Chapter III: The Model World

> A Model must be built which will get everything in without a clash; and it can do this only by becoming intricate, by mediating its unity through a great, and finely ordered, multiplicity.
>
> — C. S. Lewis (1898–1963), *The Discarded Image*

§8 Places and scenery

In this longer chapter the model world of an Inform game will be explored and examples will gradually complete the 'Ruins' begun in Chapter II. So far, 'Ruins' contains just a location of rainforest together with some rules about photography. The immediate need is for a more substantial map, beginning with a sunken chamber. Like the Forest, this too has `light`, however dim. If it didn't, the player would never see it: in Inform's world darkness prevails unless the designer provides some kind of lamp or, as in this case, ambient light.

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

This room has a `name` property even though rooms are not usually referred to by players. The nouns given are words which Inform knows "you don't need to refer to", and it's a convention of the genre that the designer should signpost the game in this way. For the game to talk about something and later deny all knowledge – "I can't see any such thing" – is not merely rude but harmful to the player's illusion of holding a conversation about a real world. Better to parry with:

```
    >examine lintel
```

That's not something you need to refer to in the course of this game.

. . . . .

Not all of the Square Chamber's décor is so irrelevant:

```
Object -> "carved inscriptions"
  with name 'carved' 'inscriptions' 'carvings' 'marks' 'markings'
           'symbols' 'moving' 'scuttling' 'crowd' 'of',
       initial
           "Carved inscriptions crowd the walls, floor and ceiling.",
       description
           "Each time you look at the carvings closely, they seem
           to be still. But you have the uneasy feeling when you
           look away that they're scuttling, moving about. Two
           glyphs are prominent: Arrow and Circle.",
  has  static pluralname;
```

The `static` attribute means that the inscriptions can't be taken or moved. As we went out of our way to describe a shaft of sunlight, we'll include that as well:

```
Object -> sunlight "shaft of sunlight"
  with name 'shaft' 'of' 'sunlight' 'sun' 'light' 'beam' 'sunbeam'
           'ray' 'rays' 'sun^s' 'sunlit' 'air' 'motes' 'dust',
       description
           "Motes of dust glimmer in the shaft of sunlit air, so
           that it seems almost solid.",
  has  scenery;
```

The `^` symbol in `"sun^s"` means an apostrophe, so the word is "sun's". This object has been given the constant name `sunlight` because other parts of the 'Ruins' source code will need to refer to it later on. Being `scenery` means that the object is not only static but also not described by the game unless actually examined by the player. A perfectionist might add a `before` rule:

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

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

△ Objects having `scenery` are assumed to be mentioned in the `description` text of the room, just as the "shaft of sunlight" is mentioned in that of the Square Chamber. Giving an object `concealed` marks it as something which is present to a player who knows about it, but hidden from the casual eye. It will not be cited in lists of objects present in the room, and "take all" will not take it, but "take secret dossier", or whatever, will work. (Designers seldom need `concealed`, but the library uses it all the time, because the player-object is `concealed`.)

. . . . .

Some scenery must spread across several rooms. The 'Ruins', for instance, are misty, and although we *could* design them with a different "mist" object in every misty location, this would become tiresome. In 'Advent', for instance, a stream runs through seven locations, while mist which (we are told) is "frequently a sign of a deep pit leading down to water" can be found in ten different caves. Here is a better solution:

```
Object low_mist "low mist"
  with name 'low' 'swirling' 'mist',
       description "The mist has an aroma reminiscent of tortilla.",
       found_in  Square_Chamber  Forest,
       before [;
           Examine, Search: ;
           Smell: <<Examine self>>;
           default: "The mist is too insubstantial.";
       ],
  has  scenery;
```

The `found_in` property gives a list of places in which the mist is found: so far, just the Square Chamber and the Forest.

△ This allows for up to 32 misty locations. If scenery has to be visible even more widely than that, or if it has to change with circumstances (for instance, if the mist drifts) then it is simpler to give a routine instead of a list. This can look at the current `location` and say whether or not the object should be present, as in the following example from a game taking place at night:

```
Object Procyon "Procyon",
  with name 'procyon' 'alpha' 'canis' 'minoris' 'star',
       description "A double-star eleven light years distant.",
       found_in [;
           return (location ofclass OutsideRoom);
       ],
  has  scenery;
```

found_in is only consulted when the player's location changes, and works by moving objects around to give the illusion that they are in several places at once: thus, if the player walks from a dark field to a hilltop, Procyon will be moved ahead to the hilltop just in advance of the player's move. This illusion is good enough for most practical purposes, but sometimes needs a little extra work to maintain, for instance if the sky must suddenly cloud over, concealing the stars. Since it often happens that an object must be removed from all the places in which it would otherwise appear, an attribute called absent is provided which overrides found_in and declares that the object is found nowhere. Whatever change is made to found_in, or in giving or removing absent, the Inform library needs also to be notified that changes have taken place. For instance, if you need to occult Procyon behind the moon for a while, then:

```
give Procyon absent; MoveFloatingObjects();
```

The library routine MoveFloatingObjects keeps the books straight after a change of state of found_in or absent.

. . . . .

Whereas Procyon is entirely visual, some scenery items may afflict the other four senses. In 'Ruins', the Forest contains the rule:

```
before [;
    Listen: "Howler monkeys, bats, parrots, macaw.";
],
```

Besides which, we have already said that the mist smells of tortilla, which means that if the player types "smell" in a place where the mist is, there should clearly be some reaction. For this, a react_before rule attached to the mist is ideal:

```
react_before [;
    Smell: if (noun == nothing) <<Smell self>>;
],
```

This is called a "react" rule because the mist is reacting to the fact that a Smell action is taking place nearby. noun is compared with nothing to see if the player has indeed just typed "smell" and not, say, "smell crocus". Thus, when the action Smell takes place near the mist, it is converted into Smell low_mist, whereas the action Smell crocus would be left alone.

The five senses all have actions in Inform: Look, Listen, Smell, Taste and Touch. Of these, Look never has a noun attached (because Examine, LookUnder and Search are provided for close-up views), Smell and Listen may or may not have while Taste and Touch always have.

• **EXERCISE 6**
(Cf. 'Spellbreaker'.) Make an orange cloud descend on the player, which can't be seen through or walked out of.

. . . . .

Rooms also react to actions that might occur in them and have their own `before` and `after` rules. Here's one for the Square Chamber:

```
before [;
    Insert:
        if (noun == eggsac && second == sunlight) {
            remove eggsac; move stone_key to self;
            "You drop the eggsac into the glare of the shaft of
            sunlight. It bubbles obscenely, distends and then
            bursts into a hundred tiny insects which run in all
            directions into the darkness. Only spatters of slime
            and a curious yellow-stone key remain on the chamber
            floor.";
        }
],
```

(The variables `noun` and `second` hold the first and second nouns supplied with an action.) As it happens this rule could as easily have been part of the definition of the eggsac or the sunlight, but `before` and `after` rules for rooms are invaluable to code up geographical oddities.

• **EXERCISE 7**
Create a room for 'Ruins' called the Wormcast, which has the oddity that anything dropped there ends up back in the Square Chamber.

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

```
after [;
    Go: if (noun == w_obj)
        print "You feel an overwhelming sense of relief.^";
],
```

will print the message when its room is entered from the "west" direction. Note that this routine returns `false`, in the absence of any code telling it to do otherwise, which means that the usual game rules resume after the printing of the message.

● **REFERENCES**

'A Scenic View' by Richard Barnett demonstrates a system for providing examinable scenery much more concisely (without defining so many objects). ●`found_in` can allow a single object to represent many different but similar objects across a game, and a good example is found in Martin Braun's `"elevator.inf"` example game, where every floor of a building has an up-arrow and a down-arrow button to summon an elevator.

§9 Directions and the map

> I wisely started with a map, and made the story fit (generally with meticulous care for distances). The other way about lands one in confusions and impossibilities, and in any case it is weary work to compose a map from a story – as I fear you have found.
>
> — J. R. R. Tolkien (1892–1973), *to Naomi Mitchison, 25 April 1954*

'Ruins' so far contains two disconnected rooms. It is time to extend it into a modest map in which, as promised, the Square Chamber lies underneath the original Forest location. For the map of the finished game, see §23 below, but here is the beginning of the first level beneath ground, showing the Square Chamber and its two main side-chambers:

```
Square Chamber ↔ Wormcast
      ↕
   Corridor
      ↕
    Shrine
```

To make these map connections, we need to add:

```
u_to Forest, e_to Wormcast, s_to Corridor,
```

to the Square Chamber. This seems a good point to add two more map connections, or rather non-connections, to the Forest as well:

```
u_to "The trees are spiny and you'd cut your hands to ribbons
    trying to climb them.",
cant_go "The rainforest is dense, and you haven't hacked
    through it for days to abandon your discovery now. Really,
    you need a good few artifacts to take back to civilization
    before you can justify giving up the expedition.",
```

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

△ Here's how this is done. When the library wants to go in a certain direction, let's say "north", it sends the message `location.n_to()` and looks at the reply: it takes `false` to mean "Player can't go that way" and says so; `true` means "Player can't go that way, and I've already said why"; and any other value is taken as the destination.

• **EXERCISE 8**
Many early games have rooms with confused exits: 'Advent' has Bedquilt, 'Acheton' has a magnetic lodestone which throws the compass into confusion, 'Zork II' has a spinning carousel room and so on. Make the Wormcast room in 'Ruins' similarly bewildering.

. . . . .

For each of the twelve standard Inform directions there is a "direction property":

```
n_to     s_to     e_to     w_to     d_to     u_to
ne_to    nw_to    se_to    sw_to    in_to    out_to
```

Each direction also has a "direction object" to represent it in the game. For instance, `n_obj` is the object whose name is "north" and which the player invokes by typing "go north" or just "n". So there are normally twelve of these, too:

```
n_obj    s_obj    e_obj    w_obj    d_obj    u_obj
ne_obj   nw_obj   se_obj   sw_obj   in_obj   out_obj
```

Confusing the direction objects with the direction properties is easily done, but they are quite different. When the player types "go north", the action is `Go n_obj`, with `noun` being `n_obj`: only when this action has survived all possible `before` rules is the `n_to` value of the current location looked at.

△ The set of direction objects is not fixed: the current direction objects are the children of a special object called `compass`, and the game designer is free to add to or take from the current stock. Here for instance is the definition of "north" made by the library:

```
CompassDirection n_obj "north wall" compass
  with name 'n' 'north' 'wall', door_dir n_to;
```

`CompassDirection` is a class defined by the library for direction objects. `door_dir` is a property more usually seen in the context of doors (see §13) and here tells Inform which direction property corresponds to which direction object.

• △**EXERCISE 9**
In the first millennium A.D., the Maya peoples of the Yucatán Peninsula had 'world colours' white (*sac*), red (*chac*), yellow (*kan*) and black (*chikin*) for what we call the compass bearings north, east, south, west (for instance west is associated with 'sunset', hence black, the colour of night). Implement this.

• △**EXERCISE 10**
In Level 9's version of 'Advent', the magic word "xyzzy" was implemented as a thirteenth direction. How can this be done?

• △**EXERCISE 11**
(Cf. 'Trinity'.) How can the entire game map be suddenly east–west reflected?

• △△**EXERCISE 12**
Even when the map is reflected, there may be many room descriptions referring to "east" and "west" by name. Reflect these too.

• △△**EXERCISE 13**
Some designers find it a nuisance to have to keep specifying all map connections twice: once east from A to B, then a second time west from B to A, for instance. Write some code to go in the `Initialise` routine making all connections automatically two-way.

• **REFERENCES**
'Advent' has a very tangled-up map in places (see the mazes) and a well-constructed exterior of forest and valley giving an impression of space with remarkably few rooms. The mist object uses `found_in` to the full, and see also the stream (a single object representing every watercourse in the game). Bedquilt and the Swiss Cheese room offer classic confused-exit puzzles. •For a simple movement rule using `e_to`, see the Office in 'Toyshop'. •The opening location of Infocom's 'Moonmist' provides a good example of `cant_go` used to guide the player in a tactful way: "(The castle is south of here.)" •The library extension `"smartcantgo.h"` by David Wagner provides a system for automatically printing out "You can only go east and north."-style messages. •Ricardo Dague's `"cmap.h"` constructs maps of American-style cities. The same author's `"makemaze.inf"` prints out Inform source code for random rectangular mazes. •Nicholas Daley and Gunther Schmidl have each independently written a `"dirs.h"` providing a "dirs" or "exits" verb which lists available exits. Marnie Parker's `"dirsmap.h"` goes further by plotting up exits in map style or writing them out in English, at the player's discretion. •Brian D. Smith's example program `"spin.inf"` abolishes the convention that a player has an in-built ability to know which way is north: it replaces conventional compass directions with "left", "right", "ahead" and "back".

§10 Food and drink

Any object with the attribute `edible` can be eaten by the player, and the library provides the action `Eat` for this. If it succeeds, the library removes whatever was eaten from the object tree, so that it disappears from play (unless the designer has written any code which later moves it back again). Two edible objects have already appeared in 'Ruins': the mushroom, §4, and the honeycomb, §5.

Because drinking is a less clear-cut matter than eating, there is no attribute called `drinkable`. Although the library does provide the action `Drink`, you have to write your own code to show what happens when the object has been drunk. Here is an example where the object is entirely consumed:

```
Object -> "glass of milk"
  with name 'glass' 'of' 'milk',
       before [;
           Drink: remove self;
                "Well, that's the sixth stomach that hillside
                of grass has been in.";
       ];
```

Other rules which might be needed, instead of simply removing the object, would be to replace it with an ''empty glass'' object, or to change some counter which records how many sips have been taken from a large supply of drink.

• **REFERENCES**

Players discover almost at once that 'Advent' contains ''tasty food'', actually tripe although the text is too delicate to say so. (Type ''get tripe ration'' if you don't believe this.) Less well known is that the moss growing on the Soft Room's walls is also edible. •The product-placement parody 'Coke Is It' contains the occasional beverage. (''You are standing at the end of a road before a small brick building. Coca-Cola... Along The Highway To Anywhere, Around The Corner From Everywhere, Coca-Cola is The Best Friend Thirst Ever Had. A small stream flows out of the building and down a gully.'') •Inform doesn't provide an automatic system for handling liquids because it is difficult to find one that would satisfy enough people for enough of the time: for more on the issues that liquids raise, see §50. In its handling of liquids the source code for the alchemical mystery 'Christminster' is much borrowed from.

§11 Clothing

The player's possessions are normally thought of as being held in the hands, but there are two alternative possibilities: some items might be being carried indirectly in a bag or other container (see §12 next), while others might be worn as clothing. Wearing a hat is evidently different from carrying it. Some consequences are left to the designer to sort out, such as taking account of the hat keeping the rain off, but others are automatically provided: for instance, a request to "drop all" will not be taken as including the hat, and the hat will not count towards any maximum number of items allowed to be held in the player's hands. The library provides two actions for clothing: `Wear` and `Disrobe`. There is already an action called `Remove` for taking items out of containers, which is why the name `Remove` is not used for taking clothes off.

'Ruins' contains only one item of clothing, found resting on an altar in the Shrine: it summons a priest.

```
Treasure -> -> mask "jade mosaic face-mask"
  with description "How exquisite it would look in the Museum.",
       initial "Resting on the altar is a jade mosaic face-mask.",
       name 'jade' 'mosaic' 'face-mask' 'mask' 'face',
       cultural_value 10,
       after [;
           Wear: move priest to Shrine;
               if (location == Shrine)
                   "Looking through the obsidian eyeslits of the
                   mosaic mask, a ghostly presence reveals itself:
                   a mummified calendrical priest, attending your
                   word.";
           Disrobe: remove priest;
       ],
  has  clothing;
```

The attribute `clothing` signifies that the mask can be worn. During the time something is worn, it has the attribute `worn`. The library's standard rules ensure that:

> An object can only have `worn` if it is `in player`, that is, if it is an immediate possession of the player.

