FUNCTIONAL PROGRAMMING – ASSIGNMENT 2

Hamza Bhatti (21223241)

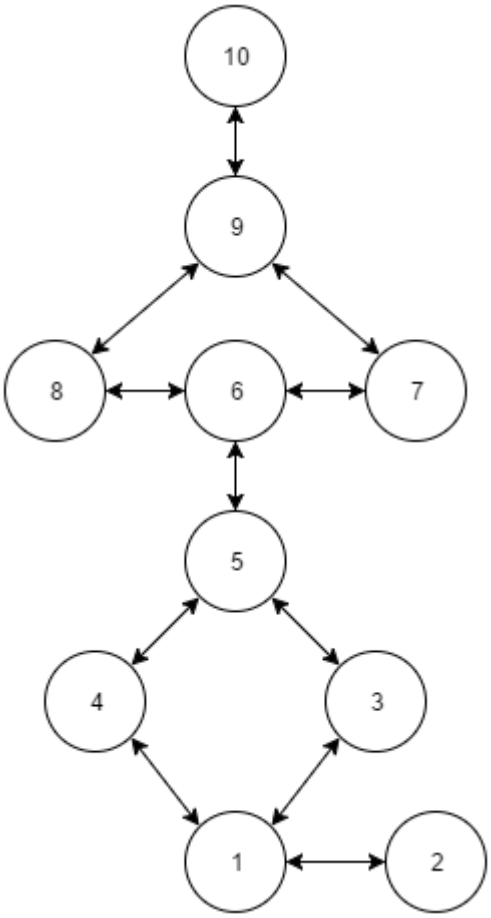
Table of Contents

Introduction

The aim of this assignment was to use knowledge of functional programming from the last assignment, and expand upon it. Code was provided through a few weeks to create a MUD game. This code needed to be collected, implement and also made better. The finished version of the MUD game has changed over the course of the assignment. These changes will be identified later in this report.

User guide

The MUD game that was created, allows the player to navigate around rooms. This is a text based game where the player enters their commands into a command line to carry out actions and move to other rooms. Below is the structure of the dungeon itself.



As can be seen from the diagram, there are up to 10 rooms, where the last room is number 10. The goal is to reach the last room and exit the dungeon. As the player goes through the dungeon, they will be able to pick up items along the way,

Game Commands

There are a few commands that the player will need to know to play the game. The commands will be entered into the command line provided. The player can write whole sentences, but the actions can only take place if the commands are part of the sentence. These commands are listed below along with a description of their usage.

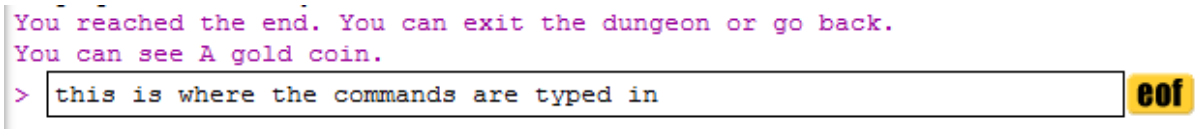
Commands	Descriptions
North, North East, East, South East, South, South West, West and North West.	These are the directions that the player can move to access different room. If the wrong direction is given, the player will not move.
Direction, Look, Examine room.	These commands will enable the player to find out where the possible exits are from the current room. This can be used in any room in the game.
Get, Pickup, Pick.	These are used to pick up objects from the room and place them in your inventory. The object that you want to pick up must be specified.
Put, Drop, Place, Remove.	These commands are used to remove items from your bag, and place them in the room the player is currently in. The command followed by the specified object must be entered to drop the item.
Inventory, Bag.	These commands can be entered to find out what objects are currently in the bag.
Exit game, Quit game, Exit, Quit.	The player can use these commands to exit the game. If the player uses these commands in the last room, a game completion message will be presented.

How To Play

Now, a demo of each of the commands, and the UI, will be provided as screenshots.

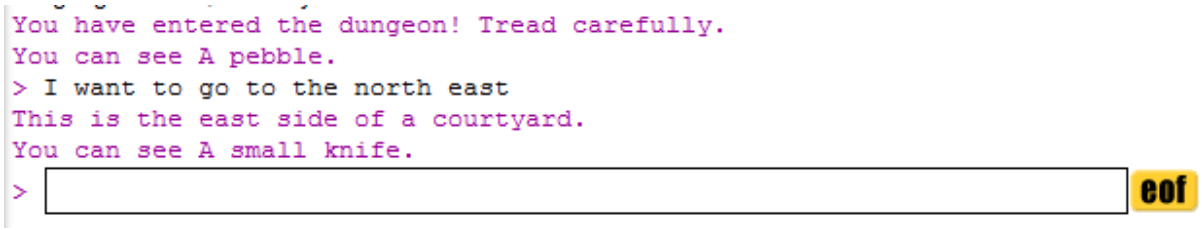
The Basic User Interface

The User will be provided with a description of the room, along with a description of the item that is in the room (if present) and an area to input their command.



