Block 2 Challenge: Building a Dungeon Game

Tony Clark

9th December 2013

1 Overview of the Challenge

The Block 2 Challenge is to construct a game that allows a player to explore a dungeon and to interact with the things that are found there. The purpose of the challenge is to provide you with the opportunity to demonstrate many of the SOBS from both Block 1 and Block 2. The purpose is **not** to produce a working dungeon game (although it would great if you can do so and if you do it will provide you with the opportunity to demonstrate some of the more excellent SOBS).

You will be provided with a series of working games in Racket. This document describes each game and includes exercises that you should attempt. You do not need to reach the end of the exercises, because we are not testing you. The exercises will provide you with the opportunity to demonstrate SOBS and will teach you about important Computer Science topics. This document is organised so that later sections introduce more advanced topics.

The Block 2 Challenge is to demonstrate your understanding of data structures, languages and programming through the construction of a dungeon game. The game can be constructed using the Racket code we have provided as part of this document. The code we provide is playable, but you may want to extend it in order to be a challenging game. Of course, you may choose to build your own dungeon, but try the exercises in this document first.

You may choose to meet the Block 2 challenge at *threshold*, *typical* or *excellent* levels. We have organized the material to support these activities:

threshold Work your way through this document and try the exercises. You don't have to do *all* the exercises, try as many as you can before moving on to the next section. You don't need to work through all of the sections. The title of each section names the file that contains the associated Racket code for the game. Each file is complete and can be loaded into DrRacket and the game will start when you type `(play)`. This will provide you with opportunities to demonstrate many of the Block1 and Block 2 SOBS.

typical Extend one of the supplied dungeon games. Several of the exercises ask you to extend the supplied code in various ways. You might like to think of your own extension to make the game more interesting.

excellent Write your own dungeon. Racket provides graphics libraries and many other features that can be used to make the game interesting to play.

The files associated with the challenge are, in order, as follows:

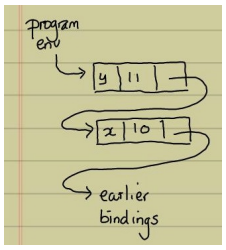
- `dungeon_rooms.rkt`
- `moving_about.rkt`
- `monsters_attack.rkt`
- `command_language.rkt`
- `items.rkt`

2 A Review of Data Diagrams

When names are introduced in a Racket program a *binding* is added to an *environment*. There is a top-level environment that is extended by a `define`. For example, after executing the following two lines:

```
(define x 10)
(define y (+ x 1))
```

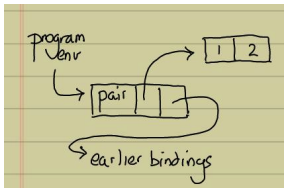
the program environment is:



When drawing diagrams, symbols, numbers, strings and booleans are just drawn as themselves. The empty list is drawn as `()` and cons-pairs are drawn as a box with compartments for the first (car) and rest (cdr). If the contents of the compartments are simple then we just write them in the compartment. Here is a simple pair:

```
(define pair (cons 1 2))
```

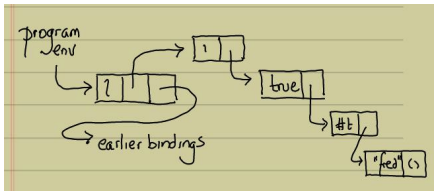
producing:



If the contents are structured and we draw a directed edge from the compartment to the structure. A list links cons-pair structures. For example:

```
(define l (list 1 'true #t "fred"))
```

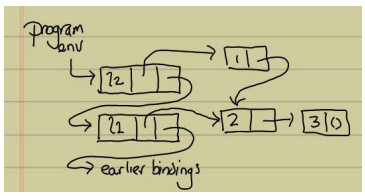
results in the program environment:



Parts of structured values might be shared. For example:

```
(define l1 (list 2 3))
(define l2 (list 1 l1))
```

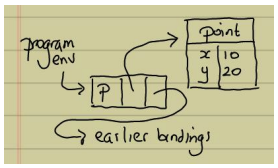
results in sharing:



Structures are drawn as boxes with their field names:

```
(struct point (x y))
(define p (point 10 20))
```

producing the following environment:



Exercise 2.1 *In order to review your understanding of Racket data diagrams try the following exercises:*

1. *What is the resulting program environment after performing the following:*

```
(define x 1)
(define y (cons x x))
```

2. *If Racket did not contain cons-pairs, you could implement them using structures. The structure definitions might be defined as follows:*

```
(struct pair (first rest))
(struct empty ())
```

   Given this definition write a program that constructs a list using `pair` *and* `empty` *containing the numbers* `1`, `2` *and* `3` *in that order. Draw out the resulting structure.*

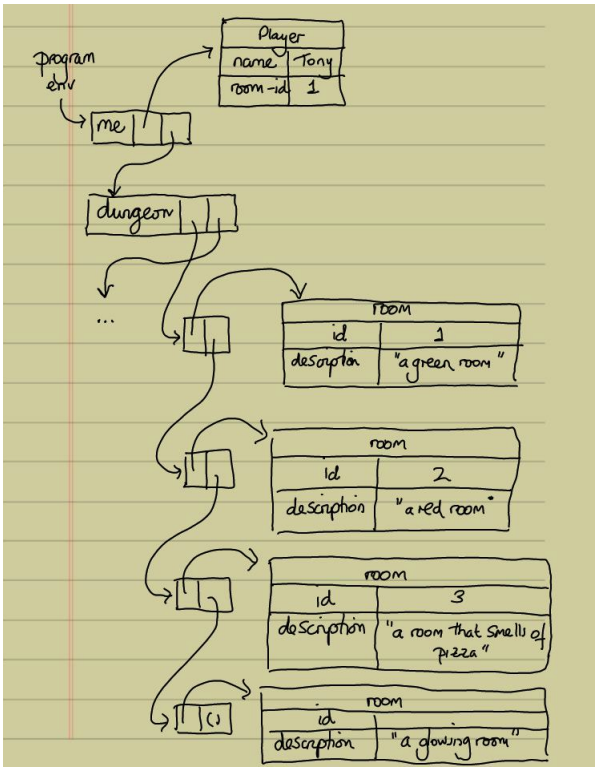


Figure 1: An Initial Dungeon State Design

3 Dungeon Rooms (dungeon_rooms.rkt)

A dungeon needs to have a collection of rooms. Each room has a description informing the player where they are. A player has a name and a location. We need to decide how to represent the linkage between a player and the location in the dungeon. If we give each room a unique identifier then we can reference the room using the identifier.

Figure 1 shows the design for a simple dungeon state. The program has two definitions for the names `me` and `dungeon`. The value of the name `me` is a player with two fields called `name` and `room-id`. The structure shows that the player is called `Tony` and is in the room with identifier `1`.

The value of `dungeon` is a list of rooms. Each room has an identifier and a description. The figure shows a dungeon with 4 rooms. As drawn, the state shows that `Tony` is in a green room.

Exercise 3.1 *Diagram 1 shows a particular arrangement of player and dungeon.*

1. *Modify the diagram to show an extra room. The room should have a new identifier. Think of a suitable description for the room.*
2. *Modify the diagram to show* `Tony` *in a room that smells of pizza.*

Having designed a simple state for the system we need to implement it in Racket. Here is the code for the structures used in the diagram:

```
; A RoomId is a Number
; A Room is (room RoomId String)
; A Player is (player Symbol RoomId)
```

```
(struct room (id description))
(struct player (name room-id))
```

The dungeon is created as a list of rooms:

```
; dungeon : List(Room)
(define dungeon
  (list
   (room 1 "a green room")
   (room 2 "a red room")
   (room 3 "a room that smells of pizza")
   (room 4 "a glowing room")))
```

A player is created in one of the rooms:

```
; me : Player
(define me (player 'Tony 1))
```

At any point in the game the player is contained in room. This information is represented by the value of the `room-id` field in `me`. Sometimes we will want to get to the room information, for example when we want to print out a description of the player's current room. In order to do that we need to get from the room identifier (contained in `me`) to the room description (contained in the room structure in `dungeon`). In order to do that we define a *function* called `get-room` that maps from room identifiers to room structures. The idea is that `(get-room id)` returns the room in the dungeon that has identifier `id`.

