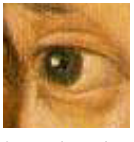
Chapter IV: Describing and Parsing

> Language disguises thought... The tacit conventions on which the understanding of everyday language depends are enormously complicated.
>
> — Ludwig Wittgenstein (1889–1951), *Tractatus*

§26 Describing objects and rooms



Talking about the state of the world is much easier than listening to the player's intentions for it. Despite this, the business of description takes up a fair part of this chapter 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.)

The simplest description of an object is its "short name". 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:

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

• **EXERCISE 62**

When referring to `animate` objects, you sometimes need to use pronouns such as "him". Define new printing routines so that `print "You throw the book at ", (PronounAcc) obj, "!";` will insert the right accusative pronouns.

△△ 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 object that doesn't exist.

. . . . .

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, either to some text or a routine to print some. Here are some possibilities, arranged as "article / name":

"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 taking no article, so the value of `article` is ignored. Objects representing named people usually have `proper`, and so might a book like "Whitaker's Almanac".

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`.

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". As mentioned above, the indefinite article becomes "some", and the player can use the pronoun "them" to refer to the object, so for instance "take them" to pick up the grapes-object.

△ 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.

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

. . . . .

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

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

For example, the dye might be defined with:

```
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.

• **EXERCISE 63**
Design a chessboard of sixty-four locations with a map corresponding to an eight-by-eight grid, so that White advances north towards Black, with pieces placed on the board according to a game position. Hint: using flexible routines for `short_name`, this can be done with just two objects, one representing all sixty-four squares, the other representing all thirty-two pieces.

. . . . .

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

```
>inventory
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 with a statement like:

```
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`, or a `supporter`, or is `transparent`, then 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 64**

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 the Inform library 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 emphasised in bold-face, then the `description`, followed by a list of the objects residing there which aren't `concealed` or `scenery`.

Chapter III 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.";
],
```

Unlike an `initial` description, this is still seen even if the pyramid has `moved`, i.e., been held by the player at some stage.

△ 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`.

△△ Here is exactly how a room description is printed. Recall from §21 that `location` holds the player's location if there is light, and `thedark` if not, and see §21 for the definition of "visibility ceiling", which roughly means the outermost thing the player can see: normally the location, but possibly a closed opaque container which the player is inside. First the top line:

(1a) A new-line is printed. The short name of the visibility ceiling is printed, using emphasised type, which on most models of computer appears as bold face.

(1b) If the player is on a `supporter`, then " (on ⟨something⟩)" is printed; if inside something (other than the visibility ceiling), 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)". (See §21 for details of `ChangePlayer`.)

(1d) A new-line is printed.

Now the long description. This step is sometimes skipped, depending on which "look mode" the player has chosen: in the normal mode, it is skipped if the player has just moved by a `Go` action into a location already visited; in "superbrief" mode it is always skipped, while in "verbose" mode it is never skipped.

(2a) Starting with the visibility ceiling and working down through "levels" of `enterable` objects containing the player, a long description is printed, as follows:
  (i) If the level is the current value of `location`, that is, the current room or else `thedark`, then: if `location` provides `describe` then the message `location.describe()` is sent. If not, then `location.description()` is sent. Every room is required to provide one or the other. (This is now redundant. In earlier Informs, `description` could only be a string, so `describe` was there in case a routine was needed.)
  (ii) If not, the level must be an `enterable` object `E`. If `E` provides it, the message `E.inside_description()` is sent.

(2b) After the description of each visibility level, the objects contained in that level are "listed" (see below).

The library has now finished, but your game gets a chance to add a postscript by means of an entry point routine:

(3) The entry point `LookRoutine` is called.

△△ Besides printing room descriptions, the `Look` action has side-effects: it can award the player some points, or mark a room with the attribute `visited`. For these rules in full, see §21.

△△ The `visited` attribute is only given to a room *after* its description has been printed for the first time. This is convenient for making the description different after the first time.

△△ When "listing objects" (as in 3a and 3b above) some objects are given a paragraph to themselves, while 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), because this has been taken care of in the short or long description already; and anything which has the attributes `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 provide this property *or* the object has `moved` and the property isn't `when_off` or `when_closed` *then* the object will be listed at the end, not given a paragraph of its own.

(4) Otherwise a new-line is printed and the property is printed (if it's a string) or run (if it's a routine). If it is a routine, it had better print something, as otherwise there will be a spurious blank line in the room description.

△ Note that although a `supporter` which is `scenery` won't be mentioned, anything on top of it may well be. If this is undesirable, set these objects on top to be `concealed`.

△ 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 §A3 for details, but 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 while remaining part of the same location, such as the stage of a theatre.

• △△**EXERCISE 65**

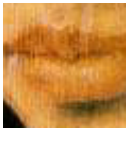
As mentioned above, the library implements "superbrief" and "verbose" modes for room description (one always omits long room descriptions, the other never does). How can verbose mode automatically print room descriptions every turn? (Some of the later Infocom games did this.)

• **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, owing to an extreme shortage of memory to store text – Scott Adams was obliged to write for machines with under 16K of free memory).

§27 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*



Listing objects tidily in a grammatical sentence is more difficult than it seems, especially when taking plurals and groups of similar objects into account. Here, for instance, is a list of 23 items printed by a room description in the demonstration 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, Punch magazine, a recent issue of the Spectator, a die and eight coins (four silver, one bronze and three gold) here.

Fortunately, the library's list-maker is available to the public by calling:

```
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 a "style" made by adding up some of the following:

`ALWAYS_BIT`	Always recurse downwards
`CONCEAL_BIT`	Misses out `concealed` or `scenery` objects
`DEFART_BIT`	Uses the definite article in list
`ENGLISH_BIT`	English sentence style, with commas and 'and'
`FULLINV_BIT`	Gives full inventory entry for each object
`INDENT_BIT`	Indents each entry according to depth
`ISARE_BIT`	Prints " is " or " are " before list
`NEWLINE_BIT`	Prints new-line after each entry
`NOARTICLE_BIT`	Prints no articles, definite or indefinite
`PARTINV_BIT`	Only brief inventory information after entry
`RECURSE_BIT`	Recurses downwards with usual rules
`TERSE_BIT`	More terse English style
`WORKFLAG_BIT`	At top level (only), only lists objects which have the `workflag` attribute

Recursing downwards means that if an object is listed, then its children are also listed, and so on for their children. The "usual rules" of `RECURSE_BIT` are that children are only listed if the parent is `transparent`, or a `supporter`, or a `container` which is `open` – which is the definition of "see-through" used throughout the Inform library. "Full inventory information" means listing objects exactly as if in an inventory, according to the rigmarole described in §26. "Brief inventory information" means listing as if in a room description: that is, noting whether objects are open, closed, empty or providing light (except that light is only mentioned when in a room which is normally dark).

The best way to decide which bits to set 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 for short periods 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 66**

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.

. . . . .

Lists are shorter, neater and more elegantly phrased if similar objects are grouped together. For instance, keys, books and torch batteries are all grouped together in lists printed by the game 'Curses'. To achieve this, the objects belonging to such a group (all the keys for instance) provide a common value of the property `list_together`. If this value is a number between 1 and 1000, the library will unobtrusively group the objects with that value together so that they appear consecutively in any list. For instance 'Curses' includes the following definitions:

