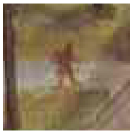
Chapter II: Introduction to Designing

But 'why then publish?' There are no rewards
   Of fame or profit when the world grows weary.
I ask in turn why do you play at cards?
   Why drink? Why read? To make some hour less dreary.
It occupies me to turn back regards
   On what I've seen or pondered, sad or cheery,
And what I write I cast upon the stream
To swim or sink. I have had at least my dream.

— Lord Byron (1788–1824), *Don Juan, canto XIV*

§4 'Ruins' begun



This chapter introduces five fundamentals of Inform: how to construct games; messages, classes and actions; and how to debug games. Chapter III then makes a systematic exploration of the model world available to Inform games. Throughout both chapters, examples gradually build up a modest game called 'Ruins', a tale of Central American archaeology in the 1930s. Here is its first state:

```
Constant Story "RUINS";
Constant Headline "^An Interactive Worked Example^
             Copyright (c) 1999 by Angela M. Horns.^";
Include "Parser";
Include "VerbLib";
Object Forest "~Great Plaza~"
  with description
           "Or so your notes call this low escarpment of limestone,
           but the rainforest has claimed it back. Dark olive
           trees crowd in on all sides, the air steams with the
           mist of a warm recent rain, midges hang in the air.
           ~Structure 10~ is a shambles of masonry which might
           once have been a burial pyramid, and little survives
           except stone-cut steps leading down into darkness below.",
  has  light;
[ Initialise;
  location = Forest;
  "^^^Days of searching, days of thirsty hacking through the briars of
  the forest, but at last your patience was rewarded. A discovery!^";
];
Include "Grammar";
```

If you can compile this tiny beginning successfully, Inform is probably set up and working properly on your computer. Compilation may take a few seconds, because although you have only written twenty lines or so, the `Include` directives paste in another seven and a half thousand. This is "the Library", a computer program which acts as umpire during play. The library is divided into three parts:

`Parser`	which decodes what the player types;
`VerbLib`	how actions, like "take" or "go north", work;
`Grammar`	the verbs and phrases which the game understands.

It does matter what order the three lines beginning with `Include` come in, and it sometimes matters where your own code goes with respect to them: objects shouldn't be declared until after the inclusion of the parser, for instance. For now, follow the structure above, with everything interesting placed between the inclusions of `VerbLib` and `Grammar`.

△ Chapter I above said that every Inform program had to contain a routine called `Main`. Games like 'Ruins' are no exception, but their `Main` routine is part of the library, so that game designers do not need to write a `Main`.

. . . . .

The two constants at the beginning are text giving the game's name and copyright message, which the library needs in order to print out the "banner" announcing the game. Similarly, the library expects to find a routine named `Initialise` somewhere in your source code. This routine is called when the game starts up, and is expected to carry out any last setting-up operations before play begins. In most games, it also prints up a 'welcome' message, but the one thing it *has* to do is to set the `location` variable to the place where the player begins. And this means that every game has to declare at least one object, too: the room where the player begins.

△ In this book places are often called "rooms" even when outdoors (like `Forest`) or underground. This goes back at least to Stephen Bishop's 1842 map of the Mammoth and Flint Ridge cave system of Kentucky, which was the setting of the first adventure game, 'Advent', also called 'Colossal Cave' (c.1975). The author, Will Crowther, was a caver and used the word "room" in its caving sense. Don Woods, who recast the game in 1976–7, confused the word further with its everyday sense. Players of adventure games continue to call locations "rooms" to this day.

. . . . .

'Ruins' is at this stage an exceedingly dull game:

> Days of searching, days of thirsty hacking through the briars of the forest, but at last your patience was rewarded. A discovery!
>
> *RUINS*
> An Interactive Worked Example
> Copyright (c) 1998 by Angela M. Horns.
> Release 1 / Serial number 990220 / Inform v6.20 Library 6/8
>
> *"Great Plaza"*
> Or so your notes call this low escarpment of limestone, but the rainforest has claimed it back. Dark olive trees crowd in on all sides, the air steams with the mist of a warm recent rain, midges hang in the air. "Structure 10" is a shambles of masonry which might once have been a burial pyramid, and little survives except stone-cut steps leading down into darkness below.
> \>`inventory`
> You are carrying nothing.
> \>`north`
> You can't go that way.
> \>`wait`
> Time passes.
> \>`quit`
> Are you sure you want to quit? `yes`

. . . . .

In an Inform game, objects are used to simulate everything: rooms and items to be picked up, scenery, the player, intangible things like mist and even some abstract ideas, like the direction "north" or the idea of "darkness". The library itself is present as an object, called `InformLibrary`, though like the concept of "north" it cannot be picked up or visited during play. All told, 'Ruins' already contains twenty-four objects.

It is time to add something tangible, by writing the following just after the definition of `Forest`:

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

The arrow `->` means that the mushroom begins inside the previous object, which is to say, the `Forest`. If the game is recompiled, the mushroom is now in play: the player can call it "speckled mushroom", "mushroom", "toadstool" and so on. It can be taken, dropped, looked at, looked under and so on. However, it only adds the rather plain line "There is a speckled mushroom here." to the Forest's description. Here is a more decorative species:

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

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

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

Now if we examine the mushroom, as is always wise, we get a cautionary hint. But the `edible` notation means that it *can* be eaten, so that for the first time the player can change the game state irrevocably: from a game with a forest and a mushroom into a game with just a forest.

The mushroom shows the two kinds of feature something can have: a "property" with some definite value or list of values and an "attribute", which is either present or not but has no particular value. `name`, `initial` and `description` are all properties, while `light` and `edible` are attributes. The current state of these properties changes during play: for instance, it can be changed by code like the following.

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

`light` is the attribute for "giving off light". The `Forest` was defined as having `light` on account of daylight, so it doesn't much matter whether or not the mushroom has `light`, but for the sake of botanical verisimilitude it won't have `light` in the final game.

. . . . .