The function is *partial* because room identifiers are numbers and not all numbers will be identifiers for rooms. Therefore we need to define what happens when `get-room` is applied to a number that is not associated with a valid room. It does not matter what value is returned so long as we can distinguish it from a valid room. In this case we will return `#f` because it allows us to use the Racket builtin function `findf` as follows:

```
; Room? is one of:
;   Room
;   #f
; get-room : RoomId -> Room?
(define (get-room id)
  (findf (λ (room) (= (room-id room) id)) dungeon))
```

The function `findf` takes a boolean valued function `p` and applies it to each element of a list `l` in turn. When `p` returns `#t` then `fundf` returns the corresponding element, otherwise if `findf` reaches the end of the list then it returns `#f`. Once loaded, the following is an interaction with DrRacket:

```
> (get-room 1)
#<room>
> (get-room 5)
#f
>
```

Exercise 3.2 *The (partial) function* `get-room` *maps room identifiers to rooms.*

1. *Add an extra room to the dungeon so that* `get-room` *returns a room when supplied with the room identifier* `5`.
2. *Suppose that you want* `get-room` *to return the symbol* `'no-room` *when it is supplied with a room identifier that does not exist. Define a new function* `get-room2` *that has this behaviour.*
3. *Suppose that you did not want to use the builtin* `findf` *function. You might use recursion to do the job. Using recursion or otherwise, redefine* `get-room` *so that it does not use* `findf`.

So far we have defined the data that represents the state of our dungeon game. Now we need to define how to play it using the data. Our first version of the game will be pretty boring (in fact a player cannot do anything except stand still), but we will incrementally add features so it gets better.

Most programs have a single entry point. In Racket this is a function that is called to start the program. For the dungeon game this will be the function called `play`. The function does not have any arguments because we do not need to tell it anything in order to start playing since it already knows what the dungeon looks like and where the player is.

We need to decide what the key steps will be when the game starts. It is polite to say *hello* to the player when the game starts and *goodbye* when the player leaves. In between the game will allow the player to enter commands and will do something in response to each of the commands. Therefore, our entry point will be:

```
(define (play)
  (say-hello)
  (command-loop)
  (say-goodbye))

(define (say-hello)
  (printf "Welcome to the dungeon ~a~%" (player-name me)))

(define (say-goodbye)
  (printf "Bye~%"))
```

Exercise 3.3 *The definition of* `say-goodbye` *is a bit rude since it does not use the players name. Modify the definition of* `say-goodbye` *to use the player's name, as in Bye Tony.*

The game will allow a player to type commands that cause something to happen in the dungeon. We will let this continue until the player quits the game, or some other event happens to force the game to stop (the player might run out of food for example). The player enters a command, it is performed, and then there are two possible outcomes: (1) do it again, or (2) the game stops. It will depend on the command whether (1) or (2) happens. Therefore, we will implement a `command-loop` that performs a test on the outcome of performing a player command. Depending on the outcome of the test, the loop will go round again, or will stop:

```
(define (command-loop)
  (describe-current-location)
  (when (perform-player-command)
    (command-loop)))
```

If the function `perform-player-command` returns `#t` then the function `command-loop` is called again, otherwise `command-loop` just returns.

We now need to describe the player's current location. Remember `get-room`? We can use that to get the description of the current location. But remember that `get-room` is partial and that we should deal with the (hopefully unlikely) case where the player winds up in an illegal location:

```
(define (describe-current-location)
  (let ((room (get-room (player-room-id me))))
    (if (room? room)
        (printf "You are in ~a~%" (room-description room))
        (printf "I have no idea where you are!~%"))))
```

Finally, we need to design a *command language*. Each sentence in the command language must be recognized by the `command-loop` function and used to determine what should happen. Let's keep it *very* simple to start off with. The only commands in our command language will be `quit` that causes the program to stop. Even though it is very simple, we will give our command language a proper definition:

```
Command Language ::=
  'quit
```

The definition above states that the only command in our language is `quit` and that it is supplied as a symbol. Having decided on the command language we can define a `perform-player-command` function. How do we get the command? Racket conveniently provides a `read` function that reads the next value from the input and returns it. So a nice simple way to get commands and process them is as follows:

```
1 ; perform-player-command -> Boolean
2 (define (perform-player-command)
3   (let ((command (read)))
4     (cond ((equal? command 'quit) #f)
5           (#t (printf "Huh?~%")
6               #t))))
```

It is worth looking at `perform-player-command` in more detail. Line 2 reads a command from the player and calls the return value `command`. The `cond` expression checks whether the command is legal. In this case we only have a single command `'quit` and this is detected on line 3. The function `perform-player-command` must return a boolean value `#t` if the player is allowed to continue with the game and `#f` if the game is to end. In the case of the `'quit` command the function returns `#f` as shown on line 3.

If no legal commands are recognised then `perform-player-command` must do *something*. It should inform the player they typed in something silly and allow the game to continue. This is what happens on lines 4 and 5.

We now have a complete game. If you load the game into DrRacket and type `(play)` then it will start. Of course, it is a bit boring at the moment because we have a single command `quit`:

```
1 > (play)
2 Welcome to the dungeon Tony
3 You are in a green room
4 help
5 Huh?
6 You are in a green room
7 quit
8 Bye
9 >
```

Now we are up and going, the exercises will give you a chance to extend the game in various ways.

Exercise 3.4 *Extend the game to do the following:*

1. *Add a new command* `help` *that prints out the commands that are available to the player.*
2. *In* `say-hello` *print out the number of rooms that the player has available to explore. For the example dungeon this might be:* `The dungeon has 4 rooms for you to explore.`
3. *If a game-turn occurs each time a player command is processed, print out the number of turns that have happened at the end of the game in* `say-goodbye`.
4. *When we define the state of a system we must consider both system states that make sense and those that do not. These are often expressed as conditions that must hold for the state to make sense and are called* state invariants. *One state invariant for the dungeon is that all rooms must have a unique identifier,* i.e., *the numbers used as room ids must be different for all rooms in the dungeon. Write a predicate called* `unique-room-ids?` *that takes a dungeon state and returns* `#t` *when the rooms in the dungeon all have different room ids and* `#f` *otherwise.*

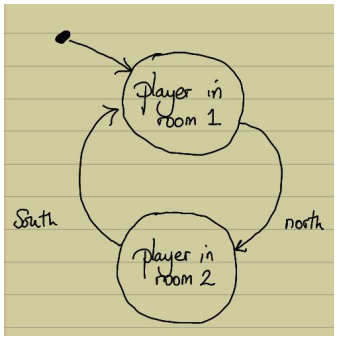


Figure 2: Moving Around the Dungeon

4 Moving About (moving_about.rkt)

The current version of the dungeon from the previous section contains a number of rooms - but we cannot explore them because there is no player command to move from room to room. This section is all about adding the `move` command to the game.

When designing the dungeon movement we need to decide how the player moves about. We will use the points of the compass: `north`, `south`, `east` and `west`. How many exits should each room have? Let's go for 1 or more.

Exercise 4.1 *Why should we insist that rooms have at least 1 exit? On the other hand, what use might a room be if it has 0 exits?*

An exit is designated by a direction and leads to another room. We might insist that if an exit from room `A` leads `north` to room `B` then there should be an exit from room `B` that leads `south` to room `A`.

Exercise 4.2 *Why should we insist that the exits work this way? On the other hand, what would happen if exits did not work together like this?*

Now that we have decided on the directions, we must design how the player moves around. Consider the simple state machine shown in figure 2. The player always starts in room 1. If they go `north` when they are in room 1 then they wind up in room 2. If they go south in room 2 they wind up in room 1 again.