If you use `move` or `remove` to shift items of clothing in your own code, or give or take away the `worn` attribute, then you too should follow this principle.

△ A risk of providing clothing for the player is that it's hard to resist the lure of realism. A tweed jacket would add some colour to 'Ruins'. But if a jacket, why not plus-four trousers, an old pair of army boots and a hat with a mosquito net? And if trousers, why not underwear? What are the consequences if the player strips off? Too much of this kind of realism can throw a game off balance, so: no tweed.

• △△**EXERCISE 14**

Design a pair of white silk gloves, left and right, which are a single object called "pair of white gloves" until the player asks to do something which implies dividing them (typing "wear left glove", say, or "put right glove in drawer"). They should automatically rejoin into the pair as soon as they are together again. (By Richard Tucker. Hint: `react_before` and `before` rules are all you need.)

• **REFERENCES**

For designers who do want to go down the "if trousers, why not underwear?" road, Denis Moskowitz's library extension `"clothing.h"` models clothing by layer and by area of body covered. •For players who also want to, the road to take is 'I-0', by Adam Cadre.

§12 Containers, supporters and sub-objects

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

In almost every game, certain objects need to be thought of as on top of or inside other objects. The library provides actions `Insert` and `PutOn` for placing things inside or on top of something, and `Remove` for taking things out of or off the top of something. Many objects, such as house-bricks, cannot sensibly contain things, and a designer usually only wants certain specific items ever to have other things on top of them. In the model world, then, only objects which the designer has given the `container` attribute can contain things, and only those given the `supporter` attribute can have items on top.

The packing case brought by our archaeologist hero to the scene of the 'Ruins' (found in the opening location, the Forest) is a thoroughly typical `container`:

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

A container can hold anything up to 100 items, but this limit can be modified by giving the container a `capacity` property.

Note that the packing case is defined having an attribute called `open`. This is essential, because the library will only allow the player to put things in a `container` if it is currently `open`. (There is no attribute called `closed`, as any

container lacking open is considered closed.) If a container has openable, the player can open and close it at will, unless it also has locked. A locked object, whether it be a door or a container, cannot be opened. But if it has lockable then it can be locked or unlocked with the key object given in its with_key property. If with_key is undeclared or equal to nothing, then no key will fit, but this will not be told to the player. The actions Open, Close, Lock and Unlock handle all of this.

• **EXERCISE 15**
Construct a musical box with a silver key.

. . . . .

An object having supporter can have up to 100 items put on top of it, or, once again, its capacity value if one is given. An object cannot be both a container and a supporter at once, and there's no concept of being "open" or "locked" for supporters. Here is an example from the Shrine:

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

See §15 for enterable and its consequences.

. . . . .

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

```
Object -> chasm "horrifying chasm"
  with name 'blackness' 'chasm' 'pit' 'horrifying' 'bottomless',
       react_before [;
           Jump: <<Enter self>>;
           Go: if (noun == d_obj) <<Enter self>>;
       ],
       before [;
           Enter: deadflag = true;
              "You plummet through the silent void of darkness!";
           JumpOver: "It's far too wide.";
       ],
       after [;
           Receive: remove noun;
```

```
                print_ret (The) noun, " tumbles silently into the
                    darkness of the chasm.";
            Search: "The chasm is deep and murky.";
        ],
    has  scenery open container;
```

(Actually the definition will grow in §23, so that the chasm reacts to an eight-foot pumice-stone ball being rolled into it.) Note the use of an `after` rule for the `Search` action: this is because an attempt to "examine" or "look inside" the chasm will cause this action. `Search` means, in effect, "tell me what is inside the container" and the `after` rule prevents a message like "There is nothing inside the chasm." from misleading the player. Note also that the chasm 'steals' any stray `Jump` action in the vicinity using `react_before` and converts it into an early death.

• **EXERCISE 16**

Make an acquisitive bag which will swallow things up but refuses to disgorge them.

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

. . . . .

Not all containment is about carrying or supporting things. For instance, suppose a machine has four levers. If the machine is fixed in place somewhere, like a printing press, the levers could be provided as four further `static` objects. But if it is portable, like a sewing machine, we need to make sure that the levers always move whenever it moves, and vice versa. The natural solution is to make the lever-objects children of the machine-object, as though the machine were a container and the levers were its contents.

However, members of an object which isn't a `container` or `supporter` are normally assumed by the library to be hidden invisibly inside. In the case of the levers, this would defeat the point. We can get around this by giving the machine the `transparent` attribute, making the levers visible but not removable.

Containers can also be `transparent`, making their contents visible even when closed. The items on top of a `supporter` are of course always visible, and it makes no sense for a supporter to be `transparent`. See §26 for further details on when contents are listed in inventories and room descriptions.

• **EXERCISE 17**
Make a glass box and a steel box, which behave differently when a lamp is shut up inside them.

• **EXERCISE 18**
Make a television set with attached power button and screen.

• △**EXERCISE 19**
Implement a macramé bag hanging from the ceiling, inside which objects are visible, audible and so forth, but cannot be touched or manipulated in any way.

△ The most difficult case to handle is when an object needs to be portable, and to have sub-objects like lamps and buttons, *and* also to be a container in its own right. The solution to this will have to be left until the "scope addition" rules in §32, but briefly: an object's `add_to_scope` property may contain a list of sub-objects to be kept attached to it but which are not its children.

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

§13 Doors

> The happiness of seizing one of these tall barriers to a room by the porcelain knob of its belly; this quick hand-to-hand, during which progress slows for a moment, your eye opens up and your whole body adapts to its new apartment.
>
> — Francis Ponge (1899–1988), *Les plaisirs de la porte*

When is a `door` not a door? It might be a rope-bridge or a ladder, for instance. Inform provides doors for any situation in which some game object is intermediate between one place and another, and might on occasion become a barrier. Doors have a good deal in common with containers, in that they need to be `open` to allow access and to this end can also have `openable`, `lockable` or `locked`. Just as with containers, any key they have should be stored in the `with_key` property. The same actions `Open`, `Close`, `Lock` and `Unlock` all apply to doors just as they apply to containers. There are four steps in creating a new `door`:

(1) give the object the `door` attribute;
(2) set its `door_to` property to the location on the other side;
(3) set its `door_dir` property to the direction which that would be, such as `n_to`;
(4) make the location's map connection in that direction point to the door itself.

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

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

```
      with_key stone_key
  has  static door openable lockable locked;
```

Note that the door is `static` – otherwise the player could pick it up and walk away with it. (Experienced play-testers of Inform games try this every time, and usually come away with a door or two.) The properties `when_closed` and `when_open` give descriptions appropriate for the door in these two states.

A door is ordinarily only present on one side of a map connection. If a door needs to be accessible, say openable or lockable, from either side, then the standard trick is to make it present in both locations using `found_in` and to fix the `door_to` and `door_dir` to be the right way round for whichever side the player is on. Here, then, is a two-way door:

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

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

. . . . .

Puzzles more interesting than lock-and-key involve writing some code to intervene when the player tries to pass through. The interactive fiction literature has no shortage of doors which only a player with no possessions can pass through, for instance.

Care is required here because two different actions can make the player pass through the door. In the Corridor above, the player might type "s" or "go south", causing the action `Go s_obj`. Or might "enter stone door" or "go through door", causing `Enter StoneDoor`. Provided the door is actually open,

the `Enter` action then looks at the door's `door_dir` property, finds that the door faces south and generates the action `Go s_obj`. Thus, *provided that the door is open*, the outcome is the same and you need only write code to trap the `Go` action.

A neater alternative is to make the `door_to` property a routine. If a `door_to` routine returns `false` instead of a room, then the player is told that the door "leads nowhere", like the broken bridge of Avignon. If `door_to` returns `true`, then the library stops the action on the assumption that something has happened and the player has been told already.

• **EXERCISE 20**
Create a plank bridge across a chasm, which collapses if the player walks across it while carrying anything.

• **EXERCISE 21**
Create a locked door which turns out to be an immaterial illusion only when the player tries to walk through it in blind faith.

• **REFERENCES**
'Advent' is especially rich in two-way doors: the steel grate in the streambed, two bridges (one of crystal, the other of rickety wood) and a door with rusty hinges. See also the iron gate in 'Balances'. •The library extension `"doors.h"` by L. Ross Raszewski defines a class called `Connector` of two-way doors, which are slotted automatically into the map for convenience. Max Kalus's further extension `"doors2.h"` enables such doors to respond to, say, "the north door" from one side and "the south door" from the other.

§14 Switchable objects

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

A `switchable` object is one which can be switched off or on, usually because it has some obvious button, lever or switch on it. The object has the attribute `on` if it's on, and doesn't have it if it's off. (So there's no attribute called `off`, just as there's no attribute called `closed`.) The actions `SwitchOn` and `SwitchOff` allow the player to manipulate anything which is `switchable`. For example:

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

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

```
Object sodium_lamp "sodium lamp"
  with name 'sodium' 'lamp' 'heavy',
       describe [;
           if (self has on)
               "^The sodium lamp squats on the ground, burning away.";
           "^The sodium lamp squats heavily on the ground.";
       ],
       battery_power 100,
       before [;
           Examine: print "It is a heavy-duty archaeologist's lamp, ";
               if (self hasnt on) "currently off.";
               if (self.battery_power < 10) "glowing a dim yellow.";
               "blazing with brilliant yellow light.";
           Burn: <<SwitchOn self>>;
```

```
            SwitchOn:
                if (self.battery_power <= 0)
                    "Unfortunately, the battery seems to be dead.";
                if (parent(self) hasnt supporter
                    && self notin location)
                    "The lamp must be securely placed before being
                    lit.";
            Take, Remove:
                if (self has on)
                    "The bulb's too delicate and the metal handle's too
                    hot to lift the lamp while it's switched on.";
        ],
        after [;
            SwitchOn: give self light;
            SwitchOff: give self ~light;
        ],
    has  switchable;
```

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

△ The reader may be wondering why the lamp needs to use a `describe` routine to give itself a description varying with its condition: why not simply write the following?

```
when_off "The sodium lamp squats heavily on the ground.",
when_on "The sodium lamp squats on the ground, burning away.",
```

The answer is that `when_on` and `when_off` properties, like `initial`, only apply until an object has been held by the player, after which it is normally given only a perfunctory mention in room descriptions. "You can also see a sodium lamp here." As the `describe` property has priority over the whole business of how objects are described in room descriptions, the above ensures that the full message always appears even if the object has become old and familiar. For much more on room descriptions, see §26.

● **REFERENCES**

The original switchable object was the brass lamp from 'Advent', which even provides verbs "on" and "off" to switch it. ●Jayson Smith's library extension `"links.h"` imitates a set of gadgets found in Andrew Plotkin's game 'Spider and Web'. In this scheme, "linkable" machines only work when linked to "actuators", which are switches of different kinds (remote controls, attachable push-buttons and so on).

§15 Things to enter, travel in and push around

> Vehicles were objects that became, in effect, mobile rooms... The code for the boat itself was not designed to function outside the river section, but nothing kept the player from carrying the deflated boat to the reservoir and trying to sail across...
>
> — Tim Anderson, *The History of Zork*

Quite so. Even the case of an entirely static object which can be climbed into or onto poses problems of realism. Sitting on a garden roller, is one in the gardens, or not? Can one reasonably reach to pick up a leaf on the ground? The Inform library leaves most of these subtleties to the designer but has at least a general concept of "enterable object". These have `enterable` and the `Enter` and `Exit` actions allow the player to get in (or on) and out (or off) of them.

Enterable items might include, say, an open-topped car, a psychiatrist's couch or even a set of manacles attached to a dungeon wall. In practice, though, manacles are an exceptional case, and one usually wants to make an `enterable` thing also a `container`, or – as in the case of the altar from 'Ruins' which appeared in the previous section – a `supporter`:

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

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

Sitting on furniture, one is hardly in a different location altogether. But suppose the player climbs into a `container` which is not `transparent` and then closes it from the inside? To all intents and purposes this has become another room. The interior may be dark, but if there's light to see by, the player will want to see some kind of room description. In any case, many enterable objects ought to look different from inside or on top. Inside a vehicle, a player might be able to see a steering wheel and a dashboard, for instance. On top of a cupboard, it might be possible to see through a skylight window.

For this purpose, any `enterable` object can provide a property called `inside_description`, which can hold a string of text or else a routine to print some text, as usual. If the exterior location is still visible, then the "inside description" is appended to the normal room description; if not, it replaces the

room description altogether. As an extreme example, suppose that the player gets into a huge cupboard, closes the door and then gets into a plastic cabinet inside that. The resulting room description might read like so:

> *The huge cupboard (in the plastic cabinet)*
> It's a snug little cupboard in here, almost a room in itself.
> In the huge cupboard you can see a pile of clothes.
> The plastic walls of the cabinet distort the view.

The second line is the `inside_description` for the huge cupboard, and the fourth is that for the plastic cabinet.

• **EXERCISE 22**
(Also from 'Ruins'.) Implement a cage which can be opened, closed and entered.

. . . . .

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

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

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

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

If you want to move the vehicle in your own code, return 3, not 2: otherwise the old location may be restored by subsequent workings. Notice that if you write no code, the default value `false` will always be returned, so enterable objects won't become vehicular unless you write them that way.

△ Because you might want to drive the car "out" of a garage, the "out" verb does not make the player get out of the car. Instead the player generally has to type something like "get out" or "exit" to make this happen.

• **EXERCISE 23**
Alter the car so that it will only drive along roads, and not through all map connections.

. . . . .

Objects like the car or, say, an antiquated wireless on casters, are too heavy to pick up but the player should at least be able to push them from place to place. When the player tries to do this, an action like `PushDir wireless` is generated.

Now, if the `before` routine for the wireless returns `false`, the game will just say that the player can't move the wireless; and if it returns `true`, the game will do nothing at all, assuming that the `before` routine has already printed something more interesting. So how does one actually tell Inform that the push should be allowed? The answer is: first call the `AllowPushDir` routine (a library routine), and then return `true`. For example ('Ruins' again), here is a ball on a north-south corridor which slopes upward at the northern end:

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

```
      ],
  has  static;
```

• △**EXERCISE 24**

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

• **REFERENCES**

For an `enterable supporter` puzzle, see the magic carpet in 'Balances' (and several items in 'Alice Through the Looking-Glass'). •When a vehicle has a sealed interior large enough to be a location, it is probably best handled as a location with changing map connections and not as a `vehicle` object moved from room to room. See for instance Martin Braun's `"elevator.inf"` example game, providing an elevator which serves eight floors of a building.

§16 Reading matter and consultation

> Making books is a skilled trade, like making clocks.
> — Jean de la Bruyère (1645-1696)

"Look up figure 18 in the engineering textbook" is a difficult line for Inform to understand, because almost anything could appear in the first part: even its format depends on what the second part is. This kind of request, and more generally

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

cause the `Consult` action. In such cases, the `noun` is the book and there is no `second` object. Instead, the object has to parse the ⟨any words here⟩ part itself. The following variables are set up to make this possible:

`consult_from` holds the number of the first word in the ⟨any...⟩ clause;
`consult_words` holds the number of words in the ⟨any...⟩ clause.

The ⟨any words here⟩ clause must contain at least one word. The words given can be parsed using library routines like `NextWord()`, `TryNumber(word-number)` and so on: see §28 for full details. As usual, the `before` routine should return `true` if it has managed to deal with the action; returning `false` will make the library print "You discover nothing of interest in...".

