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 ⟨*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 46.

## *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) 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.

### *Higher-Order Functions*

28 ⟨*mappend* 28⟩≡ (31 77)
```
(defun mappend (fn the-list)
  "Apply fn to each element of list and append the results."
  (apply #'append (mapcar fn the-list)))
```
Defines:
`mappend`, used in chunks 51 and 80.

# *A Simple Lisp Program*

29 ⟨*random-elt* 29⟩≡ (77)

```
(defun random-elt (choices)
  "Choose an element from a list at random."
  (elt choices (random (length choices))))
```

Defines:
`random-elt`, used in chunk 78.

# *Overview of Lisp*

30 ⟨*find-all* 30⟩≡ (31)

```
(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 36.

# *GPS: The General Problem Solver*

31 ⟨*src/gps.lisp* 31⟩≡

```
(in-package #:paip)
(defpackage #:paip.gps
  (:use #:cl #:lisp-unit)
  (:shadow #:debug)
  (:export #:GPS))
(in-package #:paip.gps)
```

⟨*find-all* 30⟩

⟨*mappend* 28⟩

⟨*A list of available operators* 32⟩

⟨*An operation with preconds, add-list and del-list* 33⟩

⟨*Solve a goal from a state using a list of operators* 34⟩

⟨*Achieve an individual goal* 36⟩

⟨*Achieve all goals* 35⟩

⟨*Decide if an operator is appropriate for a goal* 37⟩

⟨*Apply operator to current state* 38⟩

⟨*Auxiliary Functions* 39⟩

⟨*Nursery School Example* 49⟩

⟨*Monkey and Bananas Example* 50⟩

⟨*The Maze Searching Domain* 51⟩

⟨*Maze Tests* 52⟩

⟨*Convert existing operators* 45⟩

⟨*The Blocks World Domain* 53⟩

⟨*Print debugging information* 57⟩

⟨*GPS Tests* 59⟩

Uses debug 57, GPS 34, and use 46.

32 ⟨*A list of available operators* 32⟩≡ (31)

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

Defines:
*ops*, used in chunks 34, 36, and 46.

33 ⟨*An operation with preconds, add-list and del-list* 33⟩≡ (31)

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

Uses op 44.

34 ⟨*Solve a goal from a state using a list of operators* 34⟩≡ (31)

```
(defun GPS (state goals &optional (*ops* *ops*))
  "General Problem Solver: from state, achieve goals using *ops*."
  (remove-if-not #'action-p
                 (achieve-all (cons '(start) state) goals nil)))
```

Defines:
GPS, used in chunks 31 and 51.
Uses *ops* 32, achieve 36, achieve-all 35, and action-p 41.

35 ⟨*Achieve all goals* 35⟩≡ (31)

```
(defun achieve-all (state goals goal-stack)
  "Achieve each goal, trying several orderings."
  (some #'(lambda (goals) (achieve-each state goals goal-stack))
        (orderings goals)))

(defun achieve-each (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)))

(defun orderings (lst)
  (if (> (length lst) 1)
      (list lst (reverse lst))
      (list lst)))
```

Defines:
achieve-all, used in chunks 34 and 38.
achieve-each, never used.
orderings, never used.
Uses achieve 36.

36 ⟨*Achieve an individual goal* 36⟩≡ (31)

```
(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))
                 (appropriate-ops goal state)))))

(defun appropriate-ops (goal state)
  "Return a list of appropriate operators,
  sorted by the number of unfulfilled preconditions."
  (sort (copy-list (find-all goal *ops* :test #'appropriate-p)) #'<
        :key #'(lambda (op)
                 (count-if #'(lambda (precond)
                               (not (member-equal precond state)))
                           (op-preconds op)))))
```

Defines:
`achieve`, used in chunks 34 and 35.
`appropriate-ops`, never used.
Uses `*ops*` 32, `apply-op` 38, `appropriate-p` 37, `dbg-indent` 57, `find-all` 30, `member-equal` 47, and `op` 44.

37 ⟨*Decide if an operator is appropriate for a goal* 37⟩≡ (31)

```
(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 36.
Uses `member-equal` 47 and `op` 44.