Navigating Rooms

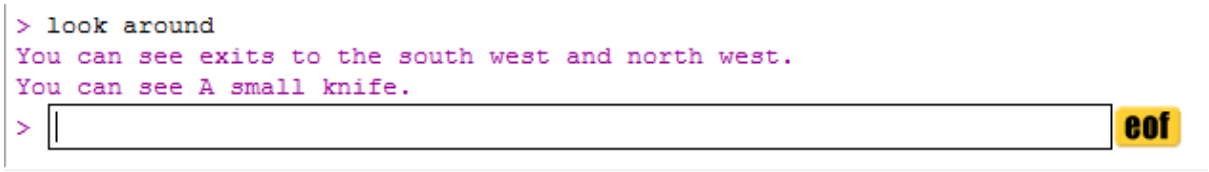
The will able to navigate around room by entering which direction they want to go.



The image above shows that the room changed once the user typed north east.

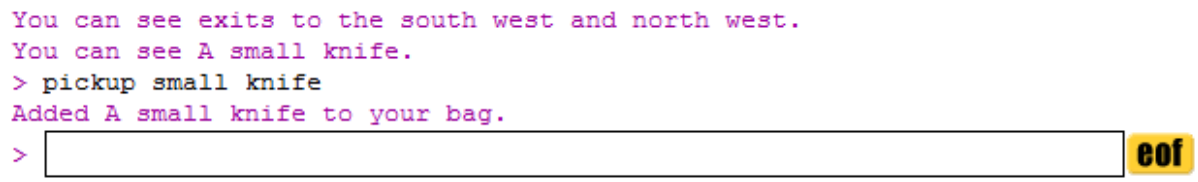
Looking For Exits

When the player enters a look command, they will be presented with the possible directions they can navigate to.



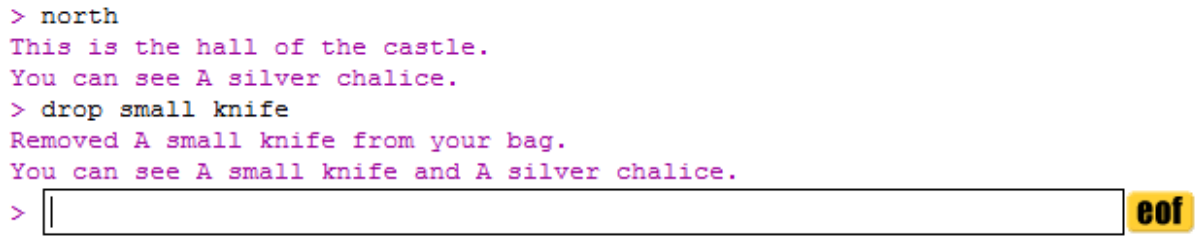
Picking Objects

When the user enters a pick command along with the item they want from the room, it will be added to the inventory. A notification is provided when the pick is successful.



Dropping Object

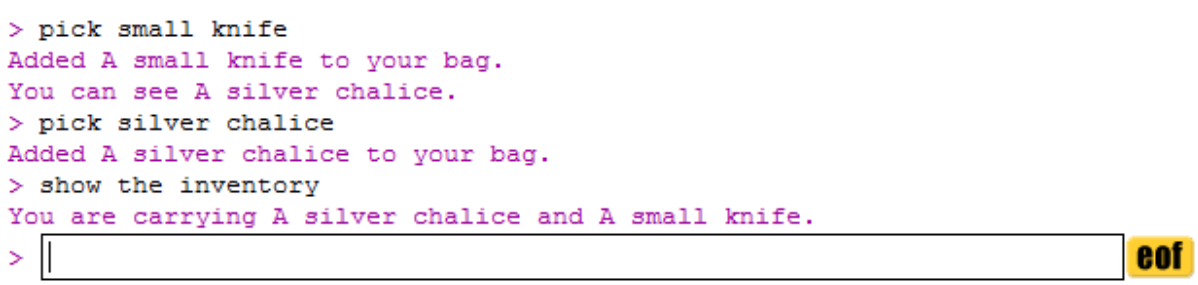
The user is able to drop item into a room. The object will then stay in that room until it is picked again.



As can be seen in the image, before the user dropped the item, there was only one item. After it was dropped, there are then two items.

Displaying Inventory

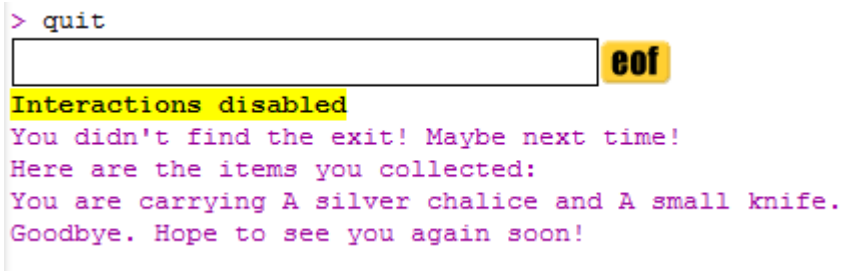
When the inventory command is entered, the contents of the bag are presented to the user.



