The Inform Designer's Manual

by Graham Nelson

Third edition
4 September 1996
as updated 16 May 1997

Contents

Chapter II: Using the Compiler

BOOK TWO: DESIGNING

Chapter III: Fundamentals

Chapter IV: The Model World

Chapter V: Describing and Parsing

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 a system for creating adventure games, and this is the book to read about it.

Infocom format 'story files' (adventure games, that is) can be played on almost any computer, from personal organisers to mainframes, with the aid of 'interpreter' programs. The task of the Inform 'compiler' is to translate a textual description of a game into a story file. The result will play identically on any machine of any model.

Inform is a suite of software, called the 'library', as well as a compiler. Without the library, it would be a major undertaking to write a description of even the smallest game. The library has two ingredients: the parser, a program for translating written English inputs into a form games can more easily understand, and the "world model", a complex web of rules common to all adventure games. Given these, the designer only needs to describe things and give any exceptional rules that apply. ("There is a bird here, which is a normal item except that you can't pick it up.")

The library is rich in detail. The parser recognises over 80 distinct verbs and a vocabulary of about 300 words even before any rooms or objects are created, and is programmable and highly flexible. It can handle ambiguities, clarify its input by asking questions ("Which key do you mean...?") and can cope properly with plurals, vagueness, conversation, pronouns and the player becoming someone else in mid-game. It can be configured to languages other than English. The world-model includes rooms, items, vehicles, duplicates, containers, doors, things on top of other things, light and darkness, switching things on and off, opening, closing and locking things, looking up information in books, entering things, scoring and so forth.

Just as Inform has two strands – compiler and library – so this manual has two parts: Programming and Designing. In Book One, small computer programs are written to perform simple calculations, never using the library. Subsections listed *in slanted text* on the detailed contents page become technical but the rest is pitched at newcomers and can be skim-read by anyone with prior experience of a programming language such as C or Pascal. Book Two is entirely about making games.

Newcomers are invited to work through §1 and §7, the "getting started" sections in Books One and Two, before reading much more of either.

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.) Passages which divert the main story, usually to tell an unexpurgated truth which may confuse or bore a newcomer, are marked with a warning triangle △ or two, and set in smaller type. Examples of program are set in `typewriter` font. Mundane or irrelevant passages in longer examples are sometimes replaced with a line reading just "`...`"

To keep Book Two from clogging up with examples, many are set as "exercises", with "answers" given in full at the back of the book. Harder exercises are marked with triangles and some are very hard indeed. I emphasize that the exercises are often intended as a way of presenting answers to deliberately difficult questions, to assist experts: the curse of Adventure design-languages is the feature which is ideal for the simple but too inflexible to cope with the complicated. For a list of exercises with page references to question and answer, see under "exercises" in the Index.

A better tutorial than attempting the exercises, then, is probably to make a simple game, as demonstrated in Chapter III, and then add an example of each new feature as you work through Chapters IV and V.

Many sections end with a 'References' paragraph referring to yet more examples which can be found in Inform's demonstration games. 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: more an incoherent collection of unusual objects). In addition, the little game 'Ruins' is developed in the course of Chapters III and IV of this manual.

Finally, the "game" called 'Museum of Inform' simulates a museum whose exhibits are solutions to the exercises in this manual.

Copyright on Inform, the program and its source code, its example games and documentation (including this book) is retained by Graham Nelson, who asserts the moral right to be identified as the author under the Copyrights, Designs and Patents Act 1988. Having said this, I am happy for it to be freely distributed to anybody who wants a copy, provided that: (a) distributed copies are not substantially different from those archived by the author, (b) this and other copyright messages are always retained in full, and (c) no profit is involved. (Exceptions to these rules must be negotiated directly with the author.) However, a story file produced with the Inform compiler (and libraries) then belongs to its author, and may be sold for profit if desired, provided that its game banner contains the information that it was compiled by Inform, and the Inform version number.

The Internet source for Inform material (executables of the compiler for different machines, source code, the library files and example games) is the German National Research Centre for Computer Science, where Volker Blasius maintains an archive at the anonymous FTP site `ftp.gmd.de`. Inform can be found at:

```
ftp://ftp.gmd.de/if-archive/infocom/compilers/inform6
```

Another useful resource is the Inform 6 home page on the 'World Wide Web', which includes Gareth Rees's 'Alice' tutorial, located at:

```
http://www.gnelson.demon.co.uk/inform.html
```

This manual describes Inform release 6.13 (or later), using library release 6/5 (or later). Earlier Inform 6 compilers and libraries are very similar but Inform 5.5 and 5/12 are very different.

This manual has evolved from seven earlier publications, once rather makeshift and sometimes defensive ("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 four 'companion volumes'. *The Craft of Adventure* is an essay on the design of adventure games; *The Z-Machine Standards Document* minutely covers the run-time format and Inform assembly language, its lowest level; and *The Inform Technical Manual* documents Inform's internal working in great detail, and includes a formal context-free grammar for the Inform language. The *Inform Translator's Manual* describes how to write a language definition file for games which speak languages other than English.

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 close of the 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, the ever avuncular Bob Newell, Robert Pelak, Gareth Rees, Jørund Rian, Dilip Sequeira, Richard Tucker, Christopher Wichura and John Wood.

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

Graham Nelson
Oxford
April 1993 – May 1997

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: The Inform programming language

> 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)

1 The language of routines

§1.1 First principles

This chapter aims to introduce beginners to Inform as though it were a general-purpose programming language (rather than a tool for designing adventure games). The examples given will be short programs performing simple calculations (rather than games). To begin with, the Inform language is:

1. **Compiled.**
   That is, the Inform compiler translates text written by the author (called the "source code") into a program (called the "object code" since it is the object of the exercise). This translation is only done once, but the resulting program can be run many times.
2. **Procedural.**
   That is, a program is divided into a number of "routines" (also called "functions" or "procedures"), each being a list of orders to be obeyed (though these orders are traditionally called "statements"). When the program is run, only one thing happens at a time: at any given moment, a single routine is being obeyed.
3. **Object-oriented.**
   That is, the fabric of a typical Inform program will be woven around "objects" being dealt with, which are regarded as being self-contained. For example, a program to simulate a warehouse might have objects representing lorries and containers; each object would have a position and contents at any given time. The program would have general rules describing "lorry" and "container" as well as actual examples of each. A lorry would have the ability to receive a message telling it to do something, such as "load up with a container and leave the warehouse".
4. **Portable.**
   That is, once Inform has compiled the source code (having found no mistakes), the resulting program can be run on almost any model of computer. It will exhibit exactly the same behaviour on each of them. It cannot depend on the "environment": it cannot suddenly run out of memory and crash, for instance.

The computer runs an Inform program (which need not be a game) with the aid of an "interpreter". There are at least 40 different interpreters available for this format (called the "Z-machine" or "Infocom format") and there may be a choice available for your model of computer: it is a good idea to get the most modern and accurate possible. Look to see if they support the Z-Machine Standard, and if so, up to what revision number.

§1.2 Example 1: Hello World

Traditionally, all programming language tutorials begin by giving a program which does nothing but print "Hello world" and stop. Here is such a program in Inform:

```
!  "Hello world" example program
[ Main;
  print "Hello world^";
];
```

The text after the exclamation mark is a "comment", that is, it is text written in the margin by the author to remind himself of what is going on here. Such text means nothing to Inform, which ignores anything on the same line and to the right of an exclamation mark.

Once commentary has been stripped out, Inform regards the source code as a list of things to look at, divided by semicolons `;`. It treats line breaks, tab characters and spaces all as so-called "white space": that is, a gap between two things whose size is unimportant. Thus, exactly the same program would be produced by the source code

```
   [
      Main   ;
print
         "Hello world^"          ;
    ]
  ;
```

or, at the other extreme, by

```
[ Main;print"Hello world^";];
```

Laying out programs legibly is a matter of forming good habits.

△ The exception to the rule about ignoring white space is inside quoted text, where

`"Hello    world^"` and `"Hello world^"`

are genuinely different pieces of text and are treated as such. Inform treats text inside quotation marks with much more care than its ordinary program material: for instance, an exclamation mark inside quotation marks will not cause the rest of its line to be thrown away as a comment.

Every program must contain a routine called `Main`, and in this example it is the only routine. When a program is set running, the first instruction obeyed is the first one in `Main`, and it carries on line by line from there. This process is called "execution". When the `Main` routine is finished, the program stops.

The routine has only one statement:

```
print "Hello world^"
```

Printing is the process of writing text onto the computer screen. This statement prints the two words "Hello world" and then skips the rest of the line (or "prints a new-line"): the ^ character, in quoted text, means "new-line". For example, the statement

```
print "Blue^Red^Green^"
```

prints up:

Blue
Red
Green

`print` is one of 28 statements in the Inform language. The full list is as follows:

```
box        break      continue   do       font      for         give
if         inversion  jump       move     new_line  objectloop  print
print_ret  quit       read       remove   restore   return      rfalse
rtrue      save       spaces     string   style     switch      while
```

(Only about 20 of these are commonly used.) §1 covers all those not concerned with objects, which are left until §3.

§1.3 Example 2: Elsinore

The following source code has three routines, `Main`, `Rosencrantz` and `Hamlet`:

```
[ Main;
  print "Hello from Elsinore.^";
  Rosencrantz();
];
[ Rosencrantz;
  print "Greetings from Rosencrantz.^";
];
[ Hamlet;
  print "The rest is silence.^";
];
```

The resulting program prints up

Hello from Elsinore.

Greetings from Rosencrantz.

but the text "The rest is silence." is never printed. Execution begins at `Main`, and "Hello from Elsinore" is printed; next, the statement `Rosencrantz()` causes the `Rosencrantz` routine to be executed. That continues until it ends with the close-routine marker `]`, whereupon execution goes back to `Main` just after the point where it left off: since there is nothing more to do in `Main`, the program finishes. Thus, `Rosencrantz` is executed but `Hamlet` is not.

In fact, when the above program is compiled, Inform notices that `Hamlet` is never needed and prints out a warning to that effect. The exact text produced by Inform varies from machine to machine, but will be something like this:

```
RISC OS Inform 6.03 (May 11th 1996)
line 8: Warning: Routine "Hamlet" declared but not used
Compiled with 0 errors and 1 warning
```

Errors are mistakes in the program which cause Inform to refuse to compile it, but this is only a warning. It alerts the programmer that a mistake may have been made (because presumably the programmer has simply forgotten to put in a statement calling `Hamlet`) but it doesn't prevent the compilation from taking place. Note that the opening line of the routine `Hamlet` occurs on the 8th line of the program above.

Usually there are mistakes in a newly-written program and one goes through a cycle of running a first draft through Inform, receiving a batch of error messages, correcting the draft according to these messages, and trying again. A typical error message would occur if, on line 3, we had mistyped `Rosncrantz()` for `Rosencrantz()`. Inform would then have produced:

```
RISC OS Inform 6.03 (May 11th 1996)
line 5: Warning: Routine "Rosencrantz" declared but not used
line 8: Warning: Routine "Hamlet" declared but not used
line 3: Error: No such constant as "Rosncrantz"
Compiled with 1 error and 2 warnings (no output)
```

The error message means that on line 3 Inform ran into a name which did not correspond to any known quantity (it isn't the name of any routine, in particular). Note that Inform never produces the final story file if errors occur during compilation: this prevents it from producing damaged story files. Note also that Inform now thinks the routine `Rosencrantz` is never used, since it didn't recognise the mistype in the way that a human reader would have done. Warnings are sometimes produced by accident this way, so it is generally a good idea to worry about fixing errors first and warnings afterward.

§1.4 Numbers and variables

Internally – that is, whatever the outward appearance – all programs essentially manipulate numbers. Inform understands "number" to be a whole number in the range -32768 to

+32767. (Special programming would be required to represent larger numbers or fractions.) There are three notations for writing numbers in Inform: here is an example of each.

```
-4205
$3f08
$$1000111010110
```

The difference is the radix, or base, in which they are expressed. The first is in decimal (base 10), the second hexadecimal (base 16, where the digits after 9 are written `a` to `f` or `A` to `F`) and the third binary (base 2). Once Inform has read in a number, it forgets which notation was used: for instance, if the source code is altered so that `$$10110` is replaced by 22, this makes no difference to the program produced.

A `print` statement can print numbers as well as text, though it always prints them back in ordinary decimal notation. For example, the program

```
[ Main;
  print "Today's number is ", $3f08, ".^";
];
```

prints up

Today's number is 16136.

since 16136 in base 10 is the same number as 3f08 in hexadecimal.

Inform recognises many other notations as "constants", that is, values which are literally described in the source code. A full list will appear later, but one other is that a single character between single quotation marks, for instance

```
'x'
```

is a constant. A "character" is a single letter or typewriter-symbol, and all that the programmer needs to know is that each possible character has its own numerical value.

△ For most characters, this numerical value is the standard ASCII value for the character: for instance, `'x'` has numerical value 120. (This is true even if Inform is being run on a model of computer which doesn't normally use the ASCII character set.) Exotic characters such as `'@ss'` (the Inform notation for German sz) have non-standard codes: see the *Z-Machine Standards Document* if you really need to know.

Finally, in this initial batch of constant notations, Inform provides two special constants:

```
true
false
```

which are used to describe the truth or otherwise of possible conditions.

△ `true` has the numerical value 1; `false` has the numerical value 0.

Inform has a concept of "variable" like that used in algebra, where it is easy but limiting to express facts using only numbers:

$$34 - 34 = 0$$
$$11 - 11 = 0$$
$$694 - 694 = 0$$

Although suggestive this fails to express the general case: that any number subtracted from itself leaves zero. We express this fact symbolically in algebra by writing

$$x - x = 0$$

where x is a variable; the implication being "whatever value x actually is, the statement is still true".

Likewise, in Inform what seems to be a word of text may be a variable which represents a number: when the source code is compiled, Inform cannot know what numerical value this text represents. When the program is run, it will always have a numerical value at any given time. If `oil_left` is a variable, the statement

```
print "There are ", oil_left, " gallons remaining.^";
```

is executed as if `oil_left` were replaced by whatever that value currently is. Later on, the same statement may be executed again, producing different text because by that time `oil_left` has a different value.

Inform can only know that text (such as `oil_left`) represents a variable if the source code has "declared" that it does. Each routine can declare its own selection of variables on its opening line. For example, in the program

```
[ Main alpha b;
  alpha = 2200;
  b = 201;
  print "Alpha is ", alpha, " while b is ", b, "^";
];
```

the `Main` routine has two variables, `alpha` and `b`. Like most names given in source code (called "identifiers"), variable names can be at most 32 characters long and may contain letters of the alphabet, decimal digits or the underscore `_` character (often used to imitate a space). To prevent them looking too much like numbers, though, they may not start with a decimal digit. (So `a44` is legal but `44a` is not.) For example:

```
turns_still_to_play
chart45
X
```

are all possible variable names. Inform ignores any difference between upper and lower case letters in such names, for example considering `CHArt45` as the same name as `chArT45`.

The `=` sign occurring twice in the above routine is an example of an "operator": a notation usually made up of the symbols on the non-alphabetic keys on a typewriter and which means something is to be done with the items it is written next to. In this context, `=` means "set equal to". When the statement `alpha = 2200` is executed at run time, the current value of the variable `alpha` becomes 2200 (and it keeps that value until another such statement changes it).

The variables `alpha` and `b` are called "local variables" because they are local to `Main`: in effect, they are its private property. The program

```
[ Main alpha;
  alpha = 2200;
  Rival();
];
[ Rival;
  print alpha;
];
```

causes an error on the `print` statement in `Rival`, since `alpha` does not exist there. Indeed, `Rival` could even have defined a variable of its own also called `alpha` and this would have been a separate variable with a probably different value.

§1.5 Arithmetic expressions

The Inform language is rich with operators, making it concise but not always very readable. Feeling comfortable with the operators is the main step towards being able to follow Inform source code. Fortunately, these operators are based on the usual rules for writing arithmetic formulae, which gives them a headstart in familiarity.

Indeed, the most commonly used operators are "arithmetic": they combine one or more numbers to give one resulting number. Whenever a number is expected in a statement, a general "expression" can be given instead: that is, a calculation giving a number as a result. For example, the statement

```
seconds = 60*minutes + 3600*hours
```

sets the variable `seconds` equal to 60 times the variable `minutes` plus 3600 times the variable `hours`. White space is not needed between operators and "operands" (the numbers to be operated on): the spaces on either side of the `+` sign are only provided for legibility.

Ordinary arithmetic is carried out with the operators `+` (plus), `-` (minus), `*` (times) and `/` (divided by).

Usually dividing one integer by another leaves a remainder: for example, 3 goes into 7 twice, with remainder 1. In Inform notation,

`7/3` evaluates to 2
`7%3` evaluates to 1

the `%` operator meaning "remainder after division", usually called just "remainder". Dividing by zero is impossible and a program which tries to do this will go wrong.

△ As a brief aside, this gives an example of how Inform can and can't help the programmer to spot mistakes. The program

```
[ Main;
  print 73/0;
];
```

produces an error when compiled:

```
line 2: Error: Division of constant by zero
>   print 73/0;
```

since Inform can see that it definitely involves doing something illegal. However, Inform fails to notice anything amiss with the equivalent program

```
[ Main x;
  x = 0;
  print 73/x;
];
```

and this program compiles correctly. The resulting story file will "crash" when it is run, that is, catastrophically halt. The moral is that just because Inform compiles a program without errors, it does not follow that the program does what the programmer intends.

In a complicated expression the order in which the operators work may affect the result. As most human readers would, Inform works out both of

```
23 + 2 * 700
2 * 700 + 23
```

to 1423, because the operator `*` has "precedence" over `+` and so is acted on first. Brackets may be used to overcome this:

```
(23 + 2) * 700
2 * (700 + 23)
```

evaluate to 17500 and 1446 respectively. Each operator has such a "precedence level". When two operators have the same precedence level (for example, `+` and `-` are of equal precedence) calculation is (almost always) "left associative", that is, carried out left to right: the notation

```
a - b - c
```

is equivalent to

```
(a - b) - c
```

The standard rules for writing mathematics give `+` and `-` equal precedence, lower than that of `*` and `/` (which are also equal). Inform agrees and also pegs `%` equal to `*` and `/`.

The final purely arithmetic operator is "unary minus". This is written as a minus sign - but is not the same as ordinary subtraction. The expression:

```
-credit
```

means the same thing as:

```
0 - credit
```

The operator - is different from all those mentioned so far because it operates only on one number. It has higher precedence than any of the five "binary" operations above. For example,

```
-credit - 5
```

means `(-credit) - 5` and not `-(credit - 5)`.

One way to imagine precedence is to think of it as glue attached to the operator. A higher level means stronger glue. Thus, in

```
23 + 2 * 700
```

the glue around the `*` is stronger than that around the `+`, so that 2 and 700 belong bound to the `*`.

Some operators do not simply act on values but actually change the current values of variables: expressions containing these are called "assignments" (because they assign values as well as working them out). One such operator is 'set equals':

```
alpha = 72
```

sets the variable `alpha` equal to 72. Just like `+` and the others, it also comes up with an answer: as it happens, this value is also 72.

The other two assignment operators are `++` and `--`, which will be familiar to any C programmer. They are unary operators, and can be used in any of the following ways:

```
variable++
++variable
variable--
--variable
```

The first of these means "read off the value of `variable`, and afterwards increase that value by one". In `++variable` the "increment" (or increase by 1) happens first, and then the value is read off. `--` acts in a similar way but "decrements" (decreases by 1). These operators are provided as convenient shorthand forms, since their effect could usually be achieved in other ways (just using `+` and `-`).

For example, suppose the `variable` has value 12. Then the result would be 12, 13, 12 or 11 respectively; the value left in `variable` afterwards would be 13, 13, 11 or 11.

Note that expressions like

```
500++        (4*alpha)--     34 = beta
```

are meaningless: the values of 500 and 34 cannot be altered, and Inform knows no way to adjust `alpha` so as to make `4*alpha` decrease by 1. All three will cause errors.

△ "Bitwise operators" are provided for manipulating binary numbers on a digit-by-digit basis, something which is often done in programs which are working with low-level data or data which has to be stored very compactly. Inform provides `&`, bitwise AND, `|`, bitwise OR and `~`, bitwise NOT. For each digit, such an operator works out the value in the answer from the values in the operands. Bitwise NOT acts on a single operand and results in the number whose i-th binary digit is the opposite of that in the operand (a 1 for a 0, a 0 for a 1). Bitwise AND (and OR) acts on two numbers and sets the i-th digit to 1 if both operands have (either operand has) i-th digit set. So, for example,

```
$$10111100 & $$01010001  ==  $$00010000
```

△ The remaining operators will be described as needed: the full table is laid out in §A1.

§1.6 Arguments and Return Values

As has already been said, in Inform jargon the word "function" is synonymous with "routine". A function might be defined as a correspondence

$$(x_1, ..., x_n) \mapsto f(x_1, ..., x_n)$$

where a set of input numbers are fed in, and a single value comes out. These input numbers are called "arguments". The value coming out is the "return value", or is said to be "returned".

All Inform routines are like this. A number of arguments are fed in when the routine is "called" (that is, set running) and there is always a single numerical result. This result is called the "return value" because it is returned to the rest of the program. Some very simple routines conceal this. For instance, consider `Sonnet`:

```
[ Main;
  Sonnet();
];
[ Sonnet;
  print "When to the sessions of sweet silent thought^";
  print "I summon up remembrance of things past^";
];
```

`Sonnet` is a routine which takes as input no arguments at all (it is an example of the $n = 0$ case), so it is called with nothing in between the round brackets. Although it does return a value (as it happens, this value is `true`) the statement `Sonnet()` simply calls the routine and throws the return value away. If `Main` were instead given by

```
[ Main;
  print Sonnet();
];
```

then the output would be

When to the sessions of sweet silent thought

I summon up remembrance of things past
1

because the `print` statement in `Main` has been told to print the number resulting from a call to `Sonnet`.

Thus in Inform there is no such thing as a "void function" or "procedure": every routine returns a number even though this may immediately be thrown away as unwanted.

When a routine is called,

```
Routine(arg1, ...)
```

the arguments given are substituted into the first variables declared for `Routine`, and execution begins running through `Routine`. Usually, there can be any number of arguments from none up to 7, though a limit of 3 applies if Inform has been told to compile an early-model story file (see §31 for details).

If execution runs into the `]` end-of-routine marker, so that the routine is finished without having specified any definite return value, then this value is `true`. (This is why the printed return value of `Sonnet` is 1: `true` has the value 1.)

§1.7 Example 3: Cubes

A more typical, though less aesthetic, example than `Sonnet`:

```
[ Main;
  print Cube(1), " ";
  print Cube(2), " ";
  print Cube(3), " ";
  print Cube(4), " ";
  print Cube(5), "^";
];
[ Cube x;
  return x*x*x;
];
```

which, when executed, prints

1 8 27 64 125

The expression `Cube(3)` is calculated by substituting the number 3 into the variable `x` when `Cube` is set running: the result of the expression is the number returned by `Cube`.

Any "missing arguments" in a routine call are set equal to zero, so the call `Cube()` is legal and does the same as `Cube(0)`.

§1.8 Conditions: `if`, `true` and `false`

Such routines are too simple, so far, even to express many mathematical functions, and more flexibility will be needed.

A "control construct" is a kind of statement which controls whether or not, and if so how many times or in what order, other statements are executed. The simplest of these is `if`:

`if` (⟨condition⟩) ⟨statement⟩

which executes the ⟨statement⟩ only if the ⟨condition⟩, when it is tested, turns out to be true. For example, when the statement

```
if (alpha == 3) print "Hello";
```

is executed, the word "Hello" is printed only if the variable `alpha` currently has value 3. It is important not to confuse the `==` operator (test whether or not equal to) with the `=` operator (set equal to).

Conditions are always given in (round) brackets. The basic conditions are as follows:

`(a == b)`	Number `a` equals number `b`
`(a ~= b)`	Number `a` doesn't equal number `b`
`(a >= b)`	`a` is greater than or equal to `b`
`(a <= b)`	`a` is less than or equal to `b`
`(a > b)`	`a` is greater than `b`
`(a < b)`	`a` is less than `b`
`(o1 in o2)`	Object `o1` possessed by `o2`
`(o1 notin o2)`	Object `o1` not possessed by `o2`
`(o1 has a)`	Object `o1` has attribute `a`
`(o1 hasnt a)`	Object `o1` hasn't attribute `a`
`(o1 provides m)`	Object `o1` provides property `m`
`(o1 ofclass c)`	Object `o1` inherits from class `c`

(The conditions relating to objects will be discussed later.) A useful extension to this set is provided by the special operator `or`, which gives alternative possibilities. For example,

```
if (alpha == 3 or 4) print "Scott";
if (alpha ~= 5 or 7 or 9) print "Amundsen";
```

where two or more values are given with the word `or` between. `Scott` is printed if `alpha` has value either 3 or 4, and `Amundsen` if the value of `alpha` is not 5, is not 7 and is not 9. `or` can be used with any of the conditions, and any number of alternatives can be given. For example

```
if (player in Forest or Village or Building) ...
```

often makes code much clearer than writing three separate conditions out; or

```
if (x > 100 or y) ...
```

can be convenient to check whether `x` is bigger than the minimum of 100 or `y`.

Conditions can also be built up from simpler ones (just as long expressions are built up from single operators) using the three logical operators `&&`, `||` and `~~` (pronounced "and", "or" and "not"). For example,

```
if (alpha == 1 && (beta > 10 || beta < -10)) print "Lewis";
if (~~(alpha > 6)) print "Clark";
```

"Lewis" is printed if alpha equals 1 and beta is outside the range -10 to 10; "Clark" is printed if alpha is less than or equal to 6.