38 ⟨*Apply operator to current state* 38⟩≡ (31)

```
(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 36.
Uses `achieve-all` 35, `dbg-indent` 57, `member-equal` 47, and `op` 44.

## *Auxiliary Functions*

39 ⟨*Auxiliary Functions* 39⟩≡ (31)
⟨*Is a condition an executing form?* 40⟩

⟨*Is x an action?* 41⟩

⟨*Is the argument a list that starts with a given atom?* 42⟩

⟨*Convert an operator to use the executing convention* 43⟩

⟨*Create an operator* 44⟩

⟨*Use a list of of operators* 46⟩

⟨*Test if an element is equal to a member of a list* 47⟩

40 ⟨*Is a condition an executing form?* 40⟩≡ (39)
```
(defun executing-p (x)
  "Is x of the form: (executing ...) ?"
  (starts-with x 'executing))
```
Defines:
`executing-p`, used in chunks 41 and 43.
Uses `starts-with` 42.

41 ⟨*Is x an action?* 41⟩≡ (39)
```
(defun action-p (x)
  "Is x something that is (start) or (executing ...)?"
  (or (equal x '(start)) (executing-p x)))
```
Defines:
`action-p`, used in chunk 34.
Uses `executing-p` 40.

42 ⟨*Is the argument a list that starts with a given atom?* 42⟩≡ (39 77)
```
(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 chunks 40 and 70.

43 ⟨*Convert an operator to use the executing convention* 43⟩≡ (39)
```
(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 44 and 45.
Uses `executing-p` 40 and `op` 44.

44 ⟨*Create an operator* 44⟩≡ (39)

```
(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 33, 36–38, 43, 50, 51, and 53.
Uses convert-op 43.

45 ⟨*Convert existing operators* 45⟩≡ (31)

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

Uses *school-ops* 49 and convert-op 43.

46 ⟨*Use a list of of operators* 46⟩≡ (39)

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

Defines:
use, used in chunks 7, 31, 52, 54, 60, 83, and 84.
Uses *ops* 32.

47 ⟨*Test if an element is equal to a member of a list* 47⟩≡ (39)

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

Defines:
member-equal, used in chunks 36–38.

## *Nursery School Example*

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

48 ⟨*Drive son to school* 48⟩≡ (49)

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

49 ⟨*Nursery School Example* 49⟩≡ (31)

```
(defparameter *school-ops*
  (list
   ⟨Drive son to school 48⟩
   (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 45 and 59.

### *Monkey and Bananas*

50 ⟨*Monkey and Bananas Example* 50⟩≡ (31)

```
(defparameter *banana-ops*
  (list
   (op 'climb-on-chair
       :preconds '(chair-at-middle-room at-middle-room on-floor)
       :add-list '(at-bananas on-chair)
       :del-list '(at-middle-room on-floor))
   (op 'push-chair-from-door-to-middle-room
       :preconds '(chair-at-door at-door)
       :add-list '(chair-at-middle-room at-middle-room)
       :del-list '(chair-at-door at-door))
   (op 'walk-from-door-to-middle-room
       :preconds '(at-door on-floor)
       :add-list '(at-middle-room)
       :del-list '(at-door))
   (op 'grasp-bananas
       :preconds '(at-bananas empty-handed)
       :add-list '(has-bananas)
       :del-list '(empty-handed))
   (op 'drop-ball
       :preconds '(has-ball)
       :add-list '(empty-handed)
       :del-list '(has-ball))
   (op 'eat-bananas
       :preconds '(has-bananas)
       :add-list '(empty-handed not-hungry)
       :del-list '(has-bananas hungry))))
```

Uses op 44.

## *The Maze Searching Domain*

51 ⟨*The Maze Searching Domain* 51⟩≡ (31)

```
(defun make-maze-ops (pair)
  "Make maze ops in both directions."
  (list (make-maze-op (first pair) (second pair))
        (make-maze-op (second pair) (first pair))))

(defun make-maze-op (here there)
  "Make an operator to move between two places."
  (op `(move from ,here to ,there)
      :preconds `((at ,here))
      :add-list `((at ,there))
      :del-list `((at ,here))))

(defparameter *maze-ops*
  (mappend #'make-maze-ops
    '((1 2) (2 3) (3 4) (4 9) (9 14) (9 8) (8 7) (7 12) (12 13)
      (12 11) (11 6) (11 16) (16 17) (17 22) (21 22) (22 23)
      (23 18) (23 24) (24 19) (19 20) (20 15) (15 10) (10 5) (20 25))))

(defun find-path (start end)
  "Search a maze for a path from start to end."
  (let ((results (GPS `((at ,start)) `((at ,end)))))
    (unless (null results)
      (cons start (mapcar #'destination
                          (remove '(start) results
                                  :test #'equal))))))

