
            if (self in benny)
                "The picture on the menu board SURE looks good.";
            else
                "It smells delicious.";
        ],
        before [;
            Take,Drink,Taste:
                if (self in benny)
                    "You should ask Benny for one first.";
                else {
                    move self to benny;
                    "You pick up the cup and swallow a mouthful. Benny's
                     WORLDWIDE REPUTATION is well deserved. Just as you
```

```
                  finish, Benny takes away the empty cup.
                  ~That will be one quidbuck, sir.~";
             }
         Buy:
             if (coin in player) <<Give coin benny>>;
             else                "You have no money.";
         Smell:
             "If your HYPERACTIVE pituitary glands are to be trusted,
              it's Colombian.";
     ];
```

There's nothing really new in this object (other than that the `name` property caters for orthographically challenged players), but notice how we don't `remove` it after the player drinks it. In an apparently absurd whim, the coffee returns to Benny magically (although this is not information that the player needs to know). Why? After you remove an object from the game, if the player attempts, say, to EXAMINE it, the interpreter will impertinently state that "You can't see any such thing". Moreover, if the player asks Benny for a second coffee, once the first one has been removed, Benny will complain "I don't think that's on the menu, sir" – a blatant lie – which was the default in Benny's `orders` property. Since the removed coffee object does not belong to Benny, it's not a noun that the player can ASK Benny FOR. By making it a child of the barman (who has the `transparent` attribute set), the coffee is still an object that players can refer to. We ensure that they don't get more cups thanks to Benny's `coffee_asked_for` property, which will remain true after the first time.

Toilet or dressing room?

Rather more of the latter, actually, since it's the only place away from curious eyes where our hero will be able to metamorphose from weakling into the bane of all evildoers. And we *really* don't want to become, erm, bogged down with details of the room's function or plumbing.

There's not a lot about the toilet room and its contents, though there will be some tricky side effects:

TYPE

```
Room    toilet "Unisex toilet"
  with  description
            "A surprisingly CLEAN square room covered with glazed-ceramic
             tiles, featuring little more than a lavatory and a light switch.
             The only exit is south, through the door and into the cafe.",
        s_to toilet_door,
  has   ~light scored;

Appliance lavatory "lavatory" toilet
  with  name 'lavatory' 'wc' 'toilet' 'loo' 'bowl' 'can' 'john' 'bog',
        before [; Examine:
            if (coin in self) {
                move coin to parent(self);
                "The latest user CIVILLY flushed it after use, but failed to
                 pick up the VALUABLE coin that fell from his pants.";
            }
        ];
```

```
Object  coin "valuable coin" lavatory
  with  name 'valuable' 'coin' 'silver' 'quidbuck',
        description "It's a genuine SILVER QUIDBUCK.",
        before [; Drop:
            "Such a valuable coin? Har, har! This must be a demonstration of
             your ULTRA-FLIPPANT jesting!";
        ],
        after [; Take:
            "You crouch into the SLEEPING DRAGON position and deftly, with
             PARAMOUNT STEALTH, you pocket the lost coin.";
        ],
  has   scored;
```

We initially place the coin as a child of the lavatory (just so that we can easily make the `if (coin in self)` one-time test). Since the lavatory does not have the `transparent` attribute set, the coin will be invisible to players until they try to EXAMINE the lavatory, an action that will move the coin into the toilet room. Once taken, the coin will remain in the inventory until the player gives it to Benny, because we trap any `Drop` actions to help the player to Do the Right Thing.

The lavatory object includes a load of helpful synonyms in its `name` property, including our favourite word `'toilet'`. That won't be a problem: the other objects here which may have TOILET in their names – the key and the door – both use the `pname` property to turn their use of `'toilet'` into a lower-priority adjective.

See that here we have the only two `scored` attributes of the game. The player will be awarded one point for entering the toilet room, and another for finding and picking up the coin.

You might have noticed that we are forcefully clearing the `light` attribute, inherited from the `Room` class. This will be a windowless space and, to add a touch of realism, we'll make the room a dark one, which will enable us to tell you about Inform's default behaviour when there's no light to see by. However, let's define first the light switch mentioned in the room's description to aid players in their dressing duties.

TYPE

```
Appliance light_switch "light switch" toilet
  with  name 'light' 'switch',
        description
            "A notorious ACHIEVEMENT of technological SCIENCE, elegant yet
             EASY to use.",
        before [; Push:
            if (self has on) <<SwitchOff self>>;
            else             <<SwitchOn  self>>;
        ],
        after [;
            SwitchOn:
                give self light;
                "You turn on the light in the toilet.";
            SwitchOff:
                give self ~light;
                "You turn off the light in the toilet.";
        ],
  has   switchable ~on;
```

Please notice the appearance of new attributes `switchable` and `on`. `switchable` enables the object to be turned on and off, and is typical of lanterns, computers, television sets, radios, and so on. The library automatically extends the description of these objects by indicating if they are currently on or off:

```
> X LIGHT SWITCH
A notorious ACHIEVEMENT of technological SCIENCE, elegant yet EASY to use.
The light switch is currently switched on.
```

Two new actions are ready to use, `SwitchOn` and `SwitchOff`. Left to themselves, they toggle the object's state between ON and OFF and display a message like:

```
You switch the brass lantern on.
```

They also take care of checking if the player fumbled and tried to turn on (or off) an object which was already on (or off). How does the library know the state of the object? This is thanks to the `on` attribute, which is set or cleared automatically as needed. You can, of course, set or clear it manually like any other attribute, with the `give` statement:

```
give self on;

give self ~on;
```

and check if a `switchable` object is on or off with the test:

```
if (light_switch has on) ...

if (light_switch hasnt on) ...
```

A `switchable` object is OFF by default. However, you'll notice that the `has` line of the object definition includes `~on`:

```
  has   switchable ~on;
```

Surely that's saying "not-on"? Surely that's what would have happened anyway if the line hadn't mentioned the attribute at all?

```
  has   switchable;
```

Absolutely true. Adding that `~on` attribute has no effect whatsoever on the game – but nevertheless it's a good idea. It's an *aide-mémoire*, a way of reminding ourselves that we start with the attribute clear, and that at some point we'll be setting it for some purpose. Trust us: it's worthwhile taking tiny opportunities like this to help yourself.

Let's see how our light switch works. We trap the `SwitchOn` and `SwitchOff` actions in the `after` property (when the switching has successfully taken place) and use them to give `light` to the light switch.

Uh, wait. To the light switch? Why not to the toilet room? Well, there's a reason and we'll see it in a minute. For now, just remember that, in order for players to see their surroundings, you need only one object in a room with the `light` attribute set. It doesn't have to be the room itself (though this is usually convenient).

After setting the `light` attribute, we display a customised message, to avoid the default:

```
You switch the light switch on.
```

which, given the name of the object, doesn't read very elegantly. We foresee that players might try to PUSH SWITCH, so we trap this attempt in a `before` property and redirect it to `SwitchOn` and `SwitchOff` actions, checking first which one is needed by testing the `on` attribute. Finally, we have made the switch a member of the class `Appliance`, so that the player doesn't walk away with it.

> NOTE: remember what we said about class inheritance? No matter what you define in the class, the object's definition has priority. The class `Appliance` defines a response for the `Push` action, but we override it here with a new behaviour.

And there was light

So the player walks into the toilet and

```
Darkness
It is pitch dark, and you can't see a thing.
```

Oops! No toilet description, no mention of the light switch, nothing. It is reasonable to think that if we have opened the toilet door to access the toilet, some light coming from the café room will illuminate our surroundings – at least until the player decides to close the door. So perhaps it would be a good idea to append a little code to the door object to account for this. A couple of lines in the `after` property will suffice:

TYPE

```
after [ ks;
    Unlock:
        if (self has locked) return false;
        print "You unlock ", (the) self, " and open it.^";
        ks = keep_silent; keep_silent = true;
        <Open self>; keep_silent = ks;
        return true;
    Open:  give toilet light;
    Close: give toilet ~light;
],
```

And this is the reason why the light switch didn't set the `light` attribute of the toilet room, but did it to itself. We avoid running into trouble if we let the open/closed states of the door control the light of the room object, and the on/off states of the switch control the light of the switch. So it is one shiny light switch. Fortunately, players are never aware of this glowing artefact.

> NOTE: now, could they? Well, if players could TAKE the light switch (which we have forbidden) and then did INVENTORY, the trick would be given away, because all objects with the `light` attribute set are listed as `(providing light)`.

So the player walks into the toilet and

```
Unisex toilet
A surprisingly CLEAN square room covered with glazed-ceramic tiles, featuring
little more than a lavatory and a light switch. The only exit is south, through
the door and into the cafe.

[Your score has just gone up by one point.]
```

Better. Now, suppose the player closes the door.

```
>CLOSE DOOR
You close the door to the cafe.

It is now pitch dark in here!
```

The player might try then to LOOK:

```
>L

Darkness
It is pitch dark, and you can't see a thing.
```

Well, no problem. We have mentioned that there is a light switch. Surely the player will now try to:

```
>TURN ON LIGHT SWITCH
You can't see any such thing.
```

Oops! Things are getting nasty here in the dark. It's probably time to leave this place and try another approach:

```
>OPEN DOOR
You can't see any such thing.
```

And this illustrates one of the terrible things about darkness in a game. You can't see anything; you can do very little indeed. All objects except those in your inventory are out of scope, unreachable, as if non-existent. Worse, if you DROP one of the objects you are carrying, it will be swallowed by the dark, never to be found until there is light to see by.

The player, who is doubtless immersed in the fantasy of the game, will now be a little annoyed. "I am in a small bathroom and I can't even reach the door I have just closed?" The player's right, of course[1]. Darkened rooms are one cliché of traditional games. Usually you move in one direction while looking for treasure in some underground cave, and suddenly arrive at a pitch black place. It's good behaviour of the game to disallow exploration of unknown dark territory, and it's a convention to bar passage to players until they return with a light source.

---

1. We're alluding here to the Classical concept of *mimesis.* In an oft-quoted essay from 1996, Roger Giner-Sorolla wrote: "I see successful fiction as an imitation or 'mimesis' of reality, be it this world's or an alternate world's. Well-written fiction leads the reader to temporarily enter and believe in the reality of that world. A *crime against mimesis* is any aspect of an IF game that breaks the coherence of its fictional world as a representation of reality."

However, if the scenario of the game features, say, the player character's home, a little apartment with two rooms, and there's no light in the kitchen, we could expect the owner of the house to know how to move around a little, perhaps groping for the light switch or even going to the refrigerator in the dark.

We are in a similar situation. The inner logic of the game demands that blind players should be able to open the door and probably operate the light switch they've just encountered. We have been telling you that an object is in scope when it's in the same room as the player. Darkness changes that rule. All objects not directly carried by the player become out of scope.

One of the advantages of an advanced design system like Inform is the flexibility to change all default behaviours to suit your particular needs. Scope problems are no different. There is a set of routines and functions to tamper with what's in scope when. We'll see just a tiny example to fix our particular problem. In the section "`Entry point routines`" of our game – after the `Initialise` routine, for instance – include the following lines:

```
[ InScope person;
    if (person == player && location == thedark && real_location == toilet) {
        PlaceInScope(light_switch);
        PlaceInScope(toilet_door);
    }
    return false;
];
```

`InScope(`*`actor_obj_id`*`)` is an entry point routine that can tamper with the scope rules for the given *`actor_obj_id`* (either the player character or a NPC). We define it with one variable (which we name as we please; it's also a good idea to name variables in an intuitive way to remind us of what they represent), `person`, and then we make a complex test to see if the player is actually in the toilet and in the dark.

We have told you that the library variable `location` holds the current room that the player is in. However, when there is no light, the variable `location` gets assigned to the value of the special library object `thedark`. It doesn't matter if we have ten dark rooms in our game; `location` will be equal to `thedark` in all of them. There is yet another variable, called `real_location`, which holds the room the player is in *even when there is no light to see by.*

So the test:

```
if (person == player && location == thedark && real_location == toilet) ...
```

is stating: if the specified actor is the `player` character *and* he finds himself in the dark *and* he actually happens to be in the toilet...

Then we make a call to one of the library routines, `PlaceInScope(`*`obj_id`*`)`, which has a very descriptive name: it places in scope the given object. In our case, we want both the door and the light switch to be within reach of the player, hence both additional lines. Finally, we must `return false`, because we want the normal

scope rules for the defined actor – the player – to apply to the rest of the objects of the game (if we returned `true`, players would find that they are able to interact with very little indeed). Now we get a friendlier and more logical response:

```
Darkness
It is pitch dark, and you can't see a thing.

>TURN ON SWITCH
You turn on the light in the toilet.

Unisex toilet
A surprisingly CLEAN square room covered with glazed-ceramic tiles, featuring
little more than a lavatory and a light switch. The only exit is south, through
the door and into the cafe.
```

And the same would happen with the door. Notice how the room description gets displayed after we pass from dark to light; this is the normal library behaviour.

There is still one final problem which, admittedly, might originate from an improbable course of action; however, it could be a nuisance. Suppose that the player enters the toilet, locks the door – which is possible in the dark now that the door is in scope – and then drops the key. There's no way to exit the toilet – because the door is locked and the key has disappeared, engulfed by the darkness – unless the player thinks to turn on the light switch, thereby placing the key in scope once more.

Why don't we add a `PlaceInScope(toilet_key)` to the above routine? Well, for starters, the key can be moved around (as opposed to the door or the light switch, which are fixed items in the toilet room). Suppose the player opens the door of the toilet, but drops the key in the café, then enters the toilet and closes the door. The condition is met and the key is placed in scope, when it's in another room. Second, this is a simple game with just a few objects, so you can define a rule for each of them; but in any large game, you might like to be able to refer to objects in bunches, and make general rules that apply to all (or some) of them.

We need to add code to the `InScope` routine, telling the game to place in scope all objects that we drop in the dark, so that we might recover them (maybe going on all fours and groping a little, but it's a possible action). We don't want the player to have other objects in scope (like the coin, for instance), so it might be good to have a way of testing if the objects have been touched and carried by the player. The attribute `moved` is perfect for this. The library sets it for every object that the player has picked up at one time in the game; `scenery` and `static` objects, and those we have not yet seen don't have `moved`. Here is the reworked `InScope` routine. There are a couple of new concepts to look at:

TYPE

```
[ InScope person item;
    if (person == player && location == thedark && real_location == toilet) {
        PlaceInScope(light_switch);
        PlaceInScope(toilet_door);
    }
    if (person == player && location == thedark)
        objectloop (item in parent(player))
            if (item has moved) PlaceInScope(item);
    return false;
];
```

We have added one more local variable to the routine, `item` – again, this is a variable we have created and named on our own; it is not part of the library. We make now a new test: if the actor is the player and the location is any dark room, then perform a certain action. We don't need to specify the toilet, because we want this rule to apply to all dark rooms (well, the only dark room in the game *is* the toilet, but we are trying to provide a general rule).

```
objectloop (variable) statement;
```

is a loop statement, one of the four defined in Inform. A loop statement is a construct that allows you to run several times through a statement (or a statement block). `objectloop` performs the *`statement`* once for every object defined in the (*`variable`*). If we were to code:

```
objectloop (item) statement;
```

then the *`statement`* would be executed once for each object in the game. However, we want to perform the statement only for those objects whose parent object is the same as the player's parent object: that is, for objects in the same room as the player, so we instead code:

```
objectloop (item in parent(player)) statement;
```

What is the actual *`statement`* that we'll repeatedly execute?

```
    if (item has moved)
        PlaceInScope(item);
```

The test: `if (item has moved)` ensures that `PlaceInScope(item)` deals only with objects with the `moved` attribute set. So: if the player is in the dark, let's go through the objects which are in the same room, one at a time. For each of them, check if it's an item that the player has at some time carried, in which case, place it in scope. All dropped objects within the room were carried at one time, so we let players recollect them even if they can't see them.

As you see, darkness has its delicate side. If you plan to have dark rooms galore in your games, bear in mind that you are in for some elaborate code (unless you let the library carry on with default rules, in which case there won't be much for your players to do).

Amazing technicolour dreamcoats

This leaves us the clothing items themselves, which will require a few tailored actions. Let's see first the ordinary garments of John Covarth:

TYPE

```
Object  clothes "your clothes"
  with  name 'ordinary' 'street' 'clothes' 'clothing',
        description
            "Perfectly ORDINARY-LOOKING street clothes for a NOBODY like
             John Covarth.",
        before [;
            Disrobe,Change:
                switch (location) {
                    street:
                        if (player in booth)
                            "Lacking Superman's super-speed, you realise that
                             it would be awkward to change in plain view of
                             the passing pedestrians.";
                        else
                            "In the middle of the street? That would be a
                             PUBLIC SCANDAL, to say nothing of revealing your
                             secret identity.";
                    cafe:
                        "Benny allows no monkey business in his
                         establishment.";
                    toilet:
                        if (toilet_door has open)
                            "The door to the bar stands OPEN at tens of
                             curious eyes. You'd be forced to arrest yourself
                             for LEWD conduct.";
                        print "You quickly remove your street clothes and
                            bundle them up together into an INFRA MINUSCULE
                            pack ready for easy transportation. ";
                        if (toilet_door has locked) {
                            give clothes ~worn; give costume worn;
                            "Then you unfold your INVULNERABLE-COTTON costume
                             and turn into Captain FATE, defender of free
                             will, adversary of tyranny!";
                        }
                        else {
                            deadflag = 3;
                            "Just as you are slipping into Captain FATE's
                             costume, the door opens and a young woman
                             enters. She looks at you and starts screaming,
                             ~RAPIST! NAKED RAPIST IN THE TOILET!!!~^^
                             Everybody in the cafe quickly comes to the
                             rescue, only to find you ridiculously jumping on
                             one leg while trying to get dressed. Their
                             laughter brings a QUICK END to your
                             crime-fighting career!";
                        }
                    thedark:
                        "Last time you changed in the dark,
                         you wore the suit inside out!";
                }
```

```
                Wear:
                    if (self has worn)
                        "You are already dressed as John Covarth.";
                    "The town NEEDS the power of Captain FATE, not the anonymity
                     of John Covarth.";
            ],
      has   clothing proper pluralname;
```

See how the object deals only with `Disrobe`, `Change` and `Wear`. `Disrobe` and `Wear` are standard library actions already defined in Inform, but we'll have to make a new verb to allow for CHANGE CLOTHES. In this game, `Disrobe` and `Change` are considered synonymous for all purposes.

The goal of the game is for players to change their clothes, so we might expect them to try this almost anywhere. What we do with the `switch` statement is to offer a variety of responses according to the `location` variable. The street (in or out of the booth) and the café all display refusals of some kind, until the player character manages to enter the toilet, where we additionally require that he locks the door before taking off his clothes. If the door is closed but not locked, he is interrupted in his naked state by a nervous woman who starts shouting, and the game is lost (this is not as unfair as it seems, because the player may always revert to the previous state with UNDO). If the door is locked, he succeeds in his transformation (we take away the `worn` attribute from the `clothes` and give it to the `costume` instead). We add a special refusal to change in the dark, forcing players to turn on the light and then, we hope, to find the coin. The `Wear` action just checks if these clothes are already being worn, to offer two different rejection responses: the goal of the game is to change into the hero's suit, after which we'll prevent a change back into ordinary clothes. So now we are dealing with a Captain Fate in full costume:

TYPE

```
Object  costume "your costume"
  with  name 'captain' 'captain^s' 'fate' 'fate^s' 'costume' 'suit',
        description
            "STATE OF THE ART manufacture, from chemically reinforced 100%
             COTTON-lastic(tm).",
        before [;
            Wear:
                if (clothes has worn)
                    "First you'd have to take off your commonplace
                     unassuming John Covarth INCOGNITO street clothes.";
            Disrobe,Change:
                if (clothes has worn)
                    "But you're not yet wearing it!";
                else
                    "You need to wear your costume to FIGHT crime!";
            Drop:
                "Your UNIQUE Captain FATE multi-coloured costume? The most
                 coveted clothing ITEM in the whole city? Certainly NOT!";
        ],
  has   clothing proper;
```

Note that we intercept the action WEAR COSTUME and hint that players should try TAKE OFF CLOTHES instead. We don't let them take off the costume once

it's being worn, and we certainly don't let them misplace it anywhere, by refusing to accept a `Drop` action.

It's a wrap

Nearly there; just a few minor odds and ends to round things off.

Initialise routine

All the objects of our game are defined. Now we must add a couple of lines to the `Initialise` routine to make sure that the player does not start the game naked:

TYPE

```
[ Initialise;
    #Ifdef DEBUG; pname_verify(); #Endif;       ! suggested by pname.h
    location = street;
    move costume to player;
    move clothes to player; give clothes worn;
    lookmode = 2;
    "^^Impersonating mild mannered John Covarth, assistant help boy at an
     insignificant drugstore, you suddenly STOP when your acute hearing
     deciphers a stray radio call from the POLICE. There's some MADMAN
     attacking the population in Granary Park! You must change into your
     Captain FATE costume fast...!^^";
];
```

Remember that we included a disambiguation package, `pname.h`? There were some additional comments in the accompanying text file that should be taken in consideration:

> pname.h provides a pname_verify routine. When DEBUG is defined, you may call pname_verify() in your Initialise() routine to verify the pname properties in your objects.

The designer of the package has made a debugging tool (a routine) to check for errors when using his library, and he tells us how to use it. So we include the suggested lines into our `Initialise` routine:

```
#Ifdef DEBUG; pname_verify(); #Endif;
```

As the text explains, what this does is: first check whether the game is being compiled in Debug mode (games are compiled by default in Strict mode, which includes Debug); if this is the case, run the `pname_verify` routine, so that it tests all `pname` properties to see if they are written correctly.

Demise of our hero

We have made three possible endings:

1. The player attempts to change in the toilet with an unlocked door.
2. The player tries to attack Benny while wearing the costume.
3. The player manages to exit the café dressed as Captain Fate.

(1) and (2) lose the game, (3) wins it. The library defaults for these two states display, respectively,

```
*** You have died ***

*** You have won ***
```

These states correspond to the values of the `deadflag` variable: 1 for losing, 2 for winning. However, we have made up different messages, because our hero does not really die – ours suffers a FATE worse than death – and because we want to give him a more descriptive winning line. Therefore, we must define a `DeathMessage` routine as we did in “William Tell”, to write our customised messages and assign them to `deadflag` values greater than 2.

TYPE

```
[ DeathMessage;
    if (deadflag == 3) print "Your secret identity has been revealed";
    if (deadflag == 4) print "You have been SHAMEFULLY defeated";
    if (deadflag == 5) print "You fly away to SAVE the DAY";
];
```

Grammar

Finally, we need to extend the existing grammar, to allow for a couple of things. We have already seen that we need a verb CHANGE. We’ll make it really simple:

TYPE

```
[ ChangeSub;
    if (noun has pluralname) print "They're";
    else                     print "That's";
    " not something you must change to save the day.";
];

Verb 'change'
    * noun                    -> Change;
```

Just notice how the verb handler checks whether the noun given is plural or singular, to display a suitable pronoun.

A further detail: when players are in the café, they might ask Benny for the coffee (as we intend and heavily hint), for a sandwich or a pastry (both mentioned in the café description), for food or a snack (mentioned here and there, and we have provided for those); but what if they try a meat pie? Or scrambled eggs? There’s just so much decoration one can reasonably insert in a game, and loading the dictionary with Benny’s full menu would be overdoing it a bit.

One might reasonably imagine that the `default` line at the end of the `Give` action in the `orders` property handles every input not already specified:

```
orders [;
    Give: switch (noun) {
        toilet_key: code for the key...
        coffee:     code for the coffee...
        food:       code for the food...
        menu:       code for the menu...
        default:
            "~I don't think that's on the menu, sir.~";
    }
],
```

Not so. The library grammar that deals with ASK BENNY FOR... is this (specifically, the last line):

```
Verb 'ask'
    * creature 'about' topic    -> Ask
    * creature 'for' noun       -> AskFor
```

You'll see the `noun` token, which means that whatever the player asks him for must be a real game object, visible at that moment. Assuming that the player mentions such an object, the interpreter finds it in the dictionary and places its internal ID in the `noun` variable, where our `switch` statement can handle it. So, ASK BENNY FOR KEY assigns the `toilet_key` object to the `noun` variable, and Benny responds. ASK BENNY FOR CUSTOMERS also works; the `default` case picks that one up. But, ASK BENNY FOR SPAGHETTI BOLOGNESE won't work: we have no object for Spaghetti Bolognese (or any other delicacy from Benny's kitchen) – the words `'spaghetti'` and `'bolognese'` simply aren't in the dictionary. This is perhaps not a major deficiency in our game, but it takes very little to allow Benny to use his default line for *any* undefined input from the player. We need to extend the existing ASK grammar:

TYPE

```
Extend 'ask'
    * creature 'for' topic      -> AskFor;
```

This line will be added to the end of the existing grammar for Ask, so it doesn't override the conventional noun-matching line. `topic` is a token that roughly means "any input at all"; the value of `noun` isn't important, because it'll be handled by the `default` case. Now players may ask Benny for a tuna sandwich or a good time; they'll get: "I don't think that's on the menu, sir", which makes Benny a barman with attitude.

And that's it; on the slightly surreal note of ASK BENNY FOR A GOOD TIME we've taken "Captain Fate" as far as we intend to. The guide is nearly done. All that's left is to recap some of the more important issues, talk a little more about compilation and debugging, and send you off into the big wide world of IF authorship.

14 • Some last lousy points

inally our three example games are written; we've shown you as much of the Inform language as we've needed to, and made a lot of observations about how and why something should be done. Despite all that, there's much that we've left unsaid, or touched on only lightly. In this chapter we'll revisit key topics and review some of the more important omissions, to give you a better feel for what matters, and what can be left on the shelf.

We'll also talk, in "Reading other people's code" on page 159, about a few ways of doing things that we've chosen *not* to tell you about, but which you're quite likely to encounter if you look at Inform code written by other designers.

The tone here is perhaps a little dry, but trust us: in walking this dusty ground we touch on just about everything that is fundamental in your overall understanding of Inform. And as always, the *Inform Designer's Manual* provides rounder and more comprehensive coverage.

Expressions

In this guide we've use the placeholder *`expression`* a few times; here's roughly what we mean.

- An *`expression`* is a single *`value`*, or several *`values`* combined using *`operators`* and sometimes parentheses `(...)`.
- Possible *`values`* include:
  - a literal number (-32768 to 32767)
  - something that's represented as a number (a character `'a'`, a dictionary word `'aardvark'`, a string `"aardvark's adventure"` or an action `##Look`)
  - the internal identifier of a constant, an object, a class or a routine
  - (only in a run-time statement, not in a compile-time directive) the contents of a variable, or the return value from a routine.
- Possible *`operators`* include:
  - an arithmetic operator: `+ - * / % ++ --`
  - a bitwise logical operator: `& | ~`
  - a numeric comparison operator: `== ~= > < >= <=`
  - an object conditional operator: `ofclass in notin provides has hasnt`
  - a boolean combinational operator: `&& || ~~`

Internal IDs

Many of the items which you define in your source file – objects, variables, routines, etc. – need to be given a name so that other items can refer to them. We call this name an item's internal identifier (because it's used only within the source file and isn't visible to the player), and we use the placeholders *obj_id*, *var_id*, *routine_id*, etc. to represent where it's used. An internal ID

- can be up to thirty-two characters long
- must start with a letter or underscore, and then continue with letters `A-Z`, underscore _ and digits `0-9` (where upper-case and lower-case letters are treated as indistinguishable)
- should generally be unique across all files: your source file, the standard library files, and any library contributions which you've used (except that a routine's local variables are not visible outside that routine).

Statements

A **statement** is an instruction intended for the interpreter, telling it what to do at run-time. It *must* be given in lower-case, and always ends with a semicolon.

Some statements, like `if`, control one or more other statements. We use the placeholder *statement_block* to represent either a single *statement*, or any number of *statements* enclosed in braces:

