The Wizard's Adventure Game [1]

Eric Bailey

October 14, 2017 [2]

In this game, you are a wizard's apprentice.
You'll explore the wizard's house.

Contents

[1] Conrad Barski. *Land of Lisp: Learn to Program in Lisp, One Game at a Time!*, chapter 5, pages 67–84. No Starch Press, 2010. ISBN 9781593273491. URL http://landoflisp.com

[2] Last updated October 17, 2017

1 ⟨* 1⟩≡

```
(in-package :cl-user)
(defpackage lol.wizard5
  (:use :cl)
  (:export :look
           :walk
           :pickup
           :inventory))
(in-package :lol.wizard5)
```

⟨*define the global variables* 2d⟩

This definition is continued in chunks 2–7.

Root chunk (not used in this document).

Defines:
`lol.wizard5`, used in chunks 7 and 8.

Uses `inventory` 7b, `look` 5f, `pickup` 6l, and `walk` 6g.

Setting the Scene

This world consists of only three locations:

2a 1. ⟨*The living room* 2a⟩≡
```
you are in the living room.
a wizard is snoring loudly on the couch.
```
This code is used in chunks 2d and 8.

2b 2. ⟨*A beautiful garden* 2b⟩≡
```
you are in a beautiful garden.
there is a well in front of you.
```
This code is used in chunks 2d and 8b.

2c 3. ⟨*The attic* 2c⟩≡
```
you are in the attic.
there is a giant welding torch in the corner.
```
This code is used in chunk 2d.

nodes is simply an *association list* with locations as keys and the previous descriptions as values.

2d ⟨*define the global variables* 2d⟩≡
```
(defparameter *nodes*
  '((living-room (⟨The living room 2a⟩))
    (garden      (⟨A beautiful garden 2b⟩))
    (attic       (⟨The attic 2c⟩))))
```
This definition is continued in chunks 3–5.
This code is used in chunk 1.
Defines:
nodes, used in chunks 5f and 8a.

Describing the Location

To find the description, ⟨*look up a location* 2e⟩ and take the `cadr`. Preferring the *functional programming* style, pass `nodes` as an argument, instead of referencing *nodes* directly.

2e ⟨*look up a location* 2e⟩≡
```
(assoc location nodes)
```
This code is used in chunk 2f.

2f ⟨* 1⟩+≡
```
(defun describe-location (location nodes)
  (cadr ⟨look up a location 2e⟩))
```
Defines:
describe-location, used in chunks 5f and 8a.

Describing the Paths

3a ⟨*garden door* 3a⟩≡
```
THERE IS A DOOR GOING WEST FROM HERE.
```
This code is used in chunks 7c and 8a.

3b ⟨*living-room paths* 3b⟩≡
```
(garden west door)
```
This definition is continued in chunk 3d.
This code is used in chunk 3g.

3c ⟨*attic ladder* 3c⟩≡
```
THERE IS A LADDER GOING UPSTAIRS FROM HERE.
```
This code is used in chunk 7c.

3d ⟨*living-room paths* 3b⟩+≡
```
(attic upstairs ladder)
```
This code is used in chunk 3g.

3e ⟨*garden path* 3e⟩≡
```
(living-room east door)
```
This code is used in chunk 3g.

3f ⟨*attic path* 3f⟩≡
```
(living-room downstairs ladder)
```
This code is used in chunk 3g.

To describe such a symbolic path, take the means (`caddr`) and direction (`cadr`) and return a descriptive list.

3h ⟨* 1⟩+≡
```
(defun describe-path (edge)
  `(there is a ,(caddr edge) going ,(cadr edge) from here.))
```

Defines:
`describe-path`, used in chunk 8a.

From the `living-room`, you can move to the `garden` by going `west` through the `door`.

Or to the `attic` by going `upstairs` via the `ladder`.

From the `garden`, you can move to the `living-room` by going `east` through the `door`.

From the `attic`, you can move to the `living-room` by going `downstairs` via the `ladder`.