(defun destination (action)
  "Find the Y in (executing (move from X to Y))."
  (fifth (second action)))
```

Defines:
`destination`, never used.
`find-path`, used in chunk 52.
`make-maze-op`, never used.
`make-maze-ops`, never used.
Uses `GPS` 34, `mappend` 28, and `op` 44.

*Tests*

52 ⟨*Maze Tests* 52⟩≡ (31)

```
(define-test maze
  (use *maze-ops*)
  (assert-equal '(1 2 3 4 9 8 7 12 11 16 17 22 23 24 19 20 25)
                (find-path 1 25)))

(define-test go-nowhere
  (use *maze-ops*)
  (assert-equal '(1) (find-path 1 1)))

(define-test maze-reverse
  (use *maze-ops*)
  (assert-equal (find-path 1 25) (reverse (find-path 25 1))))
```

Uses `find-path` 51 and `use` 46.

The moral is that when a programmer uses puns—saying what's convenient instead of what's really happening–there's bound to be trouble.

## *The Blocks World Domain*

53 ⟨*The Blocks World Domain* 53⟩≡ (31)

```
(defun make-block-ops (blocks)
  (let ((ops nil))
    (dolist (a blocks)
      (dolist (b blocks)
        (unless (equal a b)
          (dolist (c blocks)
            (unless (or (equal c a)
                        (equal c b))
              (push (move-op a b c) ops)))
          (push (move-op a 'table b) ops)
          (push (move-op a b 'table) ops))))
    ops))

(defun move-op (a b c)
  "Make an operator to move A from B to C."
  (op `(move ,a from ,b to ,c)
      :preconds `((space on ,a) (space on ,c) (,a on ,b))
      :add-list (move-ons a b c)
      :del-list (move-ons a c b)))

(defun move-ons (a b c)
  (if (eq b 'table)
      `((,a on ,c))
      `((,a on ,c) (space on ,b))))
```

⟨*Blocks World Tests* 54⟩

Defines:
`make-block-ops`, used in chunk 54.
`move-ons`, never used.
`move-op`, never used.
Uses `op` 44.

54 ⟨*Blocks World Tests* 54⟩≡ (53)

```
(define-test simplest-blocks-problem
  (use (make-block-ops '(a b)))
  (assert-equal '((start) (executing (move a from table to b)))
                (gps '((a on table) (b on table) (space on a) (space on b)
                       (space on table))
                     '((a on b) (b on table)))))

(define-test slighty-more-complex-blocks
  (use (make-block-ops '(a b)))
  (assert-equal '((start)
                  (executing (move a from b to table))
                  (executing (move b from table to a)))
                (gps '((a on b) (b on table) (space on a) (space on table))
                     '((b on a)))))

(define-test blocks-goals-order-insignificant
  (let ((ops (make-block-ops '(a b c))))
    (let ((state '((a on b) (b on c) (c on table)
                   (space on a) (space on table))))
      (assert-equal '((start)
                      (executing (move a from b to table))
                      (executing (move b from c to a))
                      (executing (move c from table to b)))
                    (gps state '((b on a) (c on b)) ops))
      (assert-equal '((start)
                      (executing (move a from b to table))
                      (executing (move b from c to a))
                      (executing (move c from table to b)))
                    (gps state '((c on b) (b on a)) ops)))))

(define-test blocks-ops-ordered-intelligently
  (let ((ops (make-block-ops '(a b c))))
    (let ((state '((c on a) (a on table) (b on table)
                   (space on c) (space on b) (space on table))))
      (assert-equal '((start)
                      (executing (move c from a to table))
                      (executing (move a from table to b)))
                    (gps state '((c on table) (a on b)) ops)))
    (let ((state '((a on b) (b on c) (c on table)
                   (space on a) (space on table))))
      (assert-equal '((start)
                      (executing (move a from b to table))
                      (executing (move b from c to a))
                      (executing (move c from table to b)))
                    (gps state '((b on a) (c on b)) ops))
      (assert-equal '((start)
                      (executing (move a from b to table))
                      (executing (move b from c to a))
                      (executing (move c from table to b)))
                    (gps state '((c on b) (b on a)) ops)))))
```

⟨*Blocks: The Sussman Anomaly* 55⟩

Uses `make-block-ops` 53 and `use` 46.

### *The Sussman Anomaly*

N.B. These results are undesirable and will be addressed in chapter 6.

55 ⟨*Blocks: The Sussman Anomaly* 55⟩≡ (54)