Exercise 4.3 *Obviously, figure 2 is a very simple dungeon. Suppose you want to design a dungeon with 4 rooms and where all rooms have 2 exits. If all rooms are reachable from the starting room, show the state machine as a diagram.*

The states shown in figure 2 can be represented as data in a Racket program. The transitions in the machine represent the commands that are entered by the player and then processed by the game. Since the transitions go between different states, the state of the data must change in some way. firstly,

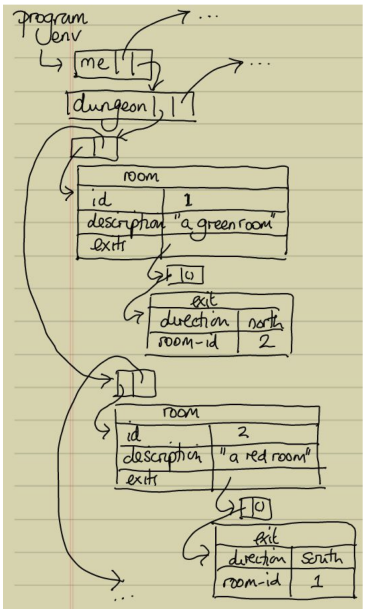


Figure 3: Adding Exits to Rooms

we design a data representation for a game stage and then we will look at how it will change when a command is processed.

When designing a data structure in this way, we do not need to write it all out. Often a data structure is made up of repetitions of the same type of element, for example a list of rooms. We only need to write out the part of the data and, providing we have shown all the different types of data involved, we can use ∴ to represent the rest because it is *more of the same*. When designing a data structure it is usual to write out some examples. Providing we choose examples that are representative of typical situations then we can generalise from them.

Figure 3 shows a particular dungeon state design for rooms with exits. It only shows part of the dungeon state because we already saw the structure of the player in figure 1 and the rest of the rooms are similar. As you can see from the sketch, the rooms have been extended with a list of exits. An exit contains a `direction` field and a `room-id` field that are used to describe where the exit leads to. Therefore, room `1` leads `north` to room `2` and room `2` leads `south` to room `1`.

Exercise 4.4 *Given the following specification for rooms, draw out the dungeon state:*

room 1 *is a green room that leads north to roomw 2 and east to room 3.*

room 2 *is a red room that leads south to room 1.*

room 3 *is a room that smells of pizza. Exits leads west to room 1 and south to room 4.*

room 4 *is a glowing room that leads north to room 3.*

Remember that the state of the game includes a player structure that contains the room identifier for the current location of the player. If rooms have exits then a player can move about from room to room. Figure 4 shows how the state of the data changes when the player who is in room 1 goes `north` to room 2. Imagine the diagrams from figure 4 being the labels on the nodes from figure 3 and you have got the general idea.

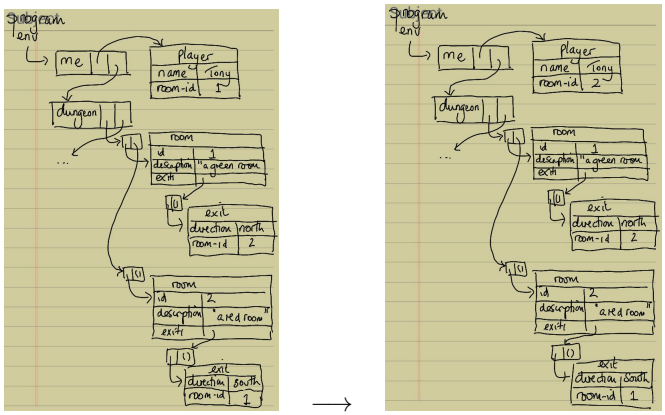


Figure 4: Player in room 1 goes north to room 2

Note that since the dungeon data structure does not change (yet!), when we write out these diagrams to describe how the system behaves, we need only write out the dungeon data structure once and and then just use `...` if we want since it should be clear what we mean. In general the rule when designing behaviour in this way is: if it does not change then just write it out once (perhaps using `...` for those parts that are *more of the same*).

Having designed the new dungeon state we can modify the code. The first thing to do is to set up some types and structures. A direction is represented as a symbol and the state is given by the following Racket structure definition:

```
; A Direction is one of:
;   'north
;   'south
;   'east
;   'west

; A RoomId is a Number

; A Room is (room RoomId String List(Exit))
; A Player is (player Symbol RoomId)
; An Exit is (exit Direction RoomId)

(struct room (id description exits))
(struct player (name room-id) #:mutable)
(struct exit (direction room-id))
```

Note that we have declared the `player` structure to be `#:mutable`. This means that we can use a procedure `set-player-room-id!` to move the player from one room to another. This is the procedure that will implement the state changes shown in figure 4 and 2.

Now we can construct a dungeon. This is the same as before except that we add a list of exits to each room:

```
(define dungeon
  (list
   (room 1 "a green room" (list (exit 'north 2) (exit 'east 3)))
   (room 2 "a red room" (list (exit 'south 1)))
   (room 3 "a room that smells of pizza" (list (exit 'west 1) (exit 'south 4)))
   (room 4 "a glowing room" (list (exit 'north 3)))))
```

A dungeon is legal when:

1. Every room has at least one exit.
2. Each exit from a room `X` in direction `d` leading to room `Y` has an exit `opposite(d)` leading from `Y` to `X`.

Exercise 4.5 *For the definition of the dungeon structures in Racket define two separate invariant checks for (1) and (2).*

The definition of `me`, `get-room`, `play`, `command-loop`, `say-hello` and `say-goodbye` is the same as before.

To handle the new command we need to be able to read it in. Unlike `quit`, a move command includes some extra information: the direction. An easy way to do this is to supply the move command as a list of two elements containing the word `move` followed by the direction, as in `(move north)`. Later we will remove this restriction, but it will take a bit more work.

The extended version of `perform-player-command` is:

```
(define (perform-player-command)
  (let ((command (read)))
    (cond ((equal? command 'quit) #f)
          ((and (list? command) (eq? (car command) 'move))
           (move-player (cadr command))
           #t)
          (#t (printf "Huh?~%")
              #t))))
```

This calls the procedure `move-player` to move a player from the current room to a new room via one of the current exits. In order to do this we need to work out the room that the player is currently in; we know how to do this using `get-room` from before. We will need a new predicate `has-exit?` that checks whether a room has an exit with a supplied direction:

```
; has-exit : Room Direction -> Boolean
(define (has-exit? room direction)
  (findf (λ (e) (eq? (exit-direction e) direction)) (room-exits room)))
```

and a procedure that gets a room id from a room and a direction:

```
; exit-leads-to : Room Direction -> RoomId
(define (exit-leads-to room direction)
  (exit-room-id
    (findf (λ (e) (eq? (exit-direction e) direction))
           (room-exits room))))
```

and define the command processor:

```
(define (move-player direction)
  (let ((room (get-room (player-room-id me))))
    (if (has-exit? room direction)
        (set-player-room-id! me (exit-leads-to room direction))
        (printf "You cannot go ~a from here~%" direction))))
```

Here is an example of playing the game:

```
> (play)
Welcome to the dungeon Tony
You are in a green room
(move north)
You are in a red room
(move south)
You are in a green room
(move east)
You are in a room that smells of pizza
quit
Bye Tony
>
```

Exercise 4.6 *Now that the game allows you to move around, implement the following extensions:*

1. *When a room is described by* `describe-current-location` *it just includes the description. It would be useful if the player is also told where the exits are, for example:*

   You are in a green room. There are exits: north and east.

