

ERIC BAILEY

# PAIP EXERCISES



# Contents

<i>Introduction to Common Lisp</i>	5
<i>Overview of Lisp</i>	9
<i>GPS: The General Problem Solver</i>	11
<i>Package</i>	15
<i>Test Runner</i>	17
<i>Chunks</i>	19
<i>Index</i>	21
<i>Bibliography</i>	23



# Introduction to Common Lisp

## Using Functions

- 1  $\langle \text{titles } 1 \rangle \equiv$  (7)
- ```
(defparameter *titles*  
  '(Mr Mrs Miss Ms Sir Madam Dr Admiral Major General)  
  "A list of titles that can appear at the start of a name.")
```
- Defines:  
\*titles\*, used in chunk 4.
- 2  $\langle \text{abstract first-name } 2 \rangle \equiv$
- ```
( $\langle \text{function first-name(name): } 3 \rangle$   
  ( $\langle \text{if the first element of name is a title } 4 \rangle$   
     $\langle \text{then return the first-name of the rest of the name } 5 \rangle$   
     $\langle \text{else return the first element of the name } 6 \rangle$ ))
```
- 3  $\langle \text{function first-name(name): } 3 \rangle \equiv$  (2)
- ```
defun first-name (name)  
  "Select the first name from a name represented as a list."
```
- 4  $\langle \text{if the first element of name is a title } 4 \rangle \equiv$  (2)
- ```
if (member (first name) *titles*)
```
- Uses \*titles\* 1.
- 5  $\langle \text{then return the first-name of the rest of the name } 5 \rangle \equiv$  (2)
- ```
(first-name (rest name))
```
- 6  $\langle \text{else return the first element of the name } 6 \rangle \equiv$  (2)
- ```
(first name)
```

## Exercises

```

7  <intro.lisp 7>≡
    (in-package :paip)
    (defpackage paip.intro
      (:use :cl :lisp-unit))
    (in-package :paip.intro)

    <titles 1>

    ;; Exercise 1.1
    <Exercise 1.1 8>

    ;; Exercise 1.2
    <Exercise 1.2 17>

    ;; Exercise 1.3
    <Exercise 1.3 24>

```

### Exercise 1.1

Define a version of `last-name` that handles “Rex Morgan MD,” “Morton Downey, Jr.,” and whatever other cases you can think of.

```

8  <Exercise 1.1 8>≡ (7)
    <suffixes 13>

    <last-name 9>

    <Exercise 1.1 tests 14>

9  <last-name 9>≡ (8)
    (defun last-name (name)
      "Select the last name from a name represented as a list."
      (if <the last element of a name is a suffix 10>
          <then return the last-name of all but the last element of the name 11>
          <else return the last element of the name 12>)))

```

Defines:

`last-name`, used in chunks 11, 15, and 16.

First, we check to see if the last element of the `name` is a suffix, i.e. whether it's a member of `*suffixes*`.

```

10 <the last element of a name is a suffix 10>≡ (9)
    (member (first (last name)) *suffixes*)

```

Uses `*suffixes*` 13.

If it is, then drop it from the `name` and return the `last-name` of the result.

```

11 <then return the last-name of all but the last element of the name 11>≡ (9)
    (last-name (butlast name))

```

Uses `last-name` 9.

Otherwise, it's the last name, so return it.

12  $\langle \text{else return the last element of the name } 12 \rangle \equiv$  (9)  
`(first (last name))`

13  $\langle \text{suffixes } 13 \rangle \equiv$  (8)  
`(defparameter *suffixes*`  
`'(MD Jr)`  
`"A list of suffixes that can appear at the end of a name.")`

Defines:

`*suffixes*`, used in chunk 10.

14  $\langle \text{Exercise 1.1 tests } 14 \rangle \equiv$  (8)  
`(define-test test-last-name`  
 $\langle \text{Rex Morgan MD } 15 \rangle$   
 $\langle \text{Morton Downey, Jr } 16 \rangle$ )

15  $\langle \text{Rex Morgan MD } 15 \rangle \equiv$  (14) Assert that the `last-name` of Rex Morgan MD is Morgan.  
`(assert-equal 'Morgan (last-name '(Rex Morgan MD)))`  
 Uses last-name 9.

16  $\langle \text{Morton Downey, Jr } 16 \rangle \equiv$  (14)  
`(assert-equal 'Downey (last-name '(Morton Downey Jr)))`  
 Uses last-name 9.

### Exercise 1.2

Write a function to exponentiate, or raise a number to an integer power.  
 For example  $(\text{power } 3 \ 2) = 3^2 = 9$ .

