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⟩

;; Exercise 1.4
⟨Exercise 1.4 28⟩

;; Exercise 1.5
⟨Exercise 1.5 33⟩
```
Uses `use` 72.

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))
```

Define some well-known suffixes.

13 ⟨*suffixes* 13⟩≡ (8)

```
(defparameter *suffixes*
  '(MD Jr. Sr. III)
  "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⟩))
```

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

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

19 ⟨*if n is zero return 1* 19⟩≡ (18) $x^0 = 1$

```
((zerop n) 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 &optional (if-null 1))
  "Return the total number of atoms in the expression,
   counting nil as an atom only in non-tail position."
  (cond ⟨if exp is nil there are if-null 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 if-null atoms* 25⟩≡ (24)

```
((null exp) if-null)
```

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) 1)
      (count-atoms (rest exp) 0)))
```

Uses `count-atoms` 24.

Exercise 1.4

28 ⟨*Exercise 1.4* 28⟩≡ (7)

```
(defun count-anywhere (item tree)
  "Count the occurrences of item anywhere within tree."
  (cond ⟨if item is equal to tree, there is one occurrence 29⟩
        ⟨if tree is an atom, there are no occurrences 30⟩
        ⟨otherwise, add the occurrence within first the first and rest of tree 31⟩))

⟨Exercise 1.4 tests 32⟩
```

Defines:
`count-anywhere`, used in chunks 31 and 32.

29 ⟨*if item is equal to tree, there is one occurrence* 29⟩≡ (28)

```
((eql item tree) 1)
```

30 ⟨*if tree is an atom, there are no occurrences* 30⟩≡ (28)

```
((atom tree) 0)
```

31 ⟨*otherwise, add the occurrence within first the first and rest of tree* 31⟩≡ (28)

```
(t (+ (count-anywhere item (first tree))
      (count-anywhere item (rest tree))))
```

Uses `count-anywhere` 28.

32 ⟨*Exercise 1.4 tests* 32⟩≡ (28)

```
(define-test test-count-anywhere
  (assert-equal 3 (count-anywhere 'a '(a ((a) b) a))))
```

Uses `count-anywhere` 28.

Exercise 1.5

33 ⟨*Exercise 1.5* 33⟩≡ (7)

```
(defun dot-product (lhs rhs)
  "Compute the mathematical dot product of two vectors."
  ⟨multiply elements of the vectors pairwise and sum the results 34⟩)

⟨Exercise 1.5 tests 35⟩
```

Defines:
`dot-product`, used in chunk 35.

34 ⟨*multiply elements of the vectors pairwise and sum the results* 34⟩≡ (33)

```
(apply #'+ (mapcar #'* lhs rhs))
```

35 ⟨*Exercise 1.5 tests* 35⟩≡ (33)

```
(define-test test-dot-product
  (assert-equal 110 (dot-product '(10 20) '(3 4))))
```

Uses `dot-product` 33.

Higher-Order Functions

36 ⟨*mappend* 36⟩≡ (38 57 103)

```
(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 46, 53, 55, 77, and 106.

A Simple Lisp Program

37 ⟨*src/simple.lisp* 37⟩≡ 38▷

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

Uses `use` 72.

A Straightforward Solution

38 ⟨*src/simple.lisp* 37⟩+≡ ◁37 41▷

```
(defun sentence ()
  (append (noun-phrase) (verb-phrase)))

(defun verb-phrase ()
  (append (Verb) (noun-phrase)))

(defun Article ()
  (one-of '(the a)))

(defun Noun ()
  (one-of '(man ball woman table)))

(defun Verb ()
  (one-of '(hit took saw liked)))

⟨one-of 39⟩

⟨random-elt 40⟩

⟨mappend 36⟩
```

Defines:
- `Article`, used in chunks 41, 42, and 51.
- `Noun`, used in chunks 41, 42, and 51.
- `sentence`, used in chunks 42, 47, 50–52, and 54.
- `Verb`, used in chunks 42 and 51.
- `verb-phrase`, used in chunks 42 and 51.

Uses `noun-phrase` 41 and `one-of` 39.

39 ⟨*one-of* 39⟩≡ (38)

```
(defun one-of (set