```
statement;

{ statement; statement; ... statement; }
```

Statements that we've met

Our games have used these statements, about half of the Inform possibilities:

```
give obj_id attribute;
give obj_id attribute  attribute  ... attribute;

if (expression) statement_block
if (expression) statement_block else statement_block

move obj_id to parent_obj_id;

new_line;

objectloop (var_id) statement_block

print value;
print value, value, ... value;

print_ret value;
print_ret value, value, ... value;

remove obj_id;

return false;
return true;
```

```
style underline; print...; style roman;

switch (expression) {
    value: statement; statement; ... statement;
    value: statement; statement; ... statement;
    ...
    default: statement; statement; ... statement;
}

"string";
"string", value, ... value;

<action>;
<action noun>;
<action noun second>;

<<action>>;
<<action noun>>;
<<action noun second>>;
```

Statements that we've not met

Although our example games haven't needed to use them, these looping statements are sometimes useful:

```
break;

continue;

do statement_block until (expression)

for (set_var : loop_while_expression : update_var) statement_block

while (expression) statement_block
```

On the other hand, we suggest that you forget about these statements for now:

```
box
font
jump
spaces
string
```

Print rules

In `print` and `print_ret` statements, each *`value`* can be:

- a numeric *`expression`*, displayed as a signed decimal number,
- a `"`*`string`*`"`, displayed literally, or
- a print rule. You can create your own, or use a standard one, including:

`(a)` *`obj_id`*	– the object's name, preceded by "a", "an" or "some"
`(the)` *`obj_id`*	– the object's name, preceded by "the"
`(The)` *`obj_id`*	– the object's name, preceded by "The"
`(number)` *`expression`*	– the numeric expression's value in words

Directives

A **directive** is an instruction intended for the compiler, telling it what to do at compile-time, while the source file is being translated into Z-code. By convention it's given an initial capital letter (though the compiler doesn't enforce this) and always ends with a semicolon.

Directives that we've met

We've used all of these directives; note that for `Class`, `Extend`, `Object` and `Verb` the full supported syntax is more sophisticated than the basic form presented here:

```
Class   class_id
  with  property  value,
        property  value,
        ...
        property  value,
  has   attribute  attribute  ... attribute;

Constant const_id;
Constant const_id = expression;
Constant const_id expression;

Extend 'verb'
    * token  token  ... token -> action
    * token  token  ... token -> action
    ...
    * token  token  ... token -> action;

Include "filename";

Object  obj_id  "external_name"  parent_obj_id
  with  property  value,
        property  value,
        ...
        property  value,
  has   attribute  attribute  ... attribute;

class_id obj_id  "external_name"  parent_obj_id
  with  property  value,
        property  value,
        ...
        property  value,
  has   attribute  attribute  ... attribute;

Release expression;

Replace routine_id;

Serial "yymmdd";

Verb 'verb'
    * token  token  ... token -> action
    * token  token  ... token -> action
    ...
    * token  token  ... token -> action;

! comment text which the compiler ignores

[ routine_id; statement; statement; ... statement; ];

#Ifdef any_id;  ... #Endif;
```

Directives that we've not met

There's only a handful of useful directives which we haven't needed to use:

```
Attribute attribute;

Global var_id;
Global var_id = expression;

Property property;

Statusline score;
Statusline time;
```

but there's a whole load which are of fairly low importance for now:

```
Abbreviate
Array
Default
End
Ifndef
Ifnot
Iftrue
Iffalse
Import
Link
Lowstring
Message
Switches
System_file
Zcharacter
```

Objects

An object is really just a collection of variables which together represent the capabilities and current status of some specific component of the model world. Full variables are called properties; simpler two-state variables are attributes.

Properties

The library defines around forty-eight standard property variables (such as `before` or `name`), but you can readily create further ones just by using them within an object definition.

You can create and initialise a property in an object's `with` segment:

```
property,                              ! set to zero/false

property value,                        ! set to a single value

property value value ... value,        ! set to a list of values
```

In each case, the *value* is either a compile-time *expression*, or an embedded routine:

```
property expression,

property [; statement; statement; ... statement; ],
```

You can refer to the value of a property:

```
self.property                             ! only within that same object

obj_id.property                           ! everywhere
```

and you can test whether an object definition includes a given property:

```
(obj_id provides property)                ! is true or false
```

Attributes

The library defines around thirty standard property attributes (such as `open` or `worn`); creating further ones is done relatively rarely.

You can initialise attributes in an object's `has` segment:

```
attribute attribute ...                   ! initially set

~attribute ~attribute ...                 ! initially unset (the default)
```

You can set and clear attributes:

```
give obj_id attribute attribute ... attribute;

give obj_id ~attribute ~attribute ... ~attribute;
```

and you can test the current setting of an attribute:

```
(obj_id has attribute)                    ! is true or false

(obj_id hasnt attribute)                  ! is false or true
```

Classes

You can test whether an object is a member of a given class:

```
(obj_id ofclass class_id)                 ! is true or false
```

The object tree

You can specify an object's parent (its location at the start of the game) as part of the object definition:

```
Object  obj_id  "external_name"  parent_obj_id
  with  ...
```

There's another syntax, involving arrows `->  ->` like this, which can also be used to establish an object's initial parent. We touch on it in "Reading other people's code" on page 159.

You can relocate an object within the tree:

```
move obj_id to parent_obj_id;
```

and you can move an object out of the tree so that it has no parent:

```
remove obj_id;
```

Given an object's `obj_id`, you can determine that object's current parent, its eldest child, and the next youngest child – the adjacent object – having the same parent. You can also count how many immediate children it has:

```
parent(obj_id)

child(obj_id)

sibling(obj_id)

children(obj_id)
```

You can test whether an `obj_id` is an immediate child of another object:

```
(obj_id in parent_obj_id)                ! is true or false

(obj_id notin parent_obj_id)             ! is false or true
```

If you need to know whether an object is a child, or grandchild, or great-grandchild, etc. of another object, use:

```
IndirectlyContains(parent_obj_id, obj_id)   ! is true or false
```

Finally, you can determine the object (if any) of which two specified objects are both children, or grandchildren, or great-grandchildren, etc. using:

```
CommonAncestor(obj_id1, obj_id2)
```

Routines

Inform provides standalone routines and embedded routines.

Standalone routines

Standalone routines are defined like this:

```
[ routine_id; statement; statement; ... statement; ];
```

and called like this:

```
routine_id()
```

Embedded routines

These are embedded as the value of an object's property:

```
property [; statement; statement; ... statement; ],
```

and are usually called automatically by the library, or manually by:

```
self.property()                          ! only within that same object

obj_id.property()                        ! everywhere
```

Arguments and local variables

Both types of routine support up to fifteen local variables – variables which can be used only by the statements within the routine, and which are automatically initialised to zero every time that the routine is called:

```
[ routine_id var_id var_id ... var_id; statement; statement; ... statement; ];

property [ var_id var_id ... var_id; statement; statement; ... statement; ],
```

You can pass up to seven arguments to a routine, by listing those arguments within the parentheses when you call the routine. The effect is simply to initialise the matching local variables to the argument values rather than to zero:

```
routine_id(expression, expression, ... expression)
```

Although it works, this technique is rarely used with embedded routines, because there is no mechanism for the library to supply argument values when calling the routine.

Return values

Every routine returns a single value, which is supplied either explicitly by some form of `return` statement:

```
[ routine_id; statement; statement; ... return expr; ];     ! returns expr

property [; statement; statement; ... return expr; ],       ! returns expr
```

or implicitly when the routine runs out of statements. If none of these *statements* is one – `return`, `print_ret`, `"..."` or `<<...>>` – that causes an explicit return, then:

```
[ routine_id; statement; statement; ... statement; ];
```

returns `true` and

```
property [; statement; statement; ... statement; ]
```

returns `false`.

This difference is *important.* Remember it by the letter pairs STEF: left to themselves, Standalone routines return True, Embedded routines return False.

Here's an example standalone routine which returns the larger of its two argument values:

```
[ Max a b; if (a > b) return a; else return b; ];
```

and here are some examples of its use (note that the first example, though legal, does nothing useful whatsoever):

```
Max(x,y);

x = Max(2,3);

if (Max(x,7) == 7) ...

switch (Max(3,y)) { ...
```

Library routines versus entry points

A library routine is a standard routine, included within the library files, which you can optionally call from your source file if you require the functionality which the routine provides. We've mentioned these library routines:

```
IndirectlyContains(parent_obj_id, obj_id)

PlaceInScope(obj_id)

PlayerTo(obj_id, flag)

StartDaemon(obj_id)

StopDaemon(obj_id)
```

By contrast, an entry point routine is a routine which you can provide in your source file, in which case the library calls it at an appropriate time. We've mentioned these optional entry point routines:

```
DeathMessage()

InScope(actor_obj_id)
```

And this, the only mandatory one:

```
Initialise()
```

There are full lists in "Library routines" on page 236 and "Optional entry points" on page 242.

Reading other people's code

Right at the start of this guide, we warned you that we weren't setting out to be comprehensive; we've concentrated on presenting the most important aspects of Inform, as clearly as we can. However, when you read the *Inform Designer's Manual*, and more especially when you look at complete games or library extensions which other designers have produced, you'll come across other ways of doing things – and it might be that you, like other authors, prefer them over our methods. Just try to find a style that suits you and, this is the important bit, be *consistent* about its use. In this section, we highlight some of the more obvious differences which you may encounter.

Code layout

Every designer has his or her own style for laying out their source code, and they're all worse than the one you adopt. Inform's flexibility makes it easy for designers to choose a style that suits them; unfortunately, for some designers this choice seems influenced by the Jackson Pollock school of art. We've advised you to be consistent, to use plenty of white space and indentation, to choose sensible names, to add comments at difficult sections, to actively *think*, as you write your code, about making it as readable as you can.

This is doubly true if you ever contemplate sharing a library extension with the rest of the community. This example, with the name changed, is from a file in the Archive:

```
[xxxx i j;
if (j==0) rtrue;
if (i in player) rtrue;
if (i has static || (i has scenery)) rtrue;
action=##linktake;
if (runroutines(j,before) ~= 0 || (j has static || (j has scenery))) {
print "You'll have to disconnect ",(the) i," from ",(the) j," first.^";
rtrue;
}
else {
if (runroutines(i,before)~=0 || (i has static || (i has scenery))) {
print "You'll have to disconnect ",(the) i," from ",(the) j," first.^";
rtrue;
}
else
if (j hasnt concealed && j hasnt static) move j to player;
if (i hasnt static && i hasnt concealed) move i to player;
action=##linktake;
if (runroutines(j,after) ~= 0) rtrue;
print "You take ",(the) i," and ",(the) j," connected to it.^";
rtrue;
}
];
```

Here's the same routine after a few minutes spent purely on making it more comprehensible; we haven't actually tested that it (still) works, though that second `else` looks suspicious:

```
[ xxxx i j;
    if (i in player || i has static or scenery || j == nothing) return true;
    action = ##LinkTake;
    if (RunRoutines(j,before) || j has static or scenery)
        "You'll have to disconnect ", (the) i, " from ", (the) j, " first.";
    else {
        if (RunRoutines(i,before) || i has static or scenery)
            "You'll have to disconnect ", (the) i, " from ", (the) j, " first.";
        else
            if (j hasnt static or concealed) move j to player;
        if (i hasnt static or concealed) move i to player;
        if (RunRoutines(j,after)) return true;
        "You take ", (the) i, " and ", (the) j, " connected to it.";
    }
];
```

We hope you'll agree that the result was worth the tiny extra effort. Code gets written once; it gets read dozens and dozens of times.

Shortcuts

There are a few statement shortcuts, some more useful than others, which you'll come across.

- These five lines all do the same thing:

```
return true;
return 1;
return;
rtrue;
];          ! at the end of a standalone routine
```

- These four lines all do the same thing:

```
return false;
return 0;
rfalse;
];          ! at the end of an embedded routine
```

- These four lines all do the same thing:

```
print "string"; new_line; return true;
print "string^"; return true;
print_ret "string";
"string";
```

- These lines are the same:

```
print value1; print value2; print value3;
print value1, value2, value3;
```

- These lines are the same:

```
<action noun second>; return true;
<<action noun second>>;
```

- These lines are also the same:

```
print "^";
new_line;
```

- These `if` statements are equivalent:

```
if (MyVar == 1 || MyVar == 3 || MyVar == 7) ...
if (MyVar == 1 or 3 or 7) ...
```

- These `if` statements are equivalent as well:

```
if (MyVar ~= 1 && MyVar ~= 3 && MyVar ~= 7) ...
if (MyVar ~= 1 or 3 or 7) ...
```

- In an `if` statement, the thing in parentheses can be *any* expression; all that matters is its value: zero (false) or anything else (true). For example, these statements are equivalent:

```
if (MyVar ~= false) ...
if (~~(MyVar == false)) ...
if (MyVar ~= 0) ...
if (~~(MyVar == 0)) ...
if (MyVar) ...
```

Note that the following statement specifically tests whether `MyVar` contains `true` (1), *not* whether its value is anything other than zero.

```
if (MyVar == true) ...
```

- If `MyVar` is a variable, the statements `MyVar++;` and `++MyVar;` work the same as `MyVar = MyVar + 1;` For example, these lines are equivalent:

```
MyVar = MyVar + 1; if (MyVar == 3) ...
if (++MyVar == 3) ...
if (MyVar++ == 2) ...
```

What's the same about `MyVar++` and `++MyVar` is that they both add one to `MyVar`. What's different about them is the value to which the construct itself evaluates: `MyVar++` returns the current value of `MyVar` and then performs the increment, whereas `++MyVar` does the "+1" first and then returns the incremented value. In the example, if `MyVar` currently contains 2 then `++MyVar` returns 3 and `MyVar++` returns 2, even though in both cases the value of `MyVar` afterwards is 3. As another example, this code (from Helga in "William Tell"):

```
Talk: self.times_spoken_to = self.times_spoken_to + 1;
    switch (self.times_spoken_to) {
        1:  score = score + 1;
            print_ret "You warmly thank Helga for the apple.";
        2:  score = score + 1;
            print_ret "~See you again soon.~";
        default: return false;
    }
],
```

could have been written more succinctly like this:

```
Talk: switch (++self.times_spoken_to) {
    1:  score++;
        print_ret "You warmly thank Helga for the apple.";
    2:  score++;
        print_ret "~See you again soon.~";
    default: return false;
    }
],
```

- Similarly, the statements `MyVar--;` and `--MyVar;` work the same as `MyVar = MyVar - 1;` Again, these lines are equivalent:

```
MyVar = MyVar - 1; if (MyVar == 7) ...
if (--MyVar == 7) ...
if (MyVar-- == 8) ...
```

"number" property and "general" attribute

The library defines a standard `number` property and a standard `general` attribute, whose roles are undefined: they are general-purpose variables available within every object to designers as and when they desire.

We recommend that you avoid using these two variables, primarily because their names are, by their very nature, so bland as to be largely meaningless. Your game will be clearer and easier to debug if you instead create new property variables – with appropriate names – as part of your `Object` and `Class` definitions.

Common properties and attributes

As an alternative to creating new individual properties which apply only to a single object (or class of objects), it's possible to devise properties and new attributes which, like those defined by the library, are available on *all* objects. The need to do this is actually quite rare, and is mostly confined to library extensions (for example, the `pname.h` extension which we encountered in "Captain Fate: take 3" on page 127 gives every object a `pname` property and a `phrase_matched` attribute). To create them, you would use these directives near the start of your source file:

```
Attribute attribute;

Property property;
```

We recommend that you avoid using these two directives unless you really do need to affect every object in your game. There is a limit of forty-eight attributes (of which the library currently defines around thirty) and sixty-two of these common properties (of which the library currently defines around forty-eight). On the other hand, the number of individual properties which you can add is virtually unlimited.

Setting up the object tree

Throughout this guide, we've defined the initial position of each object within the overall object tree either by explicitly mentioning its parent's *`obj_id`* (if any) in the first line of the object definition – what we've been calling the header information – or, for a few objects which crop up in more than one place, by using their `found_in` properties. For example, in "William Tell" we defined twenty-seven objects; omitting those which used `found_in` to define their placement at the start of the game, we're left with object definitions starting like this:

```
Room    street "A street in Altdorf"

Room    below_square "Further along the street"
Furniture   stall "fruit and vegetable stall" below_square
Prop    "potatoes" below_square
Prop    "fruit and vegetables" below_square
NPC     stallholder "Helga" below_square

Room    south_square "South side of the square"

Room    mid_square "Middle of the square"
Furniture   pole "wooden pole" mid_square

Room    north_square "North side of the square"
```

```
Room    marketplace "Marketplace near the square"
Object  tree "lime tree" marketplace
NPC     governor "governor" marketplace

Object  bow "bow"

Object  quiver "quiver"
Arrow   "arrow" quiver
Arrow   "arrow" quiver
Arrow   "arrow" quiver

Object  apple "apple"
```

You'll see that several of the objects begin the game as parents: `below_square`, `mid_square`, `marketplace` and `quiver` all have child objects beneath them; those children mention their parent as the last item of header information.

There's an alternative object syntax which is available to achieve the same object tree, using "arrows". That is, we could have defined those parent-and-child objects as:

```
Room    below_square "Further along the street"
Furniture ->  stall "fruit and vegetable stall"
Prop    -> "potatoes"
Prop    -> "fruit and vegetables"
NPC     -> stallholder "Helga"

Room    mid_square "Middle of the square"
Furniture ->  pole "wooden pole"

Room    marketplace "Marketplace near the square"
Object  -> tree "lime tree"
NPC     -> governor "governor"

Object  quiver "quiver" -
Arrow   -> "arrow"
Arrow   -> "arrow"
Arrow   -> "arrow"
```

The idea is that an object's header information *either* starts with an arrow, *or* ends with an *`obj_id`*, *or* has neither (having both isn't permitted). An object with neither has no parent: in this example, that's all the `Rooms`, and also the `bow` and the `quiver` (which are moved to the `player` object in the `Initialise` routine) and the `apple` (which remains without a parent until Helga gives it to William).

An object which starts with a single arrow `->` is defined to be a child of the nearest previous object without a parent. Thus, for example, the `tree` and `governor` objects are both children of the `marketplace`. To define a child of a child, you'd use two arrows `-> ->`, and so on. In "William Tell", that situation doesn't occur; to illustrate how it works, imagine that at the start of the game the potatoes and the other fruit and vegetables where actually *on* the stall. Then we might have used:

```
Room    below_square "Further along the street"
Furniture ->  stall "fruit and vegetable stall"
Prop    ->  -> "potatoes"
Prop    ->  -> "fruit and vegetables"
NPC     -> stallholder "Helga"
...
```

That is, the objects with one arrow (the `stall` and `stallholder`) are children of the nearest object without a parent (the `Room`), and the objects with two arrows (the produce) are children of the nearest object defined with a single arrow (the `stall`).

The advantages of using arrows include:

- You're forced to define your objects in a "sensible" order.
- Fewer *obj_ids* may need to be used (though in this game it would make no difference).

The disadvantages include:

- The fact that objects are related by the physical juxtaposition of their definitions is not necessarily intuitive to all designers.
- Especially in a crowded room, it's harder to be certain exactly how the various parent–child relationships are initialised, other than by carefully counting lots of arrows.
- If you relocate the parent within the initial object hierarchy to a higher or lower level, you'll need also to change its children by adding or removing arrows; this isn't necessary when the parent is named in the child headers.

We prefer to explicitly name the parent, but you'll encounter both forms very regularly.

Quotes in "name" properties

We went to some lengths, way back in "Things in quotes" on page 47, to explain the difference between double quotes `"..."` (strings to be output) and single quotes `'...'` (input tokens – dictionary words). Perhaps somewhat unfortunately, Inform allows you to blur this clean distinction: you *can* use double quotes in `name` properties and `Verb` directives:

```
NPC     stallholder "Helga" below_square
  with  name "stallholder" "greengrocer" "monger" "shopkeeper" "merchant"
            "owner" "Helga" "dress" "scarf" "headscarf",
...

Verb "talk" "t//" "converse" "chat" "gossip"
    * "to"/"with" creature          -> Talk
    * creature                      -> Talk;
```

Please don't do this. You'll just confuse yourself: those are dictionary words, not strings; it's just as easy – and far clearer – to stick rigidly to the preferred punctuation.

Obsolete usages

Finally, remember that Inform has been evolving since 1993. Over that time, Graham has taken considerable care to maintain as much compatibility as possible, so that games written years ago, for earlier versions of the compiler and the library, will still compile today. While generally a good thing, this brings the disadvantage that a certain amount of obsolete baggage is still lying around. You may, for example, see games using `Nearby` directives (denotes parentage, roughly the same as `->`) and `near` conditions (roughly, having the same parent), or with "`\`" controlling line breaks in long `print` statements. Try to understand them; try *not* to use them.

15 • Compiling your game

Almost as rarely as an alchemist producing gold from base metal, the compilation process turns your source file into a story file (though the more usual outcome is a reproachful explanation of why – *again* – that hasn't happened). The magic is performed by the compiler program, which takes your more or less comprehensible code and translates it into a binary file: a collection of numbers following a specific format understood only by Z-code interpreters.

On the surface, compilation is a very simple trick. You just run the compiler program, indicating which is the source file from which you wish to generate a game and presto! The magic is done.

However, the ingredients for the spell must be carefully prepared. The compiler "reads" your source code, but not as flexibly as a human would. It needs the syntax to follow some very precise rules, or it will complain that it cannot do its job under these conditions. The compiler cares little for meaning, and a lot about orthography, like a most inflexible teacher; no moist Bambi eyes are going to save you here.

Although the spell made by the compiler is always the same one, you can indicate up to a point how you want the magic to happen. There are a few options to affect the process of compilation; some you define in the source code, some with *switches* and certain commands when you run the program. The compiler will work with some default options if you don't define any, but you may change these if you need to. Many of these options are provided "just in case" special conditions apply; others are meant for use of experienced designers with advanced and complex requirements, and are best left (for now) to those proficient in the lore.

Ingredients

If the source file is not written correctly the compiler will protest, issuing either a *warning* message or an *error* message. Warnings are there to tell you that there may be a mistake that could affect the behaviour of the game at run-time; that won't stop the compiler from finishing the process and producing a story file. Errors, on the other hand, reflect mistakes that make it impossible for the compiler to output such a file. Of these, *fatal* errors stop compilation immediately, while *non-fatal* errors allow the compiler to continue reading the source file. (As you'll see in a minute, this is perhaps a mixed blessing: while it can be useful to have the compiler tell you about as many non-fatal errors as it can, you'll often find that many of them are caused by the one simple slip-up.)

Fatal errors

It's difficult – but not impossible – to cause a fatal error. If you indicate the wrong file name as source file, the compiler won't even be able to start, telling you:

```
Couldn't open source file filename
```

If the compiler detects a large number of non-fatal errors, it may abandon the whole process with:

```
Too many errors: giving up
```

Otherwise, fatal errors most commonly occur when the compiler runs out of memory or disk space; with today's computers, that's pretty unusual. However, you may hit problems if the story file, which must fit within the fairly limited resources specified by the Z-Machine, becomes too large. Normally, Inform compiles your source code into a Version 5 file (that's what the `.z5` extension you see in the output file indicates), with a maximum size of 256 Kbytes. If your game is larger than this, you'll have to compile into Version 8 file (`.z8`), which can grow up to 512 Kbytes (and you do this very simply by setting the `-v8` switch; more on that in a minute). It takes a surprising amount of code to exceed these limits; you won't have to worry about game size for the next few months, if ever.

Non-fatal errors

Non-fatal errors are much more common. You'll learn to be friends with:

```
Expected something but found something else
```

This is the standard way of reporting a punctuation or syntax mistake. If you type a comma instead of a semicolon, Inform will be looking for something in vain. The good news is that you are pointed to the offending line of code:

```
Tell.inf(76): Error:  Expected directive, '[' or class name but found found_in
>       found_in
Compiled with 1 error (no output)
```

You see the line number (76) and what was found there, so you run to the source file and take a look; any decent editor will display numbers alongside your lines if you wish, and will usually let you jump to a given line number. In this case, the error was caused by a semicolon after the description string, instead of a comma:

```
Prop    "assorted stalls"
  with  name 'assorted' 'stalls',
        description "Food, clothing, mountain gear; the usual stuff.";
        found_in street below_square,
  has   pluralname;
```

Here's a rather misleading message which maybe suggests that things in our source file are in the wrong order, or that some expected punctuation is missing:

```
Fate.inf(459): Error:  Expected name for new object or its textual short name
but found door
> Object door
Compiled with 1 error (no output)
```

In fact, there's nothing wrong with the ordering or punctuation. The problem is actually that we've tried to define a new object with an internal ID of `door` – reasonably enough, you might think, since the object *is* a door – but Inform already knows the word (it's the name of a library attribute). Unfortunately, the error message provides only the vaguest hint that you just need to choose another name: we used `toilet_door` instead.

Once the compiler is off track and can't find what was expected, it's common for the following lines to be misinterpreted, even if there's nothing wrong with them. Imagine a metronome ticking away in time with a playing record. If the record has a scratch and the stylus jumps, it may seem that the rest of the song is out of sync, when it's merely a bit "displaced" because of that single incident. This also happens with Inform, which at times will give you an enormous list of things Expected but not Found. The rule here is: correct the first mistake on the list and recompile. It may be that the rest of the song was perfect.

It would be pointless for us to provide a comprehensive list of errors, because mistakes are numerous and, anyhow, the explanatory text usually indicates what was amiss. You'll get errors if you forget a comma or a semicolon. You'll get errors if your quotes or brackets don't pair up properly. You'll get errors if you use the same name for two things. You'll get errors – for many reasons. Just read the message, go to the line it mentions (and maybe check those just before and after it as well), and make whatever seems a sensible correction.

Warnings

Warnings are not immediately catastrophic, but you should get rid of them to ensure a good start at finding run-time mistakes (see "Debugging your game" on page 173). You may declare a variable and then not use it; you may mistake assignment and arithmetic operators (= instead of ==); you may forget the comma that separates properties, etc. For all these and many other warnings, Inform has found something which is legal but doubtful.

One common incident is to return in the middle of a statement block, before the rest of statements can be reached. This is not always as evident as it looks, for instance in a case like this:

```
if (steel_door has open) {
    print_ret "The breeze blows out your lit match.";
    give match ~light;
}
```

In the above example, the `print_ret` statement returns true after the string has been printed, and the `give match ~light` line will never happen. Inform detects the fault and warns you. Probably the designer's intention was:

```
if (steel_door has open) {
    give match ~light;
    print_ret "The breeze blows out your lit match.";
}
```

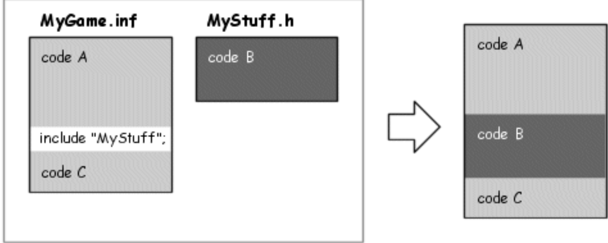
Compiling *à la carte*

One of the advantages of Inform is its portability between different systems and machines. Specific usage of the compiler varies accordingly, but some features should be in all environments. To obtain precise information about any particular version, run the compiler with the `-h1` switch – see “Switches” on page 171.

Often the compiler is run with the name of your source file as its only parameter. This tells the compiler to “read this file and from it generate a Version 5 story file of the same name”. The source file is mostly full of statements which define how the game is to behave at run-time, but will also include compile-time instructions directed at the compiler itself (although such an instruction looks a lot like a **statement**, it’s actually quite different in what it does, and is known as a **directive**). We have already seen the `Include` directive:

```
Include "filename";
```

When the compiler reaches a line like this, it looks for *`filename`* – another file also containing Inform code – and processes it as if the statements and directives included in *`filename`* were in that precise spot where the `Include` directive is.



In every Inform game we Include the library files `Parser`, `VerbLib` and `Grammar`, but we may Include other files. For example, this is the way to incorporate library extensions contributed by other people, as you saw when we incorporated `pname.h` into our “Captain Fate” game.

> NOTE: on some machines, a library file is actually called – for example – `Parser.h`, on others just `Parser`. The compiler automatically deals with such differences; you can *always* type simply `Include "Parser";` in your source file.

As you grow experienced in Inform, and your games become more complex, you may find that the source file becomes unmanageably large. One useful technique is then to divide it into a number of sections, each stored in a separate file, which you Include into a short master game file. For example:

```
!============================================================================
Constant Story "War and Peace";
Constant Headline
            "^An extended Inform example
             ^by me and Leo Tolstoy.^";

Include "Parser";
Include "VerbLib";

Include "1805";
Include "1806-11";
Include "1812A";
Include "1812B";
Include "1813-20";

Include "Grammar";

Include "Verbski";

!============================================================================
```

Switches

When you run the compiler you can set some optional controls; these are called *switches* because most of them are either on or off (although a few accept a numeric value 0–9). Switches affect compilation in a variety of ways, often just by changing the information displayed by the compiler when it's running. A typical command line (although this may vary between machines) would be:

```
inform source_file story_file switches
```

where "`inform`" is the name of the compiler, the *`story_file`* is optional (so that you can specify a different name from the *`source_file`*) and the *`switches`* are also optional. Note that switches must be preceded by a hyphen `-`; if you want to set, for instance, Strict mode, you'd write `-S`, while if you want to deactivate it, you'd write `-~S`. The tilde sign can, as elsewhere, be understood as "not". If you wish to set many switches, just write them one after another separated by spaces and each with its own hyphen, or merge them with one hyphen and no spaces:

```
inform MyGame.inf -S -s -X

inform MyGame.inf -SsX
```

Normally, all switches are off by default, except Strict mode (`-S`), which is on and checks the code for additional mistakes, also making the debugging verbs available at run time. This is the ideal setting while coding, but you should turn strict mode off (`-~S`) when you release your game to the public. This is fortunately very easy to check, since the game banner ends with the letters "SD" if the game was compiled in Strict mode:

```
Captain Fate
A simple Inform example
by Roger Firth and Sonja Kesserich.
Release 2 / Serial number 020827 / Inform v6.21 Library 6/10 SD
```

Switches are case sensitive, so you get different effects from `-x` and `-X`. Some of the more useful switches are:

`-~S`

Set compiler Strict mode off. This deactivates some additional error checking features when it reads your source file and also omits the debugging verbs from the story file (unless you then specify `-D`). Strict mode is on by default.

`-v5 -v8`

Compile to this version of story file. Versions 5 (on by default) and 8 are the only ones you should ever care about; they produce, respectively, story files with the extensions `.z5` and `.z8`. Version 5 was the Advanced Infocom design, and is the default produced by Inform. This is the version you'll normally be using, which allows file sizes up to 256 Kbytes. If your game grows beyond that size, you'll need to compile to the Version 8 story file, which is very similar to Version 5 but allows a 512 Kbytes file size.

`-D -X`

Include respectively the debugging verbs and the Infix debugger in the story file (see "Debugging your game" on page 173).

`-h1 -h2`

Display help information about the compiler. `-h1` produces information about file naming, and `-h2` about the available switches.

`-n -j`

`-n` displays the number of declared attributes, properties and actions. `-j` lists objects as they are being read and constructed in the story file.

`-s`

Offer game statistics. This provides a lot of information about your game, including the number of objects, verbs, dictionary entries, memory usage, etc., while at the same time indicating the maximum allowed for each entry. This can be useful to check whether you are nearing the limits of Inform.

`-r`

Record all the text of the game into a temporary file, useful to check all your descriptions and messages by running them through a spelling checker.

If you run the compiler with the `-h2` switch, you'll find that there are many more switches than these, offering mostly advanced or obscure features which we consider to be of little interest to beginners. However, feel free to try whatever switches catch your eye; nothing you try here will affect your source file, which is strictly read-only as far as the compiler is concerned.

16 • Debugging your game

Nobody understands the phrase *errare humanum est* quite in the same way as a programmer does. Computers are highly efficient machines capable of wondrous calculations, but they lack imagination and insist that every single item thrown at them must be presented according to certain rules previously defined. You can't negotiate with a computer; you either bow in submission or bite the dust.

Inform behaves no differently. If you make a typing or syntax mistake, the compiler will send you back to revise your work. "It was just a lousy comma!" you cry in disgust. The compiler remains silent. It has nothing to gain by argument, because it's always right. So you go and change the lousy comma. No harm done except perhaps to your pride.

Errors that are found during compilation may be tedious to correct, but are usually easy to find; after all, the compiler tries politely to point out what and where the mistake was. Trouble begins after you've managed to satisfy all of the compiler's complaints. You are rewarded by a clean screen, devoid of a list of errors, and you are offered – a gift!

A new file has appeared in your folder. A story file. Yes, *the* game. You quickly open your favourite interpreter and begin to play – only to discover the dark side of errors, the bugs. Bugs come in all shapes, colours and sizes: big, small, stupid, absurd, minor, disturbing, nerve-wracking and catastrophic. They are often unpredictable: they regale our eyes with surprising, unexpected behaviour. They defy logic: I can TAKE the key, and the game even says "Taken", but the key remains in the same place and won't appear in my inventory. Or: opening the door while wearing the fur coat causes a programming error and a cryptic message "tried to find the attribute of nothing". And many, many others.

When designing a game you try to take into consideration the states that your objects will find themselves in, but any medium-sized game has such a number of objects and actions that it's almost impossible to think of all the possible variations, permutations and possibilities.

Debugging consists in finding run-time errors, and then correcting them. Pretty easy, you might think, but no. Detection of such errors is not straightforward, since they tend to manifest themselves only under precise circumstances. Then you have to investigate your code to find out what is causing them. And then, if you discover the offending lines, you must make the appropriate changes. (There is also the case when you can't find the mistake. Don't worry, it's there somewhere. Persistence always pays off in the end.)

To help you out in this daunting task, Inform has a stock of special actions: the debugging verbs. They become available at run-time when the source file is compiled in **Debug mode** (`-D` switch) or in **Strict mode** (`-S` switch – which

includes Debug mode). In fact, the compiler has Strict mode on by default as a safety device, for it checks your code more carefully for some additional errors. When you are ready to release your game, you'll have to recompile, switching off Strict mode (`-~S`) to avoid allowing the players to benefit from the debugging verbs. We'll cover briefly a few of these actions, and tell you what they do.

Command lists

The only way to test a game is to play it. As you make progress writing code, the game grows complicated, and it becomes really tiresome to repeat all the commands every time you play. Not unusually, when you fix the behaviour of some object, you are also affecting the behaviour of other objects or actions, so it's a good idea to test everything now and then; you have to make sure that your recent changes and fixes didn't spoil something that previously worked fine.

The RECORDING command (RECORDING ON and RECORDING OFF) saves the commands that you type as you play into a text file (you'll probably be prompted for a file name). When you add a new section to the game, you can play to that point, type RECORDING ON to capture (in another file) the commands which exercise that section, and then later use your editor to append those new commands to the existing list.

The REPLAY command runs the text file created by RECORDING, playing all the stored commands in one go. This way you can very quickly check whether everything is working as it should.

You can open the file of commands with any text editor program and modify the contents as need arises: for instance, if you want to delete some commands no longer necessary because of a change to the game, or if you forgot to test some particular object and you need to add new commands.

This technique (the use of recorded lists of commands) is, and we can't emphasise it too strongly, one of the most useful testing features for a game designer.

Spill them guts

Some debugging verbs offer information about the current state of things.

TREE

> This action lists all the objects in the game and how they contain each other. You can discover the possessions of just one object by typing TREE *object*. All the objects that you have defined in the source file are turned into numbers by Inform when it compiles the story file; this command also lists those internal `obj_id` numbers.

SHOWOBJ *object*

Displays information about the *object*, the attributes it currently has and the value of its properties. The *object* can be anywhere, not necessarily in scope. You can also type the object number, if the object in question does not have a name. For instance, in “Heidi”:

```
>SHOWOBJ NEST
Object "bird's nest" (29) in "yourself"
  has container moved open workflag
  with name 'bird's' 'nest' 'twigs' 'moss',
       description "The nest is carefully woven of twigs and moss." (19230),
```

SHOWVERB *verb*

Displays the grammar of the *verb*, just like a standard `Verb` definition. This comes in handy when you have tampered with `Extend` and are not sure about the final results of your machinations. An example from “William Tell”:

```
>SHOWVERB GIVE
Verb 'feed' 'give' 'offer' 'pay'
    * held 'to' creature -> Give
    * creature held -> Give reverse
    * 'over' held 'to' creature -> Give
    * 'homage' 'to' noun -> Salute
```

The first lines reproduce the verb definition as it’s written in the library. The last line, however, is the direct consequence of our tailored `Extend`:

```
Extend 'give'
    * 'homage' 'to' noun            -> Salute;
```

SCOPE

Lists all of the objects currently in scope (in general terms, visible to the player character). More powerfully, you can type SCOPE *object* to discover which objects are in scope for the named *object*. This feature becomes useful when you have NPCs capable of tampering with their surroundings.

What on earth is going on?

There comes the time when some actions don’t produce the desired effects and you don’t know why. The following debugging verbs offer information about what the interpreter is up to, which might enable you to identify the moment when things started to go awry.

ACTIONS (or ACTIONS ON) and ACTIONS OFF

Gives information about all the actions going on. Some actions get redirected to others, and this becomes at times a source of mischief and mystery; here you get a clue what’s happening. For example, take this transcript from “William Tell”:

```
Further along the street
People are still pushing and shoving their way from the southern gate towards
the town square, just a little further north. You recognise the owner of a fruit
and vegetable stall.

Helga pauses from sorting potatoes to give you a cheery wave.

>SEARCH STALL
[ Action Search with noun 35 (fruit and vegetable stall) ]
[ Action Examine with noun 35 (fruit and vegetable stall) (from < > statement) ]
It's really only a small table, with a big heap of potatoes, some carrots and
turnips, and a few apples.
...
```

CHANGES (or CHANGES ON) and CHANGES OFF

Tracks object movements, and changes to properties and attributes:

```
Middle of the square
There is less of a crush in the middle of the square; most people prefer to
keep as far away as possible from the pole which towers here, topped with that
absurd ceremonial hat. A group of soldiers stands nearby, watching everyone who
passes.

>GO NORTH
[Setting Middle of the square.warnings_count to 1]
A soldier bars your way.

"Oi, you, lofty; forgot yer manners, didn't you? How's about a nice salute for
the vogt's hat?"

>AGAIN
[Setting Middle of the square.warnings_count to 2]

"I know you, Tell, yer a troublemaker, ain't you? Well, we don't want no bovver
here, so just be a good boy and salute the friggin' hat. Do it now: I ain't
gonna ask you again..."

>SALUTE HAT
[Setting wooden pole.has_been_saluted to 1]
You salute the hat on the pole.

"Why, thank you, sir," sneers the soldier.

>GO SOUTH
[Setting Middle of the square.warnings_count to 0]
[Setting wooden pole.has_been_saluted to 0]
[Moving yourself to South side of the square]
...
```

TIMERS (or TIMERS ON) and TIMERS OFF

This verb shows you the state of all active timers and daemons at the end of each turn. We haven't mentioned timers – similar to daemons – in this guide; you might perhaps use one to explode a bomb ten turns after lighting its fuse.

TRACE (or TRACE ON), TRACE *number* and TRACE OFF

If you turn on this powerful verb, you'll be able to follow the activity of the **parser** – that part of the library which tries to make sense of what the player types – and this will indeed be a wonderful moment of gratitude that someone else took the trouble of writing it. Since the parser does so many things, you can decide the level of detail about the displayed information with the *number* parameter, which can go from 1 (minimum info) to 5 (maximum info). By default, TRACE ON and TRACE with no number sets level 1. Trace level 1 shows the grammar line that the parser is thinking about, while level 2 shows each individual token on each grammar line that it tries. The information displayed with higher levels may become quite hacky, and you are advised to use this feature only if nothing else helps.

Super-powers

GONEAR *object*

This action lets you teleport to the room where the *object* is. This is useful when, for example, certain parts of the map are closed until the player character solves some puzzle, or if the game map is divided in different areas. If the room you want to visit has no objects, you can use...

GOTO *number*

Teleports you to the room with that internal *number*. Since rooms usually have no name, you'll have to discover the internal number of the room object (with the command TREE, for instance).

PURLOIN *object*

PURLOIN works exactly as TAKE, with the nice addition that it doesn't matter where the object is: in another room, inside a locked container, in the claws of the bloodthirsty dragon. More dangerously, it doesn't matter if the object is takeable, so you may purloin `static` or `scenery` objects. PURLOIN is useful in a variety of situations, basically when you want to test a particular feature of the game that requires the player character to have some objects handy. Instead of tediously collecting them, you may simply PURLOIN them. Be careful: it's unwise to PURLOIN objects not meant to be taken, as the game's behaviour may become unpredictable.

ABSTRACT *object* TO *object*

This verb enables you to move the first *object* to the second *object*. As with PURLOIN, both objects can be anywhere in the game. Bear in mind that the second object should logically be a `container`, a `supporter`, or something `animate`.

Infix: the harlot's prerogative

The basic debugging verbs are fairly versatile, and have the advantage that they're automatically compiled into your game unless you explicitly request otherwise. Occasionally though, you'll meet a bug which you simply can't catch using regular techniques, and that's when you might want to investigate the Infix debugger. You'll need to compile using the `-X` switch, and you'll then be able to monitor and modify almost all of your game's data and objects. For instance, you can use "`;`" to inspect – and change – a variable:

```
Inside Benny's cafe
Benny's offers the FINEST selection of pastries and sandwiches. Customers clog
the counter, where Benny himself manages to serve, cook and charge without
missing a step. At the north side of the cafe you can see a red door connecting
with the toilet.

>; deadflag
; == 0

>; deadflag = 4
; == 4

    *** You have been SHAMEFULLY defeated ***

In that game you scored 0 out of a possible 2, in 2 turns.
```

It's often quite maddening to realise that some variable is still `false` because the Chalk puzzle didn't work properly, and that you can't test the Cheese puzzle until the variable becomes `true`. Rather than quit, fix the Chalk, recompile, play back to the current position and only *then* tackle the Cheese, how much easier to just change the variable in mid-stream, and carry right on.

You can use `;WATCH` to see an object's values changing:

```
>;WATCH MID_SQUARE
; Watching object "Middle of the square" (43).

>NORTH
[Moving yourself to Middle of the square]
[Moving local people to Middle of the square]
[Moving Gessler's soldiers to Middle of the square]
[Moving your son to Middle of the square]

Middle of the square
There is less of a crush in the middle of the square; most people prefer to
keep as far away as possible from the pole which towers here, topped with that
absurd ceremonial hat. A group of soldiers stands nearby, watching everyone who
passes.
[Giving Middle of the square visited]

>NORTH
[ "Middle of the square".before() ]
[ mid_square.before() ]
[Setting Middle of the square.warnings_count to 1]
A soldier bars your way.
```

```
"Oi, you, lofty; forgot yer manners, didn't you? How's about a nice salute for
the vogt's hat?"

>NORTH
[ "Middle of the square".before() ]
[ mid_square.before() ]
[Setting Middle of the square.warnings_count to 2]

"I know you, Tell, yer a troublemaker, ain't you? Well, we don't want no bovver
here, so just be a good boy and salute the friggin' hat. Do it now: I ain't
gonna ask you again..."

>NORTH
[ "Middle of the square".before() ]
[ mid_square.before() ]
[Setting Middle of the square.warnings_count to 3]

"OK, Herr Tell, now you're in real trouble.
...
```

Infix is quite complex; it's good to have available, but it's not really a tool for novices. If you do use it, be careful: you get both the power *and* the responsibility to use it sensibly. Remember that the changes affect only the current story file while it's running; to make permanent amendments, you still need to edit the source file.

You won't need it often, but Infix can sometimes provide quick answers to tricky problems.

No matter what

Your game will still have some undetected bugs despite all your efforts to clean it up. This is normal, even for experienced designers; don't feel discouraged or demoralised. You might find it reassuring to know that our own example games in this guide – which certainly don't qualify as "complex programming" – were far from perfect at the First Edition. We blush at the following report from an extremely diligent play-tester:

> I found these things when playing "Captain Fate":
>
> - player is able to wear clothes over the costume,
> - player can change into costume in the dark unlocked bathroom without being interrupted,
> - player can drop clothes in the dark unlocked bathroom. Try REMOVE CLOTHES. X SELF. REMOVE COSTUME. INV – X SELF says that you are wearing the costume, but the inventory does not reflect this.

Fortunately, the code we've offered you in *this* edition takes care of those embarrassing issues, but it might very well happen that a few more undetected absurdities pop up from now on.

The final stage of debugging must happen elsewhere, at the hands of some wilful, headstrong and determined beta-testers; these are the people who, if you're lucky, will methodically tear your game to shreds and make extensive reports of things that don't work reliably, things that don't work as smoothly as they might, things that ought to work but don't, things that never even crossed your mind (like, uh, dropping the costume in the dark). Once you think your game is finished – in that it does all that you think it should, and you've run out of ideas on how else to test it – look for a few beta-testers; three or four is good. The IF community offers some beta-testing resources, or you can always ask in RAIF for kind souls willing to have a go at your game. Remember the golden rules:

- **Expect no mercy**. Although it hurts, a merciless approach is what you need at this time; much better to discover your errors and oversights now, before you release the game more widely. And don't forget to acknowledge your testers' assistance somewhere within the game.
- **Never say never**. If your testers suggest that the game should respond better to an attempted action, don't automatically respond with "No one's going to try that!" They already have, and will again – be grateful for your testers' devious minds and twisted psyches. Although a normal player won't try *all* of those oddball things, every player is bound to try at least *one*, and their enjoyment will be greater, the reality enhanced, if the game "understands".
- **Ask for more**. Don't treat your testers simply as validators of your programming skills, but rather as reviewers of your storytelling abilities. Encourage them to comment on how well the pieces fit together, and to make suggestions – small or radical – for improvement; don't necessarily reject good ideas just because implementing them "will take too long". For example: "the scene in the Tower of London doesn't somehow seem to belong in an Arabian Nights game", or "having to solve three puzzles in a row just to discover the plate of sheep's eyes is a little over the top", or "this five-room trek across the desert really is a bit dull; perhaps you could add a quicksand or something to liven it up?", or "the character of the eunuch in the harem seems to be lacking in something". That is, view the testers collectively not as simple spell-checkers, but rather as collaborative editors on your latest novel.

17 • *** You have won ***

I might just as well have saved the labor and sweat I had put into trying to make my reports harmless. They didn't fool the Old Man. He gave me merry hell.
– The Continental Op in Dashiell Hammett's *Red Harvest.*

Just a few final words to round things off. All that remains are the appendices, with terse but comprehensive summaries of the Inform language and its IF library, plus the source code and run-time transcripts of the games we have developed here. Our "labor and sweat" have been oriented towards making your introduction to Inform as harmless as possible, but this probably won't fool you for long. Although we believe we have covered the system's basic functionality and given you enough grounding to feel comfortably sure-footed as you roam the designing wilderness, there are still many techniques to be mastered and additional aspects to be learnt, including medium and advanced features at which we have not even hinted.

Before you give us merry hell, however, be reassured that the remaining lore, which may at times feel obscure and enigmatic, is fundamentally constructed around the principles that you have already seen. You should now be ready to browse through other documentation and resources without them seeming full of inscrutable hieroglyphs; on the contrary, you'll be able to focus on those bits you don't know about (which now, we hope, will be rather less abundant). Inform, like other powerful and flexible IF design tools, is prepared to cope with the needs of demanding authors: "I don't like the way it handles the TAKE ALL command; I wanna change it." And so you can. "I'd prefer the listings of objects organised in a prettier way." Go right ahead. "I want to have a better social life thanks to Inform." No problem, but you'll have to be one damn charming designer. Oh, well.

Inform has been designed to let you do simple things intuitively and quickly. Left to its own devices, it offers a wide range of default functionality, and we've seen that it's also easy to alter some of its standard behaviour. The desirable goal is for you to reach a state of such familiarity with the system that you can concentrate on designing your games. By "such familiarity" we are not implying that you should know the innards of the library inside out; such people exist, but they're few and far between. However, once you become reasonably proficient at typing in code, with a knowledge level similar to the one provided by this guide, a careful look at the appropriate section of the *Inform Designer's Manual* should help you through most difficulties. Admittedly, there are problems and *problems*, from the slap-on-the-head trifle to the teeth-gnashing nightmare. We advise you to put the nightmares on hold for the time being. It may be that one day you discover that their fangs were not as sharp as they seemed.

There are many interesting topics that you could pursue next. Here are a few:

- **Score**: we have seen two ways of scoring a game, but you may decide that scores have no meaning in your game. And there is yet a third built-in system for defining "tasks" worthy of reward, from "wearing the ridiculous bonnet at the Ambassador's party" to "convincing the unfriendly monkey to play the upright piano". This technique requires a bit of knowledge about...
- **Arrays**: these are enumerated lists of variables. Instead of having just one variable to play with, you can have a collection of them, indexed by number.
- **Lists and inventories**: there are many functions to let you arrange the way objects are grouped and presented to the player at run-time.
- **Vehicles**: cars, elevators, hot-air balloons, magic carpets, spaceships – or any other device in which the player may travel around.
- **Create verbs and vocabulary**: although we have already nibbled at this concept, you can fine-tune the parser to allow for all sorts of amazing commands (from magical utterances that trigger unfathomable spells, to special actions that affect many objects at once).
- **Changing the player**: who says that the player character must be a boring human being? Metamorphose the unsuspecting mortal into a virtual-reality proxy, a fantastic animal, an untouchable ghost, a powerful telepath or a telekinetic vampire. Undecided about which one? Make your game with multiple starring characters and switch between them when you want.
- **Passing of time, timed machines and events**: set a timer that ticks away, unbeknown to the player and attach it to a bomb; a door which opens only once every ten turns; a dragon with short fuse and little patience; a marching patrol of soldiers; a clock that ominously chimes the arrival of sunset and doom. Change the "turns" count on the status line into minutes, or days.
- **Mutable directions**: north is north? Not necessarily. Change the direction objects of the game to "forward", "back", and so on. You are on a ship? "fore" and "aft", "port" and "starboard" may be the thing for you. Enter a mirror and have the map and all the directions reflected.
- **Complex NPCs**: how unpredictable can the behaviour of that impertinent butler be? Can he talk, move, steal your possessions, poison your tea? Does he react coherently to the player's actions? Does he have a hidden agenda of his own? Although NPC creation is indeed a knotty craft, it's one worth mastering. "Living" NPCs increase immensely the reality of your games.
- **Techie features**: change the status line, or the command prompt. Clear the screen, or alter its colour; centre text upon it, and colour the text as well. Wait for the player to press a key and then trigger some action. Display a message one letter at a time. Add a tiny compass showing available exits at all times.

Interactive fiction mixes creativity and narrative skills with coding expertise. Usually, those games which make the biggest impact have a fair amount of both. If you feel yourself lacking one of these qualities at present, contemplate a little teamwork: there are IF collaboration lists on the Internet, where people offer to lend a hand with ideas or programming (and some very good games have come from the mixed efforts of a well-tuned collaboration). Above all, don't forget the importance of beta-testing, which may produce the feedback inspiring you to turn your decent attempt into a killing machine. There's little as obnoxious to players as a game which is obviously under-tested. Getting those bugs out is your responsibility; be sure to clean it as best you can, but never *ever* release a game until it has been kicked around by others. And remember that beta-testers are (almost certainly) experienced players, so their advice beyond the call of bug-hunting is as priceless counsel as you are likely to get. Encourage them to comment on your achievements in both programming *and* design.

Now: where to go, what to do? Allow us to insist one last time on the importance of reading the *Inform Designer's Manual*, an excellent book in all respects. While you are at it, write small games, training exercises; we don't advise you to try an epic saga for your first scenario, but if nothing else will work for you – the Think Big approach – don't let us deter you. It's a good idea to play other people's games, because you'll know the average level that players may expect; check the newsgroups for comments on good titles. Be sure around September to keep an eye open for the Interactive Fiction Competition (`http://www.ifcomp.org/`), an annual showcase for short(ish) works.

And, who knows? It might be that next year we'll all be smashed by *your* entry.

Sonja and Roger

Appendix A • How to play an IF game

Playing IF requires just a bit of instruction. All you have to do is read the descriptions and situations that appear on the screen and then tell the game what you'd like to happen next. Imagine that you're saying "I WANT TO ..."; you don't actually type those three words, but you *do* type what follows, instructing the game to do something on your behalf. Commands usually take the form of a simple imperative sentence, with a verb and a direct object (for example, typing EXAMINE THE KETTLE will display a description of the kettle, TAKE KETTLE will make it one of your belongings, and so on). If there's more than one kettle around, you can be specific (TAKE RED KETTLE); otherwise, the game will ask you something like "Which do you mean, the red kettle or the rusty kettle?" Answering RED is enough in a case like this. Some commands refer to two objects, like: PUT KETTLE ON TABLE.

To make them stand out on the page, we're showing the words that you type in capital letters. You can actually use upper-case or lower-case letters – it makes no difference – and you can usually omit words like THE (though TAKE A BATH and TAKE THE BATH may have different effects, as will TAKE A COIN and TAKE THE COIN if there are several to choose from).

To move around, use the verb GO and one of the cardinal points: GO NORTH will move you in the desired direction. Movement happens quite a lot, so you can shorten that to just NORTH, and you can even use the initial(s) of the direction in which you want to travel (easier and faster to type): N, S, E, W, NE, NW, SE and SW. Also available are UP (U), DOWN (D) and, occasionally, IN and OUT.

There is quite an impressive stock of standard actions which can generally be relied upon to do something, even if only to tell you that you're wasting your time:

ASK	DROP	INVENTORY	PUSH	SWIM	TRANSFER
BURN	EAT	JUMP	PUT	SWITCH OFF	TURN
BUY	EMPTY	KILL	READ	SWITCH ON	UNLOCK
CLEAN	ENTER	KISS	SEARCH	TAKE	WAIT
CLIMB	EXAMINE	LISTEN	SHOW	TASTE	WAVE
CLOSE	EXIT	LOCK	SING	TELL	WEAR
CUT	FILL	LOOK	SIT	THINK	
DIG	GIVE	OPEN	SLEEP	THROW	
DISROBE	GO	PRAY	SMELL	TIE	
DRINK	INSERT	PULL	STAND	TOUCH	

You don't have to play IF with a list like this open in front of you; the idea is that a good game should understand whatever seems logical for you to try next. Sometimes that will be a standard action, sometimes a verb like SALUTE or PHOTOGRAPH which, although less common, makes perfect sense in context.

You'll discover that usually many of these actions are fairly irrelevant. Try logical things first (if you have a torch, BURN may be promising, while EAT probably not). Of special interest are LOOK (or just L), to print a description of the current location; EXAMINE (or X) *object*, which gives you a detailed description of the object; INVENTORY (INV or I) lists the objects you are carrying.

You may combine some of these verbs with prepositions to expand the possibilities: LOOK THROUGH, LOOK AT, LOOK IN, LOOK UNDER all perform different actions. Remember that we're mentioning only a selection of the possible verbs; if you feel that something else ought to work, try it and see.

You can change the way the game offers descriptions of locations as you arrive in them. The default setting is usually BRIEF, which provides you with long descriptions only the first time you enter a new location. Some people like to change this to VERBOSE, which *always* gives you long location descriptions. Here are some other special commands and abbreviations you should know:

AGAIN (G) repeats the action you've just performed.
WAIT (Z) skips one turn of action while you loiter and see what happens.
QUIT ends the game.
SAVE saves your current position in the game.
RESTORE reloads a previously saved position.
RESTART starts again from the beginning.
SCORE tells you the current state of progress.
UNDO goes back one turn so that your most recent action never happened.

Often, there will be characters that you'll have to interact with. Let's suppose you find your cousin Maria: you may ASK (or TELL) MARIA ABOUT *something*, GIVE (or SHOW) *object* TO MARIA or ASK MARIA FOR *object*. Characters may be willing to help you, when you can indicate your wishes with: MARIA, GO NORTH or MARIA, TAKE THE GUN. If you are really fond of Maria, you may want to KISS her and if she offends you beyond measure, you might like to ATTACK her.

Once you've referred to an object or a character by name, you may use the pronouns IT, HIM or HER to simplify the typing process. These pronouns will remain set until you refer to any other object or character. If you wish to check the current pronoun assignments, type PRONOUNS.

As a rule of the thumb, try to keep your actions simple. Most games will actually understand long commands like TAKE ALL FROM THE BAG EXCEPT THE GREEN PEARL THEN THROW CAMEMBERT CHEESE AT UGLY MATRON, but such things are hard to type without mistakes. Also, you'll find that other inputs don't work as well: GO BACK TO THE KITCHEN or GET NEAR THE SINGING PIRATE or READ NEWSPAPER OVER THE SHERIFF'S SHOULDER will all give you error messages of some kind. Understanding the conventions of command typing is fairly intuitive and you'll quickly master it after a little experimentation.

NOTE: we're talking here about the core capabilities that most Inform games provide (though much of this is equally applicable to other IF systems). Often the designer will have extended these capabilities by defining additional commands appropriate to the nature of the game; either you'll be told about these, or they'll come naturally to mind during play. Less frequently, some designers like to tamper with the default behaviour of the parser, the interface, or with the way that commands work – maybe even disabling some of the standard ones completely. When this happens, it's common and polite practice for the game to let you know.

Appendix B • "Heidi" story

Heidi in the Forest is our first – and simplest – game. We describe it in three chapters: "Heidi: our first Inform game" on page 27, "Reviewing the basics" on page 41 and "Heidi revisited" on page 51. Here is a run-time transcript, and then the original and extended source files.

Transcript of play