Declaring objects has so far been a matter of filling in forms: fill some text into the box marked `description`, and so on. We could go much further like this, but for the sake of example it's time to add some rules:

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

The property `after` doesn't just have a string for a value: it has a routine of its own. What happens is that after something happens to the mushroom, the library asks the mushroom if it would like to react in some way. In this case, it reacts only to `Take` and `Drop`, and the only effect is that the usual messages ("Taken." "Dropped.") are replaced by new ones. (It doesn't react to `Eat`, so nothing out of the ordinary happens when it's eaten.) 'Ruins' can now manage a briefly plausible dialogue:

> *"Great Plaza"*
> Or so your notes call this low escarpment of limestone, but the rainforest has claimed it back. Dark olive trees crowd in on all sides, the air steams with the mist of a warm recent rain, midges hang in the air. "Structure 10" is a shambles of masonry which might once have been a burial pyramid, and little survives except stone-cut steps leading down into darkness below.
> A speckled mushroom grows out of the sodden earth, on a long stalk.
> `>get mushroom`
> You pick the mushroom, neatly cleaving its thin stalk.
> `>look at it`
> The mushroom is capped with blotches, and you aren't at all sure it's not a toadstool.
> `>drop it`
> The mushroom drops to the ground, battered slightly.

△ Gareth Rees persuasively advocates writing this sort of transcript, of an ideal sequence of play, first, and worrying about how to code up the design afterwards. Other designers prefer to build from the bottom up, crafting the objects one at a time and finally bringing them together into the narrative.

. . . . .

The mushroom is a little more convincing now, but still does nothing. Here is a more substantial new rule:

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

The library consults `before` just before the player's intended action would take place. So when the player tries typing, say, "eat the mushroom", what happens is: in 30% of cases, death by toadstool poisoning; and in the other 70%, a nibble of a corner of fungus, without consuming it completely.

Like `location`, `deadflag` is a variable belonging to the library. It's normally `false`, meaning that the player is still alive and playing. Setting it to `true` thus kills the player. (Setting it to 2 causes the player to win the game and there are other uses: see §21.)

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

```
    return true;   and   rtrue;
```

both do the same thing. Moreover,

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

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

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

is understood to mean the same thing. To print the text without returning, the statement print has to be written out in full. Here is an example:

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

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

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

. . . . .

The following example of "form-filling" is typical of the way that the library provides for several standard kinds of object. This one is a kind of door, which will be gone into properly in §13, but for now suffice to say that a door doesn't literally have to be a door: it can be any object which comes in between where the player is and where the player can go. Because the object is also marked as scenery (see §8), it isn't given any special paragraph of description when the Forest is described. Finally, it is marked as static to prevent the player from being able to pick it up and walk off with it.

```
Object -> steps "stone-cut steps"
  with name 'steps' 'stone' 'stairs' 'stone-cut' 'pyramid' 'burial'
            'structure' 'ten' '10',
       description
           "The cracked and worn steps descend into a dim chamber.
           Yours might be the first feet to tread them for five
           hundred years.",
       door_to Square_Chamber,
       door_dir d_to
  has  scenery static door open;
```

We also need to add a new line to the Forest's definition to tell it that the way down is by these steps:

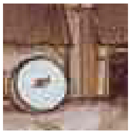
```
Object Forest "~Great Plaza~"
       ...
       d_to steps,
```

Now “examine structure 10”, “enter stone-cut pyramid” and so forth will all work.

● **EXERCISE 2**
Except of course that now ‘Ruins’ won’t compile, because Inform expects to find a room called `Square_Chamber` which the steps lead to. Design one.

§5 Introducing messages and classes

On a round ball
A workman that hath copies by, can lay
An Europe, Afrique and an Asia,
And quickly make that, which was nothing, All.
— John Donne (1571?–1631), *Valediction: Of Weeping*



Though §4 was a little vague in saying "the library asks the mushroom if it would like to react", something basic was happening in Inform terms: the object InformLibrary was sending the message before to the object mushroom. Much more on how actions take place in §6, but here is roughly what the library does if the player types "eat mushroom":

```
if (mushroom.before() == false) {
    remove mushroom;
    if (mushroom.after() == false)
        print "You eat the mushroom. Not bad.^";
}
```

The library sends the message before to ask the mushroom if it minds being eaten; then, if not, it consumes the mushroom; then it sends the message after to ask if the mushroom wishes to react in some way; then, if not, it prints the usual eating-something text. In response to the messages before and after, the mushroom is expected to reply either true, meaning "I'll take over from here", or false, meaning "carry on".

Most of the other properties in §4 are also receiving messages. For example, the message

```
mushroom.description();
```

is sent when the player tries to examine the mushroom: if the reply is false then the library prints "You see nothing special about the speckled mushroom." Now the mushroom was set up with

```
description
    "The mushroom is capped with blotches, and you aren't at all
    sure it's not a toadstool.",
```

which doesn't look like a rule for receiving a message, but it is one all the same: it means "print this text out, print a new-line and reply `true`". A more complicated rule could have been given instead, as in the following elaboration of the stone-cut steps in 'Ruins':

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

`visited` is an attribute which is currently held only by rooms which the player has been to. The glyphs will come into the game later on.

The library can send out about 40 different kinds of message, `before` and `description` being two of these. The more interesting an object is, the more ingeniously it will respond to these messages. An object which ignores all incoming messages will be lifeless and inert in play, like a small stone.

△ Some properties are just properties, and don't receive messages. Nobody ever sends a `name` message, for instance: the `name` property is just what it seems to be, a list of words.