3g ⟨*define the global variables* 2d⟩+≡
```
(defparameter *edges*
  '((living-room ⟨living-room paths 3b⟩)
    (garden      ⟨garden path 3e⟩)
    (attic       ⟨attic path 3f⟩)))
```

This code is used in chunk 1.
Defines:
`*edges*`, used in chunks 5f, 6a, and 8a.

Describing Multiple Paths at Once

To describe multiple paths:

4a 1. ⟨*Find the relevant edges.* 4a⟩≡
```
(cdr (assoc location edges))
```
This code is used in chunk 4d.

4b 2. ⟨*Convert the edges to descriptions.* 4b⟩≡
```
mapcar #'describe-path
```
This code is used in chunk 4d.

4c 3. ⟨*Join the descriptions.* 4c⟩≡
```
apply #'append
```
This code is used in chunks 4d and 5d.

4d ⟨* 1⟩+≡
```
(defun describe-paths (location edges)
  (⟨Join the descriptions. 4c⟩ (⟨Convert the edges to descriptions. 4b⟩ ⟨Find the relevant edges. 4a⟩)))
```

Defines:
describe-paths, used in chunks 5f and 8a.

Describing Objects at a Specific Location

4e ⟨*define the global variables* 2d⟩+≡
```
(defparameter *objects* '(whiskey bucket frog chain))

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

This code is used in chunk 1.
Defines:
object-locations, used in chunks 5–8.
objects, used in chunks 5–8.

4f ⟨*at-loc-p* 4f⟩≡
```
(at-loc-p (obj)
  (eq (cadr (assoc obj obj-locs)) loc))
```
This code is used in chunk 4g.

4g ⟨* 1⟩+≡
```
(defun objects-at (loc objs obj-locs)
  (labels (⟨at-loc-p 4f⟩)
    (remove-if-not #'at-loc-p objs)))
```

Defines:
objects-at, used in chunks 5–8.

Describing Visible Objects

To describe the objects visible at a given location:

5a 1. ⟨*Find the objects at the current location.* 5a⟩≡
```
(objects-at loc objs obj-loc)
```
This code is used in chunk 5d.
Uses objects-at 4g.

5b 2. ⟨*Convert the objects to descriptions.* 5b⟩≡
```
mapcar #'describe-obj
```
This code is used in chunk 5d.

3. ⟨*Join the descriptions.* 4c⟩

5c ⟨*describe-obj* 5c⟩≡
```
(describe-obj (obj)
  `(you see a ,obj on the floor.))
```
This code is used in chunk 5d.

5d ⟨* 1⟩+≡
```
(defun describe-objects (loc objs obj-loc)
  (labels (⟨describe-obj 5c⟩)
    (⟨Join the descriptions. 4c⟩
          (⟨Convert the objects to descriptions. 5b⟩
                ⟨Find the objects at the current location. 5a⟩)))))
```

Defines:
describe-objects, used in chunks 5f and 8.

Describing It All

N.B. The look function is **not** functional, since it reads global variables.

5f ⟨* 1⟩+≡
```
(defun look ()
  (append (describe-location *location* *nodes*)
          (describe-paths *location* *edges*)
          (describe-objects *location* *objects* *object-locations*)))
```

Defines:
look, used in chunks 1, 6f, and 8b.
Uses *edges* 3g, *location* 5e, *nodes* 2d, *object-locations* 4e, *objects* 4e, describe-location 2f, describe-objects 5d, and describe-paths 4d.

5e ⟨*define the global variables* 2d⟩+≡
```
(defparameter *location* 'living-room)
```
This code is used in chunk 1.
Defines:
location, used in chunks 5 and 6.

Walking Around in Our World