17  $\langle \text{Exercise 1.2 } 17 \rangle \equiv$  (7)  
 $\langle \text{square } 22 \rangle$   
 $\langle \text{power } 18 \rangle$   
 $\langle \text{Exercise 1.2 tests } 23 \rangle$

18  $\langle \text{power } 18 \rangle \equiv$  (17)  
`(defun power (x n)`  
`"Raise x to the power of n."`  
`(cond  $\langle \text{if n is zero return } 1 \ 19 \rangle$`   
 $\langle \text{if n is even return x to the power of n over two, squared } 20 \rangle$   
 $\langle \text{otherwise return x times x to the power of n minus one } 21 \rangle$ )

Defines:

`power`, used in chunks 20, 21, and 23.

19  $\langle \text{if n is zero return } 1 \ 19 \rangle \equiv$  (18)  $x^0 = 1$   
`((zerop n) 1)`

20  $\langle \text{if n is even return x to the power of n over two, squared } 20 \rangle \equiv$  (18)  
`((evenp n) (square (power x (/ n 2))))`  
 Uses power 18 and square 22.

$$x^n = \begin{cases} 1 & \text{if } n = 0, \\ (x^{n/2})^2 & \text{if } n \text{ is even,} \\ x \times x^{n-1} & \text{otherwise.} \end{cases}$$

21  $\langle \text{otherwise return } x \text{ times } x \text{ to the power of } n \text{ minus one } 21 \rangle \equiv$  (18)  
 (t (\* x (power x (- n 1))))

Uses power 18.

22  $\langle \text{square } 22 \rangle \equiv$  (17)  $\text{square}(x) = x^2$   
 (defun square (x) (expt x 2))

Defines:

square, used in chunk 20.

23  $\langle \text{Exercise 1.2 tests } 23 \rangle \equiv$  (17)  
 (define-test test-power  
 (assert-equal 9 (power 3 2)))

Uses power 18.

### Exercise 1.3

Write a function that counts the number of atoms in an expression.

For example: (count-atoms '(a (b) c)) = 3. Notice that there is something of an ambiguity in this: should (a nil c) count as three atoms, or as two, because it is equivalent to (a () c)?

24  $\langle \text{Exercise 1.3 } 24 \rangle \equiv$  (7)  
 (defun count-atoms (exp)  
 "Return the total number of non-nil atoms in the expression."  
 (cond  $\langle \text{if exp is nil there are no atoms } 25 \rangle$   
 $\langle \text{if exp is an atom there is only one } 26 \rangle$   
 $\langle \text{otherwise add the count of the atoms in the first and rest of exp } 27 \rangle$ ))

Defines:

count-atoms, used in chunk 27.

25  $\langle \text{if exp is nil there are no atoms } 25 \rangle \equiv$  (24)  
 ((null exp) 0)

26  $\langle \text{if exp is an atom there is only one } 26 \rangle \equiv$  (24)  
 ((atom exp) 1)

27  $\langle \text{otherwise add the count of the atoms in the first and rest of exp } 27 \rangle \equiv$  (24)  
 (t (+ (count-atoms (first exp))  
 (count-atoms (rest exp))))

Uses count-atoms 24.



## Overview of Lisp

```
28  <find-all 28>≡ (29)
      (defun find-all (item sequence &rest keyword-args
                        &key (test #'eql) test-not &allow-other-keys)
        "Find all those elements of sequence that match item,
        according to the keywords. Doesn't alter sequence."
        (if test-not
            (apply #'remove item sequence
                    :test-not (complement test-not) keyword-args)
            (apply #'remove item sequence
                    :test (complement test) keyword-args)))
```

Defines:

find-all, used in chunk 34.



## GPS: The General Problem Solver

```
29  <gps.lisp 29>≡
    (in-package :paip)
    (defpackage paip.gps
      (:use :cl :lisp-unit)
      (:export :GPS))
    (in-package :paip.gps)

    <find-all 28>

    <The current state: a list of conditions 30>

    <A list of available operators 31>

    <An operation with preconds, add-list and del-list 32>

    <Solve a goal from a state using a list of operators 33>

    <Achieve an individual goal 34>

    <Achieve all goals 35>

    <Decide if an operator is appropriate for a goal 36>

    <Apply operator to current state 37>

    <Define a list of operations 39>

    <Assert that a given problem is solvable 40>

    <GPS Tests 41>
    Uses GPS 33.

30  <The current state: a list of conditions 30>≡ (29)
    (defvar *state* nil "The current state: a list of conditions.")
    Defines:
    *state*, used in chunks 33-35 and 37.

31  <A list of available operators 31>≡ (29)
    (defvar *ops* nil "A list of available operators.")
    Defines:
    *ops*, used in chunks 33 and 34.
```

32 *⟨An operation with preconds, add-list and del-list 32⟩*≡ (29)

```
(defstruct op
  "An operation"
  (action nil)
  (preconds nil)
  (add-list nil)
  (del-list nil))
```

33 *⟨Solve a goal from a state using a list of operators 33⟩*≡ (29)

```
(defun GPS (*state* goals *ops*)
  "General Problem Solver: achieve all goals using *ops*."
  (if (achieve-all goals) 'solved))
```

Defines:

GPS, used in chunk 29.