2. *Implement a command* `teleport` *that uses the following function to move the player to a random room in the dungeon:*

```
(define (teleport-player)
  (let ((room (list-ref dungeon (random (length dungeon)))))
    (set-player-room-id! me (room-id room))))
```

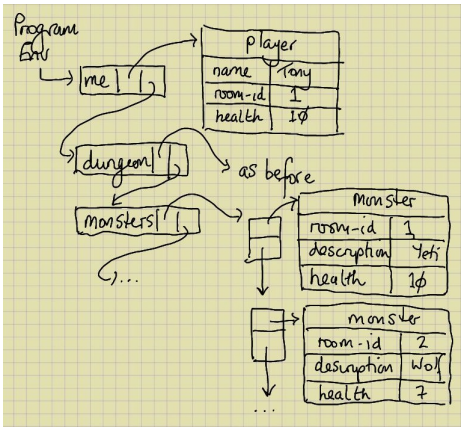


Figure 5: Monsters In the Dungeon

5 Monsters Attack! (monsters_attack.rkt)

So far, our game is a bit tame. What we need is some element of challenge. Any self respecting dungeon game includes dangerous monsters. Our next step is to add monsters to the game. You can hit the monsters, but be careful they fight back! If we add health to a player and introduce monsters with health then when a player or monster hit each other then the health gets reduced. If the health reduces to 0 then the player or monster dies. Killing a monster removes it from the game, killing the player ends the game.

We need to change the state of the dungeon in order to support the monsters. Here are the things we will need:

- A player needs some `health` which is a number. The player starts at full health which is reduced if the player is hit by a monster.
- The dungeon needs some monsters. Like the player, each monster will be in a particular room. A monster will have a description that can be used when printing out a description and when the player needs to reference the monster. For example `Yeti` might be a monster description.
- Like a player, a monster will need some `health` which will be a number. A monster starts of at full health which is reduced when the monster is hit. When a monster's health is reduced to 0 then the monster is dead and is removed from the game.

Figure 5 shows the state of the dungeon containing a player, some rooms and some monsters. The player has a name, a room identifiers and some health. The dungeon is a list of rooms as before and is not shown on the diagram. The monsters is a list of monster structures. Each monster structure contains the room identifier, a description and the current health of the monster. The diagram shows two monsters a `Yeti` and a `Wolf`. The `Yeti` has 10 health points and and `Wolf` has 7. The `Yeti` is in room 1 and the `Wolf` is in room 2. In general, there may be any number of monsters in the dungeon, so the tail of the monsters list is shown as `...` in the diagram.

Given this structure for the state of the game we can see what happens when we try out the game actions associated with monsters and players. The first action involves attacking monsters. When a player is in the same room as a monster they can attack it. If the attack is successful then some damage will be done to the monster. Figure 6 shows the result of a successful attack by `Tony` on the `Yeti`. Notice that the health points of the `Yeti` have been reduced by 2 points. So the `Yeti` is still alive, but is getting weaker.

Exercise 5.1 *Can* `Tony` *attack the* `Wolf`*? Give reasons for your answer.*

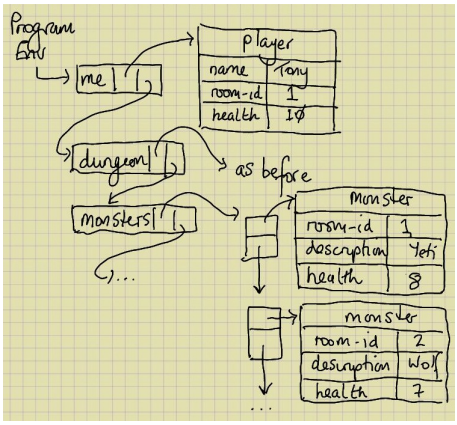


Figure 6: Player Hits Monster

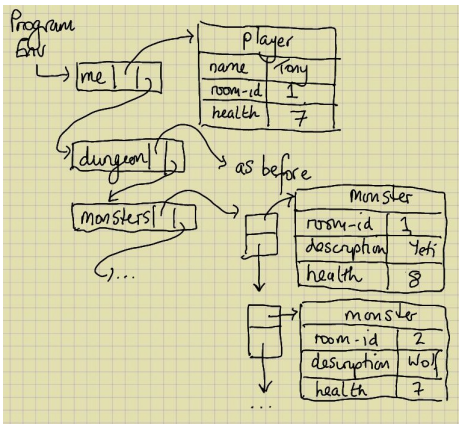


Figure 7: Monster Hits Player

Another action that can occur is when a monster attacks a player. Again, the monster must be in the same room as the player in order to attack them. If the attack is successful then damage is done just like in the case when the player attacks the monster. Figure 7 shows the result of a successful attack by the `Yeti` on the player. The health-points of `Tony` have been reduce by 3.

Player and monster attacks can continue so long as they are in the same room. We already have a player command that allows the player to move from one room to another providing there is an exit available. Presumably, a player will leave if they think the monster is winning the fight. We don't have an action that allows the monster to run away (but there is an exercise later that requires you to add this feature).

We must deal with two further situations arising from fighting with monsters: a successful attack from the player that kills the monster; a successful attack from a monster that kills the player. When a monster is killed it must be removed from the game. All the monsters in the game are held in the list that is the value of the variable `monsters`. Therefore, if we ant to remove a monster from the game we just remove it from the list. Figure 8 shows the result of a successful attack on the `Yeti` that causes its health-points to become `0`. Notice that the value of the `monsters` variable is now the tail of the original list, causing the `Yeti` to be removed, but leaving all other monsters in the game.

Suppose now, having successful dispatched the `Yeti`, `Tony` feels brave and seeks out more monsters. We already know what happens to the game state when a player moves. Figure 9 shows the result of the player moving `north` to room 2 where they encounter the `Wolf`.

Suppose that a fierce battle ensues and that eventually, after a series of successful attacks, the `Wolf`

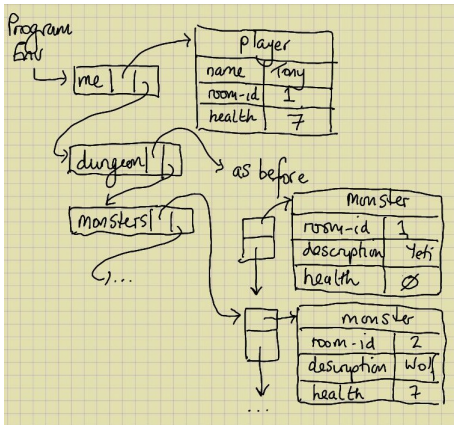


Figure 8: Monster Dies

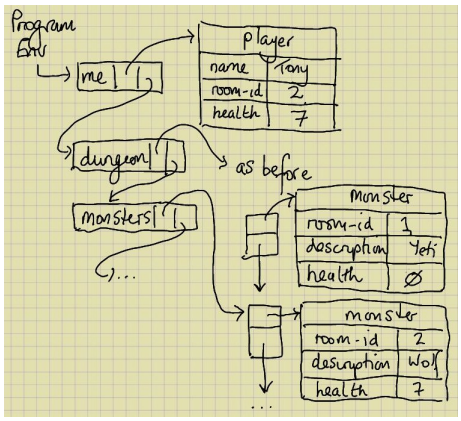


Figure 9: Player Moves

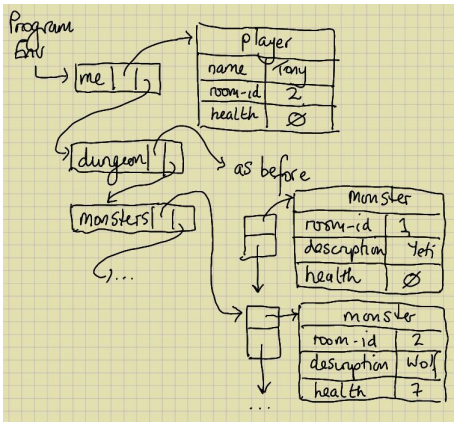


Figure 10: Player Dies

wins the fight and the player dies. Figure 10 shows the situation where the player's health-points have been reduced to 0. This is an important game state because it signals the end of the game. It has a special name and is called a *terminal state* of the game because no further state changes can happen.