Little hints are placed here and there in the 'Ruins', written in the glyphs of a not altogether authentic dialect of Mayan. Our explorer has, naturally, come equipped with the latest and finest scholarship on the subject:

```
Object dictionary "Waldeck's Mayan dictionary"
  with name 'dictionary' 'local' 'guide' 'book' 'mayan'
            'waldeck' 'waldeck^s',
       description "Compiled from the unreliable lithographs of the
           legendary raconteur and explorer ~Count~ Jean Frederic
           Maximilien Waldeck (1766??-1875), this guide contains
           what little is known of the glyphs used in the local
           ancient dialect.",
       correct false,
       before [ w1 w2 glyph;
         Consult:
```

```
            wn = consult_from;
            w1 = NextWord(); ! First word of subject
            w2 = NextWord(); ! Second word (if any) of subject
            if (consult_words==1 && w1~='glyph' or 'glyphs') glyph = w1;
            else if (consult_words==2 && w1=='glyph') glyph = w2;
            else if (consult_words==2 && w2=='glyph') glyph = w1;
            else "Try ~look up <name of glyph> in book~.";
            switch (glyph) {
                'q1': "(This is one glyph you have memorised!)^^
                     Q1: ~sacred site~.";
                'crescent': "Crescent: believed pronounced ~xibalba~,
                    though its meaning is unknown.";
                'arrow': "Arrow: ~journey; becoming~.";
                'skull': "Skull: ~death, doom; fate (not nec. bad)~.";
                'circle': "Circle: ~the Sun; also life, lifetime~.";
                'jaguar': "Jaguar: ~lord~.";
                'monkey': "Monkey: ~priest?~.";
                'bird': if (self.correct) "Bird: ~dead as a stone~.";
                     "Bird: ~rich, affluent?~.";
                default: "That glyph is so far unrecorded.";
            }
        ],
    has  proper;
```

Note that this understands any of the forms "q1", "glyph q1" or "q1 glyph". (These aren't genuine Maya glyphs, but some of the real ones once had similar names, dating from when their syllabic equivalents weren't known.)

• △△**EXERCISE 25**

To mark the 505th anniversary of William Tyndale, the first English translator of the New Testament (who was born some time around 1495 and burned as a heretic in Vilvorde, Denmark, in 1535), prepare an Inform edition.

. . . . .

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

```
Attribute legible;
...
Object textbook "textbook"
  with name 'engineering' 'textbook' 'text' 'book',
       description "What beautiful covers and spine!",
```

```
        before [;
            Consult, Read:
                "The pages are full of senseless equations.";
        ],
        has  legible;
...
[ ReadSub; <<Examine noun>>; ];
Extend 'read' first * legible -> Read;
```

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

● **REFERENCES**

Another possibility for parsing commands like "look up ⟨something⟩ in the catalogue", where any object name might appear as the ⟨something⟩, would be to extend the grammar for "look". See §30.

§17 People and animals

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

Living creatures should be given the attribute `animate` so that the library knows such an object can be talked to, given things, woken from sleep and so on. When the player treats an `animate` object as living in this way, the library calls upon that object's `life` property. This looks like `before` or `after`, but only applies to the following actions:

`Attack` The player is making hostile advances...

`Kiss` ... or amorous ones...

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

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

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

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

`Ask` The player asked about something. Just as with a "consult" topic (see §16 above), the variables `consult_from` and `consult_words` are set up to indicate which words the object might like to think about. (In addition, `second` holds the dictionary value for the first word which isn't `'the'`, but this is much cruder.)

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

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

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

If the `life` rule isn't given, or returns `false`, events take their usual course. `life` rules vary dramatically in size. The coiled snake from 'Balances' shows that even the tiniest `life` routine can be adequate for an animal:

```
Object -> snake "hissing snake"
  with name 'hissing' 'snake',
       initial "Tightly coiled at the edge of the chasm is a
           hissing snake.",
       life "The snake hisses angrily!",
  has  animate;
```

It's far from unknown for people in interactive fiction to be almost as simplistic as that, but in most games even relatively passive characters have some ability to speak or react. Here is the funerary priest standing in the 'Ruins' Shrine:

```
Object priest "mummified priest"
  with name 'mummified' 'priest',
       description
          "He is desiccated and hangs together only by will-power.
           Though his first language is presumably local Mayan,
           you have the curious instinct that he will understand
           your speech.",
       initial "Behind the slab, a mummified priest stands waiting,
           barely alive at best, impossibly venerable.",
       life [;
           Answer: "The priest coughs, and almost falls apart.";
           Ask: switch (second) {
                   'dictionary', 'book':
                        if (dictionary.correct == false)
                            "~The ~bird~ glyph... very funny.~";
                        "~A dictionary? Really?~";
                   'glyph', 'glyphs', 'mayan', 'dialect':
                        "~In our culture, the Priests are ever
                        literate.~";
                   'lord', 'tomb', 'shrine', 'temple':
                        "~This is a private matter.~";
                   'ruins': "~The ruins will ever defeat thieves.
                        In the underworld, looters are tortured
                        throughout eternity.~ A pause. ~As are
                        archaeologists.~";
```

```
                'web', 'wormcast':
                    "~No man can pass the Wormcast.~";
                'xibalba': if (Shrine.sw_to == Junction)
                        "The priest shakes his bony finger.";
                    Shrine.sw_to = Junction;
                    "The priest extends one bony finger
                    southwest toward the icicles, which
                    vanish like frost as he speaks.
                    ~Xibalb@'a, the Underworld.~";
            }
            "~You must find your own answer.~";
        Tell: "The priest has no interest in your sordid life.";
        Attack, Kiss:  remove self;
            "The priest desiccates away into dust until nothing
            remains, not a breeze nor a bone.";
        ThrowAt: move noun to location; <<Attack self>>;
        Show, Give:
            if (noun == dictionary && dictionary.correct == false) {
                dictionary.correct = true;
                "The priest reads a little of the book, laughing
                in a hollow, whispering way. Unable to restrain
                his mirth, he scratches in a correction somewhere
                before returning the book.";
            }
            "The priest is not interested in earthly things.";
    ],
has  animate;
```

The Priest only stands and waits, but some characters need to move around, or to appear and reappear throughout a game, changing in their responses and what they know. This makes for a verbose object definition full of cross-references to items and places scattered across the source code. An alternative is to use different objects to represent the character at different times or places: in 'Jigsaw', for instance, the person called ''Black'' is seven different objects.

. . . . .

Animate objects representing people with proper names, like ''Mark Antony'', need to be given the `proper` attribute, and those with feminine names, such as ''Cleopatra'', need to be both `female` and `proper`, though of course history would have been very different if... Inanimate objects sometimes have proper names, too: Waldeck's Mayan dictionary in §16 was given `proper`. See §26 for more on naming.

. . . . .

Some objects are not alive as such, but can still be spoken to: microphones, tape recorders and so on. It would be a nuisance to implement these as `animate`, since they have none of the other characteristics of life. Instead, they can be given just the attribute `talkable`, making them responsive only to conversation. They have a `life` property to handle `Answer` and so on, but it will never be asked to deal with, for instance, `Kiss`. Talkable objects can also receive orders: see the next section.

. . . . .

Designers often imagine animate objects as being altogether different from things, so it's worth noting that all the usual Inform rules apply equally well to the living. An `animate` object still has `before` and `after` routines like any other, so the short list of possible `life` rules is not as restrictive as it appears. Animate objects can also `react_before` and `react_after`, and it's here that these properties really come into their own:

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

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

Animate objects often have possessions as part of the game design. Two examples, both from 'The Lurking Horror':

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

Recall from §12 that the child-objects of an object which isn't a `container` or `supporter` are outwardly visible only if the object has the `transparent` attribute. Here, the hacker should have `transparent` and the urchin not. The parser then prevents the player from referring to whatever the urchin is hiding, even if the player has played the game before and knows what is in there.

• **EXERCISE 26**
Arrange for a bearded psychiatrist to place the player under observation, occasionally mumbling insights such as "Subject puts green cone on table. Interesting."

§18 Making conversation

To listen is far harder than to speak. This section overlaps with Chapter IV, the chapter on parsing text, and the later exercises are among the hardest in the book. As the following summary table shows, the simpler ways for the player to speak to people were covered in the previous section: this section is about "orders".

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

Here we're supposing that the game's dictionary includes "troll", "orc" and so forth, but not "wyvern", which is why "ask orc about wyvern" results in the action `Ask orc 0`. The notation `NotU...` is an abbreviation for `NotUnderstood`, of which more later. The two numbers in the "consult" column are the values of `consult_from` and `consult_words`, in cases where they are set.

. . . . .

When the player types in something like "pilot, fly south", addressing an object which has `animate` or at least `talkable`, the result is called an 'order'.

The order is sent to the pilot's `orders` property, which may if it wishes comply or react in some other way. Otherwise, the standard game rules will simply print something like "The pilot has better things to do." The 'Ruins' priest is especially unhelpful:

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

The `NotUnderstood` clause of an `orders` rule is run when the parser couldn't understand what the player typed: e.g., "pilot, fly somersaults".

△ The Inform library regards the words "yes" and "no" as being verbs, so it parses "delores, yes" into a `Yes` order. This can be a slight nuisance, as "say yes to delores" is treated differently: it gets routed through the `life` routine as an `Answer`.

△ When a `NotUnderstood` order is being passed to `orders`, the library sets up some variables to help you parse by hand if you need to. The actual order, say "fly somersaults", becomes a sort of consultation topic, with `consult_from` and `consult_words` set to the first word number and the number of words. The variable `etype` holds the parser error that would have been printed out, had it been a command by the player himself. See §33: for instance, the value `CANTSEE_PE` would mean "the pilot can't see any such object".

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

• **EXERCISE 27**
(Cf. 'Starcross'.) Construct a computer responding to "computer, theta is 180".

• **EXERCISE 28**
For many designers, `Answer` and `Tell` are just too much trouble. How can you make attempts to use these produce a message saying "To talk to someone, try 'someone, something'."?

. . . . .

When the player issues a request to an `animate` or `talkable` object, they're normally parsed in the standard way. "avon, take the bracelet" results in the order `Take bracelet` being sent to Kerr Avon, just as typing "take the bracelet" results in the action `Take bracelet` passing to the player. The range of text understood is the same, whether or not the person addressed is Avon. Sometimes, though, one would rather that different people understood entirely different grammars.

For instance, consider Zen, the flight computer of an alien spacecraft. It's inappropriate to tell Zen to pick up a teleport bracelet and the crew tend to give commands more like:

"Zen, set course for Centauro"
"Zen, speed standard by six"
"Zen, scan 360 orbital"
"Zen, raise the force wall"

"Zen, clear the neutron blasters for firing"

For such commands, an `animate` or `talkable` object can if it likes provide a `grammar` property. This is called at a time when the parser has worked out the object being addressed and has set the variables `verb_wordnum` and `verb_word` to the word number of the 'verb' and its dictionary entry, respectively. For example, in "orac, operate the teleport" `verb_wordnum` would be 3, because the comma counts as a word on its own, and `verb_word` would be `'operate'`.

Once called, the `grammar` routine can reply to the parser by returning:

`false` Meaning "carry on as usual".

`true` Meaning "you can stop parsing now because I have done it all, and put the resulting order into the variables `action`, `noun` and `second`".

`'verb'` Meaning "don't use the standard game grammar: use the grammar lines for this verb instead".

`-'verb'` Meaning "use the grammar lines for this verb, and if none of them match, use the standard game grammar as usual".

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

• △**EXERCISE 29**
Implement Charlotte, a little girl who's playing Simon Says (a game in which she only follows your instructions if you remember to say "Simon says" in front of them: so she'll disobey "charlotte, wave" but obey "charlotte, simon says wave").

• △**EXERCISE 30**
Another of Charlotte's rules is that if you say a number, she has to clap that many times. Can you play?

• △**EXERCISE 31**
Regrettably, Dyslexic Dan has always mixed up the words "take" and "drop". Implement him anyway.

. . . . .

△ When devising unusual grammars, you sometimes want to define grammar lines that the player can only use when talking to other people. The vile trick to achieve this is to attach these grammar lines to an "untypeable verb", such as `'comp,'`. This can never match what the player typed because the parser automatically separates the text "comp," into two words, "comp" and ",", with a space between them. The same will happen with any word of up to 7 letters followed by a comma or full stop. For instance, here's one way to solve the 'Starcross' computer exercise, using an untypeable verb:

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

(Here, `Control` is a "general parsing routine": see §31.) The computer itself then needs these properties:

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

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

• △△**EXERCISE 32**
How can you make a grammar extension to an ordinary verb that will apply only to Dan?

• △**EXERCISE 33**
Make an alarm clock responding to "alarm, off", "alarm, on" and "alarm, half past seven" (the latter to set its alarm time).

• △**EXERCISE 34**
Implement a tricorder (from Star Trek) which analyses nearby objects on a request like "tricorder, the quartz stratum".

• △**EXERCISE 35**
And, for good measure, a replicator responding to commands like "replicator, tea earl grey" and "replicator, aldebaran brandy".

• △△**EXERCISE 36**
And a communications badge in contact with the ship's computer, which answers questions like "computer, where is Admiral Blank". (This is best done with "scope hacking", for which see §32.)

• △△**EXERCISE 37**
Finally, construct the formidable flight computer Zen. (Likewise.)

△△ To trump one vile trick with another, untypeable verbs are also sometimes used to create what might be called 'fake fake actions'. Recall that a fake action is one which is never generated by the parser, and has no action routine. For instance, there's no `ThrownAtSub`, because `ThrownAt` is a fake. A fake fake action is a half-measure: it's a full action in every respect, including having an action routine, except that it can never be generated by the parser. The following grammar line creates three of them, called `Prepare`, `Simmer` and `Cook`:

```
Verb 'fakes.' * -> Prepare * -> Simmer * -> Cook;
```

The author is indebted for this terminology to an algebraic geometry seminar by Peter Kronheimer on fake and fake fake K3 surfaces.

. . . . .

△ Difficult "someone on the other end of a phone" situations turn up quite often in one form or another (see, for instance, the opening scene of 'Seastalker') and often a quite simple solution is fine. If you just want to make something like "michael, tell me about the crystals" work, when Michael is at the other end of the line, give the phone the `talkable` attribute and make the word `'michael'` one of its names. If several people are on the phone at different times, you can always give the phone a `parse_name` property (see §28) to respond to different names at different times.

• △**EXERCISE 38**
Via the main screen of the Starship Enterprise, Captain Jean-Luc Picard wants to see and talk to Noslen Maharg, the notorious tyrant, who is down on the planet Mrofni. Make it so.

• △△**EXERCISE 39**
Put the player in telepathic contact with Martha, who is in a sealed room some distance away, but who has a talent for telekinesis. Martha should respond to "martha, look", "ask martha about...", "say yes to martha", "martha, give me the red ball" and the like.

• **REFERENCES**
A much fuller example of a 'non-player character' is given in the example game 'The Thief', by Gareth Rees (though it's really an implementation of the gentleman in 'Zork I', himself an imitation of the pirate in 'Advent'). The thief is capable of walking around, being followed, stealing things, picking locks, opening doors and so on. •Other good definitions of `animate` objects to look at are Christopher in 'Toyshop', who will stack up building blocks on request; the kittens in 'Alice Through the Looking-Glass'; the barker in 'Balances', and the animals and dwarves of 'Advent'. •Following

people means being able to refer to them after they've left the room: see the library extension `"follower.h"` by Gareth Rees, Andrew Clover and Neil James Brown. •A wandering character with a destination to aim for needs to be able to navigate from room to room, and possibly through doors. Ideally, a designer should be able to make a simple instruction like "head for the West Ballroom" without specifying any route. Two independent library extensions allow this: `"MoveClass.h"`, by Neil James Brown and Alan Trewartha, is compatible with `"follower.h"` and is especially strong on handling doors. Volker Lanz's `"NPCEngine"` is designed for what might be called detective-mystery situations, in which the people inside a country house are behaving independently in ways which must frequently be described to the player. •Irene Callaci's `"AskTellOrder.h"` library extension file automatically handles commands in the form "ask/tell someone to do something".