Given a `direction`, ⟨*locate the path marked with the appropriate direction* 6c⟩ and ⟨*try to go in that direction* 6f⟩. Since the `direction` will be there, ⟨*match against the* `cadr` *of each path* 6b⟩.

6c ⟨*locate the path marked with the appropriate direction* 6c⟩≡
```
(find direction
      ⟨look up the available walkings paths 6a⟩
      ⟨match against the cadr of each path 6b⟩)
```
This code is used in chunk 6g.

If such a path is found, ⟨*adjust the player's position* 6d⟩, otherwise ⟨*admonish the player* 6e⟩.

6f ⟨*try to go in that direction* 6f⟩≡
```
(if next
    (progn ⟨adjust the player's position 6d⟩
           (look))
    ⟨admonish the player 6e⟩)
```
This code is used in chunk 6g.
Uses `look` 5f.

6g ⟨* 1⟩+≡
```
(defun walk (direction)
  (let ((next ⟨locate the path marked with the appropriate direction 6c⟩))
    ⟨try to go in that direction 6f⟩))
```

Defines:
`walk`, used in chunks 1 and 8b.

6a ⟨*look up the available walkings paths* 6a⟩≡
```
(cdr (assoc *location* *edges*))
```
This code is used in chunk 6c.
Uses `*edges*` 3g and `*location*` 5e.

6b ⟨*match against the* `cadr` *of each path* 6b⟩≡
```
:key #'cadr
```
This code is used in chunk 6c.

6d ⟨*adjust the player's position* 6d⟩≡
```
(setf *location* (car next))
```
This code is used in chunk 6f.
Uses `*location*` 5e.

6e ⟨*admonish the player* 6e⟩≡
```
'(you cannot go that way.)
```
This code is used in chunks 6f and 8b.

Picking Up Objects

To determine if ⟨*the object is on the floor* 6h⟩,

6i ⟨*get the list of objects here* 6i⟩≡
```
(objects-at *location* *objects* *object-locations*)
```
This code is used in chunk 6h.
Uses `*location*` 5e, `*object-locations*` 4e, `*objects*` 4e, and `objects-at` 4g.

... and check if `object` is a `member`. If so...

6j ⟨*pick it up* 6j⟩≡
```
(push (list object 'body) *object-locations*)
`(you are now carrying the ,object)
```
This code is used in chunk 6l.
Uses `*object-locations*` 4e.

Otherwise...

6k ⟨*you cannot get that.* 6k⟩≡
```
'(you cannot get that.)
```
This code is used in chunks 6l and 8b.

6h ⟨*the object is on the floor* 6h⟩≡
```
(member object ⟨get the list of objects here 6i⟩)
```
This code is used in chunk 6l.

6l ⟨* 1⟩+≡
```
(defun pickup (object)
  (if ⟨the object is on the floor 6h⟩
      (progn ⟨pick it up 6j⟩)
      ⟨you cannot get that. 6k⟩))
```

Defines:
`pickup`, used in chunks 1 and 8b.

Checking Our Inventory

To check our inventory, we ⟨*retrieve the list of carried objects* 7a⟩ and prepend (a.k.a. `cons`) the symbol `items-`.

7a ⟨*retrieve the list of carried objects* 7a⟩≡
```
(objects-at 'body *objects* *object-locations*)
```
This code is used in chunk 7b.
Uses *object-locations* 4e, *objects* 4e, and objects-at 4g.

7b ⟨* 1⟩+≡
```
(defun inventory ()
  (cons 'items- ⟨retrieve the list of carried objects 7a⟩))
```
Defines:
inventory, used in chunks 1 and 8b.

Tests

7i ⟨*test/wizard5.lisp* 7i⟩≡
```
(in-package :lol.wizard5)


(prove:plan 2)