Exercise 5.2 *Draw out diagrams for the following:*

1. *Suppose that the player in figure 5 lost the fight with the* `Yeti`*. Draw out a suitable terminal state for this situation.*
2. *Two monsters in different rooms and a player in a third room.*
3. *Two monsters attacking a player.*
4. *A state change where a player flees from a monster.*

Having designed the state of the dungeon we can now start to modify the code. We will just show the differences between the previous version and the game that includes monsters. The first thing to do is to design the Racket data structures and types that we will use:

```
; A Health is a Number
; A Player is (player Symbol RoomId Health)
; A Monster is (monster RoomId Symbol Health)

(struct player (name room-id health) #:mutable)
(struct monster (room-id description health) #:mutable)

; monsters : List(Monster)

(define monsters
  (list
   (monster 1 'Yeti 10)))
```

The global `monsters` is a list of monsters that are currently in the dungeon. The example above shows a dungeon with a single Yeti in room 1 with 10 health-points.

Exercise 5.3 *Add more monsters to rooms in the dungeon. Use interesting monster descriptions.*

By including monsters into the state of the dungeon we have introduced more possibility for error. For example, when the game starts each monster must have more than 0 health-points and must have a room identifier for a room that exists in the dungeon. We can write a predicate that is used at the start of the game to check whether the dungeon state is correct:

Exercise 5.4 *Write the functions* `each-monster-is-in-legal-room?` *and* `each-monster-is-alive?` *so that the following check can be called at the start of the game:*

```
(define (check-start)
  (and (each-monster-is-in-legal-room? monsters dungeon)
       (each-monster-is-alive? monsters)))
```

Having created the new state we need to make the game take account of the monsters. We will do this by having the game run by turns. Each turn the player gets to perform a command (for example `move` to another room) and then all the monsters in the dungeon get to perform an action. The `command-loop` is modified as follows:

```
(define (command-loop)
  (describe-current-location)
  (when (perform-player-command)
    (when (monster-actions monsters)
      (command-loop))))
```

Just as before, the `perform-player-command` returns a boolean value in order to control whether the game has finished or not. In addition, the `monster-actions` procedure can control whether the game terminates by returning a boolean value. The idea is that when the monsters perform their actions they might attack the player causing the player to be killed. If that happens, the game is over. Otherwise, the player is still alive as the command loop goes round again to the next turn:

```
; monster-actions : List(Monster) -> Boolean
(define (monster-actions monsters)
  (if (null? monsters)
      #t
      (if (monster-action (car monsters))
          (monster-actions (cdr monsters))
          #f)))
```

The only action that the monsters can do at this stage is to attack the player if they are in the same room. However, this should not happen every turn otherwise the game might get a bit predictable. We can use the Racket function `random` to make the game different each time. The function `random` takes a single integer `n` as an argument and returns an integer in the range `[0,n-1]`. However, each time it is called with the same argument it returns a *different* value. We can use this to make the game do different things each run. The predicate `chance?` takes a number `n` and uses `random` to return `#t` one time in `n`:

```
; chance : Number -> Boolean
(define (chance? n)
  (= (random n) 0))
```

Now we can define the action for a monster to be a 50% chance of attacking if it is in the same room:

```
; monster-action Monster -> Boolean
(define (monster-action monster)
  (if (and (monster-in-same-room? monster) (chance? 2))
      (monster-attacks monster)
      #t))
```

A monster is in the same room when the room identifier of the player is the same as that of the monster:

```
; monster-in-same-room? : Monster -> Boolean
(define (monster-in-same-room? monster)
  (= (player-room-id me) (monster-room-id monster)))
```

When the monster attacks we need to work out how much damage it does to the player. We can use `random` again to calculate a different amount of damage each time. Remember that `random` can return 0 so we use that value to mean that the monster tries to attack but misses the player:

```
; monster-attacks : Monster -> Boolean
(define (monster-attacks monster)
  (let ((damage (random 3)))
    (if (> damage 0)
        (monster-hits monster damage)
        (monster-misses monster))))
```

If the moster misses then we just print a message. Remember that we must return a boolean value to control whether the game continues or not. Therefore, we return `#t` when the monster misses:

```
; monster-misses : Monster -> Boolean
```

```
(define (monster-misses monster)
  (printf "The ~a misses!~%" (monster-description monster))
  #t)
```

If the monster hits then it will do some damage to the player. This means that the health-points are reduced by the damage. However we must catch the case where the damage causes the player to die:

```
; monster-hits : Monster Number -> Boolean
(define (monster-hits monster damage)
  (printf "The ~a hits!~%" (monster-description monster))
  (set-player-health! me (- (player-health me) damage))
  (if (<= (player-health me) 0)
      (player-dies)
      #t))

; player-dies : -> Boolean
(define (player-dies)
  (printf "You die!~%")
  #f)
```

It is not just the monsters that can attack, the player can hit the monsters too. That means we need another player command. Remember that we are using Racket `read` to get a single player command so we can use `(hit d)` to attack the monster whose description is `d`. Of course the monster must be in the same room as the player for the attack to be valid. The `perform-player-command` must be extended with the new command:

```
(define (perform-player-command)
  (let ((command (read)))
    (cond ((equal? command 'quit) #f)
          ((and (list? command) (eq? (car command) 'move))
           (move-player (cadr command))
           #t)
          ((and (list? command) (eq? (car command) 'hit))
           (player-hits (cadr command)))
          (#t (printf "Huh?~%")
              #t))))
```

The `player-hits` procedure must check that the target matches a monster in the same room as the player:

```

; find-monster : Symbol -> Monster or #f
(define (find-monster description)
  (findf (λ (m) (and (monster-in-same-room? m)
                     (eq? (monster-description m) description)))
     monsters))

(define (player-hits target)
  (let ((monster (find-monster target)))
    (if (monster? monster)
        (let ((damage (random 3)))
          (if (> damage 0)
              (hit-monster monster damage)
              (printf "You miss the ~a~%" target)))
        (printf "There is no ~a here.~%" target))))
```

The damage inflicted by the player on the monster is calculated using `random`. If the damage is 0 then the player misses the monster. Otherwise, `hit-monster` is used to reduce the health-points of the monster, removing it from the dungeon if the health is reduced to 0:

```
(define (hit-monster monster damage)
  (set-monster-health! monster (- (monster-health monster) damage))
  (if (<= (monster-health monster) 0)
      (monster-dies monster)
      (printf "You hit the ~a~%" (monster-description monster))))

(define (monster-dies monster)
  (printf "The ~a dies!~%" (monster-description monster))
```

```
  (set! monsters (remove monster monsters)))
```

The new dungeon game allows us to do battle with a Yeti:

```
> (play)
Welcome to the dungeon Tony
You are in a green room
There is a Yeti here.
(hit Yeti)
You hit the Yeti
The Yeti hits!
You are in a green room
There is a Yeti here.
(hit Yeti)
You miss the Yeti
The Yeti misses!
You are in a green room
There is a Yeti here.
(hit Yeti)
You hit the Yeti
You are in a green room
There is a Yeti here.
(hit Yeti)
You hit the Yeti
The Yeti misses!
You are in a green room
There is a Yeti here.
(hit Yeti)
You hit the Yeti
The Yeti misses!
You are in a green room
There is a Yeti here.
(hit Yeti)
You hit the Yeti
You are in a green room
There is a Yeti here.
(hit Yeti)
You hit the Yeti
The Yeti hits!
You are in a green room
There is a Yeti here.
(hit Yeti)
The Yeti dies!
You are in a green room
```

Exercise 5.5 *Adding monsters to the dungeon makes it play more like a game. There are lots of extensions that you could add to make it more interesting. Try the following:*

1. *It would be good if the player was told what their current health was each turn. Modify* `describe-current-location` *or otherwise to print out the value of* `(player-health me)` *on each turn.*
2. *If a player leaves a room containing a monster their should be a 50% chance that the monster follows them. Modify* `move-player` *to check for monsters in the room and for each monster to follow if* `(chance? 2)`*.*
3. *Perhaps monsters get scared too. If their health gets low they should move from the current location. Modify* `hit-monster` *to check if the health is lower than 4 and to have a 50% chance of moving the monster via one of the exits.*

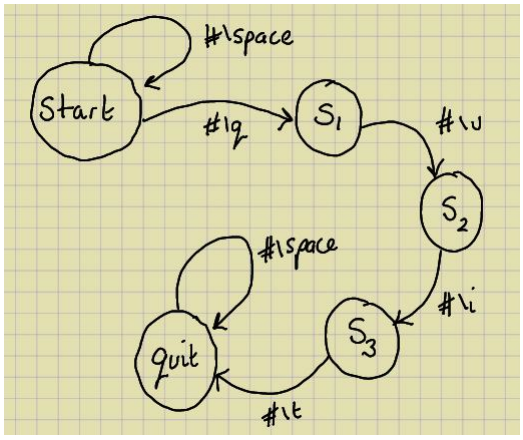


Figure 11: A State Machine Recognizer for `quit`

6 A Command Language (command_language.rkt)

The language used to play the game uses a call of the Racket procedure `read` to read a player command. This makes processing commands easy because `read` gets a single data value from the command-line. The value that is produced is then tested to determine which command the player has entered. The `read` procedure deals with the details of skipping white-space characters and turning the characters that the player types into a Racket data value.