§19 The light and the dark

```
>examine darkness
```
You can't see the darkness without a light!
```
>let there be light
```
Okay, there is light.
```
>examine the light
```
It is good.
```
>divide the light from the darkness
```
It is so.
```
>call the light "day" then call the darkness "night"
```
Called.
Called.

– from a transcript of 'The Creation', a game never written but proposed in some of Infocom's surviving documents. (''Estimated development time 8-10 months... shalts and begats and haths.'')

Sighted people observe whether it's light or dark so instantly that the matter seems self-evident. The Inform library has to use reason instead, and it rechecks this reasoning very frequently, because almost any change in the world model can affect the light:

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

The designer of an Inform game isn't allowed to tell the library ''the player is now in darkness'', because this would soon lead to inconsistencies. (If you want it to be dark, ensure that there are no light sources nearby.) Because light is automatically calculated, you can write statements like the following, and leave the library to sort out the consequences:

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

The `light` attribute means that an object is giving off light, or that a room is currently lit, for instance because it is outdoors in day-time.

• **EXERCISE 40**
Abolish darkness altogether, without having to give every location `light`.

. . . . .

When the player is in darkness, the current `location` becomes `thedark`, a special object which behaves like a room and has the short name "Darkness". Instead, the variable `real_location` always contains the actual room occupied, regardless of the light level.

The designer can "customise" the darkness in a game by altering its `initial`, `description` or `short_name` properties. For example, the `Initialise` routine of the game might include:

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

See §20 for how 'Ruins' makes darkness menacing.

. . . . .

Light is reconsidered at the start of the game, after any movement of the player, after any change of player, and at the end of each turn regardless. The presence or absence of light affects the `Look`, `Search`, `LookUnder` and `Examine` actions, and, since this is a common puzzle, also the `Go` action: you can provide a routine called

```
DarkToDark()
```

and if you do then it will be called when the player goes from one dark place to another. (It's called just before the room description for the new dark room, normally "Darkness", is printed). You could then take the opportunity to kill the player off or extract some other forfeit. If you provide no such routine, then the player can move about as freely in the darkness as in the light.

. . . . .

△△ *Darkness rules.* Here is the full definition of "when there is light". Remember that the parent of the player object may not be a room: it may be, say, a red car whose parent is a large closed cardboard box whose parent is a room.

(1) There is light exactly when the parent of the player 'offers light'.
(2) An object is see-through if:
    (a) it is `transparent`, *or*
    (b) it is a `supporter`, *or*
    (c) it is a `container` which is `open`, *or*
    (d) it is `enterable` but not a `container`.
(3) An object offers light if:
    (a) it itself has the `light` attribute set, *or*
    (b) any of its immediate possessions have light, *or*
    (c) it is see-through and its parent offers light.
(4) An object has light if:
    (a) it itself has the `light` attribute set, *or*
    (b) it is see-through and any of its immediate possessions have light, *or*
    (c) any object it places in scope using the property `add_to_scope` has light.

It may help to note that to "offer light" is to cast light inward, that is, down the object tree, whereas to "have light" is to cast light outward, that is, up the object tree. The library routines `IsSeeThrough(obj)`, `OffersLight(obj)` and `HasLightSource(obj)` check conditions (2) to (4), returning `true` or `false` as appropriate.

• **EXERCISE 41**

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

• **REFERENCES**

For a `DarkToDark` routine which discourages wandering about caves in the dark, see 'Advent'. •It is notoriously tricky to handle the gradual falling of night or a gradual change of visibility. See §51.

§20 Daemons and the passing of time

> Some, such as Sleep and Love, were never human. From this class an individual daemon is allotted to each human being as his 'witness and guardian' through life.
>
> — C. S. Lewis (1898–1963), *The Discarded Image*

> A great Daemon... Through him subsist all divination, and the science of sacred things as it relates to sacrifices, and expiations, and disenchantments, and prophecy, and magic... he who is wise in the science of this intercourse is supremely happy...
>
> — Plato (c.427–347 B.C.), *The Symposium*, in the translation by Percy Bysshe Shelley (1792–1822)

To medieval philosophers, daemons were the intermediaries of God, hovering invisibly over the world and interfering with it. They may be guardian spirits of places or people. So also with Inform: a daemon is a meddling spirit, associated with a particular game object, which gets a chance to interfere once per turn while it is 'active'. 'Advent' has five: one to deplete the lamp's batteries, three to move the bear, the pirate and the threatening little dwarves and one to close the cave when all the treasures have been collected. Though there isn't much to say about daemons, they are immensely useful, and there are some rule-based design systems for interactive fiction in which the daemon is a more fundamental concept than the object. (The early 1980s system by Scott Adams, for instance.)

The daemon attached to an object is its `daemon` routine, if one is given. However, a daemon is normally inactive, and must be explicitly activated and deactivated using the library routines

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

Daemons are often started by a game's `Initialise` routine and sometimes remain active throughout. When active, the `daemon` property of the object is called at the end of each turn, regardless of where that object is or what the circumstances, provided only that the player is still alive. This makes daemons useful for 'tidying-up operations', putting rooms back in order after the player has moved on, or for the consequences of actions to catch up with the player.

• △**EXERCISE 42**
Many games contain "wandering monsters", characters who walk around the map. Use a daemon to implement one who wanders as freely as the player, like the gentleman thief in 'Zork'.

• △**EXERCISE 43**
Use a background daemon to implement a system of weights, so that the player can only carry a certain weight before strength gives out and something must be dropped. It should allow for feathers to be lighter than lawn-mowers.

. . . . .

It's also possible to attach a timer to an object. (In other design languages, timers are called "fuses".) To set up a timer, you need to give an object two properties: `time_left` and `time_out`. Like daemons, timers are inactive until explicitly started:

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

will set `object.time_left` to `time`. This value will be reduced by 1 each turn, except that if this would make it negative, the Inform library instead sends the message

```
object.time_out()
```

once and once only, after which the timer is deactivated again. You're free to alter `time_left` yourself: a value of 0 means "will go off at the end of the present turn", so setting `time_left` to 0 triggers immediate activation. You can also deactivate the timer, so that it never goes off, by calling

```
StopTimer(object);
```

• **EXERCISE 44**
Construct an egg-timer which runs for three turns.

△ At most 32 timers or daemons can be active at the same time, together with any number of inactive ones. This limit of 32 is easily raised, though: just define the constant `MAX_TIMERS` to some larger value, putting the definition in your code before `"Parser.h"` is included.

. . . . .

There is yet a third form of timed event. If a room provides an `each_turn` routine, then the library will send the message

```
location.each_turn()
```

at the end of every turn when the player is present. Similarly, for every object `O` which is near the player and provides `each_turn`:

```
O.each_turn()
```

will be sent every turn. This would be one way to code the sword of 'Zork', for instance, which begins to glow when monsters are nearby. `each_turn` is also convenient to run creatures which stay in one place and are only active when the player is nearby. An ogre with limited patience can therefore have an `each_turn` routine which worries the player (''The ogre stamps his feet angrily!'' and so forth) while also having a timer set to go off when patience runs out.

△ ''Near the player'' actually means ''in scope'', a term which will be properly defined in §32 but which roughly translates as ''in the same place and visible''. You can change the scope rules using an `InScope` routine, say to make the 'Zork I' thief audible throughout the maze he is wandering around in. In case you want to tell whether scope is being worked out for ordinary parsing reasons or instead for `each_turn` processing, look to see whether the `scope_reason` variable has the value `EACHTURN_REASON`. (Again, see §32 for more.)

△ It is safe to move an object when its own `each_turn` rule is running, but not to move any other objects which are likely to be in scope.

• **EXERCISE 45**
('Ruins'.) Make ''the sound of scuttling claws'' approach in darkness and, after 4 consecutive turns in darkness, kill the player.

• △**EXERCISE 46**
Now try implementing the scuttling claws in a single object definition, with no associated code anywhere else in the program, not even a line in `Initialise`, and without running its `daemon` all the time.

. . . . .

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

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

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

```
Statusline time;
```

as a directive at the start of your source code.

• **EXERCISE 47**
How could you make your game take notice of the time passing midnight, so that the day of the week could be nudged on?

• △**EXERCISE 48**
Make the lighting throughout the game change at sunrise and sunset.

. . . . .

△ Here is exactly what happens at the end of each turn. The sequence is abandoned if at any stage the player dies or wins.

(1) The turns counter is incremented.
(2) The 24-hour clock is moved on.
(3) Daemons and timers are run (in no guaranteed order).
(4) `each_turn` takes place for the current room, and then for every object in scope.
(5) An entry point called `TimePasses` is called, if the game provides such a routine.
(6) Light is re-considered (see §19).
(7) Any items the player now holds which have not previously been held are given the `moved` attribute, and score is awarded if appropriate (see §22).

• △**EXERCISE 49**
Suppose the player is magically suspended in mid-air, but that anything let go of will fall out of sight. The natural way to code this is to use a daemon which gets rid of anything it finds on the floor: this is better than trapping `Drop` actions because objects might end up on the floor in many different ways. Why is `each_turn` better still?

• **EXERCISE 50**
How would a game work if it involved a month-long archaeological dig, where anything from days to minutes pass between successive game turns?

• **REFERENCES**

Daemons abound in most games. Apart from 'Advent', see the flying tortoise from 'Balances' and the chiggers from 'Adventureland'. For more ingenious uses of `daemon`, see the helium balloon and the matchbook from 'Toyshop'. •Typical timers include the burning match and the hand grenade from 'Toyshop', the endgame timer from 'Advent' and the 'Balances' cyclops (also employing `each_turn`). •'Adventureland' makes much use of `each_turn`: see the golden fish, the mud, the dragon and the bees. •The chapter of 'Jigsaw' set on the Moon runs the clock at rate −28, to allow for the length of the lunar day. •The library extension `"timewait.h"` by Andrew Clover thoroughly implements time of day, allowing the player to ''wait until quarter past three''. •Whereas Erik Hetzner's `"printtime.h"` does just the reverse: it prints out Inform's numerical values of time in the form of text like ''half past seven''. Erik is also author of `"timepiece.h"`, which models watches and clocks, allowing them to run slow or fast compared to the library's absolute notion of time. (As yet nobody has needed a relativistic world model.)

§21 Starting, moving, changing and killing the player

> Life's but a walking shadow, a poor player
> That struts and frets his hour upon the stage
> And then is heard no more; it is a tale
> Told by an idiot, full of sound and fury,
> Signifying nothing.
>
> — William Shakespeare (1564–1616), *Macbeth* V v

To recap on §4, an "entry point routine" is one provided by your own source code which the library may call from time to time. There are about twenty of these, listed in §A5, and all of them are optional but one: `Initialise`. This routine is called before any text of the game is printed, and it *can* do many things: start timers and daemons, set the time of day, equip the player with possessions, make any random settings needed and so on. It *usually* prints out some welcoming text, though not the name and author of the game, because that appears soon after when the "game banner" is printed. The only thing it *must* do is to set the `location` variable to where the player begins.

This is usually a room, possibly in darkness, but might instead be an `enterable` object inside a room, such as a chair or a bed. Like medieval romance epics, interactive fiction games often start by waking the player from sleep, sometimes by way of a dream sequence. If your game begins with verbose instructions before the first opportunity for a player to type a command, you may want to offer the chance to restore a saved game at once:

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

To equip the player with possessions, simply `move` the relevant objects to `player`.

The return value from `Initialise` is ordinarily ignored, whether `true` or `false`, and the library goes on to print the game banner. If, however, you return 2, the game banner is suppressed for now. This feature is provided for games like 'Sorcerer' and 'Seastalker' which play out a short prelude first. If you do suppress the banner from `Initialise`, you should print it no more than a few turns later on by calling the library routine `Banner`. The banner is familiar to players, reassuringly traditional and useful when testing, because it identifies which version of a game is giving trouble. Like an imprint page with an ISBN, it is invaluable to bibliographers and collectors of story files.

. . . . .

'Ruins' opens in classical fashion:

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

For the source code of the 'Ruins' `TitlePage` routine, see the exercises in §42.

. . . . .

The question "where is the player?" can be answered in three different ways. Looking at `parent(player)` tells you the object immediately containing the player, which can be a location but might instead be a chair or vehicle. So a condition such as:

```
if (player in Bridleway) ...
```

would be `false` if the player were riding a horse through the Bridleway. The safer alternative is:

```
if (location == Bridleway) ...
```

but even this would be `false` if the Bridleway were in darkness, because then `location` would be the special object `thedark` (see §19). The definitive location value is stored in `real_location`, so that:

```
if (real_location == Bridleway) ...
```

works in all cases. The condition for "is the player in a dark Bridleway?" is:

```
if (location == thedark && real_location == Bridleway) ...
```

Except for the one time in `Initialise`, you should not attempt to change either of these variables, nor to `move` the player-object by hand. One safe way to move the player in your own source code is to cause actions like

```
<Go n_obj>;
```

but for moments of teleportation it's easier to use the library routine `PlayerTo`. Calling `PlayerTo(somewhere)` makes the parent-object of the player `somewhere` and adjusts the `location` variables accordingly: it also runs through a fair number of standard game rules, for instance checking the light level and performing a `Look` action to print out the new room description. The value `somewhere` can be a room, or an `enterable` object such as a cage or a traction-engine, provided that the cardinal rule is always observed:

> The parent of the `player` object must at all times be "location-like". An object is "location-like" if either it is a location, or it has `enterable` and its parent is location-like.

In other words, you can't put the player in an `enterable` cardboard box if that box is itself shut up in a free-standing safe which isn't `enterable`. And you can't `PlayerTo(nothing)` or `PlayerTo(thedark)` because `nothing` is not an object and `thedark` is not location-like.

△ Calling `PlayerTo(somewhere,1)` moves the player without printing any room description. All other standard game rules are applied.

△ Calling `PlayerTo(somewhere,2)` is just like `PlayerTo(somewhere)` except that the room description is in the form the player would expect from typing "go east" rather than from typing "look". The only difference is that in the former case the room is (normally) given an abbreviated description if it has been visited before, whereas in the latter case the description is always given in full.

. . . . .

△△ It's perhaps worth taking a moment to say what the standard rules upon changing location are. The following rules are applied whenever a `Look` action or a call to `PlayerTo` take place.

(0) If `PlayerTo` has been called then the `parent` of the player, `location` and `real_location` are set.
(1) Any object providing `found_in` is checked. If it claims to be `found_in` the location, it is moved to that location. If not, or if it has `absent`, it is removed from the object tree. (See §8.)
(2) The availability of light is checked (see §19), and `location` is set to `thedark` if necessary.
(3) The "visibility ceiling" of the player is determined. For instance, the *VC* for a player in a closed wooden box is the box, but for a player in a closed glass box it's the location. To be more exact:
    (a) The *VC* of a room is the value of `location`, i.e., either `thedark` or the room object.
    (b) If the parent of an object is a room or is "see-through" (see §19 for this definition), the *VC* of the object is the *VC* of the parent.

(c) If not, the *VC* of the object is its parent.

(4) If the *VC* is `thedark` or (say) a box, skip this rule. Otherwise: if the *VC* has changed since the *previous* time that rule (3) produced a *VC* which wasn't an `enterable` object, then:
   (a) The message `location.initial()` is sent, if the location provides an `initial` rule. If the library finds that the player has been moved in the course of running `initial`, it goes back to rule (3).
   (b) The game's entry point routine `NewRoom` is called, if it provides one.

(5) The room description is printed out, unless these rules are being gone through by `PlayerTo(somewhere,1)`. For exactly what happens in printing a room description, see §26.

(6) If the `location` doesn't have `visited`, give it this attribute and award the player `ROOM_SCORE` points if the `location` has `scored`. (See §22.) Note that this rule looks at `location`, not `real_location`, so no points unless the room is visible.

. . . . .