The discussion above makes it look as if conditions are special kinds of expression which can only use certain operators (==, &&, or and so on). But this is not true: conditions are expressions like any other. It's legal to write

```
print (beta == 4);
```

for instance, and this results in 1 being printed if beta equals 4, and 0 otherwise. Thus:

the result of a true condition is 1;
the result of a false condition is 0.

This is why true and false are defined to be 1 and 0 respectively. Thus one might write code along the lines of

```
betaisfour = (beta == 4);
...
if (betaisfour == true) ...
```

though it would be easier to write

```
betaisfour = (beta == 4);
...
if (betaisfour) ...
```

because, just as conditions can be used as numbers, so numbers can be used as conditions. Zero is considered to be "false", and all other values are considered to be "true". Thus

```
if (1) print "Magellan";
if (0) print "da Gama";
```

always results in "Magellan", never "da Gama", being printed.

One common use of variables is as "flags". A flag can only hold the value 0 or 1, false or true according to some state of the program. The fact that a number can be used as a condition allows natural-looking statements like

```
if (lower_caves_explored) print "You've already been that way.";
```

where lower_caves_explored is a variable being used in the program as a flag.

△ Note that && and || only work out what they absolutely need to in order to decide the truth. That is,

```
if (A && B) ...
```

will work out A first. If this is false, there's no need to work out B, and it never is worked out. Only if A is true is B actually tested. This only matters when working out conditions like

```
if (x==7 && Routine(5)) ...
```

where it can be important to know that the Routine is never called if x has a value other than 7.

§1.9 Example 4: Factorials

The factorial of a positive integer n is defined as the product

$$1 \times 2 \times 3 \times ... \times n$$

so that, for example, the factorial of 4 is 24. Here is an Inform routine to calculate factorials:

```
[ Main;
  print Factorial(7), "^";
];
[ Factorial n;
  if (n==1) return 1;
  return n*Factorial(n-1);
];
```

This calculates 7 factorial and comes up with 5040. (Factorials grow rapidly and 8 factorial is already too large to hold in a standard Inform number, so calling `Factorial(8)` would give a wrong answer.)

The routine `Factorial` actually calls itself: this is called "recursion". Execution reaches "seven routines deep" before starting to return back up. Each of these copies of `Factorial` runs with its own private copy of the variable `n`.

Recursion is hazardous. If one calls the routine

```
[ Disaster;
  return Disaster();
];
```

then despite the reassuring presence of the word `return`, execution is tied up forever, unable to finish evaluating the return value. The first call to `Disaster` needs to make a second before it can finish; the second needs to make a third; and so on. This is an example of a programming error which will prove disastrous when the program is run, yet will cause no errors when the source code is compiled. (It can be proved that it is impossible to construct a compiler capable of detecting this general class of mistake. Inform does not even try.)

§1.10 Code blocks, `else` and `switch`

A feature of all control constructs is that instead of just giving a ⟨statement⟩, one can give a list of statements grouped together into a unit called a "code block". Such a group begins with an open brace `{` and ends with a close brace `}`. For example,

```
if (alpha > 5)
{   print "The square of alpha is ";
    print alpha*alpha;
    print ".^";
}
```

If `alpha` is 3, nothing is printed; if `alpha` is 9,

The square of alpha is 81.

is printed. (As usual the layout is a matter of convention: it is usual to write code blocks on margins indented inwards by some standard number of characters.) In some ways, code blocks are like routines, and at first it may seem inconsistent to write routines between `[` and `]` brackets and code blocks between braces `{` and `}`. However, code blocks cannot have private variables of their own and do not return values: and it is possible for execution to break out of code blocks again, or to jump from block to block, which is impossible with routines.

An `if` statement can optionally have the form

`if (`⟨condition⟩`)` ⟨statement1⟩ `else` ⟨statement2⟩

in which case ⟨statement1⟩ is executed if the condition is true, and ⟨statement2⟩ if it is false. For example,

```
if (alpha == 5) print "Five."; else print "Not five.";
```

Note that the condition is only checked once. The statement

```
if (alpha == 5)
{   print "Five.";
    alpha = 10;
}
else print "Not five.";
```

cannot ever print both "Five" and then "Not five".

The `else` clause has a snag attached: the problem of "hanging elses".

```
if (alpha == 1)
    if (beta == 2)
        print "Clearly if alpha=1 and beta=2.^";
    else
        print "Ambiguous.^";
```

is ambiguous as to which `if` statement the `else` attaches to. The answer (in Inform 6, though this has changed since earlier versions of the language) is that an `else` always pairs to its nearest `if`, unless there is bracing to indicate the contrary. Thus the `else` above pairs with the `beta` condition, not the `alpha` condition.

In any case it is much safer to use braces to express what is meant, as in:

```
if (alpha == 1)
{   if (beta == 2)
        print "Clearly if alpha=1 and beta=2.^";
    else
        print "Clearly if alpha=1 but beta not 2.^";
}
```

The `if...else...` construct is ideal for switching execution between two possible "tracks", like railway signals, but it is a nuisance trying to divide between many different outcomes this way. To follow the analogy, the construct `switch` is like a railway turntable.

```
print "The train on platform 1 is going to ";
switch(DestinationOnPlatform(1))
{   1: print "Dover Priory.";
    2: print "Bristol Parkway.";
    3: print "Edinburgh Waverley.";
}
```

Each possible value must be a constant, so

```
switch(alpha)
{   beta: print "The variables alpha and beta are equal!";
}
```

is illegal.

Any number of outcomes can be specified, and values can be grouped together to a common outcome. For example,

```
print "The mission STS-", num, " was flown on the Space Shuttle";
switch(num)
{   1 to 5, 9: print " Columbia.";
    6 to 8:    print " Challenger.";
    10 to 25:  if (num == 12) print " Discovery";
               print ", but it was given a flight number like 51-B.";
    default:   print ".";
}
```

will result in a true statement being printed (as long as `num` is between 1 and, at time of writing, 78), if an incomplete one. The `default` clause is executed if the original expression matches none of the other values, and it must always come last if given at all. In this case, it means that if `num` is 62, then

The mission STS-62 was flown on the Space Shuttle.

is printed.

Note that each clause is automatically a code block and needs no braces `{` to `}` to delimit it from the rest of the routine: this shorthand makes `switch` statements much more legible.

§1.11 `while`, `do...until`, `for`, `break`, `continue`

The other four Inform control constructs are all "loops", that is, ways to repeat the execution of a given statement (or code block). Discussion of one of the four, called `objectloop`, is deferred until §3.4.

The two basic forms of loop are `while` and `do`...`until`:

`while (`⟨condition⟩`)` ⟨statement⟩
`do` ⟨statement⟩ `until (`⟨condition⟩`)`

The first repeatedly tests the condition and, provided it is still true, executes the statement. (If the condition is not even true the first time, the statement is never executed.) For example:

```
[ SquareRoot n;
  x = n;
  while (x*x > n) x=x-1;
  return x;
];
```

a (fairly chronic) method for finding square roots. (If `SquareRoot(200)` is called, then `x` runs down through the values 200, 199, ..., 14, at which point `x*x <= n` since $14 \times 14 = 196$.)

The `do`...`until` loop repeats the given statement until the condition is found to be true. (Even if the condition is already satisfied, like `(true)`, the statement is always executed the first time through.)

One particular kind of `while` loop is needed so often that there is an abbreviation for it, called `for`. For example,

```
counter = 1;
while (counter <= 10)
{   print counter, " ";
    counter++;
}
```

which produces the output

1 2 3 4 5 6 7 8 9 10

(Recall that `counter++` adds 1 to the variable `counter`.) Languages like BASIC make extensive use of this kind of loop. For example, in BBC BASIC, the above loop would be written

```
FOR counter = 1 TO 10
    PRINT counter;" ";
NEXT
```

`NEXT` is a word which (slightly clumsily) means "the code block ends here", and is therefore the equivalent of Inform's `}`. The whole is used to mean "for values of the counter running through 1 to 10, do...", hence the choice of the word `FOR`.

Inform (like the language C) uses a more flexible construct than this, but which is still called `for`. It can produce any loop in the form

⟨start⟩

```
while (⟨condition⟩)
{   ...
    ⟨update⟩
}
```

where ⟨start⟩ and ⟨update⟩ are assignments. The notation to achieve this is

```
for (⟨start⟩ : ⟨condition⟩ : ⟨update⟩) ...
```

For example, the loop described above is achieved by

```
for (counter=1 : counter<=10 : counter++)
    print counter, " ";
```

Note that if the condition is false even the first time, the loop is never executed. For instance,

```
for (counter=1 : counter<0 : counter++)
    print "Banana";
```

prints nothing.

△ At this point it is worth mentioning that several assignments can be combined into a single statement in Inform. For example,

```
i++, score=50, j++
```

(three assignments separated by commas) is a single statement. This is never useful in ordinary code, where the assignments can be divided up by semicolons in the usual way. In `for` loops it is useful, though:

```
for (i=1, j=5: i<=5: i++, j--) print i, " ", j, ", ";
```

produces the output "1 5, 2 4, 3 3, 4 2, 5 1,".

Any of the three parts of a `for` statement can be omitted. If the condition is missed out, it is assumed to be always true, i.e. there is no check made to see if the loop should be ended and so the loop continues forever.

On the face of it, the following loops all repeat forever:

```
while (true) ⟨statement⟩
do ⟨statement⟩ until (false)
for (::) ⟨statement⟩
```

But there is always an escape. One way is to `return` from the current routine. Another is to `jump` to a label outside the loop (`jump` will be covered in §1.13 below). It's neater to use the statement `break`, which causes execution to "break out of" the current innermost loop or `switch` statement: it can be read as "finish early". All these ways out are entirely "safe", and there is no harm in leaving a loop only half-done.

The other simple statement used inside loops is `continue`. This causes the current iteration to end immediately, but does not end the whole loop. For example,

```
for (i=1: i<=5: i++)
{   if (i==3) continue;
    print i, " ";
}
```

will output "1 2 4 5".

§1.12 Example 5: A number puzzle

The routine `RunPuzzle` is an interesting example of a loop which, though apparently simple enough, contains a trap for the unwary.

```
[ RunPuzzle n count;
  do
  {   print n, " ";
      n = NextNumber(n);
      count++;
  }
  until (n==1);
  print "1^(taking ", count, " steps to reach 1)^";
];
[ NextNumber n;
  if (n%2 == 0) return n/2;     ! If n is even, halve it
  return 3*n + 1;               ! If n is odd, triple and add 1
];
```

The call `RunPuzzle(10)`, for example, results in the output

> 10 5 16 8 4 2 1
> (taking 6 steps to reach 1)

The source code assumes that, no matter what the initial value of `n`, enough iteration will end up back at 1. If this did not happen, the program would lock up into an infinite loop, printing numbers forever.

The routine is apparently very simple, so it would seem reasonable that by thinking carefully enough about it, we ought to be able to decide whether or not it is "safe" to use (i.e., whether it can be guaranteed to finish or not).

And yet nobody knows whether this routine is "safe". The conjecture that all `n` eventually step down to 1 is at least fifty years old but has never been proved, having resisted all mathematical attack. (Alarmingly, `RunPuzzle(27)` takes 111 iterations to fall back down to 1.)

§1.13 `quit`, `jump` and the program state

There are four statements left which control the flow of execution. `quit` ends the program immediately (as if a return had taken place from the `Main` routine). This drastic measure is best reserved for points in the program which have detected some error condition so awful that there is no point carrying on. Better yet, do not use it at all.

The `jump` statement transfers execution to some other named place in the same routine. (Some programming languages call this `goto`. Since it can be and has been put to ugly uses, the construct itself was at one time frowned on as a vulgar construct leading programmers into sin. Good use of control constructs will almost always avoid the need for `jump` and result in more legible programs. But sin is universal.)

To use `jump` a notation is needed to mark particular places in the source code. Such markers are called "labels". For example:

```
[ Main i;
  i=1;
  .Marker;
  print "I have now printed this ", i++, " times.^";
  jump Marker;
];
```

This program has one label, `Marker`. A statement consisting only of a full stop and then an identifier means "put a label here and call it this".

△△ An Inform program has the ability to save a snapshot of its entire state and to restore back to that previous state. This snapshot includes values of variables, the point where code is currently being executed, and so on. Just as we cannot know if the universe is only six thousand years old, as creationists claim, having been endowed by God with a carefully faked fossil record; so an Inform program cannot know if it has been executing all along or if it was only recently restarted. The statements required are `save` and `restore`:

> `save` ⟨label⟩
> `restore` ⟨label⟩

This is a rare example of an Inform feature which may depend on the host machine's state of health: for example, if all disc storage is full, then `save` will fail. It should always be assumed that these statements may well fail. A `jump` to the label provided occurs if the operation has been a success. (This is irrelevant in the case of a `restore` since, if all has gone well, execution is now resuming from the successful branch of the `save` statement: because that is where execution was when the state was saved.)

§1.14 Printing output

When text is printed, normally each character is printed exactly as specified in the source code. Four characters, however, have special meanings. As explained above `^` means "print a new-line". The character `~`, meaning "print a quotation mark", is needed since quotation marks otherwise finish strings. Thus,

```
"~Look,~ says Peter. ~Socks can jump.~^Jane agrees."
```

is printed as

> "Look," says Peter. "Socks can jump."
> Jane agrees.

The third remaining special character is `@`, which is used for accented characters and other unusual effects, as described below. Finally, `\` is reserved for "folding lines", and used to be needed in Inform 5 when text spilled over more than one line. (It's no longer needed but kept so that old programs still work.) If you really want to print a `~`, a `^`, an `@` or a `\`, see below.

Text still spills over more than one line, even in the present golden age of Inform 6. When a statement like

```
print "Here in her hairs
       the painter plays the spider, and hath woven
       a golden mesh t'untrap the hearts of men
       faster than gnats in cobwebs";
```

is read in by Inform, the line breaks are replaced with a single space each. Thus the text printed is: "Here in her hairs the painter plays the spider, and hath woven a golden mesh..." and so on. (There is one exception: if a line finishes with a `^` (new-line) character, then no space is added before the next line begins.)

So far, only the `print` statement has been used for printing, to print both numbers and strings (that is, double-quoted pieces of text). Since Inform is primarily a language for writing Adventure games, its business is text, and it provides many other facilities for printing.

`new_line`

is a statement which simply prints a new-line (otherwise known as a carriage return, as if the lever on the carriage of an old manual typewriter had been pulled to move it right back to the left margin and turn it forward one line). This is equivalent to

`print "^"`

but is a convenient abbreviation. Similarly,

`spaces` ⟨number⟩

prints a sequence of that many spaces.

`inversion`

prints the version number of Inform which was used to compile the program (it might, for instance, print "6.01").

`box` ⟨string1⟩ ... ⟨stringn⟩

displays a reverse-video box in the centre of the screen, containing

string1
string2
...
stringn

and is usually used for popping up quotations: for example,

```
box "Passio domini nostri" "Jesu Christi Secundum" "Joannem"
```

displays

Passio domini nostri
Jesu Christi Secundum
Joannem

(the opening line of the libretto to Arvo Pärt's 'St John Passion').

Text is normally displayed in ordinary (or "Roman") type. Its actual appearance will vary from machine to machine running the program. On many machines, it will be displayed using a "font" which is variably-pitched, so that for example a "w" will be wider on-screen than an "i". Such text is much easier to read, but makes it very difficult to print out diagrams. The statement

```
print "+------------+
      ^+   Hello    +
      ^+------------+^";
```

will print something quite irregular if the characters "-", "+" and " " (space) do not all have the same width. Because one sometimes does want to print such a diagram (to represent a sketch-map, say, or to print out a table), the statement `font` is provided:

```
font on
font off
```

`font off` switches into a fixed-pitch display style (in which all characters definitely have the same width); `font on` goes back to the original.

In addition to this, a few textual effects can be achieved.

```
style roman
```

switches to ordinary Roman text (the default), and there are also

```
style bold
style underline
style reverse
```

(`reverse` meaning "reverse colour": e.g. yellow on blue if the normal text appearance is blue on yellow). An attempt will be made to approximate these effects on any machine, but it may be that `underline` comes out as italicised text, for example, or that `bold` is rendered by printing ordinary Roman text but in a different colour.

Inform programs are starting to be written which communicate in languages other than English: Italian, Dutch, German, French and Spanish games have all been attempted. A comprehensive range of accented characters is available: these are reached with the aid of the escape character, `@`.

Most accented characters are written as `@`, followed by an accent marker, then the letter on which the accent appears:

`@^`	put a circumflex on the next letter: a,e,i,o,u,A,E,I,O or U
`@'`	put an acute on the next letter: a,e,i,o,u,y,A,E,I,O,U or Y
`` @` ``	put a grave on the next letter: a,e,i,o,u,A,E,I,O or U
`@:`	put a diaeresis on the next letter: a,e,i,o,u,A,E,I,O or U
`@c`	put a cedilla on the next letter: c or C
`@~`	put a tilde on the next letter: a,n,o,A,N or O
`@\`	put a slash on the next letter: o or O
`@o`	put a ring on the next letter: a or A

In addition, there are a few others:

`@ss`	German sz
`@<<`	continental European quotation marks
`@>>`	
`@ae`	ligatures
`@AE`	
`@oe`	
`@OE`	
`@th`	Icelandic accents
`@et`	
`@Th`	
`@Et`	
`@LL`	pound sign
`@!!`	Spanish (upside-down) exclamation mark
`@??`	Spanish (upside-down) question mark

For instance,

```
print "Les @oeuvres d'@Aesop en fran@ccais, mon @'el@`eve!";
print "Na@:ive readers of the New Yorker will re@:elect Mr Clinton.";
print "Carl Gau@ss first proved the Fundamental Theorem of Algebra.";
```

Accented characters can also be referred to as constants, like other characters. Just as `'x'` represents the character lower-case-X, so `'@^A'` represents capital-A-circumflex.

△ The `@` escape character has two other uses. One gets around the problem that, so far, it is impossible to print an "`@`". A double `@` sign, followed by a number, prints the character with this numerical code. The most useful cases are:

`@@92`	comes out as "`\`"
`@@64`	comes out as "`@`"
`@@94`	comes out as "`^`"
`@@126`	comes out as "`~`"

enabling us to print the four characters which can't be typed directly because they have other meanings.

△△ The second use is more obscure. Inform keeps a stock of 32 pseudo-variables to hold text, numbered from 0 to 31.

`@00`	prints out as the current contents of string 0
...	...
`@31`	prints out as the current contents of string 31

and these variables are set with the `string` statement:

```
string 0 "toadstool";
```

sets string 0 to the text of the word "toadstool". (There is a technical reason why these strings cannot be set equal to any text: only to literal text, as in the above example, or to strings previously declared using the `Low_string` directive.)

Finally, it is time to discuss `print`. There are two forms, `print` and `print_ret`. The only difference is that the second prints out an extra new-line character and returns from the current routine with the value `true`. Thus, `print_ret` should be read as "print and then return", and

```
print_ret "That's enough of that.";
```

is equivalent to

```
print "That's enough of that.^"; rtrue;
```

In fact, as an abbreviation, it can even be shortened to:

```
"That's enough of that.";
```

Although Inform newcomers are often confused by the fact that this apparently innocent statement actually causes a return from the current routine, it's an abbreviation which very much pays off in adventure-writing situations. Note that if the program:

```
[ Main;
  "Hello, and now for a number...";
  print 45*764;
];
```

is compiled, Inform will produce the warning message:

```
line 3: Warning: This statement can never be reached.
>   print 45*764;
```

because the bare string on line 2 is printed using `print_ret`: so the text is printed, then a new-line is printed, and then a `return` takes place immediately. As the warning message indicates, there is no way the statement on line 3 can ever be executed.

So what can be printed? The answer is a list of terms, separated by commas. For example,

```
print "The value is ", value, ".";
```

contains three terms. A term can take the following forms:

⟨a numerical quantity⟩	printed as a (signed, decimal) number
⟨text in double-quotes⟩	printed as text
(⟨rule⟩) ⟨quantity⟩	printed according to some special rule

Inform provides a stock of special printing rules built-in, and also allows the programmer to create new ones. The most important rules are:

`(char)`	print out the character which this is the numerical code for
`(string)`	print this string out
`(address)`	print out the text at this array address (this is seldom used, and then mainly to print the text of a word entry in a game's dictionary)

△ `print (string) ...` requires a little explanation.

```
x = "Hello!";
print (string) x;
```

prints out "Hello!", whereas

```
x = "Hello!";
print x;
```

prints a mysterious number. This is because strings are internally represented by numbers (just as everything else is).

The remaining stock of rules is provided for use in conjunction with the Library and is documented in Chapter V: briefly,

`(the)`	print definite article then name of this object
`(The)`	ditto, but capitalised
`(name)`	ditto, but with no article
`(a)`	ditto, but with the indefinite article
`(number)`	print this number out in English
`(property)`	(for debugging) print the name of this property
`(object)`	(ditto) print the hardware-name of this object

Note that `(the)` in lower case does something different from `(The)` with an upper case T. This is very unusual! (Directive names, which will turn up in §2, variable names and so on are allowed to use upper case and the case is simply ignored, so that `fRoG` means the same as `frog`. But statement keywords, like `print` or `(name)`, have to be in lower case – except for `(The)`.)

To create a new rule, provide a routine with this name, and use the rule-name in brackets.

§1.15 Example 6: Printing in hexadecimal

The following pair of routines provides for printing out a number as a four-digit, unsigned hexadecimal number. For example, so that

```
print (hex) 16339;
```

prints “3fd3”.

```
[ hex x y;
  y = (x & $ff00) / $100;
  x = x & $ff;
  print (hdigit) y/$10, (hdigit) y, (hdigit) x/$10, (hdigit) x;
];
[ hdigit x;
  x = x % $10;
  if (x<10) print x; else print (char) 'a'+x-10;
];
```

Once these routines have been defined, `hex` and `hdigit` are available anywhere in the same program for use as new printing rules.

§1.16 Built-in functions 1: `random` and `indirect`

Inform provides a small stock of functions ready-defined, but which are used much as other functions are. All but two of these concern objects and will be left until chapter 3.

`random` has two forms:

`random(N)`

returns a uniformly random number in the range $1, 2, ..., N$. N should always be a positive number (between 1 and 32767) for this to work properly.

`random(`two or more constant quantities, separated by commas`)`

returns a uniformly random choice from this selection. Thus,

```
print (string) random("red", "blue", "green", "purple", "orange");
```

randomly prints the name of one of these five colours (each being equally likely to appear). Likewise,

```
print random(13, 17);
```

has a 50% chance of printing 13, and a 50% chance of printing 17.

△△ The other built-in function discussed here is `indirect`.

```
indirect(function, arg1, arg2, ...)
```

calls the given function with given arguments. Thus, this is equivalent to

```
function(arg1, arg2, ...)
```

but has the additional virtue that the function can be given, not just as a literal function name, but as some calculated value:

```
indirect(random(OneRoutine, OtherRoutine), 45);
```

has a 50% chance of calling `OneRoutine(45)`, and a 50% chance of calling `OtherRoutine(45)`. `indirect` should be used with caution: if supplied with a numerical first argument which doesn't correspond to any function in the program, the program may resoundingly crash. In any event, it is often best to achieve such effects using messages to objects.

§1.17 Accepting input

△△ Inform programmers seldom need to take input from the keyboard, in practice, since in all game situations the Library's parser routines take care of all that. However, for completeness this section covers the `read` statement which is the main route by which keyboard input is taken. It will not make much sense to readers who have not yet read the rest of this book.

The syntax is

`read` ⟨text array⟩ ⟨parse buffer⟩ ⟨routine⟩

where the ⟨routine⟩ is optional: if provided, it is called just before the input takes place so that the screen's top line or lines of data (the "status line" present in many games) can be renewed.

What the statement does is to read in a single line of text (waiting until the user has finished typing a line and then pressed RETURN), copy this text into the text array and then try to comprehend it, writing the results of this comprehension exercise ("parsing") into the parse buffer.

Before the statement is reached, the program should have entered the maximum number of characters which can be accepted (say, 60) into the 0th entry of the text array; the statement will then write the actual number typed into the 1st entry, and the characters themselves into entries 2 and onward. Thus,

```
text_array -> 0 = 60;
read text_array 0;
for (n = 0: n< text_array->1: n++) print (char) text_array->(n+2);
new_line;
```

will read in a line of up to 60 characters, and then print it back again. (The array `text_array` must have been created first, and so must the local variable `n`, of course.)

Note that in this case, no "parse buffer" has been given (0 was given in its place). If, instead of 0, an array is given here, then the `read` statement makes an attempt to divide up the input text into individual words, and to match these words against the game's dictionary. See §2.5 for details.

2 The language of data structures

§2.1 Directives and constants

Every example program so far has consisted only of a sequence of routines, each within beginning and end markers `[` and `]`. Such routines have no way of communicating with each other, and therefore of sharing information with each other, except by making function calls back and forth. This arrangement is not really suited to a large program whose task may be to simulate something complicated (such as the world of an adventure game): it would be useful to have some kind of central registry of information which all routines have access to, as and when needed.

Information available to all routines in this way is said to be "global", rather than "local" to any one routine. (As will appear in §3, there is also an intermediate possibility where information is available only to a cluster of routines working on roughly the same part of a program.)

This global information can be organised in a variety of ways. Such organised groups are called "data structures". For example, a typical data structure might be a list of 10 values. The term "data structure" did not appear in §1 because information was only ever held in variables, the simplest possible kind of structure (one value on its own).

Data structures are added to Inform programs using commands called "directives" in between definitions of routines. It's important to distinguish between these, which direct Inform to do something now (usually, to create something) and the statements which occur inside routines, which are merely translated in some way but not acted on until the program has finished being compiled and is run.