⟨Test the private functions in lol.wizard5 8a⟩


⟨Test the exported functions in lol.wizard5. 8b⟩


(prove:finalize)
```
Root chunk (not used in this document).
Uses lol.wizard5 1.

7c ⟨*living-room path descriptions* 7c⟩≡
```
⟨garden door 3a⟩
⟨attic ladder 3c⟩
```
This code is used in chunk 8.

7d ⟨*living-room object descriptions* 7d⟩≡
```
YOU SEE A WHISKEY ON THE FLOOR.
YOU SEE A BUCKET ON THE FLOOR.
```
This code is used in chunk 8.

7e ⟨*garden path description* 7e⟩≡
```
THERE IS A DOOR GOING EAST FROM HERE.
```
This code is used in chunk 8b.

7f ⟨*garden object descriptions* 7f⟩≡
```
YOU SEE A FROG ON THE FLOOR.
YOU SEE A CHAIN ON THE FLOOR.
```
This code is used in chunk 8b.

7g ⟨*You've got whiskey!* 7g⟩≡
```
'(YOU ARE NOW CARRYING THE WHISKEY)
```
This code is used in chunk 8b.

7h ⟨*All you have is whiskey.* 7h⟩≡
```
'(ITEMS- WHISKEY)
```
This code is used in chunk 8b.

lol.wizard5 *(Private Parts)*

8a ⟨*Test the private functions in* lol.wizard5 8a⟩≡
```
(prove:subtest "lol.wizard5 (Private Parts)"
  (prove:is (describe-location 'living-room *nodes*)
            '(⟨The living room 2a⟩))
  (prove:is (describe-path '(garden west door))
            '(⟨garden door 3a⟩))
  (prove:is (describe-paths 'living-room *edges*)
            '(⟨living-room path descriptions 7c⟩))
  (prove:is (describe-objects 'living-room *objects* *object-locations*)
            '(⟨living-room object descriptions 7d⟩))
  (prove:is (objects-at 'living-room *objects* *object-locations*)
            '(WHISKEY BUCKET)))
```

This code is used in chunk 7i.
Uses *edges* 3g, *nodes* 2d, *object-locations* 4e, *objects* 4e, describe-location 2f, describe-objects 5d, describe-path 3h, describe-paths 4d, lol.wizard5 1, and objects-at 4g.

lol.wizard5 *(Public API)*

8b ⟨*Test the exported functions in* lol.wizard5. 8b⟩≡
```
(prove:subtest "lol.wizard5 (Public API)"
  (prove:is (look)
            '(⟨The living room 2a⟩
              ⟨living-room path descriptions 7c⟩
              ⟨living-room object descriptions 7d⟩))
  (prove:subtest "Pick up the whiskey"
    (prove:is (pickup 'whiskey)
              ⟨You've got whiskey! 7g⟩)
    (prove:is (objects-at 'living-room *objects* *object-locations*)
              '(BUCKET))
    (prove:is (describe-objects 'living-room *objects* *object-locations*)
              '(YOU SEE A BUCKET ON THE FLOOR.)))
  (prove:is (pickup 'the-pace)
            ⟨you cannot get that. 6k⟩)
  (prove:is (walk 'west)
            '(⟨A beautiful garden 2b⟩
              ⟨garden path description 7e⟩
              ⟨garden object descriptions 7f⟩))
  (prove:is (walk 'south)
            ⟨admonish the player 6e⟩)
  (prove:is (inventory)
            ⟨All you have is whiskey. 7h⟩))
```

This code is used in chunk 7i.
Uses *object-locations* 4e, *objects* 4e, describe-objects 5d, inventory 7b, lol.wizard5 1, look 5f, objects-at 4g, pickup 6l, and walk 6g.

Running the Tests

9c ⟨*Run the system tests.* 9c⟩≡
```
(prove:run-test-system :lol-test)
```
This code is used in chunk 9d.

9d ⟨*Run the system tests and exit.* 9d⟩≡
```
(uiop:quit (if ⟨Run the system tests. 9c⟩ 0 1))
```
This code is used in chunk 9j.

9e ⟨*script header* 9e⟩≡
```
#! /usr/bin/env nix-shell
#! nix-shell -i sh -p sbcl
```
This definition is continued in chunk 9.
This code is used in chunk 9j.

9j ⟨*bin/runtests* 9j⟩≡
```
⟨script header 9e⟩
     -eval "⟨Load the test package. 9h⟩" \
     -eval "⟨Run the system tests and exit. 9d⟩"

⟨script footer 9i⟩
```
Root chunk (not used in this document).