```
Constant KEY_GROUP = 1;
...
Object -> -> brass_key "small brass key"
  with ...
       list_together KEY_GROUP;
```

and similarly for the other keys. Alternatively, instead of being a small number the common value can be a string such as `"foodstuffs"`. If so, then it must either be given in a class definition or else as a constant, like so:

```
Constant FOOD_GROUP = "foodstuffs";
```

(In particular, the actual text should only be written out in one place in the source code. Otherwise two or more different strings will be made, which just happen to have the same text as each other, and Inform will consider these to be different values of `list_together`.) Lists will then cite, for instance,

> three foodstuffs (a scarlet fish, some lembas wafer and an onion)

in running text, or

> three foodstuffs:
> a scarlet fish
> some lembas wafer
> an onion

in indented lists. This only happens when two or more are gathered together. Finally, the common value of `list_together` can be a routine, such as:

```
list_together [;
    if (inventory_stage == 1) {
        print "heaps of food, notably";
        if (c_style & INDENT_BIT == 0) print " ";
            else print " --^";
    } else if (c_style & INDENT_BIT == 0)
        print " (which only reminds you how hungry you are)";
],
```

Typically this might be part of a class definition from which all the objects in question inherit. Any `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 `true` from stage 1, signal the list-maker not to print a list at all. The above example would give a conversational sentence-like list as follows:

> heaps of food, notably a scarlet fish, some lembas wafer and an onion (which only reminds you how hungry you are)

and would also look suitable in sober tall-inventory-like columns:

> heaps of food, notably --
> a scarlet fish
> some lembas wafer
> an onion

△ A `list_together` routine has the opportunity to look through the objects which are about to be listed together, by looking at some of the list-maker's variables. `parser_one` holds the first object in the group and `parser_two` holds the depth of recursion in the list, which might be needed to keep the indentation straight. Applying `x = NextEntry(x,parser_two)` will move `x` on from one object to the next in the group being listed.

△ The library variable `listing_together` is set to the first object of a group being listed, when a group is being listed, or to `nothing` the rest of the time. This is useful because an object's `short_name` routine might need to behave differently during a grouped listing to the rest of the time.

• △**EXERCISE 67**
Implement the Scrabble pieces from the example list above.

• △△**EXERCISE 68**
Implement the three denominations of coin.

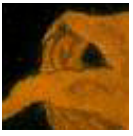
• △△**EXERCISE 69**
Implement the I Ching in the form of six coins, three gold (goat, deer and chicken), three silver (robin, snake and bison) which can be thrown to reveal gold and silver trigrams.

• △△**EXERCISE 70**
Design a class called `AlphaSorted`, members of which are always listed in alphabetical order. Although this *could* be done with `list_together`, this exercise is here to draw the reader's attention to an ugly but sometimes useful alternative. The only actions likely to produce lists are `Look`, `Search`, `Open` and `Inv`: so react to these actions by rearranging the object tree into alphabetical order before the list-writer gets going.

• **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. •Andreas Hoppler has written an alternative list-writing library extension called `"Lister.h"`, which defines a class `Lister` so that designers can customise their own listing engines for different purposes in a single game. •Anson Turner's example program `"52.inf"` models a deck of cards, and an extensive `list_together` sorts out hands of cards.

§28 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*



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". The `name` property holds an array of dictionary words, so that

```
(tomato.#name)/2 == 3
tomato.&name-->0 == 'fried'
tomato.&name-->1 == 'green'
tomato.&name-->2 == 'tomato'
```

(Recall that `X.#Y` tells you the number of `->` entries in such a property array, in this case six, so that `X.#Y/2` tells you the number of `-->` entries, in this case three.) You are quite free to alter this array during play:

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

The down side of this technique is that it's clumsy, when all's said and done, and not so very flexible, because you can't change the length of the `tomato.&name` array during play. Of course you *could* define the tomato

```
with name 'fried' 'green' 'tomato' 'blank.' 'blank.' 'blank.'
          'blank.' 'blank.' 'blank.' 'blank.' 'blank.' 'blank.'
          'blank.' 'blank.' 'blank.' 'blank.' 'blank.' 'blank.',
```

or something similar, giving yourself another (say) fifteen "slots" to put new names into, but this is inelegant even by Inform standards. Instead, an object like the tomato can be given a `parse_name` routine, allowing complete flexibility for the designer to specify just what names it does and doesn't match. It is time to begin looking into the parser and how it works.

. . . . .

The Inform parser has two cardinal principles: firstly, it is designed to be as "open-access" as possible, because a parser cannot ever be general enough for every game without being highly modifiable. This means that there are many levels on which you can augment or override what it does. Secondly, it tries to be generous in what it accepts from the player, understanding the broadest possible range of commands and making no effort to be strict in rejecting ungrammatical requests. For instance, given a shallow pool nearby, "examine shallow" has an adjective without a noun: but it's clear what the player means. In general, all sensible commands should be accepted but it is not important whether or not nonsensical ones are rejected.

The first thing the parser 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()` returns 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, then `NextWord()` returns 0 for 'not in the dictionary'. Inform creates the dictionary of a story file by taking all the `name` words of objects, all the verbs and prepositions from grammar lines, and all the words used in constants like `'frog'` written in the source code, and then sorting these into alphabetical order.

△ However, the story file's dictionary only has 9-character resolution. (And only 6 if Inform has been told to compile an early-model story file: see §45.) Thus the values of `'polyunsaturate'` and `'polyunsaturated'` are equal. Also, upper case and lower case letters are considered the same. Although dictionary words are permitted to contain numerals or typewriter symbols like `-`, `:` or `/`, these cost as much as two ordinary letters, so `'catch-22'` looks the same as `'catch-2'` or `'catch-207'`.

△△ A dictionary word can even contain spaces, full stops or commas, but if so it is 'untypeable'. For instance, `'in,out'` is an untypeable word because if the player were to type something like "go in,out", the text would be broken up into four words, `go` /

`in` / `,` / `out`. Thus `'in,out'` may be in the story file's dictionary but it will never match against any word of what the player typed. Surprisingly, this can be useful, as it was at the end of §18.

. . . . .

Since the story file's dictionary isn't always perfect, there is sometimes no alternative but to actually look at the player's text one character at a time: for instance, to check that a 12-digit phone number has been typed correctly and in full.

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. Given the above example text of "wizened man, eat the grey bread":