In fact, one directive has already appeared: the one written `[`, which means "translate the following routine up to the next `]`". In all there are 38 Inform directives, as follows:

```
Abbreviate  Array       Attribute  Class        Constant     Default
Dictionary  End         Endif      Extend       Fake_action  Global
Ifdef       Ifndef      Ifnot      Ifv3         Ifv5         Iftrue
Iffalse     Import      Include    Link         Lowstring    Message
Nearby      Object      Property   Release      Replace      Serial
Switches    Statusline  Stub       System_file  Trace        Verb
Version     [
```

Several of these are rather technical and will not be used by many programmers (such as `Trace`, `Stub`, `Default`, `System_file`, `Abbreviate`, `Dictionary`). Others control fine points of what is compiled and what isn't (`Ifdef`, `Ifnot`, and so on; `Message`, `Replace`). These not-very important directives are covered in Chapter II.

This leaves 9 central directives for creating data structures, and these are the ones which it is important to know about:

```
Array       Attribute  Class      Constant     Extend       Global
Object      Property   Verb
```

It is conventional to write these with the initial letter capitalised: this makes directives look unlike statements. `Attribute`, `Class`, `Object` and `Property` are the subject of §3.

The simplest directive with a "global" effect on the program – an effect all over the program, that is, not just in one routine – is `Constant`. The following program, an unsatisfying game of chance, shows a typical use of `Constant`.

```
Constant MAXIMUM_SCORE = 100;
[ Main;
  print "You have scored ", random(MAXIMUM_SCORE),
  " points out of ", MAXIMUM_SCORE, ".^";
];
```

The maximum score value is used twice in the routine `Main`. Of course the program is the same as it would have been if the constant definition were not present, and `MAXIMUM_SCORE` were replaced by 100 in both places where it occurs. The advantage of using `Constant` is that it makes it possible to change this value from 100 to, say, 50 with only a single change, and it makes the source code more legible to the reader by explaining what the significance of the number 100 is supposed to be.

If no value is specified for a constant, as in the line

```
Constant DEBUG;
```

then the constant is created with value 0.

§2.2 Global variables

As commented above, so far the only variables allowed have been "local variables", each private to their own routines. A "global variable" is a variable which is accessible to all code in every routine. Once a global variable has been declared, it is used in just the same way as a local variable. The directive for declaring a global variable is `Global`:

```
Global score = 36;
```

This creates a variable called `score`, which at the start of the program has the value 36. `score` can be altered or used anywhere in the program after the line on which it is defined.

§2.3 Arrays

An "array" is an indexed collection of (global) variables, holding a set of numbers organised into a sequence. It allows general rules to be given for how a group of variables should be treated. For instance, the directive

```
Array pack_of_cards --> 52;
```

creates a stock of 52 variables, referred to in the program as

```
pack_of_cards-->0   pack_of_cards-->1   ...   pack_of_cards-->51
```

There are two basic kinds of array: "word arrays" (written using `-->` as above) and "byte arrays" (written using `->` similarly). Whereas the entries of a word array can hold any number, the entries of a byte array can only be numbers in the range 0 to 255 inclusive.

(The only advantage of this is that it is more economical on memory, and beginners are advised to use word arrays instead.)

In addition to this, Inform provides arrays which have a little extra structure: they are created with the 0th entry holding the number of entries. A word array with this property is called a `table`; a byte array with this property is a `string`.

For example, the array defined by

```
Array continents table 5;
```

has six entries: `continents-->0`, which holds the number 5, and five more entries, indexed 1 to 5. (The program is free to change `continents-->0` later but this will not change the size: the size of an array can never change.) As an example of using `string` arrays:

```
Array password string "DANGER";
Array phone_number string "1978-345-2160";
...
PrintString(password);
...
PrintString(phone_number);
...
[ PrintString the_array i;
    for (i=1: i<=the_array->0: i++)
        print (char) the_array->i;
];
```

The advantage of `string` arrays, then, is that one can write a general routine like `PrintString` which works for arrays of any size.

To recapitulate, Inform provides four kinds of array in all:

```
-->   ->   table   string
```

There are also four different ways to set up an array with its initial contents (so the directive can take 16 forms in all). In all of the examples above, the array entries will all contain 0 when the program begins.

Instead, we can give a list of constant values. For example,

```
Array primes --> 2 3 5 7 11 13;
```

is a word array created with six entries, `primes-->0` to `primes-->5`, initially holding the values 2 to 13.

The third way to create an array gives some text as an initial value (because one common use for arrays is as "strings of characters" or "text buffers"). The two string arrays above were set up this way. As another example,

```
Array players_name -> "Frank Booth";
```

sets up the byte array `players_name` as if the directive had been

```
Array players_name -> ’F’ ’r’ ’a’ ’n’ ’k’ ’ ’ ’B’ ’o’ ’o’ ’t’ ’h’;
```

△△ The fourth way to create an array is obsolete and is kept only so that old programs still work. This is to give a list of values in between end-markers `[` and `]`, separated by commas or semi-colons. Please don't use this any longer.

• **WARNING**

It is up to the programmer to see that no attempt is made to read or write non-existent entries of an array. (For instance, `pack_of_cards-->1000`.) Such mistakes are notorious for causing programs to fail in unpredictable ways, difficult to diagnose. Here for example is an erroneous program:

```
Array ten --> 10;
Array fives --> 5 10 15 20 25;
[ Main n;
  for (n=1: n<=10: n++) ten-->n = -1;
  print fives-->0, "^";
];
```

This program ought to print 5 (since that's the 0-th entry in the array `fives`), but in fact it prints −1. The problem is that the entries of `ten` are `ten-->0` up to `ten-->9`, not (as the program implicitly assumes) `ten-->1` to `ten-->10`. So the value −1 was written to `ten-->10`, an entry which does not exist. At this point anything could have happened. As it turned out, the value was written into the initial entry of the next array along, "corrupting" the data there.

§2.4 Example 7: Shuffling a pack of cards

This program simulates the shuffling of a pack of playing cards. The cards are represented by numbers in the range 0 (the Ace of Hearts) to 51 (the King of Spades). The pack itself has 52 positions, from position 0 (on the top) to position 51 (on the bottom). It is therefore represented by the array

```
pack_of_cards-->i
```

whose `i`-th entry is the card number at position `i`. A new pack as produced by the factory, still in order, would therefore be represented with card `i` in position `i`: the pack would have the Ace of Hearts on top and the King of Spades on the bottom.

△ The example code shuffles the pack in a simple way, but there are more efficient methods. Here's one supplied by Dylan Thurston, giving perfect randomness in 51 exchanges.

```
pack_of_cards-->0 = 0;
for (i=1:i<52:i++)
{   j = random(i+1) - 1;
    pack_of_cards-->i = pack_of_cards-->j; pack_of_cards-->j = i;
}
```

§2.5 Seven special data structures

△ All Inform programs automatically contain seven special data structures, each being one of a kind: the object tree, the grammar, the table of actions, the release number, the serial code, the "statusline flag" and the dictionary. These data structures are tailor-made for adventure games and (except for the object tree) can be ignored for every other kind of program. So they are mostly covered in Book Two.

Example 7: Shuffling a pack of cards

```
Constant SHUFFLES = 100;
Array pack_of_cards --> 52;
[ ExchangeTwo x y z;
  !   Initially x and y are both zero
  while (x==y)
  {   x = random(52) - 1; y = random(52) - 1;
  }
  !   x and y are now randomly selected, different numbers
  !   in the range 0 to 51
  z = pack_of_cards-->x;
  pack_of_cards-->x = pack_of_cards-->y;
  pack_of_cards-->y = z;
];
[ Card n;
  switch(n%13)
  {   0: print "Ace";
      1 to 9: print n%13 + 1;
      10: print "Jack";
      11: print "Queen";
      12: print "King";
  }
  print " of ";
  switch(n/13)
  {   0: print "Hearts";
      1: print "Clubs";
      2: print "Diamonds";
      3: print "Spades";
  }
];
[ Main i;
  !   Create the pack in "factory order":
  for (i=0:i<52:i++) pack_of_cards-->i = i;
  !   Exchange random pairs of cards for a while:
  for (i=0:i<SHUFFLES:i++) ExchangeTwo();
  print "The pack has been shuffled to contain:^";
  for (i=0:i<52:i++)
      print (Card) pack_of_cards-->i, "^";
];
```

Note the use of a "printing rule" called `Card` to describe card number `i`. Note also that 100 exchanges of pairs of cards is only just enough to make the pack appear well shuffled. Redefining `SHUFFLES` as 10000 makes the program take longer; redefining it as 10 makes the result very suspect.

1. For the object tree (and the directives `Object` and `Class`), see §3.
2. For grammar (and the directives `Verb` and `Extend`), see Chapter V, §§26 and 27.
3. For actions (and the `<...>` and `<<...>>` statements and the `##` constant notation), see Chapter III, §9.
4. The release number (which is printed automatically by the library in an Inform-written adventure game) is 1 unless otherwise specified. The directive

```
Release <number>;
```

   does this. Conventionally release 1 would be the first published copy, and releases 2, 3, ... would be amended re-releases. See Chapter III, §7, for an example.
5. The serial number is set automatically to the date of compilation in the form 960822 ("22nd August 1996"). This can be overridden if desired with the directive

```
Serial "dddddd";
```

   where the text must be a string of 6 digits.
6. The "status line flag" chooses between styles of "status line" at the top of an adventure game's screen display. See Chapter IV, §18, for use of the `Statusline` directive.
7. The dictionary is automatically built by Inform. It is a stock of all the English words which the game might want to recognise from what the player has typed: it includes any words written in constants like `'duckling'`, as well as any words given in `name` values or in grammar. For example

```
if (first_word == 'herring') print "You typed the word herring!";
```

   is a legal statement. Inform notices that `herring` – because it is in single quotes – is a word the program may one day need to be able to recognise, so it adds the word to the dictionary. Note that the constant `'herring'` is a dictionary word but the constant `'h'` is the ASCII value of lower-case H. (Single-letter dictionary words are seldom needed, but can be written using an ugly syntax if need be: `#n$h` is the constant meaning "the dictionary word consisting only of the letter H".)

△△ From this description, the dictionary appears to be something into which words are poured, never to re-emerge. The benefit is felt when the `read` statement comes to try to parse some input text:

```
read text_array parse_buffer;
```

It must be emphasized that the `read` statement performs only the simplest possible form of parsing, and should not be confused with the very much more elaborate parser included in the Inform library.

What it does is to break down the line of input text into a sequence of words, in which commas and full stops count as separate words in their own right. (An example is given in Chapter V, §24.) Before using `read`, the entry

```
parse_buffer->0
```

should be set to the maximum number of words which parsing is wanted for. (Any further words will be ignored.) The number of words actually parsed from the text is written in

```
parse_buffer->1
```

and a block of data is written into the array for each of these words:

```
parse_buffer-->(n*2 - 1)
```

holds the dictionary value of the `n`-th word (if `n` counts 1, 2, 3, ...). If the word isn't in the dictionary, this value is zero.

(In addition,

```
parse_buffer->(n*4)
parse_buffer->(n*4 + 1)
```

are set to the number of letters in word `n`, and the offset of word `n` in the text array.)

For example,

```
[ PleaseTypeYesOrNo i;
  for (::)
  {   buffer->0 = 60;
      parse->0 = 1;
      print "Please type ~yes~ or ~no~> ";
      read buffer parse;
      if (parse-->1 == 'yes') rtrue;
      if (parse-->1 == 'no')  rfalse;
  }
];
```

3 The language of objects

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

§3.1 Objects and communication

The objects in a program are its constituent parts: little lumps of code and data. The starting point of an "object-oriented language" is that it's good design to tie up pieces of information in bundles with the pieces of program which deal with them. But the idea goes further:

1. An object is something you can communicate with. (It's like a company where many people work in the same building, sharing the same address: to the outside world it behaves like a single person.)
2. Information inside the object can be kept concealed from the outside world (like a company's confidential files). This is sometimes called "encapsulation".
3. The outside world can only ask the object to do something, and has no business knowing how it will be done. (The company might decide to change its stock-control system one day, but the outside world should never even notice that this has happened, even though internally it's a dramatic shift.)

All three principles have been seen already for routines: (1) you can call a routine, but you can't call "only this part of a routine"; (2) the local variables of a routine are its own private property, and the rest of the program can't find out or alter their values; (3) as long as the routine still accomplishes the same task, it can be rewritten entirely and the rest of the program will carry on working as if no change had been made.

Why bother with all this? There are two answers. First and foremost, Inform was designed to make adventure games, where objects are the right idea for representing items and places in the game. Secondly, the 'object' approach makes sense as a way of organising any large, complicated program.

The other key idea is communication. One can visualise the program as being a large group of companies, constantly writing letters to each other to request information or ask for things to be done. In a typical "message", one object A sends a detailed question or instruction to another object B, which replies with a simple answer. (Again, we've seen this already for routines: one routine calls another, and the other sends back a return value.)

Routines are only one of the four basic kinds of Inform object, which are:

routines, declared using `[...]`;
strings in double-quotes `"like so"`;
collections of routines and global variables, declared using `Object`;
prototypes for such collections, called "classes" and declared using `Class`.

These four kinds are called "metaclasses". If `O` is an object, then the function

```
metaclass(O)
```

will always tell you what kind it is, which will be one of the four values

```
Routine   String   Object   Class
```

For example,

```
metaclass("Violin Concerto no. 1")
```

evaluates to `String`, whereas

```
metaclass(Main)
```

should always be `Routine` (since `Main` should always be the name of the routine where an Inform program begins to run). From §1 we already know about metaclasses `Routine` and `String`, so it's the other two cases which this section will concentrate on.

△△ Why only these four kinds? Why are strings objects, and not (say) variables or dictionary words? Object-oriented languages vary greatly in to what extreme they take the notion of object: in the dogmatic Smalltalk-80, every ingredient of any kind in a program is called an object: the program itself, the number 17, each variable and so on. Inform is much more moderate. Routines, `Object`s and classes are genuinely object-like, and it just so happens that it's convenient to treat strings as objects (as we shall see). But Inform stops there.

§3.2 Built-in functions 2: the object tree

Routines, strings and (as we shall see) classes are scattered about in an Inform program, in no particular order, and nothing links them together. `Object` objects are special in that they are joined up in the "object tree" which grows through every Inform program.

In this tree, objects have a kind of family relationship to each other: each one has a parent, a child and a sibling. (The analogy here is with family trees.) Normally such a relation is another object in the tree, but instead it can be

```
nothing
```

which means "no object at all". For example, consider the tree:

```
Meadow
  !
Mailbox  ->  Player
  !            !
Note         Sceptre   ->   Cucumber  ->   Torch  ->   Magic Rod
                                             !
                                           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 (where grandchildren don't count as children). For instance,

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

It is a bad idea to apply these functions to the value `nothing` (since it is not an object, but a value representing the absence of one). One can detect whether a quantity is a genuine object or not using `metaclass`, for

```
metaclass(X)
```

is `nothing` for any value `X` which isn't an object: in particular,

```
metaclass(nothing)  == nothing
```

△ Hopefully it's clear why the tree is useful for writing adventure games: it provides a way to simulate the vital idea of one thing being contained inside another. But even in non-adventure game programs it can be a convenience. For instance, it is an efficient way to hold tree structures and linked lists of information.

§3.3 Creating objects 1: setting up the tree

The object tree's initial state is created with the directive `Object`. For example,

```
Object "bucket" ...
Object -> "starfish" ...
Object -> "oyster" ...
Object -> -> "pearl" ...
Object -> "sand" ...
```

(where the bulk of the definitions are here abbreviated to "`...`"), sets up the tree structure

```
       "bucket"
          !
     "starfish" --> "oyster" --> "sand"
                        !
                     "pearl"
```

The idea is that if no arrows `->` are given in the `Object` definition, then the object has no parent: if one `->` is given, then the object is a child of the last object to be defined with no arrows; if two are given, then it's a child of the last object defined with only one arrow; and so on. (The list of definitions looks a little like the tree picture turned on its side.)

An object definition consists of a "head" followed by a "body", which is itself divided into "segments" (though there the similarity with caterpillars ends). The head takes the form:

`Object` ⟨arrows⟩ ⟨name⟩ `"textual name"` ⟨parent⟩

but all of these four entries are optional.

1. The ⟨arrows⟩ are as described above. Note that if one or more arrows are given, that automatically specifies what object this is the child of, so a ⟨parent⟩ cannot be given as well.
2. The ⟨name⟩ is what the object can be called inside the program; it's analogous to a variable name.
3. The `"textual name"` can be given if the object's name ever needs to be printed by the program when it is running.
4. The ⟨parent⟩ is an object which this new object is to be a child of. (This is an alternative to supplying arrows.)

So much is optional that even the bare directive

```
Object;
```

is allowed, though it makes a nameless and featureless object which is unlikely to be useful.

§3.4 Statements for objects: move, remove, objectloop

The positions of objects in the tree are by no means fixed: they are created in a particular formation but are often shuffled around extensively during the program's execution. (In an adventure game, where the objects represent items and rooms, objects are moved in the tree whenever the player picks something up or moves around.) The statement

`move` ⟨object⟩ `to` ⟨object⟩

moves the first-named object to become a child of the second-named one. All of the first object's own children "move along with it", i.e., remain its own children. For instance, following the example in §3.2 above,

```
move Cucumber to Mailbox;
```

results in the tree

```
Meadow
  !
Mailbox  ----------->  Player
  !                      !
Cucumber  ->  Note     Sceptre  ->  Torch  ->  Magic Rod
                                      !
                                   Battery
```

It must be emphasized that `move` prints nothing on the screen, and indeed does nothing at all except to rearrange the tree. When an object becomes the child of another in this way, it always becomes the "eldest" child in the family-tree sense; that is, it is the new `child()` of its parent, pushing the previous children over into being its siblings. It is, however, illegal to move an object out of such a structure using

```
move Torch to nothing;
```

because `nothing` is not an object as such. The effect is instead achieved with

```
remove Torch;
```

which would now result in

```
Meadow                                                 Torch
  !                                                      !
Mailbox  ----------->  Player                         Battery
  !                      !
Cucumber  ->  Note     Sceptre  ->  Magic Rod
```

So the "object tree" is often fragmented into many little trees.

Since objects move around a good deal, it's useful to be able to test where an object currently is; the condition `in` is provided for this. For example,

```
Cucumber in Mailbox
```

is true if and only if the `Cucumber` is one of the *direct* children of the `Mailbox`. (`Cucumber in Mailbox` is true, but `Cucumber in Meadow` is false.) Note that

```
X in Y
```

is only an abbreviation for

```
parent(X) == Y
```

but it's worth having since it occurs so often.

The one loop statement missed out in §1 was `objectloop`.

`objectloop(`⟨variable-name⟩`)` ⟨statement⟩

runs through the ⟨statement⟩ once for each object in the tree, putting each object in turn into the variable. For example,

```
objectloop(x) print (name) x, "^";
```

prints out a list of the textual names of every object in the tree. (Objects which aren't given any textual names in their descriptions come out as "?".) More powerfully, any condition can be written in the brackets, as long as it begins with a variable name.

```
objectloop(x in Mailbox) print (name) x, "^";
```

prints the names only of those objects which are direct children of the `Mailbox` object.

§3.5 Creating objects 2: `with` properties

So far `Object`s are just tokens with names attached which can be shuffled around in a tree. They become interesting when data and routines are attached to them, and this is what the body of an object definition is for.

The body contains up to four segments, which can occur in any order; each of the four is optional. The segments are called

```
with    has    class    private
```

`class` will be left until later. The most important segment is `with`, which specifies things to be attached to the object. For example,

```
Object magpie "black-striped bird"
  with wingspan, worms_eaten;
```

attaches two variables to the bird, one called `wingspan`, the other called `worms_eaten`. Notice that when more than one variable is given, commas are used to separate them: and the object definition as a whole is ended by a semicolon, as always. The values of the magpie's variables are referred to in the rest of the program as

```
magpie.wingspan
magpie.worms_eaten
```

which can be used exactly the way normal (global) variables are used. Note that the object has to be named along with the variable, since

```
crested_glebe.wingspan
magpie.wingspan
```

are different variables.

Variables which are attached to objects in this way are called "properties". More precisely, the name `wingspan` is said to be a property, and is said to be "provided" by both the `magpie` and `crested_glebe` objects.

The presence of a property can be tested using the `provides` condition. For example,

```
objectloop (x provides wingspan) ...
```

executes the code `...` for each object `x` in the game which is defined with a `wingspan` property.

△ Although the provision of a property can be tested, it cannot be changed while the program is running. The value of `magpie.wingspan` may change, but not the fact that the magpie has a `wingspan`.

When the above magpie definition is made, the initial values of

```
magpie.wingspan
magpie.worms_eaten
```

are both 0. To create the magpie with a given wingspan, we have to specify an initial value: we do this by giving it after the name, e.g.

```
Object magpie "black-striped bird"
  with wingspan 5, worms_eaten;
```

and now the program begins with `magpie.wingspan` equal to 5, and `magpie.worms_eaten` still equal to 0. (For consistency perhaps there should be an equals sign before the 5, but if this were the syntax then Inform programs would be horribly full of equals signs.)

△ Properties can be arrays instead of global variables. If two or more consecutive values are given for the same property, it becomes an array. Thus,

```
Object magpie "black-striped bird"
  with name "magpie" "bird" "black-striped" "black" "striped",
       wingspan 5, worms_eaten;
```

`magpie.name` is not a global variable (and cannot be treated as such: it doesn't make sense to add 1 to it), it is an `-->` array. This must be accessed using two special operators, `.&` and `.#`.

```
magpie.&name
```

means "the array which is held in magpie's `name` property", so that the actual name values are in the entries

```
magpie.&name-->0
magpie.&name-->1
   ...
magpie.&name-->4
```

The size of this array can be discovered with

```
magpie.#name
```

which evaluates to the twice the number of entries, in this case, to 10. (Twice the number of entries because it is actually the number of byte array, `->`, entries: byte arrays take only half as much storage as word arrays.)

△ `name` is actually a special property created by Inform. It has the unique distinction that textual values in double-quotes (like the five words given in `magpie.name` above) are entered into the game's dictionary, and not treated as ordinary strings. (Normally one would use single-quotes for this. The rule here is anomalous and goes back to the misty origins of Inform 1.) If you prefer a consistent style, using single quotes:

```
Object magpie "black-striped bird"
  with name 'magpie' 'bird' 'black-striped' 'black' 'striped',
       wingspan 5, worms_eaten;
```

works equally well (except that single-character names like "X" then have to be written `#n$X`).

Finally, properties can also be routines. In the definition

```
Object magpie "black-striped bird"
  with name "magpie" "bird" "black-striped" "black" "striped",
       wingspan 5,
       flying_strength
       [;  return magpie.wingspan + magpie.worms_eaten;
       ],
       worms_eaten;
```

`magpie.flying_strength` is neither a variable nor an array, but a routine, given in square brackets as usual. (Note that the Object directive continues where it left off after the routine-end marker, `]`.) Routines which are written in as property values are called "embedded" and are mainly used to receive messages (as we shall see).

△△ Embedded routines are unlike ordinary ones in two ways:

1. An embedded routine has no name of its own, since it is referred to as a property such as `magpie.flying_strength` instead.
2. If execution reaches the `]` end-marker of an embedded routine, then it returns `false`, not `true` (as a non-embedded routine would). The reason for this will only become clear in Chapter III when `before` and `after` rules are discussed.

§3.6 `private` properties and encapsulation

△ An optional system is provided for "encapsulating" certain properties so that only the object itself has access to them. These are defined by giving them in a segment of the object declaration called `private`. For instance,

```
Object sentry "sentry"
  private pass_number 16339,
  with    challenge
          [ attempt;
              if (attempt == sentry.pass_number)
                  "Approach, friend!";
              "Stand off, stranger.";
          ];
```

makes the sentry provide two properties: `challenge`, which is public, and `pass_number`, which can be used only by the sentry's own embedded routines.

△△ This makes the `provides` condition slightly more interesting than it appeared in the previous section. The answer to the question of whether or not

```
sentry provides pass_number
```

depends on who's asking: this condition is true if it is tested in one of the sentry's own routines, and otherwise false. A `private` property is so well hidden that nobody else can even know whether or not it exists.

§3.7 Attributes, `give` and `has`

In addition to properties, objects have flag variables attached. (Recall that flags are variables which are either true or false: the flag is either flying, or not.) However, these are provided in a way which is quite different. Unlike property names, attribute names have to be declared before use with a directive like:

```
Attribute tedious;
```

Once this declaration is made, every object in the tree has a `tedious` flag attached, which is either true or false at any given time. The state can be tested by the `has` condition:

```
if (magpie has tedious) ...
```

tests whether the magpie's `tedious` flag is currently set, or not.

The magpie can be created already having attributes using the `has` segment in its declaration:

```
Object magpie "black-striped bird"
  with wingspan, worms_eaten
  has  tedious;
```

The `has` segment contains a list of attributes (with no commas in between) which should be initially set. In addition, an attribute can have a tilde `~` in front, indicating "this is definitely not held". This is usually what would have happened anyway, but class inheritance (see below) disturbs this.

Finally, the state of such a flag is changed in the running of the program using the `give` statement:

```
give magpie tedious;
```

sets the magpie's `tedious` attribute, and

```
give magpie ~tedious;
```

clears it again. The give statement can take a list of attributes, too:

```
give door ~locked open;
```

for example, meaning "take away `locked` and add on `open`".

§3.8 Classes and inheritance

Having covered routines and strings in §1, and `Objects` above, the fourth and final metaclass to discuss is that of "classes". A class is a kind of prototype object from which other objects are copied. These other objects are sometimes called "instances" or "members" of the class, and are said to "inherit from" it.

For example, clearly all birds ought to have wingspans, and the property