```
$ ./bin/runtests
✓ 2 tests completed (0ms)

Summary:
  All 1 file passed.
```

9k ⟨*init.lisp* 9k⟩≡
```
#-quicklisp
(let ((quicklisp-init (merge-pathnames "quicklisp/setup.lisp"
                                       (user-homedir-pathname))))
  (when (probe-file quicklisp-init)
    (load quicklisp-init)))

(push (concatenate 'string (sb-posix:getcwd) "/")
      asdf:*central-registry*)
```
Root chunk (not used in this document).

9a ⟨*Set the exit status.* 9a⟩≡
```
(if (null failures) 0 1)
```
Root chunk (not used in this document).

9b ⟨*Exit with an appropriate status code.* 9b⟩≡
```
(sb-posix:exit status)
```
Root chunk (not used in this document).

`prove` is yet another unit testing framework for Common Lisp.

See the Nixpkgs Contributors Guide for more information on using `nix-shell` as a shebang.

Run `sbcl` quietly:

9f ⟨*script header* 9e⟩+≡
```
sbcl -noinform -non-interactive \
```
This code is used in chunk 9j.

Load ⟨*init.lisp* 9k⟩ as the user initialization file:

9g ⟨*script header* 9e⟩+≡
```
     -userinit init.lisp \
```
This code is used in chunk 9j.

9h ⟨*Load the test package.* 9h⟩≡
```
(asdf:load-system :lol-test)
```
This code is used in chunk 9j.

9i ⟨*script footer* 9i⟩≡
```
# Local Variables:
# mode: sh
# End:
```
This code is used in chunk 9j.

Full Listing

```
(defparameter *nodes*
  '((living-room (you are in the living room.
                  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.))))

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

(defparameter *objects* '(whiskey bucket frog chain))

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

(defparameter *location* 'living-room)


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


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


(defun describe-paths (location edges)
  (apply #'append (mapcar #'describe-path (cdr (assoc location edges)))))


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

```
(defun describe-objects (loc objs obj-loc)
  (labels ((describe-obj (obj)
             `(you see a ,obj on the floor.)))
    (apply #'append
           (mapcar #'describe-obj
                   (objects-at loc objs obj-loc)))))


(defun look ()
  (append (describe-location *location* *nodes*)
          (describe-paths *location* *edges*)
          (describe-objects *location* *objects* *object-locations*)))


(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.))))


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


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

Chunks

Index

[3]

Glossary

association list a list of conses representing an assocation of keys with values, where the car of each cons is the key and the cdr is the associated value. 2

caddr (lambda (x) (car (cdr (cdr x)))) 3

cadr (lambda (x) (car (cdr x))) 2, 3

car

1. a. the first component of a cons; the other is the cdr.
   b. the head of a list, or nil if the list is the *empty list*.
2. the *object* that is held in the car. "The function car returns the car of a cons."

13

cdr

1. a. the second component of a cons; the other is the car.
   b. the tail of a list, or nil if the list is the *empty list*.
2. the *object* that is held in the cdr. "The function cdr returns the cdr of a cons."

13

cons

1. a compound data *object* made up of a car and a cdr.
2. to create such an *object*.
3. to create any *object* or to allocate storage.

13

empty list the list containing no elements. 13

nil represents both boolean false and the *empty list*. Alternatively notated as () to emphasize its use as an *empty list*. 13

object any Lisp datum. 13

[3] Kent M. Pitman. CLHS: Glossary. http://www.lispworks.com/documentation/HyperSpec/Body/26_a.htm, April 2005. Accessed: 2017-10-17

References

Conrad Barski. *Land of Lisp: Learn to Program in Lisp, One Game at a Time!*, chapter 5, pages 67–84. No Starch Press, 2010. ISBN 9781593273491. URL http://landoflisp.com.

Kent M. Pitman. CLHS: Glossary. http://www.lispworks.com/documentation/HyperSpec/Body/26_a.htm, April 2005. Accessed: 2017-10-17.