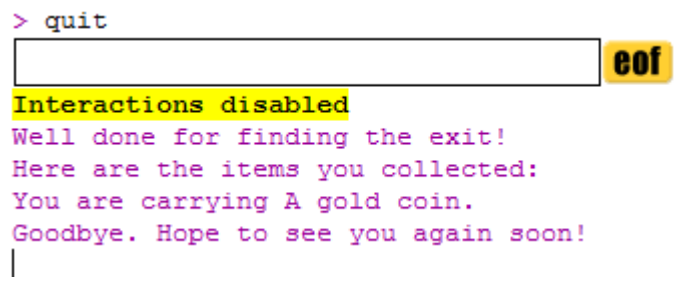
Quitting The Game

When the user quits the game, they are notified if they reached the end or not. In each case they are told what was in their inventory.

When the user has not reached the end.



When the user reaches the end.



How The Game Works

There are many functions present in the code to make the game work. In this section, these functions will be identified and descriptions will be provided to show how they work.

Navigating Rooms

To navigate the rooms, an advanced command line is used.

First, an association table is created for the room id and the description of the room:

```
; Association list for room descriptions
(define descriptions
  '((1 "You have entered the dungeon! Tread carefully.")
    (2 "Entered a small box room.")
    (3 "This is the east side of a courtyard.")
    (4 "It's the west side of a courtyard.")
    (5 "You've stumbled across an entrance to a castle.")
    (6 "This is the hall of the castle.")
    (7 "Looks like an armoury.")
    (8 "It looks like a leasure area.")
    (9 "This is a massive dining room.")
    (10 "You reached the end. You can exit the dungeon or go back.")))
```

Another association table is created to provide key words to each of the rooms:

```
; Decision table data helps drive the game, and what happens in each room
; Each room will allow all actions to take place
(define decisiontable
  `((1 ((east) 2) ((north east) 3) ((north west) 4) ,@actions)
    (2 ((west) ,@actions))
    (3 ((south west) 1) ((north west) 5) ,@actions)
    (4 ((south east) 1) ((north east) 5) ,@actions)
    (5 ((north) 6) ((south west) 4) ((south east) 3) ,@actions)
    (6 ((west) 8) ((south) 5) ((east) 7) ,@actions)
    (7 ((west) 6) ((north west) 9) ,@actions)
    (8 ((est) 6) ((north east) 9) ,@actions)
    (9 ((south west) 8) ((south east) 7) ((north) 10) ,@actions)
    (10 ((south) 9) ,@actions)))
```

When a new room is moved into, the description of the room must be returned. To do this we use get-response which takes the id of the room:

```
; Get the repsonse/ description of the room
(define (get-response id)
  ; Use the ass-ref where the list us descriptions, id is the room
  ; and function is assq
  ; car used for presentation
  (car (ass-ref descriptions id assq)))
```

We then call a new function called ass-ref. We state what list we want to use, in this case it will be the descriptions list, and we provide the room id along with a function we want to use. Assq is used to return a list inside a nested list with the same id that we provided. The ass-ref function can be seen below:

```
; Returns the list depending on the id, table and functions
(define (ass-ref assqlist id func)
  (cdr (func id assqlist)))
```

Next, we define a function, that when used, provides us with a list of keywords based on a given id. We use ass-ref function again. The list we want information from is decisiontable and we also provide the id of the room:

```
; Get the keywords for the response depending on the id
(define (get-keywords id)
  (let ((keys (ass-ref decisiontable id assq)))
    (map (lambda (key) (car key)) keys)))
```

For us to evaluate what is in a sentence to carry out a movement, we use keyword matches to determine where to move to, or what action to use. First we use the function list-of-lengths. We provide it a list and in return, we get a list of weights in place of each word. If no weighting is provided, there is no keywords found:

```
; Outputs a list in the form: (0 0 0 2 0 0)
(define (list-of-lengths keylist tokens)
  (map
   (lambda (x)
     (let ((set (lset-intersection eq? tokens x)))
       ;; Apply some weighting to the result
       (* (/ (length set) (length x)) (length set))))
   keylist))
```

Next we need to get the position of the highest weighted value:

```
; Find the index of the largest number of the keylist
(define (index-of-largest-number list-of-numbers)
  (let ((n (car (sort list-of-numbers >))))
    (if (zero? n)
      #f
      (list-index (lambda (x) (eq? x n)) list-of-numbers))))
```

To use the functions provided above, we make a wrapper that will take the id of the current room and the list of token from the command line. The function will return the id of the new room, or the weighting for an action to take place:

```
; Wrapper for multi word input
; Return the id of the room
(define (lookup id tokens)
  (let* ((record (ass-ref decisiontable id assv))
         (keylist (get-keywords id))
         (index (index-of-largest-number (list-of-lengths keylist tokens))))
    (if index
      (cadr (list-ref record index))
      #f)))
```

Objects

Hash tables are created to store objects in rooms and in inventory.

```
; Hash table to track whats in a room
(define objectdb (make-hash))

; Hash table to track what we are carrying
(define inventorydb (make-hash))
```