```
flying_strength
[;  return magpie.wingspan + magpie.worms_eaten;
],
```

(attached to the magpie in the example above) is using a formula which should work for any bird. We might achieve this by using directives as follows:

```
Class Bird
  with wingspan 7,
       flying_strength
       [;  return self.wingspan + self.worms_eaten;
       ],
       worms_eaten;
Bird   "magpie"
  with wingspan 5;
Bird   "crested glebe";
Bird   "Great Auk"
  with wingspan 15;
Bird   "early bird"
  with worms_eaten 1;
```

The first definition sets up a new class called Bird. Every example of a Bird now automatically provides wingspan, a flying_strength routine and a count of worms_eaten. Note that the four actual birds are created using the Bird class-name instead of the usual plain Object directive, but this is only a convenient short form for definitions such as:

```
Object "magpie"
  with wingspan 5
 class Bird;
```

where class is the last of the four object definition segments. It's just a list of classes which the object has to inherit from.

The Bird routine for working out flying_strength has to be written in such a way that it can apply to any bird. It has to say "the flying strength of any bird is equal to its wingspan plus the number of worms it has eaten". To do this, it has used the special value self, which means "whatever object is being considered at the moment". More of this in the next section.

Note also that the Bird with specifies a wingspan of 7. This is the value which its members will inherit, unless their own definitions over-ride this, as the magpie and great Auk objects do. Thus the initial position is:

Bird	Value of wingspan	Value of worms_eaten
magpie	5	0
crested glebe	7	0
Great Auk	15	0
early bird	7	1

△△ In rare cases, clashes between what a class says and what the object says are resolved differently: see §8.

Inform has "multiple inheritance", which means that any object can inherit from any number of classes. Thus, an object has no single class; rather, it can be a member of several classes at once.

Every object is a member of at least one class, because the four "metaclasses" `Routine`, `String`, `Object` and `Class` are themselves classes. (Uniquely, `Class` is a member of itself.) The magpie above is a member of both `Bird` and `Object`.

To complicate things further, classes can themselves inherit from other classes:

```
Class BirdOfPrey
 class Bird
 with  wingspan 15,
       people_eaten;
BirdOfPrey kestrel;
```

makes `kestrel` a member of both `BirdOfPrey` and of `Bird`. Informally, `BirdOfPrey` is called a "subclass" of `Bird`.

Given all this, it's impossible to have a function called `class`, analogous to `metaclass`, to say what class something belongs to. Instead, there is a condition called `ofclass`:

```
kestrel ofclass Class
```

is false, while

```
kestrel ofclass BirdOfPrey
kestrel ofclass Bird
kestrel ofclass Object
"Canterbury" ofclass String
```

are all true. This condition is especially handy for use with `objectloop`:

```
objectloop (x ofclass Bird) move x to Aviary;
```

moves all the birds to the `Aviary`.

§3.9 Messages

That completes the story of how to create objects, and it's time to begin communicating with them by means of messages. Every message has a sender, a receiver and some parameter values attached, and it always produces a reply (which is just a single value). For instance,

```
x = lamp.addoil(5, 80);
```

sends the message `addoil` with parameters 5 and 80 to the object `lamp`, and puts the reply value into `x`. Just as properties like `magpie.wingspan` are variables attached to objects, so messages are received by routines attached to objects, and message-sending is very like making an ordinary Inform function call. The "reply" is what was called the return value in §1, and the "parameters" used to be called function call arguments. But slightly more is involved, as will become apparent.

What does the lamp object do to respond to this message? First of all, it must do something. If the programmer hasn't specified an `addoil` routine for the lamp, then an error message will be printed out when the program is run, along the lines of

```
*** The object "lamp" does not provide the property "addoil" ***
```

Not only does `lamp.addoil` have to exist, but it has to hold one of the four kinds of object, or else `nothing`. What happens next depends on the `metaclass` of `lamp.addoil`:

`metaclass`	What happens:	The reply is:
`Routine`	the routine is called with the the given parameters	the routine's return value
`String`	the string is printed, followed by a new-line	`true`
`Object`	nothing	the object
`Class`	nothing	the class
`nothing`	nothing	`false`, or 0, or `nothing` (all different ways of writing 0)

△ If `lamp.addoil` is a list rather than a single value then the first entry is the one looked at, and the rest are ignored.

For example,

```
print kestrel.flying_strength();
```

will print out 15, by calling the `flying_strength` routine provided by the `kestrel` (the same one it inherited from `Bird`), which adds its wingspan of 15 to the number of worms it has so far eaten (none), and then returns 15. (You can see all the messages being sent in a game as it runs with the debugging verb "messages": see §30 for details.)

△ For examples of all the other kinds of receiving property, here is roughly what happens when the Inform library tries to move the player northeast from the current room (the `location`) in an adventure game:

```
x = location.ne_to();
if (x == nothing) "You can't go that way.";
if (x ofclass Object) move player to x;
```

This allows directions to be given with some flexibility in properties like `ne_to` and so on:

```
Object Octagonal_Room "Octagonal Room"
  with ...
       ne_to "The north-east doorway is barred by an invisible wall!",
       w_to  Courtyard,
       e_to
       [;  if (Amulet has worn)
           {   print "A section of the eastern wall suddenly parts before
                      you, allowing you into...^";
               return HiddenShrine;
           }
       ],
       s_to
       [;  if (random(5) ~= 1) return Gateway;
           print "The floor unexpectedly gives way, dropping you through
                  an open hole in the plaster...^";
           return random(Maze1, Maze2, Maze3, Maze4);
       ];
```

Two special variables help with the writing of message routines: `self` and `sender`. `self` always has as value the `Object` which is receiving the message, while `sender` has as value the `Object` which sent it, or `nothing` if it wasn't sent from `Object` (but from some free-standing routine). For example,

```
pass_number
[;  if (~~(sender ofclass CIA_Operative))
        "Sorry, you aren't entitled to know that.";
    return 16339;
];
```

§3.10 Access to superclass values

△ A fairly common situation in Inform coding is that one has a general class of objects, say `Treasure`, and wants to create an instance of this class which behaves slightly differently. For example, we might have

```
Class  Treasure
  with deposit
       [;  if (self provides deposit_points)
               score = score + self.deposit_points;
           else score = score + 5;
           "You feel a sense of increased esteem and worth.";
       ];
```

and we want to create an instance called Bat_Idol which (say) flutters away, resisting deposition, but only if the room is dark:

```
Treasure Bat_Idol "jewelled bat idol"
  with deposit
       [;  if (location == thedark)
           {   remove self;
               "There is a clinking, fluttering sound!";
           }
           ...
       ];
```

In place of ..., we have to copy out all of the previous code about depositing treasures. This is clumsy: what we really want is a way of sending the deposit message to Bat_Idol but "as if it had not changed the value of deposit it inherited from Treasure". We achieve this with the so-called superclass operator, ::. (The term "superclass" is borrowed from the Smalltalk-80 system, where it is more narrowly defined.) Thus, in place of ..., we could simply write:

```
self.Treasure::deposit();
```

to send itself the deposit message again, but this time diverted to the property as provided by Treasure.

The :: operator works on all property values, not just for message sending. In general,

```
object.class::property
```

evaluates to the value of the given property which the class would normally pass on (or gives an error if the class doesn't provide that property or if the object isn't a member of that class). Note that :: exists as an operator in its own right, so it is perfectly legal to write, for example,

```
x = Treasure::deposit; Bat_Idol.x();
```

To continue the avian theme, BirdOfPrey might have its own flying_strength routine:

```
flying_strength
[;  return self.Bird::flying_strength() + self.people_eaten;
],
```

reflecting the idea that, unlike other birds, these can gain strength by eating people.

§3.11 Philosophy

△△ This section is best skipped until the reader feels entirely happy with the rest of Chapter I. It is aimed mainly at those worried about whether the ideas behind the apparently complicated system of classes and objects are sound. (As Stephen Fry once put it, "Socialism is all very well in practice, but does it work in theory?") We begin with two definitions:

object
a member of the program's object tree, or a routine in the program, or a literal string in the program. (Routines and strings can't, of course, be moved around in the object tree, but the tests `ofclass` and `provides` can be applied to them, and they can be sent messages.) Objects are part of the compiled program produced by Inform.

class
an abstract name for a set of objects in the game, which may have associated with it a set of characteristics shared by its objects. Classes themselves are frequently described by text in the program's source code, but are not part of the compiled program produced by Inform.

Here are the full rules:

(1) Compiled programs are composed of objects, which may have variables attached called "properties".
(2) Source code contains definitions of both objects and classes.
(3) Any given object in the program either is, or is not, a member of any given class.
(4) For every object definition in the source code, an object is made in the final program. The definition specifies which classes this object is a member of.
(5) If an object `X` is a member of class `C`, then `X` "inherits" property values as given in the class definition of `C`.

The details of how inheritance takes place are omitted here. But note that one of the things which can be inherited from class `C` is being a member of some other class, `D`.

(6) For every class definition, an object is made in the final program to represent it, called its "class-object".

For example, suppose we have a class definition like:

```
Class Dwarf
 with beard_colour;
```

The class `Dwarf` will generate a class-object in the final program, also called `Dwarf`. This class-object exists in order to receive messages like `create` and `destroy` and, more philosophically, in order to represent the concept of "dwarfness" within the simulated world.

It is important to remember that the class-object of a class is not normally a member of that class. The concept of dwarfness is not itself a dwarf: the condition `Dwarf ofclass Dwarf` is false. Individual dwarves provide a property called `beard_colour`, but the class-object of `Dwarf` does not: the concept of dwarfness has no single beard colour.

(7) Classes which are automatically defined by Inform are called "metaclasses". There are four of these: `Class`, `Object`, `Routine` and `String`.

It follows by rule (6) that every Inform program contains the class-objects of these four, also called `Class`, `Object`, `Routine` and `String`.

(8) Every object is a member of one, and only one, metaclass:
(8.1) The class-objects are members of `Class`, and no other class.
(8.2) Routines in the program (including those given as property values) are members of `Routine` and no other class.
(8.3) Static strings in the program (including those given as property values) are members of `String`, and of no other class.

(8.4) The objects defined in the source code are members of `Object`, and possibly also of other classes defined in the source code.

It follows from (8.1) that `Class` is the unique class whose class-object is one of its own members: the condition `Class ofclass Class` is true, whereas `X ofclass X` is false for every other class `X`.

There is one other unusual feature of metaclasses, and it is a rule provided for pragmatic reasons (see below) even though it is not very elegant:

(9) Contrary to rules (5) and (8.1), the class-objects of the four metaclasses do not inherit from `Class`.

This concludes the list of rules. To see what they entail, one needs to know the definitions of the four metaclasses. These definitions are never written out in any textual form inside Inform, as it happens, but here are definitions equivalent to what actually does happen. (There is no such directive as `Metaclass`: none is needed, since only Inform itself can define metaclasses, but the definitions here pretend that there is.)

```
Metaclass Object;
```

In other words, this is a class from which nothing is inherited. So the ordinary objects described in the source code only have the properties which the source code says they have.

```
Metaclass Class
with create    [; ... ],
     recreate  [ instance; ... ],
     destroy   [ instance; ... ],
     copy      [ instance1 instance2; ... ],
     remaining [; ... ];
```

So class-objects respond only to these five messages, which are described in detail in the next section, and provide no other properties: **except** that by rule (9), the class-objects `Class`, `Object`, `Routine` and `String` provide no properties at all. The point is that these five messages are concerned with object creation and deletion at run time. But Inform is a compiler and not, like Smalltalk-80 or other highly object-oriented languages, an interpreter. We cannot create the program while it is actually running, and this is what it would mean to send requests for creation or deletion to `Class`, `Object`, `Routine` or `String`. (We could write the above routines to allow the requests to be made, but to print out some error if they ever are: but it is more efficient to have rule (9) instead.)

```
Metaclass Routine
with call      [ parameters...; ... ];
```

Routines therefore provide only `call`. See the next section for how to use this.

```
Metaclass String
with print     [; print_ret (string) self; ],
     print_to_array [ array; ... ];
```

Strings therefore provide only `print` and `print_to_array`. See the next section for how to use these.

To demonstrate this, here is an Inform code representation of what happens when the message

```
O.M(p1, p2, ...)
```

is sent.

```
if (~~(O provides M)) "Error: O doesn't provide M";
P = O.M;
switch(metaclass(P))
{   nothing, Object, Class: return P;
    Routine:                return P.call(p1, p2, ...);
    String:                 return P.print();
}
```

(The messages `call` and `print` are actually implemented by hand, so this is not actually a circular definition. Also, this is simplified to remove details of what happens if `P` is an array.)

§3.12 Sending messages to routines, strings or classes

△ In the examples so far, messages have only been sent to proper `Objects`. But it's a logical possibility to send messages to objects of the other three metaclasses too: the question is whether they are able to receive any. The answer is yes, because Inform provides 8 properties for such objects, as follows.

The only thing you can do with a `Routine` is to call it. Thus, if `Explore` is the name of a routine, then

```
Explore.call(2, 4);      and      Explore(2, 4);
```

are equivalent expressions. The message `call(2,4)` means "run this routine with parameters (2,4)". This is not quite redundant, because it can be used more flexibly than ordinary function calls:

```
x = Explore; x.call(2, 4);
```

The `call` message replies with the routine's return value.

Two different messages can be sent to a `String`. The first is `print`, which is provided because it logically ought to be, rather than because it is useful. So, for example,

```
("You can see an advancing tide of bison!").print();
```

prints out the string, followed by a new-line; the `print` message replies `true`, or 1.

△△ `print_to_array` is more useful. It copies out the text of the string into entries 2, 3, 4, ... of the supplied byte array, and writes the number of characters as a word into entries 0 and 1. That is, if `A` has been declared as a suitably large array,

```
("A rose is a rose is a rose").print_to_array(A);
```

will cause the text of the string to be copied into the entries

```
A->2, A->3, ..., A->27
```

with the value 26 written into

```
A-->0
```

And the reply value of the message is also 26, for convenience.

Five different messages can be sent to objects of metaclass Class, i.e., to classes, and these are detailed in the next section. (But an exception to this is that no messages at all can be sent to the four metaclasses `Class`, `Object`, `Routine` and `String`.)

§3.13 Creating and deleting objects

A vexed problem in all object-oriented systems is that it is often elegant to grow data structures organically, simply conjuring new objects out of mid-air and attaching them to the structure already built. The problem is that since resources cannot be infinite, there will come a point where no new objects can be conjured up. The program must be written so that it can cope with this, and this can present the programmer with some difficulty, since the conditions that will prevail when the program is being run may be hard to predict.

In an adventure-game setting, object creation is useful for something like a beach full of stones: if the player wants to pick up more and more stones, the game needs to create a new object for each stone brought into play.

Inform allows object creation, but it insists that the programmer must specify in advance what the maximum resources ever needed will be: for example, the maximum number of stones which can ever be in play. Although this is a nuisance, the reward is that the resulting program is guaranteed to work correctly on every machine running it (or else to fail in the same way on every machine running it).

The model is this. When a class is defined, a number N is specified, which is the maximum number of created instances of the class which the programmer will ever need at once. When the program is running, "instances" can be created (up to this limit); or deleted. One can imagine the class having a stock of instances, so that creation consists of giving out one of the stock-pile and deletion consists of taking one back.

Classes can receive the following five messages:

`remaining()`
What is the current value of N? That is, how many more instances can be created?

`create()`
Replies with a newly created instance, or with `nothing` if no more can be created.

`destroy(I)`
Destroys the instance `I`, which must previously have been created.

`recreate(I)`
Re-initialises the instance `I`, as if it had been destroyed and then created again.

`copy(I, J)`
Copies `I` to be equal to `J`, where both have to be instances of the class.

△ Note that `recreate` and `copy` can be sent for any instances, not just instances which have previously been created. For example,

```
Plant.copy(Gilded_Branch, Poison_Ivy)
```

copies over all the `Plant` properties and attributes from `Poison_Ivy` to `Gilded_Branch`, but leaves all the rest alone. Likewise,

```
Treasure.recreate(Gilded_Branch)
```

only resets the properties to do with `Treasure`, leaving the `Plant` properties alone.

Unless the definition of a class C is made in a special way, `C.remaining()` will always reply 0, `C.destroy()` will cause an error and `C.create()` will be refused. This is because the magic number N for a class is normally 0.

The "special way" is to give N in brackets after the class name. For example, if the class definition for `Leaf` begins:

```
Class Leaf(100) ...
```

then initially `Leaf.remaining()` will reply 100, and the first 100 `create()` messages will certainly be successful. Others will only succeed if leaves have been destroyed in the mean time. In all other respects `Leaf` is an ordinary class.

△ Object creation and destruction may need to be more sophisticated than this. For example, we might have a data structure in which every object of class `A` is connected in some way with four objects of class `B`. When a new `A` is created, four new `B`s need to be created for it; and when an `A` is destroyed, its four `B`s need to be destroyed. In an adventure game setting, we might imagine that every Dwarf who is created has to carry an Axe of his own.

When an object has been created (or recreated), but before it has been "given out" to the program, a `create` message is sent to it (if it provides `create`). This gives the object a chance to set itself up sensibly. Similarly, when an object is about to be destroyed, but before it actually is, a `destroy` message is sent to it (if it provides `destroy`). For example:

```
Class Axe(30);
Class Dwarf(7)
 with beard_colour,
      create
      [ x; self.beard_colour = random("black", "red", "white", "grey");
           !  Give this new dwarf an axe, if there are any to spare
           x = Axe.create(); if (x ~= nothing) move x to self;
      ],
      destroy
      [ x;
           !  Destroy any axes being carried by this dwarf
           objectloop (x in self && x ofclass Axe) Axe.destroy(x);
      ];
```

§3.14 Footnote on common vs. individual properties

△△ The properties used in the sections above are all examples of "individual properties", which some objects provide and others do not. There are also "common properties" which, because they are inherited from the class `Object`, are held by every member of `Object`. An example is `capacity`. The `capacity` can be read for an ordinary game object (say, a crate) even if it doesn't specify a `capacity` for itself, and the resulting "default" value will be 100. However, this is only a very weak form of inheritance – you can't change the crate's `capacity` value and the condition `crate provides capacity` evaluates to `false`.

The properties defined by the Inform library, such as `capacity`, are all common: mainly because common properties are marginally faster to access and marginally cheaper on memory. Only 62 are available, of which the library uses up 48. Individual properties, on the other hand, are practically unlimited. It is therefore worth declaring a common property only in those cases where it will be used very often in your program. You can declare common properties with the directive:

```
Property ⟨name⟩;
```

which should be made after the inclusion of "Parser" but before first use of the new name. The class `Object` will now pass on this property, with value 0, to all its members. This so-called "default value" can optionally be specified. For example, the library itself makes the declaration

```
Property capacity 100;
```

which is why all containers in a game which don't specify any particular `capacity` can hold up to 100 items.

Chapter II: Using the Compiler

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–)

4 The language of Inform

§4.1 ICL

The Inform compiler is quite configurable: it has a number of settings which can be altered to suit the convenience of the user. Many of these settings are "switches", which usually have just two possible states, off or on. However, some can be set to a single-digit number.

The other numerical settings are "memory settings", which control how much of your computer's memory Inform uses while running (too low and it may not be able to compile games of the size you desire; too high and it may choke any other programs in the computer for space).

Finally, there are "path variables", which contain text and are used to sort out filenames for the files Inform uses or creates. The usage of these variables varies widely from machine to machine, or rather, from one operating system to another.

If Inform seems to work adequately for you already, this section can safely be ignored until the day comes to compile a really big project. Times like that call for the ability to conveniently change many settings at once, and a tiny language called "ICL" is provided for you to supply detailed specifications.

On many systems, though not usually the Apple Macintosh, the user sets Inform running by typing a command at the "command line", that is, in response to a prompt printed by the computer. For example, under RISC OS one would press function key f12 from the desktop and be given the prompt `*`, to which one might reply

```
inform ruins
```

On computers with more doggedly windowed interfaces, there will be a higher-level interface of some kind provided with Inform, which should come with its own brief documentation.

The usual way to alter switches on the command line is to give a word of options after the `inform` command, introduced by a minus sign. The switches are all single letters, and by default are mostly off. For example, the `-x` switch causes Inform to print a row of hash signs as it compiles:

```
inform -x shell
RISC OS Inform 6.01 (April 25th 1996)
::############################################################
```

One hash sign is printed for every 100 textual lines of source code compiled. (On my own machine, an Acorn Risc PC 700, about 10 hashes are printed every second: that is, the compilation speed is about 1000 lines per second.) Although `-x` is provided to indicate that a slow compilation is continuing normally, many designers use it to get a feeling for how large their games are, and it's a morale boost when the row of hashes spills over onto a second screen line.

Inform has documentation built-in on the subject of switches and other ICL features, which may vary from machine to machine. Running Inform with no filename will print this "help information". In addition, `-h1` will print details of filenaming conventions in use on your machine, and `-h2` will print a list of switches and their settings.

The full command line syntax is

`inform` ⟨ICL commands⟩ ⟨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`. This is called "filename translation". No detailed information on filenaming rules is given here, because it varies so much from machine to machine: see the `-h1` on-line documentation. Note however that a filename can contain spaces if it is written in double-quotes.

One possible ICL command is to give a filename in brackets: e.g.,

```
inform -x (skyfall_setup) ...
```

sets the `-x` switch, then runs through the text file `skyfall_setup` executing each line as an ICL command. As an example, this file might read as follows:

```
! Setup file for "Skyfall"
-d                    ! Contract double spaces
$max_objects=1000     ! 500 of them snowflakes
(usual_setup)         ! include my favourite settings, too
+module_path=mods     ! keep modules in the "mods" directory
```

Note that ICL can include comments after !, just as in Inform. Otherwise, an ICL file has one command per line (with no dividing semicolons), and the possibilities are as follows:

-<switches>
set these switches; or unset any switch preceded by a tilde ~. (For example, -a~bc sets a, unsets b and sets c.)
$list
list current memory settings
$?<name>
ask for information on what this memory setting is for
$small
set the whole collection of memory settings to suitable levels for a small game
$large
ditto, for a slightly larger game
$huge
ditto, for a reasonably big one
$<name>=<quantity>
alter the named memory setting to the given level
+<name>=<filename>
set the named pathname variable to the given filename, which should be one or more filenames of directories, separated by commas
compile <filename> <filename>
compile the first-named file, containing source code, writing the output program to the (optional) second-named file
(<filename>)
execute this ICL file (files may call each other in this way)

§4.2 Controlling what is compiled

Several directives instruct Inform to "compile this part next" or "only compile this...". First,

```
Include "filename";
```

instructs Inform to compile the whole of the source code in the given file, and only carry on compiling from here once that is complete. It is exactly equivalent to removing the Include directive and replacing it with the whole file "filename". (The rules for how Inform interprets "filename" vary from machine to machine: run Inform with the -h1 switch for information.) Note that you can write

```
Include ">shortname";
```

to mean "the file called "shortname" which is in the same directory that the present file came from". This is convenient if all the files making up the source code of your game are housed together.

△ Next, there are a number of "conditional compilation" directives. They take the general form of a condition:

```
Ifdef <name>;           Is the name defined as having some meaning?
Ifndef <name>;          Is the name undefined?
Iftrue <condition>;     Is this condition true?
Iffalse <condition>;    Is this condition false?
```

followed by a chunk of Inform and then either

```
Ifnot;
```

and another chunk of Inform, or just

```
Endif;
```

At this point it is perhaps worth mentioning that (most) directives can also be interspersed with statements in routine declarations, provided they are preceded by a # sign. For example:

```
[ MyRoutine;
#Iftrue MAX_SCORE > 1000;
  print "My, what a long game we're in for!^";
#Ifnot;
  print "Let's have a quick game, then.^";
#Endif;
  PlayTheGame();
];
```

which actually only compiles one of the two `print` statements, according to what the value of the constant `MAX_SCORE` is.

△△ Four more arcane directives control conditional compilation.