In the course of this chapter, rules to interfere with actions have been attached to items, rooms and people, but not yet to the player. In §18 it was set out that an order like "austin, eat tuna" would result in the action `Eat tuna` being passed to `austin.orders`, and heavy hints were dropped that orders and actions are more or less the same thing. This is indeed so, and the player's own object has an `orders` routine. This normally does nothing and always returns `false` to mean "carry on as usual", but you can install a rule of your own instead:

```
player.orders = MyNewRule;
```

where `MyNewRule` is a new `orders` rule. This rule is applied to every action or order issued by the player. The variable `actor` holds the person asked to do something, usually but not always `player`, and the variables `action`, `noun` and `second` are set up as usual. For instance:

Example command	`actor`	`action`	`noun`	`second`
"put tuna in dish"	`player`	`Insert`	`tuna`	`dish`
"austin, eat tuna"	`Austin`	`Eat`	`tuna`	`nothing`

For instance, if a cannon goes off right next to the player, a period of partial deafness might ensue:

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

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

• △**EXERCISE 51**
Why not achieve the same effect by giving the player a `react_before` rule instead?

• **EXERCISE 52**
(Cf. 'Curses'.) Write an `orders` routine for the player so that wearing a gas mask will prevent speech.

. . . . .

The `player` object can not only be altered but switched altogether, allowing the player to play from the perspective of someone or something else at any point in the game. The player who tampers with Dr Frankenstein's brain transference machine may suddenly become the Monster strapped to the table. A player who drinks too much wine could become a drunk player object to whom many different rules apply. The "snavig" spell of 'Spellbreaker', which transforms the player to an animal like the one cast upon, could be implemented thus. Similarly the protagonist of 'Suspended', who telepathically runs a weather-control station by acting through six sensory robots, Iris, Waldo, Sensa, Auda, Poet and Whiz. In a less original setting, a player might have a team of four adventurers exploring a labyrinth, and be able to switch the one being controlled by typing the name. In this case, an `AfterLife` routine (see below) may be needed to switch the focus back to a still-living member of the team after one has met a sticky end.

The library routine `ChangePlayer(obj)` transforms the player to `obj`. Any object can be used for this. There's no need to give it any `name`, as the parser always understands pronouns like "me" and "myself" to refer to the current player-object. You may want to set its `description`, as this is the text printed if the player types "examine myself", or its `capacity`, the maximum number of items which this form of the player can carry. Finally, this player-object can have its own `orders` property and thus its own rules about what it can and can't do.

As `ChangePlayer` prints nothing, you may want to follow the call with a `<<Look>>;` action.

△ You can call `ChangePlayer` as part of a game's `Initialise` routine, but if so then you should do this before setting `location`.

△ Calling `ChangePlayer(obj,1);` does the same except that it makes the game print "(as Whoever)" during subsequent room descriptions.

△ The body dispossessed remains where it was, in play, unless you `move` it away or otherwise dispose of it. The player-object which the player begins with is a library-defined object called `selfobj`, and is described in room descriptions as "your former self".

• **EXERCISE 53**
In Central American legend, a sorceror can transform himself into a *nagual*, a familiar such as a spider-monkey; indeed, each individual has an animal self or *wayhel*, living in a volcanic land over which the king, as a jaguar, rules. Turn the player into *wayhel* form.

• **EXERCISE 54**
Alter the Wormcast of 'Ruins' (previously defined in §9) so that when in *wayhel* form, the player can pass through into a hidden burial shaft.

• **EXERCISE 55**
To complete the design of this sequence from 'Ruins', place a visible iron cage above the hidden burial shaft. The cage contains skeletons and a warning written in glyphs, but the player who enters it despite these (and they all will) passes into *wayhel* life. (The transformed body is unable to move the sodium lamp, but has nocturnal vision, so doesn't need to.) Unfortunately the player is now outside a cage which has closed around the human self which must be returned to, while the *wayhel* lacks the dexterity to open the cage. The solution is to use the Wormcast to reach the Burial Chamber, then bring its earthen roof down, opening a connection between the chamber below and the cage above. Recovering human form, the player can take the grave goods, climb up into the cage, open it from the inside and escape. Lara Croft would be proud.

. . . . .

△△ The situation becomes a little complicated if the same `orders` routine has to do service in two situations: once while its owner is a character met in the course of play, and then a second time when the player has changed into it. This could be done simply by changing the value of `orders` when the transformation takes place, but an alternative is to arrange code for a single `orders` routine like so:

```
orders [;
    if (player == self) {
        if (actor == self) {
            ! I give myself an action
        }
        else {
            ! I give someone else an order
        }
    }
```

```
        else {
            ! Someone else gives me an order
        }
    ],
```

• △△**EXERCISE 56**

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

. . . . .

"There are only three events in a man's life; birth, life and death; he is not conscious of being born, he dies in pain and he forgets to live." (Jean de la Bruyère again.) Death is indeed the usual conclusion of an adventure game, and occurs when the source code sets the library variable `deadflag` to `true`: in normal play `deadflag` is always `false`. The "standard Inform rules" never lead to the player's death, so this is something the designer must explicitly do.

Unlike life, however, interactive fiction offers another way out: the player can win. This happens if and when the variable `deadflag` is set to 2.

Any higher values of `deadflag` are considered to be more exotic ways the game can end, requiring text beyond "You have died" or "You have won". The Inform library doesn't know what this text should be, so it calls the `DeathMessage` entry point routine, which is expected to look at `deadflag` and can then print something suitable. For instance, 'Ruins' has a chasm subject to the following `before` rule:

```
before [;
    Enter: deadflag = 3;
        "You plummet through the silent void of darkness, cracking
        your skull against an outcrop of rock. Amid the pain and
        redness, you dimly make out the God with the
        Owl-Headdress...";
    JumpOver: "It is far too wide.";
],
```

and this means that it needs a `DeathMessage` routine like so:

```
[ DeathMessage;
  if (deadflag == 3) print "You have been captured";
];
```

Capture was about the worst fate that could befall you in the unspeakably inhumane world of Maya strife.

'Ruins' doesn't, but many games allow reincarnation or, as David M. Baggett points out, in fact resurrection. You too can allow this, by providing an `AfterLife` entry point routine. This gets the chance to do as it pleases before any "death message" is printed, and it can even reset `deadflag` to `false`, causing the game to resume as though nothing had happened. Such `AfterLife` routines can be tricky to write, though, because the game often has to be altered to reflect what has happened.

• **REFERENCES**

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

§22 Miscellaneous constants, scoring, quotations

> For when the One Great Scorer comes
> To write against your name,
> He marks – not that you won or lost –
> But how you played the game.
>
> — Grantland Rice (1880–1954), *Alumnus Football*

There are some constants which, if defined in your code *before* the library files are included, change the standard game rules or tell the Inform library about your game. Two such constants appeared back in §4: the strings of text `Story` and `Headline`.

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

. . . . .

The library won't allow the player to carry an indefinite number of objects. As was mentioned in §21, the limit is the value of `capacity` for the current player-object, which you're free to vary during play. The library sets up the `capacity` of the usual player-object to be equal to a constant called `MAX_CARRIED`, which is normally 100. But you can define it differently, and 'Ruins' does:

```
Constant MAX_CARRIED = 7;
```

For these purposes a `container` counts as only one object, even if it contains hundreds of other objects.

Many games, perhaps too many, involve collecting vast miscellanies of items until a use has been found for each. A small value of `MAX_CARRIED` will then annoy the player unreasonably, whereas a large one will stretch plausibility. The standard resolution is to give the player a sack for carrying spare objects, and the Inform library provides a feature whereby the designer can nominate a given `container` to be this "sack":

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

(This is from 'Toyshop': the 'Ruins' have been sacked too many times as it is.) The convenience this offers is that the game will now automatically put old, least-used objects away into the sack as the game progresses, provided the sack is still being carried:

```
>get biscuit
(putting the old striped scarf into the canvas rucksack to make room)
Taken.
```

. . . . .

The "Invisiclues" hints of some of the reissued Infocom games sometimes included a category called "For Your Amusement", listing some of the improbable things the game can do. Only the victorious should read this, as it might spoil surprises for anyone else. You can, optionally, provide such "amusing" information by defining the constant...

```
Constant AMUSING_PROVIDED;
```

... and also providing an entry point routine called `Amusing`. For a player who has won the game, but not one who has merely died, the usual question

> Would you like to RESTART, RESTORE a saved game or QUIT?

will then become

> Would you like to RESTART, RESTORE a saved game, see some suggestions for AMUSING things to do or QUIT?

(The best way to provide such suggestions, if there are many, is to use a menu system like that described in §44.) One of the worst-kept secrets of the Inform library is that an option not mentioned by this question is to type "undo", which will undo the last move and restore the player to life. If you feel that this option should be mentioned, define the constant:

```
Constant DEATH_MENTION_UNDO;
```

Finally, this end-of-game question will also mention the possibility of typing "full" to see a full score breakdown, if tasks are provided (see below).

. . . . .

The other constants you are allowed to define help keep the score. There are two scoring systems provided by the library, side by side: you can use both or neither. You can always do what you like to the library's `score` variable in any case, though the "fullscore" verb might not then fully account for what's

happened. Whatever scoring system you use, you should define MAX_SCORE, as 'Ruins' for instance does by declaring:

```
Constant MAX_SCORE = 30;
```

This is the value which the library tells to the player as the maximum score attainable in text like:

> You have so far scored 0 out of a possible 30, in 1 turn.

Note that the library does *not* check that this is the actual maximum score it's possible to clock up: and nor does it cause the game to be automatically won if the maximum is achieved. The game is won when and only when deadflag is set to 2 (see §21), regardless of score.

The simpler scoring system awards points for the first time certain objects are picked up, and the first time certain places are entered. (As long as there is light to see by: no points unless you can recognise that you've arrived somewhere interesting.) To make an item or a place carry a points bonus, give it the attribute scored. You may also want to vary the amounts of these bonuses by defining two constants:

OBJECT_SCORE	points for picking up a scored object (normally 4);
ROOM_SCORE	points for entering a scored room (normally 5)

The more elaborate scoring system keeps track of which "tasks" the player has accomplished. These are only present if the constant TASKS_PROVIDED is defined, and then the further constant NUMBER_TASKS should indicate how many tasks have to be accomplished. If this value is N, then the tasks are numbered 0, 1, 2, ..., $N-1$. The number of points gained by solving each task must be defined in a -> array with N entries called task_scores, like so:

```
Constant TASKS_PROVIDED;
Constant NUMBER_TASKS = 5;
Constant MAX_SCORE = 25;
Array task_scores -> 3 7 3 5 7;
```

Thus task 0 scores three points, task 1 scores seven points and so on. Since the entries in a -> array have to be numbers between 0 and 255, no task can have a negative score or a score higher than 255. Besides a points score, each task has a name, and these are printed by an entry point routine called PrintTaskName. For instance ('Toyshop'):

```
[ PrintTaskName task_number;
  switch (task_number) {
      0: "eating a sweet";
      1: "driving the car";
      2: "shutting out the draught";
      3: "building a tower of four";
      4: "seeing which way the mantelpiece leans";
  }
];
```

Finally, the game's source code should call `Achieved(task_number)` to tell the library that the given task has been completed. If this task has been completed before, the library will do nothing: if not, the library will award the appropriate number of points. The verb "full" will give a full score breakdown including the achieved task in all future listings.

. . . . .

When points are awarded by a call to `Achieved`, or by the player picking up a `scored` object, or visiting a `scored` place, or simply by the source code itself altering the `score` variable, no text is printed at the time. Instead, the library will normally notice at the end of the turn in question that the score has changed, and will print a message like:

[Your score has gone up by three points.]

Not all players like this feature, so it can be turned on and off with the "notify" verb, but by default it is on. The designer can also turn the feature off and on: it is off if the library's variable `notify_mode` is `false`, on if it is `true`.

. . . . .

Another (optional) entry point routine, called `PrintRank`, gets the chance to print text additional to the score. It's called `PrintRank` because the traditional "something additional" is a ranking based on the current score. Here is 'Ruins':

```
[ PrintRank;
  print ", earning you the rank of ";
  if (score == 30) "Director of the Carnegie Institution.";
  if (score >= 20) "Archaeologist.";
  if (score >= 10) "Curiosity-seeker.";
  if (score >= 5) "Explorer.";
  "Tourist.";
];
```

. . . . .

Besides the score breakdown, two more verbs are usually provided to the player: ''objects'' and ''places''. The former lists off all the objects handled by the player and where they are now; the latter lists all the places visited by the player. In some game designs, these verbs will cause problems: you can get rid of them both by defining the constant `NO_PLACES`.

• △**EXERCISE 57**

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

. . . . .

The rest of this section runs through some simple ''special effects'' which are often included in games. See Chapter VII for much more on this, and in particular see §44 for using the `"Menus.h"` library extension.

The first effect is hardly special at all: to ask the player a yes/no question. To do this, print up the question and then call the library routine `YesOrNo`, which returns true/false accordingly.

The status line is perhaps the most distinctive feature of Infocom games in play. This is the (usually highlighted) bar across the top of the screen. Usually, the game automatically prints the current game location, and either the time or the score and number of turns taken. It has the score/turns format unless the directive

```
Statusline time;
```

has been written in the program, in which case the game's 24-hour clock is displayed. See §20 for more on time-keeping.

△ If you want to change this, you need to `Replace` the parser's `DrawStatusLine` routine. This requires some assembly language programming: there are several examples of altered status lines in the exercises to §42.

. . . . .

Many games contain quotations, produced with `box` statements like so:

```
box "I might repeat to myself, slowly and soothingly,"
    "a list of quotations beautiful from minds profound;"
    "if I can remember any of the damn things."
    ""
    "-- Dorothy Parker";
```

A snag with printing such boxes is that if you do it in the middle of a turn then it will probably scroll half-off the screen by the time the game finishes printing for the turn. The right time to do so is just after the prompt (usually ">") is printed, when the screen will definitely scroll no more. You could use the `Prompt:` slot in `LibraryMessages` to achieve this (see §25), but a more convenient way is to put your box-printing into the entry point routine `AfterPrompt`, which is called at this time in every turn.

● **EXERCISE 58**

Devise a class `Quotation`, so that calling `QuoteFrom(Q)` for any quotation `Q` will cause it to be displayed at the end of the current turn, provided it hasn't been quoted before.

● **REFERENCES**

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

§23 'Ruins' revisited

> These fragments I have shored against my ruins
> —T. S. Eliot (1888–1965), *The Waste Land*

Though 'Ruins' is a small world, and distorted in shape by the need to have "one example of everything", it seems worth a few pages to gather together the fragments scattered through the book so far and complete the game.