```
(define-test blocks-the-sussman-anomaly
  (let ((start '((c on a) (a on table) (b on table)
                 (space on c) (space on b) (space on table))))
    (assert-nil (gps start '((a on b) (b on c))))
    (assert-nil (gps start '((b on c) (a on b))))))
```

### *Debugging*

56 ⟨*Debugging usage* 56⟩≡

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

;; Turn on debugging
(debug :gps)

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

Uses `dbg` 57, `debug` 57, and `undebug` 57.

57 ⟨*Print debugging information* 57⟩≡ (31)

```
(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*)
    (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*)
    (format *debug-io* "~&~V@T~?" (* 2 indent) format-string args)))
```

Defines:
*dbg-ids*, never used.
dbg, used in chunk 56.
dbg-indent, used in chunks 36 and 38.
debug, used in chunks 31 and 56.
undebug, used in chunk 56.

## *Exercises*

### *Exercise 4.2*

58 ⟨*permutations* 58⟩≡

```
(defun permutations (xs)
  (if (endp (cdr xs))
      (list xs)
      (loop for x in xs
            append (loop for ys in (permutations (remove x xs :count 1
                                                         :test  #'eq))
                         collect (cons x ys)))))
```

Defines:
permutations, never used.

## *Tests*

59 ⟨*GPS Tests* 59⟩≡ (31)

```
(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*` 49.

# ELIZA: *Dialog with a Machine*

60 ⟨*src/eliza.lisp* 60⟩≡

```
(in-package #:paip)
(defpackage #:paip.eliza
  (:use #:cl #:lisp-unit))
(in-package #:paip.eliza)
```

⟨*ELIZA: Constants* 72⟩

⟨*ELIZA: Top-Level Function* 76⟩

⟨*ELIZA: Special Variables* 75⟩

⟨*ELIZA: Data Types* 74⟩

⟨*ELIZA: Functions* 77⟩

Uses use 46.

ELIZA, one of the more well-known AI programs of the 1960s, simulates a psychotherapist, by way of a REPL.

## *Pattern Matching*

61 ⟨*ELIZA: Pattern Matching* 61⟩≡ (77)

⟨*pat-match* 63⟩

⟨*match-variable* 62⟩

⟨*segment-match* 69⟩

⟨*segment-pattern-p* 70⟩

⟨*variable-p* 71⟩

ELIZA makes heavy use of pattern matching, which is at once versatile and limited.

*Matching*

62 ⟨*match-variable* 62⟩≡ (61)

```
(defun match-variable (var input bindings)
  "Does VAR match input? Uses (or updates) and returns bindings."
  (let ((binding (get-binding var bindings)))
    (cond ((not binding) (extend-bindings var input bindings))
          ((equal input (binding-val binding)) bindings)
          (t fail))))
```

Defines:
`match-variable`, used in chunks 65 and 69.
Uses `binding-val` 73, `extend-bindings` 73, `fail` 72, and `get-binding` 73.

Verify `var` is bound to `input` in `bindings`. If bound to another value, `fail`. If unbound, `extend-bindings`.

63 ⟨*pat-match* 63⟩≡ (61)

```
(defun pat-match (pattern input &optional (bindings no-bindings))
  "Match pattern against input in the context of the bindings."
  (cond ⟨Fail if the binding list is fail 64⟩
        ⟨Match a variable 65⟩
        ⟨If pattern equals input, return bindings 66⟩
        ⟨Match a segment 67⟩
        ⟨Call pat-match recursively 68⟩
        (t fail)))
```

Defines:
`pat-match`, used in chunks 68, 69, 72, and 78.
Uses `fail` 72 and `no-bindings` 72.

64 ⟨*Fail if the binding list is fail* 64⟩≡ (63)

```
((eq bindings fail) fail)
```

Uses `fail` 72.

If the binding list is `fail`, then the match fails, because some previous match must have failed.

65 ⟨*Match a variable* 65⟩≡ (63)

```
((variable-p pattern) (match-variable pattern input bindings))
```

Uses `match-variable` 62 and `variable-p` 71.

If the `pattern` is a single variable, return the result of `match-variable`; either `bindings` (possibly extended) or `fail`.

66 ⟨*If pattern equals input, return bindings* 66⟩≡ (63)

```
((eql pattern input) bindings)
```

If `pattern` equals `input`, return `bindings` as is.

