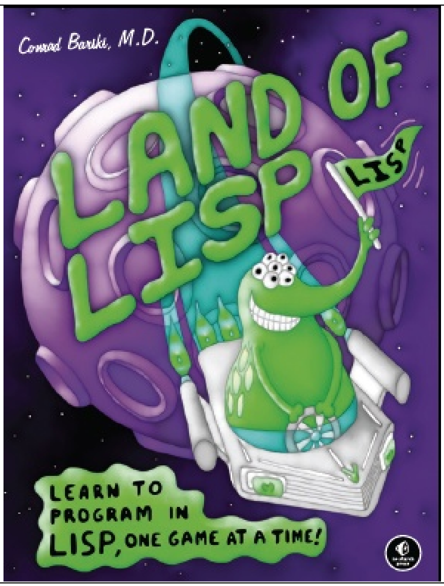
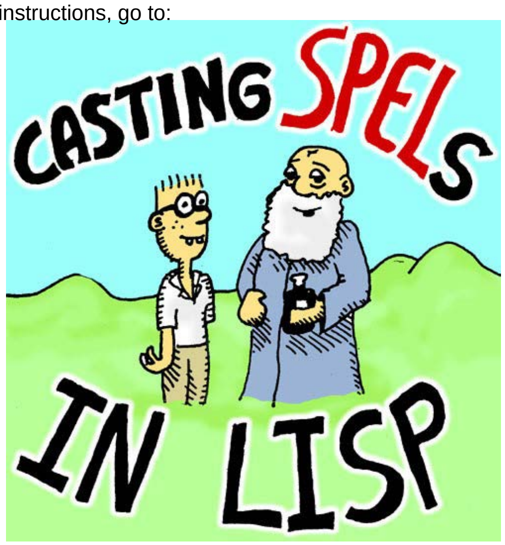


Chapter 5

Learning Lisp Console Programming.. By Building a Text Game Engine!

HOUSE
ATTIC
OUR GAME WORLD
LADDER
WELL
GARDEN
LIVING ROOM
CHAIN
FROG

For detailed instructions, go to:



http://lisperati.com/casting.html

The Game

You are a wizard’s apprentice.

You’ll explore the wizard’s house and world.

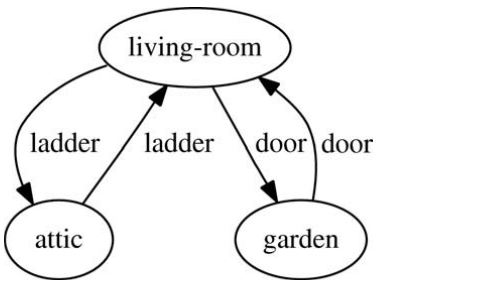
Soon, you’ll be able to solve puzzles in the Wizard’s World and win a magical (no carb) donut.

For your class project, you will **create your own adventure game**!!

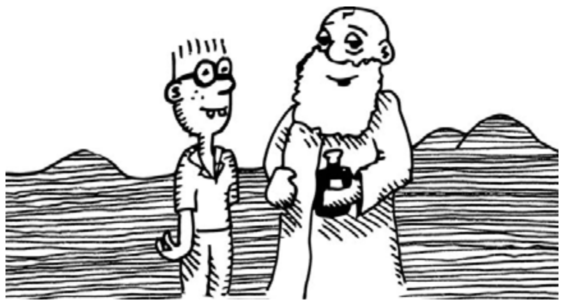
HOUSE
ATTIC
OUR GAME WORLD
LADDER
WELL
GARDEN
LIVING ROOM
CHAIN
FROG

We can visit three different **locations**:
a living room, an attic, and a garden.
Players can move between places using the door and the ladder.

$$\begin{tikzcd}
& \text{living-room} \arrow[dl, bend right, "ladder"'] \arrow[dr, "door"] & \\
\text{attic} \arrow[ur, "ladder"'] & & \text{garden} \arrow[ul, bend right, "door"']
\end{tikzcd}$$



Think of this game world as a simple **directed graph** with **three nodes** (represented as ellipses) and **four edges** (represented as arrows):



