The Inform Designer's Manual

by Graham Nelson

Second edition

23 October 1995

Contents

Chapter III: Describing and Parsing

Chapter IV: Testing and Hacking

Chapter V: Language and Compiler Reference

Chapter VI: Library Reference

Introduction

> I will build myself a copper tower
> With four ways out and no way in
> But mine the glory, mine the power...
>
> – Louis MacNeice (1907–1963), *Flight of the Heart*

Inform is an adventure-game compiler, and this is the book to read about it.

Infocom format 'story files' (adventure games, that is) can be played on almost any computer, ancient or modern, and interpreters which run them are widely available, from personal organisers to mainframes. They represent probably the most portable form in which games can ever be written, as no alteration whatever is required to move a game from one model of computer to another.

Inform is a suite of software as well as a compiler. Its library (a standard core of game routines) allows designers to begin coding at once. An Inform source file need not contain any of the parser code, or the running of the 'game universe', only descriptions and exceptions to the usual rules. This world is quite rich already, having over 80 verbs and an extensive grammar: the library understands rooms, objects, duplicates, containers, doors, things on top of other things, light, scoring, switching things on and off, opening, closing and locking things, looking up information in books, entering things, travelling about in them and so forth. The parser it uses (which can be entirely invisible to the designer, but is programmable and very flexible) is sophisticated enough to handle ambiguities, to clarify its input by asking questions and to cope properly with plurals, vagueness, conversation, pronouns and the player becoming someone else in mid-game.

The text of this book has evolved from six earlier editions. In Inform's early years, the manual was in places rather technical, with a makeshift and sometimes defensive tone ("Inform is an easel, not a painting"). There were specifications of the run-time code format and literary critiques of games gone by: like an oven manual padded out with both a cookery book and a detailed plan of the gas mains. This book contains just the instructions for the oven.

So there are three 'companion volumes'. *The Craft of Adventure* is an essay on the design of adventure games; *The Specification of the Z-Machine* covers the run-time format and Inform assembly language, its lowest level; and *The Inform Technical Manual* documents chiefly internals, for compiler maintenance and porting.

In trying to be both a tutorial and reference work, this book aims itself in style halfway between the two extremes of manual, Tedium and Gnawfinger's *Elements of Batch Processing in COBOL-66*, third edition, and Mr Blobby's *Blobby Book of Computer Fun*. (This makes some sections both leaden *and* patronising.) I have tried to make every passage tell the truth, so that even early sections are reliable for reference purposes. Passages which divert the main story, usually to tell an unexpurgated truth which may just confuse the newcomer, are marked with a warning triangle △ or two, and set in smaller type.

Many lengthy or involved examples are left as exercises, with full answers given at the back of the book. Harder exercises, marked with triangles, sometimes need knowledge of material later in the book, but most of the easier exercises can be attempted by a first-time reader. For a list of exercises with page references to question and answer, see under "exercises" in the Index.

Most sections end with a 'References' paragraph referring to yet more examples which can be found in the demonstration games which come with Inform. All of these have publically available source code (see the Inform home page): those most frequently referred to are 'Advent' (a full version of the original mainframe 'Adventure', which contains a good deal of "everyday Inform"), 'Adventureland' (a version of Scott Adams's primitive classic), 'Alice Through The Looking-Glass' (a heavily annotated game, developed in the course of Gareth Rees's WWW tutorial for Inform), 'Balances' (a short story consisting of puzzles which stretch the parser's abilities) and 'Toyshop' (hardly a game at all: more an incoherent collection of unusual objects to play with). In addition, the little game 'Ruins' is developed in the course of Chapters I and II of this manual. There is also a 'Shell' game consisting of the minimum code to get going, but it's only 14 lines long and it's essentially given in §1 anyway.

At present, the best source for Inform material (executables of the compiler for different machines, source code, the library files and example games) is the anonymous ftp site `ftp.gmd.de`, and its home directory is:

```
/if-archive/infocom/compilers/inform
```

Another useful resource is the Inform home page on the 'World Wide Web', currently maintained by Gareth Rees at:

```
http://www.cl.cam.ac.uk/users/gdr11/inform
```

This manual describes Inform 5.5 (or later), using library release 5/12 (or later).

Some of the ideas of Inform came from an incremental multi-player game called Tera, on the Cambridge University mainframe, written by Dilip Sequeira and the author in 1990 (whose compiler was called Teraform); in turn, this stole a little from David Seal and Jonathan Thackray's game assembler; which dates back to the late 1970s and was written for 'Acheton', perhaps the first worthwhile game written outside America. Still, much of the Inform kernel derives ultimately from the *IEEE Computer* article 'Zork: A Computerized Fantasy Simulation Game' by P. David Lebling, Marc S. Blank and Timothy A. Anderson; and more was suggested by Richard Tucker and Gareth Rees, among others.

The list of those who have helped the project along is legion: I should like to thank them all, porters, users and critics alike, but especially Volker Blasius, Paul David Doherty, Mark Howell, Bob Newell, Robert Pelak, Gareth Rees, Jørund Rian, Dilip Sequeira, Richard Tucker and Christopher Wichura. Gareth Rees in particular acted as proof-reader and editor for this second edition, greatly improving the text.

One final word. I should like to dedicate this book, impertinently perhaps, to our illustrious predecessors: Willie Crowther, Don Woods and the authors of Infocom, Inc.

Graham Nelson
St Anne's College, Oxford
October 1995

And if no piece of chronicle we prove,
We'll build in sonnets pretty rooms;
As well a well wrought urn becomes
The greatest ashes, as half-acre tombs.

– John Donne (1571?–1631), *The Canonization*

Chapter I: Fundamentals

1 Getting started

The examples in Chapters I and II of this manual will put together a small game called 'Ruins'. As every game does, this will start looking very like the minimal 'Shell' game supplied with Inform:

```
Constant Story "RUINS";
Constant Headline "^An Interactive Worked Example^\
             Copyright (c) 1995 by Graham Nelson.^";
Include "Parser";
Include "VerbLib";
Object Forest "Dark Forest"
  with description
          "In this tiny clearing, the pine-needle carpet is broken by \
           stone-cut steps leading down into darkness.  Dark olive \
           trees crowd in on all sides, the air steams with warm recent \
           rain, midges hang in the air.",
  has  light;
[ Initialise;
  location = Forest;
 "^^^^^Days of searching, days of thirsty hacking through the briars of \
  the forest, but at last your patience was rewarded. A discovery!^";
];
Include "Grammar";
end;
```

If you can compile this successfully, Inform is probably set up and working properly on your computer. Compilation may take a few seconds, because the game 'includes' three library files which contain a great deal more code. These files are themselves written in Inform and contain the core of ordinary rules common to all games:

`Parser`	the game's main loop, and a full parser;
`VerbLib`	routines for many game verbs, like "take";
`Grammar`	a grammar table for decoding the player's input.

The library is certainly modifiable by designers, but great effort has gone into making sure the need seldom arises. Apart from the inclusions, 'Ruins' contains:

(a) strings giving the name of the game and a copyright message, to be printed out at the appropriate moments;

(b) a routine, called `Initialise`, which is run when the game begins, and simply sets where the player starts (in the obvious place!) and prints a 'welcome' message;
(c) an object, to be the only room of the game.

'Ruins' is at this stage a very boring game:

```
Days of searching, days of thirsty hacking through the briars of the forest,
but at last your patience was rewarded. A discovery!
RUINS
An Interactive Worked Example
Copyright (c) 1995 by Graham Nelson.
Release 1 / Serial number 951006 / Inform v1502 Library 5/12
Dark Forest
In this tiny clearing, the pine-needle carpet is broken by stone-cut steps
leading down into darkness.  Dark olive trees crowd in on all sides, the air
steams with warm recent rain, midges hang in the air.
>i
You are carrying nothing.
>north
You can't go that way.
>wait
Time passes.
>quit
Are you sure you want to quit? yes
```

(The "Release" number is 1 unless you set it otherwise, putting a directive like `Release 2;` into the source code. The "Serial number" is set by Inform to the date of compilation.)

In Inform, everything is an object: rooms and items to be picked up, scenery, intangible things like mist and even some abstract ideas (like the direction 'north'). Our second object is added by writing the following just after the `Forest` ends and just before `Initialise` begins:

```
Nearby mushroom "speckled mushroom"
  with name "speckled" "mushroom" "fungus" "toadstool";
```

(`Nearby` just means that the mushroom is inside the last thing declared as an `Object`, in this case the Forest.) The mushroom now appears in the game; the player can call it "speckled mushroom", "mushroom", "toadstool" or even "speckled". It can be taken, dropped, looked at, looked under and so on. However, the description of the Forest says only "There is a speckled mushroom here.", which is still rather plain. So we might extend the definition by:

```
Nearby mushroom "speckled mushroom"
  with name "speckled" "mushroom" "fungus" "toadstool",
       initial
          "A speckled mushroom grows out of the sodden earth, on a long stalk.";
```

The `initial` message is used to tell the player about the mushroom when the Forest is described. (Once the mushroom has been picked or moved, the message is no longer used: hence the name 'initial'.) The mushroom is, however, still "nothing special" when the player asks to "look at" or "examine" it. To provide a more interesting close-up view, we must give the mushroom its own `description`:

```
Nearby mushroom "speckled mushroom"
  with name "speckled" "mushroom" "fungus" "toadstool",
       initial
          "A speckled mushroom grows out of the sodden earth, on a long stalk.",
       description
          "The mushroom is capped with blotches, and you aren't at all sure \
           it's not a toadstool.",
  has  edible;
```

Notice that the mushroom's description is split across two lines of source code. If there were no \ character, then the description would come out with a curious gap of 12 spaces between "sure" and "it's". In a string, the \ character "folds" lines together by telling Inform to ignore the line break and carry on reading from the first non-space character on the next line. (So the gap is just one space after all.) If we want to print a genuine new-line, we can include the special ^ character.

Now if we examine the mushroom, as is always wise before eating, we get a cautionary hint; and, thanks to the `edible` clause, we're now able to eat it.

△ `name`, `description` and `initial` are examples of 'properties', while `edible` and `light` are 'attributes': the difference is that the former have values, whereas the latter are just on or off. They can be defined in any order, and their values can change during play; the original definition only sets up the initial state.

We can go much further with form-filling like this, but for the sake of example we'll begin some honest programming by adding the following property to the mushroom:

```
       after
       [;  Take: "You pick the mushroom, neatly cleaving its thin stalk.";
           Drop: "The mushroom drops to the ground, battered slightly.";
       ],
```

The property `after` doesn't just have a string for a value: it has a routine of its own. Now after something happens to the mushroom, the `after` routine is called to apply any special rules to it. In this case, `Take` and `Drop` are the only actions tampered with, and the only effect is that the usual messages ("Taken." "You eat the speckled mushroom. Not bad.") are replaced. The game can now manage a brief but plausible dialogue:

```
Dark Forest
In this tiny clearing, the pine-needle carpet is broken by stone-cut steps
leading down into darkness.  Dark olive trees crowd in on all sides, the air
steams with warm recent rain, midges hang in the air.
A speckled mushroom grows out of the sodden earth, on a long stalk.
>get mushroom
You pick the mushroom, neatly cleaving its thin stalk.
```

```
>look at it
The mushroom is capped with blotches, and you aren't at all sure it's not a
toadstool.
>drop it
The mushroom drops to the ground, battered slightly.
```

The mushroom is a little more convincing now, but it doesn't do anything yet. We can give it a somewhat sad new rule by adding yet another property, this time with a more substantial routine:

```
before
[; Eat: if (random(100) <= 30)
        {   deadflag = 1;
           "The tiniest nibble is enough. It was a toadstool, \
            and a poisoned one at that!";
        }
        "You nibble at one corner, but the curious taste repels you.";
],
```

The `before` routine is called before the player's intended action takes place. So when the player tries typing, say, "eat the mushroom", what happens is: in 30% of cases, she dies of toadstool poisoning; and in the other 70%, she simply nibbles a corner of fungus (without consuming it completely).

△ Like many programming languages, Inform braces together blocks of code so that several statements can come under the `if` condition. `deadflag` is a global variable, whose value does not belong to any particular object (or routine). It is defined somewhere in the depths of the library: it's usually 0; setting it to 1 causes the game to be lost, and setting it to 2 causes a win.

In either case, the usual rule for the `Eat` action is never applied. This is because, although it isn't obvious from the code, the routine actually returns a value. (In Inform, every routine returns a value.) The command

```
"The tiniest nibble...  ...at that!";
```

is actually a shorthand form of

```
print_ret "The tiniest nibble... ...at that!";
```

What it does is to print the message (together with a carriage return), and then return from (i.e. finish running) the `before` routine, returning the value 'true'. (This is a very convenient shorthand in practice, but if often confuses newcomers: in particular, a routine reading `"Hello."; return 2;` won't return 2 as first appears: it will print "Hello." and a new-line, then return true (actually the number 1). The code coming after `"Hello.";` will simply not be reached.) To return to this example, the library knows that something has interrupted the usual rules of play because the `before` routine didn't return 'false' the way it normally would have.

• **EXERCISE 1**
The present `after` routine for the mushroom is misleading, because it says the mushroom has been picked every time it's taken (which will be odd if it's taken, dropped then taken again). Correct this to complete the definition of the 'Ruins' mushroom.

2 An invitation to Inform

Nothing so difficult as a beginning
In poesy, unless perhaps the end.
– Lord Byron (1788–1824), *Don Juan, IV iv*

All adventure games work in roughly the same way:

1. ask the player to type something;
2. parse this (i.e., decide what it means) and generate any actions it calls for;
3. work out the consequences of these actions and tell the player;
4. worry about time passing, and other things happening.

This process repeats until it ends in either victory or death. The time between one keyboard input and the next is called a 'turn', and many different things may happen during it. Most events taking place in the game are called 'actions' and are discussed in detail in §5.

Probably the most complicated programming in any adventure game goes into the parser. Inform's parser is as good as any, and is designed with the intention of being highly programmable at all levels to suit your game. This book will return to features of the parser again and again. At any rate, the parser can easily be taught to understand commands like:

```
throw three of the coins into the fountain
write funny on the lit cube
take the sword and all the crowns
what is a grue
dwarf, give me the battleaxe
```

It also asks questions when it needs to, makes inferences from partial requests and tries to make good guesses when faced with ambiguous requests. You can teach it 'grammar' for new verbs and new forms of old ones. The library starts with about 85 verbs, not counting synonyms.

Inform has a fairly rich model of the world, which understands ideas such as places, map directions, portable objects, containers, objects on top of each other, food, doors, locks and keys, vehicles, things which can pushed around, people, speech and so on. All of this model is maintained automatically. Just as you don't need to write your own parser, you don't need to tell Inform what to do when the player tries to pick up an ordinary object. Inform is a concise language to write adventure games for, because you only specify exceptions to the usual rules. (E.g., by saying in effect "this football is unusual because if you try to kick it, the following happens".)

Because Inform programs are (hopefully) well-organised lists of exceptions, they don't look like programs written in computer languages like 'BASIC': they aren't executed top-to-bottom. The basic ingredients of an Inform program are objects (an axe, the Bedquilt Room, etc.), routines (e.g. the `Initialise` routine of code which runs when the game starts up) and grammar (instructions on how to make sense of what the player means by typing, e.g., "shoot the photographer").

We have already used the name 'Inform' rather ambiguously. It tends to be used to mean three different things: the compiler itself, the programming language (i.e. the language all Inform games are written in) and the whole 'Library' system of doors, containers and so on. Chapters II to IV (and VI) of this book are about features of the Library and take the underlying language for granted, though hard-to-guess usages are usually glossed the first time they turn up in examples. A concise specification is given in Chapter V.

The rest of this section gives a brief overview of the language, which beginners may wish to skip.

Inform 'source code' (which compiles to the actual game, also called the 'story file') consists of a sequence of 'directives', which are instructions to the compiler to do something and are divided up by semi-colons. Carriage-returns (i.e. new-lines), extra spaces between words and tab characters are of no significance (except in quoted strings), so they can be used to lay out the program to the programmer's own taste. Strings of text are written between double-quotes: the ^ character means 'put a carriage return here' and the \ character 'folds' text together across lines of program.

Inform provides 26 directives to the public (and keeps another 16 to itself for testing and maintenance). So far 'Ruins' only uses 6 of these, and the core of commonly-used directives isn't much larger:

`Array` ⟨arrayname⟩... Creates an array with the given name, size and initial values: see §30.7.

`Class` ⟨classname⟩... Defines a new 'class'. This does not appear in the game, but is instead a template, making it easier to define groups of similar objects: see §5.

`Constant` ⟨constant-name⟩ ⟨value⟩ Defines a new constant with this name and value.

`End` Marks the end of the program (though this isn't needed if it occurs at the end of the file anyway).

`Extend` "⟨word⟩"... Extends the grammar understood by the game when the player's first typed word is ⟨word⟩: see §22 and §31.

`Global` ⟨varname⟩ Creates a new variable with the given name, which is initially zero, unless you add `=` ⟨some-value⟩.

`Nearby` ⟨objectname⟩ "⟨a short description⟩"... Creates an object and puts it initially inside the last object declared with an `Object` directive: see §3.

`Object` ⟨objectname⟩ "⟨a short description⟩"... Creates an object: see §3 and §30.6.

`Verb "`⟨word⟩`"...` Creates a new verb and adds it to the game's table of grammar: see §22 and §31.
`[` ⟨routinename⟩ ⟨local variables...⟩ Tells Inform that a routine of code begins here and has this list of local variables (possibly none at all).

Actual code (that is, instruction telling the computer what to do at run-time) is always held inside routines (or 'functions': the words are used interchangeably). All routines return a value. They can be called (i.e. set running) with up to 7 arguments: e.g., the statement `MoveTheFrog(pond,5);` will call `MoveTheFrog` with the two values `pond` and `5` as arguments (but will throw away the return value as unwanted). If a routine expects, say, 4 arguments it is legal to supply it with fewer, or none at all. Any arguments not supplied will be set to zero.

There are two kinds of variable: local and global. Global variables need to be explicitly created, but they are then available to every piece of code in the program. Local ones are accessible to one routine only and only while it is running. Routines may be called recursively.

The local variables (if any) for a routine are given the names listed at the end of the `[` directive. When a routine is called, the arguments it was called with are written into its first local variables. For example, if `MoveTheFrog` begins as follows:

```
[ MoveTheFrog domain hops x y z;
```

then `MoveTheFrog(pond,5);` will start the routine running with `domain` set to `pond`, `hops` set to 5 and `x`, `y` and `z` all zero.

There are four kinds of array: most common is a 'word array', defined by a directive like

```
Array pizza_toppings --> 20;
```

which makes an array of 20 global variables, called

`pizza_toppings-->0` to `pizza_toppings-->19`.

This is called a 'word' array because each entry is stored in a (16-bit) word of memory, large enough to hold any Inform value. Next there is a 'byte array', written with `->` instead of `-->`, whose entries are stored in single (8-bit) bytes. This is more economical of memory but the entries can only hold ASCII characters or numbers from 0 to 255. The other two kinds of array, `table` and `string`, are similar but have entry `0` set to the array's length. There are elaborate ways to define arrays already stocked with values: see §30.7.

Inside a routine is a sequence of statements, each followed by a semicolon. The core of commonly used statements is small:

⟨variable⟩ `=` ⟨value⟩ Set the variable to be equal to the given value from now on.

⟨variable⟩`++` Adds one to the variable. (`++` can only be applied to an actual named variable, not to an array entry or object property.)

⟨variable⟩`--` Subtracts one from the variable. (`--` can only be applied to an actual named variable, not to an array entry or object property.)

⟨routine⟩`(`⟨arguments⟩...`)` Call the routine with the given arguments (possibly none), and throw away the return value.

`return` ⟨value⟩ Finish the current routine, returning the value given. If no value is given, that value is 'true', or 1.

`rfalse` Return with 'false', or 0.

`rtrue` Return with 'true', or 1.

`<`⟨Action⟩...`>` Cause an action. See §4.

`<<`⟨Action⟩...`>>` Cause an action and then return with 'true', or 1. See §4.

`print` ⟨list-of-items⟩ Print out the given list of items (which are separated by commas). Strings in double-quotes are printed out; numerical values are printed in decimal. For the full specification see §30.14, but the most useful features are for describing objects, for which the beginning of §18 is more explanatory.

`print_ret` ⟨list-of-items⟩ Print out the given list, print a new-line and then return 'true'.

`"Something"` A statement consisting only of a single string in double quotes is an abbreviation for `print_ret` ⟨the-string⟩: thus, it prints the string, prints a new-line and returns 'true'.

`if (`⟨condition⟩`)` ⟨statement(s)⟩ Execute the statement, or group of statements contained inside braces `{` and `}`, only if the condition holds.

`for (`⟨initialise⟩`:`⟨condition⟩`:`⟨iterate⟩`)` ⟨statement(s)⟩ First, execute ⟨initialise⟩. Then repeatedly carry out the ⟨statement⟩ while the ⟨condition⟩ is true. After each iteration, execute ⟨iterate⟩. For instance, `for (i=1:i<=10:i++) print i, " ";` prints the numbers from 1 to 10, separated by spaces. See §30.18 for this and other loop constructs including `objectloop` and `while`.

Conditions include `value1 == value2` (testing for numerical equality; note that `=` is not a legal condition); `value1 ~= value2` (inequality); `<`, `>`, `<=` and `>=` as usual; and `object1 in object2` to test if one is contained directly in the other. (Similarly for `notin`.) See §30.12.

In any Inform statement, a value can be any expression which can be worked out at run-time. This can be a single number or variable, or a combination of these: see §30.10. For instance, `4+5*x` and `100-lamp.time_left` are both expressions. However, in a directive (such as `Object`) values must be constants which can be worked out at compile-time, and moreover compound values cannot be given. For instance, `45` and `lamp` (the object number of a lamp) would be legal constants but `2*5` would not. See §30.3.

△ Inform can produce several kinds of game: "Standard", "Advanced" and a new, much larger form. (Code is in almost every case portable between these formats, so that you can often make a version each way from the same program.) Left to itself Inform produces an "Advanced" game and it's inadvisable to produce a "Standard" one unless you really need to (for a small computer such as a personal organiser, or a very bad run-time interpreter): since the "Standard" format imposes several annoying restrictions. See §27.

• **REFERENCES**
For details of the imaginary machine, sometimes called the Z-machine (Z is for 'Zork') which Inform compiles games for, see §30.2 and (if you must) the *Specification of the Z-machine* document.

3 Objects, properties and attributes

> Objects make up the substance of the world. That is why they cannot be composite.
>
> – Ludwig Wittgenstein (1889–1951), *Tractatus*

> ...making philosophical sense of change runs up against what seem to be impossible philosophical difficulties. Aristotle... focuses on the central case of an object coming to have a property that it formerly lacked.
>
> – Julia Annas, *Classical Greek Philosophy*

The objects of the game form what is sometimes called a 'tree', though a better analogy would be a forest, and anyway one usually draws the whole thing upside down and uses the language of 'families' – calling them 'children' and 'parents' of each other. Anyway, here's an example:

```
 Meadow
   ↓
Mailbox  →  Player
   ↓          ↓
  Note      Sceptre  →  Cucumber  →   Torch   →  MagicRod
                                        ↓
                                     Battery
```

The `Mailbox` and `Player` are both children of the `Meadow`, which is their parent, but only the `Mailbox` is "the" child of the `Meadow`. The `Magic Rod` is the sibling of the `Torch`, which is the `sibling` of the `Cucumber`, and so on. Inform provides special functions for reading off positions in the tree: `parent`, `sibling` and `child` all do the obvious things, and in addition there's a function called `children` which counts up how many children an object has (only children: grandchildren aren't counted). For instance,

```
parent ( Mailbox )  == Meadow
children ( Player ) == 4
child ( Sceptre )   == nothing
sibling ( Torch )   == Magic Rod
```

△ `nothing` isn't really an object: it's just a convenient name for the number 0, which is the object number meaning 'no such object'. It isn't a good idea to meddle with `nothing`, or to apply functions like `parent` to it, but then there's never any need.

As the game goes on, objects move around: when an object moves, all its possessions (that is, children) go with it. The Inform statement to move an object is `move`. For instance, `move Cucumber to Mailbox;` results in the tree

```
  Meadow
     ↓
  Mailbox  →   →   →  Player
     ↓                   ↓
 Cucumber  →  Note     Sceptre  →  Torch  →  MagicRod
                                     ↓
                                  Battery
```

but it must be emphasized that `move` prints nothing on the screen, and indeed does nothing except to rearrange the tree. Using `remove`, an object can be detached from the tree altogether (so that its parent is `nothing`) though this does not delete it from the game, and it may return.

When an object is added to the possessions held by another, it appears at the front of the list, becoming "eldest" in the family-tree sense. Inform also provides the following functions, with names along the same lines:

`youngest`	end of list of possessions;
`eldest`	same as `child`;
`younger`	same as `sibling`, i.e., one step right in the picture;
`elder`	reverse of `sibling`, i.e., one step left in the picture.

Objects contain more than just a position in the tree; they also have collections of variables attached. Firstly, there are 'attributes' (in more usual computer parlance, 'flags'), which can be either on or off. These might be such conditions as "giving light", "currently worn" or "is one of the featureless white cubes". Attributes all have one-word names: like `light`, for instance, which indicates that something is giving off light. An attribute can be checked with the `has` or `hasnt` condition:

```
if (obj has locked) "But it's locked!";
if (TreasureRoom hasnt light) "All these rooms are the same in the dark.";
```

Attributes are set with the `give` command:

```
give brass_lantern light;
give iron_door locked;
```

and are similarly taken away:

```
give brass_lantern ~light;
give fake_coin ~scored;
```

the ~ sign (or tilde) standing for negation. You can give or take many at one go, as for example

```
give wooden_door open openable ~locked;
```

Secondly, there are 'properties'. These are far more elaborate, and not every object has every property. For instance, not every object has the `door_to` property (it holds the place a door leads to, so things other than doors don't usually have it). The current value of a property is got at by constructions like:

```
crystal_bridge.door_to
mushroom.before
diamond.initial
```

You can read the value of `door_to` for something like the `diamond`, and you'll just get a dull value like `nothing`, but you can't write to it: that is, you can't change `diamond.door_to` unless you declared the `diamond` with a `door_to` property.

As will be seen from examples, a property value can be many things: a string like `"frog"`, a number such as `$ffff` (this is the Inform way of writing numbers in hexadecimal, so it means 65535), an object or a routine. You can change the current value by something like

```
location.door_to = hall_of_mists;
brass_lantern.short_name = "dimly lit brass lantern";
grenade.time_left = 45;
```

• **WARNING**

The game may crash at run-time if you attempt to write to a property field which an object hasn't got. So although you can read an undeclared property (you just get the default value), you can't write to one. (Also, you can't extend a property beyond its length: see below.)

△ The Inform language does not have types as such, and strings and routines are stored as numbers: as their addresses inside the virtual machine, in fact. This means that Inform thinks `"Hello there" + 45 + 'a'` is a perfectly sensible calculation. It's up to you to be careful.

△△ Actually one can partially work out the type of a property value: see §17.

A property can hold more than just one number (be it interpreted as a string, a routine or whatever): it can hold a small array. For instance, the definition of the object `broken_shells` might contain

```
found_in  Marble_Hall  Arched_Passage  Stone_Stairs,
```

which stores a sequence of three values in the `found_in` property.

△ The .& operator produces the property as an array:

```
print (name) (broken_shells.&found_in)-->2;
```

might print "Stone Stairs". One usually needs to know how long this array is, and for that the .# operator is needed:

```
print broken_shells.#found_in;
```

would print 6, because the array is 6 bytes long, which makes 3 --> entries. (When a list is given in an object definition like this, the values are put into a --> array, where each entry takes up two bytes. However, you can if you wish read and write to it as a -> byte array.)

△△ Normally the array can be up to 64 bytes long, so you can either treat it as a -> array of length up to 64 or a --> array of length up to 32. But Standard games restrict this to 8 bytes' worth. If you give a property more than 8 bytes of data in a Standard game, Inform warns you and takes only the first 8.

• △ **EXERCISE 2**
Use the `object.&property` construction to find out whether the object in variable `obj` has the `door_to` property defined or not.

Time to make some object definitions. A typical object definition looks something like:

```
Object steps "stone-cut steps" Forest
  with name "steps" "stone" "stairs" "stone-cut",
       description
          "The cracked and worn steps descend into a dim chamber.  Yours \
           might be the first feet to tread them for five hundred years.",
       door_to Square_Chamber,
       door_dir d_to
  has  scenery door open;
```

This is the conventional way to lay out an `Object` declaration: with the header first, then `with` a list of properties and their starting values, finishing up with the attributes it initially `has`. (Though `with` and `has` can be given the other way round.)

△ For the full `Object` syntax, see §30.6.

`steps` is the name given to the object in the program, and it becomes a constant (whose value is the number of the object). The `Forest` is the object which the `steps` start out belonging to. Some objects begin with no parent: rooms, for example, or a priest who will magically appear half-way through the game. You could declare these as belonging to `nothing`, but it's simpler just to miss this out altogether:

```
Object magician "Zadok the Priest"
  with ...
```

If you do declare an object already belonging to another, as above, then the other object must have been defined earlier on the source. This restriction is useful because it prevents you from setting up a 'loop' – one object in another in a third in the first, for instance.

Objects can also be declared, in an identical way, by the `Nearby` directive. The only difference is that no initial-owner object can be given; it starts out belonging to the last thing declared as an `Object`. For example, in

```
Object hillside "Panoramic Hillside"
  with ...
Nearby the_hills "rolling hills"
  with ...
```

the `hillside` is a room to which `the_hills` will belong. Otherwise, `Nearby` is the same as `Object`, and this is just a convenience to make it easier to move things around in Inform code by cutting definitions out and pasting them in elsewhere.

For the sake of flexibility, most properties of objects can be given as routines to work out a value (instead of giving just the value). For instance, you can give a routine for `description` instead of a string, and it will be called instead of being printed. This routine can then print something suitable, perhaps changing with the circumstances. The stone-cut steps in 'Ruins' use just such a routine:

```
description
[;  print "The cracked and worn steps descend into a dim chamber. \
           Yours might ";
    if (Square_Chamber has visited)
         print "be the first feet to tread";
    else print "have been the first feet to have trodden";
    " them for five hundred years.  On the top step is inscribed \
     the glyph Q1.";
],
```

(The glyphs will be explained to the player in §11.) Note that the routine is 'anonymous': there is no name after the `[`. Since there are no local variables either, a semi-colon follows immediately. It would be legal to write the routine elsewhere in the program, with a name, and give just the name here as the property value: but less tidy.

The routine must end with either `],` or `];`. If `],` the object definition can resume where it left off. If `];`, then the object definition ends where the routine finishes.

△ The rules for embedded routines are not quite the same as those for ordinary routines. By default, embedded routines return "false", or 0 (instead of "true", or 1, which other routines return by default). Also, the handy shorthand:

⟨Action1⟩`,` ⟨Action2⟩...: ...⟨some code⟩...

is provided, which executes the code only if the action currently being considered is one of those named. There can also be a `default` clause which executes if and only if none of the others do.

△ One property is treated differently from all others, and this is the special property `name`. Its data must be a list of English words in double-quotes, as in all the above examples. (Probably the most annoying restriction of Standard games is that this means only 4 names at most can be accommodated in that format: normally you get up to 32.) The parser may not be able to do much with name-words like `"my"`, `"the"`, `"all"`, `"except"` or `"this"`, but English numbers like `"four"` or direction names like `"south"` can safely be used.

△△ This is actually the only circumstance in which Inform syntax puts dictionary words in double, rather than single, quotes.

• **REFERENCES**
To see the object tree in action, compile one of the shorter games (say 'Alice Through The Looking-Glass') with the line `Constant DEBUG;` inserted at the top: when played in this form, special debugging verbs (see §26) become available. Amongst others, "tree" displays the current tree and "routines" makes the game print a trace message each time it executes a routine which is the property value of some object.

4 Actions and reactions

> Only the actions of the just
> Smell sweet and blossom in their dust.
>
> – James Shirley (1594–1666), *The Contention of Ajax and Ulysses*

> ...a language obsessed with action, and with the joy of seeing action multiply from action, action marching relentlessly ahead and with yet more actions filing in from either side to fall into neat step at the rear, in a long straight rank of cause and effect, to what will be inevitable, the only possible end.
>
> – Donna Tartt, *The Secret History*

Inform is a language obsessed with . An 'action' is an attempt to perform one simple task: for instance,

```
Inv     Take sword    Insert gold_coin cloth_bag
```

are all examples. Here the actual actions are `Inv`, `Take` and `Insert`. An action has 0, 1 or 2 objects supplied with it (or, in a few special cases, some numerical information rather than objects): internally, actions are stored as three numbers. Most actions are triggered off by the parser, whose job can be summed up as reducing the player's keyboard commands to actions. Some actions cause others to happen, and a really complicated keyboard command ("empty the sack into the umbrella stand") can cause a long sequence of actions to begin. A good way to get a feel for this is to compile one of Inform's example games with the `DEBUG` constant defined (see §26) so that you can use the special "actions" verb to watch them happen.

It must be stressed that an action is only an attempt to do something, which may or may not succeed. Firstly, a `before` rule might interfere, as we have seen already. Secondly, the action might not even be very sensible. The parser is interested almost exclusively in syntax and will happily generate the action `Eat iron_girder` if that's what the player has asked to do.

Actions can also be generated in the program, which perfectly simulates the effect of a player typing something. As an example of why this is so useful, suppose the air in the Pepper Room causes the player to sneeze each turn and drop something at random. If the code to do this simply moves an object to the floor, then it might accidentally provide a solution to a problem like "the toffee apple sticks to your hands so you can't drop it". If, however, it generates a `Drop` action instead, then the result might read:

> You sneeze convulsively, and lose your grip on the toffee apple...
> The toffee apple sticks to your hand!

which is at least coherent and consistent. Besides this, actions are useful because some effects (e.g., looking around) are inconveniently hard to code by hand.

As an example of causing actions, an odorous `low_mist` will soon settle over 'Ruins' (see §6). It will have the `description` "The mist carries a rich aroma of broth." An alert player who reads this will immediately ask to smell the mist. This won't do him any good, as we're only going to repeat the same description. A neat way to accomplish this is to make the action `Smell low_mist` turn into the action `Examine low_mist` instead. We need only add a `before` rule for the mist as follows:

```
Smell: <Examine self>; rtrue;
```

The statement `<Examine self>` causes the action `Examine low_mist` to be triggered off immediately, after which whatever was going on at the time resumes. In this case, the action `Smell low_mist` resumes, but since we immediately return 'true' the action is stopped dead. Note that `self` is a variable: inside the mist object it has the value `low_mist`, and in general its value is the object whose property is currently being run.

Causing an action and then returning 'true' (effectively converting the present action to a different one) is needed so often that there is a shorthand form, putting the action in double angle-brackets. For example,

```
<Look>; <<ThrowAt smooth_stone spider>>;
```

will behave as if the player has asked to look around and to throw the stone at the spider, and will then return true.

Actions are processed in a simple way, but one which involves many little stages. There are three main stages:

(a) 'Before'. An opportunity for your code to interfere with or block altogether what is happening. Unless you provide such code, this stage is always passed over.
(b) 'During'. The 'Verblib' part of the library takes control and looks at the action to see if it is possible according to Inform's model of the world (for instance, only an `edible` object may be eaten; only an object in the player's possession can be thrown at somebody, and so on). If the action is impossible, it prints a complaint and stops. Otherwise the action is carried out.
(c) 'After'. An opportunity for your code to react to what has happened, after it has happened but before any text announcing it has been printed. If it chooses, your code can print and cause an entirely different outcome. If your code doesn't interfere, the library reports back to the player (with such choice words as "Dropped.").

The 'Before' stage consults your code in five ways, and occasionally it's useful to know in what order:

i. The `GamePreRoutine` is called, if you have written one. If it returns 'true', nothing else happens and the action is stopped.
ii. The `orders` property of the player is called on the same terms. For more details, see §15.
iii. And the `react_before` of every object in scope (which roughly means 'in the vicinity').
iv. And the `before` of the current room.
v. If the action has a first noun, its `before` is called on the same terms.

The 'After' stage is similar, but runs in the sequence: `react_after` rules for every object in scope (including the player object); the room's `after`; the first noun's `after` and finally `GamePostRoutine`.

During action processing, the variables `action`, `noun` and `second` contain the numbers which encode the action. For example, if `Take white_flag` is being processed, then `action` is set to the constant `##Take` (every action has a constant value corresponding to it, written in Inform as `##` followed by its name); `noun` is `white_flag` and `second` is zero.

△ Certain 'meta-verbs' cause actions which bypass the Before and After stages: these are for commands to control the game program, like `Save` or `Verify`.

△△ To some extent you can also meddle with the 'During' stage (and with the final messages produced) by cunning use of the `LibraryMessages` system. See §17.

△△ For some actions, the 'noun' (or the 'second noun') is actually a number (for instance, "set timer to 20" would probably be parsed with `noun` being `timer` and `second` being 20). The variables `inp1` and `inp2` hold object numbers only, or 1 to indicate 'some number'. (For instance, here `inp1` would be `timer` but `inp2` would be 1.)

As mentioned above, the parser can generate very peculiar actions, which are only realised to be impossible after `before` rules have taken place. For example, in 'Ruins' the parser would accept "put the mushroom in the crate" even if the mushroom were nearby but, say, sealed inside a glass jar. A `before` rule to cover this action may therefore want to check that the `mushroom is in the player` before acting.

• △ **EXERCISE 3**

This kind of snag could be avoided altogether if Inform had a 'validation stage' in action processing, to check whether an action is sensible before allowing it to get as far as `before` rules. How could this be added to Inform?

The library supports about 120 different actions and any game of serious proportion will add some more of its own. A full list of standard actions is given in §38. This list is initially daunting but you actually don't need to remember much of it, partly because complicated actions are usually reduced to simple ones. Thus, for instance, the action `<Empty rucksack table>`, meaning "empty the contents of the rucksack onto the table", is broken down into a stream of actions such as `<Remove fish rucksack>` then `<PutOn fish table>`. The library arranges things so that, in particular, the only way an object can enter the player's possession is via a `Take` or `Remove` action.

Earlier editions of this book divided up the actions into three groups, and the names 'Group 2', etc., stuck. Group 1 contains the 'meta' actions, which are not worth listing here (see §38). Group 2 contains the most important actions, which normally change the state of the game:

```
Inv, Take, Drop, Remove, PutOn, Insert, Enter, Exit, Go, Look, Examine,
Unlock, Lock, SwitchOn, SwitchOff, Open, Close, Disrobe, Wear, Eat, Search.
```

Most actions ordinarily do nothing and these form Group 3: for instance in response to `Listen` the library always responds "You hear nothing unexpected." (unless a `before` rule has dealt with the action first). Because the library never actually does anything at the 'During' stage, there is never an 'After' stage for a Group 3 action, and no `after` routines are called. In rough order of usefulness, the list is:

```
Pull, Push, PushDir [push object in direction], Turn, ThrowAt,
Consult, LookUnder [look underneath something], Search,
Listen, Taste, Drink, Touch, Smell,
Wait, Sing, Jump [jump on the spot], JumpOver, Attack,
Swing [something], Blow, Rub, Set, SetTo, Wave [something],
Burn, Dig, Cut, Tie, Fill, Swim, Climb, Buy, Squeeze,
Pray, Think, Sleep, Wake, WaveHands [i.e., just "wave"],
WakeOther [person], Kiss, Answer, Ask, ThrowAt,
Yes, No, Sorry, Strong [swear word], Mild [swear word]
```

△ Actions involving other people, like `Kiss`, need not be handled by a `before` rule: it's more convenient to use the `life` rule (see §12).

△ A very few actions (e.g., `Transfer`, `Empty`, `GetOff`) are omitted from the list above because they're always translated into more familiar ones. For instance, `InvWide` (asking for a "wide–format" inventory listing) always ends up in an `Inv`.

△ Note that some actions only ever print text and yet are in Group 2, not Group 3, because this is so useful. The most interesting example is `Search` (the searching or looking-inside-something action), whose 'During' stage is spent only on deciding whether it would be sensible to look inside the object (e.g., if it's a see-through container and there is light). Only if it's sensible is 'After' allowed to happen, and only after that is the list of contents printed out. Thus, a `before` rule applied to `Search` traps the searching of random scenery, while an `after` can be used to alter the contents-listing rules.

△△ Most of the group 2 actions – specifically,

```
Take, Drop, Insert, PutOn, Remove, Enter, Exit, Go, Unlock, Lock,
SwitchOn, SwitchOff, Open, Close, Wear, Disrobe, Eat
```

can happen "silently". If the variable `keep_silent` is set to 1, then these actions print nothing in the event of success. (E.g., if the door was unlocked as requested.) They print up objections as usual if anything goes wrong (e.g., if the suggested key doesn't fit). This is useful to implement implicit actions: for instance, to code a door which will be automatically unlocked by a player asking to go through it, who is holding the right key.

The library's actions are easily added to. Two things are necessary to add a new action: first one must provide a routine to run it, e.g.,

```
[ BlorpleSub;
  "You speak the magic word ~Blorple~.  Nothing happens.";
];
```

somewhere after the `Initialise` routine, say, to be tidy. Every action must have such a routine, the name of which is always the name of the action with `Sub` appended. The 'During' stage of processing an action consists only of calling this routine.

Secondly, one must write a new action into the game's grammar table. Far more about grammar will come later: in this case one need only add the simplest of all grammar lines

```
Verb "blorple" *                              -> Blorple;
```

after the inclusion of the `Grammar` file. (The spacing around the `*` is just a matter of convention.) The word "blorple" can now be used as a verb but it can't take any nouns.

The action `Blorple` is now a typical Inform action, which joins Group 3 above (since it doesn't do anything very interesting). One can use the command `<Blorple>;` to make it happen, and can write `before` routines to trap it, just like any Group 3 action.

△△ You can make a Group 1 action by defining the verb as `meta` (see §22); and a Group 2 one by putting the line

```
  if (AfterRoutines()==1) rtrue;
```

into the action routine after carrying out the action, and before printing a description of what has been done. (Calling `AfterRoutines` sets off the 'After' stage, which otherwise won't happen.)

△ Finally, Inform supports 'fake actions'. These are fake in two senses. Firstly they aren't mentioned in any grammar (so they can never be generated by the parser, only by a `<`, `>` command). Secondly, they have no `-Sub` routine and if they aren't trapped at the 'Before' stage then nothing further happens. Fake actions are provided to enable you to pass 'messages' to an object, which the recipient can pick up in its `before` routine. A fake action has to be explicitly declared before use, by the directive `Fake_action` ⟨Action-name⟩.

• △△ EXERCISE 4

How can you make a medicine bottle, which can be opened in a variety of ways in the game, so that the opening–code only occurs in the bottle definition?

The Library actually makes a few fake actions itself (two of which will appear in the next section). A simple example is `ThrownAt`, useful for greenhouse windows, coconut shies and the like. If a `ThrowAt` action (to throw object X at object Y) survives the `before` rules (for X), then the fake action `ThrownAt` is generated and sent to Y, allowing Y to react. E.g., a dartboard might have:

```
before
[;  ThrownAt: if (noun==dart)
              {   move dart to self; "Triple 20!"; }
              move noun to location;
              print_ret (The) noun, " bounces back off the board.";
],
```

No `after` rule applies, as the default behaviour of `ThrowAt` has by then simply rebuked the player.

• △ EXERCISE 5

`ThrownAt` would be unnecessary if Inform had an idea of `before` and `after` routines which an object could provide if it were the `second` noun of an action. How might this be implemented?

△△ Some `before` or `after` rules are intended to apply only once in the course of a game. For instance, examining the tapestry reveals a key, only once. A sneaky way to do this is to make the appropriate rule destroy itself, so for example

```
    tapestry.before = NULL;
```

removes the entire `before` rule for the tapestry. `NULL` is a special value, actually equal to -1, which routine-valued properties like `before`, `after`, `life` and `describe` hold to indicate "no routine is given".

• REFERENCES

In a game compiled with `Constant DEBUG;` present, the "actions" verb will result in trace information being printed each time any action is generated. Try putting many things into a rucksack and asking to "empty" it for an extravagant list. • Diverted actions (using `<<` and `>>`) are commonplace. They're used in about 20 places in 'Advent': a good example is the way "take water" is translated into a `Fill bottle` action. • 'Balances' uses a fake action called `Baptise` which tells one of the white cube objects that a name has been written on it. (It also sends a special 'memory' object the actions `Insert` and `Remove` to pass "learn this spell" and "forget this spell" messages.) • Sometimes you want 'fake fake actions' which are fully–fledged actions (with action routines and so on) but which aren't ever generated by the parser: see the exercises at the end of §12.

5 Classes of objects

On a round ball
A workman that hath copies by, can lay
An Europe, Afrique and an Asia,
And quickly make that, which was nothing, All.

– John Donne (1571?–1631), *Valediction: Of Weeping*

In most games there are groups of objects with certain rules in common. Inform allows you to define classes in almost exactly the same way as objects. The only difference between the layout of a class and object definition is that a class has no short name or initial location, since it does not correspond to any single real item. For example, the scoring system in 'Ruins' works as follows: the player, an archaeologist of the old school, gets a certain number of points for each 'treasure' (i.e., cultural artifact) he can filch and put away into his packing case. This is implemented with a class:

```
 Class Treasure
  with number 10,
       after
       [; Insert:
                if (second==packing_case) score=score+self.number;
                "Safely packed away.";
       ],
       before
       [; Take, Remove:
                if (self in packing_case)
                   "Unpacking such a priceless artifact had best wait \
                    until the Metropolitan Museum can do it.";
       ];
```

An object of this class inherits the properties and attributes it defines: in this case, an object of class `Treasure` picks up the given score and rules automatically. So

```
Nearby statuette "pygmy statuette"
 class Treasure
  with description
          "A menacing, almost cartoon-like statuette of a pygmy spirit \
           with a snake around its neck.",
       initial "A precious Mayan statuette rests here!",
       name "snake" "mayan" "pygmy" "spirit" "statue" "statuette";
```

inherits the `number` value of 10 and the rules about taking and dropping. If the statuette had itself set `number` to 15, say, then the value would be 15: i.e., the class would be over-ridden.

△ `number` is a general-purpose property, left free for designers to use as they please. One might instead define a new property called, say, `depositpoints` and use that, for clarity: see §17 for how to do this.

A more unusual artifact in the 'Ruins' is:

```
Nearby honeycomb "ancient honeycomb"
 class Treasure
  with article "an",
       name "ancient" "old" "honey" "honeycomb",
       description "Perhaps some kind of funerary votive offering.",
       initial "An exquisitely preserved, ancient honeycomb rests here!",
       after
       [;  Eat: "Perhaps the most expensive meal of your life.  The honey \
                tastes odd, perhaps because it was used to store the entrails \
                of the king buried here, but still like honey.";
       ],
  has  edible;
```

Now the honeycomb has two `after` rules: a private one of its own, and the one all treasures have. Both apply, but its own private one takes precedence, i.e., happens first.

△ An object can inherit from several classes at once. Moreover, a class can itself inherit from other classes, so it's easy to make a class for "like Treasure but with `number` = 8".

△ The `class` field of an object definition contains a list of classes,

`class` C_1 ... C_n

in which case the object inherits first from C_1, then from C_2 and so on. C_2 over-rides C_1 and so on along the line. These classes may well disagree with each other, so the order matters. If C_1 says `number` is 5, C_3 says it is 10 but the object definition itself says 15 then the answer is 15.

△ With some properties, the value is not replaced but added to: this is what happened with `after` above. These properties are those which were declared as `additive`, e.g. by

```
Property additive before NULL;
```

For instance, the standard Inform properties `name` and `before` are both additive. So we could add `name "treasure",` to the properties in the class definition for `Treasure`, and then all objects of that class would respond to the word "treasure", as well as their own particular names.

△△ An additive property can contain a list in which some items are strings and others routines. Should this occur, then on a `PrintOrRun` (what usually happens when a property is being looked up) the entries are executed in sequence – run if routines, printed if strings. A printed string in such a list always has a new-line printed after it; and it never stops the process of execution. In other words, the string `"Hello"` is equivalent to the routine `[; print "Hello^"; ],` (which returns false), not to the routine `[; "Hello"; ],` (which would return true and stop execution). This will seldom be useful but protects the Z-machine stack against certain misfortunes.

• **REFERENCES**

'Advent' has a similar treasure-class, and uses class definitions for the many similar maze and dead-end rooms (and the sides of the fissure). • That class definitions can be worthwhile for just two instances can be seen from the kittens-class in 'Alice Through The Looking-Glass'. • 'Balances' defines many complicated classes: see especially the white cube, spell and scroll classes. • 'Toyshop' contains one easy one (the wax candles) and one unusually hard one (the building blocks). • See §35 for which of the library's properties are additive.

Chapter II: The Model World

A Model must be built which will get everything in without a clash; and it can do this only by becoming intricate, by mediating its unity through a great, and finely ordered, multiplicity.

– C. S. Lewis (1898–1963), *The Discarded Image*

6 Places, scenery, directions and the map

It was a long cylinder of parchment, which he unrolled and spread out on the floor, putting a stone on one end and holding the other. I saw a drawing on it, but it made no sense.