```
Default <name> <value>;
```

defines ⟨name⟩ as a constant if it wasn't already the name of something: so it's equivalent to the manoeuvre

```
Ifndef <name>;
Constant <name> = <value>;
Endif;
```

Similarly,

```
Stub <name> <number>;
```

defines a routine with this name and number of local variables, if it isn't already the name of something: so it's equivalent to

```
Ifndef <name>;
[ <name> x1 x2 ... x<number>;
];
Endif;
```

△△ Large blocks of code intended to be used in many different games, such as the files which make up the Inform library, should be marked somewhere with the directive

```
System_file;
```

If this is done, it is possible for an outside program including the file to use `Replace`. The idea is that a sequence like:

```
Replace DoSomething;
...
Include "SomeLibrary";
...
[ DoSomething; "Tarantaraa!"; ];
```

allows a routine `DoSomething`, which would normally be defined in the `Include` file `"SomeLibrary"`, to be defined in this file instead. The definition in the `Include` file is simply ignored. In this way, one can override the library routines without actually having to modify the library source code. To recap, the rule here is that a routine's definition is ignored if both (a) it occurs in a declared "system file", and (b) its name has been given in a `Replace` directive.

One way to follow what is being compiled is to use the `Message` directive. The compiler can be made to print messages at compile time using:

```
Message "information"
Message error "error message"
Message fatalerror "fatal error message"
Message warning "warning message"
```

For example,

```
Ifndef VN_1610;
Message fatalerror "This code can only be compiled by Inform 6.1";
Endif;
```

(By a special rule, the condition `VN_1610`-is-defined is true if and only if the version number is 6.10 or more; similarly for other four-digit numbers beginning with a 1.) Informational messages are simply printed: e.g.,

```
Message "Library extension by Boris J. Parallelopiped";
```

just prints out this line (with a carriage return).

§4.3 Using the linker

The process of "linking" is as follows. A game being compiled (called the "external" program) may `Link` one or more pre-compiled sections of code called "modules". Suppose the game Jekyll has a subsection called Hyde. Then these two methods of making Jekyll are, nearly, equivalent:

(i) Putting `Include "Hyde";` in the source code for `"Jekyll"`, and compiling `"Jekyll"`.

(ii) Compiling `"Hyde"` with the `-M` ("module") switch set, then putting `Link "Hyde";` into the same point in the source code for `"Jekyll"`, and compiling `"Jekyll"`.

Option (ii) is much faster as long as `"Hyde"` does not change very often, since its ready-compiled module can be left lying around while `"Jekyll"` is being developed.

Because "linking the library" is by far the most common use of the linker, this is made simple. All you have to do is compile your game with the `-U` switch set, or, equivalently, to begin your source code with

```
Constant USE_MODULES;
```

(This assumes that you already have pre-compiled copies of the two library modules: if not, you'll need to make them with

```
inform -M library.parserm
inform -M library.verblibm
```

(where `library.parserm` should be replaced with the filename for your copy of the library file "parserm", and likewise for "verblibm").) Note that it is essential not to make any `Attribute` or `Property` declarations *before* the `Include "Parser"` line in the source code, though *after* that point is fine. (Library 6/2 and later will print an error message if you make this mistake, but under 6/1 it can be a source of mysterious problems.)

△△ You can also write your own library modules, or indeed subdivide a large game into many modular parts. But there are certain restrictions to the possibilities. (Real experts may want to look at the *Technical Manual* here.) Here's a brief list of these:

1. The module must make the same `Property` and `Attribute` directives as the main program. Including the library file `"linklpa.h"` ("link library properties and attributes") declares the library's stock, so it would be sensible to begin a module with

```
Include "linklpa";
```

and then include a similar file defining all the extra common properties and attributes which are needed by the program (if any).

2. The module cannot contain grammar (i.e., use `Verb` or `Extend` directives) or create fake actions.

3. The module can only use global variables defined outside the module if they are explicitly declared before use using the `Import` directive. For example,

```
Import global frog;
```

allows the rest of the module's source code to refer to the variable `frog` (which must be defined in the outside program). Note that the Include file `"linklv.h"` ("link library variables") imports all the library variables, so it would be sensible to include this.

4. An object in the module can't inherit from a class defined outside the module. (But an object outside can inherit from a class inside.)

5. Certain constant values in the module must be known at module-compile-time (and must not, for instance, be a symbol only defined outside the module). For instance: the size of an array must be known now, not later; the number of duplicate members of a `Class`; and the quantities being compared in an `Iftrue` or `Iffalse`.

6. The module can't: define the `Main` routine; use the `Stub` or `Default` directives; or define an object whose parent object is not also in the same module.

These restrictions are mild in practice. As an example, here is a short module to play with:

```
Include "linklpa";        ! Make use of the properties, attributes
Include "linklv";         ! and variables from the Library
[ LitThings x;
  objectloop (x has light)
      print (The) x, " is currently giving off light.^";
];
```

It should be possible to compile this `-M` and then to `Link` it into another game, making the routine `LitThings` exist in that game.

5 Compiler options and memory settings

It is time to give a full list of the "switches", which are the main way to make choices about how Inform will operate. (This list can always be printed out with the `-h2` switch.)

```
a   trace assembly-language (without hex dumps; see -t)
c   more concise error messages
d   contract double spaces after full stops in text
d2  contract double spaces after exclamation and question marks, too
e   economy mode (slower): make use of declared abbreviations
f   frequencies mode: show how useful abbreviations are
g   traces calls to functions (except in the library)
g2  traces calls to all functions
h   print this information
i   ignore default switches set within the file
j   list objects as constructed
k   output Infix debugging information to "gamedebug"
l   list every statement run through Inform
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 "gametext"
  s   give statistics
  t   trace assembly-language (with full hex dumps; see -a)
  u   work out most useful abbreviations (very very slowly)
  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
  y   trace linking system
  z   print memory map of the Z-machine
  D   insert "Constant DEBUG;" automatically
  E0  Archimedes-style error messages (current setting)
  E1  Microsoft-style error messages
  E2  Macintosh MPW-style error messages
  F1  use temporary files to reduce memory consumption
  M   compile as a Module for future linking
  R0  use filetype 060 + version number for games (default)
  R1  use official Acorn filetype 11A for all games
  T   enable throwback of errors in the DDE
  U   insert "Constant USE_MODULES;" automatically
```

Note that the list may vary slightly from machine to machine: `R0`, `R1` and `T` above are for Acorn RISC OS machines only, for example.

△ Note that these switches can also be selected by putting a `Switches` directive, such as `Switches xdv8s;` right at the start of the source code.

Only two switches have a really drastic effect:

`M` Makes Inform compile a "module", not a "game". See §4.3.

`v` Chooses the format of the game to be compiled. `v5` is the default, but if a game begins to overflow this, try `v8`. (The other settings are intended mainly for maintainers of Infocom interpreters to test their wares.)

`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.

Many of the remaining switches make Inform produce text as it runs, without affecting the actual compilation:

`a l m n t y` 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 RISC OS machines, error throwback will occur in the 'Desktop Development Environment'. `q` causes "this usage is obsolete" warnings to be suppressed, which may be useful when compiling very long, very old programs. 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.
`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 (the filename will be something suitable for your 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 filename will be something suitable for your 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).

`D U` When these switches are set, the constants `DEBUG` (which make the Library add the debugging suite to a game) and `USE_MODULES` (which speeds up compilation by linking in the Library rather than recompiling it) are automatically defined. This is just a convenience: it's a nuisance to keep adding and removing source code lines to do the same thing.

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. Stepping up to `-d2` also contracts 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.

△ There are two directives for setting switches, to be used if there's no other convenient way on your system (for example if you have a poor windowed front end and no command line to type on). These are:

```
Switches ⟨some settings⟩;
Version ⟨number⟩;
```

These can only be used as first lines in the program and are illegal once other directives or routines have been given. Note that

```
Version 6;
```

(for instance) is redundant, as it is equivalent to

```
Switches v6;
```

△ Inform's memory management is very flexible, but sometimes needs attention from the user, rather than being able to tinker with itself automatically. This is unfortunate but Inform has to run in some quite hostile environments and is obliged to be cautious.

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 210K (on my computer). Thus, the total memory consumption of the compiler at work will be about 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.

6 All the Inform error messages

Three kinds of error are reported by Inform: a fatal error is a breakdown severe enough to make Inform stop working at once; an error allows Inform to continue for the time being, but will cause Inform not to finally output the story file (this is to prevent damaged story files being created); and a warning means that Inform suspects you may have made a mistake, but will not take any action itself.

Fatal errors

1. *Too many errors*

```
Too many errors: giving up
```

After 100 errors, Inform stops (in case it has been given the wrong source file altogether, such as a program for a different language altogether).

2. *Input/output problems*
Most commonly, Inform has the wrong filename:

```
Couldn't open input file <filename>
Couldn't open output file <filename>
```

(and so on). More seriously the whole process of file input/output (or "I/O") may go wrong for some reason to do with the host computer: for instance, if it runs out of disc space. Such errors are rare and look like this:

```
I/O failure: couldn't read from temporary file 2
```

Normally you can only have at most 64 files of source code in a single compilation. If this limit is passed, Inform generates the error

```
Program contains too many source files: increase #define MAX_SOURCE_FILES
```

(This might happen if the same file accidentally `Include`s itself.) Finally, if a non-existent path-name variable is set in ICL, the error

```
No such path setting as <name>
```

is generated.

3. *Running out of memory*
If there is not enough memory even to get started, the following appear:

```
Run out of memory allocating <number> bytes for <something>
Run out of memory allocating array of <number>x<number> bytes for <something>
```

(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 §5 above.) In a really colossal game, it is just conceivable that you might hit

```
One of the memory blocks has exceeded 640K
```

which would need Inform to be recompiled to get around (but I do not expect anyone ever to have this trouble). Much more likely is the error

```
The story file/module exceeds version <n> limit (<number>K) by <number> bytes
```

If you're already using version 8, then the story file is full: you might be able to squeeze more game in using the `Abbreviate` directive, but basically you're near to the maximum game size possible. Otherwise, the error suggests that you might want to change the version from 5 to 8, and the game will be able to grow at least twice as large again.

Errors

There are a few conventions. Anything in double-quotes is a quotation from your source code; other strings are in single-quotes. The most common error by far takes the form

```
Expected ... but found ...
```

(of which there are over 100 kinds): most are straightforward to sort out, but a few take some practice. One of the trickiest things to diagnose is a loop statement having been misspelt. For example, the lines

```
pritn "Hello";
While (x==y) print "x is still y^";
```

produce one error each:

```
line 1: Error: Expected assignment or statement but found pritn
line 2: Error: Expected ';' but found print
```

The first is fine. The second is odd: a human immediately sees that `While` is meant to be a `while` loop, but Inform is not able to make textual guesses like this. Instead Inform decides that the code intended was

```
While (x==y); print "x is still y^";
```

with `While` assumed to be the name of a function which hasn't been declared yet. Thus, Inform thinks the mistake is that the ; has been missed out.

In that example, Inform repaired the situation and was able to carry on as normal in subsequent lines. But it sometimes happens that a whole cascade of errors is thrown up, in code which the user is fairly sure must be nearly right. What has happened is that one syntax mistake threw Inform off the right track, so that it continued not to know where it was for many lines in a row. Look at the first error message, fix that and then try again.

1. *Reading in the source-code*

```
Illegal character found in source: (char) <hexadecimal number>
Unrecognised combination in source: <text>
Alphabetic character expected after <text>
No such accented character as <text>
Name exceeds the maximum length of <number> characters: <name>
The following name is reserved by Inform for its own use as a routine name;
    you can use it as a routine name yourself (to override the standard
    definition) but cannot use it for anything else: <name>
The obsolete '#w$word' construct has been removed
Binary number expected after '$$'
Hexadecimal number expected after '$'
Too much text for one pair of 's to hold
Too much text for one pair of "s to hold
```

Note that, for instance, a ^ character is illegal in ordinary source code (producing the first error above), but is allowed within quotation marks.

2. *Variables and arrays*

```
Variable must be defined before use: <name>
'=' applied to undeclared variable
Local variable defined twice: <name>
All 236 global variables already declared
No array size or initial values given
Array sizes must be known now, not externally defined
An array must have a positive number of entries
A 'string' array can have at most 256 entries
Entries in byte arrays and strings must be known constants
Missing ';' to end the initial array values before "[" or "]"
```

The limit of 236 global variables is absolute: a program even approaching this limit should probably be making more use of object properties to store its information. "Entries... must be known constants" is a restriction on what byte or string arrays may contain: basically, numbers or characters; defined constants (such as object names) may only be used if they have already been defined. This restriction does not apply to the more normally used word and table arrays.

3. *Routines and function calls*

```
No 'Main' routine has been defined
It is illegal to nest routines using '#['
A routine can have at most 15 local variables
Argument to system function missing
System function given with too many arguments
Only constants can be used as possible 'random' results
A function may be called with at most 7 arguments
Duplicate definition of label: <name>
```

Note that the system function `random`, when it takes more than one argument, can only take constant arguments (this enables the possibilities to be stored efficiently within the program). Thus `random(random(10), location)` will produce an error. To make a random choice between non-constant values, write a `switch` statement instead.

4. *Expressions and arithmetic*

```
Missing operator: inserting '+'
Evaluating this has no effect: <operator>
'=' applied to <operator>
Brackets mandatory to clarify order of: <operator>
Missing operand for <operator>
Missing operand after <something>
Found '(' without matching ')'
No expression between brackets '(' and ')'
'or' used improperly
Division of constant by zero
Label name used as value: <name>
System function name used as value: <name>
No such constant as <name>
```

"Operators" include not only addition `+`, multiplication `*` and so on, but also more exotic Inform constructs like `-->` ("array entry") and `.` ("property value"). An example of an operator where "Evaluating this has no effect" is in the statement

```
34 * score;
```

where the multiplication is a waste of time, since nothing is done with the result. "= applied to operator" means something like

```
(4 / fish) = 7;
```

which literally means "set 4/`fish` to 7" and results in the error "= applied to /".

"Brackets mandatory to clarify order" means that an ambiguous expression like

```
frogs == ducks == geese
```

requires clarification: which `==` is to be worked out first?

5. *Miscellaneous errors in statements*

```
'do' without matching 'until'
'default' without matching 'switch'
'else' without matching 'if'
'until' without matching 'do'
'break' can only be used in a loop or 'switch' block
At most 32 values can be given in a single 'switch' case
Multiple 'default' clauses defined in same 'switch'
'default' must be the last 'switch' case
'continue' can only be used in a loop block
A reserved word was used as a print specification: <name>
No lines of text given for 'box' display
In Version 3 no status-line drawing routine can be given
The 'style' statement cannot be used for Version 3 games
```

For instance, `print (fixed) X` gives the "reserved word in print specification" error because `fixed` is a reserved statement internal keyword. Anyway, call such a printing routine something else.

6. *Object and class declarations*

```
Two textual short names given for only one object
The syntax '->' is only used as an alternative to 'Nearby'
Use of '->' (or 'Nearby') clashes with giving a parent
'->' (or 'Nearby') fails because there is no previous object
'-> -> ...' fails because no previous object is deep enough
Two commas ',' in a row in object/class definition
Object/class definition finishes with ','
Not an individual property name: <name>
No such property name as <name>
Not a (common) property name: <name>
Property should be declared in 'with', not 'private': <name>
Limit (of 32 values) exceeded for property <name>
Duplicate-number not known at compile time
The number of duplicates must be 1 to 10000
```

Note that "common properties" (those provided by the library, or those declared with `Property`) cannot be made `private`. All other properties are called "individual". The "number of duplicates" referred to is the number of duplicate instances to make for a new class, and it needs to be a number Inform can determine now, not later on in the source code (or in another module altogether). The limit 10000 is arbitrary and imposed to help prevent accidents.

7. *Grammar*

```
Two different verb definitions refer to <name>
```

```
There is no previous grammar for the verb <name>
There is no action routine called <name>
No such grammar token as <text>
'=' is only legal here as 'noun=Routine'
Not an action routine: <name>
This is a fake action, not a real one: <name>
Too many lines of grammar for verb: increase #define MAX_LINES_PER_VERB
```

At present verbs are limited to 20 grammar lines each, though this would be easy to increase. (A grammar of this kind of length can probably be written more efficiently using general parsing routines, however.)

8. *Conditional compilation*

```
'Ifnot' without matching 'If...'
Second 'Ifnot' for the same 'If...' condition
End of file reached in code 'If...'d out
This condition can't be determined
```

"Condition can't be determined" only arises for `Iftrue` and `Iffalse`, which make numerical or logical tests: for instance,

```
Iftrue #strings_offset==$4a50;
```

can't be determined because even though both quantities are constants, the `#strings_offset` will not be known until compilation is finished. On the other hand, for example,

```
Iftrue #version_number>5;
```

can be determined, as the version number was set before compilation.

9. *Miscellaneous errors in directives*

```
You can't 'Replace' a system function already used
Must specify 0 to 3 local variables for 'Stub' routine
A 'Switches' directive must come before the first constant definition
All 48 attributes already declared
All 62 properties already declared
'alias' incompatible with 'additive'
The serial number must be a 6-digit date in double-quotes
A definite value must be given as release number
A definite value must be given as version number
The version number must be in the range 3 to 8
All 64 abbreviations already declared
All abbreviations must be declared together
It's not worth abbreviating <text>
'Default' cannot be used in -M (Module) mode
'LowString' cannot be used in -M (Module) mode
```

10. *Linking and importing*

```
File isn't a module: <name>
Link: action name clash with <name>
Link: program and module give differing values of <name>
Link: module (wrongly) declared this a variable: <name>
Link: this attribute is undeclared within module: <name>
Link: this property is undeclared within module: <name>
Link: this was referred to as a constant, but isn't: <name>
Link: <type> <name> in both program and module
Link: <name> has type <type> in program but type <type> in module
Link: failed because too many extra global variables needed
Link: module (wrongly) declared this a variable: <name>
Link: this attribute is undeclared within module: <name>
Link: this property is undeclared within module: <name>
Link: this was referred to as a constant, but isn't: <name>
'Import' cannot import things of this type: <name>
'Import' can only be used in -M (Module) mode
```

Note that the errors beginning "Link:" are exactly those occurring during the process of linking a module into the current compilation. They mostly arise when the same name is used for one purpose in the current program, and a different one in the module.

11. *Assembly language*

```
Label out of range for branch
Opcode specification should have form "VAR:102"
Unknown flag: options are B (branch), S (store),
    T (text), I (indirect addressing), F** (set this Flags 2 bit)
Only one '->' store destination can be given
Only one '?' branch destination can be given
No assembly instruction may have more than 8 operands
This opcode does not use indirect addressing
Indirect addressing can only be used on the first operand
Store destination (the last operand) is not a variable
Opcode unavailable in this Z-machine version: <name>
Assembly mistake: syntax is <syntax>
Routine contains no such label as <name>
For this operand type, opcode number must be in range <range>
```

12. *None of the above*
If you should see an incomprehensible error message beginning with `***`, then Inform itself has malfunctioned. This is not meant to happen, but it's conceivable that it might occur in the process of linking in a module which has been damaged in some way.

Finally, error messages can also be produced from within the program (deliberately) using `Message`. It may be that a mysterious message is being caused by an included file written by someone other than yourself.

Warnings

1. *Questionable practices*

`This statement can never be reached`

There is no way that the statement being compiled can ever be executed when the game is played. Here is an obvious example:

```
return; print "Goodbye!";
```

where the `print` statement can never be reached, because a `return` must just have happened. Beginners often run into this example:

```
"You pick up the gauntlet."; score=score+1; return;
```

Here the `score=score+1` statement is never reached because the text, given on its own, means "print this, then print a new-line, then return from the current routine". The intended behaviour needs something like

```
print "You pick up the gauntlet.^"; score=score+1; return;
```

`<type> <name> declared but not used`

For example, a `Global` directive was used to create a variable, which was then never used in the program.

`'=' used as condition: '==' intended?`

Although a line like

```
if (x = 5) print "My name is Alan Partridge.";
```

is legal, it's probably a mistake: `x=5` sets `x` to 5 and results in 5, so the condition is always true. Presumably it was a mistype for `x==5` meaning "test `x` to see if it's equal to 5".

```
Unlike C, Inform uses ':' to divide parts of a 'for' loop
    specification: replacing ';' with ':'
```

Programmers used to the C language will now and then habitually type a `for` loop in the form

```
for (i=0; i<10; i++) ...
```

but Inform needs colons, not semicolons: however, as it can see what was intended, it makes the correction automatically and issues only a warning.

`Missing ','? Property data seems to contain the property name <name>`

The following, part of an object declaration, is legal but unlikely:

```
with found_in MarbleHall
     short_name "conch shell", name "conch" "shell",
```

As written, the `found_in` property has a list of three values: `MarbleHall`, `short_name` and `"conch shell"`. `short_name` throws up the warning because Inform suspects that a comma was missed out and the programmer intended

```
with found_in MarbleHall,
     short_name "conch shell", name "conch" "shell",
```

`This is not a declared Attribute: <name>`

Similarly, suppose that a game contains a `pen`. Then the following `give` statement is dubious but legal:

```
give MarbleDoorway pen;
```

The warning is caused because it's far more likely to be a misprint for

```
give MarbleDoorway open;
```

`Without bracketing, the minus sign '-' is ambiguous`

For example,

```
Array Doubtful --> 50 10 -20 56;
```

because Inform is not sure whether this contains three entries, the middle one being $10-20 = -10$, or four. It guesses four, but suggests brackets to clarify the situation.

`Array entry too large for a byte`

Byte `->` and `string` arrays can only hold numbers in the range 0 to 255. If a larger entry is supplied, only the remainder mod 256 is stored, and this warning is issued.

`Verb disagrees with previous verbs: <verb>`

The `Extend only` directive is used to cleave off a set of synonymous English verbs and make them into a new Inform verb. For instance, ordinarily "take", "get", "carry" and "hold" are one single Inform verb, but this directive could split off "carry" and "get" from the other two. The warning would arise if one tried to split off "take" and "drop" together, which come from different original Inform verbs. (It's still conceivably usable, which is why it's a warning, not an error.)

`This does not set the final game's statusline`

An attempt to choose, e.g., `Statusline time` within a module, having no effect on the program into which the module will one day be linked. Futile.

```
This module has a more advanced format than this release of the
    Inform 6 compiler knows about: it may not link in correctly
```

2. *Obsolete usages*

```
more modern to use 'Array', not 'Global'
use '->' instead of 'data'
use '->' instead of 'initial'
use '->' instead of 'initstr'
use 'word' as a constant dictionary address
'#a$Act' is now superceded by '##Act'
'#n$word' is now superceded by ''word''
'#r$Routine' can now be written just 'Routine'
all properties are now automatically 'long'
use the ^ character for the apostrophe in <dictionary word>
```

These all occur if Inform compiles a syntax which was correct under Inform 5 (or earlier) but has now been withdrawn in favour of something better.

△△ No Inform library file (or any other file marked `System_file`) produces warning messages. It may contain many declared but unused routines, or may contain obsolete usages for the sake of backward compatibility.

Chapter III: Fundamentals

7 Getting started

Nothing so difficult as a beginning
In poesy, unless perhaps the end.

– Lord Byron (1788–1824), *Don Juan, IV iv*

The examples in Chapters III and IV of this manual will put together a small game called 'Ruins'. Its first state is very close to 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";
```

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`	a program for decoding what the player types;
`VerbLib`	how verbs, like "take" or "drop", work;
`Grammar`	the grammar table, or what the Parser understands.

(If compilation is annoyingly slow, it should be easy enough to "link the library files", which is much faster: see §4.3) Apart from the inclusions, 'Ruins' contains:

(a) strings (that is, quoted text) giving the game's name and a copyright message, to be printed out when appropriate;
(b) a routine, called `Initialise`, which is run when the game begins, and simply sets where the player starts (not that there's much choice yet!) and prints a 'welcome' message;
(c) an object, so far the only room.

'Ruins' is at this stage an extremely dull 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 960825 / Inform v6.04 Library 6/1
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 an Inform game, objects are used to simulate everything: rooms and items to be picked up, scenery, intangible things like mist and even some abstract ideas (like the direction 'north'). The library is also present in every game, and can be thought of as a referee, or umpire, rather than part of the game's world.

Our second object is added by writing the following just after the `Forest` ends and just before `Initialise` begins:

```
Object -> mushroom "speckled mushroom"
  with name "speckled" "mushroom" "fungus" "toadstool";
```

(The arrow `->` means that the mushroom begins inside the Forest rather than alongside it.) If the game is recompiled, the mushroom is now in play: the player can call it "speckled mushroom", "mushroom", "toadstool" and so on. It can be taken, dropped, looked at, looked under and so on. However, it only adds the rather plain line "There is a speckled mushroom here." to the Forest's description. So here is a more lavish version:

```
Object -> 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`:

```
Object -> 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;
```

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

These show the two kinds of feature something can have: a "property", which has some definite value or list of values (such as `name`), and an "attribute", which is either present or not but has no particular value (such as `edible`). Values of properties change during play, and attributes come and go. For instance,

```
mushroom.description = "You're sure it's a toadstool now.";
give mushroom general;
if (mushroom has edible) print "It's definitely edible.^";
```

manipulate the attributes and properties. (`general` is the name used for an attribute with no particular meaning to the game, but which is left free for your program to use as it likes. Similarly, `number` is a general-purpose property.)

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 see if any special rules apply. In this case, `Take` and `Drop` are the only actions tampered with, and the only effect is that the usual messages ("Taken." "Dropped.") 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 still passive. 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.

Note that if the first text is printed, the rule ends there, and does not flow on into the second text. So one and only one message is printed. Here is how this is achieved: although it's not obvious from the look of the program, the `before` routine is being asked the question "Do you want to interfere with the usual rules?". It must reply, that is, "return", either "true" or "false" meaning yes or no. Because this question is asked and answered many times in a large Inform game, there are several abbreviations for how to reply. For example,

```
return true;
rtrue;
```

both do the same thing. Moreover,

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

performs three useful tasks: prints the message, then prints a carriage return, and then returns true. And this is so useful that a bare string

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

is understood to mean the same thing. To just print the text, the statement `print` has to be written out in full. Here is an example:

```
before
[; Taste: print "You extend your tongue nervously.^";
          rfalse;
];
```

In this rule, the text is printed, but the answer to "Do you want to interfere?" is no, so the game will then go on to print something anodyne like "You taste nothing unexpected." (In fact the `rfalse` was unnecessary, because if a rule like this never makes any decision, then the answer is assumed to be "false".)

• **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.

△ More generally, some `before` or `after` rules ought 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, which the properties `before`, `after`, `life` and `describe` hold to indicate "none".

Here is another typical object definition:

```
Object "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.)

Note that the first line, the so-called header, is a little different in form to those above. Firstly, it gives no "program name" for the steps (in the way that `mushroom` was given as program-name for the speckled mushroom) – there is a blank in between the `Object` directive and the text of the words "short-cut steps". This is perfectly legal, and is sensible because the program never needs to refer to the steps object directly. Secondly, the initial position of the steps is specified not by using arrows `->` but by actually quoting the object it is to be placed inside, the `Forest`. This is sometimes convenient and is only legal if the `Forest` has already been defined (earlier on in the program). Such a restriction is actually useful as it prevents you from setting up a 'loop' – one object in another in a third in the first, for instance.