However, there is a significant disadvantage to this approach. It works well for atomic commands such as `quit` because `read` returns a symbol `'quit`. For composite commands what have 2 or more components, such as move or hit, the player must type a list of elements, `(move north)` or `(hit Yeti)` so that `read` can return a single list containing the symbols.

What we would really like to do is to type a command without any unnecessary extra characters. Our command language should be:

- `quit`
- `move DIR`
- `hit MONSTER`

where `DIR` is one of `north`, `south`, `east` or `west`, and `MONSTER` is a monster descriptor such as `Yeti`.

To achieve this we need to take more control of command processing by defining a language recognizer. One way to do this is to implement a state machine. The transitions of the machine are labelled with characters that must be consumed from the input. We start in the state `start` and continually read characters from the input, moving from state to state where the linking transition is labelled with the next character. If we consume all the input characters and end up in an accepting state then we know which command has been entered. If the machine gets stuck or all the input has been read without reaching an accepting state then we know that something has gone wrong.

Figure 11 shows part of a state machine for recognizing the `quit` command. In Racket, a character is written `#\c` where `c` is a letter or the name of a special character such as `space`. The starting state of the machine is called `start`. The player types a command character by character. Any number of initial spaces are consumed with no effect because there is a transition labelled with `#\space` that cycles from `start` to `start`.

If the user types the following characters:

```
quit
```

then the machine starts in the state `start` it will end in the state `quit`. It goes through several intermediate states. If we break it down character-by-character then we get:

```
machine in state start
player types character #\q
machine in state s1
player types character #\u
machine in state s2
player types character #\i
machine in state s3
player types character #\t
machine in state quit
```

Suppose that we have a procedure called `run-machine` that takes a representation of the machine in figure 11, a state and a list of characters and returns the state that is reached by consuming the characters in the list. For example, assuming that the machine in 11 is represented as `command-state-machine`:

```
(run-machine command-state-machine 'start '(#\q #\u #\i #\t))
returns 'quit
```

We are also allowed to include spaces:

```
(run-machine
  command-state-machine
  'start
  '(#\space #\space #\q #\u #\i #\t #\space #\space))
returns 'quit
```

A machine that recognises a command language in this way can be represented as a list of transitions. Each transition has a source and target state and a character that must be processed in order for the machine to change from the source to the target state. The game can be extended with a structure for the transitions as follows:

```
; A Transition is (transition State Char State)
(struct transition (source char target))
```

Now a state machine that processes the `quit` command is defined as follows:

```
; command-state-machine : List(Transition)

(define command-state-machine
  (list
   (transition 'start #\space 'start)
   (transition 'start        #\q     's1)
   (transition 's1           #\u     's2)
   (transition 's2           #\i     's3)
   (transition 's3           #\t     'quit)
   (transition 'quit         #\space 'quit)

   // more transitions...
   ))
```

The names of the states, apart from the start and accepting states, are not important so long as the state names are different.

> **Exercise 6.1** *Given the partial definition of the command state machine for the game commands defined above, it might be nice if the player can type* `bye` *or* `stop` *in addition to* `quit` *in order to quit the game. Add in transitions to the definition given above for these new commands. In all cases the commands that quit the game should reach the accepting state called* `quit`*.*

The complete state machine for the game commands is shown in figure 12. Earlier versions of the game used `read` to read a command as a single Racket value. The state machine processes characters. So we need to find a way to get the characters that the player types and to process them one by one.

```
; command-state-machine : List(Transition)

(define command-state-machine
  (list
   (transition 'start        #\space 'start)
   (transition 'start        #\q     's1)
   (transition 's1           #\u     's2)
   (transition 's2           #\i     's3)
   (transition 's3           #\t     'quit)
   (transition 'quit         #\space 'quit)
   (transition 'start        #\m     's4)
   (transition 's4           #\o     's5)
   (transition 's5           #\v     's6)
   (transition 's6           #\e     's7)
   (transition 's7           #\space 's7)
   (transition 's7           #\n     's8)
   (transition 's8           #\o     's9)
   (transition 's9           #\r     's10)
   (transition 's10          #\t     's11)
   (transition 's11          #\h     'move-north)
   (transition 'move-north   #\space 'move-north)
   (transition 's7           #\s     's12)
   (transition 's12          #\o     's13)
   (transition 's13          #\u     's14)
   (transition 's14          #\t     's15)
   (transition 's15          #\h     'move-south)
   (transition 'move-south   #\space 'move-south)
   (transition 's7           #\e     's16)
   (transition 's16          #\a     's17)
   (transition 's17          #\s     's18)
   (transition 's18          #\t     'move-east)
   (transition 'move-east    #\space 'move-east)
   (transition 's7           #\w     's19)
   (transition 's19          #\e     's20)
   (transition 's20          #\s     's21)
   (transition 's21          #\t     'move-west)
   (transition 'move-west    #\space 'move-west)
   (transition 'start        #\h     's22)
   (transition 's22          #\i     's23)
   (transition 's23          #\t     's24)
   (transition 's24          #\space 's24)
   (transition 's24          #\Y     's25)
   (transition 's25          #\e     's26)
   (transition 's26          #\t     's27)
   (transition 's27          #\i     'hit-Yeti)
   (transition 'hit-Yeti     #\space 'hit-Yeti)))
```

Figure 12: The Command State Machine

Racket provides a procedure `read-line` that reads a single line from the input and returns it as a string. A string can be transformed into a list of characters using the Racket procedure `string->list`. The new version of `perform-player-command` is as follows:

```
(define (perform-player-command)
  (let* ((command (string->list (read-line)))
         (end-state (run-machine command-state-machine 'start command)))
    (cond ((eq? end-state 'quit) #f)
          ((eq? end-state 'move-north)  (move-player 'north))
          ((eq? end-state 'move-south)  (move-player 'south))
          ((eq? end-state 'move-east)   (move-player 'east))
          ((eq? end-state 'move-west)   (move-player 'west))
          ((eq? end-state 'hit-Yeti)    (player-hits 'Yeti))
          ((eq? end-state 'error)       (printf "Huh?~%") #t)
          (#t
            (printf "Incomplete command ~a~%" command) #t))))
```

The new version uses the procedure `run-machine` to process the list of characters starting in state `start` resulting in a state `end-state`. The rest of the procedure checks whether `end-state` is a legal accepting state for the command language and calls the appropriate procedure: `move-player` or `player-hits`.

If `end-state` is not a legal accepting state then there are two possibilities:

1. The state machine has reached an error state in which case the player typed something that was not recognised.
2. The state machine consumed all the input but did not reach an accepting state. In this case the command was incomplete.