The objects are added to the room by using two functions. The add-objects take the objectdb as an argument. This function loops for each item to add and calls another function called add-object. This takes the database the item is going to be added to, the id of the room and the object. A check occurs to see if the id is present in the db provided, it will continue. The hash ref is saved to record with let. Hash set is used to save the object to the db.

```
; Adding the object to the databse
(define (add-object db id object)
  (if (hash-has-key? db id)
      (let ((record (hash-ref db id)))
        (hash-set! db id (cons object record)))
      (hash-set! db id (cons object empty))))

; Function will load what is in our databse into an objects database
(define (add-objects db)
  (for-each
   (lambda (r)
     (add-object db (first r) (second r))) objects))

(add-objects objectdb)
```

Actions

The command line allows the player to carry out actions. The actions are given keywords to allow them to work. The lists that are used for action are look, quit, pick, put and inventory. These are provided below:

```
; Actions assoc list, where keywords will result in an action
(define look '(((directions) look) ((look) look) ((examine room) look)))
(define quit '(((exit game) quit) ((quit game) quit) ((exit) quit) ((quit) quit)))
(define pick '(((get) pick) ((pickup) pick) ((pick) pick)))
(define put '(((put) drop) ((drop) drop) ((place) drop) ((remove) drop)))
(define inventory '(((inventory) inventory) ((bag) inventory)))
```

These lists are the added to another list called actions. Quasiquaoting is used to so that the lists can be added without being totally nested. The ,@ is used splice the list.

```
; Quasiquoting the list, to give special properties
; List filled with unquote (,) Using unquote splicing ,@ so the extra list is removed
(define actions `(,@look ,@quit ,@pick ,@put ,@inventory))
```

The actions list is present in the decision table so that keywords associated with the actions can be carried out in each room.

Look

The get-directions function is used to show what exits are available for the current room. It accepts the current id of the room. It then uses assq to return the list of directions that the player can go from the room from the decisiontable. The list provided is then converted so that the number of rooms is returned. A conditional statement is then used. Depending on the amount of rooms that are available, a message will be provided. If there is one room, the direction will be added to message. If there are multiple rooms, an “and” will be inserted between the directions:

```
; Obtaining room direction

(define (slist->string l)
  (string-join (map symbol->string l)))

; Get direction with a given id
(define (get-directions id)
  ; Get the list depending on the id
  (let ((record (assq id decisiontable)))
    ; Finding the length of the list of directions
    (let* ((result (filter (lambda (n) (number? (second n))) (cdr record)))
           (n (length result)))
      ; If there are no room
      (cond ((= 0 n)
             (printf "You appear to have entered a room with no exits.\n"))
            ; 1 room
            ((= 1 n)
             (printf "You can see an exit to the ~a.\n" (slist->string (caar result))))
            (else
             ; Or more rooms
             (let* ((losym (map (lambda (x) (car x)) result))
                    (lostr (map (lambda (x) (slist->string x)) losym)))
               (printf "You can see exits to the ~a.\n" (string-join lostr " and "))))))))
```

Quit

When the user wants to quit the game, the quit-game function is used. This function take the id of the current room as an argument. A conditional is then initiated. If the id is equal to 10, being the last room, a finished game message is provided. If the room id is anything other than 10, a generic message is provided. After the condition, the inventory is provided and the game is closed.

```
; Function when quiting the game, needs the id of the room
(define (quit-game id)
  (cond
    ; If the room id was 10 (the final room)
    ((eq? id 10)
     ; Display this message
     (printf "Well done for finding the exit!\n"))
    ; Or else
    (else
     ; Give this message
     (printf "You didn't find the exit! Maybe next time!\n")))
  ; Shows what items were in the bag
  (printf "Here are the items you collected: \n")
  (display-inventory)
  ; Goodbye message
  (format #t "Goodbye. Hope to see you again soon!\n")
  ; Exit the game
  (exit))
```

Picking Up And Dropping Objects

The remove-object function is used to remove an object from the either the bag or the room. The function takes the name of the db the object is being removed from, a reference from where the objet currently is (can be bag or room id), where the item is being added, and the string object. First the database checks if the place the item is being removed from is in the database. When the hash key is found, a conditional statement is triggered. If the item wasn't found, and error message will be triggered. If the from parameter was a number (meaning the room) the object will be added to the bag with add-object function. When from is 'bag, the object will be removed from the bag and added to room.

```
; Removing objects
(define (remove-object db from add-to str)
  ; If the object isnt in the db, from the place we are removing from
  ; can be the 'bag or id (of room)
  (when (hash-has-key? db from)
    (let* ((record (hash-ref db from))
           (result (remove (lambda (x) (string-suffix-ci? str x)) record))
           (item (lset-difference equal? record result)))
      ; Trigger conditional
      (cond
        ; If the item provided doesnt exist
        ((null? item)
         ; Provide a generic error response
         (printf "Oops. Either the item isn't in the room, or you aren't carrying it!\n"))
        ; If the place we are removing from is a room/ number
        ((number? from)
         ; Give message that its added to bag
         (printf "Added ~a to your bag.\n" (first item))
         ; Use the add object functions to add to the inventory
         ; where add-to is the 'bag
         (add-object inventorydb add-to (first item))
         (hash-set! db from result))
        ; If the place we are removing from is 'bag
        ((eq? from 'bag)
         ; Give messagethe item is being moved to the bag
         (printf "Removed ~a from your bag.\n" (first item))
         ; Use the add-object function to add to the room
         ; where add-to is the room id
         (add-object objectdb add-to (first item))
         (hash-set! db from result))))))
```

