Chapter 5: Moving around

(Rooms and map connections • Floating objects • Regions • Light and darkness • Reachability, visibility, and scope • Doors and locks • Moving the player character • Path finding)

Rooms and map connections

So far, all our example games have had just one room. To add more, we declare some objects that have the (room $) trait. Room-to-room connections are defined using the (from $ go $ to $) predicate. The first parameter is the present room, the second parameter is a compass direction, and the third parameter is the neighbouring room in that direction.

The standard library provides twelve pre-defined directions: #east, #northeast, #north, #northwest, #west, #southwest, #south, #southeast, #up, #down, #in, and #out. Story authors can add more directions (such as starboard and port) by copying the rule definitions for one of the standard directions from the library, and editing them.

Here is a small, working game with three rooms:

```
#library
(room *)
```

```
#library
(room *)
(name *)          library
(from * go #east to #foyer)

#foyer
(room *)
(name *)          foyer
(from * go #west to #library)
(from * go #south to #study)

#study
(room *)
(name *)          study
(from * go #north to #foyer)

#player
(current player *)
(* is #in #library)
```

[Copy to clipboard]

Try navigating by compass directions, as well as EXITS and e.g. GO TO STUDY.

Rooms have names, and optionally dict synonyms, like any other object. In Dialog games, neighbouring rooms are in scope by default, although the player can't do all that much with them, except ENTER them. But commands such as EXITS have to be able to print the name of the room as an ordinary noun that's part of a sentence.

Of course, you are free to make use of the linguistic traits to affect how the room name is presented in various contexts. In particular, (singleton $) and (proper $) can be used to give room names a more rigid appearance:

```
#library
(room *)
```

```
#library
(room *)
(singleton *)
(name *) university library
(from * go #east to #foyer)

#foyer
(room *)
(singleton *)
(name *) grand foyer
(from * go #west to #library)
(from * go #south to #study)

#study

(room *)
(proper *)
(name *) Professor Stroopwafel's study
```

(from * go #north to #foyer)

#player
(current player *)
(* is #in #library)
[Copy to clipboard]

When the player is inside a room, the name of the room is usually displayed in the status bar, and as a bold header before the room description. This piece of text is called the *room header*. The default room header is simply the room name, with the first character converted to uppercase. But it is possible to override the (room header $) rule:

```
#belowcliff
(room *)
```

#belowcliff
(room *)
(singleton *)
(name *) area below the cliff
(room header *) Below the cliff
[Copy to clipboard]

This will make “Below the cliff” the room header, even though the room turns up as “the area below the cliff” in e.g. EXITS listings.

Sometimes, two or more directions point towards the same exit. For instance, from a rooftop, there might be a ladder leading down along the eastern wall of the building. Both DOWN and EAST should take the player to the location below the building, but the exit should only appear once in the EXITS listing. To achieve this, we regard one of the directions as secondary, and *redirect* it onto the other. Example:

```
(from #rooftop go
#down to
```

(from #rooftop go #down to #parkinglot)
(from #rooftop go #east to #down)
[Copy to clipboard]

which could lead to the following exchange:

> EXITS
Obvious exits are:
Down to the parking lot.

> E
You climb down.

Redirecting #in and #out is particularly useful.

When there's no exit in a particular direction, Dialog allows you to specify that an object is located that way. This makes it possible for the player to e.g. LOOK NORTH to examine a large mural on the wall there. This functionality is implied by the (from $Room go $Direction to $Target) predicate, whenever $Target is neither a room, a direction, nor a door (to be described shortly). Thus:

```
(from #rooftop go
#up to #sky)
```

(from #rooftop go #up to #sky)
[Copy to clipboard]

Such objects, like redirections, do not appear in EXITS listings.

What you are describing with (from $ go $ to $) are the so called *obvious exits* from a room. What actually happens when a player tries to move in a particular direction, depends on how the [leave ...] family of actions are handled. We will return to that in the chapter on actions; for now just keep in mind that the act of moving in a compass direction can be intercepted by story code, in order to block obvious exits, allow movement in non-obvious directions, trigger cutscenes, or anything else. But in the absense of any such intercepting code, the default behaviour of the [leave ...] actions (and [exits], and [go to $]) is to make use of the obvious exits.