We can now complete the game by defining `run-machine`. The first thing is to define a function that finds a transition that has a given starting state and that is labelled with a character:

```
; find-transition : List(Transition) State Char -> Transition or #f

(define (find-transition machine state char)
  (define (is-transition? t)
    (and (eq? (transition-source t) state)
         (eq? (transition-char t) char)))
  (findf is-transition? machine))
```

Now we can use `find-transition` to make successive transitions defined by a machine that consume the supplied characters in a list. When the list of characters is exhausted then we return the resulting state. If we cannot find a transition in the machine then we return the special state `error`:

```
; run-machine : List(Transition) State List(Char) -> State

(define (run-machine machine state chars)
  (if (null? chars)
      state
      (let ((transition (find-transition machine state (car chars))))
        (if transition
            (run-machine machine (transition-target transition) (cdr chars))
            'error))))
```

Exercise 6.2 *What states should the following produce? (Hint: you can type these into DrRacket to find out):*

```
(1) (run-machine command-state-machine 'start '(#\q #\u #\i))
(2) (run-machine command-state-machine 'start '(#\space #\q #\u #\i))
(3) (run-machine command-state-machine 'start '(#\space #\space #\space))
(4) (run-machine command-state-machine 'start (string->list "   hit Ye"))
(5) (run-machine command-state-machine 'start (string->list " move north   "))
```

Now we have defined a command language and have modified `perform-player-command` to use the language, the game should play as before, but we don't have to type in those pesky brackets:

```
> (play)
Welcome to the dungeon Tony
You are in a green room
There is a Yeti here.
move north
You are in a red room
move south
The Yeti misses!
You are in a green room
There is a Yeti here.
hit Yeti
You miss the Yeti
The Yeti misses!
You are in a green room
There is a Yeti here.
hit Yeti
You hit the Yeti
You are in a green room
There is a Yeti here.
   quit
Bye Tony
>
```

Exercise 6.3 *Now that you know how to implement commands, add the following to the command language of the game:*

1. `stop` *and* `bye` *should quit the game.*
2. `teleport` *should jump to anywhere in the dungeon (use* `random` *to choose a room.*
3. `rest` *should add* `1` *to the health-points of the player.*

7 Using Items (items.rkt)

So far the game allows us to move around and explore the dungeon and to fight monsters. The only thing we have to fight monsters with is our bare hands. It would be more interesting if there were some magical items in the dungeon that could be placed in rooms for the player to discover. Each item will have its own set of actions, for example we might add a magical wand so that the player can zap monsters. Such magical items are quite powerful so we should be careful to add a disadvantage to their use, for example a wand might have a chance of not working (and thereby using up a player turn) or even worse: a wand might have a chance of backfiring and causing damage to the player.

In general we want to add the following features ot the game:

1. Items are found in rooms and can be grabbed by a player. Grabbing an item removes it from the room and places it in the player's backpack.
2. Items in a player's backpack are available to use. Each item will have its own collection of actions.
3. Items in a player's backpack can be dropped and are then added to the items in the room. Items can be grabbed and dropped any number of times.

There may be many items of the same type scattered throughout the dungeon. For example, there may be two or more wands. Each item of the same type has the same special actions. Therefore, it makes sense to define the actions once and to refer to them multiple times. This is one of the features of

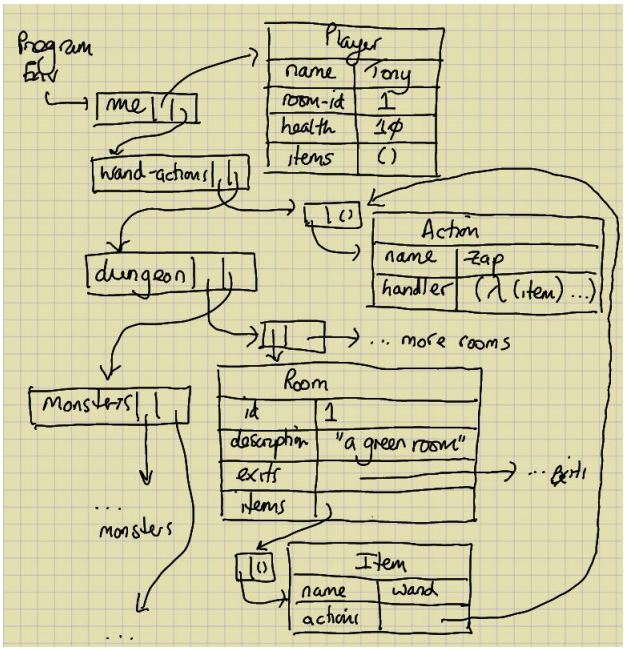


Figure 13: Wand in Room

object-oriented programming that you will learn about in Year 2. We will define a single global variable `wand-actions` that contains a list of actions that each wand can perform. In the simple case a wand can be *zapped*.

Each player must be extended with a list of items that they are currently carrying and each rom must be extended with a list of items that are currently on the floor of the room. Figure 13 shows the state of the game with these extensions. Remember that when a state diagram gets quite large we don't have to draw all of it - just the parts that of interest. Therefore, in figure 13 we only show those parts of the game state that have changed and use `...` to mean *same as last time*. The diagram show that room 1 contains a wand and that the player is not carrying anything.

A player can grab an item that is available on the floor. This will remove the item from the room structure and add it to the list of items carried by the player. Figure 14 shows the situation after the player has grabed the wand in room 1.

Exercise 7.1 *Draw out game state diagrams for the following situations:*

1. *Two wands in room* 1 *and where the player is currently carrying nothing.*
2. *A player carrying two different wands in a room with no items.*
3. *The state change that occurs when a player drop one of the two wands they are carrying.*
4. *A dungeon with three wands in different rooms.*

We now need to extend the Racket structure definitions in order to accommodate items:

```
; An ItemName is a Symbol
; An ActionName is a Symbol

; A Room is (room RoomId String List(Exit) List(Item))
; A Player is (player Symbol RoomId Health List(Item))
```

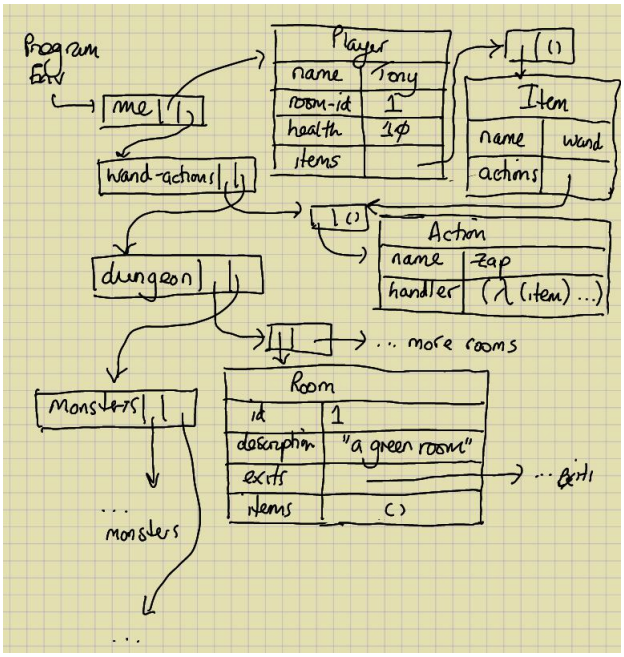


Figure 14: Player Grabs Wand

```
; An Item is (item ItemName List(Action))
; An Action is (action ActionName Fun)

(struct room (id description exits items) #:mutable)
(struct player (name room-id health items) #:mutable)
(struct item (name actions))
(struct action (name handler))
```

A player greabs an item that is available in a room and drops an item that they are carying. When a player is carrying an item they can use it. These are new commands that we need to add to the command language described in the previous section. The language can be extended as follows:

```
; command-state-machine : List(Transition)

(define command-state-machine
  (list

   // commands for move, hit, quit, etc...

   (transition 'start        #\g     's28)
   (transition 's28          #\r     's29)
   (transition 's29          #\a     's30)
   (transition 's30          #\b     'grab)
   (transition 'grab         #\space 'grab)

   (transition 'start        #\d     's31)
   (transition 's31          #\r     's32)
   (transition 's32          #\o     's33)
   (transition 's33          #\p     'drop)
   (transition 'drop         #\space 'drop)

   (transition 'start        #\u     's34)
   (transition 's34          #\s     's35)
   (transition 's35          #\e     'use)
   (transition 'use          #\space 'use)))
```

The procedure perform-player-command must be extended as follows:

```
(define (perform-player-command)
  (let* ((command (string->list (read-line)))
         (end-state (run-machine command-state-machine 'start command)))
```

```
    (cond // cases for move, hit, quit, etc...
          ((eq? end-state 'grab)
           (player-grabs))
          ((eq? end-state 'drop)
           (player-drops))
          ((eq? end-state 'error)
           (printf "Huh?~%")
           #t)
          (#t
           (printf "Incomplete command ~a~%" command)
           #t))))
```

Now we need to define the procedures `player-grabs` and `player-drops`. When a player performs a `grab` command there may be 0, 1 or more items in the room. Therefore we need to handle these all these cases. Let's keep it simple and define the cases where there is 1 item to pick up and drop. The case where there are multiple items will involve asking the player to choose. These cases are implemented in the file `items.rkt`.

