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* 5⟩

This definition is continued in chunks 7, 15, 19, 22, 26, 28, 35, 40, and 42.

Root chunk (not used in this document).

Defines:
`lol.wizard5`, used in chunks 49–51.

Uses `inventory` 42, `look` 28, `pickup` 40, and `walk` 35.

Setting the Scene

This world consists of only three locations:

2 1. ⟨*The living room* 2⟩≡
```
you are in the living room.
a wizard is snoring loudly on the couch.
```
This code is used in chunks 5, 50, and 51.

3 2. ⟨*A beautiful garden* 3⟩≡
```
you are in a beautiful garden.
there is a well in front of you.
```
This code is used in chunks 5 and 51.

4 3. ⟨*The attic* 4⟩≡
```
you are in the attic.
there is a giant welding torch in the corner.
```
This code is used in chunk 5.

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

5 ⟨*define the global variables* 5⟩≡
```
(defparameter *nodes*
  '((living-room (⟨The living room 2⟩))
    (garden      (⟨A beautiful garden 3⟩))
    (attic       (⟨The attic 4⟩))))
```
This definition is continued in chunks 14, 20, and 27.
This code is used in chunk 1.
Defines:
nodes, used in chunks 28 and 50.

Describing the Location

To find the description, ⟨*look up a location* 6⟩ and take the `cadr`. Preferring the *functional programming* style, pass `nodes` as an argument, instead of referencing *nodes* directly.

6 ⟨*look up a location* 6⟩≡
```
(assoc location nodes)
```
This code is used in chunk 7.

7 ⟨* 1⟩+≡
```
(defun describe-location (location nodes)
  (cadr ⟨look up a location 6⟩))
```
Defines:
describe-location, used in chunks 28 and 50.

Describing the Paths

8 ⟨*garden door* 8⟩≡
```
THERE IS A DOOR GOING WEST FROM HERE.
```
This code is used in chunks 43 and 50.

9 ⟨*living-room paths* 9⟩≡
```
(garden west door)
```
This definition is continued in chunk 11.
This code is used in chunk 14.

10 ⟨*attic ladder* 10⟩≡
```
THERE IS A LADDER GOING UPSTAIRS FROM HERE.
```
This code is used in chunk 43.

11 ⟨*living-room paths* 9⟩+≡
```
(attic upstairs ladder)
```
This code is used in chunk 14.

12 ⟨*garden path* 12⟩≡
```
(living-room east door)
```
This code is used in chunk 14.

13 ⟨*attic path* 13⟩≡
```
(living-room downstairs ladder)
```
This code is used in chunk 14.

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

15 ⟨* 1⟩+≡
```
(defun describe-path (edge)
  `(there is a ,(caddr edge) going ,(cadr edge) from here.))
```

Defines:
`describe-path`, used in chunk 50.

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`.

14 ⟨*define the global variables* 5⟩+≡
```
(defparameter *edges*
  '((living-room ⟨living-room paths 9⟩)
    (garden      ⟨garden path 12⟩)
    (attic       ⟨attic path 13⟩)))
```

This code is used in chunk 1.
Defines:
`*edges*`, used in chunks 28, 29, and 50.

Describing Multiple Paths at Once

To describe multiple paths:

16 1. ⟨*Find the relevant edges.* 16⟩≡
```
(cdr (assoc location edges))
```
This code is used in chunk 19.

17 2. ⟨*Convert the edges to descriptions.* 17⟩≡
```
mapcar #'describe-path
```
This code is used in chunk 19.

18 3. ⟨*Join the descriptions.* 18⟩≡
```
apply #'append
```
This code is used in chunks 19 and 26.

19 ⟨* 1⟩+≡
```
(defun describe-paths (location edges)
  (⟨Join the descriptions. 18⟩ (⟨Convert the edges to descriptions. 17⟩ ⟨Find the relevant edges. 16⟩)))
```

Defines:
describe-paths, used in chunks 28 and 50.