Floating objects

Floating objects are objects that appear to exist in several rooms at once. This is an illusion, created by moving the floating objects whenever the player character moves. The movement is handled by the standard library, based on what you declare using the ($Room attracts $Object) predicate. Thus:

```
#wallpaper
(name *)
```

```
#wallpaper
(name *) wallpaper
(descr *) Brown and austere.
(#library attracts *)
(#foyer attracts *)
(#study attracts *)
```
[Copy to clipboard]

Objects around the perimeter of a room, such as doors and e.g. #sky in (from #rooftop go #up to #sky), are attracted automatically. Thus:

```
#floor
(name *)        floor
```

```
#floor
(name *)    floor
(singleton *)
(descr *)    Vinyl, with a marble pattern.
(from #library go #down to *)
(from #foyer go #down to *)
(from #study go #down to *)
%% The following rule definitions aren't necessary:
%% (#library attracts *)
%% (#foyer attracts *)
%% (#study attracts *)
```
[Copy to clipboard]

Regions

Rooms can often be classified into a number of conceptual *regions* (possibly by geographical proximity), such as “outdoors” or “in the dungeon”. Rooms that belong to the same region tend to share properties, such as what floating objects they attract.

In Dialog, regions are modelled using traits. Thus, we might create a trait (indoors room $) that inherits most of its behaviour from the (room $) trait, but also attracts a certain set of floating objects:

```
%% Every indoors-
room is a room.
```

```
%% Every indoors-room is a room.
%% Phrased differently, an object is a room given that it's an indoors-room:
(room *(indoors-room $))
#wallpaper
(name *)            wallpaper
(descr *)           Brown and austere.
((indoors-room $) attracts *)
#floor
(name *)            floor
(singleton *)
(descr *)           Vinyl, with a marble pattern.
(from (indoors-room $) go #down to *)
#foyer
(indoors-room *)
(name *)            grand foyer
(singleton *)
(from * go #south to #study)
#study
(indoors-room *)
(name *)            Professor Stroopwafel's study
(proper *)
(from * go #north to #foyer)
```

```
(from * go #out to #north)

#player
(current player *)
(* is #in #library)
```
[Copy to clipboard]

If you try it out, you'll find that it's possible to walk around and examine the floor and wallpaper from within either room.

For very simple regions, another option is to use slash expressions:

```
(#foyer/#study/#library attracts
```

```
(#foyer/#study/#library attracts #wallpaper)

(from #foyer/#study/#library go #down to #floor)
```
[Copy to clipboard]

Light and darkness

Light travels up and down the object tree. It can pass between a child and its parent, unless the parent is opaque and the child is #under it, or the parent is closed and opaque and the child is #in it. Openable objects are opaque by default.

An object is *illuminated* by another object if the latter provides light, and light can pass between the objects.

By default, rooms are assumed to contain ambient light, so they act as light sources. Hence, most objects are illuminated by default. But it is possible to disable the ambient lighting for any room, by adding a rule to the (inherently dark $) predicate:

```
#cave
(name *)        cave
```

```
#cave
(name *)    cave
(room *)
(inherently dark *)
```
[Copy to clipboard]

Now, if the player enters that room, they will be in darkness unless the room is illuminated by some other object that provides light:

```
#lamp
(name *)        lamp
```

```
#lamp
(name *)    lamp
(item *)
(* provides light)
```
[Copy to clipboard]

Rule definitions for the ($ provides light) predicate often contain conditions in the rule body. For instance, a flashlight might provide light when it is switched on:

```
#flashlight
(name *)
```

```
#flashlight
(name *)            flashlight
(item *)
(switchable *)
(* provides light) (* is on)
```
[Copy to clipboard]

Note: The standard library makes a multi-query to ($ provides light), in order to iterate over every object that currently provides light. Be sure to add asterisks to the rule body as required, for instance if you define a trait for light-providing objects:

```
($Obj provides light)
      *(lamp $Obj)
```

```
($Obj provides light)
```

```
 *(lamp $Obj)    %% The asterisk is crucial.
 ($Obj is on)
```
[Copy to clipboard]

The standard library provides a predicate, (player can see), that succeeds when the current player character is illuminated.