· · · · ·

So the library is sending out messages to your objects all the time during play. Your objects can also send each other messages, including "new" ones that the library would never send. It's sometimes convenient to use these to trigger off happenings in the game. For example, one way to provide hints in 'Ruins' might be to include a macaw which squawks from time to time, for a variety of reasons:

```
Object -> macaw "red-tailed macaw"
  with name 'red' 'tailed' 'red-tailed' 'macaw' 'bird',
       initial "A red-tailed macaw eyes you from an upper branch.",
       description "Beautiful plumage.",
       before [;
           Take: "The macaw flutters effortlessly out of reach.";
       ],
       squawk [ utterance;
           if (self in location)
```

```
                print "The macaw squawks, ~", (string) utterance,
                    "! ", (string) utterance, "!~^^";
        ],
  has  animate;
```

(For the final version of 'Ruins' the designer thought better of the macaw and removed it, but it still makes a good example.) We might then, for instance, change the `after` rule for dropping the mushroom to read:

```
    Drop: macaw.squawk("Drop the mushroom");
         "The mushroom drops to the ground, battered slightly.";
```

so that the maddening creature would squawk "Drop the mushroom! Drop the mushroom!" each time this was done. At present it would be an error to send a `squawk` message to any object other than the macaw, since only the macaw has been given a rule telling it what to do if it receives one.

· · · · ·

In most games there are groups of objects with certain rules in common, which it would be tiresome to have to write out many times. For making such a group, a class definition is simpler and more elegant. These closely resemble object definitions, but since they define prototypes rather than actual things, they have no initial location. (An individual tree may be somewhere, but the concept of being a tree has no particular place.) So the 'header' part of the definition is simpler.

For example, the scoring system in 'Ruins' works as follows: the player, an archaeologist of the old school, gets a certain number of points for each 'treasure' (i.e., cultural artifact) he can filch and put away into his packing case. Treasures clearly have rules in common, and the following class defines them:

```
Class Treasure
 with cultural_value 5, photographed_in_situ false,
      before [;
          Take, Remove:
              if (self in packing_case)
                  "Unpacking such a priceless artifact had best wait
                  until the Carnegie Institution can do it.";
              if (self.photographed_in_situ == false)
                  "This is the 1930s, not the bad old days. Taking an
                  artifact without recording its context is simply
                  looting.";
          Photograph:
            if (self has moved)
                "What, and fake the archaeological record?";
            if (self.photographed_in_situ) "Not again.";
    ],
    after [;
        Insert:
            if (second == packing_case)
            {   score = score + self.cultural_value;
                "Safely packed away.";
            }
        Photograph: self.photographed_in_situ = true;
    ];
```

(The packing case won't be defined until §12, which is about containers.) Note that self is a variable, which always means "whatever object I am". If we used it in the definition of the mushroom it would mean the mushroom: used here, it means whatever treasure happens to be being dealt with. Explanations about Insert and Remove will come later (in §12). The action Photograph is not one of the standard actions built in to the library, and will be added to 'Ruins' in the next section.

An object of the class Treasure automatically inherits the properties and attributes given in the class definition. Here for instance is an artifact which will eventually be found in the Stooped Corridor of 'Ruins':

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

From Treasure, this statuette inherits a cultural_value score of 5 and the rules about taking and dropping treasures. If it had itself set cultural_value to 15, say, then the value would be 15, because the object's actual definition always takes priority over anything the class might have specified. Another of the five 'Ruins' treasures, which will be found in the Burial Shaft, has a subtlety in its definition:

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

The subtlety is that the honeycomb now has two `after` rules: a new one of its own, plus the existing one that all treasures have. Both apply, but the new one happens first.

△△ So comparing `cultural_value` and `after`, there seems to be an inconsistency. In the case of `cultural_value`, an object's own given value wiped out the value from the class, but in the case of `after`, the two values were joined up into a list. Why? The reason is that some of the library's properties are "additive", so that their values accumulate into a list when class inheritance takes place. Three useful examples are `before`, `after` and `name`.

△△ Non-library properties you invent (like `squawk` or `cultural_value`) will never be additive, unless you declare them so with a directive like

```
    Property additive squawk;
```

before `squawk` is otherwise mentioned. (Or you could imitate similar kinds of inheritance using the superclass operator.)

• REFERENCES
See 'Balances' for an extensive use of message-sending. The game defines several complicated classes, among them the white cube, spell and scroll classes. •'Advent' has a treasure-class similar to this one, and uses class definitions for the many similar maze and dead-end rooms, as well as the sides of the fissure. •'Toyshop' contains one easy class (the wax candles) and one unusually hard one (the building blocks). •Class definitions can be worthwhile even when as few as two objects use them, as can be seen from the two kittens in 'Alice Through the Looking-Glass'.

§6 Actions and reactions

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

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

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

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

are all examples. Here the actual actions are `Inv` (inventory), `Take` and `Insert`. An action has none, one or two objects supplied with it (or, in a few special cases, some numerical information rather than objects). It also has an "actor", the person who is to perform the action, usually the player. Most actions are triggered off by the game's parser, whose job can be summed up as reducing the player's keyboard commands to actions: "take my hat off", "remove bowler" or "togli il cappello" (if in an Italian game) might all cause the same action. Some keyboard commands, like "drop all", cause the parser to fire off whole sequences of actions: others, like "empty the sack into the umbrella stand", cause only a single action but one which may trigger off an avalanche of other actions as it takes place.

An action is only an attempt to do something: it may not succeed. Firstly, a `before` rule might interfere, as we have seen already. Secondly, the action might not even be very sensible. The parser will happily generate the action `Eat iron_girder` if the player asked to do so in good English. In this case, even if no `before` rule interferes, the normal game rules will ensure that the girder is not consumed.

Actions can also be generated by your own code, and this perfectly simulates the effect of a player typing something. For example, generating a `Look` action makes the game produce a room description as if the player had typed "look". More subtly, suppose the air in the Pepper Room causes the player to sneeze each turn and drop something at random. This could be programmed directly, with objects being moved onto the floor by explicit `move` statements. But then suppose the game also contains a toffee apple, which sticks to the player's hands. Suddenly the toffee apple problem has an unintended solution. So rather than moving the objects directly to the floor, the game should generate `Drop` actions, allowing the game's rules to be applied. The result might read:

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

which is at least consistent.