```
Heidi
A simple Inform example
by Roger Firth and Sonja Kesserich.
Release 1 / Serial number 020827 / Inform v6.21 Library 6/10 SD

In front of a cottage
You stand outside a cottage. The forest stretches east.

>VERBOSE
Heidi is now in its "verbose" mode, which always gives long descriptions of
locations (even if you've been there before).

>GO EAST

Deep in the forest
Through the dense foliage, you glimpse a building to the west. A track heads to the
northeast.

You can see a baby bird here.

>EXAMINE THE BIRD
Too young to fly, the nestling tweets helplessly.

>TAKE BIRD
Taken.

>NE

A forest clearing
A tall sycamore stands in the middle of this clearing. The path winds southwest
through the trees.

You can see a bird's nest (which is empty) here.

>PUT BIRD IN NEST
You put the baby bird into the bird's nest.

>EXAMINE THE NEST
The nest is carefully woven of twigs and moss.

>TAKE NEST
Taken.

>UP
```

```
At the top of the tree
You cling precariously to the trunk.

You can see a wide firm bough here.

>PUT NEST ON BRANCH
You put the bird's nest on the wide firm bough.

    *** You have won ***

In that game you scored 0 out of a possible 0, in 10 turns.

Would you like to RESTART, RESTORE a saved game or QUIT?
> QUIT
```

Game source code – original version

```
!============================================================================
Constant Story "Heidi";
Constant Headline
            "^A simple Inform example
             ^by Roger Firth and Sonja Kesserich.^";

Constant MAX_CARRIED 1;

Include "Parser";
Include "VerbLib";

!============================================================================
! The game objects

Object  before_cottage "In front of a cottage"
  with  description
            "You stand outside a cottage. The forest stretches east.",
        e_to forest,
  has   light;

Object  forest "Deep in the forest"
  with  description
            "Through the dense foliage, you glimpse a building to the west.
             A track heads to the northeast.",
        w_to before_cottage,
        ne_to clearing,
  has   light;

Object  bird "baby bird" forest
  with  description "Too young to fly, the nestling tweets helplessly.",
        name 'baby' 'bird' 'nestling',
  has   ;

Object  clearing "A forest clearing"
  with  description
            "A tall sycamore stands in the middle of this clearing.
             The path winds southwest through the trees.",
        sw_to forest,
        u_to top_of_tree,
  has   light;
```

```
Object  nest "bird's nest" clearing
  with  description "The nest is carefully woven of twigs and moss.",
        name 'bird^s' 'nest' 'twigs' 'moss',
  has   container open;

Object  tree "tall sycamore tree" clearing
  with  description
            "Standing proud in the middle of the clearing,
             the stout tree looks easy to climb.",
        name 'tall' 'sycamore' 'tree' 'stout' 'proud',
  has   scenery;

Object  top_of_tree "At the top of the tree"
  with  description "You cling precariously to the trunk.",
        d_to clearing,
  has   light;

Object  branch "wide firm bough" top_of_tree
  with  description "It's flat enough to support a small object.",
        name 'wide' 'firm' 'flat' 'bough' 'branch',
        each_turn [; if (nest in branch) deadflag = 2; ],
  has   static supporter;

!================================================================================
! Entry point routines

[ Initialise; location = before_cottage; ];

!================================================================================
! Standard and extended grammar

Include "Grammar";

!================================================================================
```

Game source code – revisited

```
!===============================================================================
Constant Story "Heidi";
Constant Headline
            "^A simple Inform example
             ^by Roger Firth and Sonja Kesserich.^";

Constant MAX_CARRIED 1;

Include "Parser";
Include "VerbLib";

!===============================================================================
! The game objects

Object  before_cottage "In front of a cottage"
  with  description
            "You stand outside a cottage. The forest stretches east.",
        e_to forest,
        in_to "It's such a lovely day -- much too nice to go inside.",
        cant_go "The only path lies to the east.",
  has   light;

Object  cottage "tiny cottage" before_cottage
  with  description "It's small and simple, but you're very happy here.",
        name 'tiny' 'cottage' 'home' 'house' 'hut' 'shed' 'hovel',
        before [; Enter:
            print_ret "It's such a lovely day -- much too nice to go inside.";
        ],
  has   scenery;

Object  forest "Deep in the forest"
  with  description
            "Through the dense foliage, you glimpse a building to the west.
             A track heads to the northeast.",
        w_to before_cottage,
        ne_to clearing,
  has   light;

Object  bird "baby bird" forest
  with  description "Too young to fly, the nestling tweets helplessly.",
        name 'baby' 'bird' 'nestling',
        before [; Listen:
            print "It sounds scared and in need of assistance.^";
            return true;
        ],
  has   ;

Object  clearing "A forest clearing"
  with  description
            "A tall sycamore stands in the middle of this clearing.
             The path winds southwest through the trees.",
        sw_to forest,
        u_to top_of_tree,
  has   light;
```

```
Object  nest "bird's nest" clearing
  with  description "The nest is carefully woven of twigs and moss.",
        name 'bird^s' 'nest' 'twigs' 'moss',
  has   container open;

Object  tree "tall sycamore tree" clearing
  with  description
            "Standing proud in the middle of the clearing,
             the stout tree looks easy to climb.",
        name 'tall' 'sycamore' 'tree' 'stout' 'proud',
        before [; Climb:
            PlayerTo(top_of_tree);
            return true;
        ],
  has   scenery;

Object  top_of_tree "At the top of the tree"
  with  description "You cling precariously to the trunk.",
        d_to clearing,
        after [; Drop:
            move noun to clearing;
            return false;
        ],
  has   light;

Object  branch "wide firm bough" top_of_tree
  with  description "It's flat enough to support a small object.",
        name 'wide' 'firm' 'flat' 'bough' 'branch',
        each_turn [; if (bird in nest && nest in branch) deadflag = 2; ],
  has   static supporter;

!============================================================================
! Entry point routines

[ Initialise; location = before_cottage; ];

!============================================================================
! Standard and extended grammar

Include "Grammar";

!============================================================================
```

Appendix C • "William Tell" story

illiam Tell, our second game, is also very straightforward. See "William Tell: a tale is born" on page 61, "William Tell: the early years" on page 69, "William Tell: in his prime" on page 81 and "William Tell: the end is nigh" on page 91.

Transcript of play

```
The place: Altdorf, in the Swiss canton of Uri. The year is 1307, at which time
Switzerland is under rule by the Emperor Albert of Habsburg. His local governor --
the vogt -- is the bullying Hermann Gessler, who has placed his hat atop a wooden
pole in the centre of the town square; everybody who passes through the square must
bow to this hated symbol of imperial might.

You have come from your cottage high in the mountains, accompanied by your younger
son, to purchase provisions. You are a proud and independent man, a hunter and
guide, renowned both for your skill as an archer and, perhaps unwisely (for his
soldiers are everywhere), for failing to hide your dislike of the vogt.

It's market-day: the town is packed with people from the surrounding villages and
settlements.

William Tell
A simple Inform example
by Roger Firth and Sonja Kesserich.
Release 2 / Serial number 020827 / Inform v6.21 Library 6/10 SD

A street in Altdorf
The narrow street runs north towards the town square. Local folk are pouring into
the town through the gate to the south, shouting greetings, offering produce for
sale, exchanging news, enquiring with exaggerated disbelief about the prices of the
goods displayed by merchants whose stalls make progress even more difficult.

"Stay close to me, son," you say, "or you'll get lost among all these people."

>EXAMINE YOUR SON
A quiet, blond lad of eight summers, he's fast learning the ways of mountain folk.

>GO NORTH

Further along the street
People are still pushing and shoving their way from the southern gate towards the
town square, just a little further north. You recognise the owner of a fruit and
vegetable stall.

Helga pauses from sorting potatoes to give you a cheery wave.

"Hello, Wilhelm, it's a fine day for trade! Is this young Walter? My, how he's
grown. Here's an apple for him -- tell him to mind that scabby part, but the rest's
good enough. How's Frau Tell? Give her my best wishes."
```

```
>INVENTORY
You are carrying:
  an apple
  a quiver (being worn)
    three arrows
  a bow

>TALK TO HELGA
You warmly thank Helga for the apple.

[Your score has just gone up by one point.]

>GIVE THE APPLE TO WALTER
"Thank you, Papa."

[Your score has just gone up by one point.]

>NORTH
```

South side of the square

```
The narrow street to the south has opened onto the town square, and resumes at the
far side of this cobbled meeting place. To continue along the street towards your
destination -- Johansson's tannery -- you must walk north across the square, in the
middle of which you see Gessler's hat set on that loathsome pole. If you go on,
there's no way you can avoid passing it. Imperial soldiers jostle rudely through
the throng, pushing, kicking and swearing loudly.

>EXAMINE THE SOLDIERS
They're uncouth, violent men, not from around here.

>EXAMINE HAT
You're too far away to see any detail.

>N
```

Middle of the square

```
There is less of a crush in the middle of the square; most people prefer to keep
as far away as possible from the pole which towers here, topped with that absurd
ceremonial hat. A group of soldiers stands nearby, watching everyone who passes.

>X HAT
The pole, the trunk of a small pine some few inches in diameter, stands about nine
or ten feet high. Set carefully on top is Gessler's ludicrous black and red leather
hat, with a widely curving brim and a cluster of dyed goose feathers.

>N
A soldier bars your way.

"Oi, you, lofty; forgot yer manners, didn't you? How's about a nice salute for the
vogt's hat?"

>N

"I know you, Tell, yer a troublemaker, ain't you? Well, we don't want no bovver
here, so just be a good boy and salute the friggin' hat. Do it now: I ain't gonna
ask you again..."
```

```
>N

"OK, Herr Tell, now you're in real trouble. I asked you nice, but you was too proud
and too stupid. I think it's time that the vogt had a little word with you."

And with that the soldiers seize you and Walter and, while the sergeant hurries off
to fetch Gessler, the rest drag you roughly towards the old lime tree growing in
the marketplace.

Marketplace near the square
Altdorf's marketplace, close by the town square, has been hastily cleared of stalls.
A troop of soldiers has pushed back the crowd to leave a clear space in front of
the lime tree, which has been growing here for as long as anybody can remember.
Usually it provides shade for the old men of the town, who gather below to gossip,
watch the girls, and play cards. Today, though, it stands alone... apart, that is,
from Walter, who has been lashed to the trunk. About forty yards away, you are
restrained by two of the vogt's men.

Gessler is watching from a safe distance, a sneer on his face.

"It appears that you need to be taught a lesson, fool. Nobody shall pass through
the square without paying homage to His Imperial Highness Albert; nobody, hear me?
I could have you beheaded for treason, but I'm going to be lenient. If you should
be so foolish again, you can expect no mercy, but this time, I'll let you go free...
just as soon as you demonstrate your archery skills by hitting this apple from where
you stand. That shouldn't prove too difficult; here, sergeant, catch. Balance it
on the little bastard's head."

>X GESSLER
Short, stout but with a thin, mean face, Gessler relishes the power he holds over
the local community.

>X WALTER
He stares at you, trying to appear brave and remain still. His arms are pulled back
and tied behind the trunk, and the apple nestles amid his blond hair.

>X APPLE
At this distance you can barely see it.

>SHOOT THE APPLE
Slowly and steadily, you place an arrow in the bow, draw back the string, and take
aim with more care than ever in your life. Holding your breath, unblinking, fearful,
you release the arrow. It flies across the square towards your son, and drives the
apple against the trunk of the tree. The crowd erupts with joy; Gessler looks
distinctly disappointed.

    *** You have won ***

In that game you scored 3 out of a possible 4, in 17 turns.

Would you like to RESTART, RESTORE a saved game or QUIT?
> QUIT
```

Game source code

```
!============================================================================
Constant Story "William Tell";
Constant Headline
            "^A simple Inform example
             ^by Roger Firth and Sonja Kesserich.^";
Release 2;  Serial "020827";    ! for keeping track of public releases

Constant MAX_SCORE = 4;

Include "Parser";
Include "VerbLib";

!============================================================================
! Object classes

Class   Room
  has   light;

Class   Prop
  with  before [;
            Examine: return false;
            default:
                print_ret "You don't need to worry about ", (the) self, ".";
        ],
  has   scenery;

Class   Furniture
  with  before [;
            Take,Pull,Push,PushDir:
                print_ret (The) self, " is too heavy for that.";
        ],
  has   static supporter;

Class   Arrow
  with  name 'arrow' 'arrows//p',
        article "an",
        plural "arrows",
        description "Just like all your other arrows -- sharp and true.",
        before [;
            Drop: print_ret "Much too dangerous to leave lying around.";
        ];

Class   NPC
  with  life [;
            Answer,Ask,Order,Tell:
                print_ret "Just use T[ALK] [TO ", (the) self, "].";
        ],
  has   animate;
```

```
!============================================================================
! The game objects

Room    street "A street in Altdorf"
  with  description [;
            print "The narrow street runs north towards the town square.
                Local folk are pouring into the town through the gate to the
                south, shouting greetings, offering produce for sale,
                exchanging news, enquiring with exaggerated disbelief about
                the prices of the goods displayed by merchants whose stalls
                make progress even more difficult.^";
            if (self hasnt visited)
                print "^~Stay close to me, son,~ you say,
                    ~or you'll get lost among all these people.~^";
        ],
        n_to below_square,
        s_to
            "The crowd, pressing north towards the square,
             makes that impossible.";

Prop    "south gate" street
  with  name 'south' 'southern' 'wooden' 'gate',
        description "The large wooden gate in the town walls is wide open.";

Prop    "assorted stalls"
  with  name 'assorted' 'stalls',
        description "Food, clothing, mountain gear; the usual stuff.",
        found_in street below_square,
  has   pluralname;

Prop    "merchants"
  with  name 'merchant' 'merchants' 'trader' 'traders',
        description
            "A few crooks, but mostly decent traders touting their wares
             with raucous overstatement.",
        found_in street below_square,
  has   animate pluralname;

Prop    "local people"
  with  name 'people' 'folk' 'local' 'crowd',
        description "Mountain folk, just like yourself.",
        found_in [; return true; ],
  has   animate pluralname;
```

```
!------------------------------------------------------------------------------

Room    below_square "Further along the street"
  with  description
            "People are still pushing and shoving their way from the southern
             gate towards the town square, just a little further north.
             You recognise the owner of a fruit and vegetable stall.",
        n_to south_square,
        s_to street;

Furniture   stall "fruit and vegetable stall" below_square
  with  name 'fruit' 'veg' 'vegetable' 'stall' 'table',
        description
            "It's really only a small table, with a big heap of potatoes,
             some carrots and turnips, and a few apples.",
        before [; Search: <<Examine self>>; ],
  has   scenery;

Prop    "potatoes" below_square
  with  name 'potato' 'potatoes' 'spuds',
        description
            "Must be a particularly early variety... by some 300 years!",
  has   pluralname;

Prop    "fruit and vegetables" below_square
  with  name 'carrot' 'carrots' 'turnip' 'turnips' 'apples' 'vegetables',
        description "Fine locally grown produce.",
  has   pluralname;

NPC     stallholder "Helga" below_square
  with  name 'stallholder' 'greengrocer' 'monger' 'shopkeeper' 'merchant'
            'owner' 'Helga' 'dress' 'scarf' 'headscarf',
        description
            "Helga is a plump, cheerful woman,
             concealed beneath a shapeless dress and a spotted headscarf.",
        initial [;
            print "Helga pauses from sorting potatoes
                to give you a cheery wave.^";
            if (location hasnt visited) {
                move apple to player;
                print "^~Hello, Wilhelm, it's a fine day for trade! Is this
                    young Walter? My, how he's grown. Here's an apple for him
                    -- tell him to mind that scabby part, but the rest's good
                    enough. How's Frau Tell? Give her my best wishes.~^";
            }
        ],
        times_spoken_to 0,       ! for counting the conversation topics
        life [;
            Kiss: print_ret "~Ooh, you saucy thing!~";
            Talk: self.times_spoken_to = self.times_spoken_to + 1;
                switch (self.times_spoken_to) {
                    1:  score = score + 1;
                        print_ret "You warmly thank Helga for the apple.";
                    2:  score = score + 1;
                        print_ret "~See you again soon.~";
                    default: return false;
                }
        ],
  has   female proper;
```

```
!------------------------------------------------------------------------------

Room    south_square "South side of the square"
  with  description
            "The narrow street to the south has opened onto the town square,
             and resumes at the far side of this cobbled meeting place.
             To continue along the street towards your destination --
             Johansson's tannery -- you must walk north across the square,
             in the middle of which you see Gessler's hat set on that
             loathsome pole. If you go on, there's no way you can avoid
             passing it. Imperial soldiers jostle rudely through the throng,
             pushing, kicking and swearing loudly.",
        n_to mid_square,
        s_to below_square;

Prop    "pole"
  with  name 'wooden' 'pole',
        description "You're too far away to see any detail.",
        found_in south_square north_square;

Prop    "hat"
  with  name 'hat',
        description "You're too far away to see any detail.",
        found_in south_square north_square;

Prop    "Gessler's soldiers"
  with  name 'soldier' 'soldiers',
        description "They're uncouth, violent men, not from around here.",
        before [;
            FireAt: print_ret "You're outnumbered many times.";
            Talk:   print_ret "Such scum are beneath your contempt.";
        ],
        found_in south_square mid_square north_square marketplace,
  has   animate pluralname proper;

!------------------------------------------------------------------------------

Room    mid_square "Middle of the square"
  with  description
            "There is less of a crush in the middle of the square; most
             people prefer to keep as far away as possible from the pole
             which towers here, topped with that absurd ceremonial hat. A
             group of soldiers stands nearby, watching everyone who passes.",
        n_to north_square,
        s_to south_square,
        warnings_count 0,       ! for counting the soldier's warnings
        before [; Go:
            if (noun == s_obj) {
                self.warnings_count = 0;
                pole.has_been_saluted = false;
            }
            if (noun == n_obj) {
                if (pole.has_been_saluted == true) {
                    print "^~Be sure to have a nice day.~^";
                    return false;
                }   ! end of (pole has_been_saluted)
                else {
                    self.warnings_count = self.warnings_count + 1;
                    switch (self.warnings_count) {
                    1:  print_ret "A soldier bars your way. ^^
```

```
                        ~Oi, you, lofty; forgot yer manners, didn't you?
                        How's about a nice salute for the vogt's hat?~";
                    2:  print_ret "^~I know you, Tell, yer a troublemaker,
                        ain't you? Well, we don't want no bovver here,
                        so just be a good boy and salute the friggin'
                        hat. Do it now: I ain't gonna ask you again...~";
                    default:
                        print "^~OK, ";
                        style underline; print "Herr"; style roman;
                        print " Tell, now you're in real trouble. I asked you
                            nice, but you was too proud and too stupid. I
                            think it's time that the vogt had a little word
                            with you.~
                            ^^
                            And with that the soldiers seize you and Walter
                            and, while the sergeant hurries off to fetch
                            Gessler, the rest drag you roughly towards the
                            old lime tree growing in the marketplace.^";
                        move apple to son;
                        PlayerTo(marketplace);
                        return true;
                }   ! end of switch
                }   ! end of (pole has_NOT_been_saluted)
            }   ! end of (noun == n_obj)
        ];

Furniture   pole "wooden pole" mid_square
  with  name 'wooden' 'pole' 'pine' 'hat' 'black' 'red' 'brim' 'feathers',
        description
            "The pole, the trunk of a small pine some few inches in diameter,
             stands about nine or ten feet high. Set carefully on top is
             Gessler's ludicrous black and red leather hat, with a widely
             curving brim and a cluster of dyed goose feathers.",
        has_been_saluted false,
        before [;
            Salute:
                self.has_been_saluted = true;
                print_ret "You salute the hat on the pole. ^^
                    ~Why, thank you, sir,~ sneers the soldier.";
        ],
  has   scenery;

!------------------------------------------------------------------------------

Room    north_square "North side of the square"
  with  description
            "A narrow street leads north from the cobbled square. In its
             centre, a little way south, you catch a last glimpse of the pole
             and hat.",
        n_to [;
            deadflag = 3;
            print_ret "With Walter at your side, you leave the square by the
                north street, heading for Johansson's tannery.";
        ],
        s_to "You hardly feel like going through all that again.";
```

```
!------------------------------------------------------------------------------

Room    marketplace "Marketplace near the square"
  with  description
            "Altdorf's marketplace, close by the town square, has been hastily
             cleared of stalls. A troop of soldiers has pushed back the crowd
             to leave a clear space in front of the lime tree, which has been
             growing here for as long as anybody can remember. Usually it
             provides shade for the old men of the town, who gather below to
             gossip, watch the girls, and play cards. Today, though, it
             stands alone... apart, that is, from Walter, who has been lashed
             to the trunk. About forty yards away, you are restrained by two
             of the vogt's men.",
        cant_go "What? And leave your son tied up here?";

Object  tree "lime tree" marketplace
  with  name 'lime' 'tree',
        description "It's just a large tree.",
        before [; FireAt:
            if (BowOrArrow(second)) {
                deadflag = 3;
                print_ret "Your hand shakes a little, and your arrow flies
                    high, hitting the trunk a few inches above Walter's
                    head.";
            }
            return true;
        ],
  has   scenery;