Reachability, visibility, and scope

An object is within *reach* of another object (such as the player character) when there is a path between them, via child-parent relations in the object tree, that doesn't pass through a closed object. Objects that are nested #under an object that is #wornby somebody other than the current player character are also considered out of reach. Finally, objects may explicitly be declared (out of reach $).

An object is *visible* to another object (such as the player character) when they are both illuminated, and there is a path between them, via child-parent relations in the object tree, that doesn't 1. pass through a closed, opaque object, or 2. pass underneath an opaque object. Openable objects are opaque by default.

Neither reach nor visibility extends across room boundaries, but doors and other objects that are located at the perimeter of the current room, using (from $ go $ to $), are automatically moved into the room.

To check whether an object is currently visible to the player, use (player can see $).

There is currently no simple, generic way to check whether an object is visible to some other object (e.g. a non-player character), because of the way floating objects and moving light sources are handled. But for a given story, it is often sufficient to make a pragmatic approximation, such as whether the observer and the object are in the same room.

Most actions require reachability. Of the ones that don't, some (e.g. LOOK IN) explicitly require visibility. Normally, anything that is reachable is also visible (but something that is visible might be in a closed, transparent container, and hence not reachable). But in darkness, objects tend to be reachable but not visible.

Under certain circumstances, when a player looks through a door (e.g. by looking in a compass direction), the name of the room on the other side is printed. But that is handled separately from the formal concept of visibility described here.

At any given time, a subset of the objects in the game world are considered to be *in scope*. These are the only objects that the player may currently refer to, i.e. the only objects that the parser will understand. The predicate ($ is in scope) can be used to check whether a given object is in scope, or, with a multi-query, to backtrack over every object in scope.

The default scope is everything that the player can see or reach, plus objects that are marked out of reach but would be reachable otherwise. If the current room is in scope and the player can see, neighbouring rooms are also added to the scope.

If the player cannot see, the intangible object #darkness (responding to DARKNESS and DARK) is automatically added to the scope. By default, the player can't do much with this object except examine it, which invokes (narrate darkness), printing “You are surrounded by darkness”.

It is possible to add other objects to the scope using the predicate (add $ to scope), typically with some condition, like this:

```
(add #mother to
scope)
```
```
(add #mother to scope)
 (current room #phonebooth)
```
[Copy to clipboard]

That rule allows the parser to recognize e.g. CALL MOTHER when the player is in the phone booth.

Doors and locks

Map connections can also involve *doors*. A door is a gatekeeper object (representing a physical door, an opening, or something else entirely) that either blocks or allows passage.

Whether a door admits passage or not, and whether it's possible to peek at the room on the other side, is determined by the predicates ($ blocks passage) and ($ blocks light), respectively. In the standard library, they are defined as follows:

```
($Door blocks
passage)
```
```
($Door blocks passage)
 ($Door is closed)

((opaque $Door) blocks light)
 ($Door is closed)
```

[Copy to clipboard]

Openable objects are closed and opaque by default. If you are implementing a physical door, remember to declare it openable, (openable *), and optionally to specify that it starts out open, (* is open).

The standard library provides two mechanisms for setting up door connections. The low-level method involves setting up two rules, one for the predicate (from $Room go $Direction to $Door) and one for its companion (from $Room through $Door to $Target). The high-level method is to use an access predicate that defines both rules in one go: (from $Room go $Direction through $Door to $Target). Let's build a door using the high-level method:

```
#foyer
(room *)
```

```
#foyer
(room *)
(name *) grand foyer
(singleton *)
(from * go #south through #door to #study)

#study
(room *)
(name *) Professor Stroopwafel's study
(proper *)
(from * go #north through #door to #foyer)
(from * go #out to #north)

#door
(door *)
(openable *)
(name *) small door
(descr *) It's a perfectly ordinary, but small, door.

#player
(current player *)
(* is #in #library)
```

[Copy to clipboard]

Doors usually appear in (from $ go $ to $) rules, and are therefore automatically treated as floating objects. So in the above game, you'll be able to EXAMINE DOOR, OPEN DOOR, CLOSE WOODEN etc. from either side of the door.