To use the remove function, a wrapper is used to call in the main function. The function takes the id of the current room, the input from the user and the action (either pick or drop). A condition is triggered. If the action is pick, the db to remove from is objectdb, from is the id of the room and add-to is bag. In the second condition, if the action is drop, the db to remove from is inventorydb, from is bag, add-to is the room. If either the room or bag are empty, a message is provided stating that the item is not in the room, or isn't in the bag.

```
; Pick and put wrapper function
(define (pick-and-put id input action)
  (let ((item (string-join (cdr (string-split input)))))
    ; Trigger conditional
    (cond
      ; If the action was pick, remove from the room, and add to the bag
      ((eq? action pick) (remove-object objectdb id 'bag item))
      ; If the action was to drop, remove from the bag and add to the room
      ((eq? action drop) (remove-object inventorydb 'bag id item)))))
```

Inventory

The user can view the inventory. The display-inventory function calls the display-objects function and passes the inventorydb and 'bag as an id.

```
; Display the items in the inventory
(define (display-inventory)
  ; Use display-objects function where the db is inventory
  (display-objects inventorydb 'bag))
```

Game Loop

The game loop is a wrapper function. It takes the id of the room, the input and the response keywords provided by the main function. The function uses conditional statements for actions that can be used. If the response matches a keyword, a new function will be called. Then the main function is returned to.

```
; Game loop for actions
(define (game-loop id input response)
  ; Conditions
  (cond
    ; When response is for look
    ((eq? response 'look)
     ; Get directions
     (get-directions id))
    ; When response is to pick
    ((eq? response 'pick)
     ; Pick up the item from the room
     (pick-and-put id input pick))
    ; When response is to drop
    ((eq? response 'drop)
     ; Drop the item from the bag
     (pick-and-put id input drop))
    ; When response is to check invetory
    ((eq? response 'inventory)
     ; Show the inventory
     (display-inventory))
    ; When respone to quit
    ((eq? response 'quit)
     ; Use the quit game function
     (quit-game id))))
```

Start Game

This function is the main game. It takes an initial room id, being 1. A named let is used as a loop. The description for the room is provided with get-response. The objects from the objectdb are shown for the current room. The user can then enter their commands. The return from the lookup function is saved in response named let. A conditional statement is then triggered. If the response is a number, it means a new room was being moved to and will loop back to the named let loop and show the new description. If the response had no keywords, an error message will be presented. The response can be evaluated against a list of items. These items are the actions that the player can use. If the response matches any of the action keywords, it will call the game-loop function and take the id of the room, the user input and the response.

```
; Start game with given id
(define (startgame initial-id)
  ; Named let loop, where id is the initial id and description is true
  (let loop ((id initial-id) (description #t))
    ; When description is true
    (if description
        ; Get the response/ description of the room
        (printf "~a\n" (get-response id))
        (printf ""))
    ; Display the objects of the room
    (display-objects objectdb id)
    (printf "> ")
    ; User gives input, either one word or several
    (let* ((input (string-downcase (read-line)))
           (string-tokens (string-tokenize input))
           (tokens (map string->symbol string-tokens)))
      ; Set the return of lookup, can be room id or actions key words
      (let ((response (lookup id tokens)))
        ; Conditions
        (cond
          ; If the returned response from lookup is a number/ room id
          ((number? response)
           ; Loop with new room id
           (loop response #t))
          ; When the response was not valid
          ((eq? #f response)
           ; Give a message
           (format #t "Oops, didn't understand that. Try again!\n")
           ; Loop back with the current room id
           (loop id #f))
          ; If the response matches of any of the keywords in the list
          ((memq response '(quit inventory drop pick look))
           ; Call game-loop so that actions can take place
           (game-loop id input response)
           ; Loop back with current room id
           (loop id #f)))))))

; Start the game with initial room id
(startgame 10)
```

Testing The Game

This section goes over some cases where error handling takes place. Screenshots will also be provided.

Giving A Wrong Input

When the user enters an input that has no weighting, an error response is given.

```
You have entered the dungeon! Tread carefully.
You can see A pebble.
> i want to go that way
Oops, didn't understand that. Try again!
You can see A pebble.
>
```

eof

Picking Up Objects

The user won't be able to pick up items that aren't present in the room.

```
You can see A pebble.
> pick the random thing
Oops. Either the item isn't in the room, or you aren't carrying it!
You can see A pebble.
>
```

eof

Checking Inventory

The program will identify if there is nothing in bag.

```
You can see A small knife.
> inventory
There are no items in your bag!
You can see A small knife.
>
```

eof

Dropping Item

If the user want to drop an item that doesn't exist, an error message is provided.

```
> drop the random
Oops. Either the item isn't in the room, or you aren't carrying it!
>
```

eof

Git Log

For this assignment it was required that version control could be used to have a record of the changes that had been implemented to the MUD game during development and refactoring.

commit 8d4b9aa0d2eb0311a9e76cf493696e25e1c04154
Author: unknown <hamzabhatti93@gmail.com>
Date: Thu May 5 11:03:05 2016 +0100