– John Christopher (1922–), *The White Mountains*

Back to 'Ruins': what lies at the foot of the stone steps? We'll now add four rooms, connected together:

```
Square Chamber ↔ Web
       ↕
    Corridor
       ↕
     Shrine
```

with the Square Chamber lying underneath the original Forest location. For instance, here's the Square Chamber's definition:

```
Object Square_Chamber "Square Chamber"
  with name "lintelled" "lintel" "lintels" "east" "south" "doorways",
       description
          "A sunken, gloomy stone chamber, ten yards across.  A shaft \
           of sunlight cuts in from the steps above, giving the chamber \
           a diffuse light, but in the shadows low lintelled doorways to \
           east and south lead into the deeper darkness of the Temple.",
       u_to Forest, e_to Web, s_to Corridor,
  has  light;
```

Like the Forest, this place has `light`, however dim. (If it didn't, the player would never see it, since it would be dark, and the player hasn't yet been given a lamp or torch of some kind.) Now although this is a room, and can't be referred to by the player in the way that a manipulable object can, it still can have a `name` property. These `name` words are those which Inform knows "you don't need to refer to", and it's a convention of the genre that the designer should signpost off the game in this way. Note that they'll only be looked at if what the player types is unrecognised, so the word "east" is understood quite normally; but a reference to "east lintel" will get the "don't need to refer to" treatment. This room is unfurnished, so:

```
Nearby inscriptions "carved inscriptions"
  with name "carved" "inscriptions" "carvings" "marks" "markings" "symbols"
            "moving" "scuttling" "crowd" "of",
       initial
          "Carved inscriptions crowd the walls, floor and ceiling.",
       description "Each time you look at the carvings closely, they seem \
           to be still.  But you have the uneasy feeling when you look \
           away that they're scuttling, moving about.  Their meaning \
           is lost on you.",
  has  static;
```

This is part of the fittings, hence the `static` attribute, which means it can't be taken or moved. As we went out of our way to describe a shaft of sunlight, we'll include that as well:

```
Nearby sunlight "shaft of sunlight"
  with name "shaft" "of" "sunlight" "sun" "light" "beam" "sunbeam" "ray"
            "rays" "sun^s",
       description "The shaft of sunlight glimmers motes of dust in the \
           air, making it seem almost solid."
  has  scenery;
```

Being `scenery` makes the object not only static but also not described by the game unless actually examined by the player. A true perfectionist might add a `before` rule:

```
       before
       [;  Examine, Search: ;
           default: "It's only an insubstantial shaft of sunlight.";
       ],
```

so that the player can look at or through the sunlight, but any other request involving them will be turned down. Note that a `default` rule, if given, means "any action except those already mentioned".

We can't actually get into the Square Chamber yet, though. Just because there is a map connection up from here to the Forest, it doesn't follow that there's a corresponding connection down. So we must add a `d_to` to the Forest, and while we're at it:

```
       d_to Square_Chamber,
       u_to "The trees are spiny and you'd cut your hands to ribbons \
             trying to climb them.",
```

```
cant_go "The rainforest-jungle is dense, and you haven't hacked \
    through it for days to abandon your discovery now.  Really, \
    you need a good few artifacts to take back to civilization \
    before you can justify giving up the expedition.",
```

The property `cant_go` contains what is printed when the player tries to go in a nonexistent direction, and replaces "You can't go that way". As is often the case with properties, instead of giving an actual message you can instead give a routine to print one out, to vary what's printed with the circumstances. The Forest needs a `cant_go` because in real life one could go in every direction from there: what we're doing is explaining the game rules to the player: go underground, find some ancient treasure, then get out to win. The Forest's `u_to` property is a string, not a room; this means that attempts to go up result only in that string being printed.

Rooms also have rules of their own. We might add the following `before` rule to the Square Chamber:

```
before
[;  Insert:
        if (noun==mushroom && second==sunlight)
        {   remove mushroom;
           "You drop the mushroom on the floor, in the glare of \
            the shaft of sunlight.  It bubbles obscenely, \
            distends and then bursts into a hundred tiny insects \
            which run for the darkness in every direction.  Only \
            tiny crumbs of fungus remain.";
        }
],
```

The variables `noun` and `second` hold the first and second nouns supplied with an action. Rooms have `before` and `after` routines just as objects do, and they apply to anything which happens in the given room. This particular could easily enough have been part of the definition of the mushroom or the sunlight, and in general a room's rules are best used only for geographical fixtures.

△△ Sometimes the room may be a different one after the action has taken place. The `Go` action, for instance, is offered to the `before` routine of the room which is being left, and the `after` routine of the room being arrived in. For example:

```
after
[; Go: if (noun==d_obj)
       print "You feel on the verge of a great discovery...^";
],
```

will print the message when the room is entered via the "down" direction. Note that the message is printed with the `print` command. This means that it does not automatically return true: in fact, it returns false, so the game knows that the usual rules still apply. Also, no new-line is printed automatically: but the ^ symbol means "print a new-line", so one is actually printed.

Some objects are present in many rooms at once. The 'Ruins', for instance, are misty:

```
Object low_mist "low mist"
  with name "low" "swirling" "mist",
       initial "A low mist swirls about your feet.",
       description "The mist carries a rich aroma of broth.",
       found_in  Square_Chamber  Forest,
       before
       [; Examine, Search: ;
          Smell:   <<Examine self>>;
          default: "The mist is too insubstantial.";
       ],
  has  static;
```

The `found_in` property gives a list of places in which the mist is found (so far just the Square Room and the Forest).

△ If the rainforest contained many misty rooms, it would be tedious to give the full list and even worse to have to alter it as the mist drifted about in the course of the game. Fortunately `found_in` can contain a routine instead of a list. This can look at the current `location` and say whether or not the object should be put in it when the room is entered, e.g.,

```
Object Sun "Sun",
  with ...
       found_in
       [; if (location has light) rtrue;
       ],
  has  scenery;
```

△△ `found_in` is only consulted when the player's location changes, so if the mist has to dramatically lift or move then it needs to be moved or removed 'by hand'. A good way to lift the mist forever is to `remove` it, and then give it the `absent` attribute, which prevents it from manifesting itself whatever `found_in` says.

Some pieces of scenery afflict the other four senses and need more than a visual description. For instance, the player ought to be able to smell broth anywhere near the mist. A `react_before` rule is ideal for this:

```
       react_before
       [;  Smell: if (noun==0) <<Smell low_mist>>;
       ],
```

This rule (when added to the mist) applies to any vague `Smell` action (that is, caused by the player typing just "smell", rather than "smell orange") which happens when the mist is in the vicinity of the player: and it converts the action into `Smell low_mist`. In this way the mist is able to 'steal' the action.

The five senses all have actions in Inform: `Look` we have already seen, and there are also `Listen`, `Smell`, `Taste` and `Touch`. Of these, `Look` never has a noun attached, `Smell` and `Listen` can have and `Taste` and `Touch` always have.

• EXERCISE 6

(Cf. 'Spellbreaker'.) Make an orange cloud descend on the player, which can't be seen through or walked out of.

• △ EXERCISE 7

In the first millenium A.D., the Mayan peoples of the Yucatán Peninsula had 'world colours' white (*sac*), red (*chac*), yellow (*kan*) and black (*chikin*) for what we call the compass bearings north, east, south, west (for instance west is associated with 'sunset', hence black, the colour of night). Implement this.

• △ EXERCISE 8

(Cf. 'Trinity'.) How can the entire game map be suddenly east-west reflected?

• △△ EXERCISE 9

Even when the map is reflected, there may be many room descriptions referring to "east" and "west" by name. Reflect these too.

△ The ordinary Inform directions all have the `number` property defined (initially set to zero): this is to provide a set of scratch variables useful, for instance, when coding mazes.

△△ If the constant `WITHOUT_DIRECTIONS` is defined before inclusion of the library files, then 10 of the default direction objects are not defined by the library. The designer is expected to define alternative ones (and put them in the `compass` object); otherwise the game will be rather static. (The "in" and "out" directions are still created, because they're needed for getting into and out of enterable objects.)

• REFERENCES

'Advent' has a very tangled-up map in places (see the mazes) and a well-constructed exterior of forest and valley giving an impression of space with remarkably few rooms. The mist object uses `found_in` to the full, and see also the stream (a single object representing every watercourse in the game). Bedquilt and the Swiss Cheese room offer classic confused-exit puzzles. • For a simple movement rule using `e_to`, see the Office in 'Toyshop'. • The library extension "smartcantgo.h" by David Wagner provides a system for automatically printing out "You can only go east and north."-style messages. • 'A Scenic View', by Richard Barnett, demonstrates a system for providing examinable scenery much more concisely (without defining so many objects).

7 Containers, supporters and sub-objects

> The concept of a surface is implemented as a special kind of containment. Objects which have surfaces on which other objects may sit are actually containers with an additional property of "surfaceness".
>
> – P. David Lebling, *Zork and the Future*

> The year has been a good one for the Society *(hear, hear)*. This year our members have put more things on top of other things than ever before. But, I should warn you, this is no time for complacency. No, there are still many things, and I cannot emphasize this too strongly, *not* on top of other things.
>
> – 'The Royal Society For Putting Things On Top Of Other Things'
> *Monty Python's Flying Circus*, programme 18 (1970)

Objects can be inside or on top of one another. An object which has the `container` attribute can contain things, like a box: one which has `supporter` can hold them up, like a table. (An object can't have both at once.) It can hold up to 100 items, by default: this is set by the `capacity` property. However, one can only put things inside a container when it has `open`. If it has `openable`, the player can open and close it at will, unless it also has `locked`. A `locked` object (whether it be a door or a container) cannot be opened. But if it has `lockable` then it can be locked or unlocked with the key object given in the `with_key` property. If it is undeclared, then no key will fit, but this will not be told to the player, who can try as many as he likes.

Containers (and supporters) are able to react to things being put inside them, or removed from them, by acting on the signal to `Receive` or `LetGo`. For example, deep under the 'Ruins' is a chasm which, perhaps surprisingly, is implemented as a container:

```
Nearby chasm "horrifying chasm"
  with name "blackness" "chasm" "pit" "depths" "horrifying" "bottomless",
       react_before
       [;  Jump: <<Enter self>>;
           Go: if (noun==d_obj) <<Enter self>>;
       ],
       before
       [;  Enter: deadflag=1;
               "You plummet through the silent void of darkness!";
       ],
       after
       [;  Receive: remove noun;
                print_ret (The) noun, " tumbles silently into the \
                     darkness of the chasm.";
           Search: "The chasm is deep and murky.";
       ],
  has  scenery open container;
```

(Actually the definition is a little longer, so that the chasm reacts to a huge pumice-stone ball being rolled into it; see 'Ruins'.) Note the use of an `after` rule for the `Search` action: this is because an attempt to "examine" or "look inside" the chasm will cause this action. `Search` means, in effect, "tell me what is inside the container" and the `after` rule prevents a message like "There is nothing inside the chasm." from misleading the player. Note also that the chasm 'steals' any stray `Jump` action and converts it into an early death.

• **EXERCISE 10**

Make the following, rather acquisitive bag:

```
>put fish in bag
The bag wriggles hideously as it swallows the fish.
>get fish
The bag defiantly bites itself shut on your hand until you desist.
```

△ `LetGo` and `Receive` are actually two of the fake actions: they are the actions `Insert` and `Remove` looked at from the container's point of view.

△ `Receive` is sent to an object O both when a player tries to put something in O, and put something on O. In the rare event that O needs to react differently to these, it may consult the variable `receive_action` to find out whether `##PutOn` or `##Insert` is the cause.

The 'Ruins' packing case makes a fairly typical container:

```
Nearby packing_case "packing case"
  with name "packing" "case" "box" "strongbox",
       initial
          "Your packing case rests here, ready to hold any important \
           cultural finds you might make, for shipping back to civilisation.",
       before
       [;  Take, Remove, PushDir:
               "The case is too heavy to bother moving, as long as your \
                expedition is still incomplete.";
       ],
  has  static container open;
```

Now suppose you want to make a portable television set which has four different buttons on it. Obviously when the television moves, its buttons should move with it, and the sensible way to arrange this is to make the four buttons possessions of the `television` object. Since the television isn't a `container`, though, the player can't normally "get at" (that is, refer to) its possessions. So how do we bring the buttons "into scope" so that the player can refer to them, without allowing the player to remove or add to them? The `transparent` attribute is provided for this: it simply means "the sub-objects of this object can be referred to by the player".

• **EXERCISE 11**

Implement a television set with attached power button and screen.

• **EXERCISE 12**
Make a glass box and a steel box, which would behave differently when a lamp is shut up inside them.

△ It sometimes happens that an object should have sub-objects (such as lamps and buttons) quite separately from its possessions, in which case the above solution is unsatisfactory. Fuller details will be given in the "scope addition" rules in §24, but briefly: an object's `add_to_scope` property may contain a list of sub-objects to be kept attached to it (and these sub-objects don't count as possessions).

• **EXERCISE 13**
Implement a macramé bag hanging from the ceiling, inside which objects are visible (and audible, etc.) but cannot be touched or manipulated in any way.

• **REFERENCES**
Containers and supporters abound in the example games (except 'Advent', which is too simple, though see the water-and-oil carrying bottle). Interesting containers include the lottery-board and the podium sockets from 'Balances' and the 'Adventureland' bottle. • For supporters, the hearth-rug, chessboard, armchair and mantelpiece of 'Alice Through The Looking-Glass' are typical examples; the mantelpiece and spirit level of 'Toyshop' makes a simple puzzle, and the pile of building blocks a complicated one; see also the scales in 'Balances'.

8 Doors

> Standing in front of you to the north, however, is a door surpassing anything you could have imagined. For starters, its massive lock is wrapped in a dozen six-inch thick iron chains. In addition, a certain five-headed monster...
>
> – Marc Blank and P. David Lebling, *'Enchanter'*

> O for doors to be open and an invite with gilded edges
> To dine with Lord Lobcock and Count Asthma.
>
> – W. H. Auden (1907–1973), *Song*

A useful kind of object is a `door`. This need not literally be a door: it might be a rope-bridge or a ladder, for instance. To set up a `door`:

(a) give the object the `door` attribute;
(b) set its `door_to` property to the destination;
(c) set its `door_dir` property to the direction which that would be, such as `n_to`;
(d) make the room's map connection in that direction point to the door itself.

For example, here is a closed and locked door, blocking the way into the 'Ruins' shrine:

```
Object Corridor "Stooped Corridor"
  with description "A low, square-cut corridor, running north to south, \
           stooping you over.",
       n_to Square_Chamber,
       s_to StoneDoor;
Nearby StoneDoor "stone door"
  with description "It's just a big stone door.",
       name "door" "massive" "big" "stone" "yellow",
       when_closed
           "Passage south is barred by a massive door of yellow stone.",
       when_open
           "The great yellow stone door to the south is open.",
       door_to Shrine,
       door_dir s_to,
       with_key stone_key
  has  static door openable lockable locked;
```

Note that the door is `static` – otherwise the player could pick it up and walk away with it! The properties `when_closed` and `when_open` give descriptions appropriate for the door in these two states.

Doors are rather one-way: they are only really present on one side. If a door needs to be accessible (openable and lockable from either side), a neat trick is to make it present in both locations and to fix the `door_to` and `door_dir` to the right way round for whichever side the player is on. Here, then, is a two-way door:

```
Nearby StoneDoor "stone door"
  with description "It's just a big stone door.",
       name "door" "massive" "big" "stone" "yellow",
       when_closed
           "The passage is barred by a massive door of yellow stone.",
       when_open
           "The great yellow stone door is open.",
       door_to
       [;  if (location==Corridor) return Shrine; return Corridor; ],
       door_dir
       [;  if (location==Shrine) return n_to; return s_to; ],
       with_key stone_key,
       found_in  Corridor  Shrine,
  has  static door openable lockable locked;
```

where `Corridor` has `s_to` set to `StoneDoor`, and `Shrine` has `n_to` set to `StoneDoor`. The door can now be opened, closed, entered, locked or unlocked from either side. We could also make `when_open` and `when_closed` into routines to print different descriptions of the door from inside and out.

At first sight, it isn't obvious why doors have the `door_dir` property. Why does a door need to know which way it faces? The point is that two different actions cause the player to go through the door. Suppose the door is in the south wall. The player may type "go south", which directly causes the action `Go s_obj`. Or the player may "enter door" or "go through door", causing `Enter the_door`. Provided the door is actually open, the `Enter` action then looks at the door's `door_dir` property, finds that the door faces south and generates the action `Go s_obj`. Thus, however the player tries to go through the door, it is the `Go` action that finally results.

This has an important consequence: if you put `before` and `after` routines on the `Enter` action for the `StoneDoor`, they only apply to a player typing "enter door" and not to one just typing "south". So one safe way is to trap the `Go` action. A neater method is to put some code into a `door_to` routine. If a `door_to` routine returns 0 instead of a room, then the player is told that the door "leads nowhere" (like the famous broken bridge of Avignon). If `door_to` returns 1, or 'true', then the library stops the action on the assumption that something has happened and the player has been told already.

• **EXERCISE 14**
Create a plank bridge across a chasm, which collapses if the player walks across it while carrying anything.

• **REFERENCES**
'Advent' is especially rich in two-way doors: the steel grate in the streambed, two bridges (one of crystal, the other of rickety wood) and a door with rusty hinges. See also the iron gate in 'Balances'.

9 Switchable objects

> Steven: 'Well, what does this do?' Doctor: 'That is the dematerialising control. And that over yonder is the horizontal hold. Up there is the scanner, these are the doors, that is a chair with a panda on it. Sheer poetry, dear boy. Now please stop bothering me.'
>
> – Dennis Spooner, *The Time Meddler*
> *Dr Who*, serial 17 (1965)

Objects can also be `switchable`. This means they can be turned off or on, as if they had some kind of switch on them. The object has the attribute `on` if it's on. For example:

```
Object searchlight "Gotham City searchlight" skyscraper
  with name "search" "light" "template", article "the",
       description "It has some kind of template on it.",
       when_on "The old city searchlight shines out a bat against \
                  the feather-clouds of the darkening sky.",
       when_off "The old city searchlight, neglected but still \
                   functional, sits here."
  has  switchable static;
```

Something more portable would come in handy for the explorer of 'Ruins', who would hardly have embarked on his expedition without a decent light source...

```
Object sodium_lamp "sodium lamp"
  with name "sodium" "lamp" "heavy",
       describe
       [;  if (self hasnt on)
               "^The sodium lamp squats heavily on the ground.";
          "^The sodium lamp squats on the ground, burning away.";
       ],
       number 40,
       before
       [;  Examine: print "It is a heavy-duty archaeologist's lamp, ";
               if (self hasnt on) "currently off.";
               if (self.number < 10) "glowing a dim yellow.";
               "blazing with brilliant yellow light.";
           Burn: <<SwitchOn self>>;
           SwitchOn:
               if (self.number <= 0)
                  "Unfortunately, the battery seems to be dead.";
               if (parent(self) hasnt supporter && self notin location)
                  "The lamp must be securely placed before being lit.";
           Take, Remove:
               if (self has on)
                  "The bulb's too delicate and the metal frame's too \
                   hot to move the lamp while it's switched on.";
       ],
       after
       [;  SwitchOn: give self light;
           SwitchOff: give self ~light;
       ],
  has  switchable;
```

The 'Ruins' lamp will eventually be a little more complicated, with a daemon to make the battery strength, held in the `number` property, run down and to extinguish the lamp when it runs out; and it will be pushable from place to place, making it not quite as useless as the player will hopefully think at first.

△ A point to note is that this time the `when_on` and `when_off` properties haven't been used to describe the lamp when it's on the ground: this is because once an object has been held by the player, it's normally given only a perfunctory mention in room descriptions ("You can also see a sodium lamp and a grape here."). The `describe` property has priority over the whole business of how objects are described in room descriptions. When it returns true, as above, the usual description process does nothing further. For much more on room descriptions, see §18.

• REFERENCES

The original switchable object was the brass lamp from 'Advent' (which also provides verbs "on" and "off" to switch it). (The other example games are generally pre-electric in setting.)

10 Things to enter, travel in and push around

> ...the need to navigate a newly added river prompted the invention of vehicles (specifically, a boat).
>
> – P. David Lebling, Marc Blank and Timothy Anderson

Some objects in a game are `enterable`, which means that a player can get inside or onto them. The idea of "inside" here is that the player is only half-in, as with a car or a psychiatrist's couch. (If it's more like a prison cell, then it should be a separate place.) In practice one often wants to make an `enterable` thing also a `container`, or, as in the altar from 'Ruins', a `supporter`:

```
Nearby stone_table "slab altar"
  with name "stone" "table" "slab" "altar" "great",
       initial "A great stone slab of a table, or altar, dominates the Shrine.",
  has  enterable supporter;
```

A chair to sit on, or a bed to lie down on, should also be a `supporter`.

• **EXERCISE 15**
(Also from 'Ruins'.) Implement a cage which can be opened, closed and entered.

All the classic games have vehicles (like boats, or fork lift trucks, or hot air balloons) which the player can journey in, so Inform makes this easy. Here is a simple case:

```
Object car "little red car" cave
  with name "little" "red" "car",
       description "Large enough to sit inside.  Among the controls is a \
           prominent on/off switch.  The numberplate is KAR 1.",
       when_on  "The red car sits here, its engine still running.",
       when_off "A little red car is parked here.",
       before
       [; Go: if (car has on) "Brmm!  Brmm!";
              print "(The ignition is off at the moment.)^";
       ],
  has  switchable enterable static container open;
```

Actually, this demonstrates a special rule. If a player is inside an `enterable` object and tries to move, say "north", the `before` routine for the object is called with the action `Go`, and `n_obj` as the noun. It may then return:

- 0 to disallow the movement, printing a refusal;
- 1 to allow the movement, moving vehicle and player;
- 2 to disallow but print and do nothing; or
- 3 to allow but print and do nothing.

If you want to move the vehicle in your own code, return 3, not 2: otherwise the old location may be restored by subsequent workings.

Because you might want to drive the car "out" of a garage, the "out" verb does not make the player get out of the car. Usually the player has to type something like "get out" to make this happen, though of course the rules can be changed.

• **EXERCISE 16**

Alter the car so that it won't go east.

△ Objects like the car or, say, an antiquated wireless on casters, are obviously too heavy to pick up but the player should at least be able to push them from place to place. When the player tries to do this, the `PushDir` action is generated. Now, if the `before` routine returns false, the game will just say that the player can't; and if it returns true, the game will do nothing at all, guessing that the `before` routine has already printed something more interesting. So how does one actually tell Inform that the push should be allowed? The answer is that one has to do two things: call the `AllowPushDir` routine (a library routine), and then return true. For example ('Ruins' again):

```
Nearby huge_ball "huge pumice-stone ball"
  with name "huge" "pumice" "pumice-stone" "stone" "ball",
       description "A good eight feet across, though fairly lightweight.",
       initial
           "A huge pumice-stone ball rests here, eight feet wide.",
       before
       [;  PushDir:
               if (location==Junction && second==w_obj)
                   "The corridor entrance is but seven feet across.";
               AllowPushDir(); rtrue;
           Pull, Push, Turn: "It wouldn't be so very hard to get rolling.";
           Take, Remove: "There's a lot of stone in an eight-foot sphere.";
       ],
       after
       [;  PushDir:
              if (second==s_obj) "The ball is hard to stop once underway.";
              if (second==n_obj) "You strain to push the ball uphill.";
       ],
  has  static;
```

• △ **EXERCISE 17**

The library does not normally allow pushing objects up or down. How can the pumice ball allow this?

• **REFERENCES**

For an `enterable supporter` puzzle, see the magic carpet in 'Balances' (and several items in 'Alice Through The Looking-Glass').

11 Reading matter and consultation

> Even at present... we still know very little about how access to printed materials affects human behaviour.
>
> – Elizabeth Eisenstein, *The Printing Revolution in Early Modern Europe*

look up figure 18 in the engineering textbook

is a difficult line for Inform to understand, because almost anything could appear in the first clause: even its format depends on what the second clause is. This kind of request, and more generally

look up ⟨any words here⟩ in ⟨the object⟩
read about ⟨any words here⟩ in ⟨the object⟩
consult ⟨the object⟩ about ⟨any words here⟩

cause the `Consult object` action. Note that `second` is just zero: formally, there is no second noun attached to a `Consult` action. The object has to parse the ⟨any words here⟩ part itself, in a `before` rule for `Consult`. The following variables are set up to make this possible:

`consult_from` holds the number of the first word in the ⟨any...⟩ clause;
`consult_words` holds the number of words in the ⟨any...⟩ clause (at least 1).

The words given are parsed using library routines like `NextWord()`, `TryNumber(word-number)` and so on: see §20 for full details. As usual, the `before` routine should return true if it has managed to deal with the action; returning false will make the library print "You discover nothing of interest in...".

Little hints are placed here and there in the 'Ruins', written in the glyphs of an ancient dialect of Mayan. Our explorer has, of course, come equipped with the latest and finest scholarship on the subject:

```
Object dictionary "Waldeck's Mayan dictionary"
  with name "dictionary" "local" "guide" "book" "mayan"
             "waldeck" "waldeck^s",
       description "Compiled from the unreliable lithographs of the \
           legendary raconteur and explorer ~Count~ Jean Frederic \
           Maximilien Waldeck (1766??-1875), this guide contains \
           what little is known of the glyphs used in the local \
           ancient dialect.",
       before
       [ w1 w2 glyph other;  Consult:
               if (consult_words>2) jump GlyphHelp;
               wn=consult_from;
               w1 = NextWord(); ! First word of subject
```

```
                w2 = NextWord(); ! Second word (if any) of subject
                if (consult_words==1 && w1=='glyph' or 'glyphs')
                    jump GlyphHelp;
                !  We want to recognise both "glyph q1" and "q1 glyph":
                glyph=w1; other=w2;
                if (w1=='glyph') { glyph=w2; other=w1; }
                !  So now glyph holds the name, and other the other word
                if (consult_words==2 && other~='glyph') jump GlyphHelp;
                switch(glyph)
                {   'q1': "(This is one glyph you have memorised!)^^\
                           Q1: ~sacred site~.";
                    'circle': "Circle: ~the Sun; also life, lifetime~.";
                    ...
                    default: "That glyph is so far unrecorded.";
                }
                !  All three of the ways the text can go wrong lead to
                !  this message being produced:
               .GlyphHelp; "Try ~look up <name of glyph> in book~.";
       ],
  has  proper;
```

Note that this understands any of the forms "q1", "glyph q1" or "q1 glyph" but is careful to reject, for instance, "glyph q1 glyph". (These aren't genuine Mayan glyphs, but some of the real ones have similar names, dating from when their syllabic equivalents weren't known: G8, the Lord of the Night, for instance.)

• **EXERCISE 18**

To mark the 500th anniversary of William Tyndale (the first English translator of the New Testament), prepare an edition of the four Gospels.

△△ Ordinarily, a request by the player to "read" something is translated into an `Examine` action. But the "read" verb is defined independently of the "examine" verb in order to make it easy to separate the two requests. For instance:

```
Attribute legible;
...
Object textbook "textbook"
  with name "engineering" "textbook" "text" "book",
       description "What beautiful covers and spine!",
       before
       [; Consult, Read:
           "The pages are full of senseless equations.";
       ],
       has  legible;
...
[ ReadSub; <<Examine noun>>; ];
Extend "read" first * legible                        -> Read;
```

Note that "read" causes a `Read` action only for `legible` objects, and otherwise causes `Examine` in the usual way. `ReadSub` is coded as a translation to `Examine` as well, so that if a `legible` object doesn't provide a `Read` rule then an `Examine` happens after all.

• **REFERENCES**
If you really need more elaborate topic-parsing (for, e.g., "look up ⟨something⟩ in the catalogue", where any object name might appear) then extending the grammar for `look` may be less trouble. For a good implementation see 'Encyclopaedia Frobozzica', by Gareth Rees.

12 Living creatures and conversation

To know how to live is my trade and my art.
– Michel de Montaigne (1533–1592), *Essays*

Everything that can be said can be said clearly.
– Ludwig Wittgenstein (1889–1951), *Tractatus*

This rummage through special kinds of objects finishes up with the most sophisticated kind: living ones. Note that the finer points of this section, on the arts of conversation, require some knowledge of Chapter III.

Animate objects, such as sea monsters, mad aunts or nasty little dwarves, have a property called `life`. This behaves somewhat like a `before` or `after` routine, but only applies to the following actions:

`Attack` The player is making hostile advances…

`Kiss` …or excessively friendly ones…

`WakeOther` …or simply trying to rouse the creature from sleep.

`ThrowAt` The player asked to throw `noun` at the creature.

`Give` The player asked to give `noun` to the creature…

`Show` …or, tantalisingly, just to show it.

`Ask` The player asked about something. Just as with a "consult" topic (see §11 passim), the variables `consult_from` and `consult_words` are set up to indicate which words the object might like to think about. (In addition, `second` holds the dictionary value for the first word which isn't `'the'`, but this is much cruder.)

`Tell` Likewise, the player is trying to tell the creature about something. The topic is set up just as for `Ask` (that is, `consult_from` and `consult_words` are set, and `second` also holds the first interesting word).

`Answer` This can happen in two ways. One is if the player types "answer ⟨some text⟩ to troll" or "say ⟨some text⟩ to troll"; the other is if he gives an order which the parser can't sort out, such as "troll, og south", and which the `orders` property hasn't handled already. Once again, variables are set as if it were a "consult" topic. (In addition, `noun` is set to the first word, and an attempt to read the text as a number is stored in the variable `special_number`: for instance, "computer, 143" will cause `special_number` to be set to 143.)

`Order` This catches any 'orders' which aren't handled by the `orders` property (see below); `action`, `noun` and `second` are set up as usual.

If the `life` routine doesn't exist, or returns false, events take their usual course. `life` routines tend to be quite lengthy, even for relatively static characters such as the priest who stands in the 'Ruins' Shrine:

```
Nearby priest "mummified priest"
  with name "mummified" "priest",
       description
          "He is desiccated and hangs together only by will-power.  Though \
           his first language is presumably local Mayan, you have the curious \
           instinct that he will understand your speech.",
       initial "Behind the slab, a mummified priest stands waiting, barely \
           alive at best, impossibly venerable.",
       life
       [; Answer: "The priest coughs, and almost falls apart.";
          Ask:      switch(second)
                    {   'dictionary', 'book':
                             if (dictionary has general)
                                 "~The ~bird~ glyph... very funny.~";
                            "~A dictionary? Really?";
                        'glyph', 'glyphs', 'mayan', 'dialect':
                            "~In our culture, the Priests are ever literate.~";
                        'king', 'tomb', 'shrine', 'temple', 'altar', 'slab':
                            "~The King (life! prosperity! happiness!) is buried \
                             deep under this Shrine, where you will never go.~";
                    }
                    "~You must find your own answer.~";
          Tell:   "The priest has no interest in your sordid life.";
          Attack, Kiss:  remove self;
                   "The priest desiccates away into dust until nothing \
                    remains, not a breeze nor a bone.";
          ThrowAt: move noun to location; <<Attack self>>;
          Show, Give:
                    if (noun==dictionary && dictionary hasnt general)
                    {   give dictionary general;
                       "The priest reads a little of the book, laughing \
                        in a hollow, whispering way.  Unable to restrain \
                        his mirth, he scratches in a correction somewhere \
```

```
                        before returning the book.";
                    }
                   "The priest is not very interested in earthly things.";
       ],
  has  animate;
```

(Some of the `Ask` topics are omitted for brevity.) Of course an `animate` object still has `before` and `after` routines like any other, so you can trap many other kinds of behaviour. Animate creatures can also `react_before` and `react_after`, and it's here that these properties really come into their own:

```
react_before
[; Drop: if (noun==satellite_gadget)
        print "~I wouldn't do that, Mr Bond,~ says Blofeld.^^";
   Shoot: remove beretta;
       "As you draw, Blofeld snaps his fingers and a giant \
        magnet snatches the gun from your hand.  It hits the \
        ceiling with a clang.  Blofeld silkily strokes his cat.";
];
```

If Blofeld moves from place to place, these rules move with him.

• EXERCISE 19

Arrange for a bearded psychiatrist to place the player under observation, occasionally mumbling insights such as "Subject puts green cone on table. Interesting."

Another example is the coiled snake from 'Balances', which shows that even the tiniest `life` routine can be adequate for an animal:

```
Nearby snake "hissing snake"
  with name "hissing" "snake",
       initial "Tightly coiled at the edge of the chasm is a hissing snake.",
       life [; "The snake hisses angrily!"; ],
  has  animate;
```

△ When writing general code to deal with `animate` creatures, it's sometimes convenient to have a system worked out for printing pronouns such as "her" and "He". See §18 for one way to do this.

Sometimes creatures should be `transparent`, sometimes not. Consider these two cases of `animate` characters, for instance:

- an urchin with something bulging inside his jacket pocket;
- a hacker who has a bunch of keys hanging off his belt.

The hacker is `transparent`, the urchin not. That way the parser prevents the player from referring to whatever the urchin is hiding, even if the player has played the game before, and knows what is in there and what it's called. But the player can look at and be tantalised by the hacker's keys.

When the player types in something like "pilot, fly south", and the parser is able to make sense of the request, the result is called an 'order': this is the corresponding idea to an 'action' (but happens to other people rather than to the player). This order is sent to the pilot's `orders` property, which has the chance to comply with the request (if it likes). Inform itself never carries out any orders: if no rules get in the way, it will simply print something like "The pilot has better things to do." The above priest is especially unhelpful:

```
orders
[;  Go: "^I must not leave the Shrine.^";
    NotUnderstood: "^You speak in riddles.^";
    default: "^It is not your orders I serve.^";
];
```

(The `NotUnderstood` clause is run when the parser couldn't understand what the player typed.) Something to bear in mind is that because the library regards the words "yes" and "no" as being verbs in Inform, it understands "delores, yes" as being a `Yes:` order. (This can be a slight nuisance, as "say yes to orc" is treated differently: it gets routed through the `life` routine as an `Answer`.)

△ If the `orders` property returns false (or if there wasn't an `orders` property in the first place), the order is sent either to the `Order:` part of the `life` property (if it's understood) or to the `Answer:` part (if it isn't). (This is how all orders used to be processed, and it's retained to avoid making reams of old Inform code go wrong.) If these also return false, a message like "X has better things to do" or "There is no reply" is finally printed.

To clarify the various kinds of conversation:

Command	rule	action	noun	second	consult	
"orc, take axe"	`order`	`Take`	`axe`	`0`		
"orc, yes"	`order`	`Yes`	`0`	`0`		
"ask orc for the shield"	`order`	`Give`	`player`	`shield`		
"orc, troll"	`order`	`NotU...`	`'troll'`	`orc`	3	1
"say troll to orc"	`life`	`Answer`	`'troll'`	`orc`	2	1
"answer troll to orc"	`life`	`Answer`	`'troll'`	`orc`	2	1
"orc, tell me about coins"	`life`	`Ask`	`orc`	`'coins'`	6	1
"ask orc about the big troll"	`life`	`Ask`	`orc`	`'big'`	4	3
"ask orc about wyvern"	`life`	`Ask`	`orc`	`0`	4	1
"tell orc about lost troll"	`life`	`Tell`	`orc`	`'lost'`	4	2

where "wyvern" is a word not mentioned anywhere in the program, which is why its value is `0`.

• EXERCISE 20

In some ways, `Answer` and `Tell` are just too much trouble. How can you make attempts to use these produce a message saying "To talk to someone, try 'someone, something'."?

Some objects are not alive as such, but can be spoken to: microphones, tape recorders, voice-activated computers and so on. It would be a nuisance to implement these as `animate`, since they have none of the other characteristics of life: instead, they can be given just the attribute `talkable` and `orders` and `life` properties to deal with the resulting conversation.

• EXERCISE 21
(Cf. 'Starcross'.) Construct a computer responding to "computer, theta is 180".

△ The rest of this section starts to overlap much more with Chapter III, and assumes a little familiarity with the parser.

△ The `NotUnderstood` clause of `orders` is run when the parser has got stuck parsing an order like "pilot, fly somersaults". The variable `etype` holds the parser error that would have been printed out, had it been a command by the player himself. See §25: for instance, `CANTSEE_PE` would mean "the pilot can't see any such object".

△ When the player issues requests to an `animate` or `talkable` object, they're normally parsed exactly as if they were commands by the player himself (except that the `actor` is now the person being spoken to). But sometimes one would rather they were parsed by an entirely different grammar. For instance, consider Zen, the flight computer of an alien spacecraft. It's inappropriate to tell Zen to (say) pick up a teleport bracelet and the crew tend to give commands more like:

> "Zen, set course for Centauro"
> "Zen, speed standard by six"
> "Zen, scan 360 orbital"
> "Zen, raise the force wall"
> "Zen, clear the neutron blasters for firing"

This could mostly be implemented by adding verbs like "raise" to the usual game grammar (see the 'Starcross' computer exercise above), or by carefully trapping the `Answer` rule. But this is a nuisance, especially if about half the commands you want are recognised as orders in the usual grammar but the other half aren't.

An `animate` or `talkable` object can therefore provide a `grammar` routine (if it likes). This is called at a time when the parser has worked out the object that is being addressed and has set the variables `verb_num` and `verb_word` (to the number of the 'verb' and its dictionary entry, respectively: for example, in "orac, operate the teleport" `verb_num` would be 3 (because the comma counts as a word on its own) and `verb_word` would be `'operate'`). The `grammar` routine can reply by returning:

0. The parser carries on as usual.
1. The `grammar` routine is saying it has done all the parsing necessary itself, by hand (i.e., using `NextWord`, `TryNumber`, `NounDomain` and the like): the variables `action`, `noun` and `second` must be set up to contain the resulting order.

`'verb'` The parser ignores the usual grammar and instead works through the grammar lines for the given verb (see below).

`-'verb'` Ditto, except that if none of those grammar lines work then the parser goes back and tries the usual grammar as well.

In addition, the `grammar` routine is free to do some partial parsing of the early words provided it moves on `verb_num` accordingly to show how much it's got through.

• △ EXERCISE 22
Implement Charlotte, a little girl who's playing Simon Says (a game in which she only follows your instructions if you remember to say "Simon says" in front of them: so she'll disobey "charlotte, wave" but obey "charlotte, simon says wave").

• △ EXERCISE 23

Another of Charlotte's rules is that if you say a number, she has to clap that many times. Can you play?

• △ EXERCISE 24

Regrettably, Dyslexic Dan has always mixed up the words "take" and "drop". Implement him anyway.

△ It's useful to know that if the player types a comma or a full stop, then the parser cuts these out as separate words. Because of this, a dictionary word containing up to 7 letters and then a comma or a full stop can never be matched by what the player types. This means that a verb with such a name is hidden from the ordinary grammar - but it can still be used by a `grammar` routine. For instance, here's a way to implement the 'Starcross' computer which doesn't involve creating foolish new actions. We create grammar:

```
[ Control;
  switch(NextWord())
  {   'theta': parsed_number=1; return 1;
      'phi':   parsed_number=2; return 1;
      'range': parsed_number=3; return 1;
      default: return -1;
  }
];
Verb "comp," * Control "is" number -> SetTo;
```

And the computer itself needs properties

```
       grammar [; return 'comp,'; ],
       orders
       [;  SetTo:
               switch(noun)
               {   1: print "~Theta"; 2: print "~Phi"; 3: print "~Range"; }
               print_ret " set to ", second, ".~";
           default: "~Does not compute!~";
       ];
```

This may not look easier, but it's much more flexible, as the exercises below will hopefully demonstrate.

△△ Another use for 'untypeable verbs' is to create what might be called 'fake fake actions'. Recall that a fake action is one which is never generated by the parser and is used for message-passing only, so it doesn't have an action routine and can't do anything other than send the message. Sometimes, though, you want a proper action (with its own action routine) which also can't be generated by the parser. The following example creates three of these:

```
Verb "actions." * -> Prepare * -> Simmer * -> Cook;
```

The parser never uses "actions." in its ordinary grammar, so this definition has the sole effect of creating three new actions: `Prepare`, `Simmer` and `Cook`.

• △△ EXERCISE 25

How can you make a grammar extension to an ordinary verb that will apply only to Dan?

• △ **EXERCISE 26**
Make an alarm clock responding to "alarm, off", "alarm, on" and "alarm, half past seven" (the latter to set its alarm time).

• △ **EXERCISE 27**
Implement a tricorder (from Star Trek) which analyses nearby objects on a request like "tricorder, the quartz stratum".

• △ **EXERCISE 28**
And, for good measure, a replicator responding to commands like "replicator, tea earl grey" and "replicator, aldebaran brandy".

• △△ **EXERCISE 29**
And a communications badge in contact with the ship's computer, which answers questions like "computer, where is Admiral Lebling".

• △△ **EXERCISE 30**
Finally, construct the formidable flight computer Zen.

The next two exercises really belong to §24, but are too useful (for the "someone on the other end of a phone" situation) to bury far away. Note that an alternative to these scope-hacking tricks, if you just want to implement something like "michael, tell me about the crystals" (when Michael is at the other end of the line), is to make the phone a `talkable` object and make the word `'michael'` refer to the phone (using a `parse_name` routine).

For more on scope hacking, see §24. Note that the variable `scope_reason` is always set to the constant value `TALKING_REASON` when the game is trying to work out who you wish to talk to: so it's quite easy to make the scope different for conversational purposes.

• △ **EXERCISE 31**
Via the main screen of the Starship Enterprise, Captain Picard wants to see and talk to Noslen Maharg, the notorious tyrant, who is down on the planet Mrofni. Make it so.

• △△ **EXERCISE 32**
Put the player in telepathic contact with Martha, who is in a sealed room some distance away, but who has a talent for telekinesis. Martha should respond well to "martha, look", "ask martha about...", "say yes to martha", "ask martha for red ball", "martha, give me the red ball" and the like.

• **REFERENCES**
A much fuller example of a 'non-player character' is given in the example game 'The Thief', by Gareth Rees (though it's really an implementation of the gentleman in 'Zork', himself an imitation of the pirate in 'Advent'). The thief is capable of walking around, being followed, stealing things, picking locks, opening doors and so on. • Other good definitions of `animate` objects to look at are Christopher in 'Toyshop', who will stack up building blocks on request; the kittens in 'Alice Through The Looking-Glass'; the barker in 'Balances', and the cast of 'Advent': the little bird, the snake, bear and dragon, the pirate and of course the threatening little dwarves. • Following people means being able to refer to them after they've left the room: see 'Follow my leader', also by Mr Rees, or the library extension "follower.h" by Andrew Clover. • See the Inform home page for a way round the `Yes` awkwardness. • `orders` and `grammar` are newly introduced into Inform, and so are not much seen in existing games. • For parsing topics of conversation in advanced ways, see the example game 'Encyclopaedia Frobozzica' by Gareth Rees. • To see how much a good set of characters can do for a game, try playing the prologue of 'Christminster'.

13 The light and the dark

The library maintains light by itself, and copes with events like:

a total eclipse of the sun;
fusing all the lights in the house;
your lamp going out;
a dwarf stealing it and running away;
dropping a lit match which you were seeing by;
putting your lamp into an opaque box and shutting the lid;
black smoke filling up the glass jar that the lamp is in;
the dwarf with your lamp running back into your now-dark room.

The point of this list is to demonstrate that light versus darkness is tricky to get right, and that it is best left to the library. Your code needs only to do something like

```
give lamp light;
remove match;
give glass_jar ~transparent;
move dwarf to Dark_Room;
```

and can leave the library to sort out the consequences. As the above suggests, the `light` attribute means that an object is giving off light, or that a room is currently lit, e.g. by being out of doors in day-time. If you simply never want to have darkness, a sneaky way of doing it is to put the line

```
give player light;
```

in `Initialise`. The game works as if the player herself were glowing enough to provide light to see by. So there's never darkness near the player.

The definition of "when there is light" is complicated, involving recursion both up and down. Remember that the parent of the player object may not be a room; it may be, say, a red car whose parent is a room.

Definition. There is light exactly when the parent of the player 'offers light'. An object 'offers light' if:

it itself has the `light` attribute set, **or**
any of its immediate possessions 'have light', **or**
it is see-through and its parent offers light, **or**
it is enterable and its parent offers light;

while an object 'has light' if:

it currently has the `light` attribute set, **or**
it is see-through and one of its immediate possessions has light.

The process of checking this stops as soon as light is discovered. The routines

`OffersLight(object)` and `HasLightSource(object)`

return true or false and might occasionally be useful.

△ So light is cast up and down the tree of objects. In certain contrived circumstances this might be troublesome: perhaps an opaque box, whose outside is fluorescent but whose interior is dark, and which contains an actor who needs not to have other contents of the box in scope... The dilemma could be solved by putting an inner box in the outer one.

• **EXERCISE 33**

How would you code a troll who is afraid of the dark, and needs to be bribed but will only accept a light source... so that the troll will be as happy with a goldfish bowl containing a fluorescent jellyfish as he would be with a lamp?

Each turn, light is reconsidered. The presence or absence of light affects the `Look`, `Search`, `LookUnder` and `Examine` actions, and (since this is a common puzzle) also the `Go` action: you can provide a routine called

```
DarkToDark()
```

and if you do then it will be called when the player goes from one dark room into another dark one (just before the room description for the new dark room, probably "Darkness", is printed). If you want, you can take the opportunity to kill the player off or extract some other forfeit. If you provide no such routine, then the player can move about freely (subject to any rules which apply in the places concerned).

△ When the player is in darkness, the current `location` becomes `thedark`, a special object which acts like a room and has the short name "Darkness". You can change the `initial`, `description` or `short_name` properties for this. For example, your `Initialise` routine might set

```
thedark.short_name = "Creepy, nasty darkness";
```

See §14 for how 'Ruins' makes darkness menacing.

• △ **EXERCISE 34**

Implement a pet moth which escapes if it's ever taken into darkness.

• **REFERENCES**

For a `DarkToDark` routine which discourages wandering about caves in the dark, see 'Advent'.

14 Daemons and the passing of time

Some, such as Sleep and Love, were never human. From this class an individual daemon is allotted to each human being as his 'witness and guardian' through life.

– C. S. Lewis (1898–1963), *The Discarded Image*

A great Daemon... Through him subsist all divination, and the science of sacred things as it relates to sacrifices, and expiations, and disenchantments, and prophecy, and magic... he who is wise in the science of this intercourse is supremely happy...

– Plato (c.427–347 BC), *The Symposium*

– translated by Percy Bysshe Shelley (1792–1822)

In medieval neo-Platonist philosophy, daemons are the intermediaries of God, hovering invisibly over the world and interfering with it. They may be guardian spirits of places or people. So, here, a daemon is a meddling spirit, associated with a particular game object, which gets a chance to interfere once per turn while it is 'active'. The classic example is of the dwarves of 'Advent', who appear in the cave from time to time: a daemon routine attached to the dwarf object moves it about, throws knives at the player and so on. Each object can have a `daemon` routine of its own. This is set going, and stopped again, by calling the (library) routines

```
StartDaemon(object);
StopDaemon(object);
```

Once active, the `daemon` property of the object is called as a routine each turn. Daemons are often started by a game's `Initialise` routine and sometimes remain active throughout. For instance, a lamp-battery daemon might do something every turn, while others may hide for many turns before pouncing: such as the daemon in 'Advent' which waits until the player has found all the treasures.

△ In particular, a daemon doesn't stop running just because the player has moved on to somewhere else. (Indeed, the library never stops a daemon unless told to.) Actually this is very useful, as it means daemons can be used for 'tidying-up operations', or for the consequences of the player's actions to catch up with him.

• **EXERCISE 35**

Many games contain 'wandering monsters', characters who walk around the map. Use a daemon to implement one who wanders as freely as the player, like the gentleman thief in 'Zork'.

• △ **EXERCISE 36**

Use a background daemon to implement a system of weights, so that the player can only carry a certain weight before her strength gives out and she is obliged to drop something. It should allow for feathers to be lighter than lawn-mowers.

A 'timer' (these are traditionally called 'fuses' but the author can stand only so much tradition) can alternatively be attached to an object. Alternatively, because an object can't have both a `timer` and a `daemon` going at the same time. A timer is started with

```
StartTimer(object, time);
```

in which case it will 'go off', alarm clock-style, in the given number of turns. This means that its `time_out` property will be called, once and once only, when the time comes. The timer can be deactivated (so that it will never go off) by calling

```
StopTimer(object);
```

A timer is required to provide a `time_left` property, to hold the amount of time left. (If it doesn't, an error message is printed at run-time.) You can alter `time_left` yourself: a value of 0 means 'will go off at the end of this turn', so setting `time_left` to 0 triggers immediate activation.

△ Normally, you can only have 32 timers or daemons active at the same time as each other (plus any number of inactive ones). But this limit is easily raised: just define the constant `MAX_TIMERS` to some larger value, putting the definition in your code before the `Parser` file is included.

There is yet a third form of timed event. If a room provides an `each_turn` routine, then this will be called at the end of each turn while the player is there; if an object provides `each_turn`, this is called while the object is nearby. For instance, a radio might blare out music whenever it is nearby; a sword might glow whenever monsters are nearby; or a stream running through several forest locations might occasionally float objects by.

'Each turn' is entirely separate from daemons and timers. Although an object can't have both a timer and a daemon at the same time, it can have an `each_turn` at the same time, and this is quite useful, especially to run creatures. An ogre with limited patience can therefore have an `each_turn` routine which worries the player ("The ogre stamps his feet angrily!", etc.) while also having a timer set to go off when his patience runs out.

△ 'Nearby' actually means 'in scope', a term which will be properly explained later. The idea is based on line of sight, which works well in most cases.