8 Introducing messages and classes

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 fact, messages have already appeared in §7. Recall from §3 that a message called `messagename` can be sent to an object called `objectname` with various supplied details (called `info1` and `info2` here, though there can be any number from none to 6) as follows:

```
objectname.messagename(info1, info2);
```

And the given object sends back a reply value (which is just a single quantity). This is what is really happening when the player tries to eat the mushroom: first the library sends the mushroom a `before` message to warn it that something will happen; it might reply `true`, in which case the library gives up; otherwise the eating takes place, and the library sends an `after` message to inform the mushroom of its demise.

Properties like `before`, then, are really rules to deal with incoming messages. The same applies to most of the properties in §3. For example, the message

```
mushroom.description();
```

is sent when the player tries to examine the mushroom: if the reply is `false` then the game prints "You see nothing special about the speckled mushroom." Now the mushroom was set up with

```
description
   "The mushroom is capped with blotches, and you aren't at all sure
    it's not a toadstool.",
```

which doesn't look like a rule for receiving a message, but it is one all the same: it means "print this text out, print a new-line and reply `true`". A more complicated rule could have been given instead, as in the following elaboration of the stone-cut steps in 'Ruins':

```
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.";
],
```

`visited` is an attribute which is currently held only by rooms which the player has been to. (The glyphs will be explained later on, as will the `SquareChamber` room, which is where the steps will lead down into.)

The library, i.e., the standard game rules, can send out about 40 different kinds of message, `before` and `description` being two of these. The more interesting an object is, the more ingeniously it will respond to these messages: an object which ignores all incoming messages will be lifeless and inert in play, like a small stone.

△△ In fact there are subtle differences between how the library uses properties, and message-sending: the `name` property, for example, is not really a message-receiver but is just what it appears to be – a list of useful data. Also, the library is careful not to send (for instance) a `description` message to an object which doesn't provide a rule for what to do with one. But the idea is right.

So the library is sending out messages to your objects all the time during play. Your objects can also send each other messages, including "new" ones that the library would never send. It's sometimes convenient to use these to trigger off happenings in the game. For example, suppose the 'Ruins' are home to a parrot which squawks from time to time, for a variety of reasons:

```
Object -> parrot "red-tailed parrot"
  with name "red" "tailed" "red-tailed" "parrot" "bird",
       description
          "Beautiful plumage.",
       squawk
       [ utterance;
           if (self in location)
               print "The parrot squawks, ~", (string) utterance,
                   "! ", (string) utterance, "!~^";
       ],
  has  animate;
```

We might then, for instance, change the `after` rule for dropping the mushroom to read:

```
Drop: parrot.squawk("Drop the mushroom");
      "The mushroom drops to the ground, battered slightly.";
```

so that the wretched creature would squawk "Drop the mushroom! Drop the mushroom!" each time this was done. Likewise, `squawk` messages could be sent for any number of other reasons connected with other objects. But at present it would be an error to send a `squawk` message to any object other than the parrot, since only the parrot has been given a rule telling it what to do if it receives one.

• **EXERCISE 2**
Make a medicine bottle, which can be opened in a variety of ways in the game, so that the opening–code only occurs once in the bottle definition.

In most games there are groups of objects with certain rules in common, which it would be tiresome to have to write out many times. For making such a group, a class definition is the better technique. These closely resemble object definitions, but since they define prototypes rather than actual things, they have no initial location. (An individual tree may be somewhere, but the concept of being a tree has no particular place.) So the 'header' part of the definition is simpler.

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 the following class:

```
 Class Treasure
  with cultural_value 10,
       after
       [; Insert:
                if (second==packing_case)
                    score = score + self.cultural_value;
                "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.";
       ];
```

Note that `self` is a variable, which always means "whatever object I am". If we used it in the definition of the mushroom it would mean the mushroom: used here, it means whatever treasure happens to be being dealt with. (Explanations about `Insert` and `Remove` will come later, but hopefully the idea is clear enough.) An object of the class `Treasure` inherits the properties and attributes it defines: in this case, an object of class `Treasure` picks up the given score and rules automatically. So

```
Treasure statuette "pygmy statuette"
  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 `cultural_value` score of 10 and the rules about taking and dropping. If the statuette had itself set `cultural_value` to 15, say, then the value would be 15, because the object's actual definition always takes priority over anything the class might have specified.

A more unusual artifact:

```
Treasure honeycomb "ancient honeycomb"
  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;
```

The honeycomb now has two `after` rules: a new one of its own, plus the existing one that all treasures have. Both apply, but the new one happens first.

△ So comparing `cultural_value` and `after`, there seems to be an inconsistency. In the first case, an object's own given value wiped out the value from the class, but in the second, the two values were joined up into a list. Why? The reason is that some of the library's properties are special (again) in being what's called "additive", so that their values accumulate into a list when class inheritance takes place. The three useful examples are `before`, `after` and `name`.

△△ Non-library properties you invent (like `squawk` or `cultural_value`) will never be additive, unless you write a directive like:

```
Property additive squawk;
```

(before `squawk` is otherwise mentioned) to say so.

Finally, note that an object can inherit from several classes at once (see §3 for how to give such a definition). Moreover, a class can itself inherit from other classes, so it's easy to make a class for "like Treasure but with `cultural_value` normally 8 instead of 10".

• **REFERENCES**

See 'Balances' for an extensive use of message-sending. • 'Advent' has a treasure-class similar to this one, 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 even when only two objects use them, 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).

9 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 actions. 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). Most actions are triggered off by the game's parser: in fact, the parser's job can be summed up as reducing the player's keyboard commands to actions. Sometimes one action causes another; a really complicated keyboard command ("empty the sack into the umbrella stand") may fire off quite a sequence of actions.

An action is only an attempt to do something: it 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 will happily generate the action `Eat iron_girder` if the player asked to do so in good English. In this case, even if no `before` rule interferes, the normal game rules will ensure that the girder is not consumed.

Actions can also be generated by your own code, and this perfectly simulates the effect of a player typing something. For example, generating a `Look` action makes the game produce a room description as if the player had typed "look". More subtly, suppose the air in the Pepper Room causes the player to sneeze each turn and drop something at random. This could be programmed directly, with objects being moved onto the floor by explicit `move` statements. But then suppose the game also contains a toffee apple, which sticks to the player's hands. Suddenly the toffee apple problem has an unintended solution. So rather than moving the objects directly to the floor, the game should generate `Drop` actions. 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 consistent.

As an example of causing actions, an odorous `low_mist` will soon settle over 'Ruins'. It will have the `description` "The mist carries a rich aroma of broth." The alert player who reads this will immediately type "smell mist", and we want to provide a better response than the game's stock reply "You smell nothing unexpected." An economical way of doing this is to somehow deflect the action `Smell low_mist` into the action `Examine low_mist` instead, so that the "aroma of broth" message is printed in this case too. Here is a suitable `before` rule to do that:

```
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.

Causing an action and then returning `true` (i.e., causing a new action and killing the old one) is so useful that it has an abbreviation, 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`.

At any given time, just one action is under way (though others may be waiting to resume when the current one has finished). This current action is stored in the three variables

```
action      noun      second
```

`noun` and `second` hold the objects involved, or the special value `nothing` if they aren't involved at all. `action` holds the kind of action. Its possible values can be referred to in the program using the `##` notation: for example

```
if (action == ##Look) ...
```

tests to see if the current action is a `Look`.

△ Why have `##` at all, why not just write `Look`? Partly because this way the reader can see at a glance that an action type is being referred to, but also because the name might be wanted for something else. For instance there's a variable called `score` (holding the current game score), quite different from the action type `##Score`.

△△ For a few actions, the 'noun' (or the 'second noun') is actually a number (for instance, "set timer to 20" would probably end up with `noun` being `timer` and `second` being 20). Occasionally one needs to be sure of the difference, e.g., to tell if `second` is holding a number or an object. It's then useful to know that there are two further variables, `inp1` and `inp2`, parallel to `noun` and `second` and usually equal to them – but equal to 1 to indicate "some numerical value, not an object".

The library supports about 120 different actions and any game of serious proportion will add some more of its own. This list is initially daunting but many are used only rarely and others are always knocked down into simpler actions (for example, `<Empty rucksack table>`, meaning "empty the contents of the rucksack onto the table", is broken down into a stream of actions like `<Remove fish rucksack>` and `<PutOn fish table>`). It's useful to know that an object can only enter the player's possession through a `Take` or `Remove` action: block those two and it can never be acquired whatever the player types.

The list of actions is traditionally divided into three groups, called Group 1, Group 2 and Group 3. Group 1 contains 'meta' actions for controlling the game, like `Score` and `Save`, which are treated quite differently from other actions and are not worth listing. Of the rest, actions which normally do something form Group 2, while actions which normally only print a polite refusal form Group 3. Group 2 contains:

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

It should be apparent why these do something. However, an action like `Listen` falls into Group 3: the library would normally respond to it by printing "You hear nothing unexpected." Only if your program interferes (using a `before` rule) can anything happen. Group 3 contains, in rough order of usefulness:

```
Pull, Push, PushDir [push object in direction], Turn,
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`, are often best dealt with by a `life` rule, which will be discussed in §16.

△ A 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`.

△△ The `Search` action (generated by "look inside ⟨container⟩" or "search ⟨something⟩") only ever prints text, but is in Group 2 rather than Group 3 because it does something substantial. It decides whether something is a container, and if there's enough light to see by, it prints out the contents. 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 of containers.

△△ 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 standard stock of actions is easily added to. Two things are necessary to create a new action: first one must provide a routine to make it happen. For instance:

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

Every action has to have a "subroutine" like this, the name of which is always the name of the action with `Sub` appended. Secondly, one must add grammar so that `Blorple` can actually be called for. Far more about grammar in Chapter V: for now we add the simplest of all grammar lines, a directive

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

placed 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. It can't take any nouns, so the parser will complain if the player types "blorple daisy".

`Blorple` is now a typical Group 3 action. `before` rules can be written for it, and it can be triggered off by a statement like

```
<Blorple>;
```

△△ To make it a Group 1 action, define the verb as `meta` (see §26).

△△ To make it a Group 2 action, rewrite the subroutine in the following form:

```
[ WhateverSub;
  ... do whatever the action is supposed to do,
      printing a suitable message and returning
      if it turns out not to be a sensible thing to do...
  if (AfterRoutines()==1) rtrue;
  ... print a suitable message saying that it has been done ...
];
```

(`AfterRoutines` is a library routine which sends suitable `after` messages to see if the objects want to prevent the usual message being printed.)

△ A few of the library's actions fall into none of Groups 1, 2 or 3, though these aren't proper actions at all, but are used only to signal goings-on. For instance, when the player types "throw rock at dalek", the parser generates the action `ThrowAt rock dalek`. As usual the rock is sent a `before` message asking if it objects to being thrown at a Dalek. Since the Dalek may also have an opinion on the matter, another `before` message is sent to the Dalek, but this time as if the action were something called `ThrownAt`. For example, here is a dartboard's response to a dart:

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

Such an imaginary action – usually, as in this case, a perfectly sensible action seen from the point of view of the second object involved, rather than the first – is called a "fake action". The important ones are `ThrownAt`, `Receive` and `LetGo` (the latter two being used for containers: see §11).

△△ If you really need to, you can declare a new fake action with the directive `Fake_action` ⟨Action-name⟩`;`.

• △△ **EXERCISE 3**
`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?

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 might soon happen.
(b) 'During'. The library takes control and decides if the action makes sense according to its normal world model: for example, 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, a complaint is printed and that's all. Otherwise the action is now 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 phrases as "Dropped.").

△ Group 1 actions (like `Score`) have no 'Before' or 'After' stages: you can't (easily) stop them from taking place. They aren't happening in the game's world, but in the player's.

△ 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 §16.
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 library processes the 'During' stage by calling the action's subroutine. (Subroutines like `TakeSub` make up a large part of the library.)

△ The 'After' stage only applies to Group 2 actions, as all Group 3 actions have been packed up at the 'During' stage if not 'Before'. During 'After' the sequence is as follows: `react_after` rules for every object in scope (including the player object); the room's `after`; the first noun's `after` and finally `GamePostRoutine`.

△△ Two things are fake about "fake actions" (see above): they don't have subroutines, and they never occur in the grammar of any verb (so they're never directly generated by the parser).

△ As mentioned above, the parser can generate very peculiar actions, and this sometimes needs to be remembered when writing `before` rules. Suppose a `before` rule intercepts the action of putting the mushroom in the crate, and makes something exciting happen as a result. Now even if the mushroom is, say, sealed up inside a glass jar, the parser might still generate this action: the impossibility won't be realised until 'During' time. So the exciting happening should be written as an `after` rule, when the attempt to put the mushroom in the crate has already succeeded.

• △ EXERCISE 4
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?

△△ To some extent you can even meddle with the 'During' stage (and with the final messages produced), and thus even interfere with Group 1 actions if you are unscrupulous enough, by cunning use of the `LibraryMessages` system. See §21.

• REFERENCES
In a game compiled with the `-D` switch set, typing in 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. • Sometimes you want "fake fake actions" which are fully–fledged actions (with action routines and so on) but are still never generated by the parser (see §16).

Chapter IV: 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*

10 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:

```
Object -> "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:

```
Object -> 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;
```

(The `^` symbol in `"sun^s"` means an apostrophe, so the word is "sun's".) 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 rule 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 its room is entered via the "down" direction. Note that since the message is printed with the `print` command, there is no "return true" from this routine, so it returns false: and so the usual game rules resume after the printing of the message.

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. The mist smells of broth, which means that if the player types "smell" in a place where the mist is, then she should be told about the broth. For this, a `react_before` rule attached to the mist is ideal:

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

This is called a "react" rule because the mist is reacting to the fact that a `Smell` action is taking place nearby. `noun` is compared with zero to see if the player has indeed just typed "smell" (not, say, "smell crocus"). Thus, when the action `Smell` takes place near the mist, it is converted into `Smell low_mist`; whereas the action `Smell crocus` would be left alone.

The five senses all have actions in Inform: `Look`, `Listen`, `Smell`, `Taste` and `Touch`. Of these, `Look` never has a noun attached (because `Examine` is provided for close-ups), `Smell` and `Listen` may or may not have while `Taste` and `Touch` always have.

• **EXERCISE 5**
(Cf. 'Spellbreaker'.) Make an orange cloud descend on the player, which can't be seen through or walked out of.

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`.

• △ **EXERCISE 6**
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 7**
(Cf. 'Trinity'.) How can the entire game map be suddenly east-west reflected?

• △△ **EXERCISE 8**
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).

11 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 `with_key` 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:

```
Object -> 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 9**
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 examples of actions which aren't explicitly requested by the player, but are generated by the game in the course of play (so-called "fake actions").

△ `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 is a typical container:

```
Object -> 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 we want a portable television set with 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. But members of an object which isn't a container are normally assumed by the game to be hidden invisibly inside (they are said to be "not in scope"). We have to override this in order to make the four buttons visible from outside, by giving the television the `transparent` attribute.

• **EXERCISE 10**
Implement a television set with attached power button and screen.

• **EXERCISE 11**
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, like lamps and buttons, as well as possessions, in which case the above solution is unsatisfactory. Fuller details will be given in the "scope addition" rules in §28, 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 12**
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' make a simple puzzle, and the pile of building blocks a complicated one; see also the scales in 'Balances'.

12 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;
Object -> 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:

```
Object -> 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 on each side.

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 13**
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'.

13 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.";
      ],
      battery_power 40,
      before
      [;  Examine: print "It is a heavy-duty archaeologist's lamp, ";
              if (self hasnt on) "currently off.";
              if (self.battery_power < 10) "glowing a dim yellow.";
              "blazing with brilliant yellow light.";
          Burn: <<SwitchOn self>>;
          SwitchOn:
              if (self.battery_power <= 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 lift 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 power 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.”). But 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 §22.

• **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.)

14 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. Usually, "inside" means 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`:

```
Object -> 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`.

If the player gets into a `container` and then closes it, the effect is like being in a different location. (Unless the container has the `transparent` attribute and is therefore see-through.) The interior may be dark, but if there's light to see by, the player will want to see some kind of room description. In any case, many enterable objects ought to look different from inside or on top. Inside a vehicle, a player might be able to see a steering wheel and a dashboard, for instance. On top of a cupboard, it might be possible to see through a skylight window.

For this purpose, any `enterable` object can provide an `inside_description`, which can be a string or a routine to print one, as usual. If the exterior location is still visible, then the "inside description" is added to the normal room description, and otherwise it becomes that description. As an extreme example, suppose that the player gets into a huge cupboard, closes the door behind her and then gets into a plastic cabinet inside that. The resulting room description might read like so:

> **The huge cupboard** (in the plastic cabinet)
> It's a snug little cupboard in here, almost a room in itself.
>
> In the huge cupboard you can see a pile of clothes.
>
> The plastic walls of the cabinet distort the view.

The second line is the `inside_description` for the huge cupboard, and the fourth is that for the plastic cabinet.

• **EXERCISE 14**
(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 15**
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):

```
Object -> 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 16**
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').

15 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 part: even its format depends on what the second part 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 §24 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 17**
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.

16 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 V.

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 §15 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:

```
Object -> 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 18**
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:

```
Object -> 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 §22 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", the result is called an 'order': this is the corresponding idea to an 'action'. (Indeed, if the player types "me, go south" an ordinary `Go s_obj` action is produced.)

The order is sent to the pilot's `orders` property, which may if it wishes obey or react in some other way. Otherwise, the standard game rules 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: e.g., "priest, go onrth".)

△ 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 delores" 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`	`shield`	`player`		
"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 19**
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 20**
(Cf. 'Starcross'.) Construct a computer responding to "computer, theta is 180".

△ The rest of this section starts to overlap much more with Chapter V, 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 §29: 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 21
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 22
Another of Charlotte's rules is that if you say a number, she has to clap that many times. Can you play?

•△ EXERCISE 23
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. Such a word is called an "untypeable verb", and it's useful to help a `grammar` routine to shunt parsing into a piece of game grammar which the player can never use. 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 has no action routine. Sometimes (very rarely) you want a proper action but which still can't be generated by the parser: the following example creates three.

```
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 24**
How can you make a grammar extension to an ordinary verb that will apply only to Dan?

• △ **EXERCISE 25**
Make an alarm clock responding to "alarm, off", "alarm, on" and "alarm, half past seven" (the latter to set its alarm time).

• △ **EXERCISE 26**
Implement a tricorder (from Star Trek) which analyses nearby objects on a request like "tricorder, the quartz stratum".

• △ **EXERCISE 27**
And, for good measure, a replicator responding to commands like "replicator, tea earl grey" and "replicator, aldebaran brandy".

• △△ **EXERCISE 28**
And a communications badge in contact with the ship's computer, which answers questions like "computer, where is Admiral Lebling".

• △△ **EXERCISE 29**
Finally, construct the formidable flight computer Zen.

The next two exercises really belong to §28, 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 §28. 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 30
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 31
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. • 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'.

17 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 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. because it is outdoors 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;

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, **or**
any of the things it "adds to scope" (see Chapter V) have 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 32**
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 §18 for how 'Ruins' makes darkness menacing.

• △ **EXERCISE 33**
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'.

18 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 34**
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 35**
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') can also be attached to an object. 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 especially useful to run creatures which stay in one room and are only active when the player is nearby. 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 §28). 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 36**
(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 37**
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 38
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 39
(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 40
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 41
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 much 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".

19 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()) <Restore>;
```

(If you want to make the status line invisible during an opening sequence, see §33.) `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 §33: '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 §14.)

△△ 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, set

```
player.orders = 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 42**
Why not achieve the same effect by giving the player a react_before rule instead?

• **EXERCISE 43**
(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 44

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 45

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.

20 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;
  switch(ach)
  {   0: "eating a sweet";
      1: "driving the car";
      2: "shutting out the draught";
      3: "building a tower of four";
      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 ";
  switch(score)
  {   0 to 9:   "humble rainforest Tourist.";
      10 to 19: "Investigator.";
      20 to 29: "Acquisitor.";
      30 to 49: "Archaeologist.";
      50:       "Master Archaeologist.";
  }
];
```

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 46**
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).

21 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.

To do this, a game should make a class definition for amulets: called `Amulet`, say. Then

```
if (noun ofclass Amulet) ...
```

will test to see if `noun` is one of the amulets, for instance.

The amulet's spell will be represented by the property `amulet_spell`. Typical values for this might be:

```
amulet_spell "The spell fizzles out with a dull phut! sound.",
amulet_spell
[;  if (location == thedark)
    {   give real_location light;
        "There is a burst of magical light!";
    }
],
amulet_spell HiddenVault,
amulet_spell
[;  return random(LeadRoom, SilverRoom, GoldRoom);
],
```

Then the process of casting the spell for amulet `X` is a matter of sending the message

```
X.amulet_spell();
```

which will reply with either: false, meaning nothing has happened; true, meaning that something did happen; or an object, a room to teleport the player to. Here is a routine which deals with it all:

```
[ CastSub destination;
  if (noun ofclass Amulet)
  {   if (~~(noun provides amulet_spell))
          "[Ooops. Forgot to program this amulet_spell.]";
      destination = noun.amulet_spell();
      switch(destination)
      {   false:   "Nothing happens.";
          true:    ;
          default: print "You are magically teleported to...^";
                   PlayerTo(destination);
      }
  }
  else "You only know how to cast spells with amulets.";
];
```

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

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

△ The ordinary Library's own properties, such as `description` or `e_to`, are called "common properties". They are special for the following reason: if an object `O` does not give any value for common property `P`, then `O.P` can still be looked up, though it can't be set to something else. (If you tried this with a property of your own invention, such as `amulet_spell` above, an error would be printed out at run-time.) The value of `O.P` is just the "default value" provided by the Library for property `P`: for example, the default value of `cant_go` is "You can't go that way."

△ But you can change this default value during play, using the library's `ChangeDefault` routine. For instance, at a late stage in the game:

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

Of course this cannot change defaults for properties of your own invention, because they haven't got default values.

△△ Common properties are also slightly faster to perform calculations with: the down side is that there's a strictly limited supply of them (63 in all), of which the library uses up half already. To indicate that a property needs to be a common property, use the `Property` directive. For example:

```
Property door_to;
Property capacity 100;
Property cant_go "You can't go that way.";
```

In the latter cases we are giving default values: in the former case, the default value will just be 0.

Major library extensions are rarely needed. More often, one would like simply to change the stock of standard messages, such as the "Nothing is on sale." which tends to be printed when the player asks to buy something, or the "Taken." printed when something is picked up.

This facility is available as follows. Provide a special object called `LibraryMessages`, which must be defined *between* the inclusion of the "Parser" and "VerbLib" library files. This object should have just one property, a `before` rule. For example:

```
Object LibraryMessages
  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 physically appears in the game, of course. The idea is that the `before` rule is consulted before any message is printed: if it returns false, the standard message is printed; if true, then nothing is printed, as it's assumed that this has already happened.

The `Jump` action only ever prints one message (usually "You jump on the spot."), 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. The messages and numbers are given in §A9. 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, normally set to `"^>"` (new-line followed by `>`). This is printed under the action `Prompt` (actually a fake action existing for exactly this purpose). In this way, the game's prompt can be made context-sensitive, or the "skipped line on screen each turn" convention can be removed.

△ 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 47
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
  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 sensitive 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 48
Write an Inform game in Occitan (a dialect of medieval French spoken in Provence).

The Library is itself written in Inform, and with experience it's not too hard to alter it if need be. But this is an inconvenience and an inelegant way to carry on. So here is the last resort in library modification: 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. It would be normal to copy the definition of `BurnSub` out of the library files into your own code, and then modify that copy as needed.

The most popular routine to replace is `DrawStatusLine`: see §33 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 `child` 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 V: 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*

22 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. Despite this, the business of description takes up a fair part of Chapter V since the designer of a really complex game will eventually need to know almost every rule involved. (Whereas nobody would want to know everything about the parser.)

To begin, the simplest description is the "short name" given to a single object. For instance

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

may result 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 49**
(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.

Inform tries to work out the right indefinite article for any object automatically. In English-language games, it uses 'an' when the short name starts with a vowel and 'a' when it does not (unless the name is plural, when 'some' is used in either case). You can override this by setting `article` yourself. Here are some possibilities:

> 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.

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

> the platinum bar, Benjamin Franklin, 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.

A single object whose name is plural, such as "grapes" or "marble pillars", should be given the attribute `pluralname`. As a result the library might say, e.g., "You can't open those" instead of "You can't open that". It also affects the pronoun "them" and makes the usual indefinite article "some".

△ You can give `animate` objects the attributes `male`, `female` or `neuter` to help the parser understand pronouns properly. `animate` objects are assumed to be male if you set neither alternative.