```
WordLength(4) == 3
WordAddress(4)->0 == 'e'
WordAddress(4)->1 == 'a'
WordAddress(4)->2 == 't'
```

because word number 4 is "eat". (Recall that the comma is considered as a word in its own right.)

△ The parser provides a basic routine for comparing a word against the texts `'0'`, `'1'`, `'2'`, ..., `'9999'`, `'10000'` or, in other words, against small numbers. This is the library routine `TryNumber(wordnum)`, which tries to parse the word at `wordnum` as a number and returns that number, if it finds a match. Besides numbers written out in digits, it also recognises the texts `'one'`, `'two'`, `'three'`, ..., `'twenty'`. If it fails to recognise the text as a number, it returns −1,000; if it finds a number greater than 10,000, it rounds down and returns 10,000.

. . . . .

To return to the naming of objects, the parser normally recognises any arrangement of some or all of the `name` words of an object as a noun which refers to it: and the more words, the better the match is considered to be. Thus "fried green tomato" is a better match than "fried tomato" or "green tomato" but all three are considered to match. On the other hand, so is "fried green", and "green green tomato green fried green" is considered a very good match indeed. The method is quick and good at understanding a wide variety of sensible texts, though poor at throwing out foolish ones. (An example of the parser's strategy of being generous rather than strict.) To be more precise, here is what happens when the parser wants to match some text against an object:

(1) If the object provides a `parse_name` routine, ask this routine to determine how good a match there is.
(2) If there was no `parse_name` routine, or if there was but it returned -1, ask the entry point routine `ParseNoun`, if the game has one, to make the decision.
(3) If there was no `ParseNoun` entry point, or if there was but it returned -1, look at the `name` of the object and match the longest possible sequence of words given in the `name`.

So: a `parse_name` routine, if provided, is expected to try to match as many words as possible starting from the current position of `wn` and 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, here is the fried tomato with which this section started:

```
parse_name [ n colour;
    if (self.ripe) colour = 'red'; else colour = 'green';
    while (NextWord() == 'tomato' or 'fried' or colour) n++;
    return n;
],
```

The effect of this is that if `tomato.ripe` is `true` then the tomato responds to the names "tomato", "fried" and "red", and otherwise to "tomato", "fried" and "green".

As a second example of how `parse_name` can be useful, suppose you define:

```
Object -> "fly in amber"
  with name 'fly' 'in' 'amber';
```

If the player then types "put fly in amber in hole", the parser will be thrown, because it will think "fly in amber in" is all just naming the object and then it won't know what the word "hole" is doing at the end. However:

```
Object -> "fly in amber"
  with parse_name [;
           if (NextWord() ~= 'fly' or 'amber') return 0;
           if (NextWord() == 'in' && NextWord() == 'amber')
               return 3;
           return 1;
       ];
```

Now the word "in" is only recognised as part of the fly's name if it is followed by the word "amber", and the ambiguity goes away. ("amber in amber" is also recognised, but then it's not worth the bother of excluding.)

△ `parse_name` is also used to spot plurals: see §29.

• **EXERCISE 71**
Rewrite the tomato's `parse_name` to insist that the adjectives must come before the noun, which must be present.

• **EXERCISE 72**
Create a musician called Princess who, when kissed, is transformed into "`/?%?/` (the artiste formerly known as Princess)".

• **EXERCISE 73**
Construct a drinks machine capable of serving cola, coffee or tea, using only one object for the buttons and one for the possible drinks.

• **EXERCISE 74**
Write a `parse_name` routine which looks through `name` in just the way that the parser would have done anyway if there hadn't been a `parse_name` in the first place.

• △**EXERCISE 75**
Some 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 76**
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 77**
How could the word "`#`" be made a wild-card, meaning "match any single object"?

• △△**EXERCISE 78**
And how could "`*`" be a wild-card for "match any collection of objects"? (Note: you need to have read §29 to answer this.)

• **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. •Miron Schmidt's library extension `"calyx_adjectives.h"`, based on earlier work by Andrew Clover, provides for objects to have "adnames" as well as "names": "adnames" are usually adjectives, and are regarded as being less good

matches for an object than "names". In this system "get string" would take either a string bag or a ball of string, but if both were present would take the ball of string, because "string" is in that case a noun rather than an adjective.

§29 Plural names for duplicated objects

A notorious challenge 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. 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 then similar silver and bronze coin classes)

```
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, and
(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, it's a little more subtle than this. What it groups together depends slightly on the context of the list being written. When it's writing a list which prints out details of which objects are providing light, for instance (as an inventory does), 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 ramifies further when the objects have a `parse_name` routine supplied. If they have different `parse_name` routines, the library decides that they are distinguishable. But if they have the same `parse_name` routine, for instance by inheriting it from a class definition, then the library has no alternative but to ask them. What happens is that:

(1) A variable called `parser_action` is set to the special value `##TheSame`, a value it never has at any other time;
(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.

△△ You may even want to provide a `parse_name` routine for objects which otherwise don't need one, just to speed up the process of the library telling if two objects are distinguishable – if there were 30 gold coins in one place the parser would be doing a lot of work comparing names, but you can make the decision much faster.

• △△**EXERCISE 79**
Perhaps the neatest trick of parsing in any Infocom game occurs in 'Spellbreaker', which has a set of white cubes which are indistinguishable until the player writes words onto them with a magic burin (a medieval kind of pen), after which it's possible to tell them apart. Imitate this in Inform.

. . . . .