As an example of causing actions, an odorous `low_mist` will soon settle over 'Ruins'. It will have the `description` "The mist carries an aroma reminisicent of tortilla." The alert player who reads this will immediately type "smell mist", and we want to provide a better response than the game's stock reply "You smell nothing unexpected." An economical way of doing this is to somehow deflect the action `Smell low_mist` into the action `Examine low_mist` instead, so that the "aroma of tortilla" message is printed in this case too. Here is a suitable `before` rule to do that:

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

The statement `<Examine self>` causes the action `Examine low_mist` to be triggered off immediately, after which whatever was going on at the time resumes. In this case, the action `Smell low_mist` resumes, but since we immediately return `true` the action is stopped dead.

Causing an action and then returning `true` is so useful that it has an abbreviation, putting the action in double angle-brackets. For example, the following could be added to 'Ruins' if the designer wanted to make the stone-cut steps more enticing:

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

If a player types "search steps", the parser will produce the action `Search steps` and this rule will come into play: it will generate the action `Enter steps` instead, and return `true` to stop the original `Search` action from going any further. The net effect is that one action has been diverted into another.

· · · · ·

At any given time, just one action is under way, though others may be waiting to resume when the current one has finished. The current action is always stored in the four variables

```
actor      action      noun      second
```

`actor`, `noun` and `second` hold the objects involved, or the special value `nothing` if they aren't involved at all. (There's always an `actor`, and for the time being it will always be equal to `player`.) `action` holds the kind of action. Its possible values can be referred to in the program using the `##` notation: for example

```
if (action == ##Look) ...
```

tests to see if the current action is a `Look`.

△ Why have `##` at all, why not just write `Look`? Partly because this way the reader of the source code can see at a glance that an action type is being referred to, but also because the name might be used for something else. For instance there's a variable called `score` (holding the current game score), quite different from the action type `##Score`.

△△ For a few actions, the 'noun' (or the 'second noun') is actually a number (for instance, "set timer to 20" would probably end up with `noun` being `timer` and `second` being 20). Occasionally one needs to be sure of the difference, e.g., to tell if `second` is holding a number or an object. It's then useful to know that there are two more primitive variables, `inp1` and `inp2`, parallel to `noun` and `second` and usually equal to them – but equal to 1 to indicate "some numerical value, not an object".

· · · · ·

The library supports about 120 different actions and most large games will add some more of their own. The full list, given in Table 6, is initially daunting, but for any given object most of the actions are irrelevant. For instance, if you only want to prevent an object from entering the player's possession, you need only block the `Take` action, unless the object is initially in something or on something, in which case you need to block `Remove` as well. In the author's game 'Curses', one exceptional object (Austin, the cat) contains rules concerning 15 different actions, but the average is more like two or three action-rules per object.

The list of actions is divided into three groups, called Group 1, Group 2 and Group 3:

1. Group 1 contains 'meta' actions for controlling the game, like `Score` and `Save`, which are treated quite differently from other actions as they do not happen in the "model world".
2. Actions in group 2 normally do something to change the state of the model world, or else to print important information about it. `Take` ("pick up") and `Inv` ("inventory") are examples of each. Such actions will affect any object which doesn't block them with a `before` rule.
3. Finally, group 3 actions are the ones which normally do nothing but print a polite refusal, like `Pull` ("it is fixed in place"), or a bland response, like `Listen` ("you hear nothing unexpected"). Such actions will never affect any object which doesn't positively react with a `before` rule.

△ Some of the group 2 actions can be ignored by the programmer because they are really only keyboard shorthands for the player. For example, `<Empty rucksack table>` means "empty the contents of the rucksack onto the table" and is automatically broken down into a stream of actions like `<Remove fish rucksack>` and `<PutOn fish table>`. You needn't write rules concerning `Empty`, only `Remove` and `PutOn`.

△ Most of the library's group 2 actions are able to "run silently". This means that if the variable `keep_silent` is set to `true`, then the actions print nothing in the event of success. The group 2 actions which can't run silently are exactly those ones whose successful operation does nothing but print: `Wait`, `Inv`, `Look`, `Examine`, `Search`.

• △EXERCISE 3
"The door-handle of my room... was different from all other door-handles in the world, inasmuch as it seemed to open of its own accord and without my having to turn it, so unconscious had its manipulation become..." (Marcel Proust). Use silent-running actions to make an unconsciously manipulated door: if the player tries to pass through when it's closed, print "(first opening the door)" and do so. (You need to know some of §13, the section on doors, to answer this.)

• △△EXERCISE 4
Now add "(first unlocking the door with ...)", automatically trying to unlock it using either a key already known to work, or failing that, any key carried by the player which hasn't been tried in the lock before.

· · · · ·

△ Some actions happen even though they don't arise *directly* from anything the player has typed. For instance, an action called `ThrownAt` is listed under group 3 in Table 6. It's a side-effect of the ordinary `ThrowAt` action: if the player types "throw rock at dalek", the parser generates the action `ThrowAt rock dalek`. As usual the rock is sent a `before` message asking if it objects to being thrown at a Dalek. Since the Dalek may also have an opinion on the matter, another `before` message is sent to the Dalek, but this time with the action ThrownAt. A dartboard can thus distinguish between being thrown, and having things thrown at it:

```
before [;
    ThrowAt: "Haven't you got that the wrong way round?";
    ThrownAt:
        if (noun==dart) {
            move dart to self;
            if (random(31)==1)
                print (string) random("Outer bull", "Bullseye");
            else {
                print (string) random("Single", "Double", "Triple");
                print " ", (number) random(20);
            }
            "!";
        }
        move noun to location;
        print_ret (The) noun, " bounces back off the board.";
],
```

Such an imaginary action – usually, as in this case, a perfectly sensible action seen from the point of view of the second object involved, rather than the first – is sometimes called a "fake action". Two things about it are fake: there's no grammar that produces ThrownAt, and there's no routine called ThrownAtSub. The important fake actions are ThrownAt, Receive and LetGo, the latter two being used for containers: see §12.

△△ If you really need to, you can declare a new fake action with the directive Fake_action ⟨Action-name⟩;. You can then cause this action with < and > as usual.

• △△**EXERCISE 5**
ThrownAt would be unnecessary if Inform had an idea of before and after routines which an object could provide if it were the second noun of an action. How might this be implemented?