To begin with, the stage set back in §4 was too generic, too plain. Chosen at random, it may as well become La Milpa, a site rediscovered in dense rainforest by Eric Thompson in 1938, towards the end of the glory days of archaeological exploration. (La Milpa has been sadly looted since.) Though this is something of a cliché of interactive fiction, 'Ruins' contains two objects whose purpose is to anchor the player in time and place. Lining the packing case, we find:

```
Object -> -> newspaper "month-old newspaper"
  with name 'times' 'newspaper' 'paper' 'month-old' 'old',
       description
           "~The Times~ for 26 February, 1938, at once damp and brittle
           after a month's exposure to the climate, which is much the
           way you feel yourself. Perhaps there is fog in London.
           Perhaps there are bombs.";
```

And among the player's initial possessions:

```
Object map "sketch-map of Quintana Roo"
  with name 'map' 'sketch' 'sketch-map' 'quintana' 'roo',
       description
           "This map marks little more than the creek which brought you
           here, off the south-east edge of Mexico and into deepest
           rainforest, broken only by this raised plateau.";
```

To turn from the setting to the prologue, it is a little too easy to enter the structure in the rainforest. And if the steps were always open, surely the rain would sluice in? Recall that the `Forest` includes inward map connections to the `steps`, which are a door, instead of to the `Square_Chamber` directly:

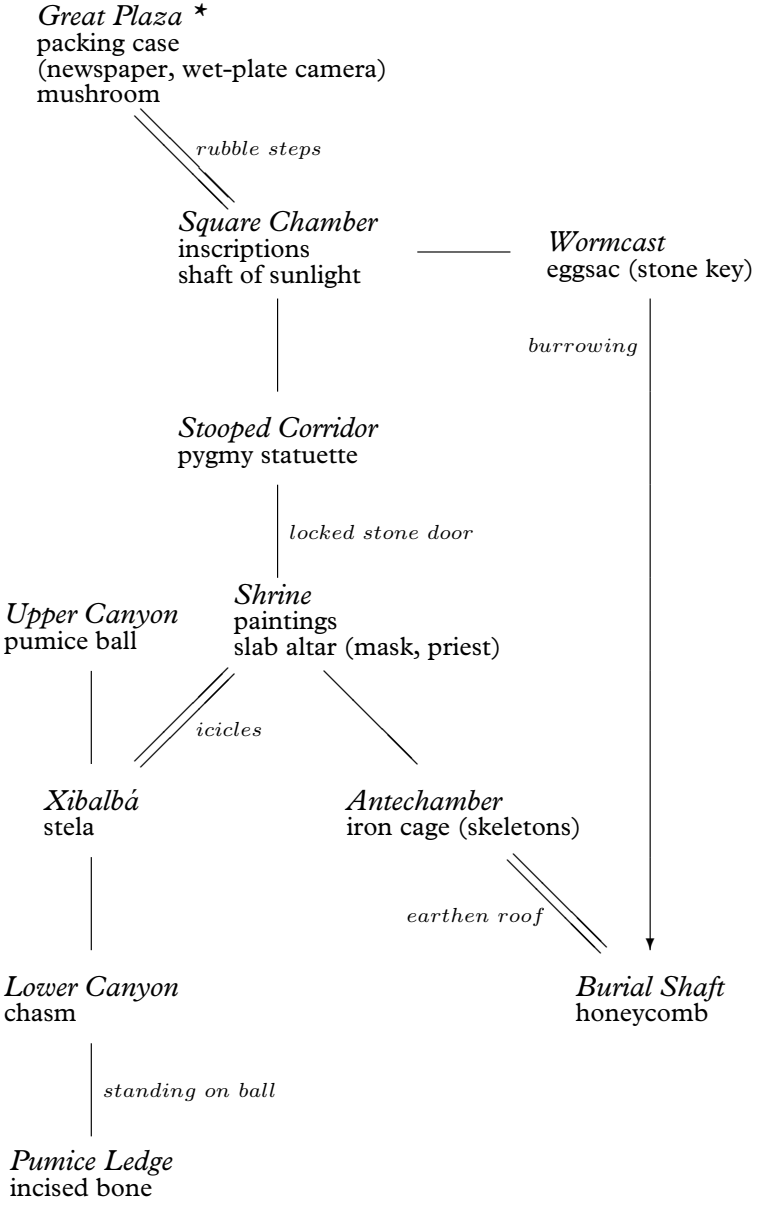
```
       d_to steps, in_to steps,
```

The steps are, however, intentionally blocked by rubble, as happened in the case of the hidden staircase found by Alberto Ruz beneath the Temple of the Inscriptions at another site, Palenque:

```
Object -> steps "stone-cut steps"
  with name 'steps' 'stone' 'stairs' 'stone-cut' 'pyramid' 'burial'
            'structure' 'ten' '10',
       rubble_filled true,
       description [;
           if (self.rubble_filled)
               "Rubble blocks the way after only a few steps.";
           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.";
       ],
       door_to [;
           if (self.rubble_filled)
               "Rubble blocks the way after only a few steps.";
           return Square_Chamber;
       ],
       door_dir d_to
  has  scenery door open;
```

Next we must face the delicate issue of how to get from the mundane 1930s to a semi-magical Maya world. The stock device of Miguel Angel Asturias's *Leyendas de Guatemala* and other founding works of magic realism (indeed, of *Wuthering Heights* come to think of it) is for the arriving, European rationalist to become fascinated by a long tale told by local peasants. This would take too much code to get right in so small a game, though, and there also remains the unresolved question of what the mushroom is for. So we delete the original `before` rule for the mushroom, which made eating it potentially fatal, and instead give it an `after`:

```
Eat: steps.rubble_filled = false;
    "You nibble at one corner, unable to trace the source of an
    acrid taste, distracted by the flight of a macaw overhead
    which seems to burst out of the sun, the sound of the beating
    of its wings almost deafening, stone falling against stone.";
```



⋆ The player begins at the Great Plaza, carrying the map, the sodium lamp and Waldeck's Mayan dictionary.

This is fairly authentic, as a cult of hallucinogenic mushrooms seems to have existed. Anyway, the player is getting off pretty lightly considering that Maya lords also went in for narcotic enemas and ritual blood-letting from the tongue and penis, an interactive fiction for which the world is not yet ready.

Descending underground, §8 alluded to an eggsac which burst on contact with natural light. Naturally, this repellent object belongs in the Wormcast, and here is its definition:

```
Object -> eggsac "glistening white eggsac",
  with name 'egg' 'sac' 'eggs' 'eggsac',
       initial "A glistening white eggsac, like a clump of frogspawn
           the size of a beach ball, has adhered itself to something
           in a crevice in one wall.",
       after [;
           Take: "Oh my.";
       ],
       react_before [;
           Go: if (location == Square_Chamber && noun == u_obj) {
                   deadflag = true;
                   "The moment that natural light falls upon the
                   eggsac, it bubbles obscenely and distends. Before
                   you can throw it away, it bursts into a hundred
                   tiny, birth-hungry insects...";
               }
       ];
```

Note the clue that some object is within the egg sac: as it turned out in §8, a stone key, released by putting the egg sac into the shaft of sunlight. The key itself has a very short definition:

```
Object stone_key "stone key"
  with name 'stone' 'key';
```

This is not an easy puzzle, but a further clue is provided by the carvings on the Stone Chamber wall, which can be translated with Waldeck's dictionary to read "becoming the Sun/life".

Given the key, the player must next solve the problem of bringing light to the Stooped Corridor, by pushing the burning sodium lamp south. This means, as promised in §14, adding a `before` rule to the lamp:

```
PushDir:
    if (location == Shrine && second == sw_obj)
        "The nearest you can do is to push the sodium lamp to
        the very lip of the Shrine, where the cave floor falls
```

```
        away.";
    AllowPushDir(); rtrue;
```

§14 also promised to run down the battery power: although since 100 turns is plenty, this rule doesn't play any real part in the game and is just window-dressing. We need to `StartDaemon(sodium_lamp)` in the `Initialise` routine, and define the lamp's `daemon` along the following lines:

```
daemon [;
    if (self hasnt on) return;
    if (--self.battery_power == 0)
        give self ~light ~on;
    if (self in location) {
        switch (self.battery_power) {
            10: "^The sodium lamp is getting dimmer!";
             5: "^The sodium lamp can't last much longer.";
             0: "^The sodium lamp fades and suddenly dies.";
        }
    }
],
```

With the obligatory light puzzle solved, the Shrine can at last be opened:

```
Object Shrine "Shrine"
  with description
           "This magnificent Shrine shows signs of being hollowed out
           from already-existing limestone caves, especially in the
           western of the two long eaves to the south.",
       n_to StoneDoor, se_to Antechamber,
       sw_to
           "The eaves taper out into a crevice which would wind
           further if it weren't jammed tight with icicles. The glyph
           of the Crescent is not quite obscured by ice.";
```

Looking up the Crescent glyph in the dictionary (§16) reveals that it stands for the word "xibalbá": asking the Priest (§17) brought into existence by wearing the jade mosaic mask (§11) melts the icicles and makes the southwest connection to Xibalbá, of which more later. No Maya game would be complete without their religiously-observed cyclical countings of time:

```
Object -> paintings "paintings"
  with name 'painting' 'paintings' 'lord' 'captive',
       initial "Vividly busy paintings, of the armoured Lord trampling
           on a captive, are almost too bright to look at, the
           graffiti of an organised mob.",
```

```
      description "The flesh on the bodies is blood-red. The markers
          of the Long Count date the event to 10 baktun 4 katun 0 tun
          0 uinal 0 kin, the sort of anniversary when one Lord would
          finally decapitate a captured rival who had been ritually
          tortured over a period of some years, in the Balkanised
          insanity of the Maya city states.",
  has  static;
```

Having called the priest "calendrical", here's another topic to Ask the priest about:

```
'paintings': "The calendrical priest frowns.
     ~10 baktun, 4 katun, that makes 1,468,800 days
     since the beginning of time: in your calendar
     19 January 909.~";
```

And also, to make the point once more, and remind the player once again of distant Europe:

```
Show, Give: ...
    if (noun == newspaper)
        "He looks at the date. ~12 baktun 16 katun 4 tun
        1 uinal 12 kin~, he declares before browsing the
        front page. ~Ah. Progress, I see.~";
```

Perhaps the player will never see either calculation: if so, it doesn't matter, as dates and calendars turn out to be this game's red herring. (Every game should have one.) The Antechamber of the Shrine is an undistinguished room...

```
Object Antechamber "Antechamber"
  with description
          "The southeastern eaves of the Shrine make a curious
          antechamber.",
       nw_to Shrine;
```

... except that this is where the iron cage (§15 and §21) is located, so that the Burial Shaft lies below, with its complex puzzle in which the player is transformed to a warthog and back again, opening the shaft. Lastly, then, in the southwest eaves of the Shrine is a natural cave entrance, which in Mayan mythology leads to the Underworld. There is supposed to be a crossroads here, but in this modest game a three-way junction is all we have space for:

```
Object Junction "Xibalb@'a"
  with description
          "Fifty metres beneath rainforest, and the sound of water
          is everywhere: these deep, eroded limestone caves
```

```
            extend like tap roots. A slither northeast by a broad
            collapsed column of ice-covered rock leads back to the
            Shrine, while a kind of canyon floor extends uphill to
            the north and downwards to south, pale white like shark's
            teeth in the diffused light from the sodium lamp above.",
        ne_to Shrine, n_to Canyon_N, u_to Canyon_N,
        s_to Canyon_S, d_to Canyon_S,
  has  light;
Treasure -> stela "stela"
  with name 'stela' 'boundary' 'stone' 'marker',
       initial
            "A modest-sized stela, or boundary stone, rests on a
            ledge at head height.",
       description
            "The carvings appear to warn that the boundary of
            Xibalb@'a, Place of Fright, is near. The Bird glyph is
            prominent.";
```

This canyon houses the eight-foot pumice stone ball (see §15) at the north end, and the chasm (§12, §21) at the south:

```
Object Canyon_N "Upper End of Canyon"
  with s_to Junction, d_to Junction,
       description
            "The higher, broader northern end of the canyon rises only
            to an uneven wall of volcanic karst.",
  has  light;
Object Canyon_S "Lower End of Canyon"
  with n_to Junction, u_to Junction,
       s_to "Into the chasm?", d_to nothing,
       description
            "At the lower, and narrower, southern end, the canyon stops
            dead at a chasm of vertiginous blackness.  Nothing can be
            seen or heard from below.",
  has  light;
```

As promised in §12, the chasm must react to having the stone ball pushed into it, which means adding this to the chasm's definition:

```
each_turn [;
    if (huge_ball in parent(self)) {
        remove huge_ball; Canyon_S.s_to = On_Ball;
        Canyon_S.description = "The southern end of the canyon
            now continues onto the pumice-stone ball, wedged into
            the chasm.";
```

```
        "^The pumice-stone ball rolls out of control down the
        last few feet of the canyon before shuddering into the
        jaws of the chasm, bouncing back a little and catching
        you a blow on the side of the forehead. You slump
        forward, bleeding, and... the pumice-stone shrinks,
        or else your hand grows, because you seem now to be
        holding it, staring at Alligator, son of seven-Macaw,
        across the ball-court of the Plaza, the heads of his
        last opponents impaled on spikes, a congregation baying
        for your blood, and there is nothing to do but to throw
        anyway, and... but this is all nonsense, and you have
        a splitting headache.";
    }
],
```

(Horribly violent, semi-religious ball-game rituals are common in early central America, though nobody knows why: all substantial Maya cities have prominent ball-courts.) A fat paragraph of text in which fairly interesting things happen, beyond the player's control, is sometimes called a "cut-scene". Most critics dislike the casual use of cut-scenes, and 'Ruins' would be a better game if the confrontation with Alligator were an interactive scene. But this manual hasn't the space. Instead, here is the final location, which represents "standing on the wedged ball":

```
Object On_Ball "Pumice-Stone Ledge"
  with n_to Canyon_S, d_to Canyon_S, u_to Canyon_S,
       description
           "An impromptu ledge formed by the pumice-stone ball,
           wedged into place in the chasm. The canyon nevertheless
           ends here.",
  has  light;
Treasure -> "incised bone"
  with name 'incised' 'carved' 'bone',
       initial
           "Of all the sacrificial goods thrown into the chasm, perhaps
           nothing will be reclaimed: nothing but an incised bone,
           lighter than it looks, which projects from a pocket of wet
           silt in the canyon wall.",
       description
           "A hand holding a brush pen appears from the jaws of
           Itzamn@'a, inventor of writing, in his serpent form.";
```

And this is where Itzamná lays down his brush, for this is the fifth and last of the cultural artifacts to collect. The game ends when they are all deposited in the packing case, a rule which means a slight expansion of the definition of `Treasure`:

```
after [;
    Insert:
        ...
        if (score == MAX_SCORE) {
            deadflag = 2;
            "As you carefully pack away ", (the) second,
            " a red-tailed macaw flutters down from the tree-tops,
            feathers heavy in the recent rain, the sound of its
            beating wings almost deafening, stone falling against
            stone... As the skies clear, a crescent moon rises above
            a peaceful jungle. It is the end of March, 1938, and it
            is time to go home.";
        }
```

△ The following sequence of 111 moves tests 'Ruins' from beginning to end: ''examine case / read newspaper / get newspaper / get camera / down / examine steps / east / up / enter structure 10 / eat mushroom / eat mushroom / down / examine inscriptions / look up arrow in dictionary / east / get eggsac / west / put eggsac in sunlight / get key / drop lamp / light lamp / look / push lamp s / get statuette / drop all except camera / photograph statuette / get key / open door / unlock door with key / open door / drop key / get pygmy / north / up / put pygmy in case / down / south / get dictionary / get newspaper / south / north / push lamp south / examine paintings / drop all but camera / photograph mask / get mask / get dictionary / get newspaper / wear mask / show dictionary to priest / show newspaper to priest / drop newspaper / ask priest about ruins / ask priest about paintings / se / nw / push lamp se / push lamp nw / sw / look up crescent in dictionary / ask priest about xibalba / sw / north / push ball south / push ball south / south / drop all but camera / remove mask / drop mask / photograph bone / get all / north / north / drop all but camera / photograph stela / get all / ne / north / north / up / put bone in case / put stela in case / put mask in case / down / east / nw / west / south / south / push lamp se / examine cage / enter cage / open cage / nw / north / north / east / down / east / up / drop all but camera / photograph honeycomb / get all / up / open cage / out / push lamp nw / north / north / up / put honeycomb in case''.

• **REFERENCES**

I am indebted to, which is to say I have roundly travestied, the following: ''Mapping La Milpa: a Maya city in northwestern Belize'' (Tourtellot, Clarke and Hammond, *Antiquity* 67 (1993), 96–108). All the same 'Ruins' favours old-fashioned ideas of Maya, a good example being the ''calendrical priests'' fondly imagined by early archaeologists before Maya writing was deciphered. •The standard all-in-one-book book is Michael

D. Coe's *The Maya* (fourth edition). •The same author's history of *Breaking the Maya Code* offers pungently vivid portraits of Sir Eric Thompson and Maximilien Waldeck. The British Museum guide *Maya Glyphs*, by S. D. Houston, is a trifle more reliable than Waldeck's work. •Numerous colour-postcard photographs by F. Monfort are collected in *Yucatan and the Maya Civilization* (Crescent Books, 1978).