Describing Objects at a Specific Location

20 ⟨*define the global variables* 5⟩+≡
```
(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 28, 37, 38, 41, 50, and 51.
objects, used in chunks 28, 37, 41, 50, and 51.

21 ⟨*at-loc-p* 21⟩≡
```
(at-loc-p (obj)
  (eq (cadr (assoc obj obj-locs)) loc))
```
This code is used in chunk 22.

22 ⟨* 1⟩+≡
```
(defun objects-at (loc objs obj-locs)
  (labels (⟨at-loc-p 21⟩)
    (remove-if-not #'at-loc-p objs)))
```

Defines:
objects-at, used in chunks 23, 37, 41, 50, and 51.

Describing Visible Objects

To describe the objects visible at a given location:

23 1. ⟨*Find the objects at the current location.* 23⟩≡
```
(objects-at loc objs obj-loc)
```
This code is used in chunk 26.
Uses objects-at 22.

24 2. ⟨*Convert the objects to descriptions.* 24⟩≡
```
mapcar #'describe-obj
```
This code is used in chunk 26.

3. ⟨*Join the descriptions.* 18⟩

25 ⟨*describe-obj* 25⟩≡
```
(describe-obj (obj)
  `(you see a ,obj on the floor.))
```
This code is used in chunk 26.

26 ⟨* 1⟩+≡
```
(defun describe-objects (loc objs obj-loc)
  (labels (⟨describe-obj 25⟩)
    (⟨Join the descriptions. 18⟩
      (⟨Convert the objects to descriptions. 24⟩
        ⟨Find the objects at the current location. 23⟩))))
```

Defines:
describe-objects, used in chunks 28, 50, and 51.

Describing It All

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

28 ⟨* 1⟩+≡
```
(defun look ()
  (append (describe-location *location* *nodes*)
          (describe-paths *location* *edges*)
          (describe-objects *location* *objects* *object-locations*)))
```

Defines:
look, used in chunks 1, 34, and 51.
Uses *edges* 14, *location* 27, *nodes* 5, *object-locations* 20, *objects* 20, describe-location 7, describe-objects 26, and describe-paths 19.

27 ⟨*define the global variables* 5⟩+≡
```
(defparameter *location* 'living-room)
```
This code is used in chunk 1.
Defines:
location, used in chunks 28, 29, 32, and 37.

Walking Around in Our World

Given a direction, ⟨*locate the path marked with the appropriate direction* 31⟩ and ⟨*try to go in that direction* 34⟩. Since the direction will be there, ⟨*match against the* cadr *of each path* 30⟩.

29 ⟨*look up the available walkings paths* 29⟩≡
```
(cdr (assoc *location* *edges*))
```
This code is used in chunk 31.
Uses *edges* 14 and *location* 27.

31 ⟨*locate the path marked with the appropriate direction* 31⟩≡
```
(find direction
      ⟨look up the available walkings paths 29⟩
      ⟨match against the cadr of each path 30⟩)
```
This code is used in chunk 35.

30 ⟨*match against the* cadr *of each path* 30⟩≡
```
:key #'cadr
```
This code is used in chunk 31.

If such a path is found, ⟨*adjust the player's position* 32⟩, otherwise ⟨*admonish the player* 33⟩.

34 ⟨*try to go in that direction* 34⟩≡
```
(if next
    (progn ⟨adjust the player's position 32⟩
           (look))
    ⟨admonish the player 33⟩)
```
This code is used in chunk 35.
Uses look 28.

35 ⟨* 1⟩+≡
```
(defun walk (direction)
  (let ((next ⟨locate the path marked with the appropriate direction 31⟩))
    ⟨try to go in that direction 34⟩))
```
Defines:
walk, used in chunks 1 and 51.

32 ⟨*adjust the player's position* 32⟩≡
```
(setf *location* (car next))
```
This code is used in chunk 34.
Uses *location* 27.

33 ⟨*admonish the player* 33⟩≡
```
'(you cannot go that way.)
```
This code is used in chunks 34 and 51.

Picking Up Objects

To determine if ⟨*the object is on the floor* 36⟩,

36 ⟨*the object is on the floor* 36⟩≡
```
(member object ⟨get the list of objects here 37⟩)
```
This code is used in chunk 40.

37 ⟨*get the list of objects here* 37⟩≡
```
(objects-at *location* *objects* *object-locations*)
```
This code is used in chunk 36.
Uses *location* 27, *object-locations* 20, *objects* 20, and objects-at 22.

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

38 ⟨*pick it up* 38⟩≡
```
(push (list object 'body) *object-locations*)
`(you are now carrying the ,object)
```
This code is used in chunk 40.
Uses *object-locations* 20.

Otherwise...

39 ⟨*you cannot get that.* 39⟩≡
```
'(you cannot get that.)
```
This code is used in chunks 40 and 51.

40 ⟨* 1⟩+≡
```
(defun pickup (object)
  (if ⟨the object is on the floor 36⟩
      (progn ⟨pick it up 38⟩)
      ⟨you cannot get that. 39⟩))
```
Defines:
pickup, used in chunks 1 and 51.

Checking Our Inventory

To check our inventory, we ⟨*retrieve the list of carried objects* 41⟩ and prepend (a.k.a. cons) the symbol items-.

41 ⟨*retrieve the list of carried objects* 41⟩≡
```
(objects-at 'body *objects* *object-locations*)
```
This code is used in chunk 42.
Uses *object-locations* 20, *objects* 20, and objects-at 22.

42 ⟨* 1⟩+≡
```
(defun inventory ()
  (cons 'items- ⟨retrieve the list of carried objects 41⟩))
```
Defines:
inventory, used in chunks 1 and 51.

Tests

49 ⟨test/wizard5.lisp 49⟩≡
```
(in-package :lol.wizard5)


(prove:plan 2)


⟨Test the private functions in lol.wizard5 50⟩


⟨Test the exported functions in lol.wizard5. 51⟩


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

lol.wizard5 (Private Parts)

