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⟩
```

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) $\text{square}(x) = x^2$
```
(defun square (x) (expt x 2))
```
Defines:
square, used in chunk 20.

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)))
```

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⟩

⟨*Decide if an operator is appropriate for a goal* 35⟩

⟨*Apply operator to current state* 36⟩

⟨*Define a list of operations* 38⟩

⟨*Assert that a given problem is solvable* 39⟩

⟨*GPS Tests* 40⟩

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, 34, and 36.

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 (every #'achieve goals) 'solved))
```

Defines:
GPS, used in chunk 29.
Uses *ops* 31, *state* 30, and achieve 34.

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 36.
Uses *ops* 31, *state* 30, apply-op 36, appropriate-p 35, and find-all 28.

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

Defines:
appropriate-p, used in chunk 34.

36 ⟨*Apply operator to current state* 36⟩≡ (29)

```
(defun apply-op (op)
  "Print a message and update *state* if op is applicable."
  (when (every #'achieve (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 34.

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

37 ⟨*Drive son to school* 37⟩≡ (38)

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

38 ⟨*Define a list of operations* 38⟩≡ (29)

```
(defparameter *school-ops*
  (list
   ⟨Drive son to school 37⟩
   (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

39 ⟨*Assert that a given problem is solvable* 39⟩≡ (29)

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

Defines:
`assert-solved`, used in chunk 40.

40 ⟨*GPS Tests* 40⟩≡ (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` 39.

Package

41 ⟨*paip.asd* 41⟩≡

```
;;;; 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

42 ⟨*runtests* 42⟩≡

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

43 ⟨*init.lisp* 43⟩≡

```
#-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

⟨*runtests* 42⟩
⟨*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

Bibliography