△△ But it does mean that the radio will be inaudible when shut up inside most containers – which is arguably fair enough – yet audible when shut up inside transparent, say glass, ones. You can always change the scope rules using an `InScope` routine to get around this. In case you want to tell whether scope is being worked out for ordinary parsing reasons or instead for `each_turn` processing, look at the `scope_reason` variable (see §24). Powerful effects are available this way – you could put the radio in scope within all nearby rooms so as to allow sound to travel. Or you could make a thief audible throughout the maze he is wandering around in, as in 'Zork I'.

• EXERCISE 37

(Why the 'Ruins' are claustrophobic.) Make "the sound of scuttling claws" approach the player in darkness and, after 4 consecutive turns in darkness, kill him.

• △ EXERCISE 38

A little harder: implement the scuttling claws in a single object definition, with no associated code anywhere else in the program (not even a line in `Initialise`) and without running its daemon all the time.

The library also has the (limited) ability to keep track of time of day as the game goes on. The current time is held in the variable `the_time` and runs on a 24-hour clock: this variable holds minutes since midnight, so it has values between 0 and 1439. The time can be set by

SetTime(60×⟨hours⟩+⟨minutes⟩, ⟨rate⟩);

The `rate` controls how rapidly time is moving: a `rate` of 0 means it is standing still (that is, that the library doesn't change it: your routines still can). A positive `rate` means that that many minutes pass between each turn, while a negative `rate` means that many turns pass between each minute. (It's usual for a timed game to start off the clock by calling `SetTime` in its `Initialise` routine.) The time only (usually) appears on the game's status line if you set

```
Statusline time;
```

as a directive somewhere in your code.

• EXERCISE 39

How could you make your game take notice of the time passing midnight, so that the day of the week could be nudged on?

• △ EXERCISE 40

(Cf. Sam Hulick's vampire game, 'Knight of Ages'.) Make the lighting throughout the game change at sunrise and sunset.

△ Exactly what happens at the end of each turn is:

1. The turns counter is incremented.
2. The 24-clock is moved on.
3. Daemons and timers are run (in no guaranteed order).
4. `each_turn` takes place for the current room, and then for everything nearby (that is, in scope).
5. The game's global `TimePasses` routine is called.
6. Light is re-considered (it may have changed as a result of events since this time last turn).

The sequence is abandoned if at any stage the player dies or wins.

• △ EXERCISE 41

Suppose the player is magically suspended in mid-air, but that anything let go of will fall out of sight. The natural way to code this is to use a daemon which gets rid of anything it finds on the floor (this is better than just trapping `Drop` actions because objects might end up on the floor in many different ways). Why is using `each_turn` better?

• EXERCISE 42

How would a game work if it involved a month-long archaeological dig, where anything from days to minutes pass between successive game turns?

• REFERENCES

Daemons abound in most games. 'Advent' uses them to run down the lamp batteries, make the bear follow you, animate the dwarves and the pirate and watch for the treasure all being found. See also the flying tortoise from 'Balances' and the chiggers from 'Adventureland'. For more ingenious uses of `daemon`, see the helium balloon, the matchbook and (particularly cunning) the pair of white gloves in 'Toyshop'. • Classic timers include the burning match and the hand grenade from 'Toyshop', the endgame timer from 'Advent' and the 'Balances' cyclops (also employing `each_turn`). • 'Adventureland' makes good use of `each_turn`: see the golden fish, the mud, the dragon and the bees. • The library extension 'timewait.h' by Andrew Clover thoroughly implements time of day, allowing the player to "wait until quarter past three".

15 Starting, moving, changing and killing the player

There are only three events in a man's life; birth, life and death; he is not conscious of being born, he dies in pain and he forgets to live.

– Jean de la Bruyère (1645–1696)

Life's but a walking shadow, a poor player
That struts and frets his hour upon the stage
And then is heard no more; it is a tale
Told by an idiot, full of sound and fury,
Signifying nothing.

– William Shakespeare (1564–1616), *Macbeth V. v*

The only compulsory task for a game's `Initialise` routine is to set the `location` variable to the place where the player should begin. This is usually a room (and is permitted to be one that's in darkness) but could instead be an object inside a room, such as a chair or a bed. If you would like to give the player some items to begin with, `Initialise` should also `move` them to `player`.

Games with a long opening sequence might want to start by offering the player a chance to restore a saved game at once. They can do so by writing the following in their `Initialise` routines:

```
print "Would you like to restore a game?  >";
if (YesOrNo()==1) <Restore>;
```

(If you want to make the status line invisible during an opening sequence, see §29.) `Initialise` normally returns 0 or 1 (it doesn't matter which), but if it returns 2 then no game banner will be printed at once. (This is for games which, like 'Sorcerer', delay their banners until after the prologue.) 'Ruins', however, opens in classical fashion:

```
[ Initialise;
  TitlePage();
  location = Forest;
  move food_ration to player;
  move sodium_lamp to player;
  move dictionary to player;
  thedark.description = "The darkness of ages presses in on you, and you \
      feel claustrophobic.";
 "^^^^^Days of searching, days of thirsty hacking through the briars of \
  the forest, but at last your patience was rewarded. A discovery!^";
];
```

(The `TitlePage` routine will be an exercise in §29: 'Ruins' is really too small a game to warrant one, but never mind.) The `location` variable needs some explanation. It holds either the current room, if there's light to see by, or the special value `thedark` (the "Darkness" object) if there isn't. In the latter case (but only in the latter case) the actual current room is held in the variable `real_location`, should you need to know it. Neither of these is necessarily the same as the parent of the `player` object. For instance, if the player sits in a jeep parked in a dark garage, then `location` is `thedark`, `real_location` is `Garage` and `parent(player)` is `jeep`.

Because of this, one shouldn't simply `move` the player object by hand. Instead, to move the player about (for teleportation of some kind), use the routine `PlayerTo(place);` (which automatically takes care of printing the new room's description if there's enough light there to see by).

△ `PlayerTo` can also be used to move the player to a `place` inside a room (e.g., a cage, or a traction engine).

△ Calling `PlayerTo(place, 1);` moves the player but prints nothing (in particular, prints no room description).

△ Calling `PlayerTo(place, 2);` will `Look` as if the player had arrived in the room by walking in as usual, so only a short description appears if the room is one that has been seen before.

△ In a process called 'scoring arrival', a room which the player has entered for the first time is given the `visited` attribute. If it was listed as `scored`, points are awarded. (See §16.)

△△ When a `Look` action takes place, or a call to `PlayerTo(place,1)`, the library 'notes arrival' as well as 'scores arrival'. 'Noting arrival' consists of checking to see if the room has changed since last time (darkness counting as a different room for this purpose). If so, the following happens:

1. If the new location has an `initial` property, this is printed if it's a string, or run if it's a routine.
2. The entry point `NewRoom` is called (if it exists).
3. Any 'floating objects', such as drifting mist, which are `found_in` many places at once, are moved into the room.

The player's whole persona can easily be changed, because the player object can itself have an `orders` routine, just as the object for any non-player character can. To replace the `orders` routine for the standard `player` object,

```
player.orders = #r$MyNewRule;
```

where `MyNewRule` is a new `orders` rule. Note that this is applied to every action or order issued by the player. The variable `actor` holds the person being told to do something, which may well be the player himself, and the variables `action`, `noun` and `second` are set up as usual. For instance, if a cannon goes off right next to the player, a period of partial deafness might ensue:

```
[ MyNewRule;
  if (actor~=player) rfalse;
  Listen: "Your hearing is still weak from all that cannon-fire.";
];
```

The `if` statement needs to be there to prevent commands like "helena, listen" from being ruled out – after all, the player can still speak.

• △ EXERCISE 43
Why not achieve the same effect by giving the player a `react_before` rule instead?

• EXERCISE 44
(Cf. 'Curses'.) Write an `orders` routine for the player so that wearing the gas mask will prevent him from talking.

△ In fact a much more powerful trick is available: the `player` can actually become a different character in the game, allowing the real player at the keyboard to act through someone else. Calling `ChangePlayer(obj)` will transform the player to `obj`. There's no need for `obj` to have names like "me" or "myself"; the parser understands these words automatically to refer to the currently-inhabited `player` object. However, it must provide a `number` property (which the library will use for workspace). The maximum number of items the player can carry as that object will be its `capacity`. Finally, since `ChangePlayer` prints nothing, you may want to conclude with a `<<Look>>;`

`ChangePlayer` has many possible applications. The player who tampers with Dr Frankenstein's brain transference machine may suddenly become the Monster strapped to the table. A player who drinks too much wine could become a 'drunk player object' to whom many different rules apply. The "snavig" spell of 'Spellbreaker', which transforms the player to an animal like the one cast upon, could be implemented thus. More ambitiously, a game could have a stock of half a dozen main characters, and the focus of play can switch between them. A player might have a team of four adventurers to explore a dungeon, and be able to switch the one being controlled by typing the name. In this case, an `AfterLife` routine – see below – may be needed to switch the focus back to a still-living member of the team after one has met a sticky end.

△ Calling `ChangePlayer(object,1);` will do the same but make the game print "(as Whoever)" during room descriptions.

△△ When the person to be changed into has an `orders` routine, things start to get complicated. It may be useful to arrange such a routine as follows:

```
orders
[; if (player==self)
    {   ! I am the player object...
        if (actor==self)
        {   ! ...giving myself an order, i.e., trying an action.
        }
        else
        {   ! ...giving someone else, the "actor", an order.
        }
    }
    else
    {   ! The player is the "actor" and is giving me an order.
    }
],
```

• △ **EXERCISE 45**

In Central American legend, a sorceror can transform himself into a *nagual*, a familiar such as a spider-monkey; indeed, each individual has an animal self or *wayhel*, living in a volcanic land over which the king, as a jaguar, rules. Turn the player into his *wayhel*.

• △△ **EXERCISE 46**

Write an `orders` routine for a Giant with a conscience, who will refuse to attack a mouse, but so that a player who becomes the Giant can be as cruel as he likes.

The player is still alive for as long as the variable `deadflag` is zero. When set to 1, the player dies; when set to 2, the player wins; and all higher values are taken as more exotic forms of death. Now Inform does not know what to call these exotica: so if they should arise, it calls the `DeathMessage` routine, which is expected to look at `deadflag` and can then print something like "You have changed".

Many games allow reincarnation (or, as David M. Baggett points out, in fact resurrection). You too can allow this, by providing an `AfterLife`. This routine gets the chance to do as it pleases before any "You are dead" type message appears, including resetting `deadflag` back to 0 – which causes the game to proceed in the normal way, rather than end. `AfterLife` routines can be tricky to write, though, because the game has to be set to a state which convincingly reflects what has happened.

• **REFERENCES**

The magic words "xyzzy" and "plugh" in 'Advent' make use of `PlayerTo`. • 'Advent' has an amusing `AfterLife` routine: for instance, try collapsing the bridge by leading the bear across, then returning to the scene after resurrection. 'Balances' has one which only slightly penalises death.

16 Miscellaneous constants and scoring

For when the One Great Scorer comes
To write against your name,
He marks – not that you won or lost –
But how you played the game.

– Grantland Rice (1880–1954), *Alumnus Football*

Some game rules can be altered by defining 'constants' at the start of the program. Two constants you *must* provide (and before including any of the library files) are the strings `Story` and `Headline`:

```
Constant Story "ZORK II";
Constant Headline "^An Interactive Plagiarism^\
             Copyright (c) 1995 by Ivan O. Ideas.^";
```

All the rest are optional, but should be defined before `Verblib` is included if they're to take effect.

The library won't allow the player to carry an indefinite number of objects: the limit allowed is the constant `MAX_CARRIED`, which you may define if you wish. If you don't define it, it's 100, which nearly removes the rule. In fact you can change this during play, since it is actually the `capacity` of the `player` which is consulted; the only use of `MAX_CARRIED` is to set this up to an initial value.

If you define `SACK_OBJECT` to be some container, then the player will automatically put old, least-used objects away in it as the game progresses, provided it is being carried. This is a feature which endears the designer greatly to players. For instance, the following code appears (in between inclusion of `Parser` and `Verblib`) in 'Toyshop':

```
Object satchel "satchel"
  with description "Big and with a smile painted on it.",
       name "satchel", article "your",
       when_closed "Your satchel lies on the floor.",
       when_open "Your satchel lies open on the floor.",
  has  container open openable;
Constant SACK_OBJECT satchel;
```

'Ruins' isn't going to provide this feature, because there are few portable objects and those there are would be incongruous if described as being in a rucksack.

Another constant is `AMUSING_PROVIDED`. If you define this, the library knows to put an "amusing" option on the menu after the game is won. It will then call `Amusing` from your code when needed. You can use this to roll closing credits, or tell the player various strange things about the game, now that there's no surprise left to spoil.

The other constants you are allowed to define help the score routines along. There are two scoring systems provided by the library, side by side: you can use both or neither.

You can always do what you like to the `score` variable in any case, though the "fullscore" verb might then not fully account for what's happened. One scores points for getting certain items or reaching certain places; the other for completing certain actions. These constants are:

`MAX_SCORE`	the maximum game score (by default 0);
`NUMBER_TASKS`	number of individual "tasks" to perform (1);
`OBJECT_SCORE`	bonus for first picking up a `scored` object (4);
`ROOM_SCORE`	bonus for first entering a `scored` room (5)

and then the individual tasks have scores, as follows:

```
Array task_scores -> t1 t2 ... tn;
```

As this is a byte array, the task scores must be between 0 and 255. Within your code, when a player achieves something, call `Achieved(task)` to mark that the task has been completed. It will only award points if this task has not been completed before. There do not have to be any "tasks": there's no need to use the scoring system provided. Tasks (and the verb "full" for full score) will only work at all if you define the constant `TASKS_PROVIDED`. The entry point `PrintTaskName` prints the name of a game task (but, of course, is only ever called in a game with `TASKS_PROVIDED` defined). For instance, ('Toyshop' again)

```
[ PrintTaskName ach;
  if (ach==0) "eating a sweet";
  if (ach==1) "driving the car";
  if (ach==2) "shutting out the draught";
  if (ach==3) "building a tower of four";
  if (ach==4) "seeing which way the mantelpiece leans";
];
```

Another entry point, called `PrintRank`, gets the chance to print something additional to the score (traditionally, though not necessarily, rankings). For instance, we bid farewell to the 'Ruins' with the following:

```
[ PrintRank;
  print ", earning you the rank of ";
  if (score >= 50) "Master Archaeologist.";
  if (score >= 30) "Archaeologist.";
  if (score >= 20) "Acquisitor.";
  if (score >= 10) "Investigator.";
  "humble rainforest Tourist.";
];
```

Normally, an Inform game will print messages like

> [Your score has gone up by three points.]

when the score changes (by whatever means). The player can turn this on and off with the "notify" verb; by default it is on. (You can alter the flag `notify_mode` yourself to control this.)

The verbs "objects" and "places" are usually provided, so the player can get a list of all handled objects (and where they now are), and all places visited. If you don't want these to be present, define the constant `NO_PLACES` before inclusion of the library.

• △ EXERCISE 47

Suppose one single room object is used internally for the 64 squares of a gigantic chessboard, each of which is a different location to the player. Then "places" is likely to result in only the last-visited square being listed. Fix this.

• REFERENCES

'Advent' contains ranks and an `Amusing` reward (but doesn't use either of these scoring systems, instead working by hand). • 'Balances' uses `scored` objects (for its cubes). • 'Toyshop' has tasks, as above. • 'Adventureland' uses its `TimePasses` entry point to recalculate the score every turn (and watch for victory).

17 Extending and redefining the Library

> A circulating library in a town is as an ever-green tree of diabolical knowledge! It blossoms through the year!
>
> – R. B. Sheridan (1751–1816), *The Rivals*

Most large games will need to enrich the 'model world': for instance, by creating a new concept such as "magic amulets". The game might contain a dozen of these, each with the power to cast a different spell. So it will need routines which can tell whether or not a given object is an amulet, and what to do when the spell is cast.

The former needs a new attribute. You can create this with the directive

```
Attribute is_amulet;
```

at the start of the program.

△△ In Standard games few spare attributes are available because the library takes most of them. To get around this limit there's a convenient dodge. It sometimes happens that an attribute is only meaningful for a particular kind of object: for instance, "spell has been read" might only be meaningful for a "scroll". With care, therefore, one may re-use the same attribute to have different meanings for different kinds of object. The syntax to declare that an attribute is being reused is

```
Attribute <new> alias <old>;
```

Thereafter Inform will treat the new and old attribute names as referring to the same attribute: it's up to the programmer to make sure this does not lead to inconsistencies. (The library already indulges in a certain amount of this chicanery.)

You can also define your own properties.

```
Property door_to;
Property article "a";
Property amulet_spell NULL;
```

are all examples of the `Property` directive, the first two from the library files. In the case of `article`, we are saying that the value `"a"` should be the default value for any object which doesn't declare an `article`.

△ You can also change the default value during play. For instance, objects with no explicit `cant_go` property of their own normally have the value "You can't go that way." You might change this with

```
ChangeDefault(cant_go, "You're a Master Adventurer now, and still \
                        you walk into walls!");
```

later on in the game. Explicitly given values of `cant_go` are unaffected by this.

△△ Properties can also `alias` and can moreover be declared as `long` or `additive`. For `additive` see §5 on classes; properties which might start as small numbers (less than 256) and be changed into large ones in play, ought to be declared as `long` in Standard games (for Advanced games there's no need).

It's easy enough to write a class definition for amulets, with a `before` rule for `Rub` so that when an amulet is rubbed, its spell is cast. Suppose we want the `amulet_spell` property to be able to hold either a string (which is to be printed out) or a routine (which is to be run). The following does just that:

```
PrintOrRun(object, property, flag);
```

(The library makes extensive use of this function.) If the (optional) flag is given and non-zero, a new-line is printed after a string (but only if it was a string). For example, given `amulet_spell` values such as:

```
amulet_spell "The spell fizzles out with a dull phut! sound.",
amulet_spell
[;  if (location~=thedark) "Nothing happens.";
    give real_location light; "There is a burst of magical light!";
],
```

then `PrintOrRun(object,amulet_spell,1)` is the right way to cast the spell.

△△ Since internally routines and strings are stored as numbers (as their packed addresses, in fact) it can be useful to find out what a property value represents. For this purpose:

$$\texttt{ZRegion(addr)} = \begin{cases} 1 & \text{if } \texttt{addr} \text{ is a valid object number} \\ 2 & \text{if a routine address} \\ 3 & \text{if a string address} \\ 0 & \text{otherwise.} \end{cases}$$

Inform guarantees that object numbers, routine addresses and string addresses are all different (i.e., that no number can represent two of these at once): *but* dictionary or array addresses may coincide with packed addresses of strings or routines.

An elaborate library extension will end up defining several new properties and attributes, some grammar, actions and verb definitions. These may neatly be packaged up into an `Include` file and placed with the other library files.

△△ If the file contains the directive `System_file;` then it will even be possible for games to `Replace` routines from it (see below).

More often, one would like a smaller-scale change to the Library: to alter one of its habitual messages, such as "Nothing is on sale." printed in response to requests to buy something (unless such requests have been trapped at the `before` stage). The message-changing feature is provided in an unusual way. To make use of it, you must provide a special object called `LibraryMessages`, which must be defined *between* the inclusion of "Parser" and of "VerbLib". Here is an example, changing two standard messages:

```
Object LibraryMessages "lm"
  with before
       [;  Jump: "You jump and float uselessly for a while in \
                   the zero gravity here on Space Station Alpha.";
           SwitchOn:
                 if (lm_n==3)
                 {   print "You power up ", (the) lm_o, "."; }
       ];
```

The object never appears in the game, of course (so it doesn't matter what its short name is), and it exists solely to have a `before` routine. As usual, returning false (0) causes the library to carry on as normal, while returning true (1) indicates that you've printed the message.

The `Jump` action only ever prints one message, but more elaborate actions such as `SwitchOn` have several (the extreme case is `Take`, with 13). `lm_n` holds the message number, which counts upwards from 1 essentially in order of use. The messages and numbers are given in §39. New message numbers may possibly be added in future, but old ones will not be renumbered.

An especially useful library message to change is the prompt, usually set to `"^>"` (new-line followed by `>`). This is printed under the fake action `Prompt` (which exists only for use by `LibraryMessages`). Thus you can make the game's prompt context-sensitive, or remove the "skipped line on screen each turn" convention.

△ This prompt is only used in ordinary game play, and not at such keyboard inputs as yes/no questions or the RESTART/RESTORE/QUIT game over choice.

• EXERCISE 48

Infocom's game 'The Witness' has the prompt "What should you, the detective, do next?" on turn one and "What next?" subsequently. Implement this.

An amusing way to see the system in action is to put

```
Object LibraryMessages "lm"
  with before
       [;  print "[", sw__var, ", ", lm_n, "] ";
       ];
```

into your game (arcane note: `sw__var`, the "switch variable", in this case holds the action number). Another amusing effect is to simply write `rtrue;` for the `before` routine, which results in an alarmingly silent game – blindfold Adventure, perhaps.

△△ Note that `LibraryMessages` can be used as a sneaky way to add extra rules onto the back of actions, since there's nothing to stop you doing real processing in a call to it; or, more happily, to make messages more sensistive to game context, so that "Nothing is on sale" might become "That's not one of the goods on sale" inside a shopping mall.

• △△ EXERCISE 49
Write an Inform game in Occitan (a dialect of medieval French spoken in Provence).

The Library is itself written in high-level Inform (with a few input-output routines dipping into assembly language) which is compiled at the same time as the rest of the game. The source files are perfectly accessible and could easily be modified. But this would make it necessary to keep a copy of the library for every game, and to make the changes afresh whenever the library is updated; and it means your modifications are not gathered together in any convenient form.

A somewhat crude method, something of a last resort (though most large games contain two or three uses of this), is to work out which routine is giving trouble and "replace" it. For example, if the directive

```
Replace BurnSub;
```

is placed in your file *before the library files are included*, Inform ignores the definition of `BurnSub` in the library files. You then have to define a routine called `BurnSub` yourself, or Inform will complain that the program refers to a routine which isn't there. It would be normal to copy the definition of `BurnSub` to your own code, and make such alterations as you need.

The favourite routine to replace is `DrawStatusLine`: see §29 for several examples.

△△ Inform even allows you to `Replace` "hardware" functions like `random`, which would normally be translated directly to machine opcodes. Obviously, replacing something like `sibling` with a software routine will impose an appreciable speed penalty and slightly increase object code size. Replacing `random` may however be useful when fixing the random number generator for game-testing purposes.

• REFERENCES
'Balances' contains a section of code (easily extractable to other games) implementing the 'Enchanter' trilogy's magic system by methods like the above. • There are several formal library extension files in existence, mostly small: see the Inform home page on the WWW. • "pluralobj.h" by Andrew Clover makes large-scale use of `LibraryMessages` to ensure that the library always uses words like "those" instead of "that" when talking about objects with names like "a heap of magazines".

Chapter III: Describing and Parsing

Language disguises thought... The tacit conventions on which the understanding of everyday language depends are enormously complicated.

– Ludwig Wittgenstein (1889–1951), *Tractatus*

18 Describing objects and rooms

And we were angry and poor and happy,
And proud of seeing our names in print.

– G. K. Chesterton (1874–1936), *A Song of Defeat*

Talking to the player about the state of the world is much easier than listening to his intentions for it, and Inform's rules for describing places and items are considerably simpler than its rules for understanding what the player types. Despite this, the business of description takes up a fair part of Chapter III since the designer eventually needs to know almost every rule involved, whereas nobody would want to know everything about the parser.

To begin, the simplest description happens when a single object is named by the game, for instance when the statement

```
print (a) brass_lamp;
```

results in "an old brass lamp" being printed. There are four such forms of `print`:

`print (the) obj`	Print the object with its definite article
`print (The) obj`	The same, but capitalised
`print (a) obj`	Print the object with indefinite article
`print (name) obj`	Print the object's short name alone

and these can be freely mixed into lists of things to `print` or `print_ret`, as for example:

```
print_ret "The ogre declines to eat ", (the) noun, ".";
```

• **EXERCISE 50**

(By Gareth Rees.) When referring to `animate` objects, one usually needs to use pronouns such as "his". Define new printing routines so that, say, `print "You throw the book at ", (PronounAcc) obj, "!";` will insert the right accusative pronoun.

△ There is also a special syntax `print object` for printing object names, but *do not use it without good reason*: it doesn't understand some of the features below and is not protected against crashing if you mistakenly try to print the name for an out of range object number.

The indefinite article for an object is held in the property `article` and is assumed to be 'a' if none is declared. That means that if the short name starts with a vowel, you need to set it to 'an'. But `article` offers much more amusement:

> a / platinum bar, an / orange balloon, your / Aunt Jemima,
> some bundles of / reeds, far too many / marbles, The / London Planetarium

If the object is given the attribute `proper` then its name is treated as a proper noun with no indefinite article, so the value of `article` is ignored.

△ The `article` property can also hold a routine to print one, in case "a pregnant mouse" has to change to "some mice".

Definite articles are always "the" (except for `proper` nouns). Thus

> the platinum bar, Aunt Jemima, Elbereth

are all printed by `print (the) ...`; the latter two objects being `proper`.

△ There's usually no need to worry about definite and indefinite articles for room objects, as Inform never has cause to print them.

The short name of an object is normally the text given in double-quotes at the head of its definition. This is very inconvenient to change during play when, for example, "blue liquid" becomes "purple liquid" as a result of a chemical reaction. A more flexible way to specify an object's short name is with the `short_name` property. To print the name of such an object, Inform does the following:

1. If the `short_name` is a string, it's printed and that's all.
2. If it is a routine, then it is called. If it returns true, that's all.
3. The text given in the header of the object definition is printed.

For example, the dye might be given:

```
short_name
[;   switch(self.number)
     {   1: print "blue ";
         2: print "purple ";
         3: print "horrid sludge"; rtrue;
     }
],
```

with `"liquid"` as the short name in its header. According to whether its `number` property is 1, 2 or 3, the printed result is "blue liquid", "purple liquid" or "horrid sludge".

△ Alternatively, define the dye with `short_name "blue liquid"` and then simply execute `dye.short_name = "purple liquid";` when the time comes.

△ Note that rooms can also be given a `short_name` routine, which might be useful to code, say, a grid of four hundred similar locations called "Area 1" up to "Area 400".

For many objects the indefinite article and short name will most often be seen in inventory lists, such as

```
>i
You are carrying:
  a leaf of mint
  a peculiar book
  your satchel (which is open)
    a green cube
```

Some objects, though, ought to have fuller entries in an inventory: a wine bottle should say how much wine is left, for instance. The `invent` property is designed for this. The simplest way to use `invent` is as a string. For instance, declaring a peculiar book with

```
invent "that harmless old book of Geoffrey's",
```

will make this the inventory line for the book. In the light of events, it could later be changed to

```
geoffreys_book.invent = "that lethal old book of Geoffrey's";
```

△ Note that this string becomes the whole inventory entry: if the object were an open container, its contents wouldn't be listed, which might be unfortunate. In such circumstances it's better to write an `invent` routine, and that's also the way to append text like "(half-empty)".

△ Each line of an inventory is produced in two stages. First, the basic line:

1a. The global variable `inventory_stage` is set to 1.
1b. The `invent` routine is called (if there is one). If it returns true, stop here.
1c. The object's indefinite article and short-name are printed.

Second, little informative messages like "(which is open)" are printed, and inventories are given for the contents of open containers:

2a. The global variable `inventory_stage` is set to 2.
2b. The `invent` routine is called (if there is one). If it returns true, stop here.
2c. A message such as "(closed, empty and providing light)" is printed, as appropriate.
2d. If it is an `open container`, its contents are inventoried.

After each line is printed, linking text such as a new-line or a comma is printed, according to the current style. For example, here is the `invent` routine used by the matchbook in 'Toyshop':

```
invent
[ i; if (inventory_stage==2)
     {   i=self.number;
         if (i==0) print " (empty)";
         if (i==1) print " (1 match left)";
         if (i>1)  print " (",i," matches left)";
     }
],
```

• △△ EXERCISE 51

Suppose you want to change the whole inventory line for an ornate box but you can't use an `invent` string, or return true from stage 1, because you still want stage 2d to happen properly (so that its contents will be listed). How can you achieve this?

The largest and most complicated messages Inform ever prints on its own initiative are room descriptions, printed when the `Look` action is carried out (so that the statement `<Look>;` triggers a room description). This is basically simple: the room's short name is printed (usually in bold-face) on a line of its own, then its `description` followed by a list of the objects lying about in it which aren't `concealed` or `scenery`.

Chapter II mentions many different properties – `initial`, `when_on`, `when_off` and so on – which contain descriptions of an object inside a room; some apply to doors, others to switchable objects and so on. All of them can be routines to print something, instead of strings to print. The full rules are given below.

This whole system can be bypassed using the `describe` property. If an object gives a `describe` routine then this is called first: if it returns true, then Inform assumes that the object has already been described, so it doesn't describe it later on. For example,

```
describe
[;  "^The platinum pyramid catches the light beautifully.";
];
```

means that even when the pyramid has been `moved` (i.e. held by the player at some stage) it will always have its own line of room description.

△ Note the initial ^ (new-line) character. Inform doesn't print a skipped line itself before calling `describe` because it doesn't know yet whether the routine will want to say anything. A `describe` routine which prints nothing and returns true makes an object invisible, as if it were `concealed`.

△△ The `Look` action does three things: it 'notes arrival', prints the room description then 'scores arrival'. Only the printing rules are given here (see §15 for the others), but they're given in full. In what follows, the word 'location' means the room object if there's light to see by, and the special "Darkness" object otherwise. First the top line:

1a. A new-line is printed. The location's short name is printed (in bold-face, if the host machine can do so).
1b. If the player is on a `supporter`, then " (on ⟨something⟩)" is printed; if inside anything else, then " (in ⟨something⟩)".
1c. " (as ⟨something⟩)" is printed if this was requested by the game's most recent call to `ChangePlayer`, if any.
1d. A new-line is printed.

Now the 'long description'. This step is skipped if the player has just moved of his own will into a location already visited, unless the game is in "verbose" mode.

2. If the location has a `describe` property, then run it. If not, look at the location's `description` property: if it's a string, print it; if it's a routine, run it.

All rooms must provide *either* a `describe` property *or* a `description` of themselves. Now for items nearby:

3a. List any objects on the floor.

3b. If the player is in or on something, list the other objects in that.

Inform has now finished, but your game gets a chance to add a postscript:

4. Call the entry point `LookRoutine`.

△ The `visited` attribute is only given to a room after its description has been printed for the first time (this happens during 'scoring arrival'). This is convenient for making the description different after the first time.

△ 'Listing objects' (as in 3a and 3b) is a complicated business. Some objects are given a line or paragraph to themselves; some are lumped together in a list at the end. The following objects are not mentioned at all: the player, what the player is in or on (if anything) and anything which is `scenery` or `concealed`. The remaining objects are looked through (eldest first) as follows:

1. If the object has a `describe` routine, run it. If it returns true, stop here and don't mention the object at all.
2. Work out the "description property" for the object:
   a. For a `container`, this is `when_open` or `when_closed`;
   b. Otherwise, for a `switchable` object this is `when_on` or `when_off`;
   c. Otherwise, for a `door` this is `when_open` or `when_closed`;
   d. Otherwise, it's `initial`.
3. If **either** the object doesn't have this property **or** the object has been held by the player before (i.e., has `moved`) and the property isn't `when_off` or `when_closed` **then** then the object will be listed at the end.
4. Otherwise a new-line is printed and the property is printed (if it's a string) or run (if it's a routine).

△ A `supporter` which is `scenery` won't be mentioned, but anything on top of it which is not `concealed` will be.

△ Objects which have just been pushed into a new room are not listed in that room's description on the turn in question. This is not because of any rule about room descriptions, but because the pushed object is moved into the new room only after the room description is made. This means that when a wheelbarrow is pushed for a long distance, the player does not have to keep reading "You can see a wheelbarrow here." every move, as though that were a surprise.

△ You can use a library routine called `Locale` to perform 'object listing'. See §36 for details: suffice to say here that the process above is equivalent to executing

```
if (Locale(location, "You can see", "You can also see")~=0)
    " here.";
```

`Locale` is useful for describing areas of a room which are sub-divided off, such as the stage of a theatre.

• **EXERCISE 52**

The library implements "superbrief" and "verbose" modes for room description (one always omits long room descriptions, the other never does). How can verbose mode automatically print room descriptions every turn? (Some of the later Infocom games did this.)

• **REFERENCES**
'Balances' often uses `short_name`, especially for the white cubes (whose names change) and lottery tickets (whose numbers are chosen by the player). 'Adventureland' uses `short_name` in simpler ways: see the bear and the bottle, for instance. • The scroll class of 'Balances' uses `invent`. • The library extension "pluralobj.h" by Andrew Clover makes large-scale use of `LibraryMessages` to ensure that the library always uses words like "those" instead of "that" when talking about objects with names like "a heap of magazines". • See the `ScottRoom` class of 'Adventureland' for a radically different way to describe rooms (in pidgin English, like telegraphese).

19 Listing and grouping objects

As some day it may happen that a victim must be found
I've got a little list – I've got a little list
Of society offenders who might well be underground,
And who never would be missed
Who never would be missed!

– W. S. Gilbert (1836–1911), *The Mikado*

The library often needs to reel off a list of objects: when an `Inv` (inventory) action takes place, for instance, or when describing the contents of a container or the duller items in a room. Lists are difficult to print out correctly 'by hand', because there are many cases to get right, especially when taking plurals into account. Fortunately, the library's list-maker is open to the public. The routine to call is:

```
WriteListFrom(object, style);
```

where the list will start from the given object and go along its siblings. Thus, to list all the objects inside `X`, list from `child(X)`. What the list looks like depends on the style, which is a bitmap you can make by adding some of the following constants:

`NEWLINE_BIT`	New-line after each entry
`INDENT_BIT`	Indent each entry according to depth
`FULLINV_BIT`	Full inventory information after entry
`ENGLISH_BIT`	English sentence style, with commas and 'and'
`RECURSE_BIT`	Recurse downwards with usual rules
`ALWAYS_BIT`	Always recurse downwards
`TERSE_BIT`	More terse English style
`PARTINV_BIT`	Only brief inventory information after entry
`DEFART_BIT`	Use the definite article in list
`WORKFLAG_BIT`	At top level (only), only list objects which have the `workflag` attribute

`ISARE_BIT`	Prints " is " or " are " before list
`CONCEAL_BIT`	Misses out `concealed` or `scenery` objects

The best way to use this is to experiment. For example, a 'tall' inventory is produced by:

```
WriteListFrom( child(player),
               FULLINV_BIT + INDENT_BIT + NEWLINE_BIT + RECURSE_BIT );
```

and a 'wide' one by:

```
WriteListFrom( child(player),
               FULLINV_BIT + ENGLISH_BIT + RECURSE_BIT );
```

which produce effects like:

```
>inventory tall
You are carrying:
  a bag (which is open)
    three gold coins
    two silver coins
    a bronze coin
  four featureless white cubes
  a magic burin
  a spell book

>inventory wide
You are carrying a bag (which is open), inside which are three gold
coins, two silver coins and a bronze coin, four featureless white
cubes, a magic burin and a spell book.
```

except that the 'You are carrying' part is not done by the list-maker, and nor is the final full stop in the second example. The `workflag` is an attribute which the library scribbles over from time to time as temporary storage, but you can use it with care. In this case it makes it possible to specify any reasonable list.

△△ `WORKFLAG_BIT` and `CONCEAL_BIT` specify conflicting rules. If they're both given, then what happens is: at the top level, but not below, everything with `workflag` is included; on lower levels, but not at the top, everything without `concealed` or `scenery` is included.

• **EXERCISE 53**

Write a `DoubleInvSub` action routine to produce an inventory like so:

```
You are carrying four featureless white cubes, a magic burin and a
spell book.  In addition, you are wearing a purple cloak and a miner's
helmet.
```

△ Finally, there is a neat way to customise the grouping together of non-identical items in lists, considerably enhancing the presentation of the game. If a collection of game objects – say, all the edible items in the game – have a common non-zero value of the property `list_together`, say 1, they will always appear adjacently in inventories, room descriptions and the like.

Alternatively, instead of being a small number the common value can be a string such as `"foodstuffs"`. If so then lists will cite, e.g.,

three foodstuffs (a scarlet fish, some lemmas and an onion)

in running text, or

three foodstuffs:
  a scarlet fish
  some lemmas
  an onion

in indented lists. This only happens when two or more are gathered together.

Finally, the common value can be a routine, such as:

```
list_together
[; if (inventory_stage==1) print "heaps of food, notably ";
   else print ", which would do you no good";
],
```

Typically this might be part of a class definition from which all the objects in question inherit. A `list_together` routine will be called twice: once, with `inventory_stage` set to 1, as a preamble to the list of items, and once (with 2) to print any postscript required. It is allowed to change `c_style` (the current list style) without needing to restore the old value and may, by returning 1 from stage 1, signal the list-maker not to print a list at all. The simple example above results in

heaps of food, notably a scarlet fish, some lemmas
and an onion, which would do you no good

Such a routine may want to make use of the variables `parser_one` and `parser_two`, which respectively hold the first object in the group and the depth of recursion in the list (this might be needed to keep indentation going properly). Applying `x=NextEntry(x,parser_two);` moves `x` on from `parser_one` to the next item in the group. Another helpful variable is `listing_together`, set up to the first object of a group being listed or to 0 whenever no group is being listed. The following list of 24 items shows some possible effects (see the example game 'List Property'):

You can see a plastic fork, knife and spoon, three hats (a fez, a Panama and a sombrero), the letters X, Y, Z, P, Q and R from a Scrabble set, a defrosting Black Forest gateau, Punch magazine, a recent issue of the Spectator, a die and eight coins (four silver, one bronze and three gold) here.

•△ EXERCISE 54
Implement the Scrabble pieces.

•△△ EXERCISE 55
Implement the three denominations of coin.

•△△ EXERCISE 56
Implement the I Ching in the form of six coins, three gold (goat, deer and chicken), three silver (robin, snake and bison) which can be thrown to reveal gold and silver trigrams.

• **REFERENCES**
A good example of `WriteListFrom` in action is the definition of `CarryingClass` from the example game 'The Thief', by Gareth Rees. This alters the examine description of a character by appending a list of what that person is carrying and wearing. • Denominations of coin are also in evidence in 'Balances' and in §21.

20 How nouns are parsed

> The Naming of Cats is a difficult matter,
> It isn't just one of your holiday games;
> You may think at first I'm as mad as a hatter
> When I tell you, a cat must have THREE DIFFERENT NAMES.
>
> – T. S. Eliot (1888–1965), *The Naming of Cats*

> Bulldust, coolamon, dashiki, fizgig, grungy, jirble, pachinko, poodle-faker, sharny, taghairm
>
> – Catachrestic words from Chambers English Dictionary

Suppose we have a tomato defined with

```
name "fried" "green" "tomato",
```

but which is going to redden later and need to be referred to as "red tomato". It's perfectly straightforward to alter the `name` property of an object, which is a word array of dictionary words. For example,

```
[ Names obj i;
  for (i=0:2*i<obj.#name:i++) print (address) (obj.&name)-->i, "^";
];
```

prints out the list of dictionary words held in `name` for a given object. We can write to this, so we could just set

```
(tomato.&name)-->1 = 'red';
```

but this is not a flexible or elegant solution, and it's time to begin delving into the parser.

△ Note that we can't change the size of the `name` array. To simulate this, we could define the object with `name` set to, say, 30 copies of an 'untypeable word' (see below) such as `'blank.'`.

The Inform parser is designed to be as "open-access" as possible, because a parser cannot ever be general enough for every game without being extremely modifiable. The first thing it does is to read in text from the keyboard and break it up into a stream of words: so the text "wizened man, eat the grey bread" becomes

`wizened` / `man` / `,` / `eat` / `the` / `grey` / `bread`

and these words are numbered from 1. At all times the parser keeps a "word number" marker to keep its place along this line, and this is held in the variable `wn`. The routine `NextWord()` returns the word at the current position of the marker, and moves it forward, i.e. adds 1 to `wn`. For instance, the parser may find itself at word 6 and trying to match "grey bread" as the name of an object. Calling `NextWord()` gives the value `'grey'` and calling it again gives `'bread'`.

Note that if the player had mistyped "grye bread", "grye" being a word which isn't mentioned anywhere in the program or created by the library, `NextWord()` returns 0 for 'misunderstood word'. Writing something like `if (w=='grye') ...` somewhere in the program makes Inform put "grye" into the dictionary automatically.

△ Remember that the game's dictionary only has 9-character resolution. Thus the values of `'polyunsaturate'` and `'polyunsaturated'` are equal. Also, upper case and lower case letters are considered the same; and although a word can contain numerals or symbols, such as `'mn8@home'`, it must begin with a letter.

△△ A dictionary word can even contain spaces, full stops or commas, but if so it is 'untypeable'. For instance, `'in,out'` is an untypeable word because if the player does type it then the parser cuts it into three, never checking the dictionary for the entire word. Thus the constant `'in,out'` can never be anything that `NextWord` returns. This can be useful (as it was in §12).

△ It can also be useful to check for numbers. The library routine `TryNumber(wordnum)` tries to parse the word at `wordnum` as a number (recognising decimal numbers and English ones from "one" to "twenty"), returning -1000 if it fails altogether, or else the number. Values exceeding 10000 are rounded down to 10000.

△△ Sometimes there is no alternative but to actually look at the player's text one character at a time (for instance, to check a 20-digit phone number). The routine `WordAddress(wordnum)` returns a byte array of the characters in the word, and `WordLength(wordnum)` tells you how many characters there are in it. Thus in the above example,

```
thetext = WordAddress(4);
print WordLength(4), " ", (char) thetext->0, (char) thetext->2;
```

prints the text "3 et".

An object can affect how its name is parsed by giving a `parse_name` routine. This is expected to try to match as many words as possible starting from the current position of `wn`, reading them in one at a time using the `NextWord()` routine. Thus it must not stop just because the first word makes sense, but must keep reading and find out how many words in a row make sense. It should return:

0 if the text didn't make any sense at all,

k if k words in a row of the text seem to refer to the object, or
-1 to tell the parser it doesn't want to decide after all.

The word marker `wn` can be left anywhere afterwards. For example:

```
Nearby thing "weird thing"
  with parse_name
       [ i; while (NextWord()=='weird' or 'thing') i++;
            return i;
       ];
```

This definition duplicates (very nearly) the effect of having defined:

```
Nearby thing "weird thing"
  with name "weird" "thing";
```

Which isn't very useful. But the tomato can now be coded up with

```
parse_name
[ i j; if (self has general) j='red'; else j='green';
     while (NextWord()=='tomato' or 'fried' or j) i++;
     return i;
],
```

so that "green" only applies until its `general` attribute has been set, whereupon "red" does.

• **EXERCISE 57**
Rewrite this to insist that the adjectives must come before the noun, which must be present.

• **EXERCISE 58**
Create a musician called Princess who, when kissed, is transformed into "`/?%?/` (the artiste formerly known as Princess)".

• **EXERCISE 59**
(Cf. 'Café Inform'.) Construct a drinks machine capable of serving cola, coffee or tea, using only one object for the buttons and one for the possible drinks.

△ `parse_name` is also used to spot plurals: see §21.

Suppose that an object doesn't have a `parse_name` routine, or that it has but it returned -1. The parser then looks at the `name` words. It recognises any arrangement of some or all of these words as a match (the more words, the better). Thus "fried green tomato" is understood, as are "fried tomato" and "green tomato". On the other hand, so are "fried green" and "green green tomato green fried green". This method is quick and good at understanding a wide variety of sensible inputs, though bad at throwing out foolish ones.

However, you can affect this by using the `ParseNoun` entry point. This is called with one argument, the object in question, and should work exactly as if it were a `parse_name` routine: i.e., returning -1, 0 or the number of words matched as above. Remember that it is called very often and should not be horribly slow. For example, the following duplicates what the parser usually does:

```
[ ParseNoun obj n;
  while (IsAWordIn(NextWord(),obj,name) == 1) n++; return n;
];
[ IsAWordIn w obj prop   k l m;
  k=obj.&prop; l=(obj.#prop)/2;
  for (m=0:m<l:m++)
      if (w==k-->m) rtrue;
  rfalse;
];
```

In this example `IsAWordIn` just checks to see if `w` is one of the entries in the word array `obj.&prop`.

• △ **EXERCISE 60**
Many adventure-game parsers split object names into 'adjectives' and 'nouns', so that only the pattern ⟨0 or more adjectives⟩ ⟨1 or more nouns⟩ is recognised. Implement this.

• △ **EXERCISE 61**
During debugging it sometimes helps to be able to refer to objects by their internal numbers, so that "put object 31 on object 5" would work. Implement this.

• △ **EXERCISE 62**
How could the word "`#`" be made a wild-card, meaning "match any single object"?

• △△ **EXERCISE 63**
And how could "`*`" be a wild-card for "match any collection of objects"?

• △△ **EXERCISE 64**
There is no problem with a calling a container "hole in wall", because the parser will understand "put apple in hole in wall" as "put (apple) in (hole in wall)". But create a fly in amber, so that "put fly in amber in hole in wall" works properly and isn't misinterpreted as "put (fly) in (amber in hole in wall)". (Warning: you may need to know about the `BeforeParsing` entry point (see §22) and the format of the `parse` buffer (see §23).)

• **REFERENCES**
Straightforward `parse_name` examples are the chess-pieces object and the kittens class of 'Alice Through The Looking-Glass'. Lengthier ones are found in 'Balances', especially in the white cubes class.

21 Plural names for duplicated objects

Abiit ad plures.

– Petronius (?–c. 66), *Cena Trimalchionis*

A notorious problem for adventure game parsers is to handle a collection of, say, ten gold coins, allowing the player to use them independently of each other, while gathering them together into groups in descriptions and inventories. This is relatively easy in Inform, and only in really hard cases do you have to provide code. There are two problems to be overcome: firstly, the game has to be able to talk to the player in plurals, and secondly vice versa. First, then, game to player:

```
Class  gold_coin_class
  with name "gold" "coin",
       plural "gold coins";
```

(and similar silver and bronze coin classes here)

```
Object bag "bag"
  with name "bag"
  has  container open openable;

Nearby co1 "gold coin"   class gold_coin_class;
Nearby co2 "gold coin"   class gold_coin_class;
Nearby co3 "gold coin"   class gold_coin_class;
Nearby co4 "silver coin" class silver_coin_class;
Nearby co5 "silver coin" class silver_coin_class;
Nearby co6 "bronze coin" class bronze_coin_class;
```

Now we have a bag of six coins. The player looking inside the bag will get

```
>look inside bag
In the bag are three gold coins, two silver coins and a bronze coin.
```

How does the library know that the three gold coins are the same as each other, but the others different? It doesn't look at the classes but the names. It will only group together things which:

(a) have a `plural` set,
(b) are 'indistinguishable' from each other.

Indistinguishable means they have the same `name` words as each other, possibly in a different order, so that nothing the player can type will separate the two.

△ Actually, the library is cleverer than this. What it groups together depends slightly on the context of the list it's writing out. When it's writing a list which prints out details of which objects are providing light, for instance (like an inventory), it won't group together two objects if one is lit and the other isn't. Similarly for objects with visible possessions or which can be worn.

△△ This all gets even more complicated when the objects have a `parse_name` routine supplied, because then the library can't use the `name` fields to tell them apart. If they have different `parse_name` routines, it decides that they're different. But if they have the same `parse_name` routine, there is no alternative but to ask them. What happens is that

1. A variable called `parser_action` is set to `##TheSame`;
2. Two variables, called `parser_one` and `parser_two` are set to the two objects in question;
3. Their `parse_name` routine is called. If it returns:
   -1 the objects are declared "indistinguishable",
   -2 they are declared different.
4. Otherwise, the usual rules apply and the library looks at the ordinary `name` fields of the objects.

`##TheSame` is a fake action. The implementation of the 'Spellbreaker cubes' in the 'Balances' game is an example of such a routine, so that if the player writes the same name on several of the cubes, they become grouped together. Note that this whole set-up is such that if the author of the `parse_name` routine has never read this paragraph, it doesn't matter and the usual rules take their course.

△△ You may even want to provide a `parse_name` routine just to speed up the process of telling two objects apart – if there were 30 gold coins the parser would be doing a lot of work comparing all their names, but you can make the decision much faster.

Secondly, the player talking to the computer. This goes a little further than just copies of the same object: many games involve collecting a number of similar items, say a set of nine crowns in different colours. Then you'd want the parser to recognise things like:

```
> drop all of the crowns except green
> drop the three other crowns
```

even though the crowns are not identical. The simple way to do this is just to put `"crowns"` in their `name` lists, and this works perfectly well most of the time.

△△ But it isn't ideal, because then the parser will think "take crowns" refers to a single object, and won't realise that the player wants to pick up all the sensibly available crowns. So the complicated (but better) way is to make the `parse_name` routine tell the parser that yes, there was a match, but that it was a plural. The way to do this is to set `parser_action` to `##PluralFound`, another fake action. So, for example:

```
Class  crown_class
  with parse_name
       [ i j;
         for (::)
         {   j=NextWord();
             if (j=='crown' or self.name) i++;
             else
             {   if (j=='crowns')
                 {   parser_action=##PluralFound; i++; }
                 else return i;
             }
         }
       ];
```

This code assumes that the crown objects have just one `name` each, their colours.

• **EXERCISE 65**
Write a 'cherub' class so that if the player tries to call them "cherubs", a message like "I'll let this go by for now, but the plural of cherub is cherubim" appears.

• **REFERENCES**
See the coinage of 'Balances'.