Change the structure of the dungeon to 10 rooms. Add a quit game function where specific messages are given and bag is displayed.

commit 561fbe451d0c7e32ae3a2535ae67c48d1cc26d6c
Author: unknown <hamzabhatti93@gmail.com>
Date: Wed May 4 13:50:10 2016 +0100

Add more line by line comments to all files.

commit 46b383b0d834ae31b33032be6694bfd1545e12b3
Author: unknown <hamzabhatti93@gmail.com>
Date: Tue May 3 16:54:15 2016 +0100

Split into seperate files. Main file is mud.rkt.

commit 22837e474a9eb437bfab8ff475825656d8be07a9
Author: unknown <hamzabhatti93@gmail.com>
Date: Tue May 3 16:09:06 2016 +0100

Refactor game loop. Remove repetition and point to another function called game-loop.

commit ffdfdfbd377bdf0091f2628187a47baf0f082462
Author: unknown <hamzabhatti93@gmail.com>
Date: Sat Apr 23 16:43:07 2016 +0100

Assignment 2 V7: Made wrapper for the pick and put functions. Also made ammendments for the display-object function. Will display different message if the bag or room is empty.

commit 4bdc244f7fee63ff3c88af2d3d140bea469355b5
Author: unknown <hamzabhatti93@gmail.com>
Date: Fri Apr 22 18:53:38 2016 +0100

Assignment 2 V6: Refactored the remove-object-from-room and bag functions by merging them into one. The wrapper functions are pick-item2 and put-item2. Game still works.

commit b482f42d532e14e0bf5c70a58e9ef81938c88958
Author: unknown <hamzabhatti93@gmail.com>
Date: Mon Apr 11 13:17:51 2016 +0100

Assignment 2: Added more items to pick up.

commit 59ee1480a1b3f63f28c7eafb2841e044f82bf548
Author: unknown <hamzabhatti93@gmail.com>
Date: Sun Apr 10 16:58:41 2016 +0100

Assignemnt 2 V5: Actions all work, didnt work before because ran off the 13th room instead of 1st.

commit 861b16e8e5f682a1205dedb60ca1670bd56ff625
Author: unknown <hamzabhatti93@gmail.com>
Date: Sun Apr 10 16:54:22 2016 +0100

Assignement2: Added more actions, but game loop isnt working.

commit 96906d8e4dc3fa50ddf56c939d19d1e044668fc8
Author: unknown <hamzabhatti93@gmail.com>
Date: Sun Apr 10 12:06:01 2016 +0100

Assingment2 V4: Going back to old version, need to redo the pick actions etc.

commit 1ba85328c34826b87bcda957d672df2853b79053
Author: unknown <hamzabhatti93@gmail.com>
Date: Tue Apr 5 16:06:11 2016 +0100

Assignment 2: Refactored assq and assv into one function.

commit 1e185e004d5e36c20a6ebb1584ab7214f2594e97
Author: unknown <hamzabhatti93@gmail.com>
Date: Tue Apr 5 16:00:53 2016 +0100

Assignment 2: Refactored the assq and assv function into one

commit bcc96029a4cf7180f828af28deebb454a0c0ed3f
Author: unknown <hamzabhatti93@gmail.com>
Date: Wed Mar 30 18:40:26 2016 +0100

Assignment 2: Updated mud with comments. Need to add more actions.

commit 5fd9df4f19a1ce861741b8e3dc1806c3792fe2fd
Author: unknown <hamzabhatti93@gmail.com>
Date: Tue Mar 29 16:29:27 2016 +0100

Assignment 2: Add rooms, need descriptions and comments.

commit a774a2865652a443ab9a63d65cc533bb294305de
Author: unknown <hamzabhatti93@gmail.com>

Date: Sun Mar 27 18:25:46 2016 +0100

Assignment 2: Using week 7 code, needs comments and an actual dungeon.

commit 5bb099e5b5a283dc567bd0bdab020ab0160ca785
Author: unknown <hamzabhatti93@gmail.com>
Date: Sun Mar 27 17:23:06 2016 +0100

Assignment 2: Problem with advanced mud, will likely need to use the advanced mud from week 7 with actions to make progress. Will commenent on those.

commit 2e8411f5d4a06ae652196e953c3bea28e2de6614
Author: unknown <hamzabhatti93@gmail.com>
Date: Sat Mar 26 16:00:29 2016 +0000

Assingment 2: Full commented code. Needs more directions and an exit.

commit 5c08dbe0735c0096dd41e28ec3ad88afe7df78ae
Author: unknown <hamzabhatti93@gmail.com>
Date: Sat Mar 26 15:18:09 2016 +0000

Assingment2: upload v1 of mud, needs comments for understanding.