50 ⟨Test the private functions in lol.wizard5 50⟩≡
```
(prove:subtest "lol.wizard5 (Private Parts)"
  (prove:is (describe-location 'living-room *nodes*)
            '(⟨The living room 2⟩))
  (prove:is (describe-path '(garden west door))
            '(⟨garden door 8⟩))
  (prove:is (describe-paths 'living-room *edges*)
            '(⟨living-room path descriptions 43⟩))
  (prove:is (describe-objects 'living-room *objects* *object-locations*)
            '(⟨living-room object descriptions 44⟩))
  (prove:is (objects-at 'living-room *objects* *object-locations*)
            '(WHISKEY BUCKET)))
```
This code is used in chunk 49.
Uses *edges* 14, *nodes* 5, *object-locations* 20, *objects* 20, describe-location 7, describe-objects 26, describe-path 15, describe-paths 19, lol.wizard5 1, and objects-at 22.

43 ⟨living-room path descriptions 43⟩≡
```
⟨garden door 8⟩
⟨attic ladder 10⟩
```
This code is used in chunks 50 and 51.

44 ⟨living-room object descriptions 44⟩≡
```
YOU SEE A WHISKEY ON THE FLOOR.
YOU SEE A BUCKET ON THE FLOOR.
```
This code is used in chunks 50 and 51.

45 ⟨garden path description 45⟩≡
```
THERE IS A DOOR GOING EAST FROM HERE.
```
This code is used in chunk 51.

46 ⟨garden object descriptions 46⟩≡
```
YOU SEE A FROG ON THE FLOOR.
YOU SEE A CHAIN ON THE FLOOR.
```
This code is used in chunk 51.

47 ⟨You've got whiskey! 47⟩≡
```
'(YOU ARE NOW CARRYING THE WHISKEY)
```
This code is used in chunk 51.

48 ⟨All you have is whiskey. 48⟩≡
```
'(ITEMS- WHISKEY)
```
This code is used in chunk 51.

lol.wizard5 *(Public API)*

51 ⟨*Test the exported functions in* lol.wizard5. 51⟩≡

```
(prove:subtest "lol.wizard5 (Public API)"
  (prove:is (look)
            '(⟨The living room 2⟩
              ⟨living-room path descriptions 43⟩
              ⟨living-room object descriptions 44⟩))
  (prove:subtest "Pick up the whiskey"
    (prove:is (pickup 'whiskey)
              ⟨You've got whiskey! 47⟩)
    (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. 39⟩)
  (prove:is (walk 'west)
            '(⟨A beautiful garden 3⟩
              ⟨garden path description 45⟩
              ⟨garden object descriptions 46⟩))
  (prove:is (walk 'south)
            ⟨admonish the player 33⟩)
  (prove:is (inventory)
            ⟨All you have is whiskey. 48⟩))
```

This code is used in chunk 49.
Uses *object-locations* 20, *objects* 20, describe-objects 26, inventory 42, lol.wizard5 1, look 28, objects-at 22, pickup 40, and walk 35.

Running the Tests

Describe `prove`

54 ⟨*Run the system tests.* 54⟩≡
```
(prove:run-test-system :lol-test)
```
This code is used in chunk 55.

55 ⟨*Run the system tests and exit.* 55⟩≡
```
(uiop:quit (if ⟨Run the system tests. 54⟩ 0 1))
```
This code is used in chunk 61.

Describe nix-shell shebang

56 ⟨*script header* 56⟩≡
```
#! /usr/bin/env nix-shell
#! nix-shell -i sh -p sbcl
```

This definition is continued in chunks 57 and 58.
This code is used in chunk 61.

61 ⟨*bin/runtests* 61⟩≡
```
⟨script header 56⟩
      -eval "⟨Load the test package. 59⟩" \
      -eval "⟨Run the system tests and exit. 55⟩"


⟨script footer 60⟩
```
Root chunk (not used in this document).

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

Summary:
  All 1 file passed.
```

62 ⟨*init.lisp* 62⟩≡
```
#-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).

52 ⟨*Set the exit status.* 52⟩≡
```
(if (null failures) 0 1)
```
Root chunk (not used in this document).

53 ⟨*Exit with an appropriate status code.* 53⟩≡
```
(sb-posix:exit status)
```
Root chunk (not used in this document).

Run `sbcl` quietly:

57 ⟨*script header* 56⟩+≡
```
sbcl -noinform -non-interactive \
```
This code is used in chunk 61.

Load ⟨*init.lisp* 62⟩ as the user initialization file:

58 ⟨*script header* 56⟩+≡
```
      -userinit init.lisp \
```
This code is used in chunk 61.

59 ⟨*Load the test package.* 59⟩≡
```
(asdf:load-system :lol-test)
```
This code is used in chunk 61.

60 ⟨*script footer* 60⟩≡
```
# Local Variables:
# mode: sh
# End:
```
This code is used in chunk 61.

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)))))


```

```
            (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, 14

list WRITE ME 13

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

`nil` represents both boolean `false` and the *empty list*. Alternatively notated as `()` to emphasize its use as an *empty list*. 13, 14

object any Lisp datum. 13

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.