22 How verbs are parsed

Grammar, which can govern even kings.
– Molière (1622–1673), *Les Femmes savantes*

The parser's fundamental method is simple. Given a stream of text like

`saint / peter / , / take / the / keys / from / paul`

it first calls the entry point `BeforeParsing` (in case you want to meddle with the text stream before it gets underway). It then works out who is being addressed, if anyone, by looking for a comma, and trying out the text up to there as a noun (anyone `animate` or anything `talkable` will do): in this case St Peter. This person is called the "actor", since he is going to perform the action, and is usually the player himself (thus, typing "myself, go north" is equivalent to typing "go north"). The next word, in this case `'take'`, is the "verb word". An Inform verb usually has several English verb words attached, which are called synonyms of each other: for instance, the library is set up with

"take" = "get" = "carry" = "hold"

all referring to the same Inform verb.

△ The parser sets up global variables `actor` and `verb_word` while working. (In the example above, their values would be the St Peter object and `'take'`, respectively.)

△△ It isn't quite that simple: names of direction objects are treated as implicit "go" commands, so that "n" is acceptable as an alternative to "go north". There are also "again", "oops" and "undo" to grapple with.

△ Also, a major feature (the `grammar` property for the person being addressed) has been missed out of this description: see the latter half of §12 for details.

Teaching the parser a new synonym is easy. Like all of the directives in this section, the following must appear *after* the inclusion of the library file `Grammar`:

```
Verb "steal" "acquire" "grab" = "take";
```

This creates another three synonyms for "take".

△ One can also prise synonyms apart, as will appear later.

The parser is now up to word 5; i.e., it has "the keys from paul" left to understand. Apart from a list of English verb-words which refer to it, an Inform verb also has a "grammar". This is a list of 1 or more "lines", each a pattern which the rest of the text might match. The parser tries the first, then the second and so on, and accepts the earliest one that matches, without ever considering later ones.

A line is itself a row of "tokens". Typical tokens might mean 'the name of a nearby object', 'the word `from`' or 'somebody's name'. To match a line, the parser must match against each token in sequence. For instance, the line of 3 tokens

⟨a noun⟩ ⟨the word `from`⟩ ⟨a noun⟩

matches the text. Each line has an action attached, which in this case is `Remove`: so the parser has ground up the original text into just four numbers, ending up with

```
actor = st_peter
action = Remove   noun = gold_keys   second = st_paul
```

What happens then is that the St Peter's `orders` routine (if any) is sent the action, and may if it wishes cooperate. If the actor had been the player, then the action would have been processed in the usual way.

△ The action for the line which is currently being worked through is stored in the variable `action_to_be`; or, at earlier stages when the verb hasn't been deciphered yet, it holds the value `NULL`.

The `Verb` directive creates Inform verbs, giving them some English verb words and a grammar. The library's `Grammar` file consists almost exclusively of `Verb` directives: here is an example simplified from one of them.

```
Verb "take" "get" "carry" "hold"
                 * "out"                        -> Exit
                 * multi                        -> Take
                 * multiinside "from" noun      -> Remove
                 * "in" noun                    -> Enter
                 * multiinside "off" noun       -> Remove
                 * "off" held                   -> Disrobe
                 * "inventory"                  -> Inv;
```

Each line of grammar begins with a *, gives a list of tokens as far as -> and then the action which the line produces. The first line can only be matched by something like "get out", the second might be matched by

take the banana
get all the fruit except the apple

and so on. A full list of tokens will be given later: briefly, `"out"` means the literal word "out", `multi` means one or more objects nearby, `noun` means just one and `multiinside` means one or more objects inside the second noun. In this book, grammar tokens are written in the style `noun` to prevent confusion (as there is also a variable called `noun`).

△△ As mentioned above, the parser thinks "take" and "get" are exactly the same. Sometimes this has odd results: "get in bed" is correctly understood as a request to enter the bed, "take in washing" is misunderstood as a request to enter the washing. You might avoid this by using `Extend only` to separate them into different grammars, or you could fix the `Enter` action to see if the variable `verb_word=='take'` or `'get'`.

△ Some verbs are `meta` - they are not really part of the game: for example, "save", "score" and "quit". These are declared using `Verb meta`, as in

```
Verb meta "score"
                *                               -> Score;
```

and any debugging verbs you create would probably work better this way, since meta-verbs are protected from interference by the game and take up no game time.

After the -> in each line is the name of an action. Giving a name in this way is what creates an action, and if you give the name of one which doesn't already exist then you must also write a routine to execute the action, even if it's one which doesn't do very much. The name of the routine is always the name of the action with `Sub` appended. For instance:

```
[ XyzzySub; "Nothing happens."; ];
Verb "xyzzy"    *                               -> Xyzzy;
```

will make a new magic-word verb "xyzzy", which always says "Nothing happens" – always, that is, unless some `before` rule gets there first, as it might do in certain magic places. `Xyzzy` is now an action just as good as all the standard ones: `##Xyzzy` gives its action number, and you can write `before` and `after` rules for it in `Xyzzy:` fields just as you would for, say, `Take`.

The library defines grammars for the 100 or so English verbs most often used by adventure games. However, in practice you very often need to alter these, usually to add extra lines of grammar but sometimes to remove existing ones. For example, consider an array of 676 labelled buttons, any of which could be pushed: it's hardly convenient to define 676 button objects. It would be more sensible to create a grammar line which understands things like

"button j16", "d11", "a5 button"

(it's easy enough to write code for a token to do this), and then to add it to the grammar for the "press" verb. The `Extend` directive is provided for exactly this purpose:

```
Extend "push"   * Button                    -> PushButton;
```

The point of `Extend` is that it is against the spirit of the Library to alter the standard library files – including the grammar table – unless absolutely necessary.

△△ Another method would be to create a single button object with a `parse_name` routine which carefully remembers what it was last called, so that the object always knows which button it represents. See 'Balances' for an example.

Normally, extra lines of grammar are added at the bottom of those already there. This may not be what you want. For instance, "take" has a grammar line

```
                * multi                     -> Take
```

quite early on. So if you want to add a grammar line which diverts "take something-edible" to a different action, like so:

```
                * edible                    -> Eat
```

($\boxed{\texttt{edible}}$ being a token matching anything which has the attribute `edible`) then it's no good adding this at the bottom of the `Take` grammar, because the earlier line will always be matched first. Thus, you really want to insert your line at the top, not the bottom, in this case. The right command is

```
Extend "take" first
                * edible                    -> Eat;
```

You might even want to over-ride the old grammar completely, not just add a line or two. For this, use

```
Extend "push" replace
                * Button                    -> PushButton;
```

and now "push" can be used only in this way. To sum up, `Extend` can take three keywords:

`replace`	completely replace the old grammar with this one;
`first`	insert the new grammar at the top of the old one;
`last`	insert the new grammar at the bottom of the old one;

with `last` being the default (which doesn't need to be said explicitly).

△ In library grammar, some verbs have many synonyms: for instance,

```
"attack" "break" "smash" "hit" "fight" "wreck" "crack"
"destroy" "murder" "kill" "torture" "punch" "thump"
```

are all treated as identical. But you might want to distinguish between murder and lesser crimes. For this, try

```
Extend only "murder" "kill" replace * animate -> Murder;
```

The keyword `only` tells Inform to extract the two verbs "murder" and "kill". These then become a new verb which is initially an identical copy of the old one, but then `replace` tells Inform to throw that away in favour of an entirely new grammar. Similarly,

```
Extend only "get" * "with" "it" -> Sing;
```

makes "get" behave exactly like "take" (as usual) except that it also recognises "with it", so that "get with it" makes the player sing but "take with it" doesn't. Other good pairs to separate might be "cross" and "enter", "drop" and "throw", "give" and "feed", "swim" and "dive", "kiss" and "hug", "cut" and "prune".

△△ Bear in mind that once a pair has been split apart like this, any subsequent extension made to one will not be made to the other.

△△ There are (a few) times when verb definition commands are not enough. For example, in the original 'Advent' (or 'Colossal Cave'), the player could type the name of a not-too-distant place which had previously been visited, and be taken there. There are several ways to code this – say, with 60 rather similar verb definitions, or with a single "travel" verb which has 60 synonyms, whose action routine looks at the parser's `verb_word` variable to see which one was typed, or even by restocking the compass object with new directions in each room – but here's another. The library will call the `UnknownVerb` routine (if you provide one) when the parser can't even get past the first word. This has two options: it can return false, in which case the parser just goes on to complain as it would have done anyway. Otherwise, it can return a verb word which is substituted for what the player actually typed. Here is a foolish example:

```
[ UnknownVerb w;
  if (w=='shazam') { print "Shazam!^"; return 'inventory'; }
  rfalse;
];
```

which responds to the magic word "shazam" by printing `Shazam!` and then, rather disappointingly, taking the player's inventory. But in the example above, it could be used to look for the word `w` through the locations of the game, store the place away in some global variable, and then return `'go'`. The `GoSub` routine could then be fixed to look at this variable.

•△△ EXERCISE 66
Why is it usually a bad idea to print text out in an `UnknownVerb` routine?

△△ If you allow a flexible collection of verbs (say, names of spells or places) then you may want a single 'dummy' verb to stand for whichever is being typed. This may make the parser produce strange questions because it is unable to sensibly print the verb back at the player, but you can fix this using the `PrintVerb` entry point.

•△△ EXERCISE 67
Implement the Crowther and Woods feature of moving from one room to another by typing its name, using a dummy verb.

• △ EXERCISE 68
Implement a lamp which, when rubbed, produces a genie who casts a spell over the player to make him confuse the words "white" and "black".

• REFERENCES
'Advent' makes a string of simple `Verb` definitions; 'Alice Through The Looking-Glass' uses `Extend` a little. • 'Balances' has a large extra grammar and also uses the `UnknownVerb` and `PrintVerb` entry points.

23 Tokens of grammar

The complete list of grammar tokens is as follows:

`"`⟨word⟩`"`	that literal word only
`noun`	any object in scope
`held`	object held by the player
`multi`	one or more objects in scope
`multiheld`	one or more held objects
`multiexcept`	one or more in scope, except the other
`multiinside`	one or more in scope, inside the other
⟨attribute⟩	any object in scope which has the attribute
`creature`	an object in scope which is `animate`
`noun =` ⟨Routine⟩	any object in scope passing the given test
`scope =` ⟨Routine⟩	an object in this definition of scope
`number`	a number only
⟨Routine⟩	any text accepted by the given routine
`special`	*any* single word or number

These tokens are all described in this section except for `scope =` ⟨Routine⟩, which is postponed to the next.

`"`⟨word⟩`"` This matches only the literal word given, normally a preposition such as `"into"`. Whereas most tokens produce a "parameter" (an object or group of objects, or a number), this token doesn't. There can therefore be as many or as few of them on a grammar line as desired.

△ Prepositions like this are unfortunately sometimes called 'adjectives' inside the parser source code, and in Infocom hackers' documents: the usage is traditional but has been avoided in this manual.

`noun` The definition of "in scope" will be given in the next section. Roughly, it means "visible to the player at the moment".

`held` Convenient for two reasons. Firstly, many actions only sensibly apply to things being held (such as `Eat` or `Wear`), and using this token in the grammar you can make sure that the action is never generated by the parser unless the object is being held. That saves on always having to write "You can't eat what you're not holding" code. Secondly, suppose we have grammar

```
Verb "eat"
                * held                          -> Eat;
```

and the player types "eat the banana" while the banana is, say, in plain view on a shelf. It would be petty of the game to refuse on the grounds that the banana is not being held. So the parser will generate a `Take` action for the banana and then, if the `Take` action succeeds, an `Eat` action. Notice that the parser does not just pick up the object, but issues an action in the proper way – so if the banana had rules making it too slippery to pick up, it won't be picked up. This is called "implicit taking".

The `multi-` tokens indicate that a list of one or more objects can go here. The parser works out all the things the player has asked for, sorting out plural nouns and words like "except" by itself, and then generates actions for each one. A single grammar line can only contain one `multi-` token: so "hit everything with everything" can't be parsed (straightforwardly, that is: you can parse *anything* with a little more effort). The reason not all nouns can be multiple is that too helpful a parser makes too easy a game. You probably don't want to allow "unlock the mystery door with all the keys" – you want the player to suffer having to try them one at a time, or else to be thinking.

`multiexcept` Provided to make commands like "put everything in the rucksack" parsable: the "everything" is matched by all of the player's possessions except the rucksack. This stops the parser from generating an action to put the rucksack inside itself.

`multiinside` Similarly, this matches anything inside the other parameter on the line, and is good for parsing commands like "remove everything from the cupboard".

⟨attribute⟩ This allows you to sort out objects according to attributes that they have:

```
Verb "use" "employ" "utilise"
                * edible                   -> Eat
                * clothing                 -> Wear
      ...and so on...
                * enterable                -> Enter;
```

though the library grammar does not contain such an appallingly convenient verb! Since you can define your own attributes, it's easy to make a token matching only your own class of object.

`creature` Same as `animate` (a hangover from older editions of Inform).

`noun = ⟨Routine⟩` The last and most powerful of the "a nearby object satisfying some condition" tokens. When determining whether an object passes this test, the parser sets the variable `noun` to the object in question and calls the routine. If it returns true, the parser accepts the object, and otherwise it rejects it. For example, the following should only apply to animals kept in a cage:

```
[ CagedCreature;
    if (noun in wicker_cage) rtrue; rfalse;
];
Verb "free" "release"
                * noun=CagedCreature        -> FreeAnimal;
```

So that only nouns which pass the `CagedCreature` test are allowed. The `CagedCreature` routine can appear anywhere in the code, though it's tidier to keep it nearby.

`scope = ⟨Routine⟩` An even more powerful token, which means "an object in scope" where scope is redefined specially. See the next section.

`number` Matches any decimal number from 0 upwards (though it rounds off large numbers to 10000), and also matches the numbers "one" to "twenty" written in English. For example:

```
Verb "type"
                * number                    -> TypeNum;
```

causes actions like `Typenum 504` when the player types "type 504". Note that `noun` is set to 504, not to an object.

• **EXERCISE 69**
(A beautiful feature stolen from David M. Baggett's game 'The Legend Lives', which uses it to great effect.) Some games produce footnotes every now and then. Arrange matters so that these are numbered `[1]`, `[2]` and so on in order of appearance, to be read by the player when "footnote 1" is typed.

△ The entry point `ParseNumber` allows you to provide your own number-parsing routine, which opens up many sneaky possibilities – Roman numerals, coordinates like "J4", very long telephone numbers and so on. This takes the form

```
[ ParseNumber buffer length;
  ...returning 0 if no match is made, or the number otherwise...
];
```

and examines the supposed 'number' held at the byte address `buffer`, a row of characters of the given `length`. If you provide a `ParseNumber` routine but return 0 from it, then the parser falls back on its usual number-parsing mechanism to see if that does any better.

△△ Note that `ParseNumber` can't return 0 to mean the number zero. Probably "zero" won't be needed too often, but if it is you can always return some value like 1000 and code the verb in question to understand this as 0. (Sorry. This was a bad design decision made too long ago to change now.)

$\boxed{\langle\text{Routine}\rangle}$ The most flexible token is simply the name of a "general parsing routine". This looks at the word stream using `NextWord` and `wn` (see §20) and should return:

-1 if the text isn't understood,
0 if it's understood but no parameter results,
1 if a number results, or
n if the object n results.

In the case of a number, the actual value should be put into the variable `parsed_number`. On an unsuccessful match (returning -1) it doesn't matter what the final value of `wn` is. On a successful match it should be left pointing to the next thing *after* what the routine understood. Since `NextWord` moves `wn` on by one each time it is called, this happens automatically unless the routine has read too far. For example:

```
[ OnAtorIn w;
  w=NextWord(); if (w=='on' or 'at' or 'in') return 0;
  return -1;
];
```

makes a token which accepts any of the words "on", "at" or "in" as prepositions (not translating into objects or numbers). Similarly,

```
[ Anything w;  while (w~=-1) w=NextWordStopped(); return 0; ];
```

accepts the entire rest of the line (ignoring it). `NextWordStopped` is a form of `NextWord` which returns -1 once the original word stream has run out.

$\boxed{\texttt{special}}$ Now a rather obsolete feature, but not withdrawn just in case it might still be useful. It's generally better to write a $\boxed{\langle\text{Routine}\rangle}$ token.

• EXERCISE 70
Write a token to detect low numbers in French, "un" to "cinq".

• △ EXERCISE 71
Write a token to detect floating-point numbers like "21", "5.4623", "two point oh eight" or "0.01", rounding off to two decimal places.

• △ EXERCISE 72
Write a token to match a phone number, of any length from 1 to 30 digits, possibly broken up with spaces or hyphens (such as "01245 666 737" or "123-4567").

• △△ **EXERCISE 73**
(Adapted from code in Andrew Clover's 'timewait.h' library extension.) Write a token to match any description of a time of day, such as "quarter past five", "12:13 pm", "14:03", "six fifteen" or "seven o'clock".

• △ **EXERCISE 74**
Code a spaceship control panel with five sliding controls, each set to a numerical value, so that the game looks like:

```
>look
Machine Room
There is a control panel here, with five slides, each of which can be
set to a numerical value.
>push slide one to 5
You set slide one to the value 5.
>examine the first slide
Slide one currently stands at 5.
>set four to six
You set slide four to the value 6.
```

△△ General parsing routines sometimes need to get at the raw text originally typed by the player. Usually `WordAddress` and `WordLength` (see §20) are adequate. If not, it's helpful to know that the parser keeps a `string` array called `buffer` holding:

> `buffer->0` = ⟨maximum number of characters which can fit in buffer⟩
> `buffer->1` = ⟨the number n of characters typed⟩
> `buffer->2`...`buffer->`$(n+1)$ = ⟨the text typed⟩

and, in parallel with this, another one called `parse` holding:

> `parse->0` = ⟨maximum number of words which can fit in buffer⟩
> `parse->1` = ⟨the number m of words typed⟩
> `parse->2`... = ⟨a four-byte block for each word, as follows⟩
>   `block-->0` = ⟨the dictionary entry if word is known, 0 otherwise⟩
>   `block->2` = ⟨number of letters in the word⟩
>   `block->3` = ⟨index to first character in the `buffer`⟩

(However, for "Standard" games the format is slightly different: in `buffer` the text begins at byte 1, not at byte 2, and its end is indicated with a zero terminator byte.) Note that the raw text is reduced to lower case automatically, even if within quotation marks. Using these buffers directly is perfectly safe but not recommended unless there's no other way, as it tends to make code rather illegible.

• △△ **EXERCISE 75**
Try to implement the parser's routines `NextWord`, `WordAddress` and `WordLength`.

• △△ **EXERCISE 76**
(Difficult.) Write a general parsing routine accepting any amount of text (including spaces, full stops and commas) between double-quotes as a single token.

• **EXCERCISE 77**

How would you code a general parsing routine which never matches anything?

• △△ **EXERCISE 78**

Why would you code a general parsing routine which never matches anything?

• △ **EXERCISE 79**

An apparent restriction of the parser is that it only allows two parameters (`noun` and `second`). Write a general parsing routine to accept a `third`. (This final exercise with general parsing routines is easier than it looks: see the specification of the `NounDomain` library routine in §36.)

24 Scope and what you can see

> He cannot see beyond his own nose. Even the fingers he outstretches from it to the world are (as I shall suggest) often invisible to him.
>
> – Max Beerbohm (1872–1956), of George Bernard Shaw

> Wherefore are these things hid?
>
> – William Shakespeare (1564–1616), *Twelfth Night*

Time to say what "in scope" means. This definition is one of the most important rules of play, because it decides what the player is allowed to refer to. You can investigate this in practice by compiling any game with the debugging suite of verbs included and typing "scope" in different places: but here are the rules in full. The following are in scope:

the player's immediate possessions;
the 12 compass directions;
if there is light (see §13), the objects in the same room as the player;
if not, any objects in the `thedark` object.

In addition, if an object is in scope then its immediate possessions are in scope, **if** it is 'see-through', which means that:

the object has `supporter`, **or**
the object has `transparent`, **or**
the object is an `open container`.

In addition, if an object is in scope then anything which it "adds to scope" is also in scope.

△ The player's possessions are in scope in a dark room – so the player can still turn his lamp on. On the other hand, a player who puts the lamp on the ground and turns it off then loses the ability to turn it back on again, because it is out of scope. This can be changed; see below.

△ The compass direction objects make sense as objects. The player can always type something like "attack the south wall" and the `before` rule for the room could trap the action `Attack s_obj` to make something unusual happen, if this is desired.

△ The parser applies scope rules to all actors, not just the player. Thus "dwarf, drop sword" will be accepted if the dwarf can see it, even if the player can't.

△ The `concealed` attribute only hides objects from room descriptions, and doesn't remove them from scope. If you want things to be both concealed and unreferrable-to, put them somewhere else! Or give them an uncooperative `parse_name` routine.

△△ Actually, the above definition is not quite right, because the compass directions are not in scope when the player asks for a plural number of things, like "take all the knives"; this makes some of the parser's plural algorithms run faster. Also, for a `multiexcept` token, the other object is not in scope; and for a `multiinside` token, only objects in the other object are in scope. This makes "take everything from the cupboard" work in the natural way.

Two library routines are provided to enable you to see what's in scope and what isn't. The first, `TestScope(obj, actor)`, simply returns true or false according to whether or not `obj` is in scope. The second is `LoopOverScope(routine, actor)` and calls the given routine for each object in scope. In each case the `actor` given is optional; if it's omitted, scope is worked out for the player as usual.

• **EXERCISE 80**
Implement the debugging suite's "scope" verb, which lists all the objects currently in scope.

• **EXERCISE 81**
Write a "megalook" verb, which looks around and examines everything nearby.

The rest of this section is about how to change the scope rules. As usual with Inform, you can change them globally, but it's more efficient and safer to work locally. To take a typical example: how do we allow the player to ask questions like the traditional "what is a grue"? The "grue" part ought to be parsed as if it were a noun, so that we could distinguish between, say, a "garden grue" and a "wild grue". So it isn't good enough to look only at a single word. Here is one solution:

```
Object questions "qs";
[ QuerySub; print_ret (string) noun.description;
];
[ Topic i;
  switch(scope_stage)
  {   1: rfalse;
      2: objectloop (i in questions) PlaceInScope(i); rtrue;
      3: "At the moment, even the simplest questions confuse you.";
  }
];
```

where the actual questions at any time are the current children of the `questions` object, like so:

```
Object q1 "long count" questions
```

```
  with name "long" "count",
       description "The Long Count is the great Mayan cycle of time, \
           which began in 3114 BC and will finish with the world's end \
           in 2012 AD.";
```

and we also have a grammar line:

```
Verb "what"
                * "is"  scope=Topic              -> Query
                * "was" scope=Topic              -> Query;
```

Note that the questions and q1 objects are out of the game for every other purpose. The name "qs" doesn't matter, as it will never appear; the individual questions are named so that the parser might be able to say "Which do you mean, the long count or the short count?" if the player asked "what is the count".

When the parser reaches `scope=Topic`, it calls the Topic routine with the variable scope_stage set to 1. The routine should return 1 (true) if it is prepared to allow multiple objects to be accepted here, and 0 (false) otherwise: as we don't want "what is everything" to list all the questions and answers in the game, we return false.

A little later on in its machinations, the parser again calls Topic with scope_stage now set to 2. Topic is now obliged to tell the parser which objects are to be in scope. It can call two parser routines to do this.

```
ScopeWithin(object)
```

puts everything inside the object into scope, though not the object itself;

```
PlaceInScope(object)
```

puts just a single object into scope. It is perfectly legal to declare something in scope that "would have been in scope anyway": or even something which is in a different room altogether from the actor concerned, say at the other end of a telephone line. Our scope routine Topic should then return

0 (false) to carry on with the usual scope rules, so that everything that would usually be in scope still is, or
1 (true) to tell the parser not to put any more objects into scope.

So at scope_stage 2 it is quite permissible to do nothing but return false, whereupon the usual rules apply. Topic returns true because it wants only question topics to be in scope, not question topics together with the usual miscellany near the player.

This is enough to deal with "what is the long count". If on the other hand the player typed "what is the lgon cnout", the error message which the parser would usually produce ("You can't see any such thing") would be unsatisfactory. So if parsing failed at this token, then Topic is called at scope_stage 3 to print out a suitable error message. It must provide one.

△ Note that ScopeWithin(object) extends the scope down through its possessions according to the usual rules, i.e., depending on their transparency, whether they're containers and so on. The definition of Topic above shows how to put just the direct possessions into scope.

• **EXERCISE 82**
Write a token which puts everything in scope, so that you could have a debugging "purloin" verb which could take anything, regardless of where it was and the rules applying to it.

Changing the global definition of scope should be done cautiously (there may be unanticipated side effects); bear in mind that scope decisions need to be taken often – every time an object token is parsed, so perhaps five to ten times in every game turn – and hence moderately quickly. The global definition can be tampered with by providing the entry point

```
InScope(actor)
```

where the `actor` is usually the player, but not always. If the routine decides that a particular object should be in scope for the actor, it should execute `InScope` and `ScopeWithin` just as above, and return true or false, as if it were at `scope_stage` 2. Thus, it is vital to return false in circumstances when you don't want to intervene.

△ The token `scope=⟨Routine⟩` takes precedence over `InScope`, which will only be reached if the routine returns false to signify 'carry on'.

△△ There are seven reasons why `InScope` might be being called; the `scope_reason` variable is set to the current one:

`PARSING_REASON`	The usual one. Note that `action_to_be` holds `NULL` in the early stages (before the verb has been decided) and later on the action which would result from a successful match.
`TALKING_REASON`	Working out which objects are in scope for being spoken to (see the end of §12 for exercises using this).
`EACHTURN_REASON`	When running `each_turn` routines for anything nearby, at the end of each turn.
`REACT_BEFORE_REASON`	When running `react_before`.
`REACT_AFTER_REASON`	When running `react_after`.
`TESTSCOPE_REASON`	When performing a `TestScope`.
`LOOPOVERSCOPE_REASON`	When performing a `LoopOverScope`.

Here are some examples. Firstly, as promised, how to change the rule that "things you've just dropped disappear in the dark":

```
[ InScope person i;
  if (person==player && location==thedark)
      objectloop (i near player)
          if (i has moved)
              PlaceInScope(i);
  rfalse;
];
```

With this routine added, the objects in the dark room the player is in are in scope only if they have `moved` (that is, have been held by the player in the past); and even then, are in scope only to the player.

• △△ EXERCISE 83

Construct a long room divided by a glass window. Room descriptions on either side should describe what's in view on the other; the window should be lookable-through; objects on the far side should be in scope, but not manipulable; and everything should cope well if one side is in darkness.

• △△ EXERCISE 84

Code the following puzzle. In an initially dark room there is a light switch. Provided you've seen the switch at some time in the past, you can turn it on and off – but before you've ever seen it, you can't. Inside the room is nothing you can see, but you can hear a dwarf breathing. If you tell the dwarf to turn the light on, he will.

As mentioned in the definition above, each object has the ability to drag other objects into scope whenever it is in scope. This is especially useful for giving objects component parts: e.g., giving a washing-machine a temperature dial. (The dial can't be a child object because that would throw it in with the clothes: and it ought to be attached to the machine in case the machine is moved from place to place.) For this purpose, the property `add_to_scope` may contain a list of objects to add.

△ Alternatively, it may contain a routine. This routine can then call `AddToScope(x)` to put any object `x` into scope. It may not, however, call `ScopeWithin` or any other scoping routines.

△△ Scope addition does *not* occur for an object moved into scope by an explicit call to `PlaceInScope`, since this must allow complete freedom in scope selections. But it does happen when objects are moved in scope by calls to `ScopeWithin(domain)`.

• EXERCISE 85

(From the tiny example game 'A Nasal Twinge'.) Give the player a nose, which is always in scope and can be held, reducing the player's carrying capacity.

• EXERCISE 86

(Likewise.) Create a portable sterilising machine, with a "go" button, a top which things can be put on and an inside to hold objects for sterilisation. (Thus it is a container, a supporter and a possessor of sub-objects all at once.)

• △△ EXERCISE 87

Create a red sticky label which the player can affix to any object in the game. (Hint: use `InScope`, not `add_to_scope`.)

• REFERENCES

'Balances' uses `scope = ⟨routine⟩` tokens for legible spells and memorised spells. • See also the exercises at the end of §12 for further scope trickery.

25 Helping the parser out of trouble

△ Once you begin programming the parser on a large scale, you soon reach the point where the parser's ordinary error messages no longer appear sensible. The `ParserError` entry point can change the rules even at this last hurdle: it takes one argument, the error type, and should return true to tell the parser to shut up, because a better error message has already been printed, or false, to tell the parser to print its usual message. The error types are all defined as constants:

`STUCK_PE`	I didn't understand that sentence.
`UPTO_PE`	I only understood you as far as...
`NUMBER_PE`	I didn't understand that number.
`CANTSEE_PE`	You can't see any such thing.
`TOOLIT_PE`	You seem to have said too little!
`NOTHELD_PE`	You aren't holding that!
`MULTI_PE`	You can't use multiple objects with that verb.
`MMULTI_PE`	You can only use multiple objects once on a line.
`VAGUE_PE`	I'm not sure what 'it' refers to.
`EXCEPT_PE`	You excepted something not included anyway!
`ANIMA_PE`	You can only do that to something animate.
`VERB_PE`	That's not a verb I recognise.
`SCENERY_PE`	That's not something you need to refer to...
`ITGONE_PE`	You can't see 'it' (the *whatever*) at the moment.
`JUNKAFTER_PE`	I didn't understand the way that finished.
`TOOFEW_PE`	Only five of those are available.
`NOTHING_PE`	Nothing to do!
`ASKSCOPE_PE`	*whatever the scope routine prints*

Each unsuccessful grammar line ends in one of these conditions. A verb may have many lines of grammar; so by the time the parser wants to print an error, all of them must have failed. The error message it prints is the most 'interesting' one: meaning, lowest down this list.

△ The `VAGUE_PE` and `ITGONE_PE` apply to all three pronouns ("it", "him" and "her"). The variable `vague_word` contains the dictionary address of which is involved (`'it'`, `'him'` or `'her'`). Note that the variables `itobj`, `himobj` and `herobj` hold the current settings of the pronouns.

△ The Inform parser resolves ambiguous inputs with a complicated algorithm based on practical experience. However, it can't have any expertise with newly-created verbs: here is how to provide it. If you define a routine

```
ChooseObjects(object, code)
```

then it's called in two circumstances. If `code` is 0 or 1, the parser is considering including the given object in an "all": 0 means the parser is intending not to include it, 1 means it intends not to. The routine should reply

- 0 (or false) to say "carry on";
- 1 to force it to be included; or
- 2 to force it to be excluded.

It may want to decide using `verb_word` (the variable storing the current verb word, e.g., `'take'`) and `action_to_be`, which is the action which would happen if the current line of grammar were successfully matched.

The other circumstance is when `code` is 2. This means the parser is sorting through a list of items (those in scope which best matched the input), trying to decide which single one is most likely to have been intended. If it can't choose a best one, it will give up and ask the player. `ChooseObjects` should then return a number from 0 to 9 (0 being the default) to give the object a score for how appropriate it is.

For instance, some designers would prefer "take all" not to attempt to take `scenery` objects (which Inform, and the original Infocom parser, will do). Let us code this, and also teach the parser that edible things are more likely to be eaten than inedible ones:

```
[ ChooseObjects obj code;
  if (code<2) { if (obj has scenery) return 2; rfalse; }
  if (action_to_be==##Eat && obj has edible) return 3;
  if (obj hasnt scenery) return 2;
  return 1;
];
```

Scenery is now excluded from "all" lists; and is further penalised in that non-scenery objects are always preferred over scenery, all else being equal. Most objects score 2 but edible things in the context of eating score 3, so "eat black" will now always choose a Black Forest gateau in preference to a black rod with a rusty iron star on the end.

• △ EXERCISE 88

Allow "lock" and "unlock" to infer their second objects without being told, if there's an obvious choice (because the player's only carrying one key), but to issue a disambiguation question otherwise. (Use `Extend`, not `ChooseObjects`.)

• REFERENCES

See 'Balances' for a usage of `ParserError`.

Chapter IV: Testing and Hacking

26 Debugging verbs and tracing

> If builders built buildings the way programmers write programs, the first woodpecker that came along would destroy civilisation.
>
> – old computing adage

Infocom claimed to have fixed nearly 2000 bugs in the course of writing 'Sorceror', which is a relatively simply game today. Adventure games are exhausting programs to test and debug because of the huge number of states they can get into, many of which did not occur to the author. (For instance, if the player solves the "last" puzzle first, do the other puzzles still work properly? Are they still fair?) The main source of error is simply the designer not noticing that some states are possible. The Inform library can't help with this, but it does contain features to help the tester to quickly reproduce states (by moving objects around freely, for instance) and to see what the current state actually is (by displaying the tree of objects, for instance).

Inform provides a small suite of debugging verbs to this end, but only if the game contains the line

```
Constant DEBUG;
```

to define the constant `DEBUG`, before including the library files. (Just in case you forget having done this, the letter `D` appears in the game banner to stop you releasing such a version by accident.)

You then get the following verbs, which can be used at any time in play:

```
purloin <anything>
abstract <anything> to <anything>
tree                tree <anything>
scope               scope <anything>
goto <number>       gonear <anything>
actions    actions on    actions off
routines   routines on   routines off
timers     timers on     timers off
trace      trace on      trace off    trace <1 to 5>
recording  recording on  recording off
replay
random
```

You can "purloin" any item or items in your game at any time, wherever you are. This clears `concealed` for anything it takes, if necessary. You can likewise "abstract" any item to any other item (meaning: move it to the other item). To get a listing of the objects in the game and how they contain each other, use "tree", and to see the possessions of one of them alone, use "tree ⟨that⟩". The command "scope" prints a list of all the objects currently in scope, and can optionally be given the name of someone else you want a list of the scope for (e.g., "scope pirate"). Finally, you can go anywhere, but since rooms don't have names understood by the parser, you have to give either the object number, which you can find out from the "tree" listing, or the name of some object in the room you want to go to (this is what "gonear" does). Turning on "actions" gives a trace of all the actions which take place in the game (the parser's, the library's or yours); turning on "routines" traces every object routine (such as `before` or `life`) that is ever called, except for `short_name` (as this would look chaotic, especially on the status line). Turning on "timers" shows the state of all active timers and daemons each turn.

The commands you type can be transcribed to a file with the "recording" verb, and run back through with the "replay" verb. (This may not work under some implementations of the ITF interpreter.) If you're going to use such recordings, you will need to fix the random number generator, and the "random" verb should render this deterministic: i.e., after any two uses of "random", the same stream of random numbers results. Random number generation is poor on some machines: you may want to `Replace` the random-number generator in software instead.

A test version of Infix, a source-level debugger for Inform, is now available from its author, Dilip Sequeira: it is an enhanced form of Mark Howell's Zip interpreter providing for breakpoints, tracing and so forth. It should ultimately be publically archived with the rest of the Inform project.

△ For Infix's benefit, Inform (if compiling with the option set) produces a file of "debugging information" (cross-references of the game file with the source code), and anyone interested in writing an Inform utility program may want to know the format of this file: see the short C program Infact which prints out the debugging information file in English.

On most interpreters, though, run-time crashes can be mysterious, since the interpreters were written on the assumption that they would only ever play Infocom game files (which are largely error-free). Zip is better here and will usually tell you why and where the problem is; given a game file address you can work back to the problem point in the source either with Mark Howell's txd (disassembler) or by running Inform with the assembler trace option on.

Here are all the ways I know to crash an interpreter at run-time (with high-level Inform code, that is; if you insist on using assembly language or the `indirect` function you're raising the stakes), arranged in decreasing order of likelihood:

- Writing to a property which an object hasn't got;
- Dividing by zero, possibly by calling `random(0)`;
- Giving a string or numerical value for a property which can only legally hold a routine, such as `before`, `after` or `life`;
- Applying `parent`, `child` or `children` to the `nothing` object;

- Using `print object` on the `nothing` object, or for some object which doesn't exist (use `print (name)`, `print (the)` etc., instead as these are safeguarded);
- Using `print (string)` or `print (address)` to print from an address outside the memory map of the game file, or an address at which no string is present (this will result in random text appearing, possibly including unprintable characters, which might crash the terminal);
- Running out of stack space in a recursive loop.

△ There are times when it's hard to work out what the parser is up to and why (actually, most times are like this). The parser is written in levels, the lower levels of which are murky indeed. Most of the interesting things happen in the middle levels, and these are the ones for which tracing is available. The levels which can be traced are:

Level 1	Grammar lines
Level 2	Individual tokens
Level 3	Object list parsing
Level 4	Resolving ambiguities and making choices of object(s)
Level 5	Comparing text against an individual object

"trace" or "trace on" give only level 1 tracing. Be warned: "trace five" can produce reams of text when you try anything at all complicated: but you do sometimes want to see it, to get a list of exactly everything that is in scope and when. There are two levels lower than that but they're too busy doing dull spade-work to waste time on looking at `parser_trace`. There's also a level 0, but it consists mostly of making arrangements for level 1, and isn't very interesting.

△△ Finally, though this is a drastic measure, you can always compile your game `-g` ('debugging code') which gives a listing of every routine ever called and their parameters. This produces an enormous melée of output. More usefully you can declare a routine with an asterisk `*` as its first local variable, which produces such tracing only for that one routine. For example,

```
[ ParseNoun * obj n m;
```

results in the game printing out lines like

```
[ParseName, obj=26, n=0, m=0]
```

every time the routine is called.

• REFERENCES

A simple debugging verb called "xdeterm" is defined in the `DEBUG` version of 'Advent', to make the game deterministic (i.e., not dependant on what the random number generator produces). • See David Wagner's library extension "showobj.h" for a debugging verb which prints out an object's current state (its property values and attributes) in a neat Inform format. (This is unfortunately slightly too long to include in the standard library.)

27 Limitations on the run-time format

How wide the limits stand
Between a splendid and an happy land.
– Oliver Goldsmith (1728–1774), *The Deserted Village*

The Infocom run-time format is well-designed, and has three major advantages: it is compact, widely portable and can be quickly executed. Nevertheless, like any rigidly defined format it imposes limitations. These are not by any means pressing. Inform itself has a flexible enough memory-management system not to impose artificial limits on numbers of objects and the like.

The format comes in several versions, of which the default is now Advanced (or version 5). Standard, version 3, games can still be compiled on request but the V3 format imposes genuine restrictions. Two new formats have recently been created for very large games: version 7 and version 8. Inform compiles these, but a small enhancement of the "Zip" interpreter is required to run them. (See the latest edition of the *Specification of the Z-Machine* for details.) This modification will hopefully become standard but in the mean time, very large games can simply be distributed with a suitably modified interpreter.

Memory. This is the only serious restriction. The maximum size of a game (in K) is given by:

V3	V4	**V5**	V6	V7	**V8**
128	256	256	576	320	512

Because games are encoded in a very compressed form, and because the centralised library of Inform is efficient in terms of not duplicating code, even 128K allows for a game at least half as large again as a typical old-style Infocom game. The default format (V5) will hold a game as large and complex as the final edition of 'Curses', substantially bigger than any Infocom game, with room to spare. V6, the late Infocom graphical format, should be avoided for text games, as it is much more difficult to interpret. The V8 format allows quite gargantuan games (one could implement, say, a merging of the 'Zork' and 'Enchanter' trilogies in it) and is recommended as the standard size for games too big to fit in V5. V7, which is implemented in a slightly different way, is provided as an alternative and may be easier to get working on old interpreters other than Zip.

Grammar. The number of verbs is limited only by memory. Each can have up to 20 grammar lines (one can recompile Inform with `MAX_LINES_PER_VERB` defined to a higher setting to increase this) and a line contains at most 6 tokens. (Using general parsing routines will prevent either restriction from biting.)

Vocabulary. There is no theoretical limit. Typical games have vocabularies of between 1000 and 2000 words, but doubling that would pose no problem.

Dictionary resolution. Dictionary words are truncated to their first 9 letters (except that non-alphabetic characters, such as hyphens, count as 2 "letters" for this purpose). They must begin with an alphabetic character and upper and lower case letters are considered equal. (In V3, the truncation is to 6 letters.)

Attributes, properties, names. 48 attributes and 63 properties are available, and each property can hold 64 bytes of data. Hence, for example, an object can have up to 32 names. These restrictions are harmless in practice: except in V3, where the numbers in question are 32, 31, 8 and 4, which begins to bite.

Special effects. V3 games cannot have special effects such as bold face and underlining. (See the next two sections.)

Objects. Limited only by memory: except in V3, where the limit is 255.

Memory management. The Z-machine does not allow dynamic allocation or freeing of memory: one must statically define an array to a suitable maximum size and live within it. Likewise, objects cannot dynamically be created or destroyed (though this is easily imitated). These restrictions greatly increase the portability of the format, and the designer's confidence that the game's behaviour is genuinely independent of the machine it's running on: memory allocation at run-time is a fraught process on many machines.

Global variables. There can only be 240 of these, and the Inform compiler uses 5 as scratch space, while the library uses slightly over 100; but since a typical game uses only a dozen of its own, code being almost always object-oriented, the restriction is never felt. An unlimited number of `Array` statements is permitted and array entries do not, of course, count towards the 240.

"Undo". No "undo" verb is available in V3.

Function calls. A function can be called with at most 7 arguments. (Or, in V3, at most 3.)

Recursion and stack usage. The limit on this is rather technical (see the *Specification of the Z-Machine*). Roughly speaking, recursion is permitted to a depth of 90 routines in almost all circumstances (and often much deeper). Direct usage of the stack via assembly language must be modest.

△ If memory does become short, there is a standard mechanism for saving about 8-10% of the memory. Inform does not usually trouble to, since there's very seldom the need, and it makes the compiler run about 10% slower. What you need to do is define abbreviations and then run the compiler in its "economy" mode (using the switch `-e`). For instance, the directive

```
Abbreviate " the ";
```

(placed before any text appears) will cause the string " the " to be internally stored as a single 'letter', saving memory every time it occurs (about 2500 times in 'Curses', for instance). You can have up to 64 abbreviations. A good list of abbreviations can be found in the *Technical Manual*: basically, avoid proper nouns and instead pick on short combinations of a space and common two- or three-letter blocks. You can even get Inform to work out by itself what a good stock of abbreviations would be: but be warned, this makes the compiler run about 29000% slower.

28 Boxes, menus and drawings

> Yes, all right, I won't do the menu... I don't think you realise how long it takes to do the menu, but no, it doesn't matter, I'll hang the picture now. If the menus are late for lunch it doesn't matter, the guests can all come and look at the picture till they are ready, right?
>
> – John Cleese and Connie Booth, *Fawlty Towers*

• **WARNING**
The special effects in this section do not work on Standard games (though an approximation to menus is produced).

One harmless effect, though not very special, is to ask the player a yes/no question. To do this, print up the question and then call the library routine `YesOrNo`, which returns true/false accordingly.

The status line is perhaps the most distinctive feature of Infocom games in play. This is the (usually highlighted) bar across the top of the screen. Usually, the game automatically prints the current game location, and either the time or the score and number of turns taken. It has the score/turns format unless the directive

```
Statusline time;
```

has been written in the program, in which case the game's 24-hour clock is displayed.

△ If you want to change this, just `Replace` the parser's private `DrawStatusLine` routine. This requires a little assembly language: see the next section for numerous examples.

About character graphic drawings: on some machines, text will by default be displayed in a proportional font (i.e., one in which the width of a letter depends on what it is, so that for example an 'i' will be narrower than an 'm'). If you want to display a diagram made up of letters, such as a map, the spacing may then be wrong. The statement `font off` ensures that any fancy font is switched off and that a fixed-pitch one is being used: after this, `font on` restores the usual state.

• **WARNING**
Don't turn the `font` on and off in the middle of a line; this doesn't look right on some machines.

△ When trying to produce a character-graphics drawing, you sometimes want to produce the `\` character, one of the four "escape characters" which can't normally be included in text. A double `@` sign followed by a number includes the character with that ASCII code; thus:

`@@64` produces the literal character `@`
`@@92` produces `\`   `@@94` produces `^`   `@@126` produces `~`

△△ Some interpreters are capable of much better character graphics (those equipped to run the Infocom game 'Beyond Zork', for instance). There is a way to find out if this feature is provided and to make use of it: see the *Specification of the Z-Machine*.

△△ A single `@` sign is also an escape character. It must be followed by a 2-digit decimal number between 0 and 31 (for instance, `@05`). What this prints is the n-th 'variable string'. This feature is not as useful as it looks, since the only legal values for such a variable string are strings declared in advance by a `LowString` directive. The `String` statement then sets the n-th variable string. For details and an example, see the answer to the east-west reversal exercise in §6.

A distinctive feature of later Infocom games was their use of epigrams. The assembly language required to produce this effect is easy but a nuisance, so there is an Inform statement to do it, `box`. For example,

```
box "I might repeat to myself, slowly and soothingly,"
    "a list of quotations beautiful from minds profound;"
    "if I can remember any of the damn things."
    ""
    "-- Dorothy Parker";
```

Note that a list of one or more lines is given (without intervening commas) and that a blank line is given by a null string. Remember that the text cannot be too wide or it will look awful on a small screen. Inform will automatically insert the boxed text into the game transcript, if one is being made. The author takes the view that this device is amusing for irrelevant quotations but irritating when it conveys vital information (such as "Beware of the Dog"). Also, some people might be running your game on a laptop with a vertically challenged screen, so it is polite to provide a "quotes off" verb.

A snag with printing boxes is that if you do it in the middle of a turn then it will probably scroll half-off the screen by the time the game finishes printing for the turn. The right time to do so is just after the prompt (usually `>`) is printed, when the screen will definitely scroll no more. You could use the `Prompt:` slot in `LibraryMessages` to achieve this, but a more convenient way is to put your box-printing into the entry point `AfterPrompt` (called at this time each turn).

• **EXERCISE 89**

Implement a routine `Quote(n)` which will arrange for the n-th quotation (where $0 \leq n \leq 49$) to be displayed at the end of this turn, provided it hasn't been quoted before.

Sometimes one would like to provide a menu of text options (for instance, when producing instructions which have several topics, or when giving clues). This can be done with the `DoMenu` routine, which imitates the traditional "Invisiclues" style. By setting `pretty_flag=0` you can make a simple text version instead; a good idea for machines with very small screens. Here is a typical call to `DoMenu`:

```
DoMenu("There is information provided on the following:^\
       ^     Instructions for playing\
       ^     The history of this game\
       ^     Credits^",
       #r$HelpMenu, #r$HelpInfo);
```

Note the layout, and especially the carriage returns. The second and third arguments are themselves routines: the notation `#r$`, seldom seen in high-level Inform, allows routine names to become ordinary numerical values. (Actually the first argument can also be a routine to print a string instead of the string itself, which might be useful for adaptive hints.) The `HelpMenu` routine is supposed to look at the variable `menu_item`. In the case when this is zero, it should return the number of entries in the menu (3 in the example). In any case it should set `item_name` to the title for the page of information for that item; and `item_width` to half its length in characters (this is used to centre titles on the screen). In the case of item 0, the title should be that for the whole menu.

The second routine, `HelpInfo` above, should simply look at `menu_item` (1 to 3 above) and print the text for that selection. After this returns, normally the game prints "Press [Space] to return to menu" but if the value 2 is returned it doesn't wait, and if the value 3 is returned it automatically quits the menu as if Q had been pressed. This is useful for juggling submenus about.

Menu items can safely launch whole new menus, and it is easy to make a tree of these (which will be needed when it comes to providing hints across any size of game).

• EXERCISE 90

Code an "Invisiclues"-style sequence of hints for a puzzle, revealed one at a time, as a menu item.

Finally, you can change the text style. The statement for this is `style` and its effects are loosely modelled on the VT100 (design of terminal). The style can be `style roman`, `style bold`, `style reverse` or `style underline`. Again, poor terminals may not be able to display these, so you shouldn't hide crucial information in them.

• REFERENCES

'Advent' contains a menu much like that above. • The "Infoclues" utility program translates UHS format hints (a standard, easy to read and write layout) into an Inform file of calls to `DoMenu` which can simply be included into a game; this saves a good deal of trouble.

29 Descending into assembly language

△△ Some dirty tricks require bypassing all of Inform's higher levels to program the Z-machine directly with assembly language. There is an element of danger in this, in that some combinations of unusual opcodes can look ugly on some incomplete or wrongly-written interpreters: so if you're doing anything complicated, test it as widely as possible.

• WARNING

Most of this section does not apply to Standard games.

The best-researched and most reliable interpreter available by far is Mark Howell's Zip; as it's also the fastest, it will hopefully 'take over' entirely. Next comes the InfoTaskForce, which is thorough and should give no serious trouble, but was written when the format was a little less well understood, and so (in some ports) gets some (rare) screen effects wrong. It also lacks an "undo" feature, so the parser's "undo" verb won't work under ITF. The other two publically-available interpreters are pinfocom and zterp, but these are unable to run Advanced games. In the last resort, sometimes it's possible to use one of Infocom's own supplied interpreters with a different game from that it came with; but only sometimes, as they may have inconvenient filenames 'wired into them'. The author recommends that anyone using exotic assembly-language features get hold of both ITF and Zip, and test on both.

Both the common interpreters are, in fact, pretty reliable. But remember that one source of unportability is inevitable. Your game may be running on a screen which is anything from a 64 by 9 pocket organiser up to a 132 by 48 X-window.