Appendix

Full Code Listing

mud.rkt

```
#lang racket

(require srfi/1)
(require srfi/13)
(require srfi/48)
(require racket/include)
; Files to include
(include "assoc_and_decision.rkt")
(include "object_functions.rkt")
(include "room_navigation.rkt")

; Function when quiting the game, needs the id of the room
(define (quit-game id)
  (cond
    ; If the room id was 10 (the final room)
    ((eq? id 10)
     ; Display this message
     (printf "Well done for finding the exit!\n"))
    ; Or else
    (else
     ; Give this message
     (printf "You didn't find the exit! Maybe next time!\n")))
  ; Shows what items were in the bag
  (printf "Here are the items you collected: \n")
  (display-inventory)
  ; Goodbye message
  (format #t "Goodbye. Hope to see you again soon!\n")
  ; Exit the game
  (exit))

; Game loop for actions
(define (game-loop id input response)
  ; Conditions
  (cond
    ; When response is for look
    ((eq? response 'look)
     ; Get directions
     (get-directions id))
    ; When response is to pick
    ((eq? response 'pick)
     ; Pick up the item from the room
     (pick-and-put id input pick))
    ; When response is to drop
    ((eq? response 'drop)
     ; Drop the item from the bag
     (pick-and-put id input drop))
    ; When response is to check invetory
    ((eq? response 'inventory)
     ; Show the inventory
     (display-inventory))
    ; When respone to quit
    ((eq? response 'quit)
     ; Use the quit game function
     (quit-game id))))

; Start game with given id
(define (startgame initial-id)
  ; Named let loop, where id is the initial id and description is true
  (let loop ((id initial-id) (description #t))
    ; When description is true
    (if description
        ; Get the response/ description of the room
        (printf "~a\n" (get-response id))
        (printf ""))
    ; Display the objects of the room
```

```
    (display-objects objectdb id)
    (printf "> ")
    ; User gives input, either one word or several
    (let* ((input (string-downcase (read-line)))
           (string-tokens (string-tokenize input))
           (tokens (map string->symbol string-tokens)))
      ; Set the return of lookup, can be room id or actions key words
      (let ((response (lookup id tokens)))
        ; Conditions
        (cond
          ; If the returned response from lookup is a number/ room id
          ((number? response)
           ; Loop with new room id
           (loop response #t))
          ; When the response was not valid
          ((eq? #f response)
           ; Give a message
           (format #t "Oops, didn't understand that. Try again!\n")
           ; Loop back with the current room id
           (loop id #f))
          ; If the response matches of any of the keywords in the list
          ((memq response '(quit inventory drop pick look))
           ; Call game-loop so that actions can take place
           (game-loop id input response)
           ; Loop back with current room id
           (loop id #f)))))))

; Start the game with initial room id
(startgame 10)
```

assoc_and_decision.rkt

```
; File contains all the association lists, along with decision table for the
; MUD game

; Association list for room descriptions
(define descriptions
  '((1 "You have entered the dungeon! Tread carefully.")
    (2 "Entered a small box room.")
    (3 "This is the east side of a courtyard.")
    (4 "It's the west side of a courtyard.")
    (5 "You've stumbled across an entrance to a castle.")
    (6 "This is the hall of the castle.")
    (7 "Looks like an armoury.")
    (8 "It looks like a leasure area.")
    (9 "This is a massive dining room.")
    (10 "You reached the end. You can exit the dungeon or go back.")))

; Association list for Objects that belong to certain rooms
(define objects
  '((1 "A pebble")
    (3 "A small knife")
    (6 "A silver chalice")
    (7 "A long sword")
    (10 "A gold coin")))


; Actions assoc list, where keywords will result in an action
(define look '(((directions) look) ((look) look) ((examine room) look)))
(define quit '(((exit game) quit) ((quit game) quit) ((exit) quit) ((quit) quit)))
(define pick '(((get) pick) ((pickup) pick) ((pick) pick)))
(define put '(((put) drop) ((drop) drop) ((place) drop) ((remove) drop)))
(define inventory '(((inventory) inventory) ((bag) inventory)))

; Quasiquoting the list, to give special properties
; List filled with unquote (,) Using unquote splicing ,@ so the extra list is removed
(define actions `(,@look ,@quit ,@pick ,@put ,@inventory))


; Decision table data helps drive the game, and what happens in each room
; Each room will allow all actions to take place
(define decisiontable
  `((1 ((east) 2) ((north east) 3) ((north west) 4) ,@actions)
    (2 ((west) ,@actions))
    (3 ((south west) 1) ((north west) 5) ,@actions)
    (4 ((south east) 1) ((north east) 5) ,@actions)
    (5 ((north) 6) ((south west) 4) ((south east) 3) ,@actions)
    (6 ((west) 8) ((south) 5) ((east) 7) ,@actions)
    (7 ((west) 6) ((north west) 9) ,@actions)
    (8 ((est) 6) ((north east) 9) ,@actions)
    (9 ((south west) 8) ((south east) 7) ((north) 10) ,@actions)
    (10 ((south) 9) ,@actions)))
```