Secondly, the player talking to the computer. Suppose a game involves collecting a number of similar items, such as 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 the `name` lists of the crown objects is not quite right, because the parser will still think that "crowns" might refer to a single specific item. Instead, put in the word `'crowns//p'`. The suffix `//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 is a single game object; you should give that object the `pluralname` attribute instead, as in §26 back at the start of this chapter.) For example the `GoldCoin` class would read:

```
Class GoldCoin
 with name 'gold' 'coin' 'coins//p',
      short_name "gold coin",
      plural "gold coins";
```

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 special value. 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;
           }
       }
      ];
```

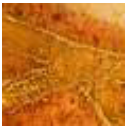
This code assumes that the crown objects have just one `name` each, their colours.

• **EXERCISE 80**
Write a 'cherub' class so that if the player tries to call them "cherubs", a message like "I'll let this go once, but the plural of cherub is cherubim" appears.

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

§30 How verbs are parsed

> ". . .I can see that the grammar gets tucked into the tales and poetry as one gives pills in jelly."
>
> — Louisa May Alcott (1832–1888), *Little Women*



Here is how the parser reads in a whole command. Given a stream of text like

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

it first breaks it into words, as shown, and then calls the entry point routine `BeforeParsing` (which you can provide, if you want to, in order to meddle with the text stream before parsing gets underway). The parser 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 matching an `animate` or `talkable` object: in this case St Peter. This person is called the "actor", since he or she is going to perform the action, and is most often the player (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 variables `actor` and `verb_word` while working. (In the example above, their values would be the St Peter object and `'take'`, respectively.)

△ This brief discussion is simplified in two ways. Firstly, it leaves out directions, because Inform considers that the name of a direction-object implies "go": thus "north" means "go north". Secondly, it misses out the `grammar` property described in §18, which can cause different actors to recognise different grammars.

• △**EXERCISE 81**

Use `BeforeParsing` to implement a lamp which, when rubbed, produces a genie who casts a spell to make the player confuse the words "white" and "black".

. . . . .

This section is about verbs, which are defined with "grammar", meaning usages of the directives `Verb` and `Extend`. The library contains a substantial amount of grammar as it is, and this forms (most of) the library file `"Grammar.h"`. Grammar defined in your own code can either build on this or selectively knock it down, but either way it should be made *after* the inclusion of `"Grammar.h"`.

For instance, making a new synonym for an existing verb is easy:

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

Now "steal", "acquire" and "grab" are synonyms for "take".

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

. . . . .

To return to the text above, the parser has now recognised the English word "take" as one of those which can refer to a particular Inform verb. It has reached word 5 and still has "the keys from paul" left to understand.

Every Inform verb has a "grammar" which consists of a list of one or more "grammar lines", each of them 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 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. Continuing the example, the parser accepts the line of three tokens

⟨one or more nouns⟩ ⟨the word `from`⟩ ⟨a noun⟩

as matching "the keys from paul".

Every grammar line has the name of an action attached, and in this case it is `Remove`: so the parser has ground up the original text into just four quantities, ending up with

```
actor = StPeter  action = Remove  noun = gold_keys  second = StPaul
```

The parser's job is now complete, and the rest of the Inform library can get on with processing the action or, as in this case, an order being addressed to somebody other than the player.

△ The action for the line 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.h"` 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 §7.) 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`).

△ Some verbs are marked as `meta` – these are the verbs leading to Group 1 actions, those which are not really part of the game's world: for example, "save", "score" and "quit". For example:

```
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 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 or 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 quite often need to alter these, usually to add extra lines of grammar but sometimes to remove existing ones. For instance, suppose you would like "drop charges" to be a command in a detection game (or a naval warfare game). This means adding a new grammar line to the "drop" verb. The Extend directive is provided for exactly this purpose:

```
Extend 'drop' * 'charges' -> DropCharges;
```

Normally, extra lines of grammar are added at the bottom of those already there, so that this will be the very last grammar line tested by the parser. This may not be what you want. For instance, "take" has a grammar line reading

```
    * multi -> Take
```

quite early on. So if you want to add a grammar line diverting "take ⟨food⟩" to a different action, like so:

```
    * edible -> Eat
```

(`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. What you need is for the new line to go in at the top, not the bottom:

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

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

```
Extend 'press' replace
    * 'charges' -> PressCharges;
```

and now the verb "press" has no other sense but this, and can't be used in the sense of pressing down on objects any more, because those grammar lines are gone. To sum up, `Extend` can optionally take one of three keywords:

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

with `last` being the default.

△ 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 'run' * 'program' -> Compute;
```

makes "run" behave exactly like "go" and "walk", as these three words are ordinarily synonymous to the library, except that it also recognises "program", so that "run program" activates a computer but "walk program" 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 change to one will not also change the other.

△△ Occasionally verb definition commands are not enough. For example, in the original 'Advent', the player could type the name of an adjacent place which had previously been visited, and be taken there. (This feature isn't included in the Inform example version of the game in order to keep the source code as simple as possible.) There are several laborious ways to code this, but here's a concise way. The library calls the `UnknownVerb` entry point 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 one way the 'Advent' room-naming might work. Suppose that every room has been given a property called `go_verb` listing the words which refer to it, so for instance the well house might be defined along these lines:

```
AboveGround Inside_Building "Inside Building"
  with description
           "You are inside a building, a well house for a
           large spring.",
       go_verb 'well' 'house' 'inside' 'building',
...
```

The `UnknownVerb` routine then looks through the possible compass directions for already-visited rooms, checking against words stored in this new property:

```
Global go_verb_direction;
[ UnknownVerb word room direction adjacent;
  room = real_location;
  objectloop (direction in compass) {
      adjacent = room.(direction.door_dir);
      if (adjacent ofclass Object && adjacent has visited
          && adjacent provides go_verb
          && WordInProperty(word, adjacent, go_verb)) {
              go_verb_direction = direction;
              return 'go.verb';
      }
  }
  if (room provides go_verb
      && WordInProperty(word, room, go_verb)) {
      go_verb_direction = "You're already there!";
      return 'go.verb';
  }
  objectloop (room provides go_verb && room has visited
              && WordInProperty(word, room, go_verb)) {
      go_verb_direction = "You can't get there from here!";
      return 'go.verb';
  }
```

```
    objectloop (room provides go_verb && room hasnt visited
                && WordInProperty(word, room, go_verb)) {
        go_verb_direction = "But you don't know the way there!";
        return 'go.verb';
    }
    rfalse;
];
```

When successful, this routine stores either a compass direction (an object belonging to the `compass`) in the variable `go_verb_direction`, or else a string to print. (Note that an `UnknownVerb` routine shouldn't print anything itself, as this might be inappropriate in view of subsequent parsing, or if the actor isn't the player.) The routine then tells the parser to treat the verb as if it were `'go.verb'`, and as this doesn't exist yet, we must define it:

```
[ Go_VerbSub;
    if (go_verb_direction ofclass String)
        print_ret (string) go_verb_direction;
    <<Go go_verb_direction>>;
];
Verb 'go.verb' * -> Go_Verb;
```

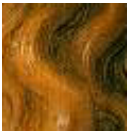
• △△**EXERCISE 82**

A minor deficiency with the above system is that the parser may print out strange responses like "I only understood you as far as wanting to go.verb." if the player types something odd like "bedquilt the nugget". How can we ensure that the parser will always say something like "I only understood you as far as wanting to go to Bedquilt."?

• **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. •Irene Callaci's `"AskTellOrder.h"` library extension file makes an elegant use of `BeforeParsing` to convert commands in the form "ask mr darcy to dance" or "tell jack to go north" to Inform's preferred form "mr darcy, dance" and "jack, go north".

§31 Tokens of grammar



The complete list of grammar tokens is given in the table below. These tokens are all described in this section except for `scope = ⟨Routine⟩`, which is postponed to the next.

`'⟨word⟩'`	that literal word only
`noun`	any object in scope
`held`	object held by the actor
`multi`	one or more objects in scope
`multiheld`	one or more held objects
`multiexcept`	one or more in scope, except the other object
`multiinside`	one or more in scope, inside the other object
`⟨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

To recap, the parser goes through a line of grammar tokens trying to match each against some text from the player's input. Each token that matches must produce one of the following five results:

(a) a single object;
(b) a "multiple object", that is, a set of objects;
(c) a number;
(d) a "consultation topic", that is, a collection of words left unparsed to be looked through later;
(e) no information at all.

Ordinarily, a single line, though it may contain many tokens, can produce at most two substantial results ((a) to (d)), at most one of which can be multiple

(b). (See the exercises below if this is a problem.) For instance, suppose the text "green apple on the table" is parsed against the grammar line:

```
* multi 'on' noun -> Insert
```

The `multi` token matches "green apple" (result: a single object, since although `multi` can match a multiple object, it doesn't have to), `'on'` matches "on" (result: nothing) and the second `noun` token matches "the table" (result: a single object again). There are two substantial results, both objects, so the action that comes out is `<Insert apple table>`. If the text had been "all the fruit on the table", the `multi` token might have resulted in a list: perhaps of an apple, an orange and a pear. The parser would then have generated and run through three actions in turn: `<Insert apple table>`, then `<Insert orange table>` and finally `<Insert pear table>`, printing out the name of each item and a colon before running the action:

```
>put all the fruit on the table
Cox's pippin: Done.
orange: Done.
Conference pear: Done.
```

The library's routine `InsertSub`, which actually handles the action, only deals with single objects at a time, and in each case it printed "Done."

. . . . .

`'⟨word⟩'` This matches only the literal word given, sometimes called a preposition because it usually is one, and produces no resulting information. (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: for instance "in", "into" and "inside" are often synonymous. As a convenient shorthand, then, you can write a series of prepositions (only) with slashes / in between, to mean "one of these words". For example:

```
* noun 'in'/'into'/'inside' noun -> Insert
```

`noun` Matches any single object "in scope", a term defined in the next section and which roughly means "visible to the player at the moment".

`held` Matches any single object which is an immediate possession of the actor. (Thus, if a key is inside a box being carried by the actor, the box might match but the key cannot.) This is convenient for two reasons. Firstly,

many actions, such as `Eat` or `Wear`, only sensibly apply to things being held. 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", and happens only for the player, not for other actors.

`multi` Matches one or more objects in scope. 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" in the process. For instance, "all the apples" and "the duck and the drake" could match a `multi` token but not a `noun` token.

`multiexcept` Matches one or more objects in scope, except that it does not match the other single object parsed in the same grammar line. This is provided to make commands like "put everything in the rucksack" come out right: the "everything" is matched by all of the player's possessions except the rucksack, which stops the parser from generating an action to put the rucksack inside itself.

`multiinside` Similarly, this matches anything inside the other single object parsed on the same grammar line, which is good for parsing commands like "remove everything from the cupboard".

`⟨attribute⟩` Matches any object in scope which has the given attribute. This is useful for sorting out actions according to context, and perhaps the ultimate example might be an old-fashioned "use" verb:

```
Verb 'use' 'employ' 'utilise'
    * edible    -> Eat
    * clothing  -> Wear
    ...
    * enterable -> Enter;
```

`creature` Matches any object in scope which behaves as if living. This normally means having `animate`: but, as an exceptional rule, if the action on the grammar line is `Ask`, `Answer`, `Tell` or `AskFor` then having `talkable` is also acceptable.

`noun = ⟨Routine⟩` "Any single object in scope satisfying some condition". 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 source 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. You can also choose whether or not it can accept a multiple object. See §32.

`number` Matches any decimal number from 0 upwards (though it rounds off large numbers to 10,000), 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. (While `inp1` is set to 1, indicating that this "first input" is intended as a number: if the noun had been the object which happened to have number 504, then `inp1` would have been set to this object, the same as `noun`.) If you need more exact number parsing, without rounding off, and including negative numbers, see the exercise below.

• **EXERCISE 83**
Some games, such as David M. Baggett's game 'The Legend Lives!' 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 false 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 `false` 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, because 0 is the same as `false`. 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.)

`topic` This token matches as much text as possible, regardless of what it says, producing no result. As much text as possible means "until the end of the typing, or, if the next token is a preposition, until that preposition is reached". The only way this can fail is if it finds no text at all. Otherwise, the variable `consult_from` is set to the number of the first word of the matched text and `consult_words` to the number of words. See §16 and §18 for examples of topics being used.

⟨Routine⟩ The most flexible token is simply the name of a "general parsing routine". As the name suggests, it is a routine to do some parsing which can have any outcome you choose, and many of the interesting things you can do with the parser involve writing one. A general parsing routine looks at the word stream using `NextWord` and `wn` (see §28) to make its decisions, and should return one of the following. Note that the values beginning `GPR_` are constants defined by the library.

`GPR_FAIL`	if there is no match;
`GPR_MULTIPLE`	if the result is a multiple object;
`GPR_NUMBER`	if the result is a number;
`GPR_PREPOSITION`	if there is a match but no result;

GPR_REPARSE	to reparse the whole command from scratch; or
O	if the result is a single object O.

On an unsuccessful match, returning GPR_FAIL, 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;
  if (NextWord() == 'on' or 'at' or 'in') return GPR_PREPOSITION;
  return GPR_FAIL;
];
```

duplicates the effect of `'on'/'at'/'in'`, that is, it makes a token which accepts any of the words "on", "at" or "in" as prepositions. Similarly,

```
[ Anything;
  while (NextWordStopped() ~= -1) ; return GPR_PREPOSITION;
];
```

accepts the entire rest of the line (even an empty text, if there are no more words on the line), ignoring it. NextWordStopped is a form of NextWord which returns the special value -1 once the original word stream has run out.

