

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>	19
<i>Test Runner</i>	21
<i>Chunks</i>	23
<i>Index</i>	25
<i>Bibliography</i>	27

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  <src/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>
  Uses use 43.

```

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)))

      (setf (symbol-function 'find-all-if) #'remove-if-not)
Defines:
  find-all, used in chunk 34.
```


GPS: The General Problem Solver

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

    <find-all 28>

    <A list of available operators 30>

    <An operation with preconds, add-list and del-list 31>

    <Solve a goal from a state using a list of operators 32>

    <Achieve an individual goal 34>

    <Achieve all goals 33>

    <Decide if an operator is appropriate for a goal 35>

    <Apply operator to current state 36>

    <Auxiliary Functions 37>

    <Define a list of operations 46>

    <Convert existing operators 42>

    <Print debugging information 48>

    <GPS Tests 49>
    Uses debug 48, GPS 32, and use 43.

30  <A list of available operators 30>≡                                     (29)
    (defvar *ops* nil "A list of available operators.")
    Defines:
    *ops*, used in chunks 32, 34, and 43.
```

31 *⟨An operation with preconds, add-list and del-list 31⟩*≡ (29)

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

Uses op 41.

32 *⟨Solve a goal from a state using a list of operators 32⟩*≡ (29)

```
(defun GPS (state goals &optional (*ops* *ops*))
  "General Problem Solver: from state, achieve goals using *ops*."
  (remove-if #'atom (achieve-all (cons '(start) state) goals nil)))
```

Defines:

GPS, used in chunk 29.

Uses *ops* 30, achieve 34, and achieve-all 33.

33 *⟨Achieve all goals 33⟩*≡ (29)

```
(defun achieve-all (state goals goal-stack)
  "Try to achieve each goal, then make sure they still hold."
  (let ((current-state state))
    (if (and (every #'(lambda (g)
                        (setf current-state
                            (achieve current-state g goal-stack)))
              goals)
          (subsetp goals current-state :test #'equal))
        current-state)))
```

Defines:

achieve-all, used in chunks 32 and 36.

Uses achieve 34.

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

```
(defun achieve (state goal goal-stack)
  "A goal is achieved if it already holds,
  or if there is an appropriate op for it that is applicable."
  (dbg-indent :gps (length goal-stack) "Goal: ~a" goal)
  (cond ((member-equal goal state) state)
        ((member-equal goal goal-stack) nil)
        (t (some #'(lambda (op) (apply-op state goal op goal-stack))
                  (find-all goal *ops* :test #'appropriate-p)))))
```

Defines:

achieve, used in chunks 32 and 33.

Uses *ops* 30, apply-op 36, appropriate-p 35, dbg-indent 48, find-all 28, member-equal 44, and op 41.

35 *⟨Decide if an operator is appropriate for a goal 35⟩*≡ (29)

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

Defines:

appropriate-p, used in chunk 34.

Uses member-equal 44 and op 41.

36 $\langle \text{Apply operator to current state 36} \rangle \equiv$ (29)

```
(defun apply-op (state goal op goal-stack)
  "Return a new, transformed state if op is applicable."
  (dbg-indent :gps (length goal-stack) "Consider: ~a" (op-action op))
  (let ((state* (achieve-all state (op-preconds op)
                             (cons goal goal-stack))))
    (unless (null state*)
      (dbg-indent :gps (length goal-stack) "Action: ~a" (op-action op))
      (append (remove-if #'(lambda (x)
                             (member-equal x (op-del-list op)))
                        state*)
              (op-add-list op))))))
```

Defines:

apply-op, used in chunk 34.

Uses achieve-all 33, dbg-indent 48, member-equal 44, and op 41.

Auxiliary Functions

37 $\langle \text{Auxiliary Functions 37} \rangle \equiv$ (29)

$\langle \text{Is a condition an executing form? 38} \rangle$

$\langle \text{Is the argument a list that starts with a given atom? 39} \rangle$

$\langle \text{Convert an operator to use the executing convention 40} \rangle$

$\langle \text{Create an operator 41} \rangle$

$\langle \text{Use a list of operators 43} \rangle$

$\langle \text{Test if an element is equal to a member of a list 44} \rangle$

38 $\langle \text{Is a condition an executing form? 38} \rangle \equiv$ (37)