△△ Very occasionally, in the darker recesses of §18 for instance, you want "fake fake actions", actions which are only halfway faked in that they still have action routines. Actually, these are perfectly genuine actions, but with the parser's grammar jinxed so that they can never be produced whatever the player types.

· · · · ·

The standard stock of actions is easily added to. Two things are necessary to create a new action: first one must provide a routine to make it happen. For instance:

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

Every action has to have a "subroutine" like this, the name of which is always the name of the action with `Sub` appended. Secondly, one must add grammar so that `Blorple` can actually be called for. Far more about grammar in Chapter IV: for now we add the simplest of all grammar lines, a directive

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

placed after the inclusion of the `Grammar` file. The word "blorple" can now be used as a verb. It can't take any nouns, so the parser will complain if the player types "blorple daisy".

`Blorple` is now a typical Group 3 action. `before` rules can be written for it, and it can be triggered off by a statement like

```
    <Blorple>;
```

The unusual action in 'Ruins', `Photograph`, needs to be a Group 2 action, since it actually does something, and objects need to be able to react with `after` rules. (Indeed, the definition of the `Treasure` class in the previous section contains just such an `after` rule.) A photographer needs a camera:

```
Object -> -> camera "wet-plate camera"
  with name 'wet-plate' 'plate' 'wet' 'camera',
       description
          "A cumbersome, sturdy, stubborn wooden-framed wet plate
          model: like all archaeologists, you have a love-hate
          relationship with your camera.";
```

(This is going to be inside a packing case which is inside the Forest, hence the two arrows `->`.) And now the action subroutine. The sodium lamp referred to will be constructed in §14.

```
[ PhotographSub;
  if (camera notin player) "Not without the use of your camera.";
  if (noun == player) "Best not. You haven't shaved since Mexico.";
  if (children(player) > 1)
     "Photography is a cumbersome business, needing the use of both
      hands. You'll have to put everything else down.";
  if (location == Forest) "In this rain-soaked forest, best not.";
  if (location == thedark) "It is far too dark.";
  if (AfterRoutines()) return;
 "You set up the elephantine, large-format, wet-plate camera, adjust
  the sodium lamp and make a patient exposure of ", (the) noun, ".";
];
```

What makes this a Group 2 action is that, if the action successfully takes place, then the library routine `AfterRoutines` is called. This routine takes care of all the standard rules to do with `after` (see below), and returns `true` if any object involved has dealt with the action and printed something already. (Failing that, the message "You set up..." will be printed.) Finally, some grammar for the parser:

```
Verb 'photograph' * noun -> Photograph;
```

This matches input like "photograph statuette", because the grammar token `noun` tells the parser to expect the name of a visible object. See §30 and §31 for much more on grammar.

△ To make a Group 1 action, define the verb as `meta` (see §30).

. . . . .

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

(1) 'Before', for group 2 and 3 actions. An opportunity for your code to interfere with or block altogether what might soon happen.
(2) 'During', for all actions. The library takes control and decides if the action makes sense according to its normal world model: for example, only an `edible` object may be eaten; only an object in the player's possession can be thrown at somebody, and so on. If the action is impossible, a complaint is printed and that's all. Otherwise the action is now carried out.
(3) 'After', for group 2 actions. An opportunity for your code to react to what has happened, after it has happened but before any text announcing it has been printed. If it chooses, your code can print and cause an entirely different outcome. If your code doesn't interfere, the library reports back to the player (with such choice phrases as "Dropped.").

△ Group 1 actions, like `Score`, have no 'Before' or 'After' stages: you can't (easily) stop them from taking place. They aren't happening in the game's world, but in the player's.

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

(1a) The `GamePreRoutine` is called, if you have written one. If it returns `true`, nothing else happens and the action is stopped.
(1b) The `orders` property of the player is called on the same terms. For more details, see §18.
(1c) And the `react_before` of every object in scope, which roughly means 'in the vicinity'. For more details, see §32.

(1d) And the `before` of the current room.
(1e) If the action has a first noun, its `before` is called on the same terms.

△ The library processes the 'During' stage by calling the action's subroutine: for instance, by calling `TakeSub`.

△ The 'After' stage only applies to Group 2 actions, as all Group 3 actions have been wound up with a complaint or a bland response at the 'During' stage. During 'After' the sequence is as follows: (3a) `react_after` rules for every object in scope (including the player object); (3b) the room's `after`; (3c) the first noun's `after` and (3d) finally `GamePostRoutine`.

△△ To some extent you can even meddle with the 'During' stage, and thus even interfere with Group 1 actions, by unscrupulous use of the `LibraryMessages` system. See §25.

· · · · ·

As mentioned above, the parser can generate decidedly odd actions, such as `Insert camel eye_of_needle`. The parser's policy is to allow any action which the player has clearly asked for at the keyboard, and it never uses knowledge about the current game position except to resolve ambiguities. For instance, "take house" in the presence of the Sydney Opera House and also a souvenir model of the same will be resolved in favour of the model. But if there is no model to cloud the issue, the parser will cheerfully generate `Take Sydney_Opera_House`.

Actions are only checked for sensibleness *after* the `before` stage. In many ways this is a good thing, because in adventure games the very unlikely is sometimes correct. But sometimes it needs to be remembered when writing `before` rules. Suppose a `before` rule intercepts the action of putting the mushroom in the crate, and exciting things happen as a result. Now even if the mushroom is, say, sealed up inside a glass jar, the parser will still generate the action `Insert mushroom crate`, and the `before` rule will still cut in, because the impossibility of the action hasn't yet been realised.

The upshot of this is that the exciting happening should be written not as a `before` but as an `after` rule, when it's known that the attempt to put the mushroom in the crate has already succeeded.