Doors can be *locked*. An object that is locked, ($ is locked), can't be opened (by the default behaviour of the [open $] action). But a *lockable* object, (lockable $), can be locked or unlocked with the right key. By trait inheritance, lockable objects are also openable. They start out locked and closed, unless you specify otherwise.

Keys are associated with lockable objects using the predicate ($ unlocks $).

```
#door
(door *)
```

```
#door
(door *)
(lockable *)
(name *) small door
(descr *) It's a perfectly ordinary, but small, door.

#key
(item *)
(name *) small key
(* unlocks #door)
```

[Copy to clipboard]

The standard actions are set up so that an attempt to walk through a closed door first triggers an automatic [open $] action, which in turn may trigger an automatic [unlock $ with $] action if the door was locked. But the latter only happens if the player is holding the right key at the time.

Now that we know about locked doors and keys, we can create a small, playable puzzle game:

```
#library
(room *)
```

```
#library
(room *)
(singleton *)
```

```
(name *) university library
(look *)  What a strange library. There's just a rug in here.
          (notice #rug)
          The exit is east.
(from * go #east to #foyer)
(from * go #out to #east)

#rug
(name *) rug
(* is #in #library)

#key
(item *)
(name *) small key
(descr *) It's a small key, of the kind that unlocks doors.
(* is #under #rug)
(* unlocks #door)

#foyer
(room *)
(singleton *)
(name *) grand foyer
(look *)  It's a grand, grand foyer.
          The library is west from here, and a
          (if) (#door is locked) (then) locked (endif)
          door leads south.
(from * go #west to #library)
(from * go #south through #door to #study)
(from * go #in to #south)

#study
(room *)
(name *) Professor Stroopwafel's study
(look *)  You solved the mystery of the locked door!
          (game over { You win! })
(proper *)
(from * go #north through #door to #foyer)
(from * go #out to #north)

#door
(door *)
(lockable *)
(name *) small door
(descr *) It's a perfectly ordinary, but small, door.
          It is currently
          (if) (* is locked) (then)
          locked.
          (else)
          unlocked.
          (endif)

#player
(current player *)
(* is #in #foyer)
```

[Copy to clipboard]

Moving the player character

The standard library uses a few global variables internally, of which (current player $) is particularly noteworthy. Story code may query this variable at any time, but mustn't update it directly using (now); that would confuse the library. The proper way to change the current player character is to make a query to (select player $).

But while you're not allowed to modify the (current player $) variable directly from within story code, you are expected to supply an initial value for it:

```
(current player #me)
```

(current player #me)
[Copy to clipboard]

Likewise, it is straightforward to define the initial location of the player character:

```
(#me is #in #study)
```

(#me is #in #study)
[Copy to clipboard]

But the location of the current player character must be changed with a query to either (move player to $Relation $Parent) or (enter $Room). The latter also prints the description of the new room (by invoking the [look] action).

Another global variable used by the library is (current room $). From the point of view of the story author, this could just as well have been a regular predicate that traverses the object tree in order to find the room that's currently enclosing the player character. But it is a global variable for performance reasons. When the player character is moved properly, by querying (move player to $ $) or (enter $), the value of this variable is updated to reflect the new location. Another thing that happens is that floating objects are moved into position.

The library uses a helper predicate called (update environment around player) to carry out the updates described above. Occasionally, it can be useful to query this predicate directly from story code.

Path finding

The standard library predicate (shortest path from $Room1 to $Room2 is $Path) computes the shortest path from $Room1 to $Room2, by considering the obvious exits listed in (from $ go $ to $) and (from $ through $ to $). The result is a list of directions.

The predicate (first step from $Room1 to $Room2 is $Direction) computes the same path, but returns only the first step. This is functionally equivalent to (shortest path from $Room1 to $Room2 is [$Direction | $]), but slightly faster and more memory efficient.

The computed path only includes visited rooms, and doesn't pass through closed doors. But it is straightforward to modify the library to relax those conditions.

Onwards to “Chapter 6: Actions” • Back to the Table of Contents