If you return GPR_NUMBER, the number which you want to be the result should be put into the library's variable parsed_number.

If you return GPR_MULTIPLE, place your chosen objects in the table multiple_object: that is, place the number of objects in multiple_object-->0 and the objects themselves in -->1, ...

The value GPR_REPARSE should only be returned if you have actually altered the text you were supposed to be parsing. This is a feature used internally by the parser when it asks "Which do you mean ...?" questions, and you can use it too, but be wary of loops in which the parser eternally changes and reparses the same text.

. . . . .

△ To parse a token, the parser uses a routine called ParseToken. This behaves almost exactly like a general parsing routine, and returns the same range of values. For instance,

```
ParseToken(ELEMENTARY_TT, NUMBER_TOKEN)
```

parses exactly as `number` does: similarly for `NOUN_TOKEN`, `HELD_TOKEN`, `MULTI_TOKEN`, `MULTIHELD_TOKEN`, `MULTIEXCEPT_TOKEN`, `MULTIINSIDE_TOKEN` and `CREATURE_TOKEN`. The call

```
ParseToken(SCOPE_TT, MyRoutine)
```

does what `scope=MyRoutine` does. In fact `ParseToken` can parse any kind of token, but these are the only cases which are both useful enough to mention and safe enough to use. It means you can conveniently write a token which matches, say, *either* the word "kit" *or* any named set of items in scope:

```
[ KitOrStuff; if (NextWord() == 'kit') return GPR_PREPOSITION;
  wn--; return ParseToken(ELEMENTARY_TT, MULTI_TOKEN);
];
```

. . . . .

• **EXERCISE 84**
Write a token to detect small numbers in French, "un" to "cinq".

• **EXERCISE 85**
Write a token called `Team`, which matches only against the word "team" and results in a multiple object containing each member of a team of adventurers in a game.

• △**EXERCISE 86**
Write a token to detect non-negative floating-point numbers like "21", "5.4623", "two point oh eight" or "0.01", rounding off to two decimal places.

• △**EXERCISE 87**
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 88**
(Adapted from code in `"timewait.h"`: see the references below.) 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 89**
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.

• △**EXERCISE 90**
Write a general parsing routine accepting any amount of text, including spaces, full stops and commas, between double-quotes as a single token.

• △**EXERCISE 91**
On the face of it, the parser only allows two parameters to an action, `noun` and `second`. Write a general parsing routine to accept a `third`. (This is easier than it looks: see the specification of the `NounDomain` library routine in §A3.)

• **EXERCISE 92**
Write a token to match any legal Inform decimal, binary or hexadecimal constant (such as `-321`, `$4a7` or `$$1011001`), producing the correct numerical value in all cases, while not matching any number which overflows or underflows the legal Inform range of −32,768 to 32,767.

• **EXERCISE 93**
Add the ability to match the names of the built-in Inform constants `true`, `false`, `nothing` and `NULL`.

• **EXERCISE 94**
Now add the ability to match character constants like `'7'`, producing the correct character value (in this case 55, the ZSCII value for the character '7').

• △△**EXERCISE 95**
Next add the ability to match the names of attributes, such as `edible`, or negated attributes with a tilde in front, such as `~edible`. An ordinary attribute should parse to its number, a negated one should parse to its number plus 100. (Hint: the library has a printing rule called `DebugAttribute` which prints the name of an attribute.)

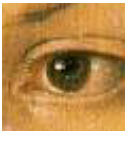
• △△**EXERCISE 96**
And now add the names of properties.

• **REFERENCES**

Once upon a time, Andrew Clover wrote a neat library extension called `"timewait.h"` for parsing times of day, and allowing commands such as ‘‘wait until quarter to three’’. L. Ross Raszewski, Nicholas Daley and Kevin Forchione each tinkered with and modernised this, so that there are now also `"waittime.h"` and `"timesys.h"`. Each has its merits.

§32 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



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 experimentally by compiling any game with the debugging suite of verbs included (see §7) and typing "scope" in interesting places. "In scope" roughly means "the compass directions, what you're carrying and what you can see". It exactly means this:

(1) the compass directions;
(2) the player's immediate possessions;
(3) if there is light, then the contents of the player's visibility ceiling (see §21 for definition, but roughly speaking the outermost object containing the player which remains visible, which is usually the player's location);
(4) if there is darkness, then the contents of the library's object `thedark` (by default there are no such contents, but some designers have been known to move objects into `thedark`: see 'Ruins');
(5) if the player is inside a `container`, then that container;
(6) if *O* is in scope and is see-through (see §21), then the contents of *O*;
(7) if *O* is in scope, then any object which it "adds to scope" (see below).

with the proviso that the `InScope` entry point (see below) can add to or replace these rules, if you write one.

It's significant that rule (3) doesn't just say "whatever is in the current location". For instance, if the player is in a closed cupboard in the Stores Room, then rule (3) means that the contents of the cupboard are in scope, but other items in the Stores Room are not.

Even in darkness the player's possessions are in scope, so the player can still turn on a lamp being carried. 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 as well as directions, and they respond to names like "the south wall" and "the floor" as well as "south" and "down".

△ The `concealed` attribute only hides objects from room descriptions, and doesn't remove them from scope. If you want things to be unreferrable-to, put them somewhere else!

△ The small print: 1. For "player", read "actor". Thus "dwarf, drop sword" will be accepted if the dwarf can see the sword even if the player can't. 2. Scope varies depending on the token being parsed: for the `multi-` tokens, compass directions are not in scope; for `multiexcept` the other object isn't in scope; for `multiinside` only the contents of the other object are in scope.

. . . . .

Two library routines 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, and if it's omitted, scope is worked out for the player as usual.

• **EXERCISE 97**
Implement the debugging suite's "scope" verb, which lists all the objects currently in scope.

• **EXERCISE 98**
Write a "megalook" verb, which looks around and examines everything in scope except the walls, floor and ceiling.

. . . . .