Anyone wanting to really push the outer limits (say, by implementing Space Invaders or NetHack) will need to refer to *The Specification of the Z-Machine*, the second edition of which has been rewritten as a comprehensive "standards" document. This is much more detailed (the definition of `aread` alone runs for two pages) and covers the whole range of assembly language. However, this section does document all those features which can't be better obtained with higher-level code.

Lines of assembly language must begin with an `@` character and then the name of the "opcode" (i.e., assembly language statement). A number of arguments, or "operands" follow (how many depends on the opcode): these may be any Inform constants, local or global variables or the stack pointer `sp`, but may not be compound expressions. `sp` does not behave like a variable: writing a value to it pushes that value onto the stack, whereas reading the value of it (for instance, by giving it as an operand) pulls the top value off the stack. Don't use `sp` unless you have to. After the operands, some opcodes require a variable (or `sp`) to write a result into. The opcodes documented in this section are as follows:

```
@split_window    lines
@set_window      window
@set_cursor      line column
@buffer_mode     flag
@erase_window    window
@set_colour      foreground background
@aread           text parse time function <result>
@read_char       1 time function <result>
@tokenise        text parse dictionary
@encode_text     ascii-text length from coded-text
@output_stream   number table
@input_stream    number
@catch           <result>
@throw           value stack-frame
@save            buffer length filename <result>
@restore         buffer length filename <result>
```

`@split_window    lines`

Splits off an upper-level window of the given number of lines in height from the main screen. This upper window usually holds the status line and can be resized at any time: nothing visible happens until the window is printed to. Warning: make the upper window tall enough to include all the lines you want to write to it, as it should not be allowed to scroll.

`@set_window      window`

The text part of the screen (the lower window) is "window 0", the status line (the upper one) is window 1; this opcode selects which one text is to be printed into. Each window has a "cursor position" at which text is being printed, though it can only be set for the upper window. Printing on the upper window overlies printing on the lower, is always done in a fixed-pitch font and does not appear in a printed transcript of the game. Note that before printing to the upper window, it is wise to use `@buffer_mode` to turn off word-breaking.

`@set_cursor      line column`

Places the cursor inside the upper window, where $(1, 1)$ is the top left character.

`@buffer_mode     flag`

This turns on (`flag=1`) or off (`flag=1`) word-breaking for the current window (that is, the practice of printing new-lines only at the ends of words, so that text is neatly formatted). It is wise to turn off word-breaking while printing to the upper window.

`@erase_window    window`

This opcode is unfortunately incorrectly implemented on some interpreters and so it can't safely be used to erase individual windows. However, it can be used with `window=-1`, and then clears the entire screen. Don't do this in reverse video mode, as a bad interpreter may (incorrectly) wipe the entire screen in reversed colours.

`@set_colour      foreground background`

If coloured text is available, set text to be foreground-against-background. The colour numbers are borrowed from the IBM PC:

```
2 = black,  3 = red,      4 = green,  5 = yellow,
6 = blue,   7 = magenta,  8 = cyan,   9 = white
0 = the current setting,  1 = the default.
```

On many machines coloured text is not available: the opcode will then do nothing.

`@aread           text parse time function <result>`

The keyboard can be read in remarkably flexible ways. This opcode reads a line of text from the keyboard, writing it into the `text` string array and 'tokenising' it into a word stream, with details stored in the `parse` string array (unless this is zero, in which case no tokenisation happens). (See the end of §23 for the format of `text` and `parse`.) While it is doing this, it calls `function(time)` every `time` tenths of a second while the user is thinking: the process ends if ever this function returns true. `<result>` is to be a variable, but the value written in it is only meaningful if you're using a "terminating characters table". Thus (by `Replace`ing the `Keyboard` routine in the library files) you could, say, move around all the characters every ten seconds of real time. Warning: not every interpreter supports this real-time feature, and most of those that do count in seconds instead of tenths of seconds.

`@read_char        1 time function <result>`

results in the ASCII value of a single keypress. Once again, the `function` is called every `time` tenths of a second and may stop this process early. Function keys return special values from 129 onwards, in the order: cursor up, down, left, right, function key f1, ..., f12, keypad digit 0, ..., 9. The first operand must be 1 (used by Infocom as a device number to identify the keyboard).

`@tokenise         text parse dictionary`

This takes the text in the `text` buffer (in the format produced by `aread`) and tokenises it (i.e. breaks it up into words, finds their addresses in the dictionary) into the `parse` buffer in the usual way but using the given `dictionary` instead of the game's usual one. (See the *Specification of the Z-machine* for the dictionary format.)

`@encode_text      ascii-text length from coded-text`

Translates an ASCII word to the internal (Z-encoded) text format suitable for use in a `@tokenise` dictionary. The text begins at `from` in the `ascii-text` and is `length` characters long, which should contain the right length value (though in fact the interpreter translates the word as far as a 0 terminator). The result is 6 bytes long and usually represents between 1 and 9 letters.

`@output_stream    number table`

Text can be output to a variety of different 'streams', possibly simultaneously. If `number` is 0 this does nothing. $+n$ switches stream n on, $-n$ switches it off. The output streams are: 1 (the screen), 2 (the game transcript), 3 (memory) and 4 (script of player's commands). The `table` can be omitted except for stream 3, when it's a `table` array holding the text printed; printing to this stream is never word-broken, whatever the state of `@buffer_mode`.

`@input_stream     number`

Switches the 'input stream' (the source of the player's commands). 0 is the keyboard, and 1 a command file (the idea is that a list of commands produced by `output_stream 4` can be fed back in again).

`@catch            <result>`

The opposite of `throw`, `catch` preserves the "stack frame" of the current routine: meaning, roughly, the current position of which routine is being run and which ones have called it so far.

`@throw            value stack-frame`

This causes the program to execute a return with `value`, but as if it were returning from the routine which was running when the `stack-frame` was caught (see `catch`). Any routines which were called in the mean time and haven't returned yet (because each one called the next) are forgotten about. This is useful to get the program out of large recursive tangles in a hurry.

`@save             buffer length filename <result>`

Saves the byte array `buffer` (of size `length`) to a file, whose (default) name is given in the `filename` (a `string` array). Afterwards, `result` holds 1 on success, 0 on failure.

```
@restore          buffer length filename <result>
```

Loads in the byte array `buffer` (of size `length`) from a file, whose (default) name is given in the `filename` (a `string` array). Afterwards, `result` holds the number of bytes successfully read.

• **WARNING**

On some interpreters, a few of these features may not work well: the extended `save` and `restore`, and `catch` / `throw` in particular. (You can always distribute your game with an interpreter that does work well.) The `tokenise` and `encode_text` opcodes work well enough, but the same effects can be achieved much better with higher-level parser programming.

• **EXERCISE 91**

In a role-playing game campaign, you might want several scenarios, each implemented as a separate Inform game. How could the character from one be saved and loaded into another?

• △ **EXERCISE 92**

Design a title page for 'Ruins', displaying a more or less apposite quotation and waiting for a key to be pressed.

• △ **EXERCISE 93**

Change the status line so that it has the usual score/moves appearance except when a variable `invisible_status` is set, when it's invisible.

• △ **EXERCISE 94**

Alter the 'Advent' example game to display the number of treasures found instead of the score and turns on the status line.

• △ **EXERCISE 95**

(From code by Joachim Baumann.) Put a compass rose on the status line, displaying the directions in which the room can be left.

• △△ **EXERCISE 96**

(Cf. 'Trinity'.) Make the status line consist only of the name of the current location, centred in the top line of the screen.

• △△ **EXERCISE 97**

Implement an Inform version of the standard 'C' routine `printf`, taking the form

```
printf(format, arg1, ...)
```

to print out the format string but with escape sequences like `%d` replaced by the arguments (printed in various ways). For example,

```
printf("The score is %e out of %e.", score, MAX_SCORE);
```

should print something like "The score is five out of ten."

• **REFERENCES**

The assembly-language connoisseur will appreciate 'Freefall' by Andrew Plotkin and 'Robots' by Torbjørn Andersson, although the present lack of on-line hints make these difficult games to win.

Chapter V: Language and Compiler Reference

Language is a cracked kettle on which we beat out tunes for bears to dance to, while all the time we long to move the stars to pity.
– Gustave Flaubert (1821–1880)

30 Language specification

The aim here is to describe the underlying language of Inform as if it were a general-purpose programming language. A few technical and diagnostic commands are relegated to the *Technical Manual* (henceforth abbreviated to *TM*), and Inform's assembly language is documented in the *Specification of the Z-Machine.* The version of the language discussed is Inform 5.5, which slightly extends previous versions.

§30.1 Source file format

When Inform reads in a file, it treats a few characters in special ways. The character `!` (when not inside single or double quotes) means the rest of the line (up to the next new-line) is a comment, and Inform throws it away, e.g.,

```
parade.number = 78;     !  Setting the number of trombones
```

The backslash `\` can be used inside strings in double-quotes `"like so"` to 'fold' them, so that the new-line and subsequent spaces are ignored: e.g., in

```
print "~Ou sont les neiges d'antan?~ \
       Marjory asks, passing the bowl of grapes.^";
```

the string is understood to have only one space (and no new-line) between the `~` and the `M` of Marjory. Inside double-quotes, the `~` is understood as a double-quote, and the `^` as a new-line: thus the above string is actually read as

"Ou sont les neiges d'antan?" Marjory asks, passing the bowl of grapes.

followed by a new-line. If you want to get an un-typeable character in a double-quoted string, or one which would otherwise cause problems, write `@@` followed by its ASCII code in decimal. (One to four digits of decimal may be given, but see the *Specification* for what values outside the normal ASCII range of 32 to 126 produce: in particular, German accented characters may be available.) For example, `@@92` produces a literal backslash and `@@64` produces a literal `@` sign. (A single `@` is also an escape character, for variable strings: see the east-west reflection exercise of §6 for brief notes.)

Otherwise, new-lines have no significance and are treated as spaces, as are tab characters.

Inside single quotes, "'", an apostrophe (i.e. a single quote) is also written ^. For instance,

```
if (word == 'can^t' or 'isaac^s') ...
```

Inform source code is a list of **directives**, which are instructions to the compiler itself (such as "create an object as follows"), and **routines**, which are pieces of code for it to compile.

§30.2 The logical machine

All Inform programs run on an imaginary computer called the Z-machine. A program consists of **routines**, which may either stand alone or be attached to particular objects (these are called "embedded routines"). Almost all data is represented by 16-bit numbers (2 bytes long). For some purposes, these are considered signed in the usual way, holding values

$$-32768 \leq n \leq 32767$$

with the hexadecimal value `$ffff` (i.e., 65535) being the same as -1. The operations of addition, subtraction, multiplication and comparison are signed; but division (rounded to the integer below), calculation of remainder after division and bitwise operations are not. So for instance $(-4)+(-1)=-5$ but $(-4)/(-1)=65532/65535=0$.

Global variables store numbers such as these, and so do **local variables** (which are local in that they belong to particular routines). In all Inform expressions, such as

```
random(100+lives_left)
```

everything is always a number: 100, `lives_left` and the result.

In addition, the machine contains **objects**. These are related in a tree, so that an object may be considered to contain other objects, which may themselves contain others, and so on. Objects are referred to by number (these count upwards from 1, with a value of 0 representing 'nothing', which is not an object but a concept). Objects carry certain variables, called **properties**, with them, and also flags (states which are either on or off) called **attributes**.

The **dictionary** contains words which might be understood by the game. Each word in the dictionary has a unique associated number (actually its address in the dictionary table), so a number may also refer to a dictionary word.

Inform has no concept of the 'type' of an expression, so the compiler will allow (say)

```
'marble' - Brass_Lamp
```

even though subtracting the object number of the brass lamp from the dictionary reference to the word "marble" is hardly going to have a meaningful result.

△ The **memory map** of the machine is divided into three. At the bottom (in terms of addresses) is dynamic memory, data which can be written or read: this is stored when a game is saved to disc. Next comes a region of static, read-only memory which can freely be read from, including (for instance) the dictionary. The lower two regions together always take up less than 64K: when a number is regarded as an "address", it refers to one byte in these regions by number, upwards from 0.

△△ The third and largest region of memory, containing the program itself and (almost) all strings (such as room descriptions), may extend the game's size to the top of memory, up to the maximum (between 128K and 512K, depending on format). It is read-only and that only in a limited way. Ordinary addresses can't reach above the 64K mark, so one cannot have a byte address into this region. Instead, every routine and string has a reference number called a "packed address"; there are commands to call the routine or to print the string with given packed address, but that's the only access allowed.

△△ Inform guarantees that the following numbers are all different:

- zero;
- -1, which equals `$ffff`, which equals the library constant *NULL*;
- the number of an object;
- a function's packed address;
- a string's packed address.

It is thus possible to partially deduce the type of a number (see the library function `ZRegion`) from its value. But note that byte addresses – in particular, dictionary addresses – are not guaranteed to differ. The translation function from packed to real addresses depends on the version number, and no other assumption should be made about it.

△△ The entire Z-machine lives in the memory map except for the **stack** (which is also stored when the game is saved). It is only accessible to assembly language and use of it is not recommended to those of a nervous disposition.

§30.3 Constants

Here is a set of example constants:

```
31415  -1  $ff  $$1001001  'lantern'  ##Look  'X'
"an emerald the size of a plover's egg"
"~Hello,~ said Peter.^~Hello, Peter,~ said Jane.^"
```

String constants in double-quotes are discussed above. Numbers can be written in decimal (in the obvious way), or in hexadecimal, preceded by a `$`,

```
$ffff  $1a  $31
```

or in binary, preceded by a double dollar `$$`. Single characters can be represented in single quotation marks, e.g.

```
'a'  '\'  '"'  'z'
```

Dictionary words are also written in single quotes, e.g.

```
'aardvark'  'tetrahedron'  'marquis'
```

△ This is a little ambiguous, but Inform knows the difference because a dictionary word contains more than one letter. Very occasionally one needs to put a one-letter word in the dictionary: to get the word 'a', for instance, write **`#n$a`** (the **`#n$`** usage is otherwise obsolete).

△ Writing the constant `'marjoram'` somewhere in the code actually inserts the word "marjoram" into the dictionary if it isn't already present. Likewise, writing `"You blink."` will compile the string automatically.

△△ These are all internally represented by numbers. Characters are held as ASCII codes; dictionary words by their reference numbers; and strings in double-quotes by their packed addresses. Note, though, that of the two conditions

```
'yes' == 'yes'          "no" == "no"
```

the first is always true (the word 'yes' is only in the dictionary once), whereas the second is probably false, because Inform has compiled two copies of the string "no", which have different packed addresses.

Actions (and fake actions) have numbers, by which it is sometimes helpful to refer to them: "the action number corresponding to `Take`" is written `##Take`.

Other constants known to Inform are:

- those already defined, by the `Constant` directive, in your code or by the library;
- names of specific objects (an object may move and alter in play, but the number which refers to it does not);
- names of attributes and properties already created;
- names of arrays, defined by the `Array` directive;
- some arcane ones always created by Inform (see the *TM*).

△ Two standard library-defined constants are `nothing`, which equates to 0, and is the "no object" value (thus, the `child` of a childless object is equal to `nothing`); and `NULL`, which equates to -1 or (in hexadecimal) equivalently to `$ffff`, used as the "not given" default value of properties which are expected to be routines (such as `before`).

△△ Finally, you can also write the packed address of a function (defined by you elsewhere in the code) as a constant. In the context of an initial value (for instance, when declaring an array entry or object property) you can just give its name. In an expression, however, the name must be preceded by `#r$`: for instance, `fn_to_call=#r$Name;`.

§30.4 Directives

A directive is an instruction to the compiler, rather than code for it to compile. Inside routines, directives must start with a `#` (to distinguish them from statements), but outside routines this is optional (and usually omitted). Directives end with a semi-colon `;` (just as statements do). The following directives define or create things:

`Array` ⟨name⟩	Make an array of data
`Attribute` ⟨name⟩	Define a new attribute
`Class` ...	Define a new class
`Constant` ⟨name⟩ ⟨value⟩	Define a named constant
`Extend` ...	Make extra grammar for an existing verb
`Fake_action` ⟨name⟩	Define a new "fake action"
`Global` ⟨name⟩ ...	Make a global variable
`Nearby`	Make an object inside the last `Object`
`Object`	Make an object

`Property` ⟨name⟩ ...	Define a new property
`Verb` ...	Make grammar for a new verb

(The ⟨value⟩ of a `Constant` is zero if unspecified; the other directives are described more fully below.) The next set affect Inform's choice of what to compile and what not to:

`End`	End compilation here
`Endif`	End of conditional compilation
`Ifdef` ⟨name⟩	Compile only if symbol is defined
`Ifndef` ⟨name⟩	Compile only if symbol is undefined
`Ifnot`	Compile only if previous `If...` failed
`Ifv3`	Compile only for Standard games
`Ifv5`	Compile only for Advanced games
`Include` ⟨file-name⟩	Include that file here
`Replace` ⟨routine-name⟩	Don't compile this library routine

△ Conditional compilation allows code for routines which need only exist in some "versions" of your games. For instance,

```
print "Welcome to the ";
#IFV3; print "Standard"; #IFNOT; print "Advanced"; #ENDIF;
print " version of Zork LVI.";
```

(The `#IFNOT` clause is optional.) Note the trailing semicolon: Inform is not C! Such clauses may be nested up to 32 deep, and may contain whole routines. They may not, however, conditionally give *part* of a statement or directive. Thus, for instance,

```
print #IFV3; "Standard"; #IFNOT; "Advanced"; #ENDIF;
```

is *not* legal.

△△ One special case is checking to see if the constant `VN_****` is defined, where `****` is a four-digit number n: it is if and only if the current Inform version number is at least n. Thus

```
#IFDEF VN_1501; print "The all new Inform show!^"; #ENDIF;
```

compiles the statement only under Inform 5.5 or later.

△△ Warning: it is possible to accidentally arrange for a block of code only to be considered on one of the two passes of the compiler: Inform will not like this. (Cf. the end of the library grammar file for an example of avoiding this problem.)

A few directives alter settings:

`Release` ⟨number⟩	Set the game's Release Number
`Serial` ⟨string⟩	Set the game's Serial Number
`Statusline` ...	Make the status line show score or time
`Switches` ⟨switches⟩	Set default compilation switches

The release number of a game (by default 1) is generally an edition number; the serial number is the compilation date in the form 950331, that is, yymmdd. Inform sets this automatically (on machines where the date is accessible), so the `Serial` directive is provided only for forgers and machines without an internal clock. `Statusline score` or `Statusline time` declare which piece of information should be displayed on screen in the top right during play. `Switches`, which if present should be the first directive in the source, sets "command-line switches" as if they had been typed as part of the command starting Inform. For instance,

```
Switches dv8;
```

declares that the game must be compiled as version-8 and that double-spaces are to be contracted.

△△ These recondite directives exist, but not for public use:

```
Default Dictionary Listsymbols Listdict Listverbs Lowstring Stub System_file
Trace Btrace Etrace Ltrace Notrace Nobtrace Noetrace Noltrace
```

(all in the *TM*). The one low-level directive which might be of practical use is `Abbreviate` (see §28), an economy measure for enormous games.

§30.5 Property and attribute definitions

Inform itself defines only one property (`name`, see below) and no attributes: all others must be declared before use (and the library defines many). The syntax is

`Attribute` ⟨name⟩ `[alias` ⟨existing-attribute⟩ `]`
`Property [`⟨qualifier⟩`]` ⟨name⟩ `[alias` ⟨existing-property⟩`]`
or `Property [`⟨qualifier⟩`]` ⟨name⟩ `[`⟨default-value⟩`]`

The `alias` form is used for making new names for existing attributes and properties, so that the same physical property can (with extreme care) be used for two different things in different contexts; the library indulges in a little of this chicanery, but it is not recommended. There are two property "qualifiers": `additive`, discussed below, and `long`.

△△ `long` is meaningful only in Standard (V3) games and obsolete anyway: under Inform 5.5, or in other versions, all properties are "long".

§30.6 Object and class definitions

An object definition consists of a header giving its name and initial residence, followed by a body of its initial properties and attributes; a class definition just has a name and such a body.

△ The full syntax of the header is

`Object` ⟨obj-name⟩ `"short name" [`⟨parent-obj⟩`]`
or `Nearby` ⟨obj-name⟩ `"short name"`
or `Class` ⟨class-name⟩

and the parent object, if given, must have already been defined. The parent of a `Nearby` object is the last object defined by `Object` rather than `Nearby`, which is usually a room definition. A class creates no specific object, so has no specific parent. The syntax for an object, then, is

```
⟨Header⟩ [,]
  class ⟨class-1⟩ ⟨class-2⟩ ... ⟨class-n⟩ [,]
  with ⟨property-name-1⟩ ⟨value-1⟩ ... ⟨value-n⟩,
    ⟨property-name-2⟩ ⟨value-1⟩ ... ⟨value-n⟩,
    ...
    ⟨property-name-n⟩ ⟨value-1⟩ ... ⟨value-n⟩ [,]
  has ⟨att-1⟩ ⟨att-2⟩ ... ⟨att-n⟩
```

Although it's conventional to write `class`, `with` and `has` in this order, actually they can be in any order and any or all can be omitted altogether: and the commas in square brackets `[,]` are optional in between these fields. The classes listed under `class` are those which the object inherits from. Each ⟨value⟩ can be any legal constant: up to 4 can be given per property in Standard games but up to 32 in other versions. In addition, a property may, *instead* of a list of constants, give as its value a (nameless) embedded routine.

△ For deep-rooted historical reasons, one property is treated differently from the others: `name`. Its data must be a list of English words in double-quotes, to be put into the dictionary. This is illogical, as dictionary words are normally referred to in single quotes: but it has the advantage that single-letter words are more easily written.

△△ The attributes ⟨att-1⟩ and so on can be taken away as well as added, thus:

```
...
has  light ~scored;
```

which is sometimes useful to over-ride an inheritance from a class definition.

△△ Here is exactly how inheritance works. Suppose an object is created with classes $C_1, ..., C_n$ (in the order they are listed). It starts out tabula rasa (a blank slate), with no attributes and no properties. It then inherits the attributes and property values of C_1; next C_2, and so on up to C_n; finally it acquires the attributes and properties from its own definition. The order is important because there may be a clash. Ordinarily, a later specified value wipes out an earlier one: if C_1 says `number` is 5, and C_2 says it is 7, then the answer is 7.

△△ An `additive` property accumulates values instead. For instance, if C_1 gives

```
name "small" "featureless" "cube",
```

and the object itself has `"green"` as `name`, the result is as if the object had been declared with

```
name "small" "featureless" "cube" "green",
```

because `name` is an additive property (whereas `number` is not).

△△ Classes themselves may inherit from sub-classes, by this same rule.

§30.7 Global variables and arrays

There are two kinds of variable, global and local (plus one special one, the stack pointer `sp`). Global variables must be declared *before* use, by a `Global` directive, so:

`Global` ⟨varname⟩
or `Global` ⟨varname⟩ `=` ⟨initial-value⟩
or `Global` ⟨array-name⟩ ⟨array-type⟩ ⟨initial-values⟩

The initial value can be any constant, and is 0 if not specified.

There are four kinds of array: byte arrays (with entries written `array->0` up to `array->`$(n-1)$), word arrays (with entries `array-->0` up to `array-->`$(n-1)$), strings and tables. The entries in a byte array or a string are just bytes, numbers from 0 to 255 (which can't be negative). Thus they cannot hold dictionary words, function or string packed-addresses, or object numbers. Entries in a word array or a table can be any Inform number.

A string is a special kind of byte array whose 0th entry is the size of the array: thus a string `s` of size 20 contains `s->0` (set to 20), and actual data entries `s->1` up to `s->20`. A table is the analogous kind of word array. For instance,

```
tab-->(random(tab-->0))
```

evaluates to a random entry from the table `tab`.

Arrays are created by

`Array` ⟨array-name⟩ ⟨array-type⟩ ⟨initial-values⟩

where the ⟨array-type⟩ is `->` (byte array), `-->` (word array), `string` or `table`. There are also four ways to give the ⟨initial-values⟩.

⟨number⟩	This many entries, initially 0
⟨value-1⟩ ... ⟨value-n⟩	$n \geq 2$ entries with these values
`"`⟨string⟩`"`	Entries are ASCII values of chars in string
`[`⟨value-1⟩ ... ⟨value-n⟩`]`	n entries with these values

The last of these is useful for making very large arrays of (usually static) data, because semicolons can be scattered freely between the values (avoiding the maximum line length). For example:

```
Array a1 -> 20;
Array a2 string "Hello";
Array a3 --> 1 3 5 7 9 11;
Array Public_Holidays table
[;  "New Year's Day"  "Twelfth Night";
    "Ash Wednesday"   "Good Friday";
    "Martin Luther King Day";
];
```

which will store

```
a1     0  0  0 ... 0           (20 bytes)
a2     5 'H' 'e' 'l' 'l' 'o'   (6 bytes)
a3     1  3  5  7  9  11       (6 words)
```

and so on. Initial values can be any legal constants, including names of other arrays or of functions.

△ The name of an `Array` is a constant whose initial value is the byte address to the area of memory where the data lives. Creating an array with `Global` has identical effect except that the name is of a global variable which initially holds this constant value. (In Inform 5.4 and earlier, all arrays were made this way, somewhat wastefully of global variables. The old keywords `data`, `initial` and `initstr` still work with `Global`, but are considered passé.)

• WARNING

You can write to the size byte (or word) of a string (or table), but that won't make the amount of space allocated any larger: nor is there any bound-checking at run time.

§30.8 Grammar and verbs

For the `Verb` and `Extend` directives, see the summary given in §31.

§30.9 Routines

Routines start with a `[` and end with a `]`. That is, they open with

`[` ⟨Routine-name⟩ ⟨local-var-1⟩ … ⟨local-var-n⟩ `;`

giving the names of local variables for the routine ($0 \leq n \leq 15$). The routine ends with just `];`. Routines embedded in object definitions, i.e. routines which are the value of a property, are the same except that no routine-name is given, and they may end in `],` if the object definition resumes after them.

The first few local variables also hold the arguments passed to the routine when it is called. That is, if you have a routine

```
[ Wander from i j; ...some code...; ];
```

and it is called by `Wander(attic)` then the local variable `from` will initially have the value `attic`. The rest all start out at zero. As a debugging aid, if an asterisk `*` is inserted between the routine name and the variable list then tracing code is compiled to print details each time the routine is called.

Function calls (that is, calls to routines) are legal with between 0 and 7 arguments (except for V3 Standard games, where the maximum is 3) and every routine always returns a value. If execution runs into the `]` at the end, that value is 'true' (or 1) for an ordinary routine, or 'false' (or 0) for an embedded one.

A routine consists of a sequence of lines of code. These come in six varieties:

- assignments (such as `i=23;`);
- statements (such as `if` or `print`);
- action commands in `<`, `>` (or `<<`, `>>`) brackets;
- function calls (see above);
- labels (such as `.PrettyPass;`), beginning with a full stop, provided for you to `jump` to (if you have no tedious scruples about the use of a goto instruction);
- assembly language lines, beginning with an `@` sign and documented in §29 and the *Specification of the Z-Machine*.

It's legal to mix in directives, but a directive inside a routine should begin with a `#` character.

§30.10 Arithmetic expressions

Arithmetic (and other) expressions can contain the following:

`+ -`	plus, minus
`* / % & |`	times, divide, remainder, bitwise and, bitwise or
`-> -->`	byte array, word array entry
`. .& .#`	property, property address, property length
`-`	unary minus
`++ --`	incrementing and decrementing variables (as in C)

The order of precedence is as shown: i.e., those on each line are equally potent, more potent than those above but less than those beneath. Expressions are not allowed to contain conditions, nor assignments: `2+(i=3/j)` is not a legal expression. Some legal examples are:

```
4*(x+3/y)   Fish(x)+Fowl(y)  lamp.time   buffer->5
```

Note that `++` and `--` can only be applied to variables, not to properties or array entries.

• **WARNING**
A division by zero error (such as n`/0` or n`%0`) may crash the game at run time.

§30.11 Built-in functions

A very few functions are built into the language of Inform itself, rather than written out longhand in the library files, but they behave like any other routines. They are:

`parent(obj)`	parent of object
`sibling(obj)`	sibling of object
`child(obj)`	eldest child of object
`children(obj)`	number of (direct) children of object
`eldest(obj)`	same as `child`
`youngest(obj)`	youngest child of object
`elder(obj)`	elder sibling of object
`younger(obj)`	same as `sibling`
`random(x)`	uniformly random number between 1 and $x \geq 1$
`indirect(addr)`	call routine with packed address `addr`, return its return value
`indirect(addr,v1)`	call `addr(v1)`, return its return value
`indirect(addr,v1,v2)`	call `addr(v1, v2)`, return its return value

• **WARNING**
`random(0)` may cause a division by zero error on some interpreters, though it should not.

△△ Although normally implemented in ‘hardware’, these routines can be `Replace`d as if they were library routines in ‘software’.

§30.12 Conditions

A simple condition is

$\langle$a$\rangle$ $\langle$relation$\rangle$ $\langle$b$\rangle$

where the relation is one of

`==`	`a` equals `b`
`~=`	`a` doesn't equal `b`
`< > >= <=`	numeric (signed) comparisons
`has`	object `a` has attribute `b`
`hasnt`	object `a` hasn't attribute `b`
`in`	object `a` is currently held by object `b`
`notin`	...is not...

Note that `in` and `notin` look only at direct possession. Something in a rucksack which the player holds, will not have `in player`, but it will have `in rucksack`. With `==` (and `~=`) only, one may also write the useful construction

⟨something⟩ `==` ⟨v1⟩ `[or` ⟨v2⟩ `[or` ⟨v3⟩`]]`

which is true if the first something is any of the values given. An idiosyncracy of Inform, for 'hardware reasons', is that you can only have three. Conditions can be combined by the `&&` and `||` operators (which have equal priority):

⟨condition1⟩ `&&` ⟨condition2⟩
⟨condition1⟩ `||` ⟨condition2⟩

true if both, or either (respectively) are true. These are always tested left to right until the outcome is known. So, for instance,

```
i==1 || Explode(2)==2
```

does not call `Explode` if `i` is 2. Examples of legal conditions are:

```
i==1 or 2 or 3
door has open || (door has locked && key in player)
```

§30.13 Assignments

There are five legal forms of assignment:

⟨variable⟩ `=` ⟨value⟩`;`
⟨variable⟩`++;` ⟨variable⟩`--;` `++`⟨variable⟩`;` `--`⟨variable⟩`;`
⟨byte-array-or-string⟩`->`⟨entry⟩ `=` ⟨value⟩`;`
⟨word-array-or-table⟩`-->`⟨entry⟩ `=` ⟨value⟩`;`
⟨object⟩`.`⟨property⟩ `=` ⟨value⟩`;`

For example:

```
i=-15-j;   i=j-->1;   albatross.weight = albatross.weight + 1;
(paintpot.&roomlist)-->i = location;   turns++;
```

Although these look logical, they are not allowed:

```
paintpot.#roomlist = 5;
paintpot.&roomlist = my_array;
```

because one cannot change the size or address of a property in play.

• **WARNING**
Attempting to write to a property which an object does not have may crash the game at run time. Likewise, you should not attempt to read or write properties of non-existent objects (such as 0, sometimes called `nothing`).

§30.14 Printing commands

A string on its own, such as

```
"The world explodes in a puff of garlic.";
```

is printed, with a new-line, and the current routine is returned from with return value 'true', i.e., 1. In addition:

`new_line`	prints a new-line
`print ...`	prints the given things
`print_ret ...`	prints, new-lines and returns 1
`spaces n`	prints *n* spaces
`font on/off`	turns proportional fonts on/off
`style ...`	in Advanced games, sets text style
`box "s1" ... "sn"`	in Advanced games, puts up quotation box
`inversion`	prints out the current Inform version number

The text `style`, normally Roman, can be changed to any *one* of

```
roman  reverse  bold  underline
```

`print` and `print_ret` take a comma-separated list of things to print out, which can be:

	"⟨string⟩"	prints this string
	⟨expression⟩	prints this number
`(char)`	⟨expression⟩	prints this ASCII character
`(name)`	⟨expression⟩	prints the name of this object
`(the)`	⟨expression⟩	prints definite article and name
`(The)`	⟨expression⟩	prints capitalised definite article and name
`(a)`	⟨expression⟩	prints indefinite article and name
`(number)`	⟨expression⟩	prints this number in English
`(string)`	⟨expression⟩	prints the string with this packed address
`(address)`	⟨expression⟩	prints the string at this byte address
(⟨Routine⟩)	⟨expression⟩	calls the Routine with this argument

Thus, for example,

```
print_ret (The) x1, " explodes messily.  Perhaps it was unwise to \
          drop it into ", (the) x2, ".";
```

produces, say,

```
The hand grenade explodes messily.  Perhaps it was unwise to drop
it into the glassworks.
```

`print (string) x` should be used to convert the numerical value of `"a string like this"` back to text. `print (address) x` is chiefly useful for printing out dictionary words: thus `print (address) 'piano'` will print the word "piano". These bracketed printing rules are easily added to. Thus, if you define a routine `SpellName(x)` to print the name of spell `x`, then

```
print "Your ", (SpellName) yomin_spell, " discharges horribly.";
```

will work nicely.

A few forms of `print` are now obsolete but still supported: `print char x` does the same as `print (char) x`, and in addition there are three old printing commands:

`print_char a`	same as `print (char) a`
`print_addr a`	same as `print (address) a`
`print_paddr a`	same as `print (string) a`

§30.15 Manipulating objects

`remove obj`	removes object from the tree
`move o1 to o2`	moves `o1` to become eldest child of `o2`
`give obj a1 ... an`	gives attributes to `obj`

Attributes beginning with a `~` are taken away rather than given.

§30.16 Returning from routines

Apart from `print_ret` (and strings in isolation), which return true, one can:

`return`	Return true, i.e., 1
`return x`	Return value x
`rtrue`	Return true, i.e., 1
`rfalse`	Return false, i.e., 0

§30.17 Blocks of code

A block of code may be entirely empty, may be a single instruction or a series of several, in which case it must be enclosed in braces `{` and `}`. Thus, for instance, in

```
if (i==1) print "The water rises!";
if (i==2) { print "The water rises further...";
            water++;
          }
```

the `if` statements contain a block of code each; the effect of

```
for (i=0:i<10:Frog(i++)) ;
```

is just to call `Frog(0)` up to `Frog(9)` (thus, an empty block can be a sensible thing to write). Blocks can be nested inside each other up to 32 deep. An `if` statement (for example) is a single statement even when it contains a great deal of code in its block: so, for example,

```
if (i>1) if (water<10) "The water is beginning to worry you.";
```

is legal. One small exception: an `if` followed by an `else` counts as two statements, and this means Inform handles "hanging elses" in a possibly unwanted way:

```
if (i==1) if (j==1) "Hello."; else "Goodbye.";
```

The `else` clause here attaches to the first `if` statement, not the second, so "Goodbye." is printed exactly when `i==2`. The moral of this is that it's wise to brace so that `else` is clearly unambiguous.

§30.18 Control constructs

Inform provides:

```
if ⟨condition⟩ ⟨block1⟩ [else ⟨block2⟩]
while ⟨condition⟩ ⟨block⟩
do ⟨block⟩ until ⟨condition⟩
for (⟨initialise⟩:⟨test⟩:⟨each time⟩)
objectloop (⟨variable⟩ in ⟨object⟩)
objectloop (⟨variable⟩ from ⟨object⟩)
objectloop (⟨variable⟩ near ⟨object⟩)
switch (⟨expression⟩) ⟨block⟩
break
jump ⟨label⟩
```

The `for` construct is essentially the same as that in C, except for the colons `:` (which in C would be semicolons). Its carries out the initial assignment(s), then executes the code for as long as the condition holds, executing the end assignment after each pass through the code. For instance,

```
for (i=1:i<=10:i++) print i, " ";
```

counts to 10. All three clauses are optional, and the empty condition is always true; multiple assignments can be made. For instance:

```
for (i=0,j=10:i<10:i++,j--) print i, " + ", j, " = ", i+j, "^";
for (::) print "Ha!^";
```

the latter laughing maniacally forever.

`break` breaks out of the current loop or switch (not quite the same as breaking out the current block of code because `if` statements don't count).

`objectloop` goes through the object tree, and is extremely useful. `from` means from the given object through its siblings; `in` means through all children of the given object, and `near` means through all children of the parent of the object. For instance, the following do the same thing:

```
objectloop (x in lamp) { ... }
for (x=child(lamp): x~=0: x=sibling(x)) { ... }
```

Note that the library creates a variable called `top_object` holding the highest existing object number: so a way to loop over every object defined in your own code is

```
for (i=selfobj+1: i<=top_object: i++) ...
```

since `selfobj` is the last of the objects created by the library.

• WARNING

When looping through the object tree, be careful if you are altering it at the same time. For instance, `objectloop (x in rucksack) remove x;` is likely to go horribly wrong – it's safer not to cut down a tree while actually climbing it. The safe way is to keep lopping branches off, `while (child(x)~=0) remove child(x);`

• WARNING

If you `jump` all the way from a routine in one object to a routine in another, then the library's `self` variable will not keep up with you. (In any case, jumping across routines is considered poor form.)

The `switch` statement takes the form:

```
switch (expression)
{   constant-value-1: ...
    constant-value-2: ...
    ...
    default: ...
}
```

the `default` clause being optional. It executes only the code which follows whichever value the expression has. There is no "case fall-through" as in C: so there's no need to keep using `break` instructions as in C. The `default` code, if given, is executed when none of the others match. Each ⟨constant-value⟩ is a comma-separated list of constants. For example,

```
switch(random(6))
{   1: "A snake slithers.";
    2 to 3: "An elephant bellows.";
    default: "The jungle is ominously silent.";
}
```

The value `c1 to c2` means "between `c1` and `c2`, inclusive".

Embedded routines can, if desired, take the similar form: switching on actions, which can conveniently be written without the need for a `##` in front of their names. For instance,

```
before
[;  Jump: "The ceiling is too low.";
    Look, Inv, Wait: ;
    default: "An invisible force holds you inactive.";
];
```

`default` is again optional.

△△ The switch is actually on the value of a variable called `sw__var`, which the library sets to the current action when calling `before` and `after`, and to its reason when calling `life`.

• EXERCISE 98

Write a routine to print out prime factorisations of numbers from 2 to 100.

§30.19 Actions

The commands

< ⟨Action⟩ [⟨first-object⟩ [⟨second-object⟩]] >
<< ⟨Action⟩ [⟨first-object⟩ [⟨second-object⟩]] >>

cause the given actions to take place. In the latter case, the current routine then returns 1, or true. The objects can be given as any expression, but the action must just be a name (with no initial `##`): unless it is bracketed, in which case any expression is allowed, e.g.

```
<< (MyAmazingAction(15)) magic_lamp>>;
```

Unpleasant things may happen if this expression doesn't evaluate to an action.

31 A summary of grammar

This section summarises the syntax more fully described in §§22-23.

A 'verb' is a set of possible initial words in keyboard command, which are treated synonymously (for example, "wear" and "don") together with a 'grammar'. A grammar is a list of 'lines' which the parser tries to match, one at a time, and accepts the first one which matches. The directive

Verb [`meta`] ⟨verb-word-1⟩ ... ⟨verb-word-n⟩ ⟨grammar⟩

creates a new verb. If it is said to be `meta` then it will count as 'out of the game': for instance "score" or "save". New synonyms can be added to an old verb with:

Verb ⟨new-word-1⟩ ... ⟨new-word-n⟩ `=` ⟨existing-verb-word⟩

An old verb can be modified with the directive

Extend [`only`] ⟨existing-word-1⟩ ... ⟨existing-word-n⟩ [⟨priority⟩] ⟨grammar⟩

If `only` is specified, the existing words given (which must all be from the same existing verb) are split off into a new independent copy of the verb. If not, the directive extends the whole existing verb. The priority can be `first` (insert this grammar at the head of the list), `last` (insert it at the end) or `replace` (throw away the old list and use this instead); the default is `last`.

A line is a list of 'tokens' together with the action generated if each token matches so that the line is accepted. The syntax of a line is

`*` ⟨token-1⟩ ⟨token-2⟩ ...⟨token-n⟩ `->` ⟨action⟩

where $0 \leq n \leq 6$. The action is named without initial `##` signs and if an action which isn't in the standard library set is named then an action routine (named with the action name followed by `Sub`) must be defined somewhere in the game.

A token matches a single particle of what has been typed. The possible tokens are:

`"`⟨word⟩`"`	that literal word only
`noun`	any object in scope

`held`	object held by the player
`multi`	one or more objects in scope
`multiheld`	one or more held objects
`multiexcept`	one or more in scope, except the other
`multiinside`	one or more in scope, inside the other
⟨attribute⟩	any object in scope which has the attribute
`creature`	an object in scope which is `animate`
`noun =` ⟨Routine⟩	any object in scope passing the given test
`scope =` ⟨Routine⟩	an object in this definition of scope
`number`	a number only
⟨Routine⟩	refer to this general parsing routine
`special`	*any* single word or number

For the `noun =` ⟨Routine⟩ token, the test routine must decide whether or not the object in the `noun` variable is acceptable and return true or false.

For the `scope =` ⟨Routine⟩ token, the routine must look at the variable `scope_stage`. If this is 1, then it must decide whether or not to allow a multiple object (such as "all") here and return true or false. If 2, then the routine may put objects into scope by calling either `PlaceInScope(obj)` to put just `obj` in, or `ScopeWithin(obj)` to put the contents of `obj` into scope. It must then return either true (to prevent any other objects from entering scope) or false (to let the parser put in all the usual objects). If `scope_stage=3`, it must print a suitable message to tell the player that this token was misunderstood.

A general parsing routine can match any text it likes. It should use `wn`, the variable holding the number of the word currently being parsed (counting from the verb being word 1) and the routine `NextWord()` to read the next word and move `wn` on by 1. The routine returns:

-1 if the user's input isn't understood,
0 if it's understood but doesn't refer to anything,
1 if there is a numerical value resulting, or
n if object n is understood.

In the case of a number, the actual value should be put into the variable `parsed_number`.On an unsuccessful match (returning -1) it doesn't matter what the final value of `wn` is. Otherwise it should be left pointing to the next thing *after* what the routine understood.

32 Compiler options and memory settings

> I was promised a horse, but what I got instead
> was a tail, with a horse hung from it almost dead.
>
> – Palladas of Alexandria (319?–400?)
>
> – translated by Tony Harrison (1937–)

The reader is warned that some details in this section are slightly different on different machines.

On most machines, Inform is run from the command line, by a command like

```
inform -xv5 balances
```

and simply typing `inform` will produce a good deal of help information about the command line options available. The command line syntax is

`inform` ⟨switches⟩ ⟨settings⟩ ⟨source file⟩ ⟨output file⟩

where only the ⟨source file⟩ is mandatory. By default, the full names to give the source and output files are derived in a way suitable for the machine Inform is running on: on a PC, for instance, `advent` may be understood as asking to compile `advent.inf` to `advent.z5`.

The switches are given in one or more groups, preceded by a minus sign `-` in the usual Unix command-line style. The current list of legal switches is:

```
a   list assembly-level instructions compiled
b   give statistics and/or line/object list in both passes
c   more concise error messages
d   contract double spaces after full stops in text
e   economy mode (slower): make use of declared abbreviations
E0  Archimedes-style error messages (current setting)
E1  Microsoft-style error messages
f   frequencies mode: show how useful abbreviations are
g   with debugging code: traces all function calls
h   print this information
i   ignore default switches set within the file
j   list objects as constructed
k   output Infix debugging information to "Game_Debug"
l   list all assembly lines
m   say how much memory has been allocated
n   print numbers of properties, attributes and actions
o   print offset addresses
p   give percentage breakdown of story file
q   keep quiet about obsolete usages
r   record all the text to "Game_Text"
s   give statistics
t   trace Z-code assembly
u   work out most useful abbreviations
```

```
  v3  compile to version-3 (Standard) story file
  v4  compile to version-4 (Plus) story file
  v5  compile to version-5 (Advanced) story file
  v6  compile to version-6 (graphical) story file
  v7  compile to version-7 (*) story file
  v8  compile to version-8 (*) story file
      (*) formats for very large games, requiring
          slightly modified game interpreters to play
  w   disable warning messages
  x   print # for every 100 lines compiled (in both passes)
  z   print memory map of the Z-machine
  T   enable throwback of errors in the DDE
```

(Thus, as long as your name doesn't have a 'y' in it, you can amuse yourself typing your name in as a switch and seeing what it does.) Note that these switches can also be selected by putting a `switches` directive into the source code before anything else, such as

```
Switches xdv5s;
```

The most useful switch is `v`, to choose the game format. For example, the above line is from the example game 'Advent', which is consequently compiled to an Advanced game. (Under Inform 5.5, this is the default anyway.) The recommended versions to use are `v3`, `v5` and `v8`.

Many of the remaining switches make Inform produce extra output, but do not affect its compilation:

`a b l m n t` Tracing options to help with maintaining Inform, or for debugging assembly language programs.
`o p s z` To print out information about the final game file: the `s` (statistics) option is particularly useful to keep track of how large the game is growing.
`c w q E T` In `c` mode, Inform does not quote whole source lines together with error messages; in `w` mode it suppresses warnings; in `T` mode, which is only present on the Acorn Archimedes, error throwback will occur in the 'Desktop Development Environment'. Inform 5.5 and later gives warnings about obsolete usages (such as `for i 1 to 5`), though it does compile them, unless `q` is set. Finally, `E` is provided since different error formats fit in better with debugging tools on different machines.
`f` Indicates roughly how many bytes the abbreviations saved.
`h` Prints out the help information (and is equivalent to just typing `inform`).
`j x` Makes Inform print out steady text to prove that it's still awake: on very slow machines this may be a convenience.
`k` Writes a "debugging information" file for the use of the Infix debugger (similarly, the filename is something suitable for the machine).
`r` Intended to help with proof-reading the text of a game: transcribes all of the text in double-quotes to the given file (whose name is something suitable for the machine).
`u` Tries to work out a good set of abbreviations to declare for your game, but *extremely slowly* (a matter of hours) and *consuming very much memory* (perhaps a megabyte).

This leaves three more switches which actually alter the game file which Inform would compile:

`d` Converts text like

```
"...with a mango.  You applaud..."
```

into the same with only a single space after the full stop, which will prevent an interpreter from displaying a spurious space at the beginning of a line when a line break happens to occur exactly after the full stop; this is to help typists who habitually double-space. Note that it does not contract double spaces after question or exclamation marks.

`e` Only in 'economy' mode does Inform actually process abbreviations, because this is seldom needed and slows the compiler by 10% or so; the game file should not play any differently if compiled this way, but will probably be shorter, if your choice of abbreviations was sensible.

`g` Makes Inform automatically compile trace-printing code on every function call; in play this will produce reams of text (several pages between each chance to type commands) but is sometimes useful. Note that in Inform 5.3 or later, this can be set on an individual command by writing `*` as its first local variable, without use of the `g` switch.

`i` Overrides any switches set by `switches` directives in the source code; so that the game can be compiled with different options without having to alter that source code.

One useful (optional) setting is the directory to take library files from: this should be preceded by a `+` sign.

Inform's memory management is about as flexible as it can be given that it has to run in some quite hostile environments. In particular, it is unable to increase the size of any stretch of memory once allocated, so if it runs out of anything it has to give up. If it does run out, it will produce an error message saying what it has run out of and how to provide more.

There are three main choices: `$small`, `$large` and `$huge`. (Which one is the default depends on the computer you use.) Even `$small` is large enough to compile all the example games, including 'Advent'. `$large` compiles almost anything and `$huge` has been used only for 'Curses' and 'Jigsaw' in their most advanced states, and even they hardly need it. A typical game, compiled with `$large`, will cause Inform to allocate about 350K of memory: and the same game about 100K less under `$small`. (These values will be rather lower if the computer Inform runs on has 16-bit integers.) In addition, Inform physically occupies about 170K (on my computer). Thus, the total memory consumption of the compiler at work will be between 4 to 500K.

Running

```
inform $list
```

will list the various settings which can be changed, and their current values. Thus one can compare small and large with:

```
inform $small $list
inform $large $list
```

If Inform runs out of allocation for something, it will generally print an error message like:

```
"Game", line 1320: Fatal error: The memory setting MAX_OBJECTS (which
is 200 at present) has been exceeded.  Try running Inform again with
$MAX_OBJECTS=<some-larger-number> on the command line.
```

and indeed

```
inform $MAX_OBJECTS=250 game
```

(say) will tell Inform to try again, reserving more memory for objects this time. Note that settings are made from left to right, so that for instance

```
inform $small $MAX_ACTIONS=200 ...
```

will work, but

```
inform $MAX_ACTIONS=200 $small ...
```

will not because the `$small` changes `MAX_ACTIONS` again. Changing some settings has hardly any effect on memory usage, whereas others are expensive to increase. To find out about, say, `MAX_VERBS`, run

```
inform $?MAX_VERBS
```

(note the question mark) which will print some very brief comments. Users of Unix, where `$` and `?` are special shell characters, will need to type

```
inform '$?list'        inform '$?MAX_VERBS'
```

and so on.

33 All the Inform error messages

Inform can produce about 250 different error messages. Since interpreters can in some cases crash horribly when given incorrect files, Inform never writes a file which caused an error, though it will permit files which incurred only warnings.