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.colour)
     {   1: print "blue ";
         2: print "purple ";
         3: print "horrid sludge"; rtrue;
     }
],
```

with "liquid" as the short name in its header. According to whether its colour 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.

△ Rooms can also be given a short_name routine, which is useful to code, say, a grid of four hundred similar locations called "Area 1" up to "Area 400". (They can be represented by just one object in the program.)

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 "list 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 50**
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 (for instance, when the statement `<Look>;` triggers a room description). What happens is: 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 residing there which aren't `concealed` or `scenery`.

Chapter IV mentioned many different properties – `initial`, `when_on`, `when_off` and so on – giving descriptions of what an object looks like when in the same room as the player; some apply to doors, others to switchable objects and so on. All of them can be routines to print text, instead of being strings to print. The precise rules are given below.

But the whole system can be bypassed using the `describe` property. If an object gives a `describe` routine then this takes priority over everything: if it returns true, the library assumes that the object has already been described, and prints nothing further. 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. The library 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 §20 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 possible).
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` (for instance, " (as a werewolf)").
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.

The library 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 (it 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, others 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 §A7 for details: suffice to say here that the process above is equivalent to executing

```
if (Locale(location, "You can see", "You can also see"))
    print " here.^";
```

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

• △△ **EXERCISE 51**
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`. • See the `ScottRoom` class of 'Adventureland' for a radically different way to describe rooms (in pidgin English, like telegraphese).

23 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 available 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 52**

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`, in the range 1 to 1000, 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 53**

Implement the Scrabble pieces.

• △△ EXERCISE 54
Implement the three denominations of coin.

• △△ EXERCISE 55
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'.

24 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. It's perfectly possible to 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 highly 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. (And only 6 if Inform has been told to compile an early-model story file: see §31.) Thus the values of `'polyunsaturate'` and `'polyunsaturated'` are equal. Also, upper case and lower case letters are considered the same. Words are permitted to contain numerals or symbols (but not at present to contain accented characters).

△△ A dictionary word can even contain spaces, full stops or commas. 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 actually be useful (as it was in §16).

△ 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:

```
Object -> 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:

```
Object -> 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 56
Rewrite this to insist that the adjectives must come before the noun, which must be present.

• EXERCISE 57
Create a musician called Princess who, when kissed, is transformed into "`/?%?/` (the artiste formerly known as Princess)".

• EXERCISE 58
(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 §25.

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 59

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 60

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 61

How could the word "#" be made a wild-card, meaning "match any single object"?

• △△ EXERCISE 62

And how could "*" be a wild-card for "match any collection of objects"?

• △△ EXERCISE 63

There is no problem with 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 §26) and the format of the `parse` buffer (see §27).)

• 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.

25 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. Two problems must 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  GoldCoin
  with name "gold" "coin",
       short_name "gold coin",
       plural "gold coins";
```

(and similar silver and bronze coin classes here)

```
Object bag "bag"
  with name "bag"
  has  container open openable;
GoldCoin ->;
GoldCoin ->;
GoldCoin ->;
SilverCoin ->;
SilverCoin ->;
BronzeCoin ->;
```

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 but 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 a `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
```

Putting the word `"crowns"` in their `name` lists is not quite right, because the parser will still think that "crowns" might refer to a specific item. Instead, put in the word `"crowns//p"`. The `//p` marks out the dictionary word "crowns" as one that can refer to more than one game object at once. (So that you shouldn't set this for the word "grapes" if a bunch of grapes was a single game object; you should give that object the `pluralname` attribute instead.) For example the `GoldCoin` class would read:

```
Class  GoldCoin
  with name "gold" "coin" "coins//p",
       short_name "gold coin",
       plural "gold coins";
```

and now when the player types "take coins", the parser interprets this as "take all the coins within reach".

△△ The only snag is that now the word `"coins"` is marked as `//p` everywhere in the game, in all circumstances. Here is a more complicated way to achieve the same result, but strictly in context of these objects alone. We need 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
  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 64**
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'.

26 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" = "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 §16 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;
```

(You can look at the grammar being used in a game with the debugging verb "showverb": see §30 for details.) 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`).

△△ Since this book was first written, the library has been improved so that "take" and "get" each have their own independent grammars. But for the sake of example, suppose they share the grammar written out above. 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`.

△ Finally, the line can end with the word `reverse`. This is only useful if there are objects and numbers in the line which occur in the wrong order. An example from the library's grammar:

```
Verb "show" "present" "display"
                * creature held                  -> Show reverse
                * held "to" creature             -> Show;
```

The point is that the `Show` action expects the first parameter to be an item, and the second to be a person. When the text "show him the shield" is typed in, the parser must reverse the two parameters "him" and "the shield" before causing a `Show` action. On the other hand, in "show the shield to him" the parameters are in the right order already.

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 65**
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 66**
Implement the Crowther and Woods feature of moving from one room to another by typing its name, using a dummy verb.

• △ **EXERCISE 67**
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.

27 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
`topic`	any text at all
`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.

It often happens that several prepositions really mean the same thing for a given verb: "in", "into" and "inside" are often equally sensible. As a convenient shorthand you can write a series of prepositions with slash marks `/` in between, to mean "one of these words". For example:

```
* noun "in"/"into"/"inside" noun      -> Insert
```

prepositions (Note that `/` can only be used with prepositions.)

△ 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 68**
(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 poor 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 §24) 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.

$\boxed{\texttt{topic}}$ This token matches as much text as possible. It should either be at the end of its grammar line, or be followed by a preposition. (The only way it can fail to match is if it finds no text at all.) The library's grammar uses this token for topics of conversation and topics looked up in books (see §§15, 16), hence the name. The parser ignores the text for now (your own code will have to think about it later), and simply sets the variables `consult_from` to the number of the first word of the matched text and `consult_words` to the number of words.

$\boxed{\texttt{special}}$ Obsolete and best avoided.

• **EXERCISE 69**
Write a token to detect low numbers in French, "un" to "cinq".

• △ **EXERCISE 70**
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 71**
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 72**
(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 73**
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 §24) 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 version 3 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 74**
Try to implement the parser's routines `NextWord`, `WordAddress` and `WordLength`.

• △△ **EXERCISE 75**
(Difficult.) Write a general parsing routine accepting any amount of text (including spaces, full stops and commas) between double-quotes as a single token.

• **EXERCISE 76**
How would you code a general parsing routine which never matches anything?

• △△ **EXERCISE 77**
Why would you code a general parsing routine which never matches anything?

• △ **EXERCISE 78**
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 §A9.)

28 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 §17), the objects in the same 'enclosure' as the player;
if not, any objects in the `thedark` object; if the player is inside a dark `container`, then that container.

The 'enclosure' of the player is usually the current location. Formally, it's the outermost object containing the player which remains visible – for instance, if the player is in a transparent cabinet in a closed, huge cupboard in the Stores Room, then the enclosure is the huge cupboard. (Thus items in the huge cupboard are in scope, subject to the remaining rules, but other items in the Stores Room are not.)

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.

△ Compass directions make sense as things. 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 79**
Implement the debugging suite's "scope" verb, which lists all the objects currently in scope.

• **EXERCISE 80**
Write a "megalook" verb, which looks around and examines everything nearby.

Formally, scope determines what you can talk about, which usually means what you can see. But what can you touch? Suppose a locked chest is inside a sealed glass cabinet. The Inform parser will allow the command "unlock chest with key" and generate the appropriate action, `Unlock chest key`, because the chest is in scope, so the command at least makes sense.

But it's impossible to carry out, because the player can't reach through the solid glass. So the library's routine for handling the `Unlock` action needs to enforce this. The library does this using a stricter rule called "touchability". The rule is that you can touch anything in scope unless there's a closed container between you and it. This applies either if you're in the container, or if it is.

Some purely visual actions don't require touchability – `Examine` or `LookUnder`, for instance. But most actions are tactile, and so will many actions created by designers. If you want to make your own action routines enforce touchability, you can call the library routine `ObjectIsUntouchable(obj)`. This either returns `false` if there's no problem in touching `obj`, or returns `true` and prints a suitable message (such as "The solid glass cabinet is in the way."). Thus, the first line of many of the library's action routines is:

```
if (ObjectIsUntouchable(noun)) return;
```

You can also call `ObjectIsUntouchable(obj, true)` to simply return true or false, and print nothing, if you'd rather provide your own failure message.
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 $\boxed{\texttt{scope=Topic}}$, 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 81**
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 `PlaceInScope` 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 $\boxed{\texttt{scope=}\langle\text{Routine}\rangle}$ 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 §16 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 82
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 83
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 84**
(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 85**
(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 86**
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 §16 for further scope trickery.

29 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 pronouns (in English, "it", "him", "her" and "them"). The variable `vague_word` contains the dictionary address of which is involved (`'it'`, `'him'`, etc.).

You can find out the current setting of a pronoun using the library's `PronounValue` routine: for instance, `PronounValue('it')` would give the object which "it" currently refers to (possibly `nothing`). Similarly `SetPronoun('it', magic_ruby)` would set "it" to mean the magic ruby object. (When something like a magic ruby suddenly appears in the middle of a turn, players will habitually call it "it".) A better way to adjust the pronouns is to call `PronounNotice(magic_ruby)`, which sets whatever pronouns are appropriate. That is, it works out if the object is a thing or a person, of what number and gender, which pronouns apply to it in the parser's current language, and so on. In code predating Inform 6.1 you may see variables called `itobj`, `himobj` and `herobj` holding the English pronoun values: these still work properly, but please use the modern system in new games.

△ 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 has decided against, 1 means it has decided in favour. The routine should reply

0	(or false) to say "carry on";
1	to force the object to be included; or
2	to force the object 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 parsers in most of the Infocom games, 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 87

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 VI: Testing and Hacking

30 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 'Sorcerer', which is a relatively simple 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, which will be added to any game compiled with the `-D` switch. If you prefer, you can include them manually by writing

```
Constant DEBUG;
```

`DEBUG` somewhere in the program before the library files are included. (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:

```
showobj <anything>
purloin <anything>
abstract <anything> to <anything>
tree               tree <anything>
scope              scope <anything>
showverb <verb>
goto <number>      gonear <anything>
actions    actions on    actions off
routines   routines on   routines off
messages   messages on   messages off
timers     timers on     timers off
trace      trace on      trace off    trace <1 to 5>
recording  recording on  recording off
replay
random
```

"showobj" is very informative about the current state of an object. 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"). "showverb" will display the grammar being used when the given verb is parsed. 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). It also describes all messages sent in the game, which is why it can also be written as "messages". 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 source-level debugger for Inform, called Infix, has been planned for some years, and may possibly be coming to fruition soon.

△ For the benefit of such tools, Inform (if compiling with the `-k` 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 *Technical Manual* for details.

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). A Standard interpreter 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).

31 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.

Games can be compiled to several "versions" of the run-time format. Unless told otherwise Inform compiled to what used to be called Advanced games (version 5). It can still compile Standard games (version 3) but doing so imposes genuine restrictions, and there is little point any more. Stepping up to the new version 8, on the other hand, allows much larger games to be compiled. Other versions exist but are not useful to present-day game designers, so the real decision is V5 versus V8.

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	512	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.

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 common 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. Note that the number of different individual properties is unlimited.

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. This restriction greatly increases 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 and V4, at most 3.)

Recursion and stack usage. The limit on this is rather technical (see the *Z-Machine Standards Document*). 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. When choosing abbreviations, avoid proper nouns and instead pick on short combinations of a space and common two- or three-letter blocks. Good choices include `" the "`, `"The"`, `", "`, `"and"`, `"you"`, `" a "`, `"ing"`, `" to"`. 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.

32 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*

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 *Z-Machine Standards Document*.

△△ 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 §10.

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 88**
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^",
        HelpMenu, HelpInfo);
```

Note the layout, and especially the carriage returns. The second and third arguments are themselves routines. (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 89**
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.

33 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 might look ugly on some incomplete or wrongly-written interpreters: so if you're doing anything complicated, test it as widely as possible.

The best-researched and most reliable interpreters available by far are Mark Howell's Zip and Stefan Jokisch's Frotz: they are also faster than their only serious rival, the InfoTaskForce, a historically important work which is fairly thorough (and should give little trouble in practice) but which was written when the format was a little less well understood. In some ports, ITF gets rarer screen effects wrong, and it lacks an "undo" feature, so the Inform "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'.)

Interpreters conforming to the Z-Machine Standard, usually but not always derived from Frotz or Zip, are reliable and widely available. But remember that one source of unportability is inevitable. Your game may be running on a screen which is anything from a 64 characters by 9 pocket organiser LCD display, up to a 132 by 48 window on a 21-inch monitor.

Anyone wanting to really push the outer limits (say, by implementing Space Invaders or NetHack) will need to refer to *The Z-Machine 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=0`) 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 §27 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 *Z-Machine Standards Document* 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**
Some of these features may not work well on obsolete interpreters which do not adhere to the Z-Machine Standard. Standard interpreters are widely available, but if seriously worried you can test whether your game is running on a good interpreter:

```
if (standard_interpreter == 0)
{   print "This game must be played on an interpreter obeying the
           Z-Machine Standard.^";
    @quit;
}
```

• **EXERCISE 90**
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 91**
Design a title page for 'Ruins', displaying a more or less apposite quotation and waiting for a key to be pressed.

• △ **EXERCISE 92**
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 93**
Alter the 'Advent' example game to display the number of treasures found instead of the score and turns on the status line.

• △ **EXERCISE 94**
(From code by Joachim Baumann.) Put a compass rose on the status line, displaying the directions in which the room can be left.

• △△ **EXERCISE 95**
(Cf. 'Trinity'.) Make the status line consist only of the name of the current location, centred in the top line of the screen.

• △△ **EXERCISE 96**
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.

Appendix: Tables and summaries

A1 Inform operators

In the table, "Level" refers to precedence level: thus `*`, on level 6, has precedence over `+`, down on level 5, but both subordinate to unary `-`, up on level 8. The "associativity" of an operator is the way it brackets up if the formula doesn't specify this: for instance, `-` is left associative because

```
a - b - c
```

is understood as

```
(a - b) - c
```

with brackets going on the left. With some Inform operators, you're not allowed to be vague like this; these are the ones whose associative is listed as "none". Thus

```
a == b == c
```

will produce an error insisting that brackets be written into the program to make clear what the meaning is. Given the table (and sufficient patience) all expressions can be broken down into order: for instance

```
a * b .& c --> d / - f
```

is calculated as

```
( a*((b.&c)-->d) ) / (-f)
```

Level	Operator	Usage	Assoc.	Purpose
0	,	binary	left	separating values to work out
1	=	binary	right	set equal to
2	&&	binary	left	logical AND
2	||	binary	left	logical OR
2	~~	unary	(pre)	logical NOT
3	==	binary	none	equal to?
3	~=	binary	none	not equal to?
3	>	binary	none	greater than?
3	>=	binary	none	greater than or equal to?
3	<	binary	none	less than?
3	<=	binary	none	less than or equal to?
3	has	binary	none	object has this attribute?
3	hasnt	binary	none	object hasn't this attribute?
3	in	binary	none	first obj a child of second?
3	notin	binary	none	first obj not a child of second?
3	ofclass	binary	none	obj inherits from class?
3	provides	binary	none	obj provides this property?
4	or	binary	left	separating alternative values
5	+	binary	left	16-bit signed addition
5	-	binary	left	16-bit signed subtraction
6	*	binary	left	16-bit signed multiplication
6	/	binary	left	16-bit signed integer division
6	%	binary	left	16-bit signed remainder
6	&	binary	left	bitwise AND
6	|	binary	left	bitwise OR
6	~	unary	(pre)	bitwise NOT
7	->	binary	left	byte array entry
7	-->	binary	left	word array entry
8	-	unary	(pre)	16-bit (signed!) negation
9	++	unary	(pre/post)	incrementd
9	--	unary	(pre/post)	decrement
10	.&	binary	left	property address
10	.#	binary	left	property length
11	(...)	binary	left	function call on right hand side
12	.	binary	left	property value
13	::	binary	left	"superclass" operator
14	(...)	binary	left	function call on left hand side

A2 Inform statements

`box` ⟨line-1⟩ ⟨line-2⟩ ... ⟨line-n⟩
`break`
`continue`
`do` ⟨code block⟩ `until` ⟨condition⟩
`font on` *or* `off`
`for (`⟨initial code⟩`:`⟨condition to carry on⟩`:`⟨update code⟩`)` ⟨code block⟩
`give` ⟨object⟩ ⟨attribute-1⟩ ... ⟨attribute-n⟩
`if` ⟨condition⟩ ⟨code block⟩
`if` ⟨condition⟩ ⟨code block⟩ `else` ⟨code-block⟩
`inversion`
`jump` ⟨label⟩
`move` ⟨object⟩ `to` ⟨destination⟩
`new_line`
`objectloop` ⟨condition choosing objects⟩ ⟨code block⟩
`print` ⟨list of printing specifications⟩
`print_ret` ⟨list of printing specifications⟩
`quit`
`read` ⟨text-buffer⟩ ⟨parsing-buffer⟩
`remove` ⟨object⟩
`restore` ⟨label⟩
`return` ⟨optional value⟩
`rfalse`
`rtrue`
`save` ⟨label⟩
`spaces` ⟨number of spaces to print⟩
`string` ⟨number⟩ ⟨text⟩
`style roman` *or* `bold` *or* `underline` *or* `reverse` *or* `fixed`
`switch (`⟨value⟩`)` ⟨block of cases⟩
`while` ⟨condition⟩ ⟨code-block⟩

Statements must be given in lower case. Code blocks consist of either a single statement or a group of statements enclosed in braces `{` and `}`. Print specifications are given as a list of one or more items, separated by commas:

`"`⟨some literal text to print⟩`"`
⟨numerical quantity⟩
`(char)` ⟨a character code⟩
`(string)` ⟨a string address⟩
`(address)` ⟨text held at this byte address⟩
`(name)` ⟨object⟩
`(a)` ⟨object⟩
`(an)` ⟨object⟩
`(the)` ⟨object⟩
`(The)` ⟨object⟩
`(property)` ⟨name of a property⟩
`(object)` ⟨internal "hardware" object short name⟩

A3 Inform directives

`Abbreviate` ⟨word-1⟩ ... ⟨word-n⟩
`Array` ⟨new-name⟩ ⟨type⟩ ⟨initial values⟩
`Attribute` ⟨new-name⟩
`Class` ⟨new-name⟩ ⟨body of definition⟩
`Constant` ⟨new-name⟩ `=` ⟨value⟩
`Default` ⟨possibly-new-name⟩
`End`
`Endif`
`Extend` ⟨grammar extension⟩
`Global` ⟨new-name⟩ `=` ⟨value⟩
`Ifdef` ⟨symbol-name⟩
`Ifndef` ⟨symbol-name⟩
`Ifnot`
`Iftrue` ⟨condition⟩
`Iffalse` ⟨condition⟩
`Import` ⟨list of imported goods⟩
`Include` ⟨source code filename⟩
`Link` ⟨module filename⟩
`Lowstring` ⟨text⟩
`Message` ⟨message-type⟩ ⟨diagnostic-message⟩
`Object` ⟨header⟩ ⟨body of definition⟩
`Property` ⟨new-name⟩
`Release` ⟨number⟩
`Replace` ⟨routine-name⟩
`Serial` `"`⟨serial number⟩`"`
`Switches` ⟨list of switches⟩
`Statusline score` *or* `time`
`System_file`
`Verb` ⟨verb-definition⟩

`Nearby`, much used in Inform 5 code, is still allowed but in modern code the notation `Object ->` is preferable. A few other directives, `Dictionary`, `Fake_action`, `Ifv3`, `Ifv5`, `Stub`, `Trace` and `Version`, are obsolete or for compiler maintenance.

A4 Grammar

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 $0 \leq n \leq 31$. 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 grammar line can optionally be followed by the word `reverse`. This signals that the action to be generated has two parameters, but which have been parsed in the wrong order and need to swapped over. (Note that a `topic` is not a parameter, and nor is a preposition.)

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
`topic`	any text at all
`special`	any single word or number

Two or more literal words (only) can be written with slash signs `/` between them as alternatives. E.g., `"in"/"on"` matches either the word "in" or the word "on".

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.

A5 Library 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 §§28 and 17 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` This object has a feminine name. In games written in English, this makes her a female person, though in other languages it might be inanimate. The parser uses this information when considering pronouns like "her". (In English, anything `animate` is assumed to be male unless `female` or `neuter` is set.)

`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 §17.) 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.

`male` This object has a masculine name. In games written in English, this makes him a male person, though in other languages it might be inanimate. The parser uses this information when considering pronouns like "him". (In English, anything `animate` is assumed to be male unless `female` or `neuter` is set.)

`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`.
`neuter`	This object's name is neither masculine nor feminine. (In English, anything without `animate` is assumed neuter, because only people and higher animals have gender. Anything `animate` is assumed male unless `female` or `neuter` is set. A robot, for instance, might be an `animate` object worth making `neuter`.)
`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`.
`pluralname`	This single object's name is in the plural. For instance, an object called "seedless grapes" should have `pluralname` set. The library will then use the pronoun "them" and the indefinite article "some" automatically.
`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.

A6 Library 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

`articles`	Array of strings

For objects: If given, these are the articles used with the object's name. (Provided for non-English languages where irregular nouns may have unusual vowel-contraction rules with articles: e.g. with French non-mute 'H'.)

`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`.

`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

`inside_description` — String or routine

For objects Printed as part or all of a room description when the player is inside the given object, which must be `enterable`.

`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 §16, 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 §16 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.

`short_name_indef`	Routine

For objects If set, this form of the short name is used when the name is prefaced by an indefinite article. (This is not useful in English-language games, but in other languages adjectival parts of names agree with the definiteness of the article.)

`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 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).

A7 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.
`InformLibrary`	Represents the library. You never need to use it, but it might sometimes be the value of `sender` when a message is received.
`InformParser`	Represents the parser.

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`.
`ObjectIsUntouchable`	Determines whether any solid barrier (that is, any `container` that is not `open`) lies between the player and `obj`. If `flag` is set, this routine never prints anything; otherwise it prints a message like "You can't, because ... is in the way." if any barrier is found. Returns `true` if a barrier is found, `false` if not.
`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` prop-

	erty).
`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.

A8 Library actions

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 `Replace`d 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 §14.

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`.

A9 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/those."
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/They're not something you can close." 2. "That's/They're already closed." 3. "You close ⟨x1⟩."
Consult: "You discover nothing of interest in ⟨x1⟩."
Cut: "Cutting that/those up would achieve little."
Dig: "Digging would achieve nothing here."
Disrobe: 1. "You're not wearing that/those." 2. "You take off ⟨x1⟩."
Drink: "There's nothing suitable to drink here."

Drop: 1. "The ⟨x1⟩ is/are already here." 2. "You haven't got that/those." 3. "(first taking ⟨x1⟩ off)" 4. "Dropped."

Eat: 1. "That's/They're plainly inedible." 2. "You eat ⟨x1⟩. Not bad."

EmptyT: 1. ⟨x1⟩ " can't contain things." 2. ⟨x1⟩ " is/are closed." ⟨x1⟩ " is/are empty already."

Enter: 1. "But you're already on/in ⟨x1⟩." 2. "That's/They're not something you can enter." 3. "You can't get into the closed ⟨x1⟩." 4. "You can only get into something freestanding." 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/are 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/don'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/are in the way." 6. "You can't, since ⟨x1⟩ leads nowhere."

Insert: 1. "You need to be holding ⟨x1⟩ before you can put it/them into something else." 2. "That/Those can't contain things." 3. "⟨x1⟩ is/are closed." 4. "You'll need to take it/them off first." 5. "You can't put something inside itself." 6. "(first taking it/them 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/They don't seem to be something you can lock." 2. "That's/They're locked at the moment." 3. "First you'll have to close ⟨x1⟩." 4. "That doesn't/Those don't seem to fit the lock." 5. "You lock ⟨x1⟩."

Look: 1. " (on ⟨x1⟩)" 2. " (in ⟨x1⟩)" 3. " (as ⟨x1⟩)" 4. "^On ⟨x1⟩ is/are ⟨list⟩" 5. "[On/In ⟨x1⟩] you/You can also see ⟨list⟩ [here]." 6. "[On/In ⟨x1⟩] you/You can see ⟨list⟩ [here]."

LookUnder: 1. "But it's dark." "You find nothing of interest."

Mild: "Quite."

ListMiscellany: 1. " (providing light)" 2. " (which is/are closed)" 3. " (closed and providing light)" 4. " (which is/are empty)" 5. " (empty and providing light)" 6. " (which is/are closed and empty)" 7. " (closed, empty and providing light)" 8. "

(providing light and being worn" 9. " (providing light" 10. " (being worn" 11. " (which is/are " 12. "open" 13. "open but empty" 14. "closed" 15. "closed and locked" 16. " and empty" 17. " (which is/are empty)" 18. " containing " 19. " (on " 20. ", on top of " 21. " (in " 22. ", inside "

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." 17. "It is pitch dark, and you can't see a thing." 18. "yourself" (the short name of the `selfobj` object) 19. "As good-looking as ever." 20. "To repeat a command like "frog, jump", just say "again", not "frog, again"." 21. "You can hardly repeat that." 22. "You can't begin with a comma." 23. "You seem to want to talk to someone, but I can't see whom." 24. "You can't talk to ⟨x1⟩." 25. "To talk to someone, try "someone, hello" or some such." 26. "(first taking `not_holding`)" 27. "I didn't understand that sentence." 28. "I only understood you as far as wanting to " 29. "I didn't understand that number." 30. "You can't see any such thing." 31. "You seem to have said too little!" 32. "You aren't holding that!" 33. "You can't use multiple objects with that verb." 34. "You can only use multiple objects once on a line." 35. "I'm not sure what "⟨pronoun⟩" refers to." 36. "You excepted something not included anyway!" 37. "You can only do that to something animate." 38. "That's not a verb I recognise." 39. "That's not something you need to refer to in the course of this game." 40. "You can't see "⟨pronoun⟩" (⟨value⟩) at the moment." 41. "I didn't understand the way that finished." 42. "None/only ⟨x1⟩ of those is/are available." 43. "Nothing to do!" 44. "There are none at all available!" 45. "Who do you mean, " 46. "Which do you mean, " 47. "Sorry, you can only have one item here. Which exactly?" 48. "Whom do you want [⟨actor⟩] to ⟨command⟩?" 49. "What do you want [⟨actor⟩] to ⟨command⟩?" 50. "Your score has just gone up/down by ⟨x1⟩ point/points." 51. "(Since something dramatic has happened, your list of commands has been cut short.)" 52. "Type a number from 1 to ⟨x1⟩, 0 to redisplay or press ENTER." 53. "[Please press SPACE.]"

No: see **Yes**

NotifyOff: "Score notification off."

NotifyOn: "Score notification on."

Objects: 1. "Objects you have handled:^" 2. "None." 3. " (worn)" 4. " (held)" 5. " (given away)" 6. " (in ⟨x1⟩)" [without article] 7. " (in ⟨x1⟩)" [with article] 8. " (inside ⟨x1⟩)" 9. " (on ⟨x1⟩)" 10. " (lost)"

Open: 1. "That's/They're not something you can open." 2. "It seems/They seem to be locked." 3. "That's/They're already open." 4. "You open ⟨x1⟩, revealing ⟨children⟩" 5. "You open ⟨x1⟩."

Order: "⟨x1⟩ has/have better things to do."

Places: "You have visited: "

Pray: "Nothing practical results from your prayer."

Prompt: 1. "`^>`"

Pronouns: 1. "At the moment, " 2. "means " 3. "is unset " 4. "no pronouns are known to the game."

Pull: 1. "It is/Those are 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/them 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/them 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/They are unfortunately closed." 2. "But it isn't/they aren'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 ⟨x1⟩ is/are closed." 6. "⟨x1⟩ is/are empty." 7. "In ⟨x1⟩ is/are ⟨list of children⟩."

Set: "No, you can't set that/those."

SetTo: "No, you can't set that/those to anything."

Show: 1. "You aren't holding ⟨x1⟩." 2. "⟨x1⟩ is/are 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/They're not something you can switch." 2. "That's/They're already off." 3. "You switch ⟨x1⟩ off."

SwitchOn: 1. "That's/They're not something you can switch." 2. "That's/They're 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/those." 6. "That seems/Those seem to belong to ⟨x1⟩." 7. "That seems/Those seem to be a part of ⟨x1⟩." 8. "That isn't/Those aren't available." 9. "⟨x1⟩ isn't/aren't open." 10. "That's/They're hardly portable." 11. "That's/They're 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."
Turn: see **Pull**
Unlock: 1. "That doesn't seem to be something you can unlock." 2. "It's/They're unlocked at the moment." 3. "That doesn't/Those don'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/those." 2. "You look ridiculous waving ⟨x1⟩."
WaveHands: "You wave, feeling foolish."
Wear: 1. "You can't wear that/those!" 2. "You're not holding that/those!" 3. "You're already wearing that/those!" 4. "You put on ⟨x1⟩."
Yes: "That was a rhetorical question."

A10 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 §29.
`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 every-day 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 §29).
`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.

A11 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. If you are linking in the library (using `USE_MODULES`) then you may not use the `Attribute` or `Property` directive in this part of the program.
   (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. 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.

A12 A short Inform lexicon

This brief dictionary of Inform jargon defines terms used in the manual, generally excepting language features set in `computer` type. Cross-references are italicised. Everything here is in the body of the text somewhere 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 *common 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* uses 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 §33, the rest being in the 'Z-machine Standards Document'.

assembler error A very low-level *error* caused by a malformed line of *assembly language*.

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 §32).

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. Classes also exist as objects in their own right and belong to a *metaclass* called `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.

common property Any *property* set by the *class* `Object` is passed on to every *object* and is called a "common property": for example, `description`. All others are *individual properties*. New properties can be declared as common with the `Property` directive. They behave similarly except that: (a) values of common properties can be read even from an *object* not providing them, the result being a special default value, which can be altered using `ChangeDefault`; (b) they are faster and slightly more economical of memory to use; (c) there are a limited number of them.

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. A numerical value given with no operator is considered true if it is non-zero and otherwise false.

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 words long. 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'.

encapsulation When an *object* declares a *property* as being `private`, its value is unavailable anywhere else in the program: it can be read or written to only by that one object itself. This close concealment of data is called encapsulation.

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* and can only be triggered by a `<...>` statement. The *library* makes use of this but other Inform code is advised not to.

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*.

importing When compiling a *module*, Inform needs to be told of any global variables it is using which are defined only in the outside program (compiled on a different occasion). Such a variable is said to be "imported" using the `Import global` directive.

indirect containment See *object tree*.

individual property Opposite of *common property*.

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, for example the 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, provides the *parser* 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 `Include`d 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.

linking The process of assimilating a previously-compiled *module* into the game now being compiled, in order to save compilation time.

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.

message A way to communicate with an *object*, specifying the object to call, the *property* being addressed (in effect, the "kind of message being sent") and possibly other parameters. A single value is returned as a reply.

metaclass There are four fundamental *classes* of *object*, such that every object belongs to exactly one of the four. These are `Object`, `Class`, `Routine` and `String`, and are called metaclasses. (Since they are examples of classes, they themselves have metaclass `Class`.)

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.

module A previously-compiled but incomplete segment of game, which is kept in order for it to be *linked* into a later compilation. It can be linked many times once created, saving much compilation time. (For example, almost the whole Library can be reduced to two modules.)

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 1. 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*. 2. More generally, classes themselves and even *routines* and *strings* are abstractly considered objects. Objects in sense (1) above, "concrete objects", are members of the *metaclass* `Object`, while classes belong to `Class`, routines to `Routine` and strings to `String`.

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* of metaclass `Object`, i.e., of concretely existing 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.)

private property See *encapsulation*.

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`, which may or may not be *provided* by any given object. Properties can be *encapsulated* for privacy. All properties are either *common* or *individual* (the latter unless otherwise declared).

provision If a *property*, such as `description`, is given in the definition of an *object* (or in the definition of a *class* which the object belongs to) then the object is said to "provide" that property.

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 An Inform program 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*. Calling a routine is analogous to sending a *message* to an *object*, and indeed routines are abstractly considered objects in their own right, belonging to *metaclass* `Routine`.

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*.)

sender When a *message* is sent from one *object* to another, the originator is called the "sender". Whenever a message is being received, the variable `sender` holds this object's identity.

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*), abstractly considered an *object* of *metaclass* `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.

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.";
],
```