Formally, scope determines what you can talk about, which usually means what you can see or hear. 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, such as `Examine` or `LookUnder`, don't require touchability. But most actions are tactile, and so are 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` and prints nothing 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`, printing 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:

```
[ QuerySub; noun.description(); return;
];
[ QueryTopic;
  switch (scope_stage) {
      1: rfalse;
      2: ScopeWithin(questions); rtrue;
      3: "At the moment, even the simplest questions confuse you.";
  }
];
Object questions;
```

where the actual questions at any time are the current children of the `questions` object, like so:

```
Object -> "long count"
  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.";
```

(which might be helpful in 'Ruins') and we also have a grammar line:

```
Verb 'what' * 'is'/'was' scope=QueryTopic -> Query;
```

The individual questions have short names 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". (As it stands this won't recognise "what is the count?". Conventionally players are supposed not to type question marks or quotes in their commands. To allow this one could always add `'count?'` as one of the names above.)

. . . . .

Here is the specification. When the parser reaches `scope=Whatever`, it calls the `Whatever` routine with the variable `scope_stage` set to 1. The routine should return `true` if it is prepared to allow multiple objects to be accepted here, and `false` otherwise. (In the example, as we don't want "what is everything" to list all the questions and answers in the game, we return `false`.)

A little later the parser calls `Whatever` again with `scope_stage` now set to 2. `Whatever` is then obliged to tell the parser which objects are to be in scope. It can call two parser routines to do this:

```
ScopeWithin(obj)
```

which puts everything inside `obj`, but not `obj` itself, into scope, and then works through rules (6) and (7) above, so that it may continue to add the contents of the contents, and so on; and

```
PlaceInScope(obj)
```

which puts just `obj` into scope. It is perfectly legal to declare something in scope that "would have been in scope anyway": or something which is in a different room altogether from the actor concerned, say at the other end of a telephone line. The scope routine `Whatever` should then return `false` if the nominated items are additional to the usual scope, or `true` if they are the only items in scope. (In the example, `QueryTopic` returns `true` because it wants the only items in scope to be its question topics, not the usual miscellany near the player.)

This is fine if the token is correctly parsed. If not, the parser may choose to ask the token routine to print a suitable error message, by calling `Whatever` with `scope_stage` set to 3. (In the example, this will happen if the player types "what is the lgon count", and `QueryTopic` replies "At the moment, even the simplest questions confuse you.", because it comes from a faintly dream-like game called 'Balances'.)

• **EXERCISE 99**

Write a token which puts everything in scope, so that you could have a debugging "purloin" which could take anything, regardless of where it was and the rules applying to it.

. . . . .

△ The global scope rules can be tampered with by providing an entry point routine called `InScope(actor)`, where `actor` is the actor whose scope is worked out. In effect, this defines the "ordinary" scope by making calls to `ScopeWithin` and `PlaceInScope`, then returning `true` or `false`, exactly as if it were a scope token at stage 2. For instance, here as promised is 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 in parent(player))
          if (i has moved) PlaceInScope(i);
  rfalse;
];
```

With this routine added, objects near the player in a dark room 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, not to anyone else.

△ The token `scope=`⟨Routine⟩ takes precedence over `InScope`, which will only be reached if the routine returns `false` to signify "carry on".

△△ There are seven reasons why `InScope` might be being called; the `scope_reason` variable is set to the current one:

`PARSING_REASON`	The usual reason. 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 §18 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`.

• △△**EXERCISE 100**

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 possible to look through; objects on the far side should be in scope, but not manipulable.

• △△**EXERCISE 101**

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 the distinctive sound of a dwarf breathing. If you tell this dwarf to turn the light on, he will.

. . . . .

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 add any object `x` to 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)`.

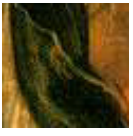
• **EXERCISE 102**
(From the tiny example game 'A Nasal Twinge'.) Give the player a nose, which is always in scope and can be held, reducing the player's carrying capacity.

• **EXERCISE 103**
(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 104**
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. •Jesse Burneko's library extension `"Info.h"` is a helpful model to follow: using a simple scope token, it allows for "consult" and "ask" commands to access topics which are provided as objects. •See also the exercises at the end of §18 for further scope trickery. •Similarly, L. Ross Raszewski's `"whatis.h"` (adapted to Inform 6 by Andrew C. Murie) and David Cornelson's `"whowhat.h"` field questions such as "what is. . ." and "who is. . .".

§33 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 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. By the time the parser wants to print an error, every one of the grammar lines in a verb will have failed. The error message chosen it prints is the most "interesting" one: meaning, lowest down this list.

If a general parsing routine you have written returns `GPR_FAIL`, then the grammar line containing it normally ends in plain `STUCK_PE`, the least interesting of all errors (unless you did something like calling the library's `ParseToken` routine before giving up, which might have set a more interesting error like `CANTSEE_PE`). But you can choose to create a new error and put it in the parser's variable `etype`, as in the following example:

```
[ Degrees d;
  d = TryNumber(wn++);
```

```
    if (d == -1000) return GPR_FAIL;
    if (d <= 360) { parsed_number = d; return GPR_NUMBER; }
    etype = "There are only 360 degrees in a circle.";
    return GPR_FAIL;
];
```

This parses a number of degrees between 0 and 360. Although `etype` normally only holds values like `VERB_PE`, which are numbers lower than 100, here we've set it equal to a string. As this will be a value that the parser doesn't recognise, we need to write a `ParserError` routine that will take care of it, by reacting to a string in the obvious way – printing it out.

```
[ ParserError error_type;
    if (error_type ofclass String) print_ret (string) error_type;
    rfalse;
];
```

This will result in conversation like so:

```
>steer down
I didn't understand that sentence.
>steer 385
There are only 360 degrees in a circle.
```

In the first case, `Degrees` failed without setting any special error message on finding that the second word wasn't a number; in the second case it gave the new, specific error message.

. . . . .

The `VAGUE_PE` and `ITGONE_PE` errors apply to all pronouns (in English, "it", "him", "her" and "them"). The variable `vague_word` contains the dictionary address of whichever pronoun is involved (`'it'`, `'him'` and so on).

You can find out the current setting of a pronoun using the library's `PronounValue` routine: for instance, `PronounValue('it')` gives the object which "it" currently refers to, possibly `nothing`. Similarly `SetPronoun('it', magic_ruby)` would set "it" to mean the magic ruby object. You might want this because, 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 object names with a pragmatic algorithm which has evolved over the years (see below). Experience also shows that no two people ever quite agree on what the parser should "naturally" do. Designers have an opportunity to influence this by providing an entry point routine called `ChooseObjects`:

```
ChooseObjects(obj, code)
```

is called in two circumstances. If `code` is `false` or `true`, the parser is considering including the given `obj` in an "all": `false` means the parser has decided against, `true` means it has decided in favour. The routine should reply

0 to accept the parser's decision;
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 choosing between a list of items which made equally good matches against some text, and would like a hint. `ChooseObjects` should then return a number from 0 to 9 to give `obj` 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 105**
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`.)

• △**EXERCISE 106**
Joyce Haslam's Inform edition of the classic Acornsoft game 'Gateway to Karos' requires a class called `FaintlyLitRoom` for rooms so dimly illuminated that "take all" is impossible. How might this work?

. . . . .

△△ Suppose we have a set of objects which have all matched equally well against the textual input, so that some knowledge of the game world is needed to resolve which of the objects – possibly only one, possibly more – is or are intended. Deciding this is called "disambiguation", and here in full are the rules used by library 6/10 to do it. The reader is cautioned that after six years, these rules are still evolving.

(1) Call an object "good" according to a rule depending on what kind of token is being matched:

`held`	Good if its parent is the actor.
`multiheld`	Good if its parent is the actor.
`multiexcept`	Good if not also the second object, if that's known yet.
`multiinside`	Good if not inside the second object, if that's known yet.
`creature`	Good if `animate`, or if the proposed action is `Ask`, `Answer`, `Tell` or `AskFor` and the object is `talkable`.
other tokens	All objects are good.

If only a single object is good, this is immediately chosen.

(2) If the token is `creature` and no objects are good, fail the token altogether, as no choice can make sense.

(3) Objects which don't fit "descriptors" used by the player are removed:
if "my", an object whose parent isn't the actor is discarded;
if "that", an object whose parent isn't the actor's location is discarded;
if "lit", an object which hasn't `light` is discarded;
if "unlit", an object which has `light` is discarded;
if "his" or some similar possessive pronoun, an object not owned by the person implied is discarded.
Thus "his lit torches" will invoke two of these rules at once.

(4) If there are no objects left, fail the token, as no choice can make sense.

(5) It is now certain that the token will not fail. The remaining objects are assigned a score as follows:
(i) $1000 \times C$ points, where C is the return value of `ChooseObjects(object,2)`. ($0 \leq C \leq 9$. If the designer doesn't provide this entry point at all then $C = 0$.)
(ii) 500 points for being "good" (see (1) above).
(iii) 100 points for not having `concealed`.

(iv) P points depending on the object's position:

$$P = \begin{cases} A & \text{if object belongs to the actor,} \\ L & \text{if object belongs to the actor's visibility ceiling,} \\ 20 & \text{if object belongs anywhere else except the compass,} \\ 0 & \text{if object belongs to the compass.} \end{cases}$$

(Recall that "visibility ceiling" usually means "location" and that the objects belonging to the compass are exactly the compass directions.) The values A and L depend on the token being parsed:

$$\begin{cases} A = 60 \quad L = 40 & \text{for } \boxed{\texttt{held}} \text{ or } \boxed{\texttt{multiheld}} \text{ tokens,} \\ A = 40 \quad L = 60 & \text{otherwise.} \end{cases}$$

(v) 10 points for not having `scenery`.
(vi) 5 points for not being the actor object.
(vii) 1 point if the object's gender, number and animation (GNA) matches one possibility implied by some pronoun typed by the player: for instance "them" in English implying plural, or "le" in French implying masculine singular.

(6d) In "definite mode", such as if the player has typed a definite article like "the", if any single object has highest score, choose that object.

(7ip) The following rule applies only in indefinite mode and provided the player has typed something definitely implying a plural, such as the words "all" or "three" or "coins". Here the parser already has a target number of objects to choose: for instance 3 for "three", or the special value of 100, meaning "an unlimited number", for "all" or "coins".
Go through the list of objects in "best guess" order (see below). Mark each as "accept" unless:
(i) it has `worn` or `concealed`;
(ii) or the action is Take or Remove and the object is held by the actor;
(iii) or the token is `multiheld` or `multiexcept` and the object isn't held by the actor;
(iv) or the target number is "unlimited" and $S/20$ (rounded down to the nearest integer) has fallen below its maximum value, where S is the score of the object.
The entry point `ChooseObjects(object,accept_flag)` is now called and can overrule the "accept"/"reject" decision either way. We keep accepting objects like this until the target is reached, or proves impossible to reach.

(8) The objects are now grouped so that any set of indistinguishable objects forms a single group. "Indistinguishable" means that no further text typed by the player could clarify which is meant (see §29). Note that there is no reason to suppose that two indistinguishable objects have the same score, because they might be in different places.

(9d) In definite mode, we know that there's a tie for highest score, as otherwise a choice would have been made at step (6d). If these highest-scoring objects belong to more than one group, then ask the player to choose which group:

```
You can see a bronze coin and four gold coins here.
>get coin
Which do you mean, the bronze coin or a gold coin?
>gold
```

The player's response is inserted textually into the original input and the parsing begins again from scratch with "get gold coin" instead of "get coin".

(10) Only two possibilities remain: either (i) we are in indefinite but singular mode, or (ii) we are in definite mode and there is a tie for highest-scoring object and all of these equal-highest objects belong to the same group. Either way, choose the "best guess" object (see below). Should this parsing attempt eventually prove successful, print up an "inference" on screen, such as

```
>get key
(the copper key)
```

only if the number of groups found in (8) is more than 1.

(BG) It remains to define "best guess". From a set of objects, the best guess is the highest-scoring one not yet guessed; and if several objects have equal highest scores, it is the earliest one to have been matched by the parser. In practice this means the one most recently taken or dropped, because the parser tries to match against objects by traversing the object-tree, and the most recently moved object tends to be the first in the list of children of its parent.

• REFERENCES

See 'Balances' for a usage of `ParserError`. •Infocom's parser typically produces error messages like "I don't know the word 'tarantula'." when the player types a word not in the game's dictionary, and some designers prefer this style to Inform's give-nothing-away approach (Inform tries not to let the player carry out experiments to see what is, and what is not, in the dictionary). Neil Cerutti's `"dunno.h"` library extension restores the Infocom format. •The library extension `"calyx_adjectives.h"`, which resolves ambiguities in parsing by placing more weight on matches with nouns than with adjectives, works by using `ChooseObjects`.