△ That's fine if it's a Group 2 action you're working with. But consider the following scenario: a siren has a cord which needs to be pulled to sound the alarm. But the siren can be behind glass, and is on the other side of a barred cage in which the player is imprisoned. You need to write a rule for `Pull cord`, but you can't place this among the cord's `after` rules because `Pull` is a group 3 action and there isn't any "after": so it has to be a `before` rule. Probably it's best to write your own code by hand to check that the cord is reachable. But an alternative is to call the library's routine:

```
    ObjectIsUntouchable(item, silent_flag, take_flag)
```

This determines whether or not the player can touch `item`, returning `true` if there is some obstruction. If `silent_flag` is `true`, or if there's no obstruction anyway, nothing will be printed. Otherwise a suitable message will be printed up, such as "The barred cage isn't open." So a safe way to write the cord's `before` rule would be:

```
    before [;
        Pull: if (ObjectIsUntouchable(self)) rtrue;
              "~Vwoorp! Vwoorp!~";
    ],
```

`ObjectIsUntouchable` can also be a convenience when writing action subroutines for new actions of your own.

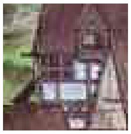
△△ If you set `take_flag`, then a further restriction will be imposed: the `item` must not belong to something or someone already: specifically, it must not be in the possession of an `animate` or a `transparent` object that isn't a `container` or `supporter`. For instance, the off button on a television set can certainly be touched, but if `take_flag` is true, then `ObjectIsUntouchable` will print up "That seems to be a part of the television set." and return `true` to report an obstacle.

• REFERENCES
In a game compiled with the `-D` for "Debugging" switch set, the "actions" verb will result in trace information being printed each time any action is generated. Try putting many things into a rucksack and asking to "empty" it for an extravagant list. •Diverted actions (using << and >>) are commonplace. They're used in about 20 places in 'Advent': a good example is the way "take water" is translated into a `Fill bottle` action. •L. Ross Raszewski's library extension `"yesno.h"` makes an interesting use of `react_before` to handle semi-rhetorical questions. For instance, suppose the player types "eat whale", an absurd command to which the game replies "You can fit a blue whale in your mouth?" Should the player take this as a literal question and type "yes", the designer might want to be able to reply "Oh. I should never have let you go through all those doors." How might this be done? The trick is that, when the game's first reply is made, an invisible object is moved into play which does nothing except to react to a `Yes` action by making the second reply.

§7 Infix and the debugging verbs

If builders built buildings the way programmers write programs,
the first woodpecker that came along would destroy civilisation.

— old computing adage


Infocom fixed 1,695 documented bugs in the course of getting 'Sorcerer' from rough draft to first released product. Alpha testing of 'A Mind Forever Voyaging' turned up one bug report every three minutes. Adventure games are exhausting programs to test and debug because of the number of states they can get into, many of them unanticipated. (For instance, if the player solves the "last" puzzle first, do the other puzzles still work properly? Are they still fair?) The main source of error is simply the designer not noticing that some states are possible. The Inform library can't help with this, but it does contain some useful features designed to help the tester. These are worth finding out about, because if you're going to code up a game, you'll be spending a lot of time testing one thing or another.

Inform has three main debugging features, each making the story file larger and slower than it needs to be: each can be turned on or off with compiler switches. One feature is "strict mode" (switch `-S`), which checks that the story file isn't committing sins such as over-running arrays or treating `nothing` as if it were an object: trying to calculate `child(nothing)`, for instance. Strict mode is on by default, and automatically sets Debug mode whenever it is on. To get rid of strict mode, compile with `-~S` to turn switch `S` off.

Over and above this is the extensive "Infix" (switch `-X`), a far more potent set of debugging verbs, which significantly adds to the size of a story file (so that a really large story file might not be able to fit it in). Infix allows you to watch changes happening to objects and monitor routines of your choice, to send messages, call routines, set variables and so forth. Like Strict mode, Infix automatically switches Debug mode on.

Debug mode (switch `-D`) adds only a small suite of commands, the "debugging verbs", to any game. For instance, typing "purloin mousetrap" allows you to take the mousetrap wherever it happens to be in the game. The debugging verbs do take up extra space, but very little, and note that even if Strict mode and Infix are both off, you can still have Debug on its own.

The Infix and Debug modes give the player of a story file what amount to god-like powers, which is fine for testing but not for final release versions. As a precaution against accidents, the end of a game's printed banner indicates Infix mode with a capital letter 'X', Debug with 'D', and Strict with 'S'. Your source code can also detect which are set: in Strict mode the constant `STRICT_MODE` is defined; in Debug mode the constant `DEBUG`; with Infix the constant `INFIX`.

§7.1 Debugging verbs for command sequences

The basic testing technique used by most designers is to keep a master-list of commands side by side with the growing game: a sequence which takes the player from the beginning, explores everywhere, thoroughly tests every blind alley, sudden death and textual response and finally wins the game.

"recording" *or* "recording on" *or* "recording off"

Records all the commands you type into a file on your machine (your interpreter will probably ask you for a filename). When a new region of game is written, you may want to turn recording on and play through it: you can then add the resulting file to the master-list which plays the entire game.

"replay"

This immensely useful verb plays the game taking commands from a file on your machine, rather than from the keyboard. This means you can test every part of the entire game with minimal effort by replaying the master-list of commands through it.

"random"

If you're going to replay such recordings, you need the game to behave predictably: so that chance events always unfold in the same way. This means nobbling the random number generator, and the "random" verb does just that: i.e., after any two uses of "random", the same stream of random numbers results. So you want the first command in your master-list to be "random" if your game has any chance events in it.

△ If you have written a large and complicated game, you may well want to release occasional updates to correct mistakes found by players. But tampering with the code always runs the risk that you may fix one thing only to upset another. A useful insurance policy is to keep not only the master list of commands, but also the transcript of the text it should produce. Then when you amend something, you can replay the master list again and compare the new transcript with the old, ideally with a program like the Unix utility "diff". Any deviations mean a (possibly unintended) side effect of something you've done.

§7.2 *Undo*

Every Inform game provides the "undo" verb, which exactly restores the position before the last turn took place. Though this is not advertised, "undo" can even be used after death or victory, typed in at the "Would you like to RESTART, RESTORE a saved game..." prompt. It can be useful to include fatal moves, followed by "undo", in the master-list of commands, so testing death as well as life.

