ERIC BAILEY

PAIP EXERCISES

Contents

Introduction to Common Lisp

Using Functions

1 ⟨*titles* 1⟩≡ (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 ⟨*abstract first-name* 2⟩≡

```
(⟨function first-name(name): 3⟩
  (⟨if the first element of name is a title 4⟩
    ⟨then return the first-name of the rest of the name 5⟩
    ⟨else return the first element of the name 6⟩))
```

3 ⟨*function first-name(name):* 3⟩≡ (2)

```
defun first-name (name)
 "Select the first name from a name represented as a list."
```

4 ⟨*if the first element of name is a title* 4⟩≡ (2)

```
if (member (first name) *titles*)
```

Uses `*titles*` 1.

5 ⟨*then return the first-name of the rest of the name* 5⟩≡ (2)

```
  (first-name (rest name))
```

6 ⟨*else return the first element of the name* 6⟩≡ (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⟩
```
Uses `use` 44.

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 ⟨*else return the last element of the name* 12⟩≡ (9)
```
(first (last name))
```

13 ⟨*suffixes* 13⟩≡ (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 ⟨*Exercise 1.1 tests* 14⟩≡ (8)
```
(define-test test-last-name
  ⟨Rex Morgan MD 15⟩
  ⟨Morton Downey, Jr 16⟩)
```

15 ⟨*Rex Morgan MD* 15⟩≡ (14)
```
(assert-equal 'Morgan (last-name '(Rex Morgan MD)))
```
Uses `last-name` 9.

Assert that the `last-name` of *Rex Morgan MD* is *Morgan*.

16 ⟨*Morton Downey, Jr* 16⟩≡ (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 `(power 3 2)` $= 3^2 = 9$.

17 ⟨*Exercise 1.2* 17⟩≡ (7)
```
⟨square 22⟩

⟨power 18⟩

⟨Exercise 1.2 tests 23⟩
```

18 ⟨*power* 18⟩≡ (17)
```
(defun power (x n)
  "Raise x to the power of n."
  (cond ⟨if n is zero return 1 19⟩
        ⟨if n is even return x to the power of n over two, squared 20⟩
        ⟨otherwise return x times x to the power of n minus one 21⟩))
```
Defines:
`power`, used in chunks 20, 21, and 23.

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

19 ⟨*if n is zero return 1* 19⟩≡ (18)
```
((zerop n) 1)
```

$x^0 = 1$

20 ⟨*if n is even return x to the power of n over two, squared* 20⟩≡ (18)
```
((evenp n) (square (power x (/ n 2))))
```
Uses `power` 18 and `square` 22.

21 ⟨*otherwise return x times x to the power of n minus one* 21⟩≡ (18)
```
(t (* x (power x (- n 1))))
```
Uses `power` 18.

22 ⟨*square* 22⟩≡ (17)
```
(defun square (x) (expt x 2))
```
Defines:
`square`, used in chunk 20.

$\text{square}(x) = x^2$

23 ⟨*Exercise 1.2 tests* 23⟩≡ (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 ⟨*Exercise 1.3* 24⟩≡ (7)
```
(defun count-atoms (exp)
  "Return the total number of non-nil atoms in the expression."
  (cond ⟨if exp is nil there are no atoms 25⟩
        ⟨if exp is an atom there is only one 26⟩
        ⟨otherwise add the count of the atoms in the first and rest of exp 27⟩))
```
Defines:
`count-atoms`, used in chunk 27.

25 ⟨*if exp is nil there are no atoms* 25⟩≡ (24)
```
((null exp) 0)
```

26 ⟨*if exp is an atom there is only one* 26⟩≡ (24)
```
((atom exp) 1)
```

27 ⟨*otherwise add the count of the atoms in the first and rest of exp* 27⟩≡ (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 35.

GPS: The General Problem Solver

29 ⟨*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 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 35⟩

⟨Achieve all goals 34⟩

⟨Decide if an operator is appropriate for a goal 36⟩

⟨Apply operator to current state 37⟩

⟨Auxiliary Functions 38⟩

⟨Define a list of operations 47⟩

⟨Convert existing operators 43⟩

⟨Print debugging information 48⟩

⟨GPS Tests 50⟩
```

Uses `debug` 48, `GPS` 33, and `use` 44.