Players move between nodes by *traveling along the edges* in either direction. Wherever the players are, they can *interact with various objects around them.*

Basic Requirements

Our game code will need to handle a few basic things:

- Look around the world
- Walk/Move to different locations
- Pick up objects
- Perform actions on the objects we are holding

Describing the Scenery with an Association List

```
;;;;  Global variable *nodes*
;;;;    defines the locations in the world

(defparameter *nodes*
 '((living-room                          ;key
     (you are in the living-room.    ;value
      a wizard is snoring loudly on the couch.))

   (garden
      (you are in a beautiful garden.
       there is a well in front of you.))

   (attic
      (you are in the attic. there is a giant
        welding torch in the corner.))))
```

Function Describe-Location

```
;;;;  An association list is a list containing
;;;;      (key value) sublists
;;;;  E.g. ( (location description)  ) in our Wiz.World
;;;;   Given the key, assoc returns the value


> (assoc 'garden *nodes*)
(GARDEN (YOU ARE IN A BEAUTIFUL GARDEN. THERE IS A WELL
  IN FRONT OF YOU.))

;;   The describe-location function uses assoc
(defun describe-location (location *nodes*)
  (cadr (assoc location *nodes*)))
```

Example

```
(defun describe-location
                  (location *nodes* )
    (cadr (assoc location *nodes* )))


>(describe-location
    'living-room *nodes*)


(YOU ARE IN THE LIVING-ROOM. A WIZARD
  IS SNORING LOUDLY ON THE COUCH.)
```

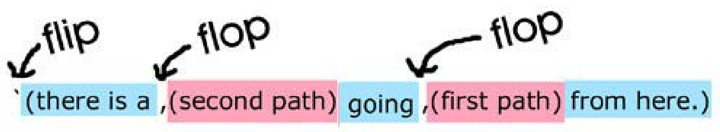
Global Variable for Paths *EDGES*

```
(defparameter *edges*
  '((living-room (garden west door)
             (attic upstairs
                    ladder))
    (garden (living-room east
                    door))
    (attic (living-room downstairs
                    ladder))))

;;;  The DESCRIBE-PATH function  uses the info in
;;; *edges* to create a sentence describing the path.

(defun describe-path (edge)
   `(there is a ,(caddr edge)
       going ,(cadr edge)
           from here. ))
```

Quasiquoting

- The **backquote** (`)

Both the single quote and backquote in Lisp “flip” a piece of code into **data mode**, but only a **backquote can also be *unquoted using the comma character***, to flip back into **code mode**.



```
`(there is a ,(second path) going ,(first path) from here.)
```

Macros

- Note: the backquote (`) is especially useful in **macros**
- Macros are ***special forms*** – their **parameters are not evaluated until** and unless needed.
- We’ve seen: `and,` `or` already

Describing Multiple Paths in 3 Steps

```
(cdr (assoc location edges)) ;#1
```

1. **Find the relevant edges.**
2.
3.

```
> (cdr (assoc 'living-room *edges*))
((GARDEN WEST DOOR) (ATTIC UPSTAIRS
  LADDER))
```

Describe Multiple Paths

```
(mapcar #'describe-path ;#2
   (cdr (assoc location edges))) ;#1
```

1.
2. **Convert the edges to descriptions.**
3.

Use the function describe-path **to generate the description of each path returned by part 1.**

```
(mapcar #’describe-path
   (cdr (assoc 'living-room *edges*))

(THERE IS A DOOR GOING WEST FROM HERE.)
(THERE IS A LADDER GOING UPSTAIRS FROM
  HERE.)
```

Concatenate the Descriptions

```
(apply #'append    ;#3
   (mapcar #'describe-path ;#2
     (cdr (assoc location edges))))) ;#1
```

1.
2.
3. **Join the descriptions together.**

Use the function append **to merge the descriptions returned by part 2 into a single list**

```
> (apply #'append '((THERE IS A DOOR
  GOING WEST FROM HERE.) (THERE IS A
  LADDER GOING UPSTAIRS FROM HERE.)))
   (THERE IS A DOOR GOING WEST FROM
  HERE. THERE IS A LADDER GOING
  UPSTAIRS FROM HERE.)
```