Grabbing an item must find the player's room and then get the items in the room. If there are 0 items then there is nothing to pick up otherwise, if there is 1 item in the room we can safely assume that the player wants to pick it up. Remember we won't deal with the multiple items case although it is in the source file. The procedure `player-grabs` returns `#t` becase the game always continues:

```
(define (player-grabs)
  (let* ((room (get-room (player-room-id me)))
         (items (room-items room)))
    (cond ((null? items)
           (printf "nothing to pick up~%"))
          ((= (length items) 1)
           (grab-item (car items) room)))
    #t))
```

Grabbing an item removes it from the room and adds it to the player:

```
(define (grab-item item room)
  (set-room-items! room (remove item (room-items room)))
  (set-player-items! me (cons item (player-items me))))
```

Dropping an item is similar to grabbing an item, however it works in reverse:

```
(define (player-drops)
  (let* ((room (get-room (player-room-id me)))
         (items (player-items me)))
    (cond ((null? items)
           (printf "you are not carrying anything.~%"))
          ((= (length items) 1)
           (drop-item (car items) room)))
    #t))

(define (drop-item item room)
  (set-player-items! me (remove item (player-items me)))
  (set-room-items! room (cons item (room-items room))))
```

To inform the player what is going on we modify the `describe-current-location` procedure to print out the items in the current location and the items that the player is currently carrying:

```
(define (describe-current-location)
  (let ((room (get-room (player-room-id me))))
    (if (room? room)
        (begin
          (printf "You are in ~a~%" (room-description room))
          (for/list ((item (player-items me)))
            (printf "You are carrying a ~a. ~%" (item-name item)))
          (for/list ((item (room-items room)))
            (printf "There is a ~a here. ~%" (item-name item)))
          (for/list ((monster (co-located-monsters)))
            (printf "There is a ~a here.~%" (monster-description monster))))
        (printf "I have no idea where you are!~%"))))
```

Using an item that is being carried will invoke one of the item-specific actions. Again we will assume there is only one action per item although the file `items.rkt` implements code that allows the player to choose from multiple actions where they are available. The first thing to do is to set up some actions for a wand:

```
; wand-actions : List(Action)
(define wand-actions
  (list (action 'zap zap-wand)))
```

and to set up the dungeon to contain at least one wand:

```
; dungeon : List(Room)
(define dungeon
  (list
   (room 1 "a green room" (list (exit 'north 2) (exit 'east 3)) (list (item 'wand wand-actions)))
   (room 2 "a red room" (list (exit 'south 1)) '())
   (room 3 "a room that smells of pizza" (list (exit 'west 1) (exit 'south 4)) '())
   (room 4 "a glowing room" (list (exit 'north 3)) '())))
```

The `player-uses` procedure must check that the player is carrying some items and then choose both the item and the action to perform. As above, we simply the explanation by dealing just with the case where there is at most one item and one action. The file `items.rkt` implements the other cases.

The procedure `player-uses` must return a boolean value to determine whether the game continues or not. Since using a magical item may cause damage to the player, the procedure checks to see of the player's health is greater than 0:

```
(define (player-uses)
  (let ((items (player-items me)))
    (cond ((null? items)
           (printf "you are not carrying anything!~%"))
          ((= (length items) 1)
           (use-item (car items)))
    (> (player-health me) 0)))
```

The procedure `use-item` is interesting because it calls the action handler in the item to use it. This is a powerful style of programming because it allows the item to contain the procedures that define its behaviour. Different items can have different behaviours.

```
(define (use-item item)
  (cond ((= (length (item-actions item)) 1)
         ((action-handler (car (item-actions item))) item))))
```

Now, all that is left to make the game playable is to implement the `zap-wand` procedure. Magical items in the dungeon can do fancy things, however they should come at some cost otherwise the game will be too easy. Therefore, we will allow wands to be zapped so that they cause damage to monsters in the same room as the player, but there will be a chance that the wand will not work (causing the player's turn to be wasted) and, even worse, there will be a chance that the wand will backfire causing damage to the player:

```
(define (zap-wand item)
  (if (chance? 3)
      (zap-hits-self)
      (zap-hits-monsters)))

(define (zap-hits-self)
  (printf "The wand backfires!~%")
  (let ((damage (random 3)))
    (set-player-health! me (- (player-health me) damage))))

(define (zap-hits-monsters)
  (if (null? (co-located-monsters))
      (printf "nothing here to zap at!~%")
      (for/list ((monster (co-located-monsters)))
        (zap-monster monster))))

(define (zap-monster monster)
  (let ((damage (random 5)))
```

```
    (if (= damage 0)
        (printf "the wand splutters.~%")
        (begin
          (printf "the wand produces fireballs that hit the ~a~%" (monster-description monster))
          (set-monster-health! monster (- (monster-health monster) damage))
          (when (<= (monster-health monster) 0)
            (set! monsters (remove monster monsters))
            (printf "the ~a dies~%" (monster-description monster)))))))
```

We can play the game:

```
> (play)
Welcome to the dungeon Tony
You are in a green room
There is a wand here.
There is a Yeti here.
grab
The Yeti misses!
You are in a green room
You are carrying a wand.
There is a Yeti here.
use
The wand backfires!
You are in a green room
You are carrying a wand.
There is a Yeti here.
use
the wand produces fireballs that hit the Yeti
You are in a green room
You are carrying a wand.
There is a Yeti here.
hit Yeti
You hit the Yeti
You are in a green room
You are carrying a wand.
There is a Yeti here.
use
The wand backfires!
You are in a green room
You are carrying a wand.
There is a Yeti here.
use
The wand backfires!
The Yeti hits!
You are in a green room
You are carrying a wand.
There is a Yeti here.
use
the wand produces fireballs that hit the Yeti
The Yeti misses!
You are in a green room
You are carrying a wand.
There is a Yeti here.
use
the wand produces fireballs that hit the Yeti
You are in a green room
You are carrying a wand.
There is a Yeti here.
use
the wand produces fireballs that hit the Yeti
You are in a green room
You are carrying a wand.
There is a Yeti here.
use
the wand splutters.
You are in a green room
You are carrying a wand.
There is a Yeti here.
use
the wand produces fireballs that hit the Yeti
the Yeti dies
You are in a green room
You are carrying a wand.
move north
You are in a red room
You are carrying a wand.
```

Exercise 7.2 *Modify the game as follows:*

1. *Add more wands to the dungeon.*
2. *Introduce a new type of item. Here are some thoughts:*
   (a) `food` *might be useful so that* `eat`*ing the food will restore health-points.*
   (b) *A teleport wand causes all co-located monsters to be removed from the current room and added to random locations in the dungeon.*
   (c) *A wand of* `summon` *causes new monsters to be added the dungeon.*
   (d) *Gold might be added the dungeon and used as a scoring mechanism. The more gold that is collected, the higher the score.*
3. *Items need not have any special actions. You could add a sword to the game that, when carried, doubles the damage caused by hitting monsters.*
4. *Wands are quite powerful, perhaps their use should be limited further by introducing a charge. Wands might initially have a charge of* `(random 5)`*. Each time they are used the charge reduced by* `1`*. When the charge runs out the wand is useless.*
5. *A dungeon with many different types of magical item might be too easy. Perhaps the player should be limited to carrying* `2` *magical items at any given time.*