object_functions.rkt

```
; File contains all the functions for objects in rooms and bag

; Loading object databse

; Hash table to track whats in a room
(define objectdb (make-hash))

; Hash table to track what we are carrying
(define inventorydb (make-hash))

; Adding the object to the databse
(define (add-object db id object)
  (if (hash-has-key? db id)
      (let ((record (hash-ref db id)))
        (hash-set! db id (cons object record)))
      (hash-set! db id (cons object empty))))

; Function will load what is in our databse into an objects database
(define (add-objects db)
  (for-each
   (lambda (r)
     (add-object db (first r) (second r))) objects))

(add-objects objectdb)

; Displaying our objects
; We use this to display either what objects are in the room or bag
(define (display-objects db id)
  ; Check if the db has the desired id
  (when (hash-has-key? db id)
    ; Save the object in record
    (let* ((record (hash-ref db id))
           ; when there are many objects, put an "and" between them
           (output (string-join record " and ")))
      ; If the output is not nothing
      (when (not (equal? output ""))
        ; If the id is the bag
        (if (eq? id 'bag)
            ; Give a message including the output
            (printf "You are carrying ~a.\n" output)
            ; Then tell them if they see something
            (printf "You can see ~a.\n" output)))))
  ; If the id isnt found
  (when (not (hash-has-key? db id))
    ; Trigger condition
    (cond
      ; If the id was bag, tell them there's nothing there
      ((eq? id 'bag) (printf "There are no items in your bag!\n"))
      ; Else, it will be the room. Tell them there is nothing there
      (else (printf "There are no items in the room!\n")))))

; Removing objects
(define (remove-object db from add-to str)
  ; If the object isnt in the db, from the place we are removing from
  ; can be the 'bag or id (of room)
  (when (hash-has-key? db from)
    (let* ((record (hash-ref db from))
           (result (remove (lambda (x) (string-suffix-ci? str x)) record))
           (item (lset-difference equal? record result)))
      ; Trigger conditional
      (cond
        ; If the item provided doesnt exist
        ((null? item)
         ; Provide a generic error response
         (printf "Oops. Either the item isn't in the room, or you aren't carrying it!\n"))
        ; If the place we are removing from is a room/ number
        ((number? from)
         ; Give message that its added to bag
         (printf "Added ~a to your bag.\n" (first item))
```

```
          ; Use the add object functions to add to the inventory
          ; where add-to is the 'bag
          (add-object inventorydb add-to (first item))
          (hash-set! db from result))
         ; If the place we are removing from is 'bag
         ((eq? from 'bag)
          ; Give messagethe item is being moved to the bag
          (printf "Removed ~a from your bag.\n" (first item))
          ; Use the add-object function to add to the room
          ; where add-to is the room id
          (add-object objectdb add-to (first item))
          (hash-set! db from result))))))

; Pick and put wrapper function
(define (pick-and-put id input action)
  (let ((item (string-join (cdr (string-split input)))))
    ; Trigger conditional
    (cond
      ; If the action was pick, remove from the room, and add to the bag
      ((eq? action pick) (remove-object objectdb id 'bag item))
      ; If the action was to drop, remove from the bag and add to the room
      ((eq? action drop) (remove-object inventorydb 'bag id item)))))

; Display the items in the inventory
(define (display-inventory)
  ; Use display-objects function where the db is inventory
  (display-objects inventorydb 'bag))
```

room_navigation.rkt

```
; File contains room navigation functions

; Obtaining room direction

(define (slist->string l)
  (string-join (map symbol->string l)))

; Get direction with a given id
(define (get-directions id)
  ; Get the list depending on the id
  (let ((record (assq id decisiontable)))
    ; Finding the length of the list of directions
    (let* ((result (filter (lambda (n) (number? (second n))) (cdr record)))
           (n (length result)))
      ; If there are no room
      (cond ((= 0 n)
             (printf "You appear to have entered a room with no exits.\n"))
            ; 1 room
            ((= 1 n)
             (printf "You can see an exit to the ~a.\n" (slist->string (caar result))))
            (else
             ; Or more rooms
             (let* ((losym (map (lambda (x) (car x)) result))
                    (lostr (map (lambda (x) (slist->string x)) losym)))
               (printf "You can see exits to the ~a.\n" (string-join lostr " and "))))))))

; Returns the list depending on the id, table and functions
(define (ass-ref assqlist id func)
  (cdr (func id assqlist)))

; Get the repsonse/ description of the room
(define (get-response id)
  ; Use the ass-ref where the list us descriptions, id is the room
  ; and function is assq
  ; car used for presentation
  (car (ass-ref descriptions id assq)))

; Get the keywords for the response depending on the id
(define (get-keywords id)
  (let ((keys (ass-ref decisiontable id assq)))
    (map (lambda (key) (car key)) keys)))


; Outputs a list in the form: (0 0 0 2 0 0)
(define (list-of-lengths keylist tokens)
  (map
   (lambda (x)
     (let ((set (lset-intersection eq? tokens x)))
       ;; Apply some weighting to the result
       (* (/ (length set) (length x)) (length set))))
   keylist))

; Find the index of the largest number of the keylist
(define (index-of-largest-number list-of-numbers)
  (let ((n (car (sort list-of-numbers >))))
    (if (zero? n)
      #f
      (list-index (lambda (x) (eq? x n)) list-of-numbers))))

; Wrapper for multi word input
; Return the id of the room
(define (lookup id tokens)
  (let* ((record (ass-ref decisiontable id assv))
         (keylist (get-keywords id))
         (index (index-of-largest-number (list-of-lengths keylist tokens))))
    (if index
      (cadr (list-ref record index))
      #f)))
```