Uses *\*ops\** 31, *\*state\** 30, *achieve* 34, and *achieve-all* 35.

34 *⟨Achieve an individual goal 34⟩*≡ (29)

```
(defun achieve (goal)
  "A goal is achieved if it already holds,
  or if there is an appropriate op for it that is applicable."
  (or (member goal *state*)
      (some #'apply-op
            (find-all goal *ops* :test #'appropriate-p))))
```

Defines:

*achieve*, used in chunks 33 and 35.

Uses *\*ops\** 31, *\*state\** 30, *apply-op* 37, *appropriate-p* 36, and *find-all* 28.

35 *⟨Achieve all goals 35⟩*≡ (29)

```
(defun achieve-all (goals)
  "Try to achieve each goal, then make sure they still hold."
  (and (every #'achieve goals) (subsetp goals *state*)))
```

Defines:

*achieve-all*, used in chunks 33 and 37.

Uses *\*state\** 30 and *achieve* 34.

36 *⟨Decide if an operator is appropriate for a goal 36⟩*≡ (29)

```
(defun appropriate-p (goal op)
  "An op is appropriate to a goal if it is in its add list."
  (member goal (op-add-list op)))
```

Defines:

*appropriate-p*, used in chunk 34.

37 *⟨Apply operator to current state 37⟩*≡ (29)

```
(defun apply-op (op)
  "Print a message and update *state* if op is applicable."
  (when (achieve-all (op-preconds op))
    (print (list 'executing (op-action op)))
    (setf *state* (set-difference *state* (op-del-list op)))
    (setf *state* (union *state* (op-add-list op)))
    t))
```

Defines:

*apply-op*, used in chunk 34.

Uses *\*state\** 30 and *achieve-all* 35.

To drive the son to school, the son must start at home and the car must work.

```

38  <Drive son to school 38>≡ (39)
    (make-op :action 'drive-son-to-school
              :preconds '(son-at-home car-works)
              :add-list '(son-at-school)
              :del-list '(son-at-home))

39  <Define a list of operations 39>≡ (29)
    (defparameter *school-ops*
      (list
        <Drive son to school 38>
        (make-op :action 'shop-installs-battery
                  :preconds '(car-needs-battery shop-knows-problem shop-has-money)
                  :add-list '(car-works))
        (make-op :action 'tell-shop-problem
                  :preconds '(in-communication-with-shop)
                  :add-list '(shop-knows-problem))
        (make-op :action 'telephone-shop
                  :preconds '(know-phone-number)
                  :add-list '(in-communication-with-shop))
        (make-op :action 'look-up-number
                  :preconds '(have-phone-book)
                  :add-list '(know-phone-number))
        (make-op :action 'give-shop-money
                  :preconds '(have-money)
                  :add-list '(shop-has-money)
                  :del-list '(have-money))))

```

## Tests

```

40  <Assert that a given problem is solvable 40>≡ (29)
    (defmacro assert-solved (form)
      '(assert-equal 'solved ,form))

```

Defines:

assert-solved, used in chunk 41.

```

41  <GPS Tests 41>≡ (29)
    (define-test complex
      (assert-equal
        (format nil "~%~{(EXECUTING ~A) ~^~%~}"
          '(look-up-number
            telephone-shop
            tell-shop-problem
            give-shop-money
            shop-installs-battery
            drive-son-to-school))
        (with-output-to-string (*standard-output*)
          (assert-solved
            (gps '(son-at-home car-needs-battery have-money have-phone-book)
              '(son-at-school)
              *school-ops*))))))

    (define-test unsolvable
      (assert-nil (gps '(son-at-home car-needs-battery have-money)
        '(son-at-school)
        *school-ops*)))

    (define-test simple
      (assert-solved (gps '(son-at-home car-works)
        '(son-at-school)
        *school-ops*)))

    (define-test money-leftover
      (assert-solved (gps '(son-at-home have-money car-works)
        '(have-money son-at-school)
        *school-ops*)))

    (define-test clobbered-sibling
      (assert-nil (gps '(son-at-home car-needs-battery have-money have-phone-book)
        '(have-money son-at-school)
        *school-ops*)))

```

Uses assert-solved 40.