Fatal errors

To begin with, fatal errors (which stop Inform in its tracks) come in three kinds, the first containing only this one:

```
Too many errors: giving up
```

After 100 errors, Inform stops (in case it has been given the wrong source file altogether). Secondly, file input/output can go wrong. Most commonly, Inform has the wrong filename:

```
Couldn't open input file <filename>
Couldn't open output file <filename>
Couldn't open transcript file <filename>
Couldn't open debugging information file <filename>
Couldn't open temporary file 1 <filename>
Couldn't open temporary file 2 <filename>
Too many files have included each other: increase #define MAX_INCLUSION_DEPTH
```

(Temporary files are used (on most machines) for temporary storage space during compilation. They are removed afterwards.) The last error only occurs if 5 files all include each other. Increasing this `#define` means re-compiling Inform from its C source, a drastic measure. The remaining file-handling errors usually mean that the disc is full: something has gone wrong with an already-open file.

```
I/O failure: couldn't read from source file
I/O failure: couldn't write to temporary file 1
I/O failure: couldn't reopen temporary file 1
I/O failure: couldn't read from temporary file 1
I/O failure: couldn't write to temporary file 2
I/O failure: couldn't reopen temporary file 2
I/O failure: couldn't read from temporary file 2
I/O failure: couldn't write to story file
I/O failure: couldn't write to transcript file
I/O failure: can't write to debugging information file
```

The third class of fatal error is Inform running out of memory. It might fail drastically, having not enough memory to get started, as follows...

```
Couldn't allocate memory
Couldn't allocate memory for an array
```

There are four similar `hallocate` errors unique to the PC 'Quick C' port. More often memory will run out in the course of compilation, like so:

```
The memory setting <setting> (which is <value> at present) has been exceeded.
Try running Inform again with $<setting>=<some-larger-number> on the command line.
```

For details of memory settings, see §32 above.

Errors

There are a few conventions. Anything in double-quotes is a quotation from your source code; other strings are in single-quotes. A message like

```
Expected ... but found "..."
```

means that Inform expected something different from what it found; if it doesn't say what it found, this usually means it found nothing (i.e. the statement was incomplete). Messages in the form

```
No such ... as "..."
Not a ...: "..."
```

mean that a name is unrecognised in the former case (say, a typing error might produce this), or is recognised but means something else in the latter case (an attempt to use a routine where a property is expected would give such an error).

To begin with, the source-code format may go awry:

```
Too many tokens on line (note: to increase the maximum, set
    $MAX_TOKENS=some-bigger-number on the Inform command line)
Line too long (note: to increase the maximum length, set
```

```
    $BUFFER_LENGTH=some-bigger-number on the Inform command line)
Too much text for one pair of "s to hold
Too much text for one pair of 's to hold
Open quotes " expected for text but found <text>
Close quotes " expected for text but found <text>
```

(Usually BUFFER_LENGTH allows about 2000 characters per line.) When giving something (such as an object) an internal name, there are rules to be obeyed; for instance, you can't give an object the same name as a property already declared:

```
Symbol name expected
Symbol names are not permitted to start with an '_'
Symbol name is too long: <text>
Duplicated symbol name: <text>
```

At the top level, the most common "no such command" errors are

```
Expected an assignment, command, directive or opcode but found <text>
Unknown directive: <text>
Expected directive or '[' but found statement <text>
Expected directive or '[' but found opcode <text>
Expected directive or '[' but found <text>
```

which means Inform didn't even understand the first word of a line. Directives to Inform can produce the following errors:

```
All 32 attributes already declared (compile as Advanced game to get an extra 16)
All 48 attributes already declared
All 30 properties already declared (compile as Advanced game to get an extra 32)
All 62 properties already declared
Expected an attribute name after 'alias'
Expected a property name after 'alias'
'alias' incompatible with 'long'
'alias' incompatible with 'additive'
'alias' refers to undefined attribute <text>
'alias' refers to undefined property <text>
All 235 global variables already declared
An initialised global variable was defined only in Pass 1
A 'string' array can have at most 256 entries
Array entry too large for a byte: <text>
Expected ']' but found '['
Misplaced '['
Misplaced ']'
Missing array definition
Expected '->', '-->', 'string' or 'table' but found <text>
Expected '=', '->', '-->', 'string' or 'table' but found <text>
```

Use of `alias` is rare, except inside the library files. The last of these errors means that a global variable has been wrongly initialised, and a common cause of this is typing, say, `global trolls 5;` instead of `global trolls = 5;`.

```
'*' divider expected, but found <text>
```

```
No such token as <text>
Expected '=' after 'scope' but found <text>
Expected routine after 'scope=' but found <text>
Expected routine after 'noun=' but found <text>
'=' is only legal here as 'noun=Routine'
'->' clause missing
No such action routine as <text>
Not an action: <text>
Too many lines of grammar for verb: increase #define MAX_LINES_PER_VERB
There is no previous grammar for the verb <text>
Two different verb definitions refer to <text>
Expected 'replace', 'last' or 'first' but found <text>
```

These are the grammatical errors, the last three concerning `extend`ed verb definitions. Normally one gets 16 grammar lines per verb. It's probably better to write grammar more carefully (using routines to parse adjectives more carefully, for instance) than to exceed this, as the game parser will otherwise slow down. The object/class definition errors are largely self-explanatory:

```
Object/class definition finishes with ','
Two commas ',' in a row in object/class definition
No such attribute as <text>
No such class as <text>
Expected 'with', 'has' or 'class' in object/class definition but found <text>
Expected an (internal) name for object but found the string <text>
An object must be defined after the one which contains it: (so far)
    there is no such object as <text>
Not an object: <text>
No such property as <text>
Not a property: <text>
```

Miscellaneous errors complete the list produced by mostly-uncommon directives:

```
Expected ';' after 'include <file>' but found <text>
A 'switches' directive must come before constant definitions
Expected 'score' or 'time' after 'statusline' but found <text>
The serial number must be a 6-digit date in double-quotes
The version number must be 3 to 8: 3 for Standard games and 5 for Advanced
Expected 'on' or 'off' after 'font' but found <text>
Defaulted constants can't be strings
Must specify 0 to 3 variables in 'stub' routine
Expected 'full' or nothing after 'etrace' but found <text>
Too many abbreviations declared
All abbreviations must be declared together
It's not worth abbreviating <text>
Expected a 'string' value
No such directive as <text>
```

Conditional compilation happens as a result of one of `#IFDEF`, `#IFV3` or `#IFV5`. The former tests whether a constant is defined; the latter test for version-3 (Standard) games or version-5

(Advanced) games. An #IFNOT section is optional but the closing #ENDIF is compulsory. In these error messages #IF... means any of the three opening clauses.

```
'#IF...' nested too deeply: increase #define MAX_IFDEF_DEPTH
'#ENDIF' without matching '#IF...'
'#IFNOT' without matching '#IF...'
Two '#IFNOT's in the same '#IF...'
End of file reached inside '#IF...'
```

Routines begin with a [and some local variables and end with]:

```
Routine has more than 15 local variables
The earliest-defined routine is not allowed to have local variables
Expected local variable but found ',' or ':' (probably the ';' after the
    '[ ...' line was forgotten)
Misplaced ']'
Comma ',' after ']' can only be used inside object/class definitions
Expected ',' or ';' after ']' but found <text>
Expected ';' after ']' but found <text>
```

The error messages for expressions are fairly simple. Note, however, that one is not allowed to type, say, `lamp.number++;` but must instead write `lamp.number = lamp.number + 1;` The ++ and -- operators can only be applied to variables, not properties or array entries.

```
Expected condition but found expression
Unexpected condition
Expected an assignment but found <text>
Expected an assignment but found an expression
Attempt to use void as a value
Attempt to use an assignment as a value
Attempt to use a condition as a value
Operator has too few arguments
Operator has too many arguments
'++' and '--' can only apply directly to variables
At most three values can be separated by 'or'
'or' can only be used with the conditions '==' and '~='
Too many brackets '(' in expression
Brackets '(' too deeply nested
Missing bracket ')' in function call
Spurious comma ','
Misplaced comma ','
Wrong number of arguments to system function
'children' takes a single argument
'youngest' takes a single argument
'elder' takes a single argument
'indirect' takes at least one argument
Type mismatch in argument <text>
A function may be called with at most 3 arguments
Malformed statement 'Function(...);'
Spurious terms after function call
Spurious terms after assignment
Spurious terms after expression
```

Next, constants. Note that dictionary words cannot start with a non-alphabetic character, which means that Infocom-style 'debugging verbs' which traditionally begin with a # are not allowed.

```
No such variable as <text>
No such constant as <text>
Not a constant: <text>
Reserved word as constant: <text>
No such routine as <text>
Dictionary words must begin with a letter of the alphabet
Dictionary word not found for constant <text>
```

Loop constructs: note that 'old-style' `for` loops are a rather obsolete form (e.g., `for i 1 to 10`), and the more flexible style `for (i=1:i<=10:i++)` is now preferred.

```
'if' statement with more than one 'else'
'else' attached to a loop block
'for' loops too deeply nested
':' expected in 'for' loop
Second ':' expected in 'for' loop
Concluding ')' expected in 'for' loop
'to' missing in old-style 'for' loop
'to' expected in old-style 'for' loop
Final value missing in old-style 'for' loop
'{' required after an old-style 'for' loop
Old-style 'for' loops must have simple final values
Open bracket '(' expected in 'objectloop'
'objectloop' must be 'from', 'near' or 'in' something
Close bracket ')' expected in 'objectloop'
Braces '{' are compulsory unless the condition is bracketed
Unmatched '}' found
Brace mismatch in previous routine
Expected bracketed expression but found <text>
Switch value too long or a string: perhaps a statement
    accidentally ended with a comma?
A 'default' clause must come last
A 'default' rule must come last
Two 'default' clauses in 'switch'
Two 'default' rules given
Expected label after 'jump' but found <text>
'do' without matching 'until'
'until' without matching 'do'
```

Inform checks the level of braces when a routine closes so that it can recover as quickly as possible from a mismatch; this last error means at least one brace is still open when the routine finishes.

```
Expected some attributes to 'give'
Expected 'to <object>' in 'move'
Expected 'to' in 'move' but found <text>
Expected ',' in 'print' list but found <text>
Expected 'style' to be 'roman', 'bold', 'underline' or 'reverse' but found <text>
Expected a parse buffer for 'read'
```

```
Expected some properties to 'write'
The object to 'write' must be a variable or constant
Expected property value to 'write'
Expected 'byte' or 'word' in 'put'
Expected 'byte' or 'word' in 'put' but found <text>
```

These are miscellaneous commands and put and write are quite obsolete. Only action commands like <Take brass_lantern> remain:

```
Action name too long or a string: perhaps a statement accidentally ended
    with a comma?
The longest action command allowed is '<Action noun second>'
Angle brackets '>' do not match
Missing '>' or '>>'
Action given in constant does not exist
Action name over 60 characters long: <text>
Expected ':' (after action) but found <text>
```

The first of these may be caused by something like:

```
           "You load the crossbow bolt.",
         Drop:  "The bolt falls to the floor with a thump.";
```

where a comma has been typed instead of a semicolon after the first string, so that Inform thinks you are giving a rule for two actions, one being Drop and the other apparently called "You load the crossbow bolt.".

Internal and assembler errors

By now we have descended to the ninth circle of Inform: the assembler. These errors are fairly unmysterious (if only because they seldom happen), but sometimes one is told something odd like

```
      No such label as _f456
```

which is caused by Inform failing to recover properly from a previous brace mismatch error. Just ignore this and fix the earlier error (which will also have been reported). The "no such variable" error is occasionally seen when an unknown variable is first referred to by being written to.

```
No such assembly opcode as <text>
Too many arguments
Can't store to that (no such variable)
Branch too far forward: use '?'
No such return condition for branch
Can't branch to a routine, only to a label
No such label as <text>
Not a label: <text>
```

To conclude with, there are a few internal error messages:

```
A label has moved between passes because of a low-level error just before
    (perhaps an improper use of a routine address as a constant)
Object has altered in memory usage between passes: perhaps an attempt to
    use a routine name as value of a small property
Duplicated system symbol name <text>
Internal error - unknown directive code
Internal error - unknown compiler code
```

The last three should not happen. The first two occasionally do, and cause some confusion. Inform performs two tests regularly as a safeguard to make sure that code has not come out substantially different in its two passes. The first failure occurs in code, the second for object/class definitions. Whatever caused this should be something unusual, low-level and just before the error occurred: if you get these errors with innocent high-level code, then probably Inform should provide a suitable error message, so please e-mail the author with a sample of offending code.

Warnings

Inform can produce any number of warnings without shutting down. Note that Inform tries to give only warnings when it hits Advanced-game features in what is to be a Standard game, for the sake of portability. Nevertheless it is probably better to use `#IFV3` and `#IFV5` clauses around any pieces of code which are to be different in these two versions (if, indeed, you want two different versions).

```
Local variable unused: <text>
Since it is defined before inclusion of the library, game-play will begin
    not at 'Main' but at the routine <text>
Ignoring Advanced-game status routine
Ignoring this Advanced-game command
Missing ','?  Property data seems to contain the property name <text>
Standard-game limit of 8 bytes per property exceeded (use Advanced to get
    64), so truncating property <text>
Ignoring Advanced-game opcode <text>
Ignoring Standard-game opcode <text>
```

In addition, Inform 5.5 makes rather more thorough syntax checks than its predecessors and (unless the -q switch is set) may produce the following "obsolete usage" warnings:

```
'#a$Action' is now superceded by '##Action'
'#w$word' is now superceded by ''word''
'#n$word' is now superceded by ''word''
All properties are now automatically 'long'
Use the ^ character for an apostrophe in a dictionary word, e.g. 'peter^s'
Use 'word' as a constant dictionary address
Modern 'for' syntax is 'for (start:condition:update)'
Use 'array->byte=value' or 'array-->word=value'
Write properties using the '.' operator
'If' conditions should be bracketed
Assembly-language lines should begin with '@'
Directives inside routines should begin with '#'
An object should only have one internal name
Use '->' instead of 'data'
Use '->' instead of 'initial'
Use '->' instead of 'initstr'
```

Chapter VI: Library Reference

34 The attributes

Here is a concise account of all the normal rules concerning all the library's attributes, except that: rules about how the parser sorts out ambiguities are far too complicated to include here, but should not concern designers anyway; and the definitions of 'scope' and 'darkness' are given in §§24 and 13 respectively. These rules are the result of pragmatism and compromise, but are all easily modifiable.

`absent` A 'floating object' (one with a `found_in` property, which can appear in many different rooms) which is `absent` will in future no longer appear in the game. Note that you cannot make a floating object disappear merely by giving it `absent`, but must explicitly `remove` it as well.

`animate` "Is alive (human or animal)." Can be spoken to in "richard, hello" style; matches the `creature` token in grammar; picks up "him" or "her" (according to gender) rather than "it", likewise "his"; an object the player is changed into becomes `animate`; some messages read "on whom", etc., instead of "on which"; can't be taken; its subobjects "belong to" it rather than "are part of" it; messages don't assume it can be "touched" or "squeezed" as an ordinary object can; the actions `Attack`, `ThrowAt` are diverted to `life` rather than rejected as being 'futile violence'.

`clothing` "Can be worn."

`concealed` "Concealed from view but present." The player object has this; an object which was the player until `ChangePlayer` happened loses this property; a `concealed door` can't be entered; does not appear in room descriptions.

`container` Affects scope and light; object lists recurse through it if `open` (or `transparent`); may be described as closed, open, locked, empty; a possession will give it a `LetGo` action if the player tries to remove it, or a `Receive` if something is put in; things can be taken or removed from it, or inserted into it, but only if it is `open`; likewise for "transfer" and "empty"; room descriptions describe using `when_open` or `when_closed` if given; if there is no defined `description`, an `Examine` causes the contents to be searched (i.e. written out) rather than a message "You see nothing special about..."; `Search` only reveals the contents of `container`s, otherwise saying "You find nothing".

`door` "Is a door or bridge." Room descriptions describe using `when_open` or `when_closed` if given; and an `Enter` action becomes a `Go` action. If a `Go` has to go through this object, then: if `concealed`, the player "can't go that way"; if not `open`, then the player is told either that this cannot be ascended or descended (if the player tried "up" or "down"), or that it is in the way (otherwise); but if neither, then its `door_to` property is consulted to see where it leads; finally, if this is zero, then it is said to "lead nowhere" and otherwise the player actually moves to the location.

`edible` "Can be eaten" (and thus removed from game).

`enterable` Affects scope and light; only an `enterable` on the floor can be entered. If an `enterable` is also a `container` then it can only be entered or exited if it is `open`.

`female`	Only applies to `animates` (and cannot have a `found_in` list for arcane reasons), and only affects the parser's use of pronouns: it says "her" is appropriate but "him" and "his" are not.
`general`	A general-purpose attribute, defined by the library but never looked at or altered by it. This is left free to mean something different for each object: often used by programmers for something like "the puzzle for this object has been solved".
`light`	"Is giving off light." (See §13.) Also: the parser understands "lit", "lighted", "unlit" using this; inventories will say "(providing light)" of it, and so will room descriptions if the current `location` is ordinarily dark; it will never be automatically put away into the player's `SACK_OBJECT`, as it might plausibly be inflammable or the main light source.
`lockable`	Can be locked or unlocked by a player holding its key object, which is given by the property `with_key`; if a `container` and also `locked`, may be called "locked" in inventories.
`locked`	Can't be opened. If a `container` and also `lockable`, may be called "locked" in inventories.
`moved`	"Has been or is being held by the player." Objects (immediately) owned by the player after `Initialise` has run are given it; at the end of each turn, if an item is newly held by the player and is `scored`, it is given `moved` and `OBJECT_SCORE` points are awarded; an object's `initial` message only appears in room descriptions if it is un`moved`.
`on`	"Switched on." A `switchable` object with this is described by `with_on` in room descriptions; it will be called "switched on" by `Examine`.
`open`	"Open door or container." Affects scope and light; lists (such as inventories) recurse through an `open container`; if a `container`, called "open" by some descriptions; things can be taken or removed from an `open container`; similarly inserted, transferred or emptied. A `container` can only be entered or exited if it is both `enterable` and `open`. An `open door` can be entered. Described by `when_open` in room descriptions.
`openable`	Can be opened or closed, unless `locked`.
`proper`	Its short name is a proper noun, and never preceded by "the" or "The". The player's object must have this (so something changed into will be given it).
`scenery`	Not listed by the library in room descriptions; "not portable" to be taken; "you are unable to" pull, push, or turn it.
`scored`	The player gets `OBJECT_SCORE` points for picking it up for the first time; or, if a room, `ROOM_SCORE` points for visiting it for the first time.
`static`	"Fixed in place" if player tries to take, remove, pull, push or turn.
`supporter`	"Things can be put on top of it." Affects scope and light; object lists recurse through it; a possession will give it a `LetGo` action if the player tries to remove it, or a `Receive` if something is put in; things can be taken or removed from it, or put on it; likewise for transfers; a player inside it is said to be "on" rather than "in" it; room descriptions list its contents in separate paragraphs if it is itself listed.
`switchable`	Can be switched on or off; listed as such by `Examine`; described using `when_on` or `when_off` in room descriptions.
`talkable`	Player can talk to this object in "thing, do this" style. This is useful for microphones and the like, when `animate` is inappropriate.
`transparent`	"Contents are visible." Affects scope and light; a `transparent` container is treated as if it were `open` for printing of contents.

`visited`	"Has been or is being visited by the player." Given to a room immediately after a `Look` first happens there: if this room is `scored` then `ROOM_SCORE` points are awarded. Affects whether room descriptions are abbreviated or not.
`workflag`	Temporary flag used by Inform internals, also available to outside routines; can be used to select items for some lists printed by `WriteListFrom`.
`worn`	"Item of clothing being worn." Should only be an object being immediately carried by player. Affects inventories; doesn't count towards the limit of `MAX_CARRIED`; won't be automatically put away into the `SACK_OBJECT`; a `Drop` action will cause a `Disrobe` action first; so will `PutOn` or `Insert`.

Note that very few attributes sensibly apply to rooms: only really `light`, `scored` and `visited`, together with `general` if you choose to use it. Note also that an object cannot be both a `container` and a `supporter`; and that the old attribute `autosearch`, which was in earlier releases, has been withdrawn as obsolete.

35 The properties

The following table lists every library-defined property. The banner headings give the name, what type of value makes sense and the default value (if other than 0). The symbol ⊕ means "this property is additive" so that inherited values from class definitions pile up into a list, rather than wipe each other out. Recall that 'false' is the value 0 and 'true' the value 1.

`n_to, s_to, e_to, w_to, ...` — Room, object or routine

For rooms These twelve properties (there are also `ne_to`, `nw_to`, `se_to`, `sw_to`, `in_to`, `out_to`, `u_to` and `d_to`) are the map connections for the room. A value of 0 means "can't go this way". Otherwise, the value should either be a room or a `door` object: thus, `e_to` might be set to `crystal_bridge` if the direction "east" means "over the crystal bridge".
Routine returns The room or object the map connects to; or 0 for "can't go this way"; or 1 for "can't go this way; stop and print nothing further".
Warning Do not confuse the direction properties `n_to` and so on with the twelve direction objects, `n_obj` et al.

`add_to_scope` — List of objects or routine

For objects When this object is in scope, so are all those listed, or all those nominated by the routine. A routine given here should call `PlaceInScope(obj)` to put `obj` in scope.
No return value

`after` — Routine `NULL` ⊕

Receives actions after they have happened, but before the player has been told of them.
For rooms All actions taking place in this room.
For objects All actions for which this object is `noun` (the first object specified in the command); and all fake actions for it.
Routine returns False to continue (and tell the player what has happened), true to stop here (printing nothing).

The `Search` action is a slightly special case. Here, `after` is called when it is clear that it would be sensible to look inside the object (e.g., it's an open container in a light room) but before the contents are described.

`article`	String or routine `"a"`

For objects Indefinite article for object or routine to print one.
No return value

`before`	Routine `NULL` ⊕

Receives advance warning of actions (or fake actions) about to happen.
For rooms All actions taking place in this room.
For objects All actions for which this object is `noun` (the first object specified in the command); and all fake actions, such as `Receive` and `LetGo` if this object is the container or supporter concerned.
Routine returns False to continue with the action, true to stop here (printing nothing).
First special case: A vehicle object receives the `Go` action if the player is trying to drive around in it. In this case:
Routine returns 0 to disallow as usual; 1 to allow as usual, moving vehicle and player; 2 to disallow but do (and print) nothing; 3 to allow but do (and print) nothing. If you want to move the vehicle in your own code, return 3, not 2: otherwise the old location may be restored by subsequent workings.
Second special case: in a `PushDir` action, the `before` routine must call `AllowPushDir()` and then return true in order to allow the attempt (to push an object from one room to another) to succeed.

`cant_go`	String or routine `"You can't go that way."`

For rooms Message, or routine to print one, when a player tries to go in an impossible direction from this room.
No return value

`capacity`	Number or routine `100`

For objects Number of objects a `container` or `supporter` can hold.
For the player-object Number of things the player can carry (when the player is this object); the default player object (`selfobj`) has `capacity` initially set to the constant `MAX_CARRIED`.

`daemon`	Routine `NULL`

This routine is run each turn, once it has been activated by a call to `StartDaemon`, and until stopped by a call to `StopDaemon`.
Warning The same object cannot have both a `daemon` and a `time_out` property.

`describe`	Routine `NULL` ⊕

For objects Called when the object is to be described in a room description, before any paragraph break (i.e., skipped line) has been printed. A sometimes useful trick is to print nothing in this routine and return true, which makes an object 'invisible'.
For rooms Called before a room's long ("look") description is printed.
Routine returns False to describe in the usual way, true to stop printing here.

`description`	String or routine

For objects The `Examine` message, or a routine to print one out.
For rooms The long ("look") description, or a routine to print one out.
No return value

`door_dir` — Direction property or routine

For compass objects When the player tries to go in this direction, e.g., by typing the name of this object, then the map connection tried is the value of this direction property for the current room. For example, the `n_obj` "north" object normally has `door_dir` set to `n_to`.
For objects The direction that this `door` object goes via (for instance, a bridge might run east, in which case this would be set to `e_to`).
Routine returns The direction property to try.

`door_to` — Room or routine

For objects The place this door object leads to. A value of 0 means "leads nowhere".
Routine returns The room. Again, 0 (or false) means "leads nowhere". Further, 1 (or true) means "stop the movement action immediately and print nothing further".

`each_turn` — String or routine `NULL` ⊕

String to print, or routine to run, at the end of each turn in which the object is in scope (after all timers and daemons for that turn have been run).
No return value

`found_in` — List of rooms or routine

This object will be found in all of the listed rooms, or if the routine says so, unless it has the attribute `absent`. If an object in the list is not a room, it means "present in the same room as this object".
Routine returns True to be present, otherwise false. The routine can look at the current `location` in order to decide.
Warning This property is only looked at when the player changes rooms.

`grammar` — Routine

For `animate` *or* `talkable` *objects* This is called when the parser has worked out that the object in question is being spoken to, and has decided the `verb_word` and `verb_wordnum` (the position of the verb word in the word stream) but hasn't yet tried any grammar. The routine can, if it wishes, parse past some words (provided it moves `verb_wordnum` on by the number of words it wants to eat up).
Routine returns False to carry on as usual; true to indicate that the routine has parsed the entire command itself, and set up `action`, `noun` and `second` to the appropriate order; or a dictionary value for a verb, such as `'take'`, to indicate "parse the command from this verb's grammar instead"; or minus such a value, e.g. `-'take'`, to indicate "parse from this verb and then parse the usual grammar as well".

`initial` — String or routine

For objects The description of an object not yet picked up, used when a room is described; or a routine to print one out.
For rooms Printed or run when the room is arrived in, either by ordinary movement or by `PlayerTo`.
Warning If the object is a `door`, or a `container`, or is `switchable`, then use one of the `when_` properties rather than `initial`.
No return value

`invent` Routine

This routine is for changing an object's inventory listing. If provided, it's called twice, first with the variable `inventory_stage` set to 1, second with it set to 2. At stage 1, you have an entirely free hand to print a different inventory listing.
Routine returns Stage 1: False to continue; true to stop here, printing nothing further about the object or its contents.
At stage 2, the object's indefinite article and short name have already been printed, but messages like " (providing light)" haven't. This is an opportunity to add something like " (almost empty)".
Routine returns Stage 2: False to continue; true to stop here, printing nothing further about the object or its contents.

`life` Routine `NULL` ⊕

This routine holds rules about `animate` objects, behaving much like `before` and `after` but only handling the person-to-person events:

```
Attack Kiss WakeOther ThrowAt Give Show Ask Tell Answer Order
```

See §12, and see also the properties `orders` and `grammar`.
Routine returns True to stop and print nothing, false to resume as usual (for example, printing "Miss Gatsby has better things to do.").

`list_together` Number, string or routine

For objects Objects with the same `list_together` value are grouped together in object lists (such as inventories, or the miscellany at the end of a room description). If a string such as `"fish"` is given, then such a group will be headed with text such as `"five fish"`.
A routine, if given, is called at two stages in the process (once with the variable `inventory_stage` set to 1, once with it set to 2). These stages occur before and after the group is printed; thus, a preamble or postscript can be printed. Also, such a routine may change the variable `c_style` (which holds the current list style). On entry, the variable `parser_one` holds the first object in the group, and `parser_two` the current depth of recursion in the list. Applying `x=NextEntry(x,parser_two);` moves `x` on from `parser_one` to the next item in the group. Another helpful variable is `listing_together`, set up to the first object of a group being listed (or to 0 whenever no group is being listed).
Routine returns Stage 1: False to continue, true not to print the group's list at all.
Routine returns Stage 2: No return value.

`orders` Routine

For `animate` *or* `talkable` *objects* This carries out the player's orders (or doesn't, as it sees fit): it looks at `actor`, `action`, `noun` and `second` to do so. Unless this object is the current player, `actor` is irrelevant (it is always the player) and the object is the person being ordered about.
If the player typed an incomprehensible command, like "robot, og sthou", then the action is `NotUnderstood` and the variable `etype` holds the parser's error number.
If this object is the current player then `actor` is the person being ordered about. `actor` can either be this object – in which case an action is being processed, because the player has typed an ordinary command – or can be some other object, in which case the player has typed an order. See §15 for how to write `orders` routines in these cases.
Routine returns True to stop and print nothing further; false to continue. (Unless the object is the current player, the `life` routine's `Order` section gets an opportunity to meddle next; after that, Inform gives up.)

`name`	List of dictionary words ⊕

For objects A list of dictionary words referring to this object.
Warning The `parse_name` property of an object may take precedence over this, if present.
For rooms A list of words which the room understands but which refer to things which "do not need to be referred to in this game"; these are only looked at if all other attempts to understand the player's command have failed.
Warning Uniquely in Inform syntax, these dictionary words are given in double quotes `"thus"`, whereas in all other circumstances they would be `'thus'`. This means they can safely be only one letter long without ambiguity.

`number`	Any value

A general purpose property left free: conventionally holding a number like "number of turns' battery power left".
For compass objects Note that the standard compass objects defined by the library all provide a `number` property, in case this might be useful to the designer.
For the player-object Exception: an object to be used as a player-object must provide one of these, and musn't use it for anything.

`parse_name`	Routine

For objects To parse an object's name (this overrides the `name` but is also used in determining if two objects are describably identical). This routine should try to match as many words as possible in sequence, reading them one at a time by calling `NextWord()`. (It can leave the "word marker" variable `wn` anywhere it likes).
Routine returns 0 if the text didn't make any sense at all, −1 to make the parser resume its usual course (looking at the `name`), or the number of words in a row which successfully matched. In addition to this, if the text matched seems to be in the plural (for instance, a blind mouse object reading `blind mice`), the routine can set the variable `parser_action` to the value `##PluralFound`. The parser will then match with all of the different objects understood, rather than ask a player which of them is meant.
A `parse_name` routine may also (voluntarily) assist the parser by telling it whether or not two objects which share the same `parse_name` routine are identical. (They may share the same routine if they both inherit it from a class.) If, when it is called, the variable `parser_action` is set to `##TheSame` then this is the reason. It can then decide whether or not the objects `parser_one` and `parser_two` are indistinguishable.
Routine returns −1 if the objects are indistinguishable, −2 if not.

`plural`	String or routine

For objects The plural name of an object (when in the presence of others like it), or routine to print one; for instance, a wax candle might have `plural` set to `"wax candles"`.
No return value

`react_after`	Routine

For objects Acts like an `after` rule, but detects any actions in the vicinity (any actions which take place when this object is in scope).
Routine returns True to print nothing further; false to carry on.

`react_before`	Routine

For objects Acts like a `before` rule, but detects any actions in the vicinity (any actions which take place when this object is in scope).
Routine returns True to stop the action, printing nothing; false to carry on.

`short_name`	Routine

For objects The short name of an object (like “brass lamp”), or a routine to print it.
Routine returns True to stop here, false to carry on by printing the object’s ‘real’ short name (the string given at the head of the object’s definition). It’s sometimes useful to print text like `"half-empty "` and then return false.

`time_left`	Number

Number of turns left until the timer for this object (if set, which must be done using `StartTimer`) goes off. Its initial value is of no significance, as `StartTimer` will write over this, but a timer object must provide the property. If the timer is currently set, the value 0 means “will go off at the end of the current turn”, the value 1 means “...at the end of next turn” and so on.

`time_out`	Routine `NULL`

Routine to run when the timer for this object goes off (having been set by `StartTimer` and not in the mean time stopped by `StopTimer`).
Warning The same object cannot have both a `daemon` and a `time_out`.
Warning A timer object must also provide a `time_left` property.

`when_closed`	String or routine

For objects Description, or routine to print one, of something closed (a `door` or `container`) in a room’s long description.
No return value

`when_open`	String or routine

For objects Description, or routine to print one, of something open (a `door` or `container`) in a room’s long description.
No return value

`when_on`	String or routine

For objects Description, or routine to print one, of a `switchable` object which is currently switched on, in a room’s long description.
No return value

`when_off`	String or routine

For objects Description, or routine to print one, of a `switchable` object which is currently switched off, in a room’s long description.
No return value

`with_key`	Object `nothing`

The key object needed to lock or unlock this `lockable` object. A player must explicitly name it as the key being used and be holding it at the time. The value `nothing`, or 0, means that no key fits (though this is not made clear to the player, who can try as many as he likes).

36 Library-defined objects and routines

The library defines the following special objects:

`compass`	To contain the directions. A direction object provides a `door_dir` property, and should have the `direction` attribute. A compass direction with `enterable`, if there is one (which there usually isn't), will have an `Enter` action converted to `Go`.
`n_obj, ...`	Both the object signifying the abstract concept of 'northness', and the 'north wall' of the current room. (Thus, if a player types "examine the north wall" then the action `Examine n_obj` will be generated.) Its `door_dir` property holds the direction property it corresponds to (`n_to`). The other such objects are `s_obj`, `e_obj`, `w_obj`, `ne_obj`, `nw_obj`, `se_obj`, `sw_obj`, `u_obj`, `d_obj`, `in_obj` and `out_obj`. Note that the parser understands "ceiling" to refer to `u_obj` and "floor" to refer to `d_obj`. (`in_obj` and `out_obj` differ slightly, because "in" and "out" are verbs with other effects in some cases; these objects should not be removed from the `compass`.)
`thedark`	A pseudo-room representing 'being in darkness'. `location` is then set to this room, but the player object is not moved to it. Its `description` can be changed to whatever "It is dark here" message is desired.
`selfobj`	The default player-object. Code should never refer directly to `selfobj`, but only to `player`, a variable whose value is usually indeed `selfobj` but which might become `green_frog` if the player is transformed into one.

The following routines are defined in the library and available for public use:

`Achieved(task)`	Indicate the `task` is achieved (which only awards score the first time).
`AddToScope(obj)`	Used in an `add_to_scope` routine of an object to add another object into scope whenever the first is in scope.
`AllowPushDir()`	Signal that an attempt to push an object from one place to another should be allowed.
`CDefArt(obj)`	Print the capitalised definite article and short name of `obj`. Equivalent to `print (The) obj;`.
`ChangeDefault(p,v)`	Changes the default value of property `p`. (But this won't do anything useful to `name`.)
`ChangePlayer(obj,flag)`	Cause the player at the keyboard to play as the given object, which must have a `number` property supplied. If the `flag` is set to 1, then subsequently print messages like "(as Ford Prefect)" in room description headers. This routine, however, prints nothing itself.
`DefArt(obj)`	Print the definite article and short name of `obj`. Equivalent to `print (the) obj;`.
`DoMenu(text,R1,R2)`	Produce a menu, using the two routines given.
`EnglishNumber(x)`	Prints out `x` in English (e.g., "two hundred and seventy-seven").
`HasLightSource(obj)`	Returns true if `obj` 'has light'.
`InDefArt(obj)`	Print the indefinite article and short name of `obj`. Equivalent to `print (a) obj;`.
`Locale(obj,tx1,tx2)`	Prints out the paragraphs of room description which would appear if `obj` were the room: i.e., prints out descriptions of objects in `obj` according to the usual rules. After describing the objects which have their own paragraphs, a list is given of the remaining ones. The string

	`tx1` is printed if there were no previous paragraphs, and the string `tx2` otherwise. (For instance, you might want "On the ledge you can see" and "On the ledge you can also see".) After the list, nothing else is printed (not even a full stop) and the return value is the number of objects in the list (possibly zero).
`LoopOverScope(R,actor)`	Calls routine `R(obj)` for each object `obj` in scope. `actor` is optional: if it's given, then scope is calculated for the given actor, not the player.
`NextWord()`	Returns the next dictionary word in the player's input, moving the word number `wn` on by one. Returns 0 if the word is not in the dictionary or if the word stream has run out.
`NextWordStopped()`	As `NextWord`, but returning -1 when the word stream has run out.
`NounDomain(o1,o2,type)`	This routine is one of the keystones of the parser: the objects given are the domains to search through when parsing (almost always the location and the actor) and the `type` indicates a token. The only tokens safely usable are: 0: `noun`, 1: `held` and 6: `creature`. The routine parses the best single object name it can from the current position of `wn`. It returns 0 (no match), an object number or the constant `REPARSE_CODE` (to indicate that it had to ask a clarifying question: this reconstructed the input drastically and the parser must begin all over again). `NounDomain` should only be used by general parsing routines and these should always return `REPARSE_CODE` if it does. Note that all of the usual scope and name-parsing rules apply to the search performed by `NounDomain`.
`OffersLight(obj)`	Returns true if `obj` 'offers light'.
`PlaceInScope(obj)`	Puts `obj` into scope for the parser.
`PlayerTo(place,flag)`	Move the player to `place`. Unless `flag` is given and is 1, describe the player's surroundings.
`PrintShortName(obj)`	Print the short name of `obj`. (This is protected against `obj` having a meaningless value.) Equivalent to `print (name) obj;`.
`ScopeWithin(obj)`	Puts the contents of `obj` into scope, recursing downward according to the usual scope rules.
`SetTime(time,rate)`	Set the game clock (a 24-hour clock) to the given `time` (in seconds since the start of the day), to run at the given `rate` r: $r = 0$ means it does not run, if $r > 0$ then r seconds pass every turn, if $r < 0$ then $-r$ turns pass every second.
`StartDaemon(obj)`	Makes the daemon of `obj` active, so that its `daemon` routine will be called every turn.
`StartTimer(obj,time)`	Starts the timer of `obj`, set to go off in `time` turns, at which time its `time_out` routine will be called (it must provide a `time_left` property).
`StopDaemon(obj)`	Makes the daemon of `obj` inactive, so that its `daemon` routine is no longer called.
`StopTimer(obj)`	Stops the timer of `obj`, so that it won't go off after all.
`TestScope(obj,actor)`	Returns true if `obj` is in scope; otherwise false. `actor` is optional: if it's given, then scope is calculated for the given actor, not the player.
`TryNumber(wordnum)`	Tries to parse the word at `wordnum` as a number (recognising decimal numbers and English ones from "one" to "twenty"), returning -1000 if it fails altogether, or the number. Values exceeding 10000 are rounded

	down to 10000.
`UnsignedCompare(a,b)`	Returns 1 if $a > b$, 0 if $a = b$ and $a < b$, regarding a and b as unsigned numbers between 0 and 65535 (or `$ffff`). (The usual $>$ condition performs a signed comparison.)
`WordAddress(n)`	Returns the byte array containing the raw text of the n-th word in the word stream.
`WordLength(n)`	Returns the length of the raw text of the n-th word in the word stream.
`WriteListFrom(obj,s)`	Write a list of `obj` and its siblings, with the style being `s` (a bitmap of options).
`YesOrNo()`	Assuming that a question has already been printed, wait for the player to type "yes" or "no", returning true or false accordingly.
`ZRegion(value)`	Works out the type of `value`, if possible. Returns 1 if it's a valid object number, 2 if a routine address, 3 if a string address and 0 otherwise.

37 Entry points and meaningful constants

Entry points are routines which you can provide, if you choose to, and which are called by the library routines to give you the option of changing the rules. All games *must* define an `Initialise` routine, which is obliged to set the `location` variable to a room; the rest are optional.

`AfterLife`	When the player has died (a condition signalled by the variable `deadflag` being set to a non-zero value other than 2, which indicates winning), this routine is called: by setting `deadflag=0` again it can resurrect the player.
`AfterPrompt`	Called just after the prompt is printed: therefore, called after all the printing for this turn is definitely over. A useful opportunity to use `box` to display quotations without them scrolling away.
`Amusing`	Called to provide an 'afterword' for players who have won: for instance, it might advertise some features which a successful player might never have noticed. (But only if you have defined the constant `AMUSING_PROVIDED` in your own code.)
`BeforeParsing`	Called after the parser has read in some text and set up the `buffer` and `parse` tables, but has done nothing else yet (except to set the word marker `wn` to 1). The routine can do anything it likes to these tables, and can leave the word marker anywhere; there is no meaningful return value.
`ChooseObjects(obj,c)`	When `c` is 0, the parser is processing an "all" and has decided to exclude `obj` from it; when `c` is 1, it has decided to include it. Returning 1 forces inclusion, returning 2 forces exclusion and returning 0 lets the parser's decision stand. When `c` is 2, the parser wants help in resolving an ambiguity: using the `action_to_be` variable the routine must decide how appropriate `obj` is for the given action and return a score of 0 to 9 accordingly. See §25.

`DarkToDark`	Called when a player goes from one dark room into another one; a splendid excuse to kill the player off.
`DeathMessage`	Prints up "You have died" style messages, for `deadflag` values of 3 or more. (If you choose ever to set `deadflag` to such.)
`GamePostRoutine`	A kind of super-`after` rule, which applies to all actions in the game, whatever they are: use only in the last resort.
`GamePreRoutine`	A kind of super-`before` rule, which applies to all actions in the game, whatever they are: use only in the last resort.
`Initialise`	A compulsory routine, which must set `location` and is convenient for miscellaneous initialising, perhaps for random settings.
`InScope`	An opportunity to place extra items in scope during parsing, or to change the scope altogether. If `et_flag` is 1 when this is called, the scope is being worked out for `each_turn` reasons; otherwise for everyday parsing.
`LookRoutine`	Called at the end of every `Look` description.
`NewRoom`	Called when the room changes, before any description of it is printed. This happens in the course of ordinary movements or use of `PlayerTo`, but may not happen if the game uses `move` to shift the player object directly.
`ParseNoun(obj)`	To do the job of parsing the `name` property (if `parse_name` hasn't done it already). This takes one argument, the object in question, and returns a value as if it were a `parse_name` routine.
`ParseNumber(text,n`	An opportunity to parse numbers in a different (or additional) way. The text to be parsed is a byte array of length `n` starting at `text`.
`ParserError(pe)`	The chance to print different parser error messages (like "I don't understand that sentence"). `pe` is the parser error number (see §25).
`PrintRank`	Completes the printing of the score. You might want to change this, so as to make the ranks something like "junior astronaut" or "master catburglar" or whatever suits your game.
`PrintVerb(v)`	A chance to change the verb printed out in a parser question (like "What do you want to (whatever)?") in case an unusual verb via `UnknownVerb` has been constructed. `v` is the dictionary address of the verb. Returns true (or 1) if it has printed something.
`PrintTaskName(n)`	Prints the name of task `n` (such as "driving the car").
`TimePasses`	Called after every turn (but not, for instance, after a command like "score" or "save"). It's much more elegant to use timers and daemons, or `each_turn` routines for individual rooms – using this is a last resort.
`UnknownVerb`	Called by the parser when it hits an unknown verb, so that you can transform it into a known one.

The following constants, if defined in a game, change settings made by the library. Those described as "To indicate that..." have no meaningful value; one simply defines them by, e.g., the directive `Constant DEBUG;`.

`AMUSING_PROVIDED`	To indicate that an `Amusing` routine is provided.
`DEBUG`	To indicate that the special "debugging" verbs are to be included.
`Headline`	Style of game and copyright message.
`MAX_CARRIED`	Maximum number of (direct) possessions the player can carry.

MAX_SCORE	Maximum game score.
MAX_TIMERS	Maximum number of timers or daemons active at any one time (defaults to 32).
NO_PLACES	To indicate that the "places" and "objects" verbs should not be allowed.
NUMBER_TASKS	Number of 'tasks' to perform.
OBJECT_SCORE	Score for picking up a `scored` object for the first time.
ROOM_SCORE	Score for visiting up a `scored` room for the first time.
SACK_OBJECT	Object which acts as a 'rucksack', into which the game automatically tidies away things for the player.
Story	Story name, conventionally in CAPITAL LETTERS.
TASKS_PROVIDED	To indicate that "tasks" are provided.
WITHOUT_DIRECTIONS	To indicate that the standard compass directions are to be omitted.

38 The actions and fakes

The actions implemented by the library are in three groups. Group 1 consists of actions associated with 'meta'-verbs, which are not subject to game rules. (If you want a room where the game can't be saved, as for instance 'Spellbreaker' cunningly does, you'll have to tamper with `SaveSub` directly, using a `Replaced` routine.)

```
1a. Quit, Restart, Restore, Verify, Save, ScriptOn, ScriptOff, Pronouns,
    Score, Fullscore, LMode1, LMode2, LMode3, NotifyOn, NotifyOff,
    Version, Places, Objects.
```

(`Lmode1`, `Lmode2` and `Lmode3` switch between "brief", "verbose" and "superbrief" room description styles.) In addition, but only if `DEBUG` is defined, so that the debugging suite is present, group 1 contains

```
1b. TraceOn, TraceOff, TraceLevel, ActionsOn, ActionsOff, RoutinesOn,
    RoutinesOff, TimersOn, TimersOff, CommandsOn, CommandsOff, CommandsRead,
    Predictable, XPurloin, XAbstract, XTree, Scope, Goto, Gonear.
```

Group 2 contains actions which sometimes get as far as the 'after' stage, because the library sometimes does something when processing them.

```
2. Inv, InvTall, InvWide, Take, Drop, Remove, PutOn, Insert, Transfer,
   Empty, Enter, Exit, GetOff, Go, GoIn, Look, Examine, Search, Give, Show,
   Unlock, Lock, SwitchOn, SwitchOff, Open, Close, Disrobe, Wear, Eat.
```

Group 3 contains the remaining actions, which never reach 'after' because the library simply prints a message and stops at the 'during' stage.

```
3. Yes, No, Burn, Pray, Wake, WakeOther [person], Consult,
   Kiss, Think, Smell, Listen, Taste, Touch, Dig,
   Cut, Jump [jump on the spot], JumpOver, Tie, Drink,
   Fill, Sorry, Strong [swear word], Mild [swear word], Attack, Swim,
   Swing [something], Blow, Rub, Set, SetTo, WaveHands [ie, just "wave"],
```

```
Wave [something], Pull, Push, PushDir [push something in a direction],
Turn, Squeeze, LookUnder [look underneath something],
ThrowAt, Answer, Buy, Ask, AskFor, Sing, Climb, Wait, Sleep.
```

△ The actions `PushDir` and `Go` (while the player is inside an `enterable` object) have special rules: see §10.

The library also defines 8 fake actions:

`LetGo, Receive, ThrownAt, Order, TheSame, PluralFound, Miscellany, Prompt`

`LetGo`, `Receive` and `ThrownAt` are used to allow the `second` noun of `Insert`, `PutOn`, `ThrowAt`, `Remove` actions to intervene; `Order` is used to process actions through somebody's `life` routine; `TheSame` and `PluralFound` are defined by the parser as ways for the program to communicate with it; `Miscellany` and `Prompt` are defined as slots for `LibraryMessages`.

39 Library message numbers

Answer: "There is no reply."
Ask: "There is no reply."
Attack: "Violence isn't the answer to this one."
Blow: "You can't usefully blow that."
Burn: "This dangerous act would achieve little."
Buy: "Nothing is on sale."
Climb: "I don't think much is to be achieved by that."
Close: 1. "That's not something you can close." 2. "It's already closed." 3. "You close ⟨x1⟩."
Consult: "You discover nothing of interest in ⟨x1⟩."
Cut: "Cutting that up would achieve little."
Dig: "Digging would achieve nothing here."
Disrobe: 1. "You're not wearing that." 2. "You take off ⟨x1⟩."
Drink: "There's nothing suitable to drink here."
Drop: 1. "Already on the floor." 2. "You haven't got that." 3. "(first taking ⟨x1⟩ off)" 4. "Dropped."
Eat: 1. "That's plainly inedible." 2. "You eat ⟨x1⟩. Not bad."
EmptyT: 1. ⟨x1⟩ " can't contain things." 2. ⟨x1⟩ " is closed." ⟨x1⟩ " is empty already."
Enter: 1. "But you're already on/in ⟨x1⟩." 2. "That's not something you can enter." 3. "You can't get into the closed ⟨x1⟩." 4. "You can only get into something on the floor." 5. "You get onto/into ⟨x1⟩."
Examine: 1. "Darkness, noun. An absence of light to see by." 2. "You see nothing special about ⟨x1⟩." 3. "⟨x1⟩ is currently switched on/off."
Exit: 1. "But you aren't in anything at the moment." 2. "You can't get out of the closed ⟨x1⟩." 3. "You get off/out of ⟨x1⟩."

Fill: "But there's no water here to carry."

FullScore: 1. "The score is/was made up as follows:^" 2. "finding sundry items" 3. "visiting various places" 4. "total (out of `MAX_SCORE`)"

GetOff: "But you aren't on ⟨x1⟩ at the moment."

Give: 1. "You aren't holding ⟨x1⟩." 2. "You juggle ⟨x1⟩ for a while, but don't achieve much." 3. "⟨x1⟩ doesn't seem interested."

Go: 1. "You'll have to get off/out of ⟨x1⟩ first." 2. "You can't go that way." 3. "You are unable to climb ⟨x1⟩." 4. "You are unable to descend ⟨x1⟩." 5. "You can't, since ⟨x1⟩ is in the way." 6. "You can't, since ⟨x1⟩ leads nowhere."

Insert: 1. "You need to be holding it before you can put it into something else." 2. "That can't contain things." 3. "Alas, it is closed." 4. "You'll need to take it off first." 5. "You can't put something inside itself." 6. "(first taking it off)^" 7. "There is no more room in ⟨x1⟩." 8. "Done." 9. "You put ⟨x1⟩ into ⟨second⟩."

Inv: 1. "You are carrying nothing." 2. "You are carrying"

Jump: "You jump on the spot, fruitlessly."

JumpOver: "You would achieve nothing by this."

Kiss: "Keep your mind on the game."

Listen: "You hear nothing unexpected."

LMode1: " is now in its normal brief printing mode, which gives long descriptions of places never before visited and short descriptions otherwise."

LMode2: " is now in its verbose mode, which always gives long descriptions of locations (even if you've been there before)."