30 ⟨*The current state: a list of conditions* 30⟩≡

```
(defvar *state* nil "The current state: a list of conditions.")
```

Defines:
  `*state*`, never used.

31 ⟨*A list of available operators* 31⟩≡ (29)

```
(defvar *ops* nil "A list of available operators.")
```

Defines:
ops, used in chunks 33, 35, and 44.

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))
```

Uses op 42.

33 ⟨*Solve a goal from a state using a list of operators* 33⟩≡ (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* 31, achieve 35, and achieve-all 34.

34 ⟨*Achieve all goals* 34⟩≡ (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 33 and 37.
Uses achieve 35.

35 ⟨*Achieve an individual goal* 35⟩≡ (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 33 and 34.
Uses *ops* 31, apply-op 37, appropriate-p 36, dbg-indent 48, find-all 28,
member-equal 45, and op 42.

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-equal goal (op-add-list op)))
```

Defines:
  `appropriate-p`, used in chunk 35.
Uses `member-equal` 45 and `op` 42.

37 ⟨*Apply operator to current state* 37⟩≡ (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 35.
Uses `achieve-all` 34, `dbg-indent` 48, `member-equal` 45, and `op` 42.

Auxiliary Functions

38 ⟨*Auxiliary Functions* 38⟩≡ (29)
  ⟨*Is a condition an executing form?* 39⟩

  ⟨*Is the argument a list that starts with a given atom?* 40⟩

  ⟨*Convert an operator to use the executing convention* 41⟩

  ⟨*Create an operator* 42⟩

  ⟨*Use a list of of operators* 44⟩

  ⟨*Test if an element is equal to a member of a list* 45⟩

39 ⟨*Is a condition an executing form?* 39⟩≡ (38)

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

Defines:
  `executing-p`, used in chunk 41.
Uses `starts-with` 40.

40 ⟨*Is the argument a list that starts with a given atom?* 40⟩≡ (38)

```
(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 39.

41 ⟨*Convert an operator to use the executing convention* 41⟩≡ (38)
```
(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 42 and 43.
Uses `executing-p` 39 and `op` 42.

42 ⟨*Create an operator* 42⟩≡ (38)
```
(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 32, 35–37, and 41.
Uses `convert-op` 41.

43 ⟨*Convert existing operators* 43⟩≡ (29)
```
(mapc #'convert-op *school-ops*)
```
Uses `convert-op` 41.

44 ⟨*Use a list of of operators* 44⟩≡ (38)
```
(defun use (oplist)
  "Use oplist as the default list of operators."
  (length (setf *ops* oplist)))
```
Defines:
`use`, used in chunks 7, 29, and 51.
Uses `*ops*` 31.

45 ⟨*Test if an element is equal to a member of a list* 45⟩≡ (38)
```
(defun member-equal (item list)
  (member item list :test #'equal))
```
Defines:
`member-equal`, used in chunks 35–37.

Nursery School Example

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

46 ⟨*Drive son to school* 46⟩≡ (47)
```
(make-op :action 'drive-son-to-school
         :preconds '(son-at-home car-works)
         :add-list '(son-at-school)
         :del-list '(son-at-home))
```

47 ⟨*Define a list of operations* 47⟩≡ (29)

```
(defparameter *school-ops*
  (list
   ⟨Drive son to school 46⟩
   (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))))
```

Debugging

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

;; Turn on debugging
(debug :gps)

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

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 undebug (&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`, never used.
- `dbg-indent`, used in chunks 35 and 37.
- `debug`, used in chunk 29.
- `undebug`, never used.

Tests

49 ⟨*Assert that a given problem is solvable 49*⟩≡

```
  (defmacro assert-solved (form)
    `(assert-equal 'solved ,form))
```

Defines:
- `assert-solved`, never used.

50 ⟨*GPS Tests* 50⟩≡ (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*)))
```

Package

51 ⟨*paip.asd* 51⟩≡

```
;;;; 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)
```

Uses use 44.

Test Runner

52 ⟨*runtests* 52⟩≡

```
#! /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))"
```

53 ⟨*init.lisp* 53⟩≡

```
#-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

Index