```
(defun executing-p (x)
  "Is x of the form: (executing ...) ?"
  (starts-with x 'executing))
```

Defines:

executing-p, used in chunk 40.

Uses starts-with 39.

39 $\langle \text{Is the argument a list that starts with a given atom? 39} \rangle \equiv$ (37)

```
(defun starts-with (list x)
  "Is this a list whose first element is x?"
  (and (consp list) (eql (first list) x)))
```

Defines:

starts-with, used in chunk 38.

40 \langle Convert an operator to use the executing convention 40 $\rangle \equiv$ (37)

```
(defun convert-op (op)
  "Make op conform to the (EXECUTING op) convention."
  (unless (some #'executing-p (op-add-list op))
    (push (list 'executing (op-action op)) (op-add-list op)))
  op)
```

Defines:

convert-op, used in chunks 41 and 42.

Uses executing-p 38 and op 41.

41 \langle Create an operator 41 $\rangle \equiv$ (37)

```
(defun op (action &key preconds add-list del-list)
  "Make a new operator that obeys the (EXECUTING op) convention."
  (convert-op (make-op :action action
                      :preconds preconds
                      :add-list add-list
                      :del-list del-list)))
```

Defines:

op, used in chunks 31, 34–36, and 40.

Uses convert-op 40.

42 \langle Convert existing operators 42 $\rangle \equiv$ (29)

```
(mapc #'convert-op *school-ops*)
```

Uses *school-ops* 46 and convert-op 40.

43 \langle Use a list of operators 43 $\rangle \equiv$ (37)

```
(defun use (oplist)
  "Use oplist as the default list of operators."
  (length (setf *ops* oplist)))
```

Defines:

use, used in chunks 7, 29, and 50.

Uses *ops* 30.

44 \langle Test if an element is equal to a member of a list 44 $\rangle \equiv$ (37)

```
(defun member-equal (item list)
  (member item list :test #'equal))
```

Defines:

member-equal, used in chunks 34–36.

Nursery School Example

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

45 \langle Drive son to school 45 $\rangle \equiv$ (46)

```
(make-op :action 'drive-son-to-school
        :preconds '(son-at-home car-works)
        :add-list '(son-at-school)
        :del-list '(son-at-home))
```

46 *⟨Define a list of operations 46⟩*≡ (29)

```
(defparameter *school-ops*
  (list
    ⟨Drive son to school 45⟩
    (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))))
```

Defines:

school-ops, used in chunks 42 and 49.

Debugging

47 *⟨Debugging usage 47⟩*≡

```
;; Example call
(dbg :gps "The current goal is: ~a" goal)

;; Turn on debugging
(debug :gps)

;; Turn off debugging
(undebug :gps)
```

Uses dbg 48, debug 48, and undebug 48.

```

48 <Print debugging information 48>≡ (29)
  (defvar *dbg-ids* nil
    "Identifiers used by dbg")

  (defun dbg (id format-string &rest args)
    "Print debugging info if (DEBUG ID) has been specified."
    (when (member id *dbg-ids*)
      (fresh-line *debug-io*)
      (apply #'format *debug-io* format-string args)))

  (defun debug (&rest ids)
    "Start dbg output on the given ids."
    (setf *dbg-ids* (union ids *dbg-ids*)))

  (defun undebg (&rest ids)
    "Stop dbg on the ids. With no ids, stop dbg altogether."
    (setf *dbg-ids* (if (null ids) nil
      (set-difference *dbg-ids* ids))))