LMode3: " is now in its superbrief mode, which always gives short descriptions of locations (even if you haven't been there before)."

Lock: 1. "That doesn't seem to be something you can lock." 2. "It's locked at the moment." 3. "First you'll have to close it." 4. "That doesn't seem to fit the lock." 5. "You lock ⟨x1⟩."

Look: 1. "on" 2. "in" 3. "as" 4. "^On ⟨x1⟩ is/are ⟨list of children⟩" 5. "[On/In ⟨x1⟩] you/You can also see ⟨list of children⟩ [here]." 6. "[On/In ⟨x1⟩] you/You can see ⟨list of children⟩ [here]."

LookUnder: 1. "But it's dark." "You find nothing of interest."

Mild: "Quite."

Miscellany: 1. "(considering the first sixteen objects only)^" 2. "Nothing to do!" 3. " You have died " 4. " You have won " 5. (The RESTART/RESTORE/QUIT and possibly FULL and AMUSING query, printed after the game is over.) 6. "[Your interpreter does not provide undo. Sorry!]" 7. "Undo failed. [Not all interpreters provide it.]" 8. "Please give one of the answers above." 9. "^It is now pitch dark in here!" 10. "I beg your pardon?" 11. "[You can't "undo" what hasn't been done!]" 12. "[Can't "undo" twice in succession. Sorry!]" 13. "[Previous turn undone.]" 14. "Sorry, that can't be corrected." 15. "Think nothing of it." 16. ""Oops" can only correct a single word."

No: see **Yes**

NotifyOff: "Score notification off."

NotifyOn: "Score notification on."

Objects: 1. "Objects you have handled:^" 2. "None."

Open: 1. "That's not something you can open." 2. "It seems to be locked." 3. "It's already open." 4. "You open ⟨x1⟩, revealing ⟨list of children of x1⟩" 5. "You open ⟨x1⟩."

Order: "⟨x1⟩ has better things to do."

Places: "You have visited: "

Pray: "Nothing practical results from your prayer."
Prompt: 1. "^>"
Pull: 1. "It is fixed in place." 2. "You are unable to." 3. "Nothing obvious happens." 4. "That would be less than courteous."
Push: see **Pull**
PushDir: 1. "Is that the best you can think of?" 2. "That's not a direction." 3. "Not that way you can't."
PutOn: 1. "You need to be holding ⟨x1⟩ before you can put it on top of something else." 2. "You can't put something on top of itself." 3. "Putting things on ⟨x1⟩ would achieve nothing." 4. "You lack the dexterity." 5. "(first taking it off)^" 6. "There is no more room on ⟨x1⟩." 7. "Done." 8. "You put ⟨x1⟩ on ¡second¿."
Quit: 1. "Please answer yes or no." 2. "Are you sure you want to quit? "
Remove: 1. "It is unfortunately closed." 2. "But it isn't there now." 3. "Removed."
Restart: 1. "Are you sure you want to restart? " 2. "Failed."
Restore: 1. "Restore failed." 2. "Ok."
Rub: "You achieve nothing by this."
Save: 1. "Save failed." 2. "Ok."
Score: "You have so far/In that game you scored ⟨score⟩ out of a possible `MAX_SCORE`, in ⟨turns⟩ turn/turns"
ScriptOn: 1. "Transcripting is already on." 2. "Start of a transcript of"
ScriptOff: 1. "Transcripting is already off." 2. "^End of transcript."
Search: 1. "But it's dark." 2. "There is nothing on ⟨x1⟩." 3. "On ⟨x1⟩ is/are ⟨list of children⟩." 4. "You find nothing of interest." 5. "You can't see inside, since it is closed." 6. "⟨x1⟩ is empty." 7. "In ⟨x1⟩ is/are ⟨list of children⟩."
Set: "No, you can't set that."
SetTo: "No, you can't set that to anything."
Show: 1. "You aren't holding ⟨x1⟩." 2. "⟨x1⟩ is unimpressed."
Sing: "Your singing is abominable."
Sleep: "You aren't feeling especially drowsy."
Smell: "You smell nothing unexpected."
Sorry: "Oh, don't apologise."
Squeeze: 1. "Keep your hands to yourself." 2. "You achieve nothing by this."
Strong: "Real adventurers do not use such language."
Swim: "There's not enough water to swim in."
Swing: "There's nothing sensible to swing here."
SwitchOff: 1. "That's not something you can switch." 2. "That's already off." 3. "You switch ⟨x1⟩ off."
SwitchOn: 1. "That's not something you can switch." 2. "That's already on." 3. "You switch ⟨x1⟩ on."
Take: 1. "Taken." 2. "You are always self-possessed." 3. "I don't suppose ⟨x1⟩ would care for that." 4. "You'd have to get off/out of ⟨x1⟩ first." 5. "You already have that." 6. "That seems to belong to ⟨x1⟩." 7. "That seems to be a part of ⟨x1⟩." 8. "That isn't available." 9. "⟨x1⟩ is not open." 10. "That's hardly portable." 11. "That's fixed in place." 12. "You're carrying too many things already." 13. "(putting ⟨x1⟩ into `SACK_OBJECT` to make room)"
Taste: "You taste nothing unexpected."
Tell: 1. "You talk to yourself a while." 2. "This provokes no reaction."

Touch: 1. "Keep your hands to yourself!" 2. "You feel nothing unexpected." 3. "If you think that'll help."
Think: "What a good idea."
Tie: "You would achieve nothing by this."
ThrowAt: 1. "Futile." 2. "You lack the nerve when it comes to the crucial moment."
Transfer: 1. "That isn't in your possession." "First pick that up."
Turn: see **Pull**
Unlock: 1. "That doesn't seem to be something you can unlock." 2. "It's unlocked at the moment." 3. "That doesn't seem to fit the lock." 4. "You unlock ⟨x1⟩."
VagueGo: "You'll have to say which compass direction to go in."
Verify: 1. "The game file has verified as intact." 2. "The game file did not verify properly, and may be corrupted (or you may be running it on a very primitive interpreter which is unable properly to perform the test)."
Wait: "Time passes."
Wake: "The dreadful truth is, this is not a dream."
WakeOther: "That seems unnecessary."
Wave: 1. "But you aren't holding that." 2. "You look ridiculous waving ⟨x1⟩."
WaveHands: "You wave, feeling foolish."
Wear: 1. "You can't wear that!" 2. "You're not holding that!" 3. "You're already wearing that!" 4. "You put on ⟨x1⟩."
Yes: "That was a rhetorical question."

What order the program should be in

This section summarises Inform's "this has to be defined before that can be" rules.

1. The three library files, `Parser`, `Verblib` and `Grammar` must be included in that order.
   (a) Before inclusion of `Parser`: you must define the constants `Story` and `Headline`; the constant `DEBUG` must be defined here, if anywhere; similarly for `Replace` directives; but you may not yet define global variables, objects or routines.
   (b) Between `Parser` and `Verblib`: if a 'sack object' is to be included, it should be defined here, and the constant `SACK_OBJECT` set to it; the `LibraryMessages` object should be defined here, if at all; likewise the `task_scores` array.
   (c) Before inclusion of `Verblib`: the constants

   `MAX_CARRIED`, `MAX_SCORE`, `NUMBER_TASKS`, `OBJECT_SCORE`,
   `ROOM_SCORE`, `AMUSING_PROVIDED` and `TASKS_PROVIDED`

   must be defined before this (if ever).
   (d) Before inclusion of `Grammar`: `Verb` and `Extend` directives cannot be used.
   (e) After inclusion of `Grammar`: It's too late to define any entry point routines.
2. Any `Switches` directive must come before the definition of any constants.
3. If an object begins inside another, it must be defined after its parent.
4. `Global` variables must be declared earlier in the program than the first reference to them.
5. Properties, attributes and classes must be declared earlier than their first usage in an object definition.
6. General parsing and scope routines must be defined before being quoted in grammar tokens.
7. Nothing can be defined after the `End` directive.

A short Inform lexicon

This appendix is a brief dictionary of Inform jargon: it doesn't contain definitions of anything set in `computer` type (except to clarify ambiguities), for which see Chapters V and VI. Cross-references are italicised. All the information here is more fully explained in the body of the text, and can be found via the index.

action A single attempted action by the *player*, such as taking a lamp, generated either by the *parser* or in code. It is stored as three numbers, the first being the *action number*, the others being the *noun* and *second noun* (if any: otherwise 0).

action number A number identifying which kind of action is under way, e.g., `Take`, which can be written as a *constant* by prefacing its name with `##`.

action routine The *routine* of code executed when an *action* has been allowed to take place. What marks it out as the routine in question is that its name is the name of the *action* with `Sub` appended, as for instance `TakeSub`.

actor The *parser* can interpret what the *player* types as either a request for the player's own character to do something, in which case the actor is the player's object, or to request somebody else to do something, in which case the actor is the person being spoken to. This affects the parser significantly because the person speaking and the person addressed may be able to see different things.

additive An additive *property* is one whose value accumulates into a list held in a *word array*, rather than being over-written as a single value, during *inheritance* from *classes*.

Advanced game The default Inform *format* of *story file*, also known as Version 5. It can be extended (see *version*) if needed. *Standard games* should no longer be used unless necessary.

alias A single *attribute* or *property* may be used for two different purposes, with different names, provided care is exercised to avoid clashes: the two names are called aliases. The *library* does this: for instance, `time_out` is an alias for `daemon`.

ambiguity Arises when the *player* has typed something vague like "fish" in circumstances when many nearby objects might be called that. The *parser* then resolves this, possibly in conjunction with the program.

argument A parameter specified in a *routine* call, such as `7` in the call `AwardPoints(7)`.

array An indexed collection of global variables. There are four kinds, *byte arrays* `->`, *word arrays* `-->`, *strings* and *tables*.

assembly language The *Z-machine* runs a sequence of low-level instructions, or assembly lines (also called opcodes). These can be programmed directly as Inform *statements* by prefixing them with `@`, but only a few are documented in this manual, in §27, the rest being in the 'Specification of the Z-machine'.

assembler error A very low-level *error* caused by a malformed line of *assembly language*. These occasionally occur as a knock-on effect from Inform failing to recover from higher-level errors.

assignment A *statement* which sets the value of a *global* or *local variable*, or *array* entry.

attribute An *object* can be created as having certain attributes, which are simple off-or-on states (or flags), which can then be given, tested for or taken away by the program. For example, `light` represents the state "is giving off light".

block of code See *code block*.

box A rectangle of text, usually displayed in reverse video onto the screen and with text such as a quotation inside (see §26).

byte An 8-bit cell of memory, capable of holding numbers between 0 and 255.

byte address The whole lower part of the *memory map* of the *Z-machine* can be regarded as a *byte array*, and a byte address is an index into this. E.g., byte address 0 refers to the lowest byte in the machine (which always holds the *version* number). *Dictionary* words are internally stored as byte addresses.

byte array An *array* indexed with the `->` *operator* whose entries are only 1 byte each: they can therefore hold numbers between 0 and 255, or ASCII characters, but not strings or object numbers.

character A single letter 'A' or symbol '*', written as a *constant* using the notation `'A'`, and internally stored as its ASCII code. Can be printed using `print (char)`.

child See *object tree*.

class A template for an *object* definition, giving certain properties and attributes which are *inherited* by any objects defined as being of this class.

code block A collection of *statements* can be grouped together into a block using braces `{` and `}` so that they count as a single unit for `if` statements, what is to be done inside a `for` loop, etc.

compass The `compass` *object*, created by the *library* but never tangible to the *player* during the game, is used to hold the currently valid *direction objects*.

compiler The Inform program itself, which transmutes Inform programs (or source code) into the *story file* which is played with the use of an *interpreter* at *run-time*.

condition A state of affairs which either is, or isn't, true at any given moment, such as `x == y`, often written in round brackets `(` and `)`. The central operator `==` is also called the condition.

constant An explicitly written-out number, such as `34` or `$$10110111`; or the internal name of an object, such as `brass_lamp`, whose value is its *object number*; or the internal name of an array, whose value is its *byte address*; or a word defined by either the *library* or Inform code as meaning a particular value; or a *character*, written `'X'` and whose value is its ASCII code; or a *dictionary word*, written `'word'` and whose value is its *byte address*; or an *action*, written `##Action` and whose value is its *action number*; or a *routine* written `#r$Routine` whose value is its *packed address*; or the name of a *property* or *attribute* or *class*.

containment See *object tree*.

cursor An invisible notional position at which text is being printed in the upper *window*, when the windows are split; the origin is $(1,1)$ in the top left.

daemon A *routine* attached to an *object* which, once started, is run once during the end sequence of every *turn* until explicitly stopped. Used to manage events happening as time passes by, or to notice changes in the state of the game which require some activity.

default value See *property*.

description The usually quite long piece of text attached to an *object*; if it's a *room*, then this is the long description printed out when the room is first visited; otherwise it will usually be printed when the object is examined by the *player*.

dictionary A list kept inside the *Z-machine* of all the words ordinarily understood by the game, such as "throw" and "mauve", usually between about 300 and 2000 in length. Inform automatically puts this list together from all the `name` values of objects and all usages of constants like `'word'`. Dictionary words are stored to a resolution of 9 characters (6 for *Standard games*), written `'thus'` (provided they have more than one letter; otherwise `#n$x` for the word "x"; except as values of the special `name` property) and are internally referred to by numbers which are their *byte addresses* inside the list.

direct containment See *object tree*.

direction object An object representing both the abstract idea of a direction and the wall which is in that direction: for instance, `n_obj` represents "northness" and the north wall of the current *room*. Typing "go north" causes the *parser* to generate the *action* `Go n_obj`. The current direction objects are exactly those currently inside the `compass` object and they can be dynamically changed. The `door_dir` property of a direction object holds its corresponding direction property.

direction property The *library* creates 12 direction properties: `n_to`, `s_to`, etc., `u_to`, `d_to`, `in_to` and `out_to`. These are used to give *map* connections from *rooms* and indicate directions which doors and *direction objects* correspond to.

directive A line of Inform code which instructs the *compiler* to do something, such as to define a new *constant*; it takes immediate effect and does not correspond to anything happening at *run-time*. These are not normally written inside *routines* but can be if prefaced by a `#` character.

eldest child See *object tree*.

embedded routine A *routine* defined as the *property* value of an *object*, which is defined without a name of its own, and which by default returns 'false' rather than 'true'.

entry point A *routine* in an Inform program which is directly called by the *library* to intervene in the normal operation of the game (if the routine so wishes). Provision of entry points is optional, except for `Initialise`, which must always occur in every game.

error When the *compiler* finds something in the program which it can't make sense of, it produces an error (which will eventually prevent it from generating a *story file*, so that it cannot generate an illegal *story file* which would fail at *run-time*). If the error is *fatal* the compiler stops at once.

examine message See *description*.

expression A general piece of Inform code which determines a numerical value. It may be anything from a single *constant* to a bracketed calculation of *variable*, *property* or *array* values, such as `3+(day_list-->(calendar.number))`.

fake action A form of *action* which has no corresponding *action routine* and will have no effect after the `before`-processing stage of considering an action is over. A fake action is never generated by the *parser*, only by the program, which can use it to pass a message to an *object*.

fake fake action A form of *action* which does have an *action routine* and is processed exactly as ordinary actions are, but which is never generated by the *parser*, only by the program, which can use it to pass a message to an *object*.

fatal error An *error* found by the *compiler* which causes it to give up immediately; for instance, a disc being full or memory running out are fatal.

format See *version*.

function See *routine*.

fuse See *timer*.

global variable A variable which can be used by every routine in the program.

grammar A list of *lines* which is attached to a particular *verb*. The *parser* decodes what the *player* has typed by trying to match it against each line in turn of the grammar attached to the *verb* which the first word of the player's input corresponds to.

hardware function A function which is used just like any other *routine* but which is not defined anywhere in the *library* or *program*: the *compiler* provides it automatically, usually converting the apparent call to a routine into a single line of *assembly language*.

indirect containment See *object tree*.

inheritance The process in which *property* values and *attribute* settings specified in a *class* definition are passed on to an *object* defined as having that class.

internal name See *name*.

interpreter A program for some particular model of computer, such as an IBM PC, which reads in the *story file* of a game and allows someone to play it. A different interpreter is needed for each model of computer (though generic source codes exist which make it relatively easy to produce these).

inventory 1. Verb, imperative: a demand for a list of the items one is holding; 2. noun: the list itself. (When Crowther and Woods were writing the original 'Advent', they were unable to think of a good imperative verb and fell back on the barely sensible "take inventory", which was soon corrupted into the not at all sensible "inventory", thence "inv" and finally "i".)

library The 'operating system' for the *Z-machine*: a large segment of Inform code, written out in three *library files*, which manages the model world and consults the game's program now and then to give it a chance to make interesting things happen.

library files The three files `parser`, `verblib` and `grammar` containing the source code of the *library*. These are normally `Included` in the code for every Inform game.

library routine A *routine* provided by the *library* which is 'open to the public' in that the designer's program is allowed to call and make use of it.

line One possible pattern which the *parser* might match against what the *player* has typed beyond the initial *verb* word. A *grammar* line consists of a sequence of *tokens*, each of which must be matched in sequence, plus an *action* which will be generated if the line successfully matches.

local variable A variable attached to a particular *routine* (or, more precisely, a particular call to a routine: if a routine calls itself, then the parent and child incarnation have independent copies of the local variables) whose value is inaccessible to the rest of the program. Also used to hold the *arguments* of the call.

long A *property* whose values must always be stored as *words*, or *word arrays*, rather than *bytes* or *byte arrays*. A safely ignorable concept since except for *Standard games* all properties are long.

logical machine See *Z-machine*.

low string A string which can be used as the value of a *variable string*, printed with the `@` escape character. Must be declared with `Lowstring`.

map The geographical design of the game, divided into areas called *rooms* with connections between them in different directions. The *story file* doesn't contain an explicit map table but stores the information implicitly in the definition of the room *objects*.

memory map Internally, the *Z-machine* contains a large *array* in whose values the entire story file and all its data structures are stored. Particular cells low down in this array are indexed by *byte addresses*, and *routines* and *strings* which are lodged higher up are referred to by *packed addresses*. The organisation of this array (which ranges of indices correspond to what) is called the memory map.

meta-verb A *verb* whose actions are always commands from the *player* to the game, rather than requests for something to happen in the model world: for instance, "quit" is meta but "take" is not.

multiple object The *parser* matches a *token* with a multiple object when the *player* has either explicitly referred to more than one *object* (e.g. "drop lamp and basket") or implicitly done so (e.g. "drop everything" when this amounts to more than 1 item); though the match is only made if the token will allow it.

names An *object* has three kinds of name: 1. its internal name, a word such as `brass_lamp`, which is a *constant* referring to it within the program; 2. its short name, such as "dusty old brass lamp" or "Twopit Room", which is printed in *inventories* or before a *room description*; 3. *dictionary words* which appear as values of its `name` property, such as `"dusty"`, `"brass"`, etc., which the *player* can type to refer to it.

noun The first parameter (usually an *object* but possibly a number) which the *parser* has matched in a *line* of *grammar* is the noun for the *action* which is generated. It is stored in the `noun` variable (not to be confused with the `noun` token).

object The physical substance of the game's world is divided up into indivisible objects, such as 'a brass lamp' or 'a meadow'. These contain each other in a hierarchy called the *object tree*. An object may be defined with an initial location (another object) and must have an *internal name* and a *short name*; attached to it throughout the game are variables called *attributes* and *properties* which reflect its current state. The definition of an object may make it *inherit* initial settings for this state from one or more *classes*.

object number *Objects* are automatically numbered from 1 upwards, in order of definition, and the *internal name* of an object is in fact a *constant* whose value is this number.

object tree The hierarchy of containment between *objects*. Each has a 'parent', though this may be 'nothing' (to indicate that it is uncontained, as for instance *rooms* are) and possibly some 'children' (the objects directly contained within it). The 'child' of an object is the 'eldest' of these children, the one most recently moved within it or, if none have been moved into it since the start of play, the first one defined as within it. The 'sibling' of this child is then the next eldest, or may be 'nothing' if there is no next eldest. Note that if A is inside B which is itself inside C, then C 'directly contains' B but only 'indirectly contains' A: and we do not call A one of the children of C.

obsolete usage A point in the program using Inform syntax which was correct under some previous version of the *compiler* but is no longer correct (usually because there is a neater way to express the same idea). Inform often allows these but, if so, issues *warnings*.

opcodes See *assembly language*.

operator A symbol in an *expression* which acts on one or more sub-expressions, combining their values to produce a result. This may be arithmetic, as in + or /, or to do with array or property value indexing, as in `->` or `.&`. Note that *condition* operators such as `==` are not formally expression operators.

order An instruction by the *player* for somebody else to do something. For instance, "policeman, give me your hat" is an order. The order is parsed as if an *action* but is then processed in the other person's *object* definition.

packed address A number encoding the location of a *routine* or *string* within the *memory map* of the *Z-machine*.

parent See *object tree*.

parser That part of the *library* which, once per turn, issues the *prompt*; asks the *player* to type something; looks at the initial *verb* word; tries to match the remaining words against one of the *lines* of *grammar* for this verb and, if successful, generates the resulting *action*.

player 1. the person sitting at the keyboard at *run-time*, who is playing the game; 2. his character inside the model world of the game. (There is an important difference - one has access to the "undo" verb. The other actually dies.)

prompt The text printed to invite the *player* to type: usually just `>`.

property 1. The value of a variable attached to a particular *object*, accessible throughout the program, which can be a single *word*, an *embedded routine* or an *array* of values; 2. a named class of such variables, such as `description`, one of which may be held by any object created, and which has a default value for any object not explicitly created with one.

resolution See *dictionary*.

return value See *routine*.

room The geography of a game is subdivided into parcels of area called rooms, within which it is (usually) assumed that the *player* has no particular location but can reach all corners of easily and without giving explicit instruction to do so. For instance, "the summit of Scafell

Pike" might be such an area, while "the summit of Ben Nevis" (being a large L-shaped ridge) would probably be divided into three or four. These rooms fit together into the *map* and each is implemented as an *object*.

room description See *description*.

routine In an Inform program (and indeed in the *Z-machine*) code is always executed in routines, each of which is called (possibly with *arguments*) and must return a particular *word* value, though this is sometimes disguised from the programmer because (for example) the *statement* `return;` actually returns true (1) and the statement `ExplodeBomb();` makes the call to the routine but throws away the return value subsequently. Routines are permitted to call themselves (if the programmer wants to risk it) and have their own *local variables*.

rule *Embedded routines* given as values of a *property* like `before` or `after` are sometimes loosely called rules, because they encode exceptional rules of the game such as "the 10-ton weight cannot be picked up". However, there is no formal concept of 'rule'.

run-time The time when an *interpreter* is running the *story file*, i.e., when someone is actually playing the game, as distinct from 'compile-time' (when the *compiler* is at work making the story file). Some errors (such as an attempt to divide a number by zero) can only be detected at run-time.

scope To say that an *object* is in scope to a particular *actor* is roughly to say that it is visible, and can sensibly be referred to.

second noun The second parameter (usually an *object* but possibly a number) which the *parser* has matched in a *line* of *grammar* is the second noun for the *action* generated. It is stored in the `second` variable.

see-through An *object* is called this if it has `transparent`, or is an open `container`, or is a `supporter`. Roughly this means 'if the object is visible, then its *children* are visible'. (This criterion is often applied in the *scope* (and 'light') rules inside the *library*.)

short name See *name*.

sibling See *object tree*.

statement A single instruction for the game to carry out at *run-time*; a *routine* is a collection of statements. These include *assignments* and *assembly language* but not *directives*.

status line The region at the top of the screen which, in play, usually shows the current score and location, and which is usually printed in reversed colours for contrast.

story file The output of the *compiler* is a single file containing everything about the game produced, in a *format* which is standard. To be played, the file must be run with an *interpreter*. Thus only one file is needed for every Inform game created, and only one auxiliary program must be written for every model of computer which is to run such games. In this way story files are absolutely portable across different computers.

Standard game An old *format* (version 3) of *story file* which should no longer be used unless absolutely necessary (to run on very small computers) since it imposes tiresome restrictions.

string 1. a literal piece of text such as `"Mary had a fox"` (which is a *constant* internally represented by a number, its *packed address*, and may be created as a *low string*); 2. a form of *byte array* in which the 0th entry holds the number of entries (so called because such an array is usually used as a list of *characters*, i.e. a string variable); 3. see *variable string*.

switch 1. certain *objects* are 'switchable', meaning they can be turned off or on by the *player*; 2. options set by the programmer when the *compiler* starts are called switches; 3. a `switch` *statement* is one which switches execution, like a railway turntable, between different lines according to the current value of an *expression*.

synonym Two or more words which refer to the same *verb* are called synonyms (for example, "wear" and "don").

table A form of *word array* in which the 0th entry holds the number of entries.

timer A *routine* attached to a particular *object* which, once set, will be run after a certain number of *turns* have passed by. (Sometimes called a 'fuse'.)

token A particle in a *line* of *grammar*, which the *parser* tries to match with one or more words from what the *player* has typed. For instance, the token `held` can only be matched by an *object* the *actor* is holding.

tree See *object tree*.

turn The period in play between one typed command and another.

type A *word* value may represent a literal number, a *packed address*, an *action number*, etc., and there is only a limited way to tell what the 'type' of it is: see §17.

untypeable word A *dictionary word* which contains at least one space, full stop or comma and therefore can never be recognised by the *parser* as one of the words typed by the *player*.

variable A named value which can be set or compared so that it varies at *run-time*. It must be declared before use (the *library* declares many such). Variables are either *local* or *global*; entries in *arrays* (or the *memory map*) and *properties* of *objects* behave like global variables.

variable string (Not the same as a *string* (3) variable.) There are 32 of these, which can only be set (to a *string* (1) which must have been defined as a *low string*) or printed out (using the `@` escape character).

vehicle An *object* which the *player* character can travel around in.

verb 1. a collection of *synonymous* one-word English verbs for which the *parser* has a *grammar* of possible *lines* which a command starting with one of these verbs might take; 2. one of the one-word English verbs.

version The *compiler* can produce 6 different formats of *story file*, from Version 3 (or *Standard*) to Version 8. By default it produces Version 5 (or *Advanced*) which is the most portable.

warning When the *compiler* finds something in the program which it disapproves of (for example, an *obsolete usage*) or thinks might be a mistake, it issues a warning message. This resembles an *error* but does not prevent successful compilation; a working *story file* can still be produced.

window (Except in *Standard games*) the screen is divided into two windows, an upper, fixed window usually containing the *status line* and the lower, scrolling window usually holding the text of the game. One can divert printing to the upper window and move a *cursor* about in it.

word 1. an English word in the game's *dictionary*; 2. almost all numbers are stored in 16-bit words of memory which unlike *bytes* can hold any *constant* value, though they take twice as much storage space up.

word array An *array* indexed with the `-->` *operator* whose entries are *words*: they can therefore hold any *constant* values.

youngest child See *object tree*.

Z-machine The imaginary computer which the *story file* is a program for. One romantically pretends that this is built from circuitboards and microchips (using terms like 'hardware') though in fact it is merely simulated at *run-time* by an *interpreter* running on some (much more sophisticated) computer. Z is for 'Zork'.

Answers to all the exercises

> World is crazier and more of it than we think,
> Incorrigibly plural. I peel and portion
> A tangerine and spit the pips and feel
> The drunkenness of things being various.
>
> – Louis MacNeice (1907–1963), *Snow*

• **1** Change the mushroom's `after` rule to:

```
after
[;  Take: if (self hasnt general)
          {   give self general;
             "You pick the mushroom, neatly cleaving its thin stalk.";
          }
          "You pick up the slowly-disintegrating mushroom.";
    Drop: "The mushroom drops to the ground, battered slightly.";
],
```

Note that `general` is a general-purpose attribute, always free for the designer to use. The 'neatly cleaving' message can only happen once, because after that the mushroom object (which calls itself `self`) must have `general`.

• **2** `if (obj.&door_to == 0) { ... }`

• **3** Put any validation rules desired into the `GamePreRoutine`. For example, the following will filter out any stray `Drop` actions for unheld objects:

```
[ GamePreRoutine;
  if (action==Drop && noun notin player)
     print_ret "You aren't holding ", (the) noun, ".";
  rfalse;
];
```

• **4** Declare a fake action called, say, `OpenUp`. Then:

```
Object medicine "guaranteed child-proof medicine bottle" cupboard
  with name "medicine" "bottle",
       description "~Antidote only: no preventative effect.~",
       before
       [;  Open, Unlock: "It's adult-proof too.";
           OpenUp: give self open ~locked; "The bottle cracks open!";
       ],
  has  container openable locked;
```

Any other code in the game can execute `<OpenUp medicine>` to crack open the bottle. For brevity, this solution assumes that the bottle is always visible to the player when it is opened.

• **5** Briefly: provide a `GamePreRoutine` which tests to see if `second` is a non-zero object. If it is, then call a `second_before` property for the object (having already created this new property), and so on.

• **6**

```
Object orange_cloud "orange cloud"
  with name "orange" "cloud",
       react_before
       [;  Look: "You can't see for the orange cloud surrounding you.";
           Go, Exit: "You wander round in circles, choking.";
           Smell: if (noun==0) "Cinnamon?  No, nutmeg.";
       ],
  has  scenery;
```

Directions (such as "north") are objects called `n_obj`, `s_obj` and so on: in this case, `in_obj`. (They are not to be confused with the property names `n_to` and so on.) Moreover, you can change these directions: as far as Inform is concerned, a direction is any object in the special object `compass`.

• **7** Define four objects along the lines of:

```
Object white_obj "white wall" compass
  with name "white" "sac" "wall", article "the", door_dir n_to
  has  scenery;
```

and add the following line to `Initialise`:

```
remove n_obj; remove e_obj; remove w_obj; remove s_obj;
```

(We could even `alias` a new property `white_to` to be `n_to`, and then enter map directions in the source code using Mayan property names.) As a fine point of style, turquoise (*yax*) is the world colour for 'here', so add a grammar line to make this cause a "look":

```
Verb "turquoise" "yax" * -> Look;
```

• **8**

```
[ SwapDirs o1 o2 x;
  x=o1.door_dir; o1.door_dir=o2.door_dir; o2.door_dir=x; ];
[ ReflectWorld;
  SwapDirs(e_obj,w_obj); SwapDirs(ne_obj,nw_obj); SwapDirs(se_obj,sw_obj);
];
```

• **9** This is a prime candidate for using variable strings `@nn`. Briefly: at the head of the source, define

```
Lowstring east_str "east"; Lowstring west_str "west";
```

and then add two more routines to the game,

```
[ NormalWorld; String 0 east_str; String 1 west_str; ];
[ ReversedWorld; String 0 west_str; String 1 east_str; ];
```

where `NormalWorld` is called in `Initialise` or to go back to normal, and `ReversedWorld` when the reflection happens. Write `@00` in place of `east` in any double-quoted printable string, and similarly `@01` for `west`. It will be printed as whichever is currently set. (Inform provides up to 32 such variable strings.)

•10

```
Nearby bag "toothed bag"
  with name "toothed" "bag",
       description "A capacious bag with a toothed mouth.",
       before
       [; LetGo: "The bag defiantly bites itself \
                  shut on your hand until you desist.";
          Close: "The bag resists all attempts to close it.";
       ],
       after
       [; Receive:
                 print_ret "The bag wriggles hideously as it swallows ",
                           (the) noun, ".";
       ],
  has  container open;
```

•11

```
Object television "portable television set" lounge
  with name "tv" "television" "set" "portable",
       before
       [;  SwitchOn: <<SwitchOn power_button>>;
           SwitchOff: <<SwitchOff power_button>>;
           Examine: <<Examine screen>>;
       ],
  has  transparent;
Nearby power_button "power button"
  with name "power" "button" "switch",
       after
       [;  SwitchOn, SwitchOff: <<Examine screen>>;
       ],
  has  switchable;
Nearby screen "television screen"
  with name "screen",
       before
       [;  Examine: if (power_button hasnt on) "The screen is black.";
                "The screen writhes with a strange Japanese cartoon.";
       ];
```

• 12

```
Nearby glass_box "glass box with a lid"
  with name "glass" "box" "with" "lid"
  has  container transparent openable open;
Nearby steel_box "steel box with a lid"
  with name "steel" "box" "with" "lid"
  has  container openable open;
```

• 13 (The description part of this answer uses a routine from §19, but is only decoration.) Note the careful use of `inp1` and `inp2` rather than `noun` or `second`: see the note at the end of §4.

```
Nearby macrame_bag "macrame bag"
  with name "macrame" "bag" "string" "net" "sack",
       react_before
       [;  Examine, Search, Listen, Smell: ;
           default:
               if (inp1>1 && inp1 in self)
                   print_ret (The) inp1, " is tucked away in the bag.";
               if (inp2>1 && inp2 in self)
                   print_ret (The) inp2, " is tucked away in the bag.";
       ],
       describe
       [;  print "^A macrame bag hangs from the ceiling, shut tight";
           if (child(self)==0) ".";
           print ".  Inside you can make out ";
           WriteListFrom(child(self), ENGLISH_BIT); ".";
       ],
  has  container transparent;
Object watch "gold watch" macrame_bag
  with name "gold" "watch",
       description "The watch has no hands, oddly.",
       react_before
       [;  Listen: if (noun==0 or self) "The watch ticks loudly."; ];
```

• 14 The "plank breaking" rule is implemented here in its `door_to` routine. Note that this returns 'true' after killing the player.

```
Nearby PlankBridge "plank bridge"
  with description "Extremely fragile and precarious.",
       name "precarious" "fragile" "wooden" "plank" "bridge",
       when_open
           "A precarious plank bridge spans the chasm.",
       door_to
       [;  if (children(player)~=0)
           {   deadflag=1;
              "You step gingerly across the plank, which bows under \
               your weight. But your meagre possessions are the straw \
```

```
                which breaks the camel's back! There is a horrid crack...";
            }
            print "You step gingerly across the plank, grateful that \
                   you're not burdened.^";
            if (location==NearSide) return FarSide; return NearSide;
        ],
        door_dir
        [;  if (location==NearSide) return s_to; return n_to;
        ],
        found_in NearSide FarSide,
  has   static door open;
```

There might be a problem with this solution if your game also contained a character who wandered about, and whose code was clever enough to run `door_to` routines for any `door`s it ran into. If so, `door_to` could perhaps be modified to check that the `actor` is the `player`.

• 15

```
Nearby cage "iron cage"
  with name "iron" "cage" "bars" "barred" "iron-barred",
       when_open
          "An iron-barred cage, large enough to stoop over inside, \
           looms ominously here.",
       when_closed "The iron cage is closed.",
  has  enterable container openable open static;
```

• 16 Change the car's `before` to

```
    before
    [; Go: if (noun==e_obj)
           {   print "The car will never fit through your front door.^";
               return 2;
           }
           if (car has on) "Brmm!  Brmm!";
           print "(The ignition is off at the moment.)^";
    ],
```

• 17 Insert these lines into the `before` rule for `PushDir`:

```
                if (second==u_obj) <<PushDir self n_obj>>;
                if (second==d_obj) <<PushDir self s_obj>>;
```

• 18

```
Nearby bible "black Tyndale Bible"
  with name "bible" "black" "book",
       initial "A black Bible rests on a spread-eagle lectern.",
       description "A splendid foot-high Bible, which must have survived \
```

```
            the burnings of 1520.",
        before
        [ w x; Consult:
                wn = consult_from; w = NextWord();
                switch(w)
                {   'matthew': x="Gospel of St Matthew";
                    'mark': x="Gospel of St Mark";
                    'luke': x="Gospel of St Luke";
                    'john': x="Gospel of St John";
                    default: "There are only the four Gospels.";
                }
                if (consult_words==1)
                    print_ret "You read the ", (string) x, " right through.";
                w = TryNumber(wn);
                if (w==-1000)
                    print_ret "I was expecting a chapter number in the ",
                              (string) x, ".";
                print_ret "Chapter ", (number) w, " of the ", (string) x,
                      " is too sacred for you to understand now.";
        ];
```

• **19** Note that whether reacting before or after, the psychiatrist does not cut any actions short, because `react_before` and `react_after` both return false.

```
Nearby psychiatrist "bearded psychiatrist"
  with name "bearded" "doctor" "psychiatrist" "psychologist" "shrink",
       initial "A bearded psychiatrist has you under observation.",
       life
       [;  "He is fascinated by your behaviour, but makes no attempt to \
             interfere with it.";
       ],
       react_after
       [;  Insert: print "~Subject puts ", (name) noun, " in ",
                         (name) second, ". Interesting.~^^";
           Look: print "~Pretend I'm not here,~ says the psychiatrist.^";
       ],
       react_before
       [;  Take, Remove: print "~Subject feels lack of ", (the) noun,
                 ". Suppressed Oedipal complex? Mmm.~^";
       ],
  has  animate;
```

• **20** Add the following lines, after the inclusion of `Grammar`:

```
[ SayInsteadSub; "[To talk to someone, please type ~someone, something~ \
or else ~ask someone about something~.]"; ];
Extend "answer" replace * ConTopic -> SayInstead;
Extend "tell"   replace * ConTopic -> SayInstead;
```

A slight snag is that this will throw out "nigel, tell me about the grunfeld defence" (which the library will normally convert to an `Ask` action, but can't if the grammar for "tell" is missing); to avoid this, you could instead `Replace` the `TellSub` routine (see §17) by the `SayInsteadSub` one.

• **21** There are several ways to do this. The easiest is to add more grammar to the parser and let it do the hard work:

```
Nearby computer "computer"
  with name "computer",
       orders
       [;  Theta: print_ret "^Theta now set to ", noun, ".^";
           default: print_ret "^Please rephrase.^";
       ],
  has  talkable;
...
[ ThetaSub; "You must tell your computer so."; ];
Verb "theta" * "is" number -> Theta;
```

• **22** Obviously, a slightly wider repertoire of actions might be a good idea, but:

```
Nearby Charlotte "Charlotte"
  with name "charlotte" "charlie" "chas",
       grammar
       [;  give self ~general;
           wn=verb_wordnum;
           if (NextWord()=='simon' && NextWord()=='says')
           {   give self general;
               verb_wordnum=verb_wordnum+2;
           }
       ],
       orders
       [ i;  if (self hasnt general) "Charlotte sticks her tongue out.";
           WaveHands: "Charlotte waves energetically.";
           default: "^Don't know how,^ says Charlotte.";
       ],
       initial "Charlotte wants to play Simon Says.",
  has  animate female proper;
```

• **23** First add a `Clap` verb (this is easy). Then give Charlotte a `number` property (initially 0, say) and add these three lines to the end of Charlotte's `grammar` routine:

```
self.number=TryNumber(verb_wordnum);
if (self.number~=-1000)
{    action=##Clap; noun=0; second=0; rtrue; }
```

Her `orders` routine now needs a local variable called `i`, and the new clause:

```
Clap: if (self.number==0) "Charlotte folds her arms.";
      for (i=0:i<self.number:i++)
      {   print "Clap! ";
          if (i==100)
              print "(You must be regretting this by now.) ";
          if (i==200)
              print "(What a determined girl she is.) ";
      }
      if (self.number>100)
         "^^Charlotte is a bit out of breath now.";
     "^^~Easy!~ says Charlotte.";
```

• **24** The interesting point here is that when the `grammar` property finds the word "take", it accepts it and has to move `verb_wordnum` on by one to signal that a word has been parsed succesfully.

```
Nearby Dan "Dyslexic Dan"
  with name "dan" "dyslexic",
       grammar
       [;  if (verb_word == 'take') { verb_wordnum++; return 'drop'; }
           if (verb_word == 'drop') { verb_wordnum++; return 'take'; }
       ],
       orders
       [ i;
           Take: print_ret "~What,~ says Dan, ~ you want me to take ",
                     (the) noun, "?~";
           Drop: print_ret "~What,~ says Dan, ~ you want me to drop ",
                     (the) noun, "?~";
           Inv: "~That I can do,~ says Dan. ~I'm empty-handed.~";
           No: "~Right you be then.~";
           Yes: "~I'll be having to think about that.~";
           default: "~Don't know how,~ says Dan.";
       ],
       initial "Dyslexic Dan is here.",
  has  animate proper;
```

• **25** Suppose Dan's grammar (but nobody else's) for the "examine" verb is to be extended. His grammar routine should also contain:

```
if (verb_word == 'examine' or 'x')
{   verb_wordnum++; return -'danx,'; }
```

(Note the crudity of this: it looks at the actual verb word, so you have to check any synonyms yourself.) The verb "danx," must be declared later:

```
Verb "danx," * "conscience" -> Inv;
```

and now "Dan, examine conscience" will send him an `Inv` order: but "Dan, examine cow pie" will still send `Examine cow_pie` as usual.

• 26

```
[ PrintTime x; print (x/60), ":", (x%60)/10, (x%60)%10; ];
Nearby alarm_clock "alarm clock"
  with name "alarm" "clock",
       number 480,
       description
       [;  print "The alarm is ";
           if (self has general) print "on, "; else print "off, but ";
           print_ret "the clock reads ", (PrintTime) the_time,
                  " and the alarm is set for ", (PrintTime) self.number, ".";
       ],
       react_after
       [;  Inv:  if (self in player)   { new_line; <<Examine self>>; }
           Look: if (self in location) { new_line; <<Examine self>>; }
       ],
       daemon
       [;  if (the_time >= self.number && the_time <= self.number+3
                && self has general) "^Beep! Beep! The alarm goes off.";
       ],
       grammar [; return 'alarm,'; ],
       orders
       [;  SwitchOn:  give self general; StartDaemon(self); "~Alarm set.~";
           SwitchOff: give self ~general; StopDaemon(self); "~Alarm off.~";
           SetTo:     self.number=noun; <<Examine self>>;
           default: "~Commands are on, off or a time of day only, pliz.~";
       ],
       life
       [;  Ask, Answer, Tell:
              "[Try ~clock, something~ to address the clock.]";
       ],
  has  talkable;
```

and add a new verb to the grammar:

```
Verb "alarm," * "on"       -> SwitchOn
              * "off"      -> SwitchOff
              * TimeOfDay -> SetTo;
```

(using the `TimeOfDay` token from the exercises of §23). Note that since the word "alarm," can't be matched by anything the player types, this verb is concealed from ordinary grammar. The orders we produce here are not used in the ordinary way (for instance, the action `SwitchOn` with no `noun` or `second` would never ordinarily be produced by the parser) but this doesn't matter: it only matters that the grammar and the `orders` property agree with each other.

• 27

```
Nearby tricorder "tricorder"
  with name "tricorder",
       grammar [; return 'tc,'; ],
       orders
       [;  Examine: if (noun==player) "~You radiate life signs.~";
                print "~", (The) noun, " radiates ";
                if (noun hasnt animate) print "no ";
               "life signs.~";
           default: "The tricorder bleeps.";
       ],
       life
       [;  Ask, Answer, Tell: "The tricorder is too simple.";
       ],
  has  talkable;
...
Verb "tc,"    * noun      -> Examine;
```

• 28

```
Object replicator "replicator"
  with name "replicator",
       grammar [;  return 'rc,'; ],
       orders
       [;  Give:
                print_ret "The replicator serves up a cup of ",
                    (name) noun, " which you drink eagerly.";
           default: "The replicator is unable to oblige.";
       ],
       life
       [;  Ask, Answer, Tell: "The replicator has no conversation skill.";
       ],
  has  talkable;
Nearby earl_grey "Earl Grey tea"    with name "earl" "grey" "tea";
Nearby brandy    "Aldebaran brandy" with name "aldebaran" "brandy";
Nearby water     "distilled water"  with name "distilled" "water";
...
Verb "rc,"    * held      -> Give;
```

The point to note here is that the `held` token means 'held by the replicator' here, as the `actor` is the replicator, so this is a neat way of getting a 'one of the following phrases' token into the grammar.

• 29 This is similar to the previous exercises. One creates an attribute called `crewmember` and gives it to the crew objects: the `orders` property is

```
       orders
       [;  Examine:
                if (parent(noun)==0)
```

```
                    print_ret "~", (name) noun,
                        " is no longer aboard this demonstration game.~";
                print_ret "~", (name) noun, " is in ", (name) parent(noun), ".~";
            default: "The computer's only really good for locating the crew.";
        ],
```

and the grammar simply returns 'stc,' which is defined as

```
[ Crew i;
  switch(scope_stage)
  {  1: rfalse;
     2: for (i=selfobj+1:i<=top_object:i++)
            if (i has crewmember) PlaceInScope(i); rtrue;
  }
];
Verb "stc,"   * "where" "is" scope=Crew -> Examine;
```

An interesting point is that the scope routine doesn't need to do anything at stage 3 (usually used for printing out errors) because the normal error-message printing system is never reached. Something like "computer, where is Comminder Doto" causes a ##NotUnderstood order.

• 30

```
Object Zen "Zen" Flight_Deck
  with name "zen" "flight" "computer",
       initial "Square lights flicker unpredictably across a hexagonal \
                 fascia on one wall, indicating that Zen is on-line.",
       grammar [;  return 'zen,'; ],
       orders
       [;  Show: print_ret "The main screen shows a starfield, \
                      turning through ", noun, " degrees.";
           Go:  "~Confirmed.~  The ship turns to a new bearing.";
           SetTo: if (noun==0) "~Confirmed.~  The ship comes to a stop.";
               if (noun>12) print_ret "~Standard by ", (number) noun,
                                 " exceeds design tolerances.~";
               print_ret "~Confirmed.~  The ship's engines step to \
                   standard by ", (number) noun, ".";
           Take: if (noun~=force_wall) "~Please clarify.~";
                 "~Force wall raised.~";
           Drop: if (noun~=blasters)   "~Please clarify.~";
              "~Battle-computers on line.  \
                 Neutron blasters cleared for firing.~";
           default: "~Language banks unable to decode.~";
       ],
  has  talkable proper static;
Nearby force_wall "force wall"     with name "force" "wall" "shields";
Nearby blasters "neutron blasters" with name "neutron" "blasters";
...
Verb "zen,"   * "scan" number "orbital"        -> Show
              * "set" "course" "for" Planet    -> Go
              * "speed" "standard" "by" number -> SetTo
```

```
            * "raise" held                    -> Take
            * "clear" held "for" "firing"    -> Drop;
```

Dealing with `Ask`, `Answer` and `Tell` are left to the reader.

•31

```
    [ InScope;
       if (action_to_be == ##Examine or ##Show or ##ShowR)
           PlaceInScope(noslen_maharg);
       if (scope_reason == TALKING_REASON)
           PlaceInScope(noslen_maharg);
    ];
```

Note that `ShowR` is a variant form of `Show` in which the parameters are 'the other way round': thus "show maharg the phaser" generates `ShowR maharg phaser` internally, which is then converted to the more usual `Show phaser maharg`.

•32 Martha and the sealed room are defined as follows:

```
Object sealed_room "Sealed Room"
  with description
          "I'm in a sealed room, like a squash court without a door, \
           maybe six or seven yards across",
  has  light;
Nearby ball "red ball" with name "red" "ball";
Nearby martha "Martha"
  with name "martha",
       orders
       [ r; r=parent(self);
           Give:
               if (noun notin r) "~That's beyond my telekinesis.~";
               if (noun==self) "~Teleportation's too hard for me.~";
               move noun to player;
               print_ret "~Here goes...~ and Martha's telekinetic talents \
                   magically bring ", (the) noun, " to your hands.";
           Look:
               print "~", (string) r.description;
               if (children(r)==1) ".  There's nothing here but me.~";
               print ".  I can see ";
               WriteListFrom(child(r),CONCEAL_BIT+ENGLISH_BIT);
              ".~";
           default: "~Afraid I can't help you there.~";
       ],
       life
       [;  Ask: "~You're on your own this time.~";
           Tell: "Martha clucks sympathetically.";
           Answer: "~I'll be darned,~ Martha replies.";
       ],
   has animate female concealed proper;
```

but the really interesting part is the `InScope` routine to fix things up:

```
[ InScope actor;
   if (actor==martha) PlaceInScope(player);
   if (actor==player && scope_reason==TALKING_REASON)
       PlaceInScope(martha);
   rfalse;
];
```

Note that since we want two-way communication, the player has to be in scope to Martha too: otherwise Martha won't be able to follow the command "martha, give me the fish", because "me" will refer to something beyond her scope.

• **33** Just test if `HasLightSource(gift)==1`.

• **34** We could solve this using a daemon, but for the sake of demonstrating a feature of `thedark` we won't. In `Initialise`, write `thedark.initial = #r$GoMothGo;` and add the routine:

```
[ GoMothGo;
   if (moth in player)
   {   remove moth;
      "As your eyes try to adjust, you feel a ticklish sensation \
       and hear a tiny fluttering sound.";
   }
];
```

• **35** This is a crude implementation, for brevity (the real Zork thief has an enormous stock of attached messages). A `life` routine is omitted, and of course this particular thief steals nothing. See 'The Thief' for a much fuller, annotated implementation.

```
Nearby thief "thief"
  with name "thief" "gentleman" "mahu" "modo",
       each_turn "^The thief growls menacingly.",
       daemon
       [ i p j n k;
           if (random(3)~=1) rfalse;
           p=parent(thief);
           objectloop (i in compass)
           {   j=p.(i.door_dir);
               if (ZRegion(j)==1 && j hasnt door) n++;
           }
           if (n==0) rfalse;
           k=random(n); n=0;
           objectloop (i in compass)
           {   j=p.(i.door_dir);
               if (ZRegion(j)==1 && j hasnt door) n++;
               if (n==k)
               {   move self to j;
                   if (p==location) "^The thief stalks away!";
                   if (j==location) "^The thief stalks in!";
```

```
                rfalse;
            }
        }
    ],
  has  animate;
```

`ZRegion(j)` works out what kind of value `j` is: 1 means 'is a valid object number'. So the thief walks at random but never via doors, bridges and the like (because these may be locked or have rules attached); it's only a first approximation, and in a good game one should occasionally see the thief do something surprising, such as open a secret door. As for the `name`, note that 'The Prince of darkness is a gentleman. Modo he's called, and Mahu' (William Shakespeare, *King Lear* III iv).