67 ⟨*Match a segment* 67⟩≡ (63)

```
((segment-pattern-p pattern) (segment-match pattern input bindings))
```

Uses `segment-match` 69 and `segment-pattern-p` 70.

When both `pattern` and `input` are lists and the `(car pattern)` is a segment variable, match the variable to the initial part of the `input` and attempt to match `(cdr pattern)` to the rest.

68 ⟨*Call* `pat-match` *recursively* 68⟩≡ (63)

```
((and (consp pattern) (consp input))
 (pat-match (rest pattern) (rest input)
            (pat-match (first pattern) (first input)
                       bindings)))
```

Uses `pat-match` 63.

69 ⟨*segment-match* 69⟩≡ (61)

```
(defun segment-match (pattern input bindings &optional (start 0))
  "Match the segment pattern ((?* var) . pat) against input."
  (let ((var (second (first pattern)))
        (pat (rest pattern)))
    (if (null pat)
        (match-variable var input bindings)
        (let ((pos (position (first pat) input
                             :start start :test #'equal)))
          (if (null pos)
              fail
              (let ((b2 (pat-match pat (subseq input pos) bindings)))
                (if (eq b2 fail)
                    (segment-match pattern input bindings (+ pos 1))
                    (match-variable var (subseq input 0 pos) b2))))))))
```

Defines:
  `segment-match`, used in chunk 67.
Uses `fail` 72, `match-variable` 62, and `pat-match` 63.

## *Predicates*

70 ⟨*segment-pattern-p* 70⟩≡ (61)

```
(defun segment-pattern-p (pattern)
  "Is this a segment matching pattern: ((?* var) . pat)"
  (and (consp pattern)
       (starts-with (first pattern) '?*)))
```

Defines:
  `segment-pattern-p`, used in chunk 67.
Uses `starts-with` 42.

71 ⟨*variable-p* 71⟩≡ (61)

```
(defun variable-p (x)
  "Is x a variable (a symbol beginning with '?')?"
  (and (symbolp x)
       (equal (char (symbol-name x) 0) #\?)))
```

Defines:
  `variable-p`, used in chunk 65.

## *Constants*

72 ⟨*ELIZA: Constants* 72⟩≡ (60)

```
(defconstant fail nil
  "Indicates pat-match failure")

(defconstant no-bindings '((t . t))
  "Indicates pat-match success, with no variables.")
```

Defines:
  `fail`, used in chunks 62–64, 69, and 78.
  `no-bindings`, used in chunks 63 and 73.
Uses `pat-match` 63.

73 ⟨*ELIZA: Binding Functions* 73⟩≡ (77)

```
(defun get-binding (var bindings)
  "Find a (variable . value) pair in a binding list."
  (assoc var bindings))

(defun binding-val (binding)
  "Get the value part of a single binding."
  (cdr binding))

(defun lookup (var bindings)
  "Get the value part (for var) from a binding list."
  (binding-val (get-binding var bindings)))

(defun extend-bindings (var val bindings)
  "Add a (var . value) pair to a binding list."
  (cons (cons var val)
        (if (eq bindings no-bindings)
            nil
            bindings)))
```

Defines:
`binding-val`, used in chunk 62.
`extend-bindings`, used in chunk 62.
`get-binding`, used in chunk 62.
`lookup`, never used.
Uses `no-bindings` 72.

## *Rules*

74 ⟨*ELIZA: Data Types* 74⟩≡ (60)

```
(defun rule-pattern (rule) (first rule))
(defun rule-responses (rule) (rest rule))
```

Defines:
`rule-pattern`, used in chunk 78.
`rule-responses`, used in chunk 78.

75 ⟨*ELIZA: Special Variables* 75⟩≡ (60)

```
(defparameter *eliza-rules*
  '((((?* ?x) hello (?* ?y))
     (How do you do.  Please state your problem.))
    (((?* ?x) I want (?* ?y))
     (What would it mean if you got ?y)
     (Why do you want ?y) (Suppose you got ?y soon))
    (((?* ?x) if (?* ?y))
     (Do you really think its likely that ?y) (Do you wish that ?y)
     (What do you think about ?y) (Really-- if ?y))
    (((?* ?x) no (?* ?y))
     (Why not?) (You are being a bit negative)
     (Are you saying "NO" just to be negative?))
    (((?* ?x) I was (?* ?y))
     (Were you really?) (Perhaps I already knew you were ?y)
     (Why do you tell me you were ?y now?))
    (((?* ?x) I feel (?* ?y))
     (Do you often feel ?y ?))
    (((?* ?x) I felt (?* ?y))
     (What other feelings do you have?))))
```