§7.3 *Debugging verbs which print useful information*

"showobj" ⟨anything⟩

"showobj" is very informative about the current state of an object, revealing which attributes it presently has and the values of its properties. ⟨anything⟩ can be the name of any object anywhere in the game (not necessarily in sight), or even the number of an object, which you might need if the object doesn't have a name.

"tree" *or* "tree" ⟨anything⟩

To see a listing of the objects in the game and how they contain each other, type "tree", and to see the possessions of one of them alone, use "tree ⟨that⟩". So "tree me" is quite like "inventory".

"showverb" ⟨verb⟩

For instance, "showverb unlock". This prints out what the parser thinks is the grammar for the named verb, in the form of an Inform `Verb` directive. This is useful if you're using the `Extend` directive to alter the library's grammar and want to check the result.

"scope" *or* "scope" ⟨anything⟩

Prints a list of all the objects currently in scope, and can optionally be given the name of someone else you want a list of the scope for ("scope pirate"). Roughly speaking, something is in your scope if you can see it from where you are: see §32.

§7.4 Debugging verbs which trace activity behind the scenes

Tracing is the process of printing up informative text which describes changes as they happen. Each of the following verbs can be given on its own, which sets tracing on; or followed by the word "on", which does the same; or followed by "off", which turns it off again.

"actions" *or* "actions on" *or* "actions off"

Traces all the actions generated in the game. For instance, here's what happens if you unlock and then enter the steel grate in 'Advent':

```
>enter grate
[ Action Enter with noun 51 (steel grate) ]
[ Action Go with noun 51 (steel grate) (from < > statement) ]
...
```

Which reveals that the `Enter` action has handed over to `Go`.

"messages" *or* "messages on" *or* "messages off"

Traces all messages sent between objects in the game. (Except for `short_name` messages, because this would look chaotic, especially when printing the status line.)

"timers" *or* "timers on" *or* "timers off"

Turning on "timers" shows the state of all active timers and daemons at the end of each turn. Typing this in the start of 'Advent' reveals that three daemons are at work: two controlling the threatening little dwarves and the pirate, and one which monitors the caves to see if every treasure has been found yet.

"changes" *or* "changes on" *or* "changes off"

Traces all movements of any object and all changes of attribute or property state.

```
>switch lamp on
[Giving brass lantern on]
[Giving brass lantern light]
You switch the brass lantern on.
[Setting brass lantern.power_remaining to 329]
In Debris Room
```

> You are in a debris room filled with stuff washed in from the surface. A low wide passage with cobbles becomes plugged with mud and debris here, but an awkward canyon leads upward and west.
> A note on the wall says, "Magic word XYZZY."
> A three foot black rod with a rusty star on one end lies nearby.
> [Giving In Debris Room visited]

Warning: this verb has effect only if the story file was compiled with the `-S` switch set, which it is by default.

△ Two things "changes" will not notice: (i) changes in the `workflag` attribute, because this flickers rapidly on and off with only temporary significance as the parser works, and (ii) changes to the entries in an array which is held in a property.

"trace" *or* "trace" ⟨number⟩ *or* "trace off"

There are times when it's hard to work out what the parser is up to and why (actually, most times are like this: but sometimes it matters). The parser is written in levels, the lower levels of which are murky indeed. Most of the interesting things happen in the middle levels, and these are the ones for which tracing is available. The levels which can be traced are:

Level 1	Parsing a ⟨grammar line⟩
Level 2	Individual tokens of a ⟨grammar line⟩
Level 3	Parsing a ⟨noun phrase⟩
Level 4	Resolving ambiguities and making choices of object(s)
Level 5	Comparing text against an individual object

"trace" or "trace on" give only level 1 tracing. Be warned: "trace 5" can produce reams and reams of text. There is a level lower even than that, but it's too busy doing dull spade-work to waste time printing. There's also a level 0, but it consists mostly of making arrangements for level 1 and doesn't need much debugging attention.

§7.5 *Debugging verbs which alter the game state in supernatural ways*

"purloin" ⟨anything⟩

You can "purloin" any item or items in your game at any time, wherever you are, even if they wouldn't normally be takeable. A typical use: "purloin all keys". Purloining something automatically takes away the `concealed` attribute, if necessary.

“abstract” ⟨anything⟩ “to” ⟨anything⟩

You can likewise “abstract" any item to any other item, meaning: move it to the other item. This is unlikely to make sense unless the other item is a `container`, `supporter` or `animate` object.

“goto” ⟨room number⟩

Teleports you to the numbered room. Because rooms don’t usually have names, referring to them by number (as printed in the “tree” output) is the best that can be done. . .

“gonear” ⟨anything⟩

. . . unless you can instead name something which is in that room. So for instance “gonear trident” teleports to the room containing the trident.

§7.6 *The Infix verbs*

Although Infix adds relatively few additional verbs to the stock, they are immeasurably stronger. All of them begin with a semicolon ; and the convention is that anything you type beginning with a semicolon is addressed to Infix.

“;” ⟨expression⟩

This calculates the value of the expression and prints it out. At first sight, this is no more than a pocket calculator, and indeed you can use it that way:

```
>; 1*2*3*4*5*6*7
; == 5040
```

But the ⟨expression⟩ can be *almost any Inform expression*, including variables, constants, action, array, object and class names, routine calls, message-sending and so on. It can be a condition, in which case the answer is 0 for `false` and 1 for `true`. It can even be an assignment.

```
>; score
; == 36
>; score = 1000
; == 1000
```

```
[Your score has just gone up by nine hundred and sixty-four points.]
>; brass_lantern has light
; false
>; lamp.power_remaining = 330
(brass lantern (39))
; == 330
>; child(wicker cage)
(wicker cage (55))
; == "nothing" (0)
>; children(me)
(yourself (20))
; == 4
```