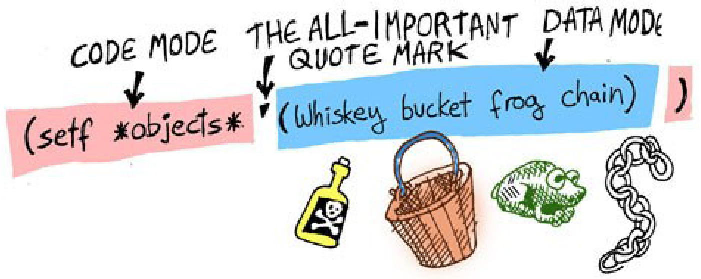
Describing Multiple Paths is Complete

```
(defun describe-paths (location
                             edges)
  (apply #'append ;#3
   (mapcar #'describe-path ;#2
     (cdr (assoc location edges)))));#1
```

1. **Find the relevant edges.**
2. **Convert the edges to descriptions.**
3. **Join the descriptions.**

Describing Objects at a Specific Location

```
> (defparameter *objects*
    '(whiskey bucket frog chain))
*OBJECTS*
```


CODE MODE
THE ALL-IMPORTANT QUOTE MARK
DATA MODE
(setf *objects* '(whiskey bucket frog chain))

Object Locations

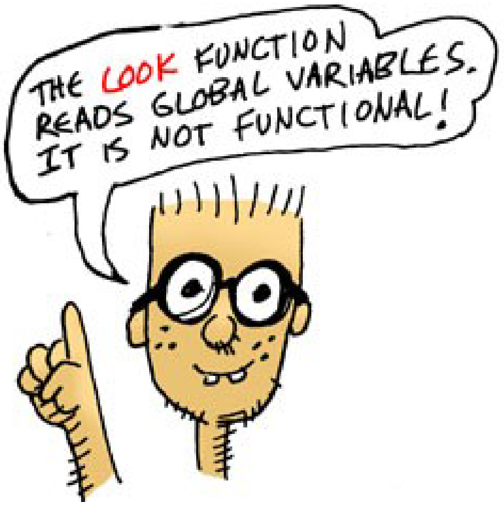
```
;;;; Create a global variable storing an
;;;;  association list of the items and
;;;;  their locations in the world

(defparameter *object-locations*
  '((whiskey living-room)
    (bucket living-room)
    (chain garden)
    (frog garden)))
```

Create a Function to Generate a List of Objects and Locations

```
(defun objects-at (loc objs obj-locs)
   (labels
    ((at-loc-p (obj)        ; function defn
        (eq (cadr
               (assoc obj obj-locs))
             loc)))
        (remove-if-not
             #'at-loc-p objs)))
```

Describing Everything

```
(defun look ()
  (append             ; put together all descriptions
     (describe-location
         *location* *nodes*)
      (describe-paths
         *location* *edges*)
      (describe-objects
         *location* *objects*
         *object-locations*)))
```


THE LOOK FUNCTION READS GLOBAL VARIABLES. IT IS NOT FUNCTIONAL!

Moving Around

```
(defun walk (direction)
   (let ((next
         (find direction
            (cdr (assoc
                 *location* *edges*))
            :key #'cadr)))
  (if next
    (progn
       (setf *location*
         (car next))
       (look))
      '(you cannot go that way.))))
```

Pickup Function

```
(defun pickup (object)
   (cond
     ((member
          object
          (objects-at *location*
           *objects*
               *object-locations*))
         (push
           (list object 'body)
            *object-locations*)
          `(you are now carrying the
               ,object))
     (t
          '(you cannot get that.))))
```

Inventory

```
(defun inventory ()
  (cons 'items-
         (objects-at
            'body
            *objects*
             *object-locations*)))

> (inventory)
(ITEMS- WHISKEY)
```

Summary

- There you have it!
- We now have a basic engine for a text adventure game.

We can

- See what is in a location with `look`,
- Move between places with `walk`,
- Add objects to our bag with `pickup`, and
- Check what is in our bag with `inventory`.