# Package

```
42  <paip.asd 42>≡  
    ;;;; paip.asd  
  
    (asdf:defsystem paip  
      :description "Paradigms of Artificial Intelligence Programming exercises"  
      :author "Eric Bailey <eric@ericb.me>"  
      ;; TODO :license "Specify license here"  
      :depends-on (:lisp-unit)  
      :serial t  
      :components ((:module "src"  
                    :components  
                    ((:module "paip"  
                      :components  
                      ((:file "intro")  
                      (:file "gps"))))))))  
  
    (defpackage paip  
      (:use :cl)  
      (in-package :paip))
```





# Test Runner

```
43 <runtests 43>≡
    #! /usr/bin/env nix-shell
    #! nix-shell -i sh -p sbcl

    # N.B. quicklisp must be installed and configured.

    sbcl -noinform -non-interactive \
        -userinit src/paip/init.lisp \
        -eval "(in-package :paip.$1)" \
        -eval "(let* ((results (lisp-unit:run-tests :all :paip.$1))
                      (failures (lisp-unit:failed-tests results))
                      (status (if (null failures) 0 1)))
                (lisp-unit:print-failures results)
                (sb-posix:exit status))"
```

```
44 <init.lisp 44>≡
    #-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*)

    (asdf:load-system :paip)
```



# Chunks

*⟨A list of available operators 31⟩*  
*⟨abstract first-name 2⟩*  
*⟨Achieve all goals 35⟩*  
*⟨Achieve an individual goal 34⟩*  
*⟨An operation with preconds, add-list and del-list 32⟩*  
*⟨Apply operator to current state 37⟩*  
*⟨Assert that a given problem is solvable 40⟩*  
*⟨Decide if an operator is appropriate for a goal 36⟩*  
*⟨Define a list of operations 39⟩*  
*⟨Drive son to school 38⟩*  
*⟨else return the first element of the name 6⟩*  
*⟨else return the last element of the name 12⟩*  
*⟨Exercise 1.1 8⟩*  
*⟨Exercise 1.1 tests 14⟩*  
*⟨Exercise 1.2 17⟩*  
*⟨Exercise 1.2 tests 23⟩*  
*⟨Exercise 1.3 24⟩*  
*⟨find-all 28⟩*  
*⟨function first-name(name): 3⟩*  
*⟨GPS Tests 41⟩*  
*⟨gps.lisp 29⟩*  
*⟨if exp is an atom there is only one 26⟩*  
*⟨if exp is nil there are no atoms 25⟩*  
*⟨if n is even return x to the power of n over two, squared 20⟩*  
*⟨if n is zero return 1 19⟩*  
*⟨if the first element of name is a title 4⟩*  
*⟨init.lisp 44⟩*  
*⟨intro.lisp 7⟩*  
*⟨last-name 9⟩*  
*⟨Morton Downey, Jr 16⟩*  
*⟨otherwise add the count of the atoms in the first and rest of exp 27⟩*  
*⟨otherwise return x times x to the power of n minus one 21⟩*  
*⟨paip.asd 42⟩*  
*⟨power 18⟩*

*⟨Rex Morgan MD 15⟩*  
*⟨runtests 43⟩*  
*⟨Solve a goal from a state using a list of operators 33⟩*  
*⟨square 22⟩*  
*⟨suffixes 13⟩*  
*⟨The current state: a list of conditions 30⟩*  
*⟨the last element of a name is a suffix 10⟩*  
*⟨then return the last-name of all but the last element of the name 11⟩*  
*⟨then return the first-name of the rest of the name 5⟩*  
*⟨titles 1⟩*

# *Index*

\*ops\*: [31](#), [33](#), [34](#)  
\*state\*: [30](#), [33](#), [34](#), [35](#), [37](#)  
\*suffixes\*: [10](#), [13](#)  
\*titles\*: [1](#), [4](#)  
achieve: [33](#), [34](#), [35](#)  
achieve-all: [33](#), [35](#), [37](#)  
apply-op: [34](#), [37](#)  
appropriate-p: [34](#), [36](#)  
assert-solved: [40](#), [41](#)  
count-atoms: [24](#), [27](#)  
find-all: [28](#), [34](#)  
GPS: [29](#), [33](#)  
last-name: [9](#), [11](#), [15](#), [16](#)  
power: [18](#), [20](#), [21](#), [23](#)  
square: [20](#), [22](#)



## *Bibliography*