NPC     governor "governor" marketplace
  with  name 'governor' 'vogt' 'Hermann' 'Gessler',
        description
            "Short, stout but with a thin, mean face, Gessler relishes the
             power he holds over the local community.",
        initial [;
            print "Gessler is watching from a safe distance,
                a sneer on his face.^";
            if (location hasnt visited)
                print "^~It appears that you need to be taught a lesson,
                    fool. Nobody shall pass through the square without paying
                    homage to His Imperial Highness Albert; nobody, hear me?
                    I could have you beheaded for treason, but I'm going to
                    be lenient. If you should be so foolish again, you can
                    expect no mercy, but this time, I'll let you go free...
                    just as soon as you demonstrate your archery skills by
                    hitting this apple from where you stand. That shouldn't
                    prove too difficult; here, sergeant, catch. Balance it on
                    the little bastard's head.~^";
        ],
        life [;
            Talk: print_ret "You cannot bring yourself to speak to him.";
        ],
```

```
        before [; FireAt:
            if (BowOrArrow(second)) {
                deadflag = 3;
                print_ret "Before the startled soldiers can react, you turn
                    and fire at Gessler; your arrow pierces his heart,
                    and he dies messily. A gasp, and then a cheer,
                    goes up from the crowd.";
            }
            return true;
        ],
  has   male;

!==============================================================================
! The player's possessions

Object  bow "bow"
  with  name 'bow',
        description "Your trusty yew bow, strung with flax.",
        before [;
            Drop,Give: print_ret "You're never without your trusty bow.";
        ]
  has   clothing;

Object  quiver "quiver"
  with  name 'quiver',
        description
            "Made of goatskin, it usually hangs over your left shoulder.",
        before [;
            Drop,Give:
                print_ret "But it was a present from Hedwig, your wife.";
        ],
  has   container open clothing;

Arrow   "arrow" quiver;
Arrow   "arrow" quiver;
Arrow   "arrow" quiver;

NPC     son "your son"
  with  name 'son' 'your' 'boy' 'lad' 'Walter',
        description [;
            if (location == marketplace)
                print_ret "He stares at you, trying to appear brave and
                    remain still. His arms are pulled back and tied behind
                    the trunk, and the apple nestles amid his blond hair.";
            else
                print_ret "A quiet, blond lad of eight summers, he's fast
                    learning the ways of mountain folk.";
        ],
        life [;
            Give:
                score = score + 1;
                move noun to self;
                print_ret "~Thank you, Papa.~";
            Talk:
                if (location == marketplace)
                    print_ret "~Stay calm, my son, and trust in God.~";
                else
                    print_ret "You point out a few interesting sights.";
        ],
```

```
        before [;
            Examine,Listen,Salute,Talk: return false;
            FireAt:
                if (location == marketplace) {
                    if (BowOrArrow(second)) {
                        deadflag = 3;
                        print_ret "Oops! Surely you didn't mean to do that?";
                    }
                    return true;
                }
                else
                    return false;
            default:
                if (location == marketplace)
                    print_ret "Your guards won't permit it.";
                else
                    return false;
        ],
        found_in [; return true; ],
  has   male proper scenery transparent;

Object  apple "apple"
  with  name 'apple',
        description [;
            if (location == marketplace)
                print_ret "At this distance you can barely see it.";
            else
                print_ret "The apple is blotchy green and brown.";
        ],
        before [;
            Drop: print_ret "An apple is worth quite a bit --
                better hang on to it.";
            Eat: print_ret "Helga intended it for Walter...";
            FireAt:
                if (location == marketplace) {
                    if (BowOrArrow(second)) {
                        score = score + 1;
                        deadflag = 2;
                        print_ret "Slowly and steadily, you place an arrow in
                            the bow, draw back the string, and take aim with
                            more care than ever in your life. Holding your
                            breath, unblinking, fearful, you release the
                            arrow. It flies across the square towards your
                            son, and drives the apple against the trunk of
                            the tree. The crowd erupts with joy;
                            Gessler looks distinctly disappointed.";
                    }
                    return true;
                }
                else
                    return false;
        ];
```

```
!============================================================================
! Entry point routines

[ Initialise;
    location = street;
    lookmode = 2;       ! like the VERBOSE command
    move bow to player;
    move quiver to player; give quiver worn;
    player.description =
        "You wear the traditional clothing of a Swiss mountaineer.";
    print_ret "^^
        The place: Altdorf, in the Swiss canton of Uri. The year is 1307,
        at which time Switzerland is under rule by the Emperor Albert of
        Habsburg. His local governor -- the vogt -- is the bullying
        Hermann Gessler, who has placed his hat atop a wooden pole in
        the centre of the town square; everybody who passes through the
        square must bow to this hated symbol of imperial might.
        ^^
        You have come from your cottage high in the mountains,
        accompanied by your younger son, to purchase provisions. You are
        a proud and independent man, a hunter and guide, renowned both
        for your skill as an archer and, perhaps unwisely (for his soldiers
        are everywhere), for failing to hide your dislike of the vogt.
        ^^
        It's market-day: the town is packed with people from the
        surrounding villages and settlements.^";
];

[ DeathMessage; print "You have screwed up a favourite folk story"; ];

!============================================================================
! Standard and extended grammar

Include "Grammar";

!----------------------------------------------------------------------------

[ TalkSub;
    if (noun == player) print_ret "Nothing you hear surprises you.";
    if (RunLife(noun,##Talk) ~= false) return;  ! consult life[; Talk: ]
    print_ret "At the moment, you can't think of anything to say.";
];

Verb 'talk' 't//' 'converse' 'chat' 'gossip'
    * 'to'/'with' creature          -> Talk
    * creature                      -> Talk;
```

```
!------------------------------------------------------------------------------
[ BowOrArrow o;
    if (o == bow or nothing || o ofclass Arrow) return true;
    print "That's an unlikely weapon, isn't it?^";
    return false;
];

[ FireAtSub;
    if (noun == nothing)
        print_ret "What, just fire off an arrow at random?";
    if (BowOrArrow(second))
        print_ret "Pretty dangerous, don't you think?";
];

Verb 'fire' 'shoot' 'aim'
    *                               -> FireAt
    * noun                          -> FireAt
    * 'at' noun                     -> FireAt
    * 'at' noun 'with' noun         -> FireAt
    * noun 'with' noun              -> FireAt
    * noun 'at' noun                -> FireAt reverse;

!------------------------------------------------------------------------------

[ SaluteSub;
    if (noun has animate) print_ret (The) noun, " acknowledges you.";
    print_ret (The) noun, " takes no notice.";
];

Verb 'bow' 'nod' 'kowtow' 'genuflect'
    * 'at'/'to'/'towards' noun      -> Salute;

Verb 'salute' 'greet' 'acknowledge'
    * noun                          -> Salute;

Extend 'give'
    * 'homage' 'to' noun            -> Salute;

Extend 'wave'
    * 'at' noun                     -> Salute;


!------------------------------------------------------------------------------

[ UntieSub; print_ret "You really shouldn't try that."; ];

Verb 'untie' 'unfasten' 'unfix' 'free' 'release'
    * noun                          -> Untie;

!==============================================================================
```

Compile-as-you-go

Your understanding of how the "William Tell" game works will be considerably enhanced if you type in the code for yourself as you read through the guide. However, it takes us four chapters to describe the game, which isn't complete and playable until the end of Chapter 9. Even if you make no mistakes in your typing, the game won't compile without errors before that point, because of references in earlier chapters to objects which aren't presented until later chapters (for example, Chapter 6 mentions the `bow` and `quiver` objects, but we don't define them until Chapter 7). This is a bit of a nuisance, because as a general rule we advise you to compile frequently – more or less after every change you make to a game – in order to detect syntax and spelling mistakes as soon as possible.

Fortunately, there's a fairly easy way round the difficulty, though it involves a little bit of cheating. The trick is temporarily to add minimal definitions – often called "stubs" – of the objects whose full definitions have yet to be provided.

For example, if you try to compile the game in the state that it's reached by the end of Chapter 6, you'll get this:

```
Tell.inf(16): Warning:  Class "Room" declared but not used
Tell.inf(19): Warning:  Class "Prop" declared but not used
Tell.inf(27): Warning:  Class "Furniture" declared but not used
Tell.inf(44): Error:  No such constant as "street"
Tell.inf(46): Error:  No such constant as "bow"
Tell.inf(47): Error:  No such constant as "quiver"
Compiled with 3 errors and 3 warnings
```

However, by adding these lines to the end of your game file:

TYPE

```
! ============================================================
! TEMPORARY DEFINITIONS NEEDED TO COMPILE AT THE END OF CHAPTER 6

Room    street;
Object  bow;
Object  quiver;
```

a compilation should now give only this:

```
Tell.inf(19): Warning:  Class "Prop" declared but not used
Tell.inf(27): Warning:  Class "Furniture" declared but not used
Compiled with 2 warnings
```

That's a lot better. It's not worth worrying about those warnings, since it's easy to understand where they come from; anyway, they'll go away shortly. More important, there are no errors, which means that you've probably not made any major typing mistakes. It also means that the compiler has created a story file, so you can try "playing" the game. If you do, though, you'll get this:

```
William Tell
A simple Inform example
by Roger Firth and Sonja Kesserich.
Release 2 / Serial number 020827 / Inform v6.21 Library 6/10 SD

(street)
** Library error 11 (27,0) **
** The room "(street)" has no "description" property **

>
```

Whoops! We've fallen foul of Inform's rule saying that every room must have a `description` property, to be displayed by the interpreter when you enter that room. Our `street` stub hasn't got a `description`, so although the game compiles successfully, it still causes an error to be reported at run-time.

The best way round this is to extend the definition of our `Room` class, thus:

TYPE

```
Class   Room
  with  description "UNDER CONSTRUCTION",
  has   light;
```

By doing this, we ensure that *every* room has a description of some form; normally we'd override this default value with something meaningful – "The narrow street runs north towards the town square..." and so on – by including a `description` property in the object's definition. However, in a stub object used only for testing, a default description is sufficient (and less trouble):

```
William Tell
A simple Inform example
by Roger Firth and Sonja Kesserich.
Release 2 / Serial number 020827 / Inform v6.21 Library 6/10 SD

(street)
UNDER CONSTRUCTION

>INVENTORY
You are carrying:
  a (quiver) (being worn)
  a (bow)

>EXAMINE QUIVER
You can't see any such thing.

>
```

You'll notice a couple of interesting points. Because we didn't supply external names with our `street`, `bow` and `quiver` stubs, the compiler has provided some for us – `(street)`, `(bow)` and `(quiver)` – simply by adding parentheses around the internal IDs which we used. And, because our `bow` and `quiver` stubs have no `name` properties, we can't actually refer to those objects when playing the game. Neither of these points would be acceptable in a finished game, but for testing purposes at this early stage – they'll do.

So far, we've seen how the addition of three temporary object definitions enables us to compile the incomplete game, in its state at the end of Chapter 6. But once

we reach the end of Chapter 7, things have moved on, and we now need a different set of stub objects. For a test compilation at this point, remove the previous set of stubs, and instead add these – `south_square` and `apple` objects, and a dummy action handler to satisfy the `Talk` action in Helga's `life` property:

TYPE

```
! ================================================================
! TEMPORARY DEFINITIONS NEEDED TO COMPILE AT THE END OF CHAPTER 7

Room    south_square;
Object  apple;

[ TalkSub; ];
```

Similarly, at the end of Chapter 8, replace the previous stubs by these if you wish to check that the game compiles:

TYPE

```
! ================================================================
! TEMPORARY DEFINITIONS NEEDED TO COMPILE AT THE END OF CHAPTER 8

Room    marketplace;
Object  apple;
NPC     son;

[ TalkSub; ];
[ FireAtSub; ];
[ SaluteSub; ];
```

Finally, by the end of Chapter 9 the game is complete, so you can delete the stubs altogether.

Used with care, this technique of creating a few minimal stub objects can be convenient – it enables you to "sketch" a portion of your game in outline form, and to compile and test the game in that state, without needing to create complete object definitions. However, you've got very little interaction with your stubs, so don't create too many of them. And of course, never forget to flesh out the stubs into full definitions as soon as you can.

Appendix D • "Captain Fate" story

aptain Fate is our third and final game; it's a little longer and more complex than its predecessors. See "Captain Fate: take 1" on page 105, "Captain Fate: take 2" on page 115, "Captain Fate: take 3" on page 127 and "Captain Fate: the final cut" on page 137.

Transcript of play

```
Impersonating mild mannered John Covarth, assistant help boy at an insignificant
drugstore, you suddenly STOP when your acute hearing deciphers a stray radio call
from the POLICE. There's some MADMAN attacking the population in Granary Park! You
must change into your Captain FATE costume fast...!

Captain Fate
A simple Inform example
by Roger Firth and Sonja Kesserich.
Release 2 / Serial number 020827 / Inform v6.21 Library 6/10 SD

On the street
On one side -- which your HEIGHTENED sense of direction indicates is NORTH --
there's an open cafe now serving lunch. To the south, you can see a phone booth.

>EXAMINE ME
In your secret identity's outfit, you manage most efficaciously to look like a
two-cent loser, a good-for-nothing wimp.

>INVENTORY
You are carrying:
  your clothes (being worn)
  your costume

>X COSTUME
STATE OF THE ART manufacture, from chemically reinforced 100% COTTON-lastic(tm).

>REMOVE CLOTHES
In the middle of the street? That would be a PUBLIC SCANDAL, to say nothing of
revealing your secret identity.

>X BOOTH
It's one of the old picturesque models, a red cabin with room for one caller.

>ENTER IT
With implausible celerity, you dive inside the phone booth.

>REMOVE CLOTHES
Lacking Superman's super-speed, you realise that it would be awkward to change in
plain view of the passing pedestrians.

>OUT
You get out of the phone booth.

On the street
On one side -- which your HEIGHTENED sense of direction indicates is NORTH --
there's an open cafe now serving lunch. To the south, you can see a phone booth.
```

```
>X CAFE
The town's favourite for a quick snack, Benny's cafe has a 50's ROCKETSHIP look.

>ENTER CAFE
With an impressive mixture of hurry and nonchalance you step into the open cafe.

Inside Benny's cafe
Benny's offers the FINEST selection of pastries and sandwiches. Customers clog the
counter, where Benny himself manages to serve, cook and charge without missing a
step. At the north side of the cafe you can see a red door connecting with the
toilet.

>OPEN DOOR
It seems to be locked.

>X IT
A red door with the unequivocal black man-woman silhouettes marking the entrance
to hygienic facilities. There is a scribbled note stuck on its surface.

>READ THE NOTE
You apply your ENHANCED ULTRAFREQUENCY vision to the note and squint in
concentration, giving up only when you see the borders of the note begin to blacken
under the incredible intensity of your burning stare. You reflect once more how
helpful it would've been if you'd ever learnt to read.

A kind old lady passes by and explains: "You have to ask Benny for the key, at the
counter."

You turn quickly and begin, "Oh, I KNOW that, but..."

"My pleasure, son," says the lady, as she exits the cafe.

>X BENNY
A deceptively FAT man of uncanny agility, Benny entertains his customers crushing
coconuts against his forehead when the mood strikes him.

>ASK BENNY FOR THE KEY
"Toilet is only fer customers," he grumbles, looking pointedly at a menu board
behind him.

>X MENU
The menu board lists Benny's food and drinks, along with their prices. Too bad
you've never learnt how to read, but luckily there is a picture of a big cup of
coffee among the incomprehensible writing.

>BENNY,GIVE ME A COFFEE
With two gracious steps, Benny places his world-famous Cappuccino in front of you.

>BENNY,GIVE ME THE KEY
Benny tosses the key to the rest rooms on the counter, where you grab it with a
dextrous and precise movement of your HYPER-AGILE hand.

>UNLOCK DOOR WITH KEY
You unlock the door to the toilet and open it.

>N

Unisex toilet
A surprisingly CLEAN square room covered with glazed-ceramic tiles, featuring
```

little more than a lavatory and a light switch. The only exit is south, through the door and into the cafe.

[Your score has just gone up by one point.]

>CLOSE DOOR
You close the door to the cafe.

It is now pitch dark in here!

>SWITCH ON LIGHT
You turn on the light in the toilet.

Unisex toilet
A surprisingly CLEAN square room covered with glazed-ceramic tiles, featuring little more than a lavatory and a light switch. The only exit is south, through the door and into the cafe.

>LOCK DOOR WITH KEY
You lock the door to the cafe.

>X LAVATORY
The latest user CIVILLY flushed it after use, but failed to pick up the VALUABLE coin that fell from his pants.

>TAKE COIN
You crouch into the SLEEPING DRAGON position and deftly, with PARAMOUNT STEALTH, you pocket the lost coin.

[Your score has just gone up by one point.]

>REMOVE CLOTHES
You quickly remove your street clothes and bundle them up together into an INFRA MINUSCULE pack ready for easy transportation. Then you unfold your INVULNERABLE-COTTON costume and turn into Captain FATE, defender of free will, adversary of tyranny!

>UNLOCK DOOR WITH KEY
You unlock the door to the cafe and open it.

>S

Inside Benny's cafe
Benny's offers the FINEST selection of pastries and sandwiches. Customers clog the counter, where Benny himself manages to serve, cook and charge without missing a step. At the north side of the cafe you can see a red door connecting with the toilet.

Nearby customers glance at your costume with open curiosity.

On the counter is a cup of coffee.

>DRINK COFFEE
You pick up the cup and swallow a mouthful. Benny's WORLDWIDE REPUTATION is well deserved. Just as you finish, Benny takes away the empty cup. "That will be one quidbuck, sir."

>PAY COIN TO BENNY
With marvellous ILLUSIONIST gestures, you produce the coin from the depths of your BULLET-PROOF costume as if it had popped out from Benny's ear! People around you

```
clap politely.  Benny accepts the coin and gives it a SUSPICIOUS bite. "Thank you,
sir. Come back anytime," he says.

>GIVE KEY TO BENNY
Benny nods as you ADMIRABLY return his key.

"Didn't know there was a circus in town," comments one customer to another. "Seems
like the clowns have the day off."

>S
You step onto the sidewalk, where the passing pedestrians recognise the rainbow
EXTRAVAGANZA of Captain FATE's costume and cry your name in awe as you JUMP with
sensational momentum into the BLUE morning skies!

    *** You fly away to SAVE the DAY ***

In that game you scored 2 out of a possible 2, in 32 turns.

Would you like to RESTART, RESTORE a saved game or QUIT?
> QUIT
```

Game source code

```
!============================================================================
Constant Story "Captain Fate";
Constant Headline
            "^A simple Inform example
             ^by Roger Firth and Sonja Kesserich.^";
Release 2;  Serial "020827";    ! for keeping track of public releases

Constant MANUAL_PRONOUNS;
Constant MAX_SCORE     2;
Constant OBJECT_SCORE  1;
Constant ROOM_SCORE    1;

Replace MakeMatch;              ! required by pname.h
Replace Identical;
Replace NounDomain;
Replace TryGivenObject;

Include "Parser";
Include "pname";                ! pname.h is from the Archive

Object  LibraryMessages         ! must be defined between Parser and VerbLib
  with  before [;
            Buy:    "Petty commerce has rarely interested you.";
            Dig:    "Your keen senses detect NOTHING underground worth your
                     immediate attention.";
            Pray:   "You won't need to bother almighty DIVINITIES to save
                     the day.";
            Sing:   "Alas! That is not one of your many superpowers.";
            Sleep:  "A hero is ALWAYS on the watch.";
            Sorry:  "Captain FATE has no time for apologies, only for
                     ACTION.";
            Strong: "An unlikely vocabulary for a HERO like you.";
            Swim:   "You quickly turn all your ATTENTION towards locating a
                     suitable place to EXERCISE your superior strokes,
                     but alas! you find none.";
```

```
            Miscellany:
                if (lm_n == 19)
                    if (clothes has worn)
                        "In your secret identity's outfit, you manage most
                         efficaciously to look like a two-cent loser, a
                         good-for-nothing wimp.";
                    else
                        "Now that you are wearing your costume, you project
                         the image of power UNBOUND, of ballooned,
                         multicoloured MUSCLE, of DASHING yet MODEST chic.";
                if (lm_n == 38)
                    "That's not a verb you need to SUCCESSFULLY save the
                     day.";
                if (lm_n == 39)
                    "That's not something you need to refer to in order to
                     SAVE the day.";
        ];

Include "VerbLib";

!============================================================================
! Object classes

Class   Room
  with  description "UNDER CONSTRUCTION",
  has   light;

Class   Appliance
  with  before [; Take,Pull,Push,PushDir:
            "Even though your SCULPTED adamantine muscles are up to the task,
             you don't favour property damage.";
        ],
  has   scenery;

!============================================================================
! The game objects

Room    street "On the street"
  with  name 'city' 'buildings' 'skyscrapers' 'shops' 'apartments' 'cars',
        description [;
            if (player in booth)
                "From this VANTAGE point, you are rewarded with a broad view
                 of the sidewalk and the entrance to Benny's cafe.";
            else
                "On one side -- which your HEIGHTENED sense of direction
                 indicates is NORTH -- there's an open cafe now serving
                 lunch. To the south, you can see a phone booth.";
        ],
        before [; Go:
            if (player in booth && noun == n_obj) <<Exit booth>>;
        ],
        n_to [; <<Enter outside_of_cafe>>; ],
        s_to [; <<Enter booth>>; ],
        in_to "But which way?",
        cant_go
            "No time now for exploring! You'll move much faster in your
             Captain FATE costume.";
```

```
Object  "pedestrians" street
  with  name 'passing' 'people' 'pedestrians',
        description
            "They're just PEOPLE going about their daily HONEST business.",
        before [;
            Examine: return false;
            default: "The passing pedestrians are of NO concern to you.";
        ],
  has   animate pluralname scenery;

Appliance booth "phone booth" street
  with  name 'old' 'red' 'picturesque' 'phone' 'booth' 'cabin'
            'telephone' 'box',
        description
            "It's one of the old picturesque models, a red cabin with room
             for one caller.",
        before [;
            Open:  "The booth is already open.";
            Close: "There's no way to close this booth.";
        ],
        after [; Enter:
            "With implausible celerity, you dive inside the phone booth.";
        ],
  has   enterable container open;

Appliance "sidewalk" street
  with  name 'sidewalk' 'pavement' 'street',
        article "the",
        description
            "You make a quick surveillance of the sidewalk and discover much
             to your surprise that it looks JUST like any other sidewalk in
             the CITY!";

Appliance outside_of_cafe "Benny's cafe" street
  with  name 'benny^s' 'cafe' 'entrance',
        description
            "The town's favourite for a quick snack, Benny's cafe has a 50's
             ROCKETSHIP look.",
        before [; Enter:
            print "With an impressive mixture of hurry and nonchalance you
                step into the open cafe.^";
            PlayerTo(cafe);
            return true;
        ],
  has   enterable proper;

!------------------------------------------------------------------------------
```

```
Room    cafe "Inside Benny's cafe"
  with  description [;
            print "Benny's offers the FINEST selection of pastries and
                sandwiches. Customers clog the counter, where Benny himself
                manages to serve, cook and charge without missing a step. At
                the north side of the cafe you can see a red door connecting
                with the toilet.";
            if (costume has worn && self.first_time_out == false) {
                self.first_time_out = true;
                StartDaemon(customers);
                print "^^Nearby customers glance at your costume with open
                    curiosity.";
            }
            new_line;
        ],
        first_time_out false,           ! Captain Fate's first appearance?
        before [; Go: if (noun ~= s_obj) return false;
            if (benny.coffee_not_paid == true ||
                benny.key_not_returned == true) {
                print "Just as you are stepping into the street, the big hand
                    of Benny falls on your shoulder.";
                if (benny.coffee_not_paid == true &&
                    benny.key_not_returned == true)
                    "^^~Hey! You've got my key and haven't paid for the
                     coffee. Do I look like a chump?~ You apologise as only a
                     HERO knows how to do and return inside.";
                if (benny.coffee_not_paid == true)
                    "^^~Just waidda minute here, Mister,~ he says.
                     ~Sneaking out without paying, are you?~ You quickly
                     mumble an excuse and go back into the cafe. Benny
                     returns to his chores with a mistrusting eye.";
                if (benny.key_not_returned == true)
                    "^^~Just where you think you're going with the toilet
                     key?~ he says. ~You a thief?~ As Benny forces you back
                     into the cafe, you quickly assure him that it was only
                     a STUPEFYING mistake.";
            }
            if (costume has worn) {
                deadflag = 5;               ! you win!
                "You step onto the sidewalk, where the passing pedestrians
                 recognise the rainbow EXTRAVAGANZA of Captain FATE's costume
                 and cry your name in awe as you JUMP with sensational
                 momentum into the BLUE morning skies!";
            }
        ],
        s_to street,
        n_to toilet_door;

Appliance counter "counter" cafe
  with  name 'counter' 'bar',
        article "the",
        description
            "The counter is made of an astonishing ALLOY of metals, CRUMB- &
             SPILL-RESISTANT and EASY to clean. Customers enjoy their snacks
             with UTTER tranquillity, safe in the notion that the counter can
             take it all.",
  has   supporter;
```

```
Object  food "Benny's snacks" cafe
  with  name 'food' 'pastry' 'pastries' 'sandwich' 'sandwiches' 'snack'
            'snacks' 'doughnut',
        before [; "There is no time for FOOD right now."; ],
  has   scenery proper;

Object  menu "menu" cafe
  with  name 'informative' 'menu' 'board' 'picture' 'writing',
        description
            "The menu board lists Benny's food and drinks, along with their
             prices. Too bad you've never learnt how to read, but luckily
             there is a picture of a big cup of coffee among the
             incomprehensible writing.",
        before [; Take:
            "The board is mounted on the wall behind Benny. Besides, it's
             useless WRITING.";
        ]
  has   scenery;

Object  customers "customers" cafe
  with  name 'customers' 'people' 'customer' 'men' 'women',
        description [;
            if (costume has worn)
                "Most seem to be concentrating on their food, but some do
                 look at you quite blatantly. Must be the MIND-BEFUDDLING
                 colours of your costume.";
            else
                "A group of HELPLESS and UNSUSPECTING mortals, the kind
                 Captain FATE swore to DEFEND the day his parents choked on a
                 DEVIOUS slice of RASPBERRY PIE.";
        ],
        life [;
            Ask,Tell,Answer:
                if (costume has worn)
                    "People seem to MISTRUST the look of your FABULOUS
                     costume.";
                else
                    "As John Covarth, you attract LESS interest than Benny's
                     food.";
            Kiss:
                "There's no telling what sorts of MUTANT bacteria these
                 STRANGERS may be carrying around.";
            Attack:
                "Mindless massacre of civilians is the qualification for
                 VILLAINS. You are SUPPOSED to protect the likes of these
                 people.";
        ],
        orders [;
            "These people don't appear to be of the cooperative sort.";
        ],
```