## ELIZA *Proper*

76 ⟨*ELIZA: Top-Level Function* 76⟩≡ (60)

```
(defun eliza ()
  "Respond to user input using pattern matching rules."
  (loop
    (print 'eliza>)
    (write (flatten (use-eliza-rules (read))) :pretty t)))
```

Defines:
  `eliza`, used in chunk 84.
Uses `flatten` 80 and `use-eliza-rules` 78.

77 ⟨*ELIZA: Functions* 77⟩≡ (60)

⟨*use-eliza-rules* 78⟩

⟨*switch-viewpoint* 79⟩

⟨*ELIZA: Pattern Matching* 61⟩

⟨*ELIZA: Binding Functions* 73⟩

⟨*flatten* 80⟩

⟨*random-elt* 29⟩

⟨*mappend* 28⟩

⟨*Is the argument a list that starts with a given atom?* 42⟩

78 ⟨*use-eliza-rules* 78⟩≡ (77)

```
(defun use-eliza-rules (input)
  "Find some rule with which to transform the input."
  (some #'(lambda (rule)
            (let ((result (pat-match (rule-pattern rule) input)))
              (unless (eq result fail)
                (sublis (switch-viewpoint result)
                        (random-elt (rule-responses rule))))))
        *eliza-rules*))
```

Defines:
`use-eliza-rules`, used in chunk 76.
Uses `fail` 72, `pat-match` 63, `random-elt` 29, `rule-pattern` 74, `rule-responses` 74, and `switch-viewpoint` 79.

79 ⟨*switch-viewpoint* 79⟩≡ (77)

```
(defun switch-viewpoint (words)
  "Change I to you and vice versa, and so on."
  (sublis '((I . you) (you . I) (me . you) (am . are))
          words))
```

Defines:
`switch-viewpoint`, used in chunk 78.

80 ⟨*flatten* 80⟩≡ (77)

```
(defun flatten (the-list)
  "Append together elements (or lists) in the list."
  (mappend #'mklist the-list))

(defun mklist (x)
  "Return x if it is a list, otherwise (x)."
  (if (listp x)
      x
      (list x)))
```

Defines:
`flatten`, used in chunk 76.
`mklist`, never used.
Uses `mappend` 28.

# *Build Software Tools*

## *An Interactive Interpreter Tool*

```
(defun program ()
  (loop
    (print prompt)
    (print (transform (read)))))
```

81 ⟨*interactive-interpreter* 81⟩≡ (83)

```
(defun interactive-interpreter (prompt transformer)
  "(`prompt' for and) read an expression, `transform' it and print the result."
  (loop
    (handler-case
        (progn
          (if (stringp prompt)
              (print prompt)
              (funcall prompt))
          (print (funcall transformer (read))))
      (error (condition)
        (format t "~&;; Error ~a ignored. Back to top level."
                condition)))))
```

⟨*prompt-generator* 82⟩

Defines:
`interactive-interpreter`, never used.

82 ⟨*prompt-generator* 82⟩≡ (81)

```
(defun prompt-generator (&optional (num 0) (ctl-string "[~d] "))
  "Return a function that prints prompts like [1], [2], etc."
  #'(lambda () (format t ctl-string (incf num))))
```

Defines:
`prompt-generator`, never used.

## *Package*

83 ⟨*src/tools.lisp* 83⟩≡

```
(in-package #:paip)
(defpackage #:paip.tools
  (:use #:cl #:lisp-unit))
(in-package #:paip.tools)

⟨interactive-interpreter 81⟩
```

Uses use 46.

# *Package*

84 ⟨*paip.asd* 84⟩≡

```
;;;; 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")
                 (:file "eliza")
                 (:file "tools")))))

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

Uses `eliza` 76 and `use` 46.

# *Test Runner*

85 ⟨*bin/runtests* 85⟩≡

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

86 ⟨*init.lisp* 86⟩≡

```
#-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*

⟨*the last element of a name is a suffix* 10⟩
⟨*The Maze Searching Domain* 51⟩
⟨*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* 46⟩
⟨*use-eliza-rules* 78⟩
⟨*variable-p* 71⟩

# *Index*

*Bibliography*