As mentioned above, `general` is a general-purpose attribute, free for the designer to use. The 'neatly cleaving' message can only happen once, because after that the mushroom object must have `general`. Note that the mushroom is allowed to call itself `self` instead of `mushroom`.

• **2**

```
Object medicine "guaranteed child-proof medicine bottle" cupboard
  with name "medicine" "bottle",
       description "~Antidote only: no preventative effect.~",
       openup
       [;  give self open ~locked; "The bottle cracks open!";
       ],
  has  container openable locked;
```

Any other code in the game can send the message `medicine.openup()` to crack open the bottle. For brevity, this solution assumes that the bottle is always visible to the player when it is opened – if not the printed message will be incongruous.

• **3** Briefly: provide a `GamePreRoutine` which tests to see if `second` is an object, rather than `nothing` or a number. If it is, check whether the object has a `second_before` rule (i.e. test the condition `(object provides second_before)`). If it has, send the `second_before` message to it, and return the reply as the return value from `GamePreRoutine`.

• **4** 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)
      "You aren't holding ", (the) noun, ".";
  rfalse;
];
```

• 5

```
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;
```

• 6 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` called `white_to` to be `n_to`, and then enter map directions in the source code using Mayan direction 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;
```

• 7

```
[ 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);
];
```

• 8 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.)

•9

```
Object -> 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:
                  "The bag wriggles hideously as it swallows ",
                  (the) noun, ".";
       ],
  has  container open;
```

•10

```
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;
Object -> power_button "power button"
  with name "power" "button" "switch",
       after
       [;  SwitchOn, SwitchOff: <<Examine screen>>;
       ],
  has  switchable;
Object -> 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.";
       ];
```

•11

```
Object -> glass_box "glass box with a lid"
  with name "glass" "box" "with" "lid"
  has  container transparent openable open;
Object -> steel_box "steel box with a lid"
  with name "steel" "box" "with" "lid"
  has  container openable open;
```

• **12** (The `describe` part of this answer but is only decoration.) Note the careful use of `inp1` and `inp2` rather than `noun` or `second`: see the note at the end of §9.

```
Object -> 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 -> -> "gold watch"
  with name "gold" "watch",
       description "The watch has no hands, oddly.",
       react_before
       [;  Listen: if (noun==0 or self) "The watch ticks loudly."; ];
```

• **13** The "plank breaking" rule is implemented here in its `door_to` routine. Note that this returns 'true' after killing the player.

```
Object -> 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 `doors` it ran into. If so, `door_to` could perhaps be modified to check that the `actor` is the `player`.

• 14

```
Object -> 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.",
       inside_description "You stare out through the bars.",
  has  enterable container openable open transparent static;
```

• 15 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.)^";
],
```

• 16 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>>;
```

• 17

```
Object -> 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)
                "You read the ", (string) x, " right through.";
            w = TryNumber(wn);
            if (w==-1000)
                "I was expecting a chapter number in the ",
                           (string) x, ".";
            "Chapter ", (number) w, " of the ", (string) x,
                  " is too sacred for you to understand now.";
    ];
```

• **18** Note that whether reacting before or after, the psychiatrist does not cut any actions short, because `react_before` and `react_after` both return false.

```
Object -> 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;
```

• **19** 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 * topic -> SayInstead;
Extend "tell"   replace * topic -> 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 of making the above directives) `Replace` the `TellSub` routine (see §21) by the `SayInsteadSub` one.

• **20** There are several ways to do this. The easiest is to add more grammar to the parser and let it do the hard work:

```
Object -> 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;
```

• **21** Obviously, a slightly wider repertoire of actions might be a good idea, but:

```
Object -> 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;
```

(The variable `i` isn't needed yet, but will be used by the code added in the answer to the next exercise.)

• **22** 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.";
```

• **23** 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.

```
Object -> 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: "~What,~ says Dan, ~ you want me to take ",
                     (the) noun, "?~";
           Drop: "~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;
```

• **24** 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.

• 25

```
[ PrintTime x; print (x/60), ":", (x%60)/10, (x%60)%10; ];
Object -> 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 ";
           "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 §27). 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.

• 26

```
Object -> 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;
```

• **27**

```
Object replicator "replicator"
  with name "replicator",
       grammar [;  return 'rc,'; ],
       orders
       [;  Give:
                if (noun in self)
                    "The replicator serves up a cup of ",
                    (name) noun, " which you drink eagerly.";
                "^That is not something I can replicate.^";
           default: "The replicator is unable to oblige.";
       ],
       life
       [;  Ask, Answer, Tell: "The replicator has no conversation skill.";
       ],
  has  talkable;
Object -> "Earl Grey tea"    with name "earl" "grey" "tea";
Object -> "Aldebaran brandy" with name "aldebaran" "brandy";
Object -> "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.

• **28** 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)
                    "^", (name) noun,
                        " is no longer aboard this demonstration game.^";
                "^", (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: objectloop (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.

• 29

```
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: "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) "^Standard by ", (number) noun,
                             " exceeds design tolerances.^";
               "^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.^";
           NotUnderstood: "^Language banks unable to decode.^";
           default: "^Information. That function is unavailable.^";
       ],
  has  talkable proper static;
Object -> force_wall "force wall"     with name "force" "wall" "shields";
Object -> 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.

• 30

```
[ 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`.

• 31 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;
Object -> ball "red ball" with name "red" "ball";
Object -> 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;
               "~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.

• **32** Just test if `HasLightSource(gift)==1`.

• **33** 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 = 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.";
   }
];
```

• **34** 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.

```
Object -> 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 (j ofclass Object && j hasnt door) n++;
           }
           if (n==0) rfalse;
           k=random(n); n=0;
           objectloop (i in compass)
           {   j=p.(i.door_dir);
               if (j ofclass Object && 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;
```

(Not forgetting to `StartDaemon(thief)` at some point, for instance in the game's `Initialise` routine.) 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).

• **35** We shall use a new property called `weight` and decide that any object which doesn't provide any particular weight will weigh 10 units. Clearly, an object which contains other objects will carry their weight too, so:

```
[ WeightOf obj t i;
   if (obj provides weight) t = obj.weight; else t = 10;
   objectloop (i in obj) t = t + WeightOf(i);
   return t;
];
```

Once every turn we shall check how much the player is carrying and adjust a measure of the player's fatigue accordingly. There are many ways we could choose to calculate this: for the sake of example we'll define two constants:

```
Constant CARRYING_STRENGTH = 500;
Constant HEAVINESS_THRESHOLD = 100;
```

Initially the player's strength will be the maximum possible, which we'll set to 500. Each turn the amount of weight being carried is substracted from this, but 100 is also added on (without exceeding the maximum value). So if the player carries more than 100 units, then her strength declines, but by dropping things to get the weight below 100 she can allow it to recover. If she drops absolutely everything, her entire strength will recuperate in at most 5 turns. Exhaustion sets in if her strength reaches 0, and at this point she is forced to drop something, which gives her strength a slight boost. Anyway, here's an implementation of all this:

```
Object weight_monitor
  with players_strength,
       warning_level 5,
       activate
       [;  self.players_strength = CARRYING_STRENGTH; StartDaemon(self);
       ],
       daemon
       [ w s b bw;
            if (location ~= Weights_Room) { StopDaemon(self); return; }
            s = self.players_strength
                - WeightOf(player) + HEAVINESS_THRESHOLD;
            if (s<0) s=0; if (s>CARRYING_STRENGTH) s=CARRYING_STRENGTH;
            self.players_strength = s;
            if (s==0)
```

```
        {   bw=-1;
            objectloop(b in player)
                if (WeightOf(b) > bw) { bw = WeightOf(b); w=b; }
            self.players_strength = self.players_strength + bw;
            print "^Exhausted with carrying so much, you decide
                to discard ", (the) w, ": "; <<Drop w>>;
        }
        w=s/100; if (w==self.warning_level) return;
        self.warning_level = w;
        switch(w)
        {   3: "^You are feeling a little tired.";
            2: "^You possessions are weighing you down.";
            1: "^Carrying so much weight is wearing you out.";
            0: "^You're nearly exhausted enough to drop everything
                 at an inconvenient moment.";
        }
   ];
```

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.) Finally, of course, at some point the weight monitor has to be sent an `activate` message to get things going.

• **36** See the next answer.

• **37**

```
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
       [;  if (location~=thedark) { self.number=0; StopDaemon(self); rtrue; }
           switch(++(self.number))
           {   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...";
         }
     ];
```

• **38** Either set a daemon to watch for `the_time` suddenly dropping, or put such a watch in the game's `TimePasses` routine.

• **39** 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;
  objectloop (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.

• **40** 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.

• **41** 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.

• **42** 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 §9 for the exact sequence of events when actions are processed.

• **43**

```
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.";
],
```

• **44** 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
       [;  Go, Look, Examine, Eat, Smell, Taste, Touch: rfalse;
           default: "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".

• **45**

```
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.";
],
```

• **46** 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`.

• **47** Put the following definition between inclusion of “Parser” and “Verblib”:

```
Object LibraryMessages
  with before
       [;  Prompt: if (turns==1)
                       print "What should you, the detective, do now?^>";
                   else
                       print "What next?^>";
                   rtrue;
       ];
```

• **48** See the *Inform Translator’s Manual*. One must provide a new grammar file (generating the same actions but from different syntax), tables showing how pronouns, possessives and articles work in the new language, a sheaf of translated library messages and so on. But it can be done.

• **49** 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"; } ];
```

• **50** 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"; ],
```

• **51** This answer is cheating, as it needs to know about the library’s `lookmode` variable (set to 1 for normal, 2 for verbose or 3 for superbrief). Simply include:

```
[ TimePasses;
  if (action~=##Look && lookmode==2) <Look>;
];
```

• 52

```
[ 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);
  ".";
];
```

• 53

```
Class Letter
  with list_together
       [;  if (inventory_stage==1)
           {   print "the letters ";
               if (~~(c_style & ENGLISH_BIT))   c_style = c_style + ENGLISH_BIT;
               if (~~(c_style & NOARTICLE_BIT)) c_style = c_style + NOARTICLE_BIT;
               if (c_style & NEWLINE_BIT)       c_style = c_style - NEWLINE_BIT;
               if (c_style & INDENT_BIT)        c_style = c_style - INDENT_BIT;
           }
           else print " from a Scrabble set";
       ],
       short_name
       [;  if (listing_together ofclass Letter) rfalse;
           print "letter ", (object) self, " from a Scrabble set"; rtrue;
       ],
       article "the";
```

and then as many letters as desired, along the lines of

```
Letter -> "X" with name "x";
```

• 54

```
Class  Coin
  with name "coin" "coins//p",
       description "A round unstamped disc, presumably local currency.",
       list_together "coins",
       plural
       [;  print (string) (self.&name)-->0;
           if (~~(listing_together ofclass Coin)) print " coins";
       ],
       short_name
       [;  if (listing_together ofclass Coin)
           {   print (string) (self.&name)-->0; rtrue; }
       ],
       article
       [;  if (listing_together ofclass Coin) print "one"; else print "a";
       ];
Class  Gold_coin   class Coin with name "gold";
Class  Silver_coin class Coin with name "silver";
Class  Bronze_coin class Coin with name "bronze";
SilverCoin -> "silver coin";
... and so on
```

• 55 Firstly, a printing rule to print the state of coins. Coin-objects will have a property called `way_up` which is always either 1 or 2:

```
[ Face x; if (x.way_up==1) print "Heads"; else print "Tails"; ];
```

There are two kinds of coin but we'll implement them with three classes: `Coin` and two subcategories, `GoldCoin` and `SilverCoin`. Since the coins only join up into trigrams when present in groups of three, we need a routine to detect this:

```
[ CoinsTogether cla i x y;
  objectloop (i ofclass cla)
  {   x=parent(i);
      if (y==0) y=x; else { if (x~=y) return 0; }
  }
  return y;
];
```

Thus `CoinsTogether(cla)` decides whether all objects of class `cla` are in the same place. (`cla` will always be either `GoldCoin` or `SilverCoin`.) We must now write the class definitions:

```
Class  Coin
  with name "coin" "coins//p",
       way_up 1, article "the",
       after
       [; Drop, PutOn:
             self.way_up = random(2); print (Face) self;
             if (CoinsTogether(self.which_class))
```

```
            {   print ". The ";
                if (self.which_class == GoldCoin)
                    print "gold"; else print "silver";
                " trigram is now ", (Trigram) self.which_class;
            }
            ".";
       ];
[ CoinLT k i c;
  if (inventory_stage==1)
  {   if (self.which_class == GoldCoin)
          print "the gold"; else print "the silver";
      print " coins ";
      k=CoinsTogether(self.which_class);
      if (k==location  k has supporter)
      {   objectloop (i ofclass self.which_class)
          {   print (name) i;
              switch(++c)
              {  1: print ", "; 2: print " and ";
                 3: print " (showing the trigram ",
                    (Trigram) self.which_class, ")";
              }
          }
          rtrue;
      }
      if (~~(c_style & ENGLISH_BIT))   c_style = c_style + ENGLISH_BIT;
      if (~~(c_style & NOARTICLE_BIT)) c_style = c_style + NOARTICLE_BIT;
      if (c_style & NEWLINE_BIT)       c_style = c_style - NEWLINE_BIT;
      if (c_style & INDENT_BIT)        c_style = c_style - INDENT_BIT;
  }
  rfalse;
];
Class  GoldCoin class Coin
  with name "gold", which_class GoldCoin,
       list_together [; return CoinLT(); ];
Class  SilverCoin class Coin
  with name "silver", which_class SilverCoin,
       list_together [; return CoinLT(); ];
```

(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 (change)" 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.) To resume: whenever coins are listed together, they are grouped into gold and silver. Whenever trigrams are visible they are to be described by either `Trigram(GoldClass)` or `Trigram(SilverClass)`:

```
Array gold_trigrams -->   "fortune" "change" "river flowing" "chance"
                           "immutability" "six stones in a circle"
                           "grace" "divine assistance";
Array silver_trigrams --> "happiness" "sadness" "ambition" "grief"
                           "glory" "charm" "sweetness of nature"
                           "the countenance of the Hooded Man";
[ Trigram cla i k state;
  objectloop (i ofclass cla)
  {   print (Face) i; if (k++<2) print ","; print " ";
      state=state*2 + (i.way_up-1);
  }
  if (cla == GoldCoin) i=gold_trigrams; else i=silver_trigrams;
  print "(", (string) i-->state, ")";
];
```

(These interpretations of the coins are quite bogus.) Finally, we have to make the six actual coins:

```
GoldCoin ->   "goat"    with name "goat";
GoldCoin ->   "deer"    with name "deer";
GoldCoin ->   "chicken" with name "chicken";
SilverCoin -> "robin"   with name "robin";
SilverCoin -> "snake"   with name "snake";
SilverCoin -> "bison"   with name "bison";
```

• 56

```
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;
],
```

• 57

```
Object -> "/?%?/ (the artiste formally known as Princess)"
  with name "princess" "artiste" "formally" "known" "as",
       short_name
       [;   if (self hasnt general) { print "Princess"; rtrue; }
       ],
       react_before
       [;  Listen: print_ret (name) self, " sings a soft siren song.";
       ],
```

```
       initial
       [;  print_ret (name) self, " is singing softly.";
       ],
       parse_name
       [ x n; 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 == '/')
            {   while (wn<=parse->1 && WordAddress(wn++)<x+5) n++;
                return n;
            }
            return -1;
       ],
       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;
```

• **58** 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.

```
Object -> drinksmat "drinks machine",
  with name "drinks" "machine",
       initial
          "A drinks machine here has buttons for Cola, Coffee and Tea.",
  has  static;
Object -> 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.";
    ];
```

• **59** 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;
];
```

• **60**

```
[ ParseNoun obj;
  if (NextWord() == 'object' && TryNumber(wn) == obj) return 2;
  wn--; return -1;
];
```

• **61**

```
[ ParseNoun;
  if (WordLength(wn)==1 && WordAddress(wn)->0 == '#') return 1;
  return -1;
];
```

• **62**

```
[ ParseNoun;
  if (WordLength(wn)==1 && WordAddress(wn)->0 == '#') return 1;
  if (WordLength(wn)==1 && WordAddress(wn)->0 == '*')
  {   parser_action = ##PluralFound; return 1; }
  return -1;
];
```

• **63** 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';
  }
];
```

• **64**

```
Global c_warned = false;
Class  Cherub
  with parse_name
       [ i j flag;
         for (flag=true:flag:flag=false)
         {   j=NextWord();
             if (j=='cherub' or j==self.name) flag=true;
             if (j=='cherubs' && (~~c_warned))
             {   c_warned=true;
                 parser_action=##PluralFound; flag=true;
 print "(I'll let this go once, but the plural of cherub is cherubim.)^";
             }
             if (j=='cherubim')
             {   parser_action=##PluralFound; flag=true; }
             i++;
         }
         return i-1;
       ];
```

Then again, Shakespeare even wrote "cherubins" in 'Twelfth Night', so who are we to censure?

• **65** Because the parser might go on to reject the line it's working on: for instance, if the player typed "shazam splurge" then the message "Shazam!" followed by a parser complaint will be somewhat unedifying.

• **66** The scheme will work like this: any room that ought to have a name should have a `place_name` property set to a dictionary word; say, the Bedquilt cave could be called `'bedquilt'`. Clearly you should only be allowed to type this from adjacent rooms. So we'll implement the following: you can only move by name to those rooms listed in the current room's `to_places` property. For instance, the Soft Room might have `to_places` set to

```
to_places Bedquilt Slab_Room Twopit_Room;
```

Now the code: if the player's verb is not otherwise understood, we'll check it to see if it's a place name of a nearby room, and if so store that room's object number in `goto_room`, converting the verb to `'go#room'` (which we'll deal with below).

```
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 ", (name) 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'.

• 67

```
Object -> genies_lamp "brass lamp"
  with name "brass" "lamp",
       before
       [; Rub: if (self hasnt general) give self general;
               else give self ~general;
              "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 wink.";
       ];
Object -> white_stone "white stone" with name "white" "stone";
Object -> 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;
       }
   }
];
```

• 68

```
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)
        "No footnote [", noun, "] has been mentioned.";
    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.";
```

• 69 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;
];
```

• 70 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;
];
```

• **71** 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 (ch~='-') { f=1; if (i~=0) return -1; } }
          else
          {   if (pp<MAX_PHONE_LENGTH)
                  dialled_number->(pp++)=ch-'0';
          }
      }
  } until (f==1  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;
```

• **72** 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;
];
```

• **73** 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;
```

• **74** (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); ];
```

• **75** (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.)

• 76

```
[ NeverMatch; return -1; ];
```

• 77 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).

• 78 (See the `NounDomain` specification in §A9.) 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;
];
```

• 79

```
Global scope_count;
[ PrintIt obj; print_ret ++scope_count, ": ", (a) obj, " (", obj, ")"; ];
[ ScopeSub; LoopOverScope(PrintIt);
  if (scope_count==0) "Nothing is in scope.";
];
Verb meta "scope" *                                -> Scope;
```

• 80

```
[ MegaExam obj; print "^", (a) obj, ": "; <Examine obj>; ];
[ MegaLookSub; <Look>; LoopOverScope(MegaExam); ];
Verb meta "megalook" *                             -> MegaLook;
```

• **81** 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)
  {   objectloop (i ofclass Object) 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'.

• **82** 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;
Window_Room window_w "West of Window"
  with far_side window_e;
Window_Room window_e "East of Window"
  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;
];
```

• **83** 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;
Object -> 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;
Object -> diamond "shiny diamond"
  with name "shiny" "diamond"
  has  scored;
Object -> 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.

• **84** 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;
```

•85

```
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;
```

•86 The label object itself is not too bad:

```
Object -> 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).

• **87** 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`.

• **88**

```
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;
];
```

• **89** 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.";
 }
```

• **90** 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.)

• **91** 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;
];
```

• **92** 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;
];
```

• **93** 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;
];
```

•94 `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;
];
```

•95 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.)

• **96** 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;
  }
];
```

Index