In the dialogue above, from ‘Advent’ compiled with `-X`, the player called the same item both `brass_lantern`, the name it has in the source code, and “lamp”, the name it normally has in the game. When Infix is unable to understand a term like “lamp” as referring to the source code, it tries to match it to an object somewhere in the game, and prints up any guess it makes. This is why it printed “(brass lantern (39))”. (39 happens to be the object number of the brass lantern.) Pronouns like “me” and “it” can also be used to refer to objects.

```
>; StopDaemon(pirate)
; == 1
>; InformLibrary.begin_action(##Take, black_rod)
black rod with a rusty star on the end: Taken.
; == 0
```

The routine `StopDaemon` will appear later in §20: roughly speaking, “daemons” control random interventions, and stopping them is useful to keep (say) the bearded pirate from appearing and disrupting the replay of a sequence of commands. The second example shows a message being sent, though there is a simpler way to cause actions:

“;<” ⟨action⟩ ⟨noun⟩ ⟨second⟩

which generates any action of your choice, whether or not the given ⟨noun⟩ and ⟨second⟩ (which are optional) are in your scope. Once generated, the action is subject to all the usual rules:

```
>;< Take black_rod
; <Take (the black rod with a rusty iron star on the end)
That isn't available.
```

Three Inform statements for changing objects are also available from Infix:

";give" ⟨expression⟩ ⟨attribute⟩
";move" ⟨expression⟩ "to" ⟨expression⟩
";remove" ⟨expression⟩

These do just what you'd expect. ";give" is especially useful if, as often happens, you find that you can't test some complicated new area of a game because you've forgotten to do something basic, such as to make a door a `door`, or to set up an outdoor afternoon location as having `light`. Using ";give" you can illuminate any dark place you stumble into:

```
Darkness
It is pitch dark, and you can't see a thing.
>;give real_location light
; give (the At "Y2") light
>wait
Time passes.
At "Y2"
You are in a large room, with a passage to the south, a passage to the west,
and a wall of broken rock to the east. There is a large "Y2" on a rock in the
room's center.
```

(The waiting was because Inform only checks light at the end of each turn, but ";give" and the other debugging verbs are set up to occupy no game time at all -- which is often useful, even if it's a slight nuisance here.) Infix also extends the facilities for watching the changing state of the game:

";watch" *or* ";w" ⟨named routine⟩
";watch" *or* ";w" ⟨named routine⟩ "off"
";watch" *or* ";w" ⟨object⟩
";watch" *or* ";w" ⟨object⟩ "off"

When a named routine is being watched, text is printed each time it is called, giving its name and the values of the arguments it started up with. For instance, ";watch StartTimer" will ensure that you're told of any `StartTimer(object, time_to_run)` call, and so of any timer that begins working. Watching an object, say ";watch lamp", will notify you when:

(1) any attribute of the lamp is given or taken away;
(2) any property of the lamp is set;
(3) any message is sent to a routine attached to the lamp;
(4) the lamp is moved;

(5) anything is moved to the lamp.

You can also watch things in general:

";watch objects"	watches every object
";watch timers"	watches timers and daemons each turn
";watch messages"	watches every message sent
";watch actions"	watches all actions generated

The final two Infix verbs extend the facilities for looking at things.

";examine" *or* ";x" ⟨something⟩
";inventory" *or* ";i"

";inventory" tells you the names known to Infix, which is a practical way to find out which classes, routines, objects and so on are inside the story file. ";examine" looks at the ⟨something⟩ and tells you whatever Infix knows about it: (a) numbers are translated to hexadecimal and to ZSCII values, where possible; (b) objects are "shown"; (c) classes are listed; (d) constants have their values given; (e) attributes produce a list of all objects currently having them; (f) properties produce a list of all objects currently providing them; (g) dictionary words are described; (h) verbs have their grammars listed; (i) arrays have their definitions shown and their contents listed; (j) global variables have their values given; (k) actions produce a list of all grammar known to the parser which can produce them. For instance:

```
>;x 'silver'
; Dictionary word 'silver' (address 27118): noun
>;x buffer
; Array buffer -> 120
; == 120 9 59 120 32 98 117 102 102 101 114 0 101 114 32 (then 105 zero
entries)
```

△ You can watch routines even without Infix, and the Inform language provides two features for this. Firstly, you can declare a routine with an asterisk `*` immediately after the name, which marks it for watching. For example, declaring a routine as follows

```
[ AnalyseObject * obj n m;
```

results in the game printing out lines like

```
[AnalyseObject, obj=26, n=0, m=0]
```

every time the routine is called. A more drastic measure is to compile the story file with the `-g` set. The ordinary setting `-g` or `-g1` marks every routine in your own source code to be watched, but not routines in the library (more accurately, not to routines defined in any "system file"). The setting `-g2` marks every routine from anywhere, but be warned, this produces an enormous melée of output.

△ If you do have Infix present, then you can always type ";watch ... off" to stop watching any routine marked with a `*` in the source code. Without Infix, there's no stopping it.

△△ At present, there is no source-level debugger for Inform. However, for the benefit of any such tool which somebody might like to write, Inform has a switch `-k` which makes it produce a file of "debugging information" to go with the story file. This file mostly contains cross-references between positions in the game and lines of source code. The details of its format are left to the *Technical Manual*.

● **REFERENCES**

Several exercises in this book are about defining new debugging verbs to test one thing or another, and most games have invented a few of their own. Early versions of 'Curses', for instance, allowed various supernatural actions beginning with "x" once the player had typed the command "xallow 57" (the author was then living above a carpet shop at 57 High Street, Oxford). But Paul David Doherty eventually disassembled the story file and found the secret. Moral: place any new debugging verbs inside an `Ifdef DEBUG;` directive, unless you plan on leaving them in as "Easter eggs" for people to find. ●A simple debugging verb called "xdeterm" is defined in the `DEBUG` version of 'Advent': it takes random events out of the game. ●Marnie Parker's library extension `"objlstr.h"`, which has contributions from Tony Lewis, provides a useful debugging verb "list", allowing for instance "list has door lockable locked", which lists all objects having that combination of attributes.