§24 The world model described

This section is a self-contained summary of the concepts and systematic organising principles used by Inform to present the illusion of describing a physically real environment, with which the protagonist of a game interacts. All details of implementation are ignored and Inform jargon is either avoided or explained. While many of the rules are standard to all world models used for interactive fiction, some are not, and the footnotes remark on some of the more interesting cases. The next section, §25, buries itself back into implementation to discuss how to add new rules or to change those rules below which you don't agree with. The description below is arranged as follows: ¶1. Substance; ¶2. Containment; ¶3. Space; ¶4. Sense; ¶5. Time; ¶6. Action.

1. Substance

1.1. Objects make up the substance of the world: its places and their contents, abstract relations between these, transient states the world can be in, actors and trends (such as the flowing of a river).

1.2. At any given time, every object in the world model is one and only one of the following kinds: the player; a room; the darkness object; an item; the compass object; a compass direction; or something which is out of play.[1]

1.2.1. The player object represents the protagonist of the game.

1.2.2. A room represents some region of space, not necessarily with walls or indoors.

1.2.3. The darkness pseudo-room represents the experience of being in a dark place, and has no specific location in space.

[1] The compass pseudo-item and the darkness pseudo-room are anomalies. The compass arises from a generic convention which is unrealistic but avoids making the game needlessly tiresome: that the player has a perfect inherent sense of direction. The representation of darkness is less defensible. Although vaguely justifiable from the assumption that the experience of being in one entirely dark place is much like another, it came about instead for reasons of implementation: partly to bundle up various darkness-related texts as though they were room descriptions, and partly because early versions of the Inform run-time format (version 3 of the Z-machine) imposed a restriction that the status line displayed above the screen could only be the short name of an object.

1.2.4. An item represents some body (or group of similar bodies) with a definite spatial position at any given time. It is not necessarily solid but its substance is indivisible.

1.2.5. The compass pseudo-item represents the frame of reference within the protagonist's head, and is not an actual compass with its attendant hazards of being dropped, broken, stolen or becoming invisible in pitch darkness.

1.2.6. A compass direction represents a potential direction of movement, such as "northeast", "down", "in" or "starboard".

1.2.7. Objects out of play represent nothing in the model world and the protagonist does not interact with them. Out of play objects are mostly things which once existed within the model world but which were destroyed, or which have not yet been brought into being.

1.2.8. An object can change its kind as the game progresses: for instance a compass direction can be taken out of play, and certain items can become the player, making the object which was previously the player now merely an item.

1.3. Objects are indivisible even if the player may see internal structure to them, such as the four legs which are part of a chair. Pieces of objects only appear in the model if additional objects are provided for them.[2]

1.4. Objects have internal states and are therefore distinguishable from each other by more than their position in the containment tree. Some are "open", some are "concealed" and so on, and they are given descriptions and other specifications by the designer.

1.4.1. Some objects are "switchable" between two mutually exclusive states, "on" and "off". These represent machines, trapdoors and the like, which behave differently when set to when unset, and which generally have rules about the process of setting and unsetting.

1.4.2. Some objects are "lockable" and someone with the specified "key" object can switch between mutually exclusive states, "locked" and "unlocked". These objects represent containers and doors with locks.

1.4.3. Some objects are "openable" and therefore in one of two mutually exclusive states, "open" and "closed". If such an object is closed and also locked then it cannot be opened. These objects represent containers and doors.

[2] The atomic theory of matter. Since there are a finite number of atoms each with a finite range of possible states and positions, Heraclitus' doctrine that one cannot stand in the same river twice is false within the model world, and this can detract from its realism. It is sometimes too easy for the protagonist to exactly undo his actions as if they had never been and to return exactly to the world as it was before. Another problem with ¶1.3 is that liquids need to be divisible ("some water" becoming "some water" and "some water").

2. Containment

2.1. Some objects are contained within other objects in what is sometimes called a tree, meaning that: (i) an object can either be contained in one other object (called its ''parent''), or not contained in any; (ii) there is no ''loop'' of objects such that each is contained in the next and the last is contained in the first. The pattern of containment changes frequently during play but (i) and (ii) always hold.[3]

2.1.1. The objects contained within something are kept in order of how long they have been there. The first possession (sometimes called the ''child'') is the one most recently arrived, and the last is the one which has been contained for longest.

2.1.2. In some games there are objects called ''floating objects'' which represent something found in many locations, such as a stream flowing through the map, or a pervasive cloud. These give the appearance of violating rule ¶2.1, but do not: the effect is an illusion brought about by making the floating object belong at all times to the same room as the player.

2.2. The following rules remain true at all times:

2.2.1. A room is not contained.

2.2.2. The darkness object and the compass are not contained.

2.2.3. A compass direction is contained in the compass object but itself contains nothing.

2.2.4. An item is always contained; either in the player, in another item or a room.

2.2.5. The player is always contained in a visitable object. An object is ''visitable'' if either (a) it is a room, or (b) it is enterable and it has a visitable parent.

2.2.6. An object out of play is either contained in another object out of play, or else not contained at all.[4]

2.3. Containment models a number of subtly different kinds of belonging:

2.3.1. The contents of a room object are near each other (usually within sight and touch) in the space represented by the room. For instance, the player,

[3] Infocom world models all included rule (ii) but their implementations made no systematic effort to enforce this, so that Infocom were perpetually fixing bugs arising from putting two containers inside each other.

[4] Without knowing the context, and looking at the object tree alone, it isn't easy to distinguish a room from an object out of play, which can make writing good debugging features tricky.

a wooden door and a table might all be contained in a room representing the inside of a small hut.[5]

2.3.2. The contents of the compass are the compass directions available in principle to an actor. If ''north'' is removed from the compass, then north becomes meaningless even if an actor is in a room with a north exit; if ''aft'' is added, then it need not follow that any room actually has an exit leading aft.

2.3.3. The contents of the player fall into two categories: those which are ''worn'', and the rest.

2.3.3.1. Worn objects represent clothing or accessories held onto the body without the need for hands, such as a belt or a rucksack.

2.3.3.2. The rest represent items being held in the player's hands.

2.3.4. The contents of an item model different kinds of belonging, depending on the nature of the item:[6]

2.3.4.1. Some items are ''containers'': they represent boxes, bottles, bags, holes in the wall and so on. The contents of a container are considered to be physically inside it. At any given time a container can be ''open'' or ''closed''.

2.3.4.2. Some items are ''supporters'': they represent tables, plinths, beds and so on. The contents of a supporter are considered to be physically on top of it.

2.3.4.3. Some items are ''animate'': they represent people, sentient creatures generally and higher animals. The contents of an animate object are considered to be carried by it.

2.3.4.4. Some items are ''enterable'', meaning that it is possible for the player to be contained within them. A large tea-chest might be an enterable container; a bed might be an enterable supporter. In the case of an enterable which is neither container nor supporter, and which contains the player, the player is considered to be confined close to the enterable: for instance, by a pair of manacles.

2.3.4.5. Failing this, the contents of an item represent pieces or components of it, such as a lever attached to machinery, or a slot cut into a slab of masonry.

---

[5] This part of the model was invented by the early mainframe 'Zork'. The Crowther and Woods 'Advent', and later the Scott Adams games, required all objects to belong to a room but had a pseudo-room which represented ''being carried by the player''. Objects were equated with potential possessions and no object represented the player.

[6] The model in ¶2.3.4 has attracted criticism as being simplistic in three respects: (a) an object is assumed not to be both a container and a supporter, so what about an oven?; (b) while the player has a distinction between items worn and items carried, animate objects do not; (c) being inside and being on top of are modelled, but being underneath or behind are not.

3. Space

3.1. Spatial arrangement on the small scale (at ranges of a few moments' walking distance or less) is modelled by considering some objects to lie close together and others to lie far apart.

3.1.1. Objects ultimately contained in the same room are considered to be close enough together that walking between the two is a largely unconscious act.

3.1.1.1. The model takes no account of directions from one such object to another, except as described in ¶2.3.4 above (e.g., that contents of a supporter are on top of it).

3.1.1.2. All objects with the same parent are considered to be equidistant from, and to have equal access to, each other.[7]

3.1.2. Objects ultimately contained in different rooms are considered to be so far apart that they will not ordinarily interact. Because of this the model takes no account of one being further away than another.[8]

3.2. Spatial arrangement on the large scale (at ranges of an appreciable walking distance or more) is modelled by joining rooms together at their edges, much as a patchwork quilt is made.

3.2.1. Rooms joined together represent areas which are adjacent in that they are separated by so short a walk that the walker does not have opportunity to think twice and turn back, or to stop halfway and do something else.

3.2.2. Rooms are joined either by a map connection or a door.[9]

3.2.2.1. Map connections come in different kinds, representing different directions in the geography of the world. These physical directions are north, south, east, west, northeast, northwest, southeast, southwest, up, down, in and out.

3.2.2.2. Each compass direction corresponds to a single physical direction at any given moment, and this represents the actual direction which a player will walk in if he tries to walk in a given direction within his own frame of reference.[10]

---

[7] In ¶4 it will become apparent that a measure of distance between two objects is given by their distance apart in the containment tree.

[8] Note that ¶3.1.2 says that objects far apart cannot interact, but ¶3.1.1 does *not* say that objects close together can do. A honey bee in a sealed hive cannot interact with or be aware of the delivery man carrying the hive to its new beekeeper. Rules on which close objects can interact are the subject of ¶4.

[9] There is no requirement for such a join to be usable from the other side.

[10] For instance, on board a ship a player may try to walk "starboard", that being a

3.2.2.3. A ''door'' is an item representing something which comes between two locations, which must be passed through or by in order to go from one to the other, and which it requires some conscious decision to use.[11]

3.2.2.4. As with a compass direction, a door corresponds to a single physical direction at any given moment.

4. Sense

4.1. The senses are used in the world model primarily to determine whether the player can, or cannot, interact with a nearby object. Three different kinds of accessibility are modelled: touch, sight and awareness.[12]

4.2. Awareness = sight + touch, that is, the player is aware of something if it can be seen or touched.[13]

4.2.1. Awareness represents the scope of the player's ability to interact with the world in a single action. Although the player may pursue a grand strategy, he must do so by a series of tactical moves each within the scope of awareness.[14]

4.3. There are only two strengths of light: good enough to read by and pitch blackness, which we shall call ''light'' and ''dark''.

4.3.1. Some containers and other items are specified as being transparent to light, meaning that light can pass through from what contains them to what they contain (for instance a glass box or a machine whose contents are the

---

compass direction, but will in fact move in some physical direction such as northeast. Games have also been designed in which the player's frame of reference consists only of ''left'', ''right'', ''forward'', ''back'', ''up'' and ''down'', whose assignment to physical directions changes continuously in play.

[11] Thus a vault door, a plank bridge or a ventilation duct high on one wall would be represented by doors, but an open passageway or a never-locked and familiar door within a house would instead be represented by map connections.

[12] Hearing, taste and smell are not modelled: instead Inform's implementation provides convenient verbs and actions for designers to add their own ad-hoc rules.

[13] Awareness = sense is a strongly restrictive position. Thanks to simplistic parsers, in some early games the player is somehow aware of all objects everywhere, even those not yet encountered. In some more modern games awareness = sense + recent memory. For instance an item dropped in a dark room can be picked up again, since it is assumed that the player can remember where it is.

[14] The Inform parser enforces this by recognising only those typed commands which request interaction with objects the player is aware of at a given time.

buttons on the its front panel), while others are opaque (for instance a wooden box or a spy who keeps all her belongings concealed).

4.3.2. An object is called "see-through" if it is transparent, or if it is a supporter, or if it is an open container, or if it is the player.

4.3.3. Some rooms are specified by the designer as having ambient light (those representing outdoor locations, caves with fluorescent ore formations, strip-lit office buildings and the like); some items are specified as giving off light (those representing torches, lanterns and the like). The room or item is said to be "lit".

4.3.4. There is light for the player to see by only if there is a lit object, close to the player in the sense of ¶3, such that every object between them is see-through. (For instance, if a player is in a sealed glass box in a cupboard which also contains a key, the glass box comes between player and key, but the cupboard does not.)[15]

4.4. What the player can touch depends on whether there is light.

4.4.1. In the light, the player can touch anything close to the player provided that (a) every object between them is see-through and (b) none of the objects between them is a closed container.

4.4.2. In the dark, the player can touch (a) anything contained in the player and (b) the enterable object which the player is contained in (if any).[16]

4.5. What the player can see also depends on whether there is light, but also on whether the designer has specified that any nearby items are "concealed", which models their being hidden from view. Concealed objects remain touchable but you would need to know they were there, since sight alone would not reveal this. On once being picked up, a concealed object ceases to be concealed.

4.5.1. In the light, the player can see any non-concealed object close to the player provided that (a) every object between them is see-through and (b) none of the objects between them is concealed.

4.5.2. In the dark, the player can see nothing.

---

[15] This definition is not as realistic as it looks. Translucency is equated with transparency, presenting problems for glazed glass jars. "Close to the player" implies that light never spills over from one room to another, for instance through an open window on a sunny day.

[16] Equivalently, in the dark you can touch exactly those objects adjacent to you in the containment tree. Thus anything which can be touched in the dark can also be touched in the light but not vice versa.

5. Time

5.1. The passage of time is represented by describing and changing the model world at regular intervals, each cycle being called a ''turn''. The interval from one such moment to the next is considered to be the time occupied by carrying out these changes, so that all basic changes (i.e., actions: see ¶6) consume the same unit of time.

5.2. Changes in the model world are carried out by a number of independent processes called ''daemons''. Some daemons are built in and others added by the designer of a particular game, conventionally by associating them with certain objects over which they have sway. A turn consists of the daemons being invoked one at a time until each has been given the chance to intervene, or to decline to intervene.

5.2.1. Daemons are invoked in the sequence: action daemon, any designed daemons (in no particular order), each-turn daemon, scoring daemon, clock daemon.

5.2.2. Daemons have the opportunity to carry out arbitrary changes to the state of the objects, but should be designed to violate the rules of the model world as little as possible. The built-in daemons do not violate them at all.

5.2.3. Certain designed daemons are ''timers'', meaning that they are set to decline to intervene for a set number of turns and will then act once and once only (unless or until reset).

5.3. The action daemon consults the player at the keyboard by asking which action should be performed and then performing it. (See ¶6.) The action daemon is invoked first in each turn.

5.4. The each-turn daemon polls any object of which the player is aware. If it has been specified by the designer as having an each-turn rule, then that rule is applied.

5.5. The scoring daemon keeps track of the length of the game so far by keeping count of the number of turns. It also measures the player's progress by keeping score:

5.5.1. Points are awarded if the player is for the first time carrying an item which the designer has marked as ''scored''.

5.5.2. Also if the player is for the first time inside a room which the designer has marked as ''scored'', provided there is light to see by.

5.5.3. The library groups score ranges into ranks, with names such as ''Beginner'' or ''Expert'' specified by the designer. If this feature is used at all then every possible score should correspond to one and only one rank.

5.6. The clock daemon records the time of day to the nearest minute. (Day, month and year are not modelled.)

5.6.1. Between one change of state and the next, the clock is normally advanced by one minute, but the designer (not the player) can arrange for this to be varied in play either to several changes per minute, or several minutes per change.

5.6.2. The designer can also change the time at any point.

5.6.3. Time passes at a constant rate for all objects, so that the player's measurement of the passage of time is the same as everybody else's.[17]

5.7. Time stops immediately when any daemon declares the player's death or victory.

5.7.1. Only rules provided by the designer will do this: the model world's normal rules are set up so that, whatever the player asks to do, time will continue indefinitely.[18]

6. Action

6.1. An action is a single impulse by the player to do something, which if feasible would take sufficiently little time to carry out that there would be no opportunity to change one's mind half-way or leave it only partly carried out.