```
        number_of_comments 0,           ! for counting the customer comments
        daemon [;
            if (location == cafe && random(2) == 1) {
                self.number_of_comments = self.number_of_comments + 1;
                switch (self.number_of_comments) {
                    1:  "^~Didn't know there was a circus in town,~ comments
                         one customer to another. ~Seems like the clowns have
                         the day off.~";
                    2:  "^~These fashion designers don't know what to do to
                         show off,~ snorts a fat gentleman, looking your way.
                         Those within earshot try to conceal their smiles.";
                    3:  "^~Must be carnival again,~ says a man to his wife,
                         who giggles, stealing a peek at you.
                         ~Time sure flies.~";
                    4:  "^~Bad thing about big towns~, comments someone to
                         his table companion, ~is you get the damnedest bugs
                         coming out from toilets.~";
                    5:  "^~I sure WISH I could go to work in my pyjamas,~
                         says a girl in an office suit to some colleagues.
                         ~It looks SO comfortable.~";
                    default: StopDaemon(self);
                }
            }
        ],
  has   scenery animate pluralname;

Object  benny "Benny" cafe
  with  name 'benny',
        description
            "A deceptively FAT man of uncanny agility, Benny entertains his
             customers crushing coconuts against his forehead when the mood
             strikes him.",
        coffee_asked_for false,         ! has player asked for a coffee?
        coffee_not_paid  false,         ! is Benny waiting to be paid?
        key_not_returned false,         ! is Benny waiting for the key?
        life [;
            Give: switch (noun) {
                clothes:
                    "You NEED your unpretentious John Covarth clothes.";
                costume:
                    "You NEED your stupendous ACID-PROTECTIVE suit.";
                toilet_key:
                    self.key_not_returned = false;
                    move toilet_key to benny;
                    "Benny nods as you ADMIRABLY return his key.";
                coin:
                    remove coin;
                    self.coffee_not_paid = false;
                    "With marvellous ILLUSIONIST gestures, you produce the
                     coin from the depths of your BULLET-PROOF costume as if
                     it had popped out from Benny's ear! People around you
                     clap politely.  Benny accepts the coin and gives it a
                     SUSPICIOUS bite. ~Thank you, sir. Come back anytime,~
                     he says.";
            }
```

```
            Attack:
                if (costume has worn) {
                    deadflag = 4;
                    print "Before the horror-stricken eyes of the surrounding
                        people, you MAGNIFICENTLY jump OVER the counter and
                        attack Benny with REMARKABLE, albeit NOT sufficient,
                        speed. Benny receives you with a TREACHEROUS
                        upper-cut that sends your GRANITE JAW flying through
                        the cafe.^^
                        ~These guys in pyjamas think they can bully innocent
                        folk,~ snorts Benny, as the EERIE hands of DARKNESS
                        engulf your vision and you lose consciousness.";
                }
                else
                    "That would be an unlikely act for MEEK John Covarth.";
            Kiss:   "This is no time for MINDLESS infatuation.";
            Ask,Tell,Answer:
                "Benny is too busy for idle chit-chat.";
        ],
        orders [;   ! handles ASK BENNY FOR X and BENNY, GIVE ME XXX
            Give: switch (noun) {
                toilet_key:
                    if (toilet_key in player)
                        "But you DO have the key already.";
                    if (self.coffee_asked_for == true) {
                        move toilet_key to player;
                        self.key_not_returned = true;
                        "Benny tosses the key to the rest rooms on the
                         counter, where you grab it with a dextrous and
                         precise movement of your HYPER-AGILE hand.";
                    }
                    else
                        "~Toilet is only fer customers,~ he grumbles,
                         looking pointedly at a menu board behind him.";
                coffee:
                    if (self.coffee_asked_for == true)
                        "One coffee should be enough.";
                    move coffee to counter;
                    self.coffee_asked_for = true;
                    self.coffee_not_paid = true;
                    "With two gracious steps, Benny places his world-famous
                     Cappuccino in front of you.";
                food:
                    "Food will take too much time, and you must change NOW.";
                menu:
                    "With only the smallest sigh, Benny nods towards the menu
                     on the wall behind him.";
                default:
                    "~I don't think that's on the menu, sir.~";
            }
        ],
  has   scenery animate male proper transparent;
```

```
Object  coffee "cup of coffee" benny
  with  name 'cup' 'of' 'coffee' 'steaming' 'cappuccino'
            'cappucino' 'capuccino' 'capucino',
        initial "On the counter, the steaming Cappuccino awaits you.",
        description [;
            if (self in benny)
                "The picture on the menu board SURE looks good.";
            else
                "It smells delicious.";
        ],
        before [;
            Take,Drink,Taste:
                if (self in benny)
                    "You should ask Benny for one first.";
                else {
                    move self to benny;
                    "You pick up the cup and swallow a mouthful. Benny's
                     WORLDWIDE REPUTATION is well deserved. Just as you
                     finish, Benny takes away the empty cup.
                     ~That will be one quidbuck, sir.~";
                }
            Buy:
                if (coin in player) <<Give coin benny>>;
                else                "You have no money.";
            Smell:
                "If your HYPERACTIVE pituitary glands are to be trusted,
                 it's Colombian.";
        ];

Object  outside_of_toilet "toilet" cafe
  with  name 'toilet' 'bath' 'rest' 'room' 'bathroom' 'restroom',
        before [;
            Enter:
                if (toilet_door has open) {
                    PlayerTo(toilet);
                    return true;
                }
                else
                    "Your SUPERB deductive mind detects that the DOOR is
                     CLOSED.";
            Examine:
                if (toilet_door has open)
                    "A brilliant thought flashes through your SUPERLATIVE
                     brain: detailed examination of the toilet would be
                     EXTREMELY facilitated if you entered it.";
                else
                    "With a TREMENDOUS effort of will, you summon your
                     unfathomable ASTRAL VISION and project it FORWARD
                     towards the closed door... until you remember that it's
                     Dr Mystere who's the one with mystic powers.";
            Open:  <<Open  toilet_door>>;
            Close: <<Close toilet_door>>;
            Take,Push,Pull: "That would be PART of the building.";
        ],
  has   scenery openable enterable;
```

```
Object  toilet_door
  with  pname '.x' 'red' '.x' 'toilet' 'door',
        short_name [;
            if (location == cafe) print "door to the toilet";
            else                  print "door to the cafe";
            return true;
        ],
        description [;
            if (location == cafe)
                "A red door with the unequivocal black man-woman silhouettes
                 marking the entrance to hygienic facilities. There is a
                 scribbled note stuck on its surface.";
            else
                "A red door with no OUTSTANDING features.";
        ],
        found_in cafe toilet,
        before [ ks;
            Open:
                if (self hasnt locked || toilet_key notin player)
                    return false;
                ks = keep_silent; keep_silent = true;
                <Unlock self toilet_key>; keep_silent = ks;
                return true;
            Lock:
                if (self hasnt open) return false;
                print "(first closing ", (the) self, ")^";
                ks = keep_silent; keep_silent = true;
                <Close self>; keep_silent = ks;
                return false;
        ],
        after [ ks;
            Unlock:
                if (self has locked) return false;
                print "You unlock ", (the) self, " and open it.^";
                ks = keep_silent; keep_silent = true;
                <Open self>; keep_silent = ks;
                return true;
            Open:  give toilet light;
            Close: give toilet ~light;
        ],
        door_dir [;
            if (location == cafe) return n_to;
            else                  return s_to;
        ],
        door_to [;
            if (location == cafe) return toilet;
            else                  return cafe;
        ],
        with_key toilet_key,
  has   scenery door openable lockable locked;
```

```
Object  toilet_key "toilet key" benny
  with  pname '.x' 'toilet' 'key',
        article "the",
        invent [;
            if (clothes has worn) print "the CRUCIAL key";
            else                  print "the used and IRRELEVANT key";
            return true;
        ],
        description
            "Your SUPRA PERCEPTIVE senses detect nothing of consequence
             about the toilet key.",
        before [;
            if (self in benny)
                "You SCAN your surroundings with ENHANCED AWARENESS,
                 but fail to detect any key.";
        ];

Object  "scribbled note" cafe
  with  name 'scribbled' 'note',
        description [;
            if (self.read_once == false) {
                self.read_once = true;
                "You apply your ENHANCED ULTRAFREQUENCY vision to the note
                 and squint in concentration, giving up only when you see the
                 borders of the note begin to blacken under the incredible
                 intensity of your burning stare. You reflect once more how
                 helpful it would've been if you'd ever learnt to read.
                 ^^A kind old lady passes by and explains:
                 ~You have to ask Benny for the key, at the counter.~^^
                 You turn quickly and begin, ~Oh, I KNOW that, but...~^^
                 ~My pleasure, son,~ says the lady, as she exits the cafe.";
            }
            else
                "The scorched undecipherable note holds no SECRETS from
                 you NOW! Ha!";
        ],
        read_once false,                ! has the player read the note once?
        before [; Take:
            "No reason to start collecting UNDECIPHERABLE notes.";
        ],
  has   scenery;

!------------------------------------------------------------------------------

Room    toilet "Unisex toilet"
  with  description
            "A surprisingly CLEAN square room covered with glazed-ceramic
             tiles, featuring little more than a lavatory and a light switch.
             The only exit is south, through the door and into the cafe.",
        s_to toilet_door,
  has   ~light scored;
```

```
Appliance light_switch "light switch" toilet
  with  name 'light' 'switch',
        description
            "A notorious ACHIEVEMENT of technological SCIENCE, elegant yet
             EASY to use.",
        before [; Push:
            if (self has on) <<SwitchOff self>>;
            else             <<SwitchOn  self>>;
        ],
        after [;
            SwitchOn:
                give self light;
                "You turn on the light in the toilet.";
            SwitchOff:
                give self ~light;
                "You turn off the light in the toilet.";
        ],
  has   switchable ~on;

Appliance lavatory "lavatory" toilet
  with  name 'lavatory' 'wc' 'toilet' 'loo' 'bowl' 'can' 'john' 'bog',
        before [; Examine:
            if (coin in self) {
                move coin to parent(self);
                "The latest user CIVILLY flushed it after use, but failed to
                 pick up the VALUABLE coin that fell from his pants.";
            }
        ];

Object  coin "valuable coin" lavatory
  with  name 'valuable' 'coin' 'silver' 'quidbuck',
        description "It's a genuine SILVER QUIDBUCK.",
        before [; Drop:
            "Such a valuable coin? Har, har! This must be a demonstration of
             your ULTRA-FLIPPANT jesting!";
        ],
        after [; Take:
            "You crouch into the SLEEPING DRAGON position and deftly, with
             PARAMOUNT STEALTH, you pocket the lost coin.";
        ],
  has   scored;

!============================================================================
! The player's possessions

Object  clothes "your clothes"
  with  name 'ordinary' 'street' 'clothes' 'clothing',
        description
            "Perfectly ORDINARY-LOOKING street clothes for a NOBODY like
             John Covarth.",
        before [;
            Disrobe,Change:
                switch (location) {
                    street:
                        if (player in booth)
                            "Lacking Superman's super-speed, you realise that
                             it would be awkward to change in plain view of
                             the passing pedestrians.";
                        else
                            "In the middle of the street? That would be a
```

```
                         PUBLIC SCANDAL, to say nothing of revealing your
                         secret identity.";
                cafe:
                    "Benny allows no monkey business in his
                     establishment.";
                toilet:
                    if (toilet_door has open)
                        "The door to the bar stands OPEN at tens of
                         curious eyes. You'd be forced to arrest yourself
                         for LEWD conduct.";
                    print "You quickly remove your street clothes and
                        bundle them up together into an INFRA MINUSCULE
                        pack ready for easy transportation. ";
                    if (toilet_door has locked) {
                        give clothes ~worn; give costume worn;
                        "Then you unfold your INVULNERABLE-COTTON costume
                         and turn into Captain FATE, defender of free
                         will, adversary of tyranny!";
                    }
                    else {
                        deadflag = 3;
                        "Just as you are slipping into Captain FATE's
                         costume, the door opens and a young woman
                         enters. She looks at you and starts screaming,
                         ~RAPIST! NAKED RAPIST IN THE TOILET!!!~^^
                         Everybody in the cafe quickly comes to the
                         rescue, only to find you ridiculously jumping on
                         one leg while trying to get dressed. Their
                         laughter brings a QUICK END to your
                         crime-fighting career!";
                    }
                thedark:
                    "Last time you changed in the dark,
                     you wore the suit inside out!";
            }
        Wear:
            if (self has worn)
                "You are already dressed as John Covarth.";
            "The town NEEDS the power of Captain FATE, not the anonymity
             of John Covarth.";
    ],
  has   clothing proper pluralname;

Object  costume "your costume"
  with  name 'captain' 'captain^s' 'fate' 'fate^s' 'costume' 'suit',
        description
            "STATE OF THE ART manufacture, from chemically reinforced 100%
             COTTON-lastic(tm).",
        before [;
            Wear:
                if (clothes has worn)
                    "First you'd have to take off your commonplace
                     unassuming John Covarth INCOGNITO street clothes.";
            Disrobe,Change:
                if (clothes has worn)
                    "But you're not yet wearing it!";
                else
                    "You need to wear your costume to FIGHT crime!";
            Drop:
                "Your UNIQUE Captain FATE multi-coloured costume? The most
```

```
                 coveted clothing ITEM in the whole city? Certainly NOT!";
            ],
  has   clothing proper;

!===============================================================================
! Entry point routines

[ Initialise;
    #Ifdef DEBUG; pname_verify(); #Endif;       ! suggested by pname.h
    location = street;
    move costume to player;
    move clothes to player; give clothes worn;
    lookmode = 2;
    "^^Impersonating mild mannered John Covarth, assistant help boy at an
     insignificant drugstore, you suddenly STOP when your acute hearing
     deciphers a stray radio call from the POLICE. There's some MADMAN
     attacking the population in Granary Park! You must change into your
     Captain FATE costume fast...!^^";
];

[ DeathMessage;
    if (deadflag == 3) print "Your secret identity has been revealed";
    if (deadflag == 4) print "You have been SHAMEFULLY defeated";
    if (deadflag == 5) print "You fly away to SAVE the DAY";
];

[ InScope person item;
    if (person == player && location == thedark && real_location == toilet) {
        PlaceInScope(light_switch);
        PlaceInScope(toilet_door);
    }
    if (person == player && location == thedark)
        objectloop (item in parent(player))
            if (item has moved) PlaceInScope(item);
    return false;
];

!===============================================================================
! Standard and extended grammar

Include "Grammar";

[ ChangeSub;
    if (noun has pluralname) print "They're";
    else                     print "That's";
    " not something you must change to save the day.";
];

Verb 'change'
    * noun                  -> Change;

Extend 'ask'
    * creature 'for' topic  -> AskFor;

!===============================================================================
```

Compile-as-you-go

"Captain Fate" suffers from the same difficulty as "William Tell": if you type the code sequentially as you read through the guide, the game won't compile until you reach the end of Chapter 13. To compile and test as you go, add these stubs to the end of the game file when you reach the end of Chapter 10:

```
! ============================================================
! TEMPORARY DEFINITIONS NEEDED TO COMPILE AT THE END OF CHAPTER 10

Room    cafe;
Object  clothes;
```

Replace those stubs with these at the end of Chapter 11:

```
! ============================================================
! TEMPORARY DEFINITIONS NEEDED TO COMPILE AT THE END OF CHAPTER 11

Room    toilet;
Object  clothes;
Object  costume;
```

and with these at the end of Chapter 12:

```
! ============================================================
! TEMPORARY DEFINITIONS NEEDED TO COMPILE AT THE END OF CHAPTER 12

Room    toilet;
Object  clothes;
Object  costume;
Object  coin;
Object  coffee;
Object  food;
Object  menu;
```

At the end of Chapter 13 the game is complete, so you can delete the temporary stubs.

Appendix E • Inform language

efer to this appendix for a succinct but essentially complete summary of the Inform programming language; it covers everything that we've met in this guide, plus various constructs which didn't occur naturally, and others of an advanced or obscure nature.

Literals

In the specialised language of computing, the basic unit of data storage is an eight-bit **byte**, able to store a value 0..255. That's too small to be useful for holding anything other than a single character, so most computers also work with a group of two, four or eight bytes known as a **word** (not to be confused with Inform's dictionary word). In the Z-Machine a storage word comprises two bytes, and you can specify in various ways the literal values to be stored.

- Decimal: `-32768` to `32767`
  Hexadecimal: `$0` to `$FFFF`
  Binary: `$$0` to `$$1111111111111111`
- Action: `##Look`
- Character: `'a'`
- Dictionary word: `'aardvark'` (up to nine characters significant); use circumflex "^" to denote apostrophe.
  Plural word: `'aardvarks//p'`
  Single-character word: `"a"` (`name` property only) or `'a//'`
- String: `"aardvark's adventure"` (maximum around 4000 characters); can include special values including:

`^`	newline
`~`	double quotes " " "
`@@64`	at sign "@"
`@@92`	backslash "\"
`@@94`	circumflex "^"
`@@126`	tilde "~"
``@`a``	`a` with a grave accent "à", et al
`@LL`	pound sign "£", et al
`@00` ... `@31`	low string 0..31

Names

The identifier of an Inform *const_id*, *var_id*, *array*, *class_id*, *obj_id*, *property*, *attribute*, *routine_id* or *label*. Up to 32 characters: alphabetic (case not significant), numeric and underscore, with the first character not a digit.

Constants

Named word values, unchanging at run-time, which are by default initialised to zero:

```
Constant const_id;
Constant const_id = expr;
```

Standard constants are `true` (1), `false` (0) and `nothing` (0), also `NULL` (–1).

To define a constant (unless it already exists):

```
Default const_id expr;
```

Variables and arrays

Named word/byte values which can change at run-time and are by default initialised to zero.

A **global** variable is a single word:

```
Global var_id;
Global var_id = expr;
```

A **word array** is a set of global words accessed using *array*`-->0`,*array*`-->1`,... *array*`-->(`*N*`-1)`:

```
Array array --> N;
Array array --> expr1 expr2... exprN;
Array array --> "string";
```

A **table array** is a set of global words accessed using *array*`-->1`, *array*`-->2`,... *array*`-->`*N*, with *array*`-->0` initialised to *N*:

```
Array array table N;
Array array table expr1 expr2...
    exprN;
Array array table "string";
```

A **byte array** is a set of global bytes accessed using *array*`->0`, *array*`->1`,... *array*`->(`*N*`-1)`:

```
Array array -> N;
Array array -> expr1 expr2... exprN;
Array array -> "string";
```

A **string array** is a set of global bytes accessed using *array*`->1`, *array*`->2`,... *array*`->`*N*, with *array*`->0` initialised to *N*:

```
Array array string N;
Array array string expr1 expr2...
    exprN;
Array array string "string";
```

In all these cases, the characters of the initialising *string* are unpacked to the individual word/byte elements of the array.

See also Objects (for **property** variables) and Routines (for **local** variables).

Expressions and operators

Use parentheses `(...)` to control the order of evaluation.

Arithmetic/logical expressions support these operators:

`p + q`	addition
`p - q`	subtraction
`p * q`	multiplication
`p / q`	integer division
`p % q`	remainder
`p++`	increments *p*, returns orig value
`++p`	increments *p*, returns new value
`p--`	decrements *p*, returns orig value
`--p`	decrements *p*, returns new value
`p & q`	bitwise AND
`p | q`	bitwise OR
`~p`	bitwise NOT (inversion)

Conditional expressions return `true` or `false`; *q* may be a list of choices `q1 or q2 or... qN`:

`p == q`	*p* is equal to *q*
`p ~= q`	*p* isn't equal to *q*
`p > q`	*p* is greater than *q*
`p < q`	*p* is less than *q*
`p >= q`	*p* is greater than or equal to *q*
`p <= q`	*p* is less than or equal to *q*
`p ofclass q`	object *p* is of class *q*
`p in q`	object *p* is a child of object *q*
`p notin q`	object *p* isn't a child of object *q*
`p provides q`	object *p* provides property *q*
`p has q`	object *p* has attribute *q*
`p hasnt q`	object *p* hasn't attribute *q*

Boolean expressions return `true` or `false`; if *p* has determined the result, *q* is not evaluated:

`p && q`	both *p* and *q* are true (non-zero)
`p || q`	either *p* or *q* is true (non-zero)
`~~p`	*p* is false (zero)

To return –1, 0 or 1 based on unsigned comparison:

```
UnsignedCompare(p,q)
```

To return `true` if object *q* is a child or grandchild or... of *p*:

```
IndirectlyContains(p,q)
```

To return the closest common parent of two objects (or `nothing`):

```
CommonAncestor(p,q)
```

To return a random number 1..*N*, or one from a list of constant values:

```
random(N)
random(value,value, ... value)
```

Classes and objects

To declare a `class_id` – a template for a family of objects – where the optional `(N)` limits instances created at run-time:

```
Class  class_id(N)
  class class_id class_id... class_id
  with  prop_def,
        ...
        prop_def,
  has   attr_def attr_def... attr_def;
```

To declare an `obj_id`, "`Object`" can instead be a `class_id`, the remaining four header items are all optional, and `arrows` (`->`, `-> ->`, ...) and `parent_obj_id` are incompatible:

```
Object  arrows obj_id "ext_name"
    parent_obj_id
  class class_id class_id... class_id
  with  prop_def,
        ...
        prop_def,
  has   attr_def attr_def... attr_def;
```

The `class`, `with` and `has` (and also the rarely-used `private`) segments are all optional, and can appear in any order.

To determine an object's class as one of `Class`, `Object`, `Routine`, `String` (or `nothing`):

```
metaclass(obj_id)
```

has segment: Each `attr_def` is either of:

```
attribute
~attribute
```

To change attributes at run-time:

```
give obj_id attr_def... attr_def;
```

with/private segments: Each `prop_def` declares a variable (or word array) and can take any of these forms (where a `value` is an expression, a string or an embedded routine):

```
property
property value
property value value... value
```

A property variable is addressed by `obj_id.property` (or within the object's declaration as `self.property`).

Multiple `values` create a property array; in this case `obj_id.#property` is the number of **bytes** occupied by the array, the entries can be accessed using `obj_id.&property-->0`, `obj_id.&property-->1`, ... , and `obj_id.property` refers to the value of the first entry.

A property variable inherited from an object's class is addressed by `obj_id.class_id::property`; this gives the original value prior to any changes within the object.

Manipulating the object tree

To change object relationships at run-time:

```
move obj_id to parent_obj_id;
remove obj_id;
```

To return the parent of an object (or `nothing`):

```
parent(obj_id)
```

To return the first child of an object (or `nothing`):

```
child(obj_id)
```

To return the adjacent child of an object's parent (or `nothing`):

```
sibling(obj_id)
```

To return the number of child objects directly below an object:

```
children(obj_id)
```

Message passing

To a class:

```
class_id.remaining()
class_id.create()
class_id.destroy(obj_id)
class_id.recreate(obj_id)
class_id.copy(to_obj_id,from_obj_id)
```

To an object:

```
obj_id.property(a1,a2, ... a7)
```

To a routine:

```
routine_id.call(a1,a2, ... a7)
```

To a string:

```
string.print()
string.print_to_array(array)
```

Uncommon and deprecated statements

To jump to a labelled statement:

```
jump label;
...
.label; statement;
```

To terminate the program:

```
quit;
```

To save and restore the program state:

```
save label;
...
restore label;
```

To output the Inform compiler version number:

```
inversion;
```

To accept data from the current input stream:

```
read text_array parse_array
    routine_id;
```

To assign to one of 32 'low string' variables:

```
string N "string";
```

```
Lowstring string_var  "string";
string N string_var;
```

Statements

Each `statement` is terminated by a semicolon ";".

A `statement_block` is a single `statement` or a series of `statements` enclosed in braces `{...}`.

An exclamation "`!`" starts a comment – the rest of the line is ignored.

A common statement is the assignment:

```
var_id = expr;
```

There are two forms of multiple assignment:

```
var_id = var_id = ... = expr;
var_id = expr, var_id = expr, ... ;
```

Routines

A routine can have up to 15 **local variables**: word values which are private to the routine and which by default are set to zero on each call. Recursion is permitted.

A **standalone** routine:

- has a name, by which it is called using `routine_id()`; can also be called indirectly using `indirect(routine_id,a1,a2, ... a7)`
- can take arguments, using `routine_id(a1,a2, ... a7)`, whose values initialise the equivalent local variables
- returns `true` at the final "`]`"

```
[ routine_id
    local_var local_var... local_var;
    statement;
    statement;
    ...
    statement;
];
```

A routine **embedded** as the value of an object property:

- has no name, and is called when the property is invoked; can also be called explicitly using `obj_id.property()`
- accepts arguments only when called explicitly
- returns `false` at the final "`]`"

```
property [
    local_var local_var... local_var;
    statement;
    statement;
    ...
    statement;
]
```

Routines return a single value, when execution reaches the final "`]`" or an explicit return statement:

```
return expr;
return;
rtrue;
rfalse;
```

Flow control

To execute statements if *expr* is `true`; optionally, to execute other statements if *expr* is `false`:

```
if (expr)
    statement_block
if (expr)
    statement_block
else
    statement_block
```

To execute statements depending on the value of *expr*:

```
switch (expr) {
    value: statement;... statement;
    value: statement;... statement;
    ...
    default: statement;... statement;
}
```

where each *value* can be given as:

```
constant
lo_constant to hi_constant
constant,constant,... constant
```

And, if you really must:

```
jump label;
...
.label; statement;
```

Loop control

To execute statements while *expr* is `true`:

```
while (expr)
    statement_block
```

To execute statements until *expr* is `true`:

```
do
    statement_block
    until (expr)
```

To execute statements while a variable changes:

```
for (set_var : loop_while_expr :
    update_var)
        statement_block
```

To execute statements for all defined objects:

```
objectloop (var_id)
    statement_block
```

To execute statements for all objects selected by *expr*:

```
objectloop (expr_starting_with_var)
    statement_block
```

To jump out of the current innermost loop or switch:

```
break;
```

To immediately start the next iteration of the current loop:

```
continue;
```

Displaying information

To output a list of values:

```
print value,value,... value;
```

To output a list of values followed by a newline, then return `true` from the current routine:

```
print_ret value,value,... value;
```

If the first (or only) *value* is a string, "`print_ret`" can be omitted:

```
"string",value, ... value;
```

Each *value* can be an expression, a string or a rule.

An **expression** is output as a signed decimal value.

A **string** in quotes "..." is output as text.

A **rule** is one of:

`(number)` *expr*	the *expr* in words
`(char)` *expr*	the *expr* as a single character
`(string)` *addr*	the string at the *addr*
`(address)` *addr*	the dictionary word at the *addr*
`(name)` *obj_id*	the external (short) name of the *obj_id*
`(a)` *obj_id*	the short name preceded by "`a/an`", by "`some`", or by nothing for proper nouns
`(the)` *obj_id*	the short name preceded by "`the`"
`(The)` *obj_id*	the short name preceded by "`The`"
`(`*routine_id*`)` *value*	the output when calling *routine_id*`(`*value*`)`

To output a newline character:

```
new_line;
```

To output multiple spaces:

```
spaces expr;
```

To output text in a display box:

```
box "string" "string"... "string";
```

To change from regular to fixed-pitch font:

```
font off;
...
font on;
```

To change the font attributes:

```
style bold;        ! use any of these
style underline;   !
style reverse;     !
...
style roman;
```

Verbs and actions

To specify a new verb:

```
Verb 'verb' 'verb'... 'verb'
   * token  token... token -> action
   * token  token... token -> action
   ...
   * token  token... token -> action;
```

where instead "Verb" can be "Verb meta", "*action*" can be "*action* reverse"; *tokens* are optional and each is one of:

'*word*'	that literal word
'*w1*'/'*w2*'/...	any one of those literal words
attribute	an object with that attribute
creature	an object with animate attribute
held	an object held by the player
noun	an object in scope
noun=*routine_id*	an object for which *routine_id* returns true
scope=*routine_id*	an object in this re-definition of scope
multiheld	one or more objects held by the player
multi	one or more objects in scope
multiexcept	as multi, omitting the specified object
multiinside	as multi, omitting those in specified object
topic	any text
number	any number
routine_id	a general parsing routine

To add synonyms to an existing verb:

```
Verb 'verb' 'verb'... =
     'existing_verb';
```

To modify an existing verb:

```
Extend 'existing_verb' last
   * token  token... token -> action
   * token  token... token -> action
   ...
   * token  token... token -> action;
```

where instead "Extend" can be "Extend only" and "last" can be omitted, or changed to "first" or "replace".

To explicitly trigger a defined action (both *noun* and *second* are optional, depending on the *action*):

```
<action noun second>;
```

To explicitly trigger a defined action, then return true from the current routine:

```
<<action noun second>>;
```

Other useful directives

To include a directive within a routine definition [...], insert a hash "#" as its first character.

To conditionally compile:

```
Ifdef name;     ! use any one of these
Ifndef name;    !
Iftrue expr;    !
Iffalse expr;   !
    ...
Ifnot;
    ...
Endif;
```

To display a compile-time message:

```
Message "string";
```

To include the contents of a file, searching the Library path:

```
Include "source_file";
```

To include the contents of a file in the same location as the current file:

```
Include ">source_file";
```

To specify that a library routine is to be replaced:

```
Replace routine_id;
```

To set the game's release number (default is 1), serial number (default is today's *yymmdd*) and status line format (default is score):

```
Release expr;
Serial "yymmdd";
Statusline score;
Statusline time;
```

To declare a new attribute common to all objects:

```
Attribute attribute;
```

To declare a new property common to all objects:

```
Property property;
Property property expr;
```

Uncommon and deprecated directives

You're unlikely to need these; look them up in the *Designer's Manual* if necessary.