  (defun dbg-indent (id indent format-string &rest args)
    "Print indented debugging info if (DEBUG ID) has been specified."
    (when (member id *dbg-ids*)
      (fresh-line *debug-io*)
      (dotimes (i indent) (princ " " *debug-io*))
      (apply #'format *debug-io* format-string args)))

```

Defines:

- *dbg-ids*, never used.
- dbg, used in chunk 47.
- dbg-indent, used in chunks 34 and 36.
- debug, used in chunks 29 and 47.
- undebg, used in chunk 47.

Tests

```

49  <GPS Tests 49>≡ (29)
    (define-test complex
      (assert-equal
        (cons '(start)
              (mapcar #'(lambda (step) (list 'executing step))
                      '(look-up-number
                        telephone-shop
                        tell-shop-problem
                        give-shop-money
                        shop-installs-battery
                        drive-son-to-school))))
        (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-equal '((start) (executing drive-son-to-school))
                    (gps '(son-at-home car-works)
                          '(son-at-school)
                          *school-ops*)))

    (define-test money-leftover
      (assert-equal '((start) (executing drive-son-to-school))
                    (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 `*school-ops*` 46.

Package

```
50  <paip.asd 50>≡
    ;;;; 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"
                          :serial t
                          :components
                          ((:file "intro")
                           (:file "gps")))))

    (defpackage #:paip
      (:use #:cl))
    (in-package #:paip)

    Uses use 43.
```


Test Runner

```
51 <bin/runtests 51>≡
    #! /usr/bin/env nix-shell
    #! nix-shell -i sh -p sbcl

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

    sbcl -noinform -non-interactive \
        -userinit 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))"

52 <init.lisp 52>≡
    #-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 30>
<abstract first-name 2>
<Achieve all goals 33>
<Achieve an individual goal 34>
<An operation with preconds, add-list and del-list 31>
<Apply operator to current state 36>
<Auxiliary Functions 37>
<bin/runtests 51>
<Convert an operator to use the executing convention 40>
<Convert existing operators 42>
<Create an operator 41>
<Debugging usage 47>
<Decide if an operator is appropriate for a goal 35>
<Define a list of operations 46>
<Drive son to school 45>
<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 49>
<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 52>
<Is a condition an executing form? 38>
<Is the argument a list that starts with a given atom? 39>
<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 50⟩
⟨power 18⟩
⟨Print debugging information 48⟩
⟨Rex Morgan MD 15⟩
⟨Solve a goal from a state using a list of operators 32⟩
⟨square 22⟩
⟨src/gps.lisp 29⟩
⟨src/intro.lisp 7⟩
⟨suffixes 13⟩
⟨Test if an element is equal to a member of a list 44⟩
⟨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⟩
⟨Use a list of of operators 43⟩

Index

dbg-ids: [48](#)
ops: [30](#), [32](#), [34](#), [43](#)
school-ops: [42](#), [46](#), [49](#)
suffixes: [10](#), [13](#)
titles: [1](#), [4](#)
achieve: [32](#), [33](#), [34](#)
achieve-all: [32](#), [33](#), [36](#)
apply-op: [34](#), [36](#)
appropriate-p: [34](#), [35](#)
convert-op: [40](#), [41](#), [42](#)
count-atoms: [24](#), [27](#)
dbg: [47](#), [48](#)
dbg-indent: [34](#), [36](#), [48](#)
debug: [29](#), [47](#), [48](#)
executing-p: [38](#), [40](#)
find-all: [28](#), [34](#)
GPS: [29](#), [32](#)
last-name: [9](#), [11](#), [15](#), [16](#)
member-equal: [34](#), [35](#), [36](#), [44](#)
op: [31](#), [34](#), [35](#), [36](#), [40](#), [41](#)
power: [18](#), [20](#), [21](#), [23](#)
square: [20](#), [22](#)
starts-with: [38](#), [39](#)
undebug: [47](#), [48](#)
use: [7](#), [29](#), [43](#), [50](#)

Bibliography