6.1.1. An action "succeeds" if the activity in question does take place within the model world. If not, it "fails".[19]

6.1.2. Not all impulses are sensible or feasible. Some actions fail because circumstances happen to frustrate them, but others could never have succeeded in any circumstances.

6.1.3. Some actions (the so-called "group 3 actions") have a model in which, once the impulse is verified as being feasible, all that happens is that a message along the lines of "nothing much happens" is given.

[17] This is a restriction, though not because it ignores relativistic effects (at walking speeds these are of the order of 1 part in 10^{17}). Suppose the player enters a room where time runs slow and all actions take five times longer than they would elsewhere. This means that daemons which handle changes far away from the player also run five times slower than normal, relative to the model world's clock.

[18] Thus victory does not occur automatically when all scored objects are found and all scored rooms visited; death does not occur automatically when the player crosses from a dark room to another dark room, and so on.

[19] It does not necessarily follow that any objects will change their states: an action to look under something will result only in text being printed out, but this is successful because the looking under has become part of the history of the model world.

6.1.4. Some actions imply the need for other actions to take place first. In such cases the first action is tried, and only if this is successful will the second action be tried.

6.1.4.1. An action which requires an object to be held (such as eating) will cause a take action for that object.

6.1.4.2. A particular container, specified by the designer as the "sack object", is such that when the player is carrying the maximum legal number of items, any further take action causes the least recently taken item to be put into the sack first.[20]

6.1.4.3. A drop action for a piece of clothing being worn will cause a remove-clothing action first.

6.1.5. Other actions are recognised as being composites and so are split into a sequence of simpler constituent actions.

6.1.5.1. Actions involving multiple objects specified by the player as a collective batch ("take six buttons", "drop all") are split into one action for each object.

6.1.5.2. Emptying a container is split up into individual remove and drop actions.

6.1.5.3. Entering something which would require intermediate objects to be exited or entered is split into a sequence of exits and entrances.

6.2. An action can involve no objects other than the player, or else one other object of which the player is aware, or else two other objects of which the player is aware.[21]

6.2.1. The following actions fail if the player cannot touch the object(s) acted on: taking, dropping, removing from a container, putting something on or inside something, entering something, passing through a door, locking and unlocking, switching on or off, opening, closing, wearing or removing clothing, eating, touching, waving something, pulling, pushing or turning, squeezing, throwing, attacking or kissing, searching something.

6.2.2. The following actions fail if the player cannot see the object(s) acted on: examining, searching or looking under something.[22]

---

[20] Casual assumptions, in this case that all games have a single rucksack-like container, often make world models needlessly restrictive. Compare the Scott Adams game engine, in which one object is designated as "the lamp".

[21] The Inform parser will not generate actions concerning objects of which the player is unaware.

[22] Thus there is only one action requiring you to both see and touch the object acted on: searching.

6.3. The actions modelled are grouped under five headings below: actions of sense, alteration, arrangement, movement and communication. There is also one inaction: waiting, in which the player chooses to do nothing for the turn.

6.4. Actions of sense are those which seek information about the world without changing it: inventory, examining, consulting, looking, looking under, searching, listening, tasting, touching and smelling.[23]

6.4.1. "Look" describes only those parts of the room which can be seen. (In particular, concealed objects are omitted.) In addition, certain objects are "scenery" and are omitted from specific mention in room descriptions because, although visible, they are either too obvious to mention (such as the sky) or are mentioned already in the room-specific text.

6.5. Actions of alteration are those in which the player changes something without moving it: opening, closing, locking, unlocking, switching on and off, wearing and removing clothing, and eating.

6.5.1. A successful eating action results in the object being taken out of play.[24]

6.6. Actions of arrangement are those in which the player rearranges the spatial arrangement of things: taking, dropping, removing, inserting, putting on top of, transferring, emptying, emptying onto or into, pulling, pushing, turning and throwing.

6.6.1. Pulling, pushing and turning are only minimally provided: they are checked to see if the player can indeed carry out the action, but then nothing happens unless the designer writes code to make it happen, because the model doesn't include directions of pointing.

6.6.2. Actions of arrangement fail if the object is "static", meaning that it is fixed in place.

6.7. Actions of movement are those in which the player moves about: going, entering, exiting, getting off and pushing something from one room to another.

6.7.1. An attempt to enter a container or door fails if it is not open.

6.7.2. Only an enterable object or a door can be entered.

6.7.3. The player can only get off or exit from the enterable object currently holding him.

---

[23] Reading is not distinguished from examining: instead the consult action provides for looking things up in books.

[24] This is the only circumstance in which the rules for the model world destroy an object.

6.7.4. Certain enterable objects are ''vehicles''. Within a vehicle, movement actions are permitted as if the player were standing on the floor. Otherwise, no movements are permitted if the player is within an enterable object, except to enter or exit.

6.7.4.1. If the player travels in a vehicle, the vehicle object is also moved to the new room, and the player remains within it.

6.7.5. Certain objects are ''pushable'' and accompany the player on an otherwise normal movement. These, too, move to the new room.

6.7.6. An attempt to move through a concealed door fails as if the door were not there at all.[25]

6.8. Actions of communication are those in which the player relates to other people within the model world: giving, showing, waking somebody up, attacking, kissing, answering, telling, asking about and asking for something.

6.8.1. Actions of communication fail unless the object is animate, except that certain ''talkable'' objects can be addressed in conversation.

[25] I have no idea what this is doing in the Inform world model, but it seems to be there: perhaps the writing-room puzzle in the 'Curses' attic needed it. Andrew Plotkin: ''I've always said that the definition of concealed was ad-hoc and not well understood by anybody.''

§25 Extending and redefining the world model

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

In *The History of Zork*, Tim Anderson summed up the genesis of the 'Zork I' world model – and perhaps also its exodus, that is, its adaptation to other games – when he commented that "the general problem always remained: anything that changes the world you're modelling changes practically everything in the world you're modelling." Substantial changes to the world model often have profound implications and can lead to endless headaches in play-testing if not thought through. Even a single object can upset the plausibility of the whole: a spray-can, for instance, unless its use is carefully circumscribed. On the other hand, introducing whole categories of objects often causes no difficulty at all, if they do not upset existing ideas such as that of door, location, container and so on. For instance, the set of collectable Tarot cards in the game 'Curses' have numerous rules governing their behaviour, but never cause the basic rules of play to alter for other items or places.

. . . . .

In making such an extension the natural strategy is simply to define a new class of objects, and to take advantage of Inform's message system to make designing such objects as easy and flexible as possible. For example, suppose we need a class of Egyptian magical amulets, small arrowhead-like totems worn on the wrist and made of semi-precious minerals, with cartoon-like carvings. (The Ashmolean Museum, Oxford, has an extensive collection.) Each amulet is to have the power (but only if worn) to cast a different spell. Almost all of the code for this will go into a class definition called `Amulet`. This means that

```
if (noun ofclass Amulet) ...
```

provides a convenient test to see if an object `noun` is an amulet, and so forth. (This imposes a restriction that an object can't start or stop being an amulet in the course of play, because class membership is forever. If this restriction were unacceptable, a new attribute would need to be created instead, in the same way that the standard world model recognises any object with the attribute `container` as a container, rather than having a `Container` class.)

Suppose the requirement is that the player should be able to type "cast jasper amulet", which would work so long as the jasper amulet were being worn. It seems sensible to create an action called `Cast`, and this necessitates creating an action subroutine to deal with it:

```
[ CastSub;
  "Nothing happens.";
];
Verb 'cast' 'invoke' * noun -> Cast;
```

Nothing happens here because the code is kept with the `Amulet` class instead:

```
Class Amulet
 with amulet_spell "Nothing happens.",
      before [ destination;
          Cast:
              if (self hasnt worn)
                  "The amulet rattles loosely in your hand.";
              destination = self.amulet_spell();
              switch (destination) {
                  false: "Nothing happens.";
                  true: ;
                  default: print "Osiris summons you to...^";
                      PlayerTo(destination);
              }
              rtrue;
      ],
  has clothing;
```

Thus every `Amulet` provides an `amulet_spell` message, which answers the question "you have been cast: what happens now?" The reply is either `false`, meaning nothing has happened; `true`, meaning that something did happen; or else an object or room to teleport the player to.

From the designer's point of view, once the above extension has been made, amulets are easy to create and have legible code. Here are four example spells:

```
amulet_spell "The spell fizzles out with a dull phut! sound.",
amulet_spell [;
    if (location == thedark) {
        give real_location light;
        "There is a burst of magical light!";
    }
],
amulet_spell HiddenVault,
```

```
amulet_spell [;
    return random(LeadRoom, SilverRoom, GoldRoom);
],
```

An elaborate library extension will end up defining many classes, grammar, actions and verb definitions, and these may neatly be packaged up into an `Include` file and to be placed among the library files.

△△ Such a file should contain the directive `System_file;`, as then other designers will be able to `Replace` routines from it, just as with the rest of the library.

. . . . .

So much for extending the Inform model with new classes: the rest of the section is about modifying what's ordinarily there. The simplest change, but often all that's needed, is to change a few of the standard responses called "library messages", such as the "Nothing is on sale." which tends to be printed when the player asks to buy something, or the "Taken." when something is picked up. (To change every message, and with it the language of the game, see §34.)

To set new library messages, provide a special object called `LibraryMessages`, which must be defined *between* the inclusion of the `"Parser.h"` and `"Verblib.h"` library files. This object should have just one property, a `before` rule. For example:

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

This object is never visible in the game, but its `before` rule is consulted before any message is printed: if it returns `false`, the standard message is printed; if `true`, then nothing is printed, as it's assumed that this has already happened.

The `Jump` action only ever prints one message (usually "You jump on the spot."), but more elaborate actions such as `SwitchOn` have several, and `Take` has thirteen. The library's variable `lm_n` holds the message number, which counts upwards from 1. In some cases, the object being talked about is held

in `lm_o`. The messages and numbers are given in §A4. New message numbers may possibly be added in future, but old ones will not be renumbered.

An especially useful library message to change is the prompt, normally set to `"^>"` (new-line followed by >). This is printed under the action `Prompt` (actually a fake action existing for this very purpose). You can use this to make the game's prompt context-sensitive, or to remove the new-line from before the prompt.

• **EXERCISE 59**
Infocom's game 'The Witness' has the prompt "What should you, the detective, do next?" on turn one and "What next?" subsequently. Implement this.

△ `LibraryMessages` can also be used as a flexible way to alter the rules governing individual actions. Here are two examples in the guise of exercises.

• △**EXERCISE 60**
Under the standard world model (¶6.7.4 in §24 above), a player standing on top of something is not allowed to type, say, "east" to leave the room: the message "You'll have to get off . . . first" is printed instead. Change this.

• △**EXERCISE 61**
Under standard rules (¶6.6.1 in §24 above), a player trying to "push" something which is not `static` or `scenery` or `animate` will find that "Nothing obvious happens". Add the rule that an attempt to push a `switchable` item is to be considered as an attempt to switch it on, if it's off, and vice versa. (This might be useful if a game has many buttons and levers.)

. . . . .

The Library is itself written in Inform, and with experience it's not too hard to alter it if need be. But to edit and change the library files themselves is an inconvenience and an inelegant way to carry on, because it would lead to needing a separate copy of all the files for each project you work on. Because of this, Inform allows you to `Replace` any routine or routines of your choice from the library, giving a definition of your own which is to be used instead of the one in the standard files. For example, if the directive

```
Replace BurnSub;
```

is placed in your file *before the library files are included*, Inform ignores the definition of `BurnSub` in the library files. You then have to define a routine called `BurnSub` yourself: looking in the library file `"Verblib.h"`, the original turns out to be tiny:

```
[ BurnSub; L__M(##Burn,1,noun); ];
```

All this does is to print out library message number 1 for `Burn`, the somewhat preachy "This dangerous act would achieve little." You could instead write a fuller `BurnSub` providing for a new concept of an object being "on fire".

△△ Inform even allows you to `Replace` "hardware" functions like `random` or `parent`, which would normally be translated directly to machine opcodes. This is even more "at your own risk" than ordinary usages of `Replace`.

. . . . .

What are the implications of fire likely to be? One way to find out is to read through the world model (§24) and see how fire ought to affect each group of rules. Evidently we have just created a new possible internal state for an object, which means a new rule under ¶1, but it doesn't stop there:

1.4.4. Some objects are "flammable" and therefore in one of two mutually exclusive states, "on fire" and "not on fire".

2.4. If an object on fire is placed in a flammable container or on a flammable supporter, that too catches fire.

2.5. If a container or supporter is on fire, any flammable object within or on top of it catches fire.

4.3.3.1. Any object on fire provides light.

4.4.3. The player cannot touch any object on fire unless (say) wearing the asbestos gloves.

5.4.1. All flammable objects have a "lifespan", a length of time for which they can burn before being consumed. The each-turn daemon subtracts one from the lifespan of any object on fire, and removes it from play if the lifespan reaches zero.

One could go further than this: arguably, certain rooms should also be flammable, so that an object on fire which is dropped there would set the room ablaze; and the player should not survive long in a burning room; and we have not even provided a way to douse the flames, except by waiting for fires to burn themselves out. But the above rules will maintain a reasonable level of plausibility. ¶1.4.4 is provided by defining a new class of `Flammable` objects, which contains an `each_turn` routine implementing ¶5.4.1, and an `on_fire` attribute. The same `each_turn` can take care of ¶2.4 and ¶2.5, and can give the object `light` if it's currently `on_fire`, thus solving ¶4.3.3.1. But, while it would be easy to add simple rules like "you can't *take* an object which is on fire", ¶4.4.3 in its fullest form is more problematic, and means replacing the `ObjectIsUntouchable` routine. Giving any object on fire a `before` rule preventing the player from using any of the "touchy" actions on it would go some of the way, but wouldn't handle subtler cases, like a player not being

allowed to reach for something through a burning hoop. Nor is this everything: burning objects will need to be talked about differently when alight, and this will call for using the powerful descriptive features in Chapter IV.

● **REFERENCES**

'Balances' implements the 'Enchanter' trilogy's magic system by methods like the above. ●Approximately seventy library extensions have been contributed by members of the Inform community and more are placed at `ftp.gmd.de` with each month that goes by. Often short and legible, they make good examples of Inform coding even if you don't want to use them. Many are cited in ''references'' paragraphs throughout the book: here are others which seem more appropriate here. ●`"money.h"`, by Erik Hetzner, is a textbook case of a class-based extension to Inform providing a new aspect to the world model which doesn't much overlap with what's already there: notes and coinage. ●Conversely, Marnie Parker's `"OutOfRch.h"` exemplifies a change that needs to permeate the existing world model to be effective: it defines which areas of a location are within reach from which other areas, so that for instance a player sitting on a chair might only be able to reach items on the adjacent table and not a window on the far wall. ●In some graphical adventure games, interactivity sometimes cuts out at a significant event and an unchangeable movie animation is shown instead: this is sometimes called a ''cut-scene''. Such games sometimes allow the player to replay any movies seen so far, reviewing them for clues missed previously. `"movie.h"`, by L. Ross Raszewski, projects the textual version of movies like this, thus providing a framework for cut-scenes. ●`"infotake.h"`, by Joe Merical, shifts the Inform model world back to the style of 'Zork': printing Zorkesque messages, providing a ''diagnose'' verb and so on. ●Anson Turner's `"animalib"` retains the core algorithms of the Inform library (principally the parser and list-writer) but redesigns the superstructure of properties and attributes with the aim of a cleaner, more consistent world model. Although this alternative library is in its early stages of development, its code makes interesting reading. For instance, some child-objects represent items held inside their parents and others represent items on top of their parents. The standard Inform library distinguishes these cases by looking at the attributes of the parent-object – whether it is a `supporter` or a `container`. Contrariwise, `"animalib"` distinguishes them by looking at attributes of the child, so that the different children of a single parent can all be contained in different ways.