```
Abbreviate "string"... "string";
End;
Import var_id var_id ... var_id;
Link "compiled_file";
Stub routine_id N;
Switches list_of_compiler_switches;
System_file;
```

Appendix F • Inform library

ibrary files define Inform's model world, turning a conventional programming language into a text adventure development system. Here are the library constants, variables and routines, the standard object properties and attributes, the verb grammars and actions.

Library objects

`compass`
: A `container` object holding the twelve direction objects `d_obj` `e_obj` `in_obj` `n_obj` `ne_obj` `nw_obj` `out_obj` `s_obj` `se_obj` `sw_obj` `u_obj` `w_obj`.

`LibraryMessages`
: If defined (between Includes of `Parser` and `VerbLib`), changes standard library messages:

```
Object  LibraryMessages
  with  before [;
        action: "string";
        action: "string";
        action: switch (lm_n) {
                value: "string";
                value: "string",
                    (a) lm_o,".";
                ...
                }
            ...
        ];
```

`selfobj`
: The default player object. Avoid: use instead the `player` variable, which usually refers to `selfobj`.

`thedark`
: A pseudo-room which becomes the `location` when there is no light (although the player object is not moved there).

Library constants

In addition to the standard constants `true` (1), `false` (0) and `nothing` (0), the Library defines `NULL` (–1) for an *action*, *property* or *pronoun* whose current value is undefined.

User-defined constants

Some constants control features rather than represent values.

`AMUSING_PROVIDED`
: Activates the `Amusing` entry point.

`DEATH_MENTION_UNDO`
: Offers "UNDO the last move" at game end.

`DEBUG`
: Activates the debug commands.

`Headline = "string"`
: **Mandatory**: game style, copyright info, etc.

`MANUAL_PRONOUNS`
: Pronouns reflect only objects mentioned by the player.

`MAX_CARRIED = expr`
: Limit on direct possessions that the player can carry (default 100).

`MAX_SCORE = expr`
: Maximum game score (default 0).

`MAX_TIMERS = expr`
: Limit on active timers/daemons (default 32).

`NO_PLACES`
: "`OBJECTS`" and "`PLACES`" verbs are barred.

`NUMBER_TASKS = expr`
: Number of `scored` tasks (default 1).

`OBJECT_SCORE = expr`
: For taking a `scored` object for the first time (default 4).

`ROOM_SCORE = expr`
: For visiting a `scored` room for the first time (default 5).

`SACK_OBJECT = obj_id`
: A `container` object where the game places held objects.

`Story = "string"`
: **Mandatory**: the name of the story.

`TASKS_PROVIDED`
: Activates the task scoring system.

`USE_MODULES`
: Activates linking with pre-compiled library modules.

`WITHOUT_DIRECTIONS`
: De-activates standard compass directions (bar "`IN`" and "`OUT`"). Place alternative directions in the `compass`.

Library variables

`action`
The current *action*.

`actor`
The target of an instruction: the player, or an NPC.

`deadflag`
Normally 0: 1 indicates a regular death, 2 indicates that the player has won, 3 or more denotes a user-defined end.

`inventory_stage`
Used by `invent` and `list_together` properties.

`keep_silent`
Normally `false`; `true` makes most group 2 actions silent.

`location`
The player's current room; unless that's dark, when it contains `thedark`, `real_location` contains the room.

`notify_mode`
Normally `true`: `false` remains silent when score changes.

`noun`
The primary focus object for the current action.

`player`
The object acting on behalf of the human player.

`real_location`
The player's current room when in the dark.

`score`
The current score.

`second`
The secondary focus object for the current action.

`self`
The object which received a message. (Note: a run-time variable, not a compile-time constant.)

`sender`
The object which sent a message (or `nothing`).

`task_scores`
A byte array holding scores for the task scoring system.

`the_time`
The game's clock, in minutes 0..1439 since midnight.

`turns`
The game's turn counter.

`wn`
The input stream word number, counting from 1.

Library routines

`Achieved(expr)`
A scored task has been achieved.

`AfterRoutines()`
In a group 2 action, controls output of "after" messages.

`AllowPushDir()`
An object can be pushed from one location to another.

`Banner()`
Prints the game banner.

`ChangePlayer(obj_id,flag)`
Player assumes the persona of the *obj_id*. If the optional *flag* is `true`, room descriptions include "(as *object*)".

`CommonAncestor(obj_id1,obj_id2)`
Returns the nearest object which has a parental relationship to both *obj_ids*, or `nothing`.

`DictionaryLookup(byte_array,length)`
Returns address of word in dictionary, or 0 if not found.

`DrawStatusLine()`
Refreshes the status line; happens anyway at end of each turn.

`GetGNAOfObject(obj_id)`
Returns gender-number-animation 0..11 of the *obj_id*.

`HasLightSource(obj_id)`
Returns `true` if the *obj_id* has light.

`IndirectlyContains(parnt_obj_id,obj_id)`
Returns `true` if *obj_id* is currently a child or grand-child or great-grand-child... of the *parent_object*.

`IsSeeThrough(obj_id)`
Returns `true` if light can pass through the *obj_id*.

`Locale(obj_id,"string1","string2")`
Describes the contents of *obj_id*, and returns their number. After objects with own paragraphs, the rest are listed preceded by *string1* or *string2*.

`LoopOverScope(routine_id,actor)`
Calls *routine_id(obj_id)* for each *obj_id* in scope. If the optional *actor* is supplied, that defines the scope.

`MoveFloatingObjects()`
Adjusts positions of game's `found_in` objects.

`NextWord()`
Returns the next dictionary word in the input stream, incrementing `wn` by one. Returns `false` if the word is not in the dictionary, or if the input stream is exhausted.

`NextWordStopped()`
Returns the next dictionary word in the input stream, incrementing `wn` by one. Returns `false` if the word is not in the dictionary, –1 if the input stream is exhausted.

`NounDomain(obj_id1,obj_id2,type)`
Performs object parsing; see also `ParseToken()`.

`ObjectIsUntouchable(obj_id,flag)`
Tests whether there is a barrier – a container object which is not open – between player and `obj_id`. Unless the optional `flag` is `true`, outputs "You can't because ... is in the way". Returns `true` is a barrier is found, otherwise `false`.

`OffersLight(obj_id)`
Returns `true` if the `obj_id` offers light.

`ParseToken(type,value)`
Performs general parsing; see also `NounDomain()`.

`PlaceInScope(obj_id)`
Used in an `add_to_scope` property or `scope=` token to put the `obj_id` into scope for the parser.

`PlayerTo(obj_id,flag)`
Moves the player to `obj_id`. Prints its description unless optional `flag` is 1 (no description) or 2 (as if walked in).

`PrintOrRun(obj_id,property,flag)`
If `obj_id.property` is a string, output it (followed by a newline unless optional `flag` is `true`), and return `true`. If it's a routine, run it and return what the routine returns.

`PronounNotice(obj_id)`
Associates an appropriate pronoun with the `obj_id`.

`PronounValue('pronoun')`
Returns the object to which `'it'` (or `'him'`, `'her'`, `'them'`) currently refers, or `nothing`.

`ScopeWithin(obj_id)`
Used in an `add_to_scope` property or `scope=` token to put the contents of the `obj_id` in scope for the parser.

`SetPronoun('pronoun',obj_id)`
Defines the `obj_id` to which a given `pronoun` refers.

`SetTime(expr1,expr2)`
Sets `the_time` to `expr1` (in mins 0..1439 since midnight), running at `expr2` –
+ve: `expr2` minutes pass each turn;
–ve: `-expr2` turns take one minute;
zero: time stands still.

`StartDaemon(obj_id)`
Starts the `obj_id`'s daemon.

`StartTimer(obj_id,expr)`
Starts the `obj_id`'s timer, initialising its `time_left` to `expr`. The object's `time_out` property will be called after that number of turns have elapsed.

`StopDaemon(obj_id)`
Stops the `obj_id`'s daemon.

`StopTimer(obj_id)`
Stops the `obj_id`'s timer.

`TestScope(obj_id,actor)`
Returns `true` if the `obj_id` is in scope, otherwise `false`. If the optional `actor` is supplied, that defines the scope.

`TryNumber(expr)`
Parses word `expr` in the input stream as a number, recognising decimals, also English words one..twenty. Returns the number 1..10000, or -1000 if the parse fails.

`UnsignedCompare(expr1,expr2)`
Returns –1 if `expr1` is less than `expr2`, 0 if `expr1` equals `expr2`, and 1 if `expr1` is greater than `expr2`. Both expressions are unsigned, in the range 0..65535.

`WordAddress(expr)`
Returns a byte array containing the raw text of word `expr` in the input stream.

`WordInProperty(word,obj_id,property)`
Returns `true` if the dictionary `word` is listed in the `property` values for the `obj_id`.

`WordLength(expr)`
Returns the length of word `expr` in the input stream.

`WriteListFrom(obj_id,expr)`
Outputs a list of `obj_id` and its siblings, in the given style, an `expr` formed by adding any of: `ALWAYS_BIT`, `CONCEAL_BIT`, `DEFART_BIT`, `ENGLISH_BIT`, `FULLINV_BIT`, `INDENT_BIT`, `ISARE_BIT`, `NEWLINE_BIT`, `PARTINV_BIT`, `RECURSE_BIT`, `TERSE_BIT`, `WORKFLAG_BIT`.

`YesOrNo()`
Returns `true` if the player types "`YES`", `false` for "`NO`".

`ZRegion(arg)`
Returns the type of its `arg`: 3 for a string address, 2 for a routine address, 1 for an object number, or 0 otherwise.

Object properties

Where the *value* of a property can be a routine, several formats are possible (but remember: embedded "]" returns false, standalone "]" returns true):

```
property [; stmt; stmt; ... ]

property [; return routine_id(); ]

property [; routine_id(); ]

property  routine_id
```

In this appendix, "⊕" marks an additive property. Where a Class and an Object of that class both define the same property, the value specified for the Object normally overrides the value inherited from the Class. However, if the property is additive then both values apply, with the Object's value being considered first.

add_to_scope
: For an object: additional objects which follow it in and out of scope. The *value* can be a space-separated list of *obj_ids*, or a routine which invokes PlaceInScope() or ScopeWithin() to specify objects.

after ⊕
: For an object: receives every *action* and *fake_action* for which this is the *noun*. For a room: receives every *action* which occurs here.

  The *value* is a routine of structure similar to a switch statement, having cases for the appropriate *actions* (and an optional default as well); it is invoked after the action has happened, but before the player has been informed. The routine should return false to continue, telling the player what has happened, or true to stop processing the action and produce no further output.

article
: For an object: the object's indefinite article – the default is automatically "a", "an" or "some". The *value* can be a string, or a routine which outputs a string.

articles
: For a non-English object: its definite and indefinite articles. The *value* is an array of strings.

before ⊕
: For an object: receives every *action* and *fake_action* for which this is the *noun*. For a room: receives every *action* which occurs here.

  The *value* is a routine invoked before the action has happened. See after.

cant_go
: For a room: the message when the player attempts an impossible exit. The *value* can be a string, or a routine which outputs a string.

capacity
: For a container or supporter object: the number of objects which can be placed in or on it – the default is 100.
  For the player: the number which can be carried – selfobj has an initial capacity of MAX_CARRIED.

  The *value* can be a number, or a routine which returns a number.

d_to
: For a room: a possible exit. The *value* can be
  - false (the default): not an exit;
  - a string: output to explain why this is not an exit;
  - a *room*: the exit leads to this room;
  - a door object: the exit leads through this door;
  - a routine which should return false, a string, a *room*, a door object, or true to signify "not an exit" and produce no further output.

daemon
: The *value* is a routine which can be activated by StartDaemon(*obj_id*) and which then runs once each turn until deactivated by StopDaemon(*obj_id*).

describe ⊕
: For an object: called before the object's description is output. For a room: called before the room's (long) description is output.

  The *value* is a routine which should return false to continue, outputting the usual description, or true to stop processing and produce no further output.

description
: For an object: its description (output by Examine).
  For a room: its long description (output by Look).

  The *value* can be a string, or a routine which outputs a string.

door_dir
: For a compass object (d_obj, e_obj, ...): the direction in which an attempt to move to this object actually leads.
  For a door object: the direction in which this door leads.

  The *value* can be a directional property (d_to, e_to, ...), or a routine which returns such a property.

`door_to`
For a `door` object: where it leads. The *`value`* can be

- `false` (the default): leads nowhere;
- a string: output to explain why door leads nowhere;
- a *`room`*: the door leads to this room;
- a routine which should return `false`, a string, a *`room`*, or `true` to signify "leads nowhere" without producing any output.

`e_to`
See `d_to`.

`each_turn ⊕`
Invoked at the end of each turn (after all appropriate daemons and timers) whenever the object is in scope. The *`value`* can be a string, or a routine.

`found_in`
For an object: the rooms where this object can be found, unless it has the `absent` attribute. The *`value`* can be

- a space-separated list of *`rooms`* (where this object can be found) or *`obj_ids`* (whose locations are tracked by this object);
- a routine which should return `true` if this object can be found in the current location, otherwise `false`.

`grammar`
For an `animate` or `talkable` object: the *`value`* is a routine called when the parser knows that this object is being addressed, but has yet to test the grammar. The routine should return `false` to continue, `true` to indicate that the routine has parsed the entire command, or a dictionary word ('*`word`*' or –'*`word`*').

`in_to`
See `d_to`.

`initial`
For an object: its description before being picked up.
For a room: its description when the player enters the room.

The *`value`* can be a string, or a routine which outputs a string.

`inside_description`
For an `enterable` object: its description, output as part of the room description when the player is inside the object.

The *`value`* can be a string, or a routine which outputs a string.

`invent`
For an object: the *`value`* is a routine for outputting the object's inventory listing, which is called twice. On the first call nothing has been output; `inventory_stage` has the value 1, and the routine should return `false` to continue, or `true` to stop processing and produce no further output. On the second call the object's indefinite article and short name have been output, but not any subsidiary information; `inventory_stage` has the value 2, and the routine should return `false` to continue, or `true` to stop processing and produce no further output.

`life ⊕`
For an `animate` object: receives person-to-person *`actions`* (`Answer`, `Ask`, `Attack`, `Give`, `Kiss`, `Order`, `Show`, `Tell`, `ThrowAt and WakeOther`) for which this is the *`noun`*. The *`value`* is a routine of structure similar to a `switch` statement, having cases for the appropriate *`actions`* (and an optional default as well). The routine should return `false` to continue, telling the player what has happened, or `true` to stop processing the action and produce no further output.

`list_together`
For an object: groups related objects when outputting an inventory or room contents list. The *`value`* can be

- a *`number`*: all objects having this value are grouped;
- a *`string`*: all objects having this value are grouped as a count of the string;
- a routine which is called twice. On the first call nothing has been output; `inventory_stage` has the value 1, and the routine should return `false` to continue, or `true` to stop processing and produce no further output. On the second call the list has been output; `inventory_stage` has the value 2, and there is no test on the return value.

`n_to`
See `d_to`.

`name ⊕`
Defines a space-separated list of words which are added to the Inform dictionary. Each word can be supplied in apostrophes `'...'` or quotes `"..."`; in all other cases only words in apostrophes update the dictionary.

For an object: identifies this object.
For a room: outputs "does not need to be referred to".

`ne_to`
See `d_to`.

`number`
For an object or room: the *value* is a general-purpose variable freely available for use by the program. A `player` object must provide (but not use) this variable.

`nw_to`
See `d_to`.

`orders`
For an `animate` or `talkable` object: the *value* is a routine called to carry out the player's orders. The routine should return `false` to continue, or `true` to stop processing the action and produce no further output.

`out_to`
See `d_to`.

`parse_name`
For an object: the *value* is a routine called to parse an object's name. The routine should return zero if the text makes no sense, –1 to cause the parser to resume, or the positive number of words matched.

`plural`
For an object: its plural form, when in the presence of others like it. The *value* can be a string, or a routine which outputs a string.

`react_after`
For an object: detects nearby actions – those which take place when this object is in scope. The *value* is a routine invoked after the action has happened, but before the player has been informed. See `after`.

`react_before`
For an object: detects nearby actions – those which take place when this object is in scope. The *value* is a routine invoked before the action has happened. See `after`.

`s_to`
`se_to`
See `d_to`.

`short_name`
For an object: an alternative or extended short name. The *value* can be a string, or a routine which outputs a string. The routine should return `false` to continue by outputting the object's *actual* short name (from the head of the object definition), or `true` to stop processing the action and produce no further output.

`short_name_indef`
For a non-English object: the short name when preceded by an indefinite object. The *value* can be a string, or a routine which outputs a string.

`sw_to`
See `d_to`.

`time_left`
For a timer object: the *value* is a variable to hold the number of turns left until this object's timer – activated and initialised by `StartTimer(`*obj_id*`)` – counts down to zero and invokes the object's `time_out` property.

`time_out`
For a timer object: the *value* is a routine which is run when the object's `time_left` value – initialised by `StartTimer(`*obj_id*`)`, and not in the meantime cancelled by `StopTimer(`*obj_id*`)` – counts down to zero.

`u_to`
`w_to`
See `d_to`.

`when_closed`
`when_open`
For a `container` or `door` object: used when including this object in a room's long description. The *value* can be a string, or a routine which outputs a string.

`when_off`
`when_on`
For a `switchable` object: used when including this object in a room's long description. The *value* can be a string, or a routine which outputs a string.

`with_key`
For a `lockable` object: the *obj_id* (generally some kind of key) needed to lock and unlock the object, or `nothing` if no key fits.

Object attributes

`absent`
For a floating object (one with a `found_in` property, which can appear in many rooms): is no longer there.

`animate`
For an object: is a living creature.

`clothing`
For an object: can be worn.

`concealed`
For an object: is present but hidden from view.

`container`
For an object: other objects can be put in (but not on) it.

`door`
For an object: is a door or bridge between rooms.

`edible`
For an object: can be eaten.

`enterable`
For an object: can be entered.

`female`
For an `animate` object: is female.

`general`
For an object or room: a general-purpose flag.

`light`
For an object or room: is giving off light.

`lockable`
For an object: can be locked; see the `with_key` property.

`locked`
For an object: can't be opened.

`male`
For an `animate` object: is male.

`moved`
For an object: is being, or has been, taken by the player.

`neuter`
For an `animate` object: is neither male nor female.

`on`
For a `switchable` object: is switched on.

`open`
For a `container` or `door` object: is open.

`openable`
For a `container` or `door` object: can be opened.

`pluralname`
For an object: is plural.

`proper`
For an object: the short name is a proper noun, therefore not to be preceded by "The" or "the".

`scenery`
For an object: can't be taken; is not listed in a room description.

`scored`
For an object: awards `OBJECT_SCORE` points when taken for the first time. For a room: awards `ROOM_SCORE` points when visited for the first time.

`static`
For an object: can't be taken.

`supporter`
For an object: other objects can be put on (but not in) it.

`switchable`
For an object: can be switched off or on.

`talkable`
For an object: can be addressed in "object, do this" style.

`transparent`
For a `container` object: objects inside it are visible.

`visited`
For a room: is being, or has been, visited by the player.

`workflag`
Temporary internal flag, also available to the program.

`worn`
For a `clothing` object: is being worn.

Optional entry points

These routines, if you supply them, are called when shown.

`AfterLife()`
Player has died; `deadflag=0` resurrects.

`AfterPrompt()`
The ">" prompt has been output.

`Amusing()`
Player has won; `AMUSING_PROVIDED` is defined.

`BeforeParsing()`
The parser has input some text, set up the buffer and parse tables, and initialised `wn` to 1.

`ChooseObjects(`*`object`*`,`*`flag`*`)`
Parser has found "`ALL`" or an ambiguous noun phrase and decided that *`object`* should be excluded (`flag` is 0), or included (`flag` is 1). The routine should return 0 to let this stand, 1 to force inclusion, or 2 to force exclusion. If `flag` is 2, parser is undecided; routine should return appropriate score 0..9.

`DarkToDark()`
The player has gone from one dark room to another.

`DeathMessage()`
The player has died; `deadflag` is 3 or more.

`GamePostRoutine()`
Called after all *`actions`*.

`GamePreRoutine()`
Called before all *`actions`*.

`Initialise()`
Mandatory; note British spelling: called at start. Must set `location`; can return 2 to suppress game banner.

`InScope()`
Called during parsing.

`LookRoutine()`
Called at the end of every `Look` description.

`NewRoom()`
Called when room changes, before description is output.

`ParseNoun(`*`object`*`)`
Called to parse the *`object`*'s name.

`ParseNumber(`*`byte_array`*`,`*`length`*`)`
Called to parse a number.

`ParserError(`*`number`*`)`
Called to handle an error.

`PrintRank()`
Completes the output of the score.

`PrintTaskName(`*`number`*`)`
Prints the name of the task.

`PrintVerb(`*`addr`*`)`
Called when an unusual verb is printed.

`TimePasses()`
Called after every turn.

`UnknownVerb()`
Called when an unusual verb is encountered.

Group 1 actions

Group 1 actions support the 'meta' verbs. These are the standard actions and their triggering verbs.

`FullScore`	"`FULLSCORE`", "`FULL [SCORE]`"
`LMode1`	"`BRIEF`", "`NORMAL`"
`LMode2`	"`LONG`", "`VERBOSE`"
`LMode3`	"`SHORT`", "`SUPERBRIEF`"
`NotifyOff`	"`NOTIFY OFF`"
`NotifyOn`	"`NOTIFY [ON]`"
`Objects`	"`OBJECTS`"
`Places`	"`PLACES`"
`Pronouns`	"`[PRO]NOUNS`"
`Quit`	"`DIE`", "`Q[UIT]`"
`Restart`	"`RESTART`"
`Restore`	"`RESTORE`"
`Save`	"`CLOSE`"
`Score`	"`SCORE`"
`ScriptOff`	"`[TRAN]SCRIPT OFF`", "`NOSCRIPT`", "`UNSCRIPT`"
`ScriptOn`	"`[TRAN]SCRIPT [ON]`"
`Verify`	"`VERIFY`"
`Version`	"`VERSION`"

and the debug tools.

`ActionsOff`	"`ACTIONS OFF`"
`ActionsOn`	"`ACTIONS [ON]`"
`ChangesOff`	"`CHANGES OFF`"
`ChangesOn`	"`CHANGES [ON]`"
`CommandsOff`	"`RECORDING OFF`"
`CommandsOn`	"`RECORDING [ON]`"
`CommandsRead`	"`REPLAY`"
`Gonear`	"`GONEAR`"
`Goto`	"`GOTO`"
`Predictable`	"`RANDOM`"
`RoutinesOff`	"`MESSAGES OFF`", "`ROUTINES OFF`"
`RoutinesOn`	"`MESSAGES [ON]`", "`ROUTINES [ON]`"
`Scope`	"`SCOPE`"
`Showobj`	"`SHOWOBJ`"
`Showverb`	"`SHOWVERB`"
`TimersOff`	"`DAEMONS OFF`", "`TIMERS OFF`"
`TimersOn`	"`DAEMONS [ON]`", "`TIMERS [ON]`"
`TraceLevel`	"`TRACE` *`number`*"
`TraceOff`	"`TRACE OFF`"
`TraceOn`	"`TRACE [ON]`"
`XAbstract`	"`ABSTRACT`"
`XPurloin`	"`PURLOIN`"
`XTree`	"`TREE`"

Group 2 actions

Group 2 actions usually work, given the right circumstances.

Close	"CLOSE [UP]", "COVER [UP]", "SHUT [UP]"
Disrobe	"DISROBE", "DOFF", "REMOVE", "SHED", "TAKE OFF"
Drop	"DISCARD", "DROP", "PUT DOWN", "THROW"
Eat	"EAT"
Empty	"EMPTY [OUT]"
EmptyT	"EMPTY IN|INTO|ON|ONTO|TO"
Enter	"CROSS", "ENTER", "GET IN|INTO|ON|ONTO", "GO IN|INSIDE|INTO|THROUGH", "LEAVE IN|INSIDE|INTO|THROUGH", "LIE IN|INSIDE|ON", "LIE ON TOP OF", "RUN IN|INSIDE|INTO|THROUGH", "SIT IN|INSIDE|ON", "SIT ON TOP OF", "STAND ON", "WALK IN|INSIDE|INTO|THROUGH"
Examine	"CHECK," "DESCRIBE", "EXAMINE", "L[OOK] AT", "READ", "WATCH", "X"
Exit	"EXIT", "GET OFF|OUT|UP", "LEAVE", "OUT[SIDE]", "STAND [UP]"
GetOff	"GET OFF"
Give	"FEED [TO]", "GIVE [TO]", "OFFER [TO]", "PAY [TO]"
Go	"GO", "LEAVE", "RUN", "WALK"
GoIn	"CROSS", "ENTER", "IN[SIDE]"
Insert	"DISCARD IN|INTO", "DROP DOWN|IN|INTO", "INSERT IN|INTO", "PUT IN|INSIDE|INTO", "THROW DOWN|IN|INTO"
Inv	"I[NV]", "INVENTORY", "TAKE INVENTORY"
InvTall	"I[NV] TALL", "INVENTORY TALL"
InvWide	"I[NV] WIDE", "INVENTORY WIDE"
Lock	"LOCK WITH"
Look	"L[OOK]"
Open	"OPEN", "UNCOVER", "UNDO", "UNWRAP"
PutOn	"DISCARD ON|ONTO", "DROP ON|ONTO", "PUT ON|ONTO", "THROW ON|ONTO"
Remove	"GET FROM", "REMOVE FROM", "TAKE FROM|OFF"
Search	"L[OOK] IN|INSIDE|INTO|THROUGH", "SEARCH"
Show	"DISPLAY [TO]", "PRESENT [TO]", "SHOW [TO]"
SwitchOff	"CLOSE OFF", "SCREW OFF", "SWITCH OFF", "TURN OFF", "TWIST OFF"
SwitchOn	"SCREW ON", "SWITCH ON", "TURN ON", "TWIST ON"
Take	"CARRY", "GET", "HOLD", "PEEL [OFF]", "PICK UP", "REMOVE", "TAKE"
Transfer	"CLEAR TO", "MOVE TO", "PRESS TO", "PUSH TO", "SHIFT TO", "TRANSFER TO"
Unlock	"OPEN WITH", "UNDO WITH", "UNLOCK WITH"
VagueGo	"GO", "LEAVE", "RUN", "WALK"
Wear	"DON", "PUT ON", "WEAR"

Group 3 actions

Group 3 actions are by default stubs which output a message and stop at the "before" stage (so there is no "after" stage).

Answer	"ANSWER TO", "SAY TO", "SHOUT TO", "SPEAK TO"
Ask	"ASK ABOUT"
AskFor	"ASK FOR"
Attack	"ATTACK", "BREAK", "CRACK", "DESTROY", "FIGHT", "HIT", "KILL", "MURDER", "PUNCH", "SMASH", "THUMP", "TORTURE", "WRECK"
Blow	"BLOW"
Burn	"BURN [WITH]", "LIGHT [WITH]"
Buy	"BUY" "PURCHASE"
Climb	"CLIMB [OVER|UP]", "SCALE"
Consult	"CONSULT ABOUT|ON", "LOOK UP IN", "READ ABOUT IN", "READ IN"
Cut	"CHOP," "CUT", "PRUNE", "SLICE"
Dig	"DIG [WITH]"
Drink	"DRINK", "SIP", "SWALLOW"
Fill	"FILL"
Jump	"HOP", "JUMP", "SKIP"
JumpOver	"HOP OVER", "JUMP OVER", "SKIP OVER"
Kiss	"EMBRACE", "HUG", "KISS"
Listen	"HEAR", "LISTEN [TO]"
LookUnder	"LOOK UNDER"
Mild	Various mild swearwords.
No	"NO"
Pray	"PRAY"
Pull	"DRAG" "PULL"

`Push`	"`CLEAR`", "`MOVE`", "`PRESS`", "`PUSH`", "`SHIFT`"
`PushDir`	"`CLEAR`", "`MOVE`", "`PRESS`", "`PUSH`", "`SHIFT`"
`Rub`	"`CLEAN`", "`DUST`", "`POLISH`", "`RUB`", "`SCRUB`", "`SHINE`", "`SWEEP`", "`WIPE`"
`Set`	"`ADJUST`", "`SET`"
`SetTo`	"`ADJUST TO`", "`SET TO`"
`Sing`	"`SING`"
`Sleep`	"`NAP`", "`SLEEP`"
`Smell`	"`SMELL`", "`SNIFF`"
`Sorry`	"`SORRY`"
`Squeeze`	"`SQUASH`", "`SQUEEZE`"
`Strong`	Various strong swearwords.
`Swim`	"`DIVE`", "`SWIM`"
`Swing`	"`SWING [ON]`"
`Taste`	"`TASTE`"
`Tell`	"`TELL ABOUT`"
`Think`	"`THINK`"
`ThrowAt`	"`THROW AGAINST|AT|ON|ONTO`"
`Tie`	"`ATTACH [TO]`", "`FASTEN [TO]`", "`FIX [TO]`", "`TIE [TO]`"
`Touch`	"`FEEL`," "`FONDLE`", "`GROPE`", "`TOUCH`"
`Turn`	"`ROTATE`", "`SCREW`", "`TURN`", "`TWIST`", "`UNSCREW`"
`Wait`	"`WAIT`" "`Z`"
`Wake`	"`AWAKE[N]`", "`WAKE [UP]`"
`WakeOther`	"`AWAKE[N]`", "`WAKE [UP]`"
`Wave`	"`WAVE`"
`WaveHands`	"`WAVE`"
`Yes`	"`Y[ES]`"

Fake actions

Fake actions handle some special cases, or represent "real" actions from the viewpoint of the second object.

`LetGo`	Generated by `Remove`.
`ListMiscellany`	Outputs a range of inventory messages.
`Miscellany`	Outputs a range of utility messages.
`NotUnderstood`	Generated when parser fails to interpret some `orders`.
`Order`	Receives things not handled by `orders`.
`PluralFound`	Tells the parser that `parse_name()` has identified a plural object.
`Prompt`	Outputs the prompt, normally ">".
`Receive`	Generated by `Insert` and `PutOn`.
`TheSame`	Generated when parser can't distinguish between two objects.
`ThrownAt`	Generated by `ThrowAt`.

Appendix G • Glossary

During our travels, we've encountered certain terms which have particular significance in the context of the Inform text adventure development system; here are brief definitions of many of those specialised words and phrases.

action – the generated effect of the player's input, usually by the **parser** but also occasionally by the designer's code. It refers to a single task to be processed by Inform, such as DROP KETTLE, and it's stored in four numbers: one each for the action itself and the `actor` object who is to perform it (the player or an **NPC**), one for the `noun` – or direct object, if present – and a fourth for the `second` noun – if it exists, for example the "POT" in THROW KETTLE AT POT.

alpha-testing – the testing which is carried out by the game's designer, in a futile attempt to ensure that it does everything that it should and nothing that it shouldn't. See also **beta-testing**.

argument – a parameter supplied in a call to a **routine**, which is the actual value for one of the routine's defined local variables. For example, the argument is 8 in the call `MyRoutine(8)`. The definition of the routine includes the variable that will hold the argument, in this case `x`:

```
[ MyRoutine x; ... ];
```

ASCII file – see **text file**.

assignment – a statement which sets or changes the value of a **variable**. There are three in Inform: `=` (set equal to), `++` (add one to the current value), `--` (subtract one from the current value).

attributes – named flags that can be defined for an object after the keyword `has`. An attribute is either present (on) or not present (off). The designer may test from any other part of the program *if* an object *has* a certain attribute, *give* an attribute to an object or take it away as need arises. For instance, the attribute `container` states that the object is capable of having other objects placed inside it.

avatar – see **player**.

banner – information about a game which is displayed at the start of play.

beta-testing – the testing which is carried out by a small band of trusted volunteers, prior to general public release, during which the gross inadequacy of the designer's **alpha-testing** effort becomes painfully apparent.

binary file – a computer file containing binary data – 0s and 1s – which is created by a program and which only a program can understand.

bold type – used to highlight a term explained in this glossary.

child – see **object tree**.

class – a special **object** template from which other objects can inherit **properties** and/or **attributes**. The template must begin with the word `Class` and must have an internal identifier. Objects that wish to inherit from a class usually begin with the internal ID of the class in place of the

word `Object`, but may instead define a segment `class` followed by the class's internal ID. The designer may test whether an object belongs to – is a member of – a class.

code block – see **statement block**.

comment – text which starts with an exclamation mark `!` and which is ignored by the compiler when it reads the **source file**; added to improve the file's layout or for explanatory notes.

compile-time – the time when the **compiler** is at work making the **story file**. See also **run-time**.

compiler – a program that reads the source code written by the designer and turns it into a **story file**, which can then be played by a Z-Machine **interpreter**.

constant – a particular value which is defined at **compile-time**, always stays the same and cannot be changed while the game is being played. Common examples include numbers, strings and the internal IDs of objects, any of which can be either written out explicitly or set as the value of a named `Constant`.

Debug mode – a option which causes to compiler to include extra code into the story file, thus making it easier for the designer to understand what's happening while a game is being tested prior to release. See also **Strict mode**.

designer – a person who uses Inform to create a text adventure game: in other words, gentle reader, you.

dictionary – the collection of all input words "understood" by the game.

dictionary word – a word written in single quotes `'...'` within the **source file**, usually (but not exclusively) as one of the values assigned to an object's `name` property. All such words are stored in the **dictionary**, which is consulted by the **parser** when attempting to make sense of a player's command. Only the first nine characters are significant (thus `'cardiogram'` and `'cardiograph'` are treated as the same word). Use `'coins//p'` to mark "`coins`" as plural, referring to all coin objects which are present. Use `'t//'` to enter the single-character word "t" into the dictionary (`'t'` is a constant representing a character value).

directive – a line of Inform code which asks the **compiler** to do something there and then, at **compile-time**; typical examples are to Include the contents of another file, or to set aside some space within the story file where a variable value may be stored. Not to be confused with a **statement**, which asks the compiler to compose an instruction which the interpreter will obey at **run-time**; typical examples are to display some text, or to change the value held within a variable's storage space.

editor – a general-purpose program for creating and modifying **text files**.

embedded routine – a routine that is defined in the body of an object, as the value of one of its **properties**. Unlike a **standalone routine**, an embedded routine doesn't have a name of its own, and returns `false` if execution reaches the terminating marker `]`.

entry point – one of a predefined list of optional routines which, if you

provide it, will be called by the library either to produce some supplementary output or to return a value causing the library to change its default behaviour.

false – a logical state which is the opposite of **true**, represented by the value 0.

flag – a variable which can take only two possible values.

function – see **routine**.

global variable – a variable not specific to any routine or object, which can be used by any routine in the game.

inheritance – the process by which an **object** belonging to a **class** acquires the properties and attributes of said class. Inheritance happens automatically; the designer has just to create class definitions, followed by objects having those classes.

interpreter – a program that reads the **story file** of a game and enables people to play it. Interpreters must be platform-specific (that is, they will be different programs for each operating system), thus allowing the story file to be universal and platform-independent.

italic type – used for emphasis, and as a placeholder to represent a value which you should supply.

library – a group of text files, part of the Inform system, that includes the **parser**, definitions for the **model world**, language files, grammar definitions and a customised stock of default answers and behaviour for the player's actions. The library will make frequent calls to the game file to see if the designer wants to override those defaults.

library files – the actual files containing the source code of the library. There are basically three (although these three Include other files as well): `parser.h`, `verblib.h` and `grammar.h`, and they should be Included in every Inform game.

library routine – one of a set of routines included as part of the library which the designer can call to perform some commonly useful task.

local variable – a variable which is part of only one **routine**; its value remains unavailable to other routines in the game. The value of a local variable is *not* preserved between calls to the routine.

model world – the imaginary environment which the player character inhabits.

newline – the ASCII control character(s) used to mark the end of a line of text.

NPC – a non-player character; any character other than the protagonist. Could range from an opponent or love interest to a pet gerbil or a random pedestrian.

object – a group of **routines** and **variables** bundled up together in a coherent unit. Objects represent the items that make up the model world (a torch; a car; a beam of light; etc.), a fact which organises the designer's code in sensible chunks, easy to manage. Each object has two parts: the header, which comprises the internal ID, the external name and its defined parent (all fields are optional), and the body, which

comprises the property variables and attribute flags particular to that object, if any.

object tree – a hierarchy that defines objects' relationships in terms of containment. Each **object** is either contained within another object – its parent – or is *not* contained; objects such as rooms which are not within another object have the constant `nothing` (0) as a parent. An object contained within another is a child. For example, a shoe inside a box: the box is the shoe's parent and the shoe is a child of the box. Consider now this box being inside the wardrobe. The box is a child of the wardrobe, but the shoe is still a child of the box, not the wardrobe. In a normal game, the object tree will undergo many transformations as the result of the player's activities.

parent – see **object tree**.

parser – part of the **library** which is responsible for analysing the player's input and trying to make sense of it, dividing it into separate words (verb, nouns) and trying to match them against the words stored in the game's **dictionary** and the actions defined in the game's grammar. If the player's input makes sense, the parser will trigger the resulting **action**; if not, it will complain that it didn't understand.

PC – 1. a personal computer; 2. the player character (see **player**).

player – 1. the final user of the game, normally a person full of radical opinions about your capabilities as a designer; 2. a variable referring to the **object** – sometimes known as an "avatar" – which currently represents that user within the **model world**.

print rule – a customised rule to apply while in a `print` or `print_ret` statement, to control the manner in which an item of data is to be displayed. For example: `print (The) noun, " is mine."` is telling the game to use a capitalised definite article for the noun. The library defines a stock of print rules, and designers may create some of their own.

properties – variables attached to a single **object**, of which they are a part. They are defined in the body of the object after the keyword `with` and have a name and a value. The latter (which defaults to 0) can be a number, a string `"..."`, a dictionary word `'...'` or an embedded routine `[;...]`; it can also be a list of those separated by spaces. The value of an object's property can be tested and changed from any part of the game. The fact that an object provides a property may be tested.

RAIF – the `rec.arts.int-fiction` Usenet newsgroup for IF designers.

RGIF – the `rec.games.int-fiction` Usenet newsgroup for IF players.

room – an **object** which defines a geographical unit into which the map of the **model world** is divided. Rooms have no parent object (or, more precisely, their parent object is `nothing`) and they represent the places where the player character is at any given moment – the player character can't be in more than one room at a time. Note that the name "room" does not imply necessarily "indoors". A clearing, a sandy beach, the top of a tree, even

floating in outer space – these are all possible room objects.

routine – in general terms, a routine is a computer program that makes some specific calculation, following an ordered set of instructions; this is the only unit of coherent and executable code understood by Inform. More practically, a routine is a collection of **statements** which are written between markers `[...]`. When a routine is "called", possibly with arguments – specific values for its defined variables, if they exist – the interpreter executes the statements in sequence. If the interpreter encounters a `return` statement, or reaches the `]` at the end of the routine, it immediately stops executing statements in the routine and resumes execution at the statement which called that routine. Every routine returns a value, which is either supplied by the `return` statement or implied by the `]` at the end of the routine. See **embedded routine** and **standalone routine**.

run-time – the period of time when the **interpreter** is running a **story file** (that is, someone is playing the game). See also **compile-time**.

source file – a text file containing your game defined using the Inform language.

standalone routine – a routine which is not part of an object. Unlike an **embedded routine**, it must provide a name of its own, and it returns `true` when execution reaches the terminating marker `]` .

statement – a single instruction to be executed at **run-time**. See also **directive**.

statement block – a group of **statements** bundled up together between braces `{...}`, which are then treated as a single unit – as if they were only one statement. They commonly appear in loops and conditions.

story file – a binary file which is the output of the **compiler** and can be played through the use of an **interpreter** (also known as Z-code file or game file). The format of story files is standard and platform-independent.

Strict mode – an option which causes the **compiler** to include extra code into the story file, thus making it easier to detect certain design mistakes while a game is being played. This mode automatically invokes **Debug mode**.

string – a piece of text between double quotes `"..."`, to be displayed for the player's benefit at **run-time**.

switch – 1. an optional keyword or symbol to operate special features of the compiler. 2. a statement which decides among different paths of execution according to the value of an expression.

text file – a computer file containing words and phrases which a human can read.

true – a logical state which is the opposite of **false**, represented by any value other than zero (typically 1).

variable – a named value which can change during **run-time**. It must be declared before use, either as a `Global` variable (available to any routine within the game), or as a local variable (part of one specific routine and usable by that routine alone). Variables have a name and a value; it's the value

which is capable of change, not the name. Object **properties** behave as variables.

Z-code file – see **story file**.

Z-Machine – a virtual machine (an imaginary computer simulated by the **interpreter**) on which story files run. Z stands for "Zork", the first ever Infocom title.

Index

Symbols

! (comment character) 29
"..." (character string) 29, 47
"..." (statement) 107, 161
& (bitwise AND operator) 60
&& (boolean AND operator) 58, 60
(...) (in an expression) 151
. (property operator) 64
; (terminating character) 30, 31
<...> (statement) 75, 161
<<...>> (statement) 76, 161
= (assignment operator) 48, 83
== (equality operator) 83
++ (increment operator) 162
-- (decrement operator) 162
-> (indicating object parentage) 164
[...] (routine definition) 158
^ (apostrophe in dictionary word) 36, 48
^ (newline in string) 47
{...} (statement block definition) 78
| (bitwise OR operator) 60
|| (boolean OR operator) 60, 73
~ (double quotes in string) 47
~ (unsetting compiler switches) 22, 171
~ (used to negate attributes) 63, 66
~= (inequality operator) 135

A

accented characters 111
action (library variable) 60
actions 52, 56, 60, 175, 242
after (library property) 56, 58
ambiguous objects 127
animate (library attribute) 71, 76, 125
apostrophes 35, 36, 48
Apple Macintosh 23
Archive, IF 11, 19, 127
arguments (of a routine) 55, 59, 93, 158
article (library property) 75, 125
assignment statements 48, 162
Attribute (directive) 163
attributes 42, 44, 66, 156, 163, 241

B

banners 29, 62
BBEdit Lite editor 24
before (library property) 52, 54, 55, 58, 67
beta-testing 180, 183

C

cant_go (library property) 54, 59
capacity (library property) 44
"Captain Fate" story 105–150, 211–227
child (built-in routine) 157
children (built-in routine) 157
Class (directive) 64
classes 64–68, 156, 230
clothing (library attribute) 73
comments 29
CommonAncestor (library routine) 157
compiler (software tool) 17, 19, 22, 24, 167, 171
  Debug mode 172
  errors and warnings 167
  Strict mode 22, 171, 172, 174
  switches 22, 171
compiling 24, 28, 167–172, 208–210, 227
Constant (directive) 29, 41
container (library attribute) 36, 44, 108

D

d_to (library property) 33
daemon (library property) 117, 176
darkness 142
deadflag (library variable) 40, 90
DeathMessage (entry point routine) 90, 149
Debug mode 172
debugging 173–180
default (in switch statement) 88

description (library property) 31, 43, 64, 69
dictionary words 35, 48, 165
directives 154
door (library attribute) 119
door_dir (library property) 119
door_to (library property) 119
DOSI (mnemonic) 48

E

e_to (library property) 33, 43
each_turn (library property) 40, 43
editor (software tool) 18, 23, 24
  BBEdit Lite 24
  NotePad 18, 23
  SimpleText 18, 24
  TextEdit 18, 24
  TextPad 23
else (in if statement) 80, 85
embedded routines 40, 50, 52, 92, 157, 231
Endif (directive) 148
enterable (library attribute) 108, 112
entry point routines 159, 242
errors (from the compiler) 167
expressions 151, 230
Extend (directive) 101, 175

F

false (built-in constant) 52, 60
FAQs
  Inform 11
  RAIF 11
female (library attribute) 77
files
  library 18, 30
  library extension 127, 163, 170
  source 18, 24, 167
  source template 27, 30
  story 18, 19, 24, 167
flags, *see* attributes
found_in (library property) 72, 82, 97

G

general (library attribute) 162
give (statement) 63, 156
Global (directive) 41
global variables 41
grammar definitions 99
Grammar.h (library file) 30, 101

H

has (object operator) 70, 156
has (part of Object directive) 31, 42, 44, 156
hasnt (object operator) 70, 156
Headline (library constant) 29
"Heidi" story 27–60, 189–193

I

IBM PC 19
if (statement) 48, 58, 60
IF Archive 11, 19, 127
IF Competition 183
Ifdef (directive) 148
in (object operator)) 157
in_to (library property) 33, 53
Include (directive) 30, 170
IndirectlyContains (library routine) 157
indistinguishable objects 74
Infix debugger 172, 178
Infocom games 25, 172
Inform Designer's Manual, The 10, 11, 21, 123, 151, 159, 181, 183
Inform FAQ 11
Inform home page 11
Informary, The 11
inheritance 65
initial (library property) 77
Initialise (entry point routine) 30, 33, 50, 62, 148
InScope (entry point routine) 143
inside_description (library property) 109
interpreter (software tool) 17, 19, 25, 167
invent (library property) 125

K

keep_silent (library variable) 123

L

library actions 242
library attributes 241
library constants 235
library files 18, 30
    contributed 127, 163, 170
library objects 235
library properties 238
library routines 55, 59, 159, 236
library variables 236
`LibraryMessages` (library object) 113
`life` (library property) 77, 117, 130
`light` (library attribute) 31, 44, 64, 65
`lm_n` (library variable) 114
local variables 94, 158
`location` (library variable) 33, 78, 97, 143
`lockable` (library attribute) 119
`locked` (library attribute) 119
`lookmode` (library variable) 63

M

`male` (library attribute) 97
`MANUAL_PRONOUNS` (library constant) 107
`MAX_CARRIED` (library constant) 39
`MAX_SCORE` (library constant) 107
mimesis 142
`move` (statement) 56, 59, 63, 156
`moved` (library attribute) 144

N

`n_to` (library property) 33
`name` (library property) 35, 43, 48, 74, 129, 165
`ne_to` (library property) 33
`new_line` (statement) 118, 161
non-player characters, *see* NPCs
NotePad editor 18, 23
`nothing` (built-in constant) 60, 94
`notin` (object operator) 157
`noun` (grammar token) 99
`noun` (library variable) 57, 60
NPCs 76, 97
`number` (library property) 162
`nw_to` (library property) 33

O

`Object` (directive) 31, 65
object attributes 42, 44, 66, 156, 163, 241
object classes 64–68, 156, 230
object header 42
object names
    external 31, 35, 42, 71, 121
    internal ID 33, 35, 42, 43, 71, 152, 229
object properties 42, 43, 58, 66, 155, 163, 238
object stubs 208, 227
object tree 44, 174
    changing 45, 57, 231
    setting up initial state 36, 45, 156, 163
`OBJECT_SCORE` (library constant) 107
objects 31, 42–47, 155–157, 230
`ofclass` (object operator) 94, 156
`on` (library attribute) 140
`open` (library attribute) 36, 44, 108
`openable` (library attribute) 108
operators 151
`or` (keyword used in conditions) 73, 94
`orders` (library property) 133
`out_to` (library property) 33

P

`parent` (built-in routine) 157
parentheses 151
`parse_name` (library property) 128
parser 127, 177
`Parser.h` (library file) 30
`phrase_matched` (attribute) 163
`PlaceInScope` (library routine) 143
`player` (library variable) 42, 57, 63
player character 31, 76
`PlayerTo` (library routine) 55
`plural` (library property) 75
`pluralname` (library attribute) 71
`pname` (property) 129, 139, 163
`pname.h` (library extension) 128, 163, 170
`print` (statement) 52, 59, 107, 161

print rules 67
print_ret (statement) 55, 59, 107, 161
pronouns 107
proper (library attribute) 77
properties 42, 43, 58, 66, 155, 163, 238
Property (directive) 163
provides (object operator) 156

Q

quotes, single versus double 47, 165

R

RAIF 11, 12, 180
random (built-in routine) 118
real_location (library variable) 143
rec.arts.int-fiction (RAIF) 11, 12, 180
rec.games.int-fiction (RGIF) 11
Release (directive) 62
remove (statement) 156
Replace (directive) 128
return (statement) 52, 57, 59, 158, 161
return values 158
RGIF 11
ROOM_SCORE (library constant) 107
rooms 31
routines 30, 48, 157, 231
    embedded 40, 50, 52, 92, 157, 231
    entry point 159
    library 55, 59, 159, 236
    returning values 92, 93, 158
    standalone 49, 92, 157, 231
    with arguments 55, 59, 93, 158
Ruins.inf (example game) 21

S

s_obj (library object) 84
s_to (library property) 33
scenery (library attribute) 38, 44
score (library variable) 77, 97, 107
scored (library attribute) 107, 139
se_to (library property) 33
second (library variable) 60
self (library variable) 67, 70
semicolons 30, 31, 152, 154
Serial (directive) 62
short_name (library property) 121
sibling (built-in routine) 157
SimpleText editor 18, 24
source files 18, 24, 167
    template 27, 30
standalone routines 49, 92, 157, 231
StartDaemon (library routine) 117
statement blocks 79
statements 40, 48, 152, 231
    assignment 48
static (library attribute) 38, 44
STEF (mnemonic) 94, 158
StopDaemon (library routine) 117
Story (library constant) 29
story files 18, 19, 24, 167
    versions 5 and 8 172
Strict mode 22, 171, 172, 174
strings of characters 29, 47, 165
stub objects 208, 227
supporter (library attribute) 38, 44
sw_to (library property) 33
switch (statement) 87
switchable (library attribute) 140
switches (compiler control) 171
    unsetting 171
syntax colouring 24

T

TADS 9
TextEdit editor 18, 24
TextPad editor 23
thedark (library object) 143
topic (grammar token) 150
transparent (library attribute) 97, 125
true (built-in constant) 52, 60

U

u_to (library property) 33

V

variables
  global 41
  library 236
  local 94, 158
  property 43
`Verb` (directive) 99, 165, 175
`VerbLib.h` (library file) 30
`verblibm.h` (library file) 103
`visited` (library attribute) 70, 78

W

`w_to` (library property) 33
warnings (from the compiler) 167
white space 29, 159
"William Tell" story 13–16, 17, 61–104, 195–210
`with` (part of `Object` directive) 31, 42, 43, 155
`with_key` (library property) 120
words in the dictionary 35, 48, 165
`worn` (library attribute) 63

Z

Z-code files, *see* story files
Z-Machine 24, 25