• **36** First define a new property for object weight:

```
Property weight 10;
```

(10 being an average sort of weight). Containers weigh more when they hold things, so we will need:

```
[ WeightOf obj t i;
   t = obj.weight;
   objectloop (i in obj) t=t+WeightOf(i);
   return t;
];
```

Now for the daemon which monitors the player's fatigue:

```
Object weigher "weigher"
  with number 500,
       time_left 5,
       daemon
       [ w s b bw;
            w=WeightOf(player)-100-player.weight;
            s=self.number; s=s-w; if (s<0) s=0; if (s>500) s=500;
            self.number = s;
            if (s==0)
            {   bw=-1;
                objectloop(b in player)
                    if (WeightOf(b)>bw) { bw=WeightOf(b); w=b; }
                print "^Exhausted with carrying so much, you decide \
                    to discard ", (the) w, ": "; <<Drop w>>;
            }
            w=s/100; if (w==self.time_left) rfalse;
            if (w==3) print "^You are feeling a little tired.^";
            if (w==2) print "^You possessions are weighing you down.^";
            if (w==1) print "^Carrying so much weight is wearing you out.^";
            self.time_left = w;
       ];
```

Notice that items are actually dropped with `Drop` actions: one of them might be, say, a wild boar, which would bolt away into the forest when released. The daemon tries to drop the heaviest item. (Obviously a little improvement would be needed if the game contained, say, an un-droppable but very heavy ball and chain.) Now the daemon is going to run every turn forever, but needs to be started: so put `StartDaemon(weigher);` into the game's `Initialise` routine.

• **37** See the next answer.

• **38**

```
Object tiny_claws "sound of tiny claws" thedark
  with article "the",
       name "tiny" "claws" "sound" "of" "scuttling" "scuttle"
            "things" "creatures" "monsters" "insects",
       initial "Somewhere, tiny claws are scuttling.",
       before
       [;  Listen: "How intelligent they sound, for mere insects.";
           Touch, Taste: "You wouldn't want to.  Really.";
           Smell: "You can only smell your own fear.";
           Attack: "They easily evade your flailing about in the dark.";
           default: "The creatures evade you, chittering.";
       ],
       each_turn [; StartDaemon(self); ],
       number 0,
       daemon
       [ i; if (location~=thedark) { self.number=0; StopDaemon(self); rtrue; }
            i=self.number+1; self.number=i;
            switch(i)
            {   1: "^The scuttling draws a little nearer, and your breathing \
                     grows loud and hoarse.";
                2: "^The perspiration of terror runs off your brow.  The \
                     creatures are almost here!";
                3: "^You feel a tickling at your extremities and kick outward, \
                     shaking something chitinous off.  Their sound alone \
                     is a menacing rasp.";
                4: deadflag=1;
                   "^Suddenly there is a tiny pain, of a hypodermic-sharp fang \
                     at your calf.  Almost at once your limbs go into spasm, \
                     your shoulders and knee-joints lock, your tongue swells...";
            }
       ];
```

• **39** Either set a daemon to watch for `the_time` suddenly dropping, or put such a watch in the game's `TimePasses` routine.

• **40** A minimal solution is as follows:

```
Constant SUNRISE  360;  ! i.e., 6 am
Constant SUNSET  1140;  ! i.e., 7 pm
Attribute outdoors;     ! Give this to external locations
```

```
Attribute lit;          ! And this to artificially lit ones
Global day_state = 2;
[ TimePasses f obj;
  if (the_time >= SUNRISE && the_time < SUNSET) f=1;
  if (day_state == f) rfalse;
  for (obj=1: obj<=top_object: obj++)
  {   if (obj has lit) give obj light;
      if (obj has outdoors && obj hasnt lit)
      {   if (f==0) give obj ~light; else give obj light;
      }
  }
  if (day_state==2) { day_state = f; return; }
  day_state = f; if (location hasnt outdoors) return;
  if (f==1) "^The sun rises, illuminating the landscape!";
 "^As the sun sets, the landscape is plunged into darkness.";
];
```

In the `Initialise` routine, set the time (using `SetTime`) and then call `TimePasses` to set all the `light` attributes accordingly. Note that with this system, there's no need to set `light` at all: that's automatic.

• **41** Because you don't know what order daemons will run in. A 'fatigue' daemon which makes the player drop something might come after the 'mid-air' daemon has run for this turn. Whereas `each_turn` happens after daemons and timers have run their course, and can fairly assume no further movements will take place this turn.

• **42** It would have to provide its own code to keep track of time, and it can do this by providing a `TimePasses()` routine. Providing "time" or even "date" verbs to tell the player would also be a good idea.

• **43** Two reasons. Firstly, there are times when we want to be able to trap orders to other people, which `react_before` does not. Secondly, the player's `react_before` rule is not necessarily the first to react. In the case of the player's deafness, a cuckoo may have already used `react_before` to sing. But it would have been safe to use `GamePreRoutine`, if a little untidy (because a rule about the player would not be part of the player's definition, which makes for confusing source code). See §4 for the exact sequence of events when actions are processed.

• **44**

```
       orders
       [;  if (gasmask hasnt worn) rfalse;
           if (actor==self && action~=##Answer or ##Tell or ##Ask) rfalse;
          "Your speech is muffled into silence by the gas mask.";
       ],
```

• **45** The common man's *wayhel* was a lowly mouse. Since we think much more highly of the player:

```
Object hog "Warthog" Caldera
  with name "wart" "hog" "warthog", description "Muddy and grunting.",
       number 0,
```

```
        initial "A warthog snuffles and grunts about in the ash.",
        orders
        [; if (action~=##Go or ##Look or ##Examine)
               "Warthogs can't do anything as tricky as that!";
        ],
  has   animate proper;
```

and we just `ChangePlayer(warthog);`. Note that the same `orders` routine applies to the player-as-human typing "warthog, listen" as to the player-as-warthog typing just "listen".

• **46**

```
orders
[;  if (player==self)
    {   if (actor~=self)
            "You only become tongue-tied and gabble.";
        rfalse;
    }
    Attack: "The Giant looks at you with doleful eyes. \
              ~Me not be so bad!~";
    default: "The Giant is unable to comprehend your instructions.";
],
```

• **47** Give the "chessboard" room a `short_name` routine (it probably already has one, to print names like "Chessboard d6") and make it change the short name to "the gigantic Chessboard" if and only if `action` is currently set to `##Places`.

• **48** Put the following definition between inclusion of "Parser" and "Verblib":

```
Object LibraryMessages "lm"
  with before
       [;  Prompt: if (turns==1)
                       print "What should you, the detective, do now?^>";
                   else
                       print "What next?^>";
                   rtrue;
       ];
```

• **49** The details are left to the reader. One must provide a new grammar file (generating the same actions but from different syntax) and a very large `LibraryMessages` object.

• **50** Simply define the following (for accusative, nominative and capitalised nominative pronouns, respectively):

```
[ PronounAcc i;
    if (i hasnt animate) print "it";
    else { if (i has female) print "her"; else print "him"; } ];
[ PronounNom i;
    if (i hasnt animate) print "it";
```

```
    else { if (i has female) print "she"; else print "he"; } ];
[ CPronounNom i;
    if (i hasnt animate) print "It";
    else { if (i has female) print "She"; else print "He"; } ];
```

• **51** Use the `invent` routine to signal to `short_name` and `article` routines to change their usual habits:

```
invent
[;  if (inventory_stage==1) give self general;
    else give self ~general;
],
short_name
[;  if (self has general) { print "box"; rtrue; } ],
article
[;  if (self has general) { print "that hateful"; rtrue; }
    else print "a"; ],
```

• **52** This answer is cheating, as it needs to know about the `lookmode` variable (set to 1 for normal, 2 for verbose or 3 for superbrief). Simply include:

```
[ TimePasses;
  if (action~=##Look && lookmode==2) <Look>;
];
```

• **53**

```
[ DoubleInvSub i count1 count2;
  print "You are carrying ";
  objectloop (i in player)
  {   if (i hasnt worn) { give i workflag; count1++; }
      else { give i ~workflag; count2++; }
  }
  if (count1==0) print "nothing.";
  else
  WriteListFrom(child(player),
      FULLINV_BIT + ENGLISH_BIT + RECURSE_BIT + WORKFLAG_BIT);
  if (count2==0) ".";
  print ".  In addition, you are wearing ";
  objectloop (i in player)
  {   if (i hasnt worn) give i ~workflag; else give i workflag;
  }
  WriteListFrom(child(player),
      ENGLISH_BIT + RECURSE_BIT + WORKFLAG_BIT);
  ".";
];
```

•54

```
Attribute is_letter;
Class letter
  with list_together
       [;  if (inventory_stage==1)
           {   print "the letters ";
               if (c_style & ENGLISH_BIT == 0) c_style = c_style + ENGLISH_BIT;
              if (c_style & NOARTICLE_BIT == 0) c_style = c_style + NOARTICLE_BIT;
               if (c_style & NEWLINE_BIT ~= 0) c_style = c_style - NEWLINE_BIT;
               if (c_style & INDENT_BIT ~= 0)  c_style = c_style - INDENT_BIT;
           }
           else print " from a Scrabble set";
       ],
       short_name
       [;  if (listing_together has is_letter) rfalse;
           print "letter ", object self, " from a Scrabble set"; rtrue;
       ],
       article "the",
  has  is_letter;
```

and then as many letters as desired, along the lines of

```
Nearby s1 "X" class letter with name "x";
```

•55

```
Attribute is_coin;
Class  coin_class
  with name "coin",
       description "A round unstamped disc, presumably local currency.",
       parse_name
       [ i j w;
         if (parser_action==##TheSame)
         {   if ((parser_one.&name)-->0 == (parser_two.&name)-->0) return -1;
             return -2;
         }
         w=(self.&name)-->0;
         for (::i++)
         {   j=NextWord();
             if (j=='coins') parser_action=##PluralFound;
             else if (j~='coin' or w) return i;
         }
       ],
       list_together "coins",
       plural
       [;  print (string) (self.&name)-->0;
           if (listing_together hasnt is_coin) print " coins";
```

```
        ],
        short_name
        [;  if (listing_together has is_coin)
            {   print (string) (self.&name)-->0; rtrue; }
        ],
        article
        [;  if (listing_together has is_coin) print "one"; else print "a";
        ],
  has   is_coin;
Class  gold_coin_class   class coin_class with name "gold";
Class  silver_coin_class class coin_class with name "silver";
Class  bronze_coin_class class coin_class with name "bronze";
Nearby coin1 "silver coin" class silver_coin_class;
... and so on
```

• **56** For brevity, the following answer omits the routines: `CoinsTogether(attr)` which finds if the three coins with this `attr` (`is_gold` or `is_silver`) are together, returning 0 if they aren't and otherwise the object of which they are children; and `Trigram(attr)` which prints out the trigram currently showing on the coins of that `attr`, e.g., "Tails, Tails, Heads (Chen)".

```
Attribute is_gold; Attribute is_silver;
[ Face x; if (x.number==1) print "Heads"; else print "Tails"; ];
[ CoinsLT attr k i c;
  if (inventory_stage==1)
  {   if (attr==is_gold) print "the gold"; else print "the silver";
      print " coins ";
      k=CoinsTogether(attr);
      if (k==location)
      {   for (i=selfobj+1:i<=top_object:i++)
          {   if (i has attr)
              {   print (name) i;
                  switch(++c)
                  {  1: print ", "; 2: print " and ";
                     3: print " (showing the trigram ", (Trigram) attr, ")";
                  }
              }
          }
          rtrue;
      }
      if (c_style & ENGLISH_BIT == 0) c_style = c_style + ENGLISH_BIT;
      if (c_style & NOARTICLE_BIT == 0) c_style = c_style + NOARTICLE_BIT;
      if (c_style & NEWLINE_BIT ~= 0) c_style = c_style - NEWLINE_BIT;
      if (c_style & INDENT_BIT ~= 0)  c_style = c_style - INDENT_BIT;
  }
  rfalse;
];
Class  coin
  with number 1, article "the",
```

```
        parse_name
        [ i j w;
          if (parser_action==##TheSame) return -2;
          w='gold'; if (self has is_silver) w='silver';
          for (::i++)
          {   j=NextWord();
              if (j=='coins') parser_action=##PluralFound;
              else if (j~='coin' or w or self.name) return i;
          }
        ],
        after
        [ j;  Drop, PutOn:
                  self.number=random(2); print (Face) self, ". ";
                  if (self has is_gold) j=is_gold; else j=is_silver;
                  if (CoinsTogether(j)~=0)
                  {   print "The ";
                      if (j==is_gold) print "gold"; else print "silver";
                      print_ret " trigram is now ", (Trigram) j, ".";
                  }
                  new_line; rtrue;
        ];
Class  gold_coin class coin
 has   is_gold with list_together [; return CoinsLT(is_gold); ];
Class  silver_coin class coin
 has   is_silver with list_together [; return CoinsLT(is_silver); ];
...
Nearby goat "goat" class gold_coin with name "goat";
Nearby deer "deer" class gold_coin with name "deer";
Nearby chicken "chicken" class gold_coin with name "chicken";
Nearby robin "robin" class silver_coin with name "robin";
Nearby snake "snake" class silver_coin with name "snake";
Nearby bison "bison" class silver_coin with name "bison";
```

There are two unusual points here. Firstly, the `CoinsLT` routine is not simply given as the common `list_together` value in the `coin` class since, if it were, all six coins would be grouped together: we want two groups of three, so the gold and silver coins have to have different `list_together` values. Secondly, if a trigram is together and on the floor, it is not good enough to simply append text like "showing Tails, Heads, Heads (Tui)" at `inventory_stage` 2 since the coins may be listed in a funny order: for example, in the order snake, robin, bison. In that event, the order the coins are listed in doesn't correspond to the order their values are listed in, which is misleading. So instead `CoinsLT` takes over entirely at `inventory_stage` 1 and prints out the list of three itself, returning true to stop the list from being printed out by the library as well.

• 57

```
parse_name
[ i j w; if (self has general) j='red'; else j='green';
         w=NextWord();
         while (w==j or 'fried')
         {   w=NextWord(); i++;
```

```
        }
        if (w=='tomato') return i+1;
        return 0;
],
```

•58

```
Nearby princess "/?%?/ (the artiste formerly known as Princess)"
  with name "princess" "artiste" "formerly" "known" "as",
       short_name
       [;   if (self hasnt general) { print "Princess"; rtrue; }
       ],
       parse_name
       [ x; if (self hasnt general)
            {   if (NextWord()=='princess') return 1;
                return 0;
            }
            x=WordAddress(wn);
            if (   x->0 == '/' && x->1 == '?' && x->2 == '%'
                && x->3 == '?' && x->4 == '/') return 1;
            return -1;
       ],
       react_before
       [;  Listen: if (noun==0)
               print_ret (name) self, " sings a soft siren song.";
       ],
       life
       [;  Kiss: give self general; self.life = NULL;
              "In a fairy-tale transformation, the Princess \
               steps back and astonishes the world by announcing \
               that she will henceforth be known as ~/?%?/~.";
       ],
  has  animate proper female;
```

•59 Something to note here is that the button can't be called just "coffee" when the player's holding a cup of coffee: this means the game responds sensibly to the sequence "press coffee" and "drink coffee". Also note the way `itobj` is set to the delivered drink, so that "drink it" works nicely.

```
Nearby drinksmat "drinks machine",
  with name "drinks" "machine",
       initial
          "A drinks machine here has buttons for Cola, Coffee and Tea.",
  has  static;
Nearby thebutton "drinks machine button"
  has  scenery
 with  parse_name
```

```
        [ i flag type;
            for (: flag == 0: i++)
            {   flag = 1;
                switch(NextWord())
                {   'button', 'for': flag = 0;
                    'coffee': if (type == 0) { flag = 0; type = 1; }
                    'tea':    if (type == 0) { flag = 0; type = 2; }
                    'cola':   if (type == 0) { flag = 0; type = 3; }
                }
            }
            if (type==drink.number && i==2 && type~=0 && drink in player)
                return 0;
            self.number=type; return i-1;
        ],
        number 0,
        before
        [; Push, SwitchOn:
             if (self.number == 0)
                "You'll have to say which button to press.";
             if (parent(drink) ~= 0) "The machine's broken down.";
             drink.number = self.number; move drink to player; itobj = drink;
             print_ret "Whirr!  The machine puts ", (a) drink, " into your \
                 glad hands.";
           Attack: "The machine shudders and squirts cola at you.";
           Drink:  "You can't drink until you've worked the machine.";
        ];
Object  drink "drink"
  with  parse_name
        [ i flag type;
            for (: flag == 0: i++)
            {   flag = 1;
                switch(NextWord())
                {   'drink', 'cup', 'of': flag = 0;
                    'coffee': if (type == 0) { flag = 0; type = 1; }
                    'tea':    if (type == 0) { flag = 0; type = 2; }
                    'cola':   if (type == 0) { flag = 0; type = 3; }
                }
            }
            if (type ~= 0 && type ~= self.number) return 0;
            return i-1;
        ],
        short_name
        [;  print "cup of ";
            switch (self.number)
            { 1: print "coffee"; 2: print "tea"; 3: print "cola"; }
            rtrue;
        ],
        number 0,
```

```
before
[; Drink: remove self;
    "Ugh, that was awful.  You crumple the cup and responsibly \
     dispose of it.";
];
```

•60 Create a new property `adjective`, and move names which are adjectives to it: for instance,

```
name "tomato" "vegetable", adjective 'fried' 'green' 'cooked',
```

(Recall that dictionary words can only be written in " quotes for the `name` property.) Then (using the same `IsAWordIn` routine),

```
[ ParseNoun obj n m;
  while (IsAWordIn(NextWord(),obj,adjective) == 1) n++; wn--;
  while (IsAWordIn(NextWord(),obj,noun) == 1) m++;
  if (m==0) return 0; return n+m;
];
```

•61

```
[ ParseNoun obj;
  if (NextWord() == 'object' && TryNumber(wn) == obj) return 2;
  wn--; return -1;
];
```

•62

```
[ ParseNoun;
  if (WordLength(wn)==1 && WordAddress(wn)-->0 == '#') return 1;
  return -1;
];
```

•63

```
[ ParseNoun;
  if (WordLength(wn)==1 && WordAddress(wn)-->0 == '#') return 1;
  if (WordLength(wn)==1 && WordAddress(wn)-->0 == '*')
  {   parser_action = ##PluralFound; return 1; }
  return -1;
];
```

• **64** The trick is to convert "fly in amber" into "fly fly amber" (a harmless name) before the parser gets under way.

```
[ BeforeParsing i j;
  for (i=parse->1,j=2:j<i:j++)
  {   wn=j-1;
      if (NextWord()=='fly' && NextWord()=='in' && NextWord()=='amber')
          parse-->(j*2-1) = 'fly';
  }
];
```

• **65**

```
Global c_warned = 0;
Class  cherub_class
  with parse_name
       [ i j flag;
         for (flag=1:flag==1:flag=0)
         {   j=NextWord();
             if (j=='cherub' or j==self.name) flag=1;
             if (j=='cherubs' && c_warned==0)
             {   c_warned=1;
                 parser_action=##PluralFound; flag=1;
 print "(I'll let this go once, but the plural of cherub is cherubim.)^";
             }
             if (j=='cherubim')
             {   parser_action=##PluralFound; flag=1; }
             i++;
         }
         return i-1;
       ];
```

Then again, Shakespeare even writes "cherubins" in 'Twelfth Night', so who are we to censure?

• **66** Because the parser might go on to reject the line it's working on: for instance, if the player typed "inventory splurge" then the message "Shazam!" followed by a parser complaint will be somewhat unedifying.

• **67** Define two properties:

```
Property place_name;
Property to_places;
```

The scheme will work like this: a named room should have the `place_name` property set to a single dictionary word; say, the Bedquilt cave could be called `'bedquilt'`. Then in any room, a list of those other rooms which can be moved to in this way should appear in the `to_places` entry. For instance,

```
to_places Bedquilt Slab_Room Twopit_Room;
```

Now the code: see if a not-understood verb is a place name of a nearby room, and if so store that room's object number in goto_room, converting the verb to a dummy.

```
Global goto_room;
[ UnknownVerb word p i;
    p = location.&to_places; if (p==0) rfalse;
    for (i=0:(2*i)<location.#to_places:i++)
        if (word==(p-->i).place_name)
        {   goto_room = p-->i; return 'go#room';
        }
    rfalse;
];
[ PrintVerb word;
    if (word=='go#room')
    { print "go to "; PrintShortName(goto_room); rtrue; }
    rfalse;
];
```

(The supplied PrintVerb is icing on the cake: so the parser can say something like "I only understood you as far as wanting to go to Bedquilt." in reply to, say, "bedquilt the nugget".) It remains only to create the dummy verb:

```
[ GoRoomSub;
    if (goto_room hasnt visited) "But you have never been there.";
    PlayerTo(goto_room);
];
Verb "go#room"  *                                 -> GoRoom;
```

Note that if you don't know the way, you can't go there! A purist might prefer instead to not recognise the name of an unvisited room, back at the UnknownVerb stage, to avoid the player being able to deduce names of nearby rooms from this 'error message'.

• 68

```
Nearby genies_lamp "brass lamp"
  with name "brass" "lamp",
       before
       [; Rub: if (self hasnt general) give self general;
               else give self ~general;
               print_ret "A genie appears from the lamp, declaring:^^\
                          ~Mischief is my sole delight:^ \
                          If white means black, black means white!~^^\
                          She vanishes away with a vulgar parting wink.";
       ];
Nearby white_stone "white stone" with name "white" "stone";
Nearby black_stone "black stone" with name "black" "stone";
...
[ BeforeParsing;
   if (genies_lamp hasnt general) return;
   for (wn=1::)
   {   switch(NextWordStopped())
```

```
        {   'white': parse->(wn*2-3) = 'black';
            'black': parse->(wn*2-3) = 'white';
            -1: return;
        }
    }
];
```

•69

```
Constant MAX_FOOTNOTES 10;
Array footnotes_seen -> MAX_FOOTNOTES;
Global footnote_count;
[ Note n i pn;
    for (i=0:i<footnote_count:i++)
        if (n==footnotes_seen->i) pn=i;
    if (footnote_count==MAX_FOOTNOTES) "** MAX_FOOTNOTES exceeded! **";
    if (pn==0) { pn=footnote_count++; footnotes_seen->pn=n; }
    print " [",pn+1,"]";
];
[ FootnoteSub n;
    if (noun>footnote_count)
    {   print "No footnote [",noun,"] has been mentioned.^"; rtrue; }
    if (noun==0) "Footnotes count upward from 1.";
    n=footnotes_seen->(noun-1);
    print "[",noun,"]  ";
    switch(n)
    {   0: "This is a footnote.";
        1: "D.G.REG.F.D is inscribed around English coins.";
        2: "~Jackdaws love my big sphinx of quartz~, for example.";
    }
];
Verb "footnote" "note" * number              -> Footnote;
```

And then you can code, for instance,

```
    print "Her claim to the throne is in every pocket ", (Note) 1,
       ", her portrait in every wallet.";
```

•70 The general parsing routine needed is:

```
[ FrenchNumber n;
    switch(NextWord())
    {   'un', 'une': n=1;
        'deux': n=2;
        'trois': n=3;
        'quatre': n=4;
        'cinq': n=5;
        default: return -1;
```

```
    }
    parsed_number = n; return 1;
];
```

• 71 First we must decide how to store floating-point numbers internally: in this case we'll simply store $100x$ to represent x, so that "5.46" will be parsed as 546.

```
[ DigitNumber n type x;
  x = NextWordStopped(); if (x==-1) return -1; wn--;
  if (type==0)
  {   x = WordAddress(wn);
      if (x->n>='0' && x->n<='9') return (x->n) - '0';
      return -1;
  }
  if (x=='nought' or 'oh') { wn++; return 0; }
  x = TryNumber(wn++); if (x==-1000  x>=10) x=-1; return x;
];
[ FloatingPoint a x b w d1 d2 d3 type;
  a = TryNumber(wn++);
  if (a==-1000) return -1;
  w = NextWordStopped(wn); if (w==-1) return a*100;
  x = NextWordStopped(wn); if (x==-1) return -1; wn--;
  if (w=='point') type=1;
  else
  {   if (WordAddress(wn-1)->0~='.'  WordLength(wn-1)~=1)
          return -1;
  }
  d1 = DigitNumber(0,type);
  if (d1==-1) return -1;
  d2 = DigitNumber(1,type); d3 = DigitNumber(2,type);
  b=d1*10; if (d2>=0) b=b+d2; else d3=0;
  if (type==1)
  {   x=1; while (DigitNumber(x,type)>=0) x++; wn--;
  }
  else wn++;
  parsed_number = a*100 + b;
  if (d3>=5) parsed_number++;
  return 1;
];
```

• 72 Again, the first question is how to store the number dialled: in this case, into a `string` array. The token is:

```
Constant MAX_PHONE_LENGTH 30;
Array dialled_number string MAX_PHONE_LENGTH;
[ PhoneNumber f a l ch pp i;
  pp=1; if (NextWordStopped()==-1) return 0;
```

```
  do
  {   a=WordAddress(wn-1); l=WordLength(wn-1);
      for (i=0:i<l:i++)
      {   ch=a->i;
          if (ch>='0' && ch<='9')
          {   if (pp<MAX_PHONE_LENGTH) dialled_number->(pp++)=ch-'0';
          }
          else
          {   if (ch~='-') f=1; if (i~=0) return -1;   }
      }
      if (f==1)
      {   if (pp==1) return -1; dialled_number->0 = pp-1; return 0; }
  } until (NextWordStopped()==-1);
  if (pp==1) return -1;
  dialled_number->0 = pp-1;
  return 0;
];
```

To demonstrate this in use,

```
[ DialPhoneSub i;
  print "You dialled <";
  for (i=1:i<=dialled_number->0:i++) print dialled_number->i;
  ">";
];
Verb "dial"  * PhoneNumber -> DialPhone;
```

• **73** The time of day will be returned as a number in the usual Inform time format: as hours times 60 plus minutes (on the 24-hour clock, so that the 'hour' part is between 0 and 23).

```
Constant TWELVE_HOURS 720;
[ NumericTime hr mn word x;
  if (hr>=24) return -1;
  if (mn>=60) return -1;
  x=hr*60+mn; if (hr>=13) return x;
  x=x%TWELVE_HOURS; if (word=='pm') x=x+TWELVE_HOURS;
  if (word~='am' or 'pm' && hr==12) x=x+TWELVE_HOURS;
  return x;
];
[ MyTryNumber wordnum i j;
  i=wn; wn=wordnum; j=NextWordStopped(); wn=i;
  switch(j)
  {   'twenty-five': return 25;
      'thirty': return 30;
      default: return TryNumber(wordnum);
  }
];
[ TimeOfDay i j k flag loop ch hr mn;
```

```
i=NextWord();
switch(i)
{  'midnight': parsed_number=0; return 1;
   'midday', 'noon': parsed_number=TWELVE_HOURS; return 1;
}
!   Next try the format 12:02
j=WordAddress(wn-1); k=WordLength(wn-1);
flag=0;
for (loop=0:loop<k:loop++)
{   ch=j->loop;
    if (ch==':' && flag==0 && loop~=0 && loop~=k-1) flag=1;
    else { if (ch<'0') flag=-1; if (ch>'9') flag=-1; }
}
if (k<3) flag=0; if (k>5) flag=0;
if (flag==1)
{   for (loop=0:j->loop~=':':loop++, hr=hr*10)
        hr=hr+j->loop-'0';
    hr=hr/10;
    for (loop++:loop<k:loop++, mn=mn*10)
        mn=mn+j->loop-'0';
    mn=mn/10;
    j=NextWordStopped();
    parsed_number=NumericTime(hr, mn, j);
    if (parsed_number<0) return -1;
    if (j~='pm' or 'am') wn--;
    return 1;
}
!   Next the format "half past 12"
j=-1; if (i=='half') j=30; if (i=='quarter') j=15;
if (j<0) j=MyTryNumber(wn-1); if (j<0) return -1;
if (j>=60) return -1;
k=NextWordStopped();
if (k==-1)
{   hr=j; if (hr>12) return -1; jump TimeFound; }
if (k=='o^clock' or 'am' or 'pm')
{   hr=j; if (hr>12) return -1; jump TimeFound; }
if (k=='to' or 'past')
{   mn=j; hr=MyTryNumber(wn);
    if (hr<=0)
    {   switch(NextWordStopped())
        {   'noon', 'midday': hr=12;
            'midnight': hr=0;
            default: return -1;
        }
    }
    if (hr>=13) return -1;
    if (k=='to') { mn=60-mn; hr=hr-1; if (hr==-1) hr=23; }
    wn++; k=NextWordStopped();
```

```
        jump TimeFound;
    }
    hr=j; mn=MyTryNumber(--wn);
    if (mn<0) return -1; if (mn>=60) return -1;
    wn++; k=NextWordStopped();
   .TimeFound;
    parsed_number = NumericTime(hr, mn, k);
    if (parsed_number<0) return -1;
    if (k~='pm' or 'am' or 'o^clock') wn--;
    return 1;
];
```

• **74** Here goes: we could implement the buttons with five separate objects, essentially duplicates of each other. (And by using a class definition, this wouldn't look too bad.) But if there were 500 slides this would be less reasonable.

```
[ ASlide w n;
   if (location~=Machine_Room) return -1;
   w=NextWord(); if (w=='slide') w=NextWord();
   switch(w)
   {   'first', 'one': n=1;
       'second', 'two': n=2;
       'third', 'three': n=3;
       'fourth', 'four': n=4;
       'fifth', 'five': n=5;
       default: return -1;             !  Failure!
   }
   w=NextWord(); if (w~='slide') wn--;    !  (Leaving word counter at the
                                            !  first misunderstood word)
   parsed_number=n;
   return 1;                              !  Success!
];
Global slide_settings --> 5;             !  A five-word array
[ SetSlideSub;
   slide_settings-->(noun-1) = second;
   print_ret "You set slide ", (number) noun,
             " to the value ", second, ".";
];
[ XSlideSub;
   print_ret "Slide ", (number) noun, " currently stands at ",
       slide_settings-->(noun-1), ".";
];
Extend "set" first
           * ASlide "to" number                  -> SetSlide;
Extend "push" first
           * ASlide "to" number                  -> SetSlide;
Extend "examine" first
           * ASlide                              -> XSlide;
```

• **75** (See the `Parser` file.) `NextWord` roughly returns `parse-->(w*2-1)` (but it worries a bit about commas and full stops).

```
[ WordAddress w; return buffer + parse->(w*4+1); ];
[ WordLength w; return parse->(w*4); ];
```

• **76** (Cf. the blackboard code in 'Toyshop'.)

```
Global from_char; Global to_char;
[ QuotedText i j f;
   i = parse->((++wn)*4-3);
   if (buffer->i=='"')
   {   for (j=i+1:j<=(buffer->1)+1:j++)
           if (buffer->j=='"') f=j;
       if (f==0) return -1;
       from_char = i+1; to_char=f-1;
       if (from_char>to_char) return -1;
       while (f> (parse->(wn*4-3))) wn++; wn++;
       return 0;
   }
   return -1;
];
```

Note that in the case of success, the word marker `wn` is moved beyond the last word accepted (since the Z-machine automatically tokenises a double-quote as a single word). The text is treated as though it were a preposition, and the positions where the quoted text starts and finishes in the raw text `buffer` are recorded, so that an action routine can easily extract the text and use it later. (Note that `""` with no text inside is not matched by this routine but only because the last `if` statement throws out that one case.)

• **77**

```
[ NeverMatch; return -1; ];
```

• **78** Perhaps to arrange better error messages when the text has failed all the 'real' grammar lines of a verb (see 'Encyclopaedia Frobozzica' for an example).

• **79** (See the `NounDomain` specification in §36.) This routine passes on any `REPARSE_CODE`, as it must, but keeps a matched object in its own `third` variable, returning the 'skip this text' code to the parser. Thus the parser never sees any third parameter.

```
Global third;
[ ThirdNoun x;
  x=NounDomain(player,location,0);
  if (x==REPARSE_CODE) return x; if (x==0) return -1; third = x;
  return 0;
];
```

• 80

```
Global scope_count;
[ PrintIt obj; print_ret ++scope_count, ": ", (a) obj, " (", obj, ")"; ];
[ ScopeSub; LoopOverScope(#r$PrintIt);
  if (scope_count==0) "Nothing is in scope.";
];
Verb meta "scope" *                                  -> Scope;
```

• 81

```
[ MegaExam obj; print "^", (a) obj, ": "; <Examine obj>; ];
[ MegaLookSub; <Look>; LoopOverScope(#r$MegaExam); ];
Verb meta "megalook" *                               -> MegaLook;
```

• 82 A slight refinement of such a "purloin" verb is already defined in the library (if the constant `DEBUG` is defined), so there's no need. But here's how it could be done:

```
[ Anything i;
  if (scope_stage==1) rfalse;
  if (scope_stage==2)
  {   for (i=1:i<=top_object:i++) PlaceInScope(i); rtrue; }
  "No such in game.";
];
```

(This disallows multiple matches for efficiency reasons – the parser has enough work to do with such a huge scope definition as it is.) Now the token `scope=Anything` will match anything at all, even things like the abstract concept of 'east'.

• 83 Note the sneaky way looking through the window is implemented, and that the 'on the other side' part of the room description isn't printed in that case.

```
Property far_side;
Class  window_room
  with description
          "This is one end of a long east/west room.",
       before
       [;  Examine, Search: ;
           default:
             if (inp1~=1 && noun~=0 && noun in self.far_side)
                 print_ret (The) noun, " is on the far side of \
                    the glass.";
             if (inp2~=1 && second~=0 && second in self.far_side)
                 print_ret (The) second, " is on the far side of \
                    the glass.";
       ],
       after
       [;  Look:
             if (ggw has general) rfalse;
```

```
             print "^The room is divided by a great glass window";
             if (location.far_side hasnt light) " onto darkness.";
             print ", stretching from floor to ceiling.^";
             if (Locale(location.far_side,
                    "Beyond the glass you can see",
                    "Beyond the glass you can also see")~=0) ".";
      ],
  has  light;
Object window_w "West of Window" class window_room
  with far_side window_e;
Object window_e "East of Window" class window_room
  with far_side window_w;
Object ggw "great glass window"
  with name "great" "glass" "window",
       before
       [ place; Examine, Search: place=location;
               if (place.far_side hasnt light)
                   "The other side is dark.";
               give self general;
               PlayerTo(place.far_side,1); <Look>; PlayerTo(place,1);
               give self ~general;
               give place.far_side ~visited; rtrue;
       ],
       found_in window_w window_e,
  has  scenery;
```

A few words about `inp1` and `inp2` are in order. `noun` and `second` can hold either objects or numbers, and it's sometimes useful to know which. `inp1` is equal to `noun` if that's an object, or 1 if that's a number; likewise for `inp2` and `second`. (In this case we're just being careful that the action `SetTo eggtimer 35` wouldn't be stopped if object 35 happened to be on the other side of the glass.) We also need:

```
[ InScope actor;
   if (actor in window_w && window_e has light) ScopeWithin(window_e);
   if (actor in window_e && window_w has light) ScopeWithin(window_w);
   rfalse;
];
```

• **84** For good measure, we'll combine this with the previous rule about `moved` objects being in scope in the dark. The following can be inserted into the 'Shell' game:

```
Object coal "dull coal" Blank_Room
  with name "dull" "coal";
Object Dark_Room "Dark Room"
  with description "An empty room with a west exit.",
       each_turn
       [; if (self has general) self.each_turn=0;
          else "^You hear the breathing of a dwarf.";
       ],
```

```
       w_to Blank_Room;
Nearby light_switch "light switch"
  with name "light" "switch",
       initial "On one wall is the light switch.",
       after
       [; SwitchOn: give Dark_Room light;
          SwitchOff: give Dark_Room ~light;
       ],
  has  switchable static;
Nearby diamond "shiny diamond"
  with name "shiny" "diamond"
  has  scored;
Nearby dwarf "dwarf"
  with name "voice" "dwarf",
       life
       [; Order: if (action==##SwitchOn && noun==light_switch)
                 {   give Dark_Room light general;
                     give light_switch on; "~Right you are, squire.~";
                 }
       ],
  has  animate;
[ InScope person i;
  if (parent(person)==Dark_Room)
  {   if (person==dwarf  Dark_Room has general)
          PlaceInScope(light_switch);
  }
  if (person==player && location==thedark)
      objectloop (i near player)
          if (i has moved  i==dwarf)
              PlaceInScope(i);
  rfalse;
];
```

Note that the routine puts the light switch in scope for the dwarf – if it didn't, the dwarf would not be able to understand "dwarf, turn light on", and that was the whole point.

• **85** In the `Initialise` routine, move `newplay` somewhere and `ChangePlayer` to it, where:

```
Object newplay "yourself"
  with description "As good-looking as ever.", number 0,
       add_to_scope nose,
       capacity 5,
       before
       [;  Inv: if (nose has general) print "You're holding your nose.  ";
           Smell: if (nose has general)
                     "You can't smell a thing with your nose held.";
       ],
  has  concealed animate proper transparent;
Object nose "nose"
  with name "nose", article "your",
```

```
        before
        [; Take: if (self has general)
                     "You're already holding your nose.";
                  if (children(player) > 1) "You haven't a free hand.";
                  give self general; player.capacity=1;
                 "You hold your nose with your spare hand.";
           Drop: if (self hasnt general) "But you weren't holding it!";
                  give self ~general; player.capacity=5;
                  print "You release your nose and inhale again.  ";
                  <<Smell>>;
        ],
  has  scenery;
```

•86

```
Object steriliser "sterilising machine"
  with name "washing" "sterilising" "machine",
       add_to_scope  top_of_wm  go_button,
       before
       [;  PushDir: AllowPushDir(); rtrue;
               Receive:
                   if (receive_action==##PutOn)
                       <<PutOn noun top_of_wm>>;
           SwitchOn: <<Push go_button>>;
       ],
       after
       [;  PushDir: "It's hard work, but the steriliser does roll.";
       ],
       initial
       [;  print "There is a sterilising machine on casters here (a kind of \
               chemist's washing machine) with a ~go~ button.  ";
           if (children(top_of_wm)~=0)
           {   print "On top";
               WriteListFrom(child(top_of_wm), ISARE_BIT + ENGLISH_BIT);
               print ".  ";
           }
           if (children(self)~=0)
           {   print "Inside";
               WriteListFrom(child(self), ISARE_BIT + ENGLISH_BIT);
               print ".  ";
           }
       ],
  has  static container open openable;
Object top_of_wm "top of the sterilising machine",
  with article "the",
  has  static supporter;
Object go_button "~go~ button"
  with name "go" "button",
```

```
      before [; Push, SwitchOn: "The power is off."; ],
  has  static;
```

• **87** The label object itself is not too bad:

```
Nearby label "red sticky label"
  with name "red" "sticky" "label",
       number 0,
       before
       [;  PutOn, Insert:
               if (self.number~=0)
               {   print "(first removing the label from ",
                   (the) self.number, ")^"; self.number=0; move self to player;
               }
               if (second==self) "That would only make a red mess.";
               self.number=second; remove self;
               print_ret "You affix the label to ", (the) second, ".";
       ],
       react_after
       [ x; x=self.number; if (x==0) rfalse;
           Look: if (x in location)
                   print "^The red sticky label is stuck to ", (the) x, ".^";
           Inv:  if (x in player)
                   print "^The red sticky label is stuck to ", (the) x, ".^";
       ],
       each_turn
       [;  if (parent(self)~=0) self.number=0; ];
```

Note that `label.number` holds the object the label is stuck to, or 0 if it's unstuck: and that when it is stuck, it is removed from the object tree. It therefore has to be moved into scope, so we need the rule: if the labelled object is in scope, then so is the label.

```
Global disable_self;
[ InScope actor i1 i2;
  if (label.number==0) rfalse; if (disable_self==1) rfalse;
  disable_self=1;
  i1 = TestScope(label, actor);
  i2 = TestScope(label.number, actor);
  disable_self=0;
  if (i1~=0) rfalse;
  if (i2~=0) PlaceInScope(label);
  rfalse;
];
```

This routine has two interesting points: firstly, it disables itself while testing scope (since otherwise the game would go into an endless recursion), and secondly it only puts the label in scope if it isn't already there. This is just a safety precaution to prevent the label reacting twice to actions (and isn't really necessary since the label can't already be in scope, but is included for the sake of example).

•88 Firstly, create an attribute `is_key` and give it to all the keys in the game. Then:

```
Global assumed_key;
[ DefaultLockSub;
  print "(with ", (the) assumed_key, ")^"; <<Lock noun assumed_key>>;
];
[ DefaultLockTest i count;
  if (noun hasnt lockable) rfalse;
  objectloop (i in player)
      if (i has is_key) { count++; assumed_key = i; }
  if (count==1) rtrue; rfalse;
];
Extend "lock" first * noun = DefaultLockTest -> DefaultLock;
```

(and similar code for "unlock"). Note that "lock strongbox" is matched by this new grammar line only if the player only has one key: the `DefaultLock strongbox` action is generated: which is converted to, say, `Lock strongbox brass_key`.

•89

```
Array quote_done -> 50;
Global next_quote = -1;
[ Quote i;
  if (quote_done->i==0) { quote_done->i = 1; next_quote = i; }
];
[ AfterPrompt;
  switch(next_quote)
  {   0: box "His stride is wildernesses of freedom:"
             "The world rolls under the long thrust of his heel."
             "Over the cage floor the horizons come."
             ""
             "-- Ted Hughes, ~The Jaguar~";
      1: ...
  }
  next_quote = -1;
];
```

•90 Note the magic line of assembly code here, which only works for Advanced games:

```
[ GiveHint hint keypress;
  print (string) hint; new_line; new_line;
  @read_char 1 0 0 keypress;
  if (keypress == 'H' or 'h') rfalse;
  rtrue;
];
```

And a typical menu item using it:

```
 if (menu_item==1)
 {   print "(Press ENTER to return to menu, or H for another hint.)^^";
     if (GiveHint("(1/3)  What kind of bird is it, exactly?")==1) return 2;
```

```
    if (GiveHint("(2/3)  Magpies are attracted by shiny items.")==1) return 2;
    "(3/3)  Wave at the magpie with the kitchen foil.";
}
```

• **91** By encoding the character into a byte array and using `@save` and `@restore`. The numbers in this array might contain the character's name, rank and abilities, together with some coding system to show what possessions the character has (a brass lamp, 50 feet of rope, etc.)

• **92** Note that we wait for a space character (32) or either kind of new-line which typical ASCII keyboards produce (10 or 13), just to be on the safe side:

```
[ TitlePage i;
   @erase_window -1; print "^^^^^^^^^^^^^";
   i = 0->33; if (i==0) i=80; i=(i-50)/2;
   style bold; font off; spaces(i);
   print "                      RUINS^";
   style roman; print "^^"; spaces(i);
   print "          [Please press SPACE to begin.]^";
   font on;
   box "And make your chronicle as rich with praise"
       "As is the ooze and bottom of the sea"
       "With sunken wreck and sumless treasures."
       ""
       "-- William Shakespeare, ~Henry V~ I. ii. 163";
   do { @read_char 1 0 0 i; } until (i==32 or 10 or 13);
   @erase_window -1;
];
```

• **93** First put the directive `Replace DrawStatusLine;` before including the library; define the global variable `invisible_status` somewhere. Then give the following redefinition:

```
[ DrawStatusLine i width posa posb;
   if (invisible_status==1) return;
   @split_window 1; @set_window 1; @set_cursor 1 1; style reverse;
   width = 0->33; posa = width-26; posb = width-13;
   spaces (width-1);
   @set_cursor 1 2;  PrintShortName(location);
   if (width > 76)
   {   @set_cursor 1 posa; print "Score: ", sline1;
       @set_cursor 1 posb; print "Moves: ", sline2;
   }
   if (width > 63 && width <= 76)
   {   @set_cursor 1 posb; print sline1, "/", sline2;
   }
   @set_cursor 1 1; style roman; @set_window 0;
];
```

• **94** First put the directive `Replace DrawStatusLine;` before including the library. Then add the following routine anywhere after `treasures_found`, an 'Advent' variable, is defined:

```
[ DrawStatusLine;
   @split_window 1; @set_window 1; @set_cursor 1 1; style reverse;
   spaces (0->33)-1;
   @set_cursor 1 2;  PrintShortName(location);
   if (treasures_found > 0)
   {   @set_cursor 1 50; print "Treasure: ", treasures_found;
   }
   @set_cursor 1 1; style roman; @set_window 0;
];
```

• **95** `Replace` with the following. (Note the use of `@@92` as a string escape, to include a literal backslash character, and `@@124` for a vertical line.)

```
Constant U_POS 28; Constant W_POS 30; Constant C_POS 31;
Constant E_POS 32; Constant IN_POS 34;
[ DrawStatusLine i;
    @split_window 3; @set_window 1; style reverse; font off;
    @set_cursor 1 1; spaces (0->33)-1;
    @set_cursor 2 1; spaces (0->33)-1;
    @set_cursor 3 1; spaces (0->33)-1;
    @set_cursor 1 2;  print (name) location;
    @set_cursor 1 51; print "Score: ", sline1;
    @set_cursor 1 64; print "Moves: ", sline2;
    if (location ~= thedark)
    {   ! First line
        if (location.u_to ~= 0)  { @set_cursor 1 U_POS; print "U"; }
        if (location.nw_to ~= 0) { @set_cursor 1 W_POS; print "@@92"; }
        if (location.n_to ~= 0)  { @set_cursor 1 C_POS; print "@@124"; }
        if (location.ne_to ~= 0) { @set_cursor 1 E_POS; print "/"; }
        if (location.in_to ~= 0) { @set_cursor 1 IN_POS; print "I"; }
        ! Second line
        if (location.w_to ~= 0)  { @set_cursor 2 W_POS; print "-"; }
                                   @set_cursor 2 C_POS; print "o";
        if (location.e_to ~= 0)  { @set_cursor 2 E_POS; print "-"; }
        ! Third line
        if (location.d_to ~= 0)  { @set_cursor 3 U_POS; print "D"; }
        if (location.sw_to ~= 0) { @set_cursor 3 W_POS; print "/"; }
        if (location.s_to ~= 0)  { @set_cursor 3 C_POS; print "@@124"; }
        if (location.se_to ~= 0) { @set_cursor 3 E_POS; print "@@92"; }
        if (location.out_to ~= 0){ @set_cursor 3 IN_POS; print "O"; }
    }
    @set_cursor 1 1; style roman; @set_window 0; font on;
];
```

• **96** The tricky part is working out the number of characters in the location name, and this is where `@output_stream` is so useful. This time `Replace` with:

```
Array printed_text table 64;
[ DrawStatusLine i j;
  i = 0->33; if (i==0) i=80;
  font off;
  @split_window 1; @buffer_mode 0; @set_window 1;
  style reverse; @set_cursor 1 1; spaces(i);
  printed_text-->0 = 64;
  @output_stream 3 printed_text;
  print (name) location;
  @output_stream -3;
  j=(i-(printed_text-->0))/2;
  @set_cursor 1 j; print (name) location; spaces(j-1);
  style roman;
  @buffer_mode 1; @set_window 0; font on;
];
```

Note that the table can hold 128 characters (plenty for this purpose), and that these are stored in `printed_text->2` to `printed_text->129`; the length printed is held in `printed_text-->0`. ('Trinity' actually does this more crudely, storing away the width of each location name.)

• **97** The following implementation is limited to a format string $2 \times 64 = 128$ characters long, and six subsequent arguments. `%d` becomes a decimal number, `%e` an English one; `%c` a character, `%%` a (single) percentage sign and `%s` a string.

```
Array printed_text table 64;
Array printf_vals --> 6;
[ Printf format p1 p2 p3 p4 p5 p6   pc j k;
  printf_vals-->0 = p1; printf_vals-->1 = p2; printf_vals-->2 = p3;
  printf_vals-->3 = p4; printf_vals-->4 = p5; printf_vals-->5 = p6;
  printed_text-->0 = 64; @output_stream 3 printed_text;
  print (string) format; @output_stream -3;
  j=printed_text-->0;
  for (k=2:k<j+2:k++)
  {   if (printed_text->k == '%')
      {   switch(printed_text->(++k))
          {   '%': print "%";
              'c': print (char) printf_vals-->pc++;
              'd': print printf_vals-->pc++;
              'e': print (number) printf_vals-->pc++;
              's': print (string) printf_vals-->pc++;
              default: print "<** Unknown printf escape **>";
          }
      }
      else print (char) printed_text->k;
  }
];
```

• **98** `Primes(100)`, where:

```
[ Primes i j k l;
  for (j=2:j<=i:j++)
  {   print j, " : "; l=j;
      while (l > 1)
      for (k=2:k<=l:k++)
          if (l%k == 0) { l=l/k; print k, " "; break; }
      new_line;
  }
];
```

(which was the first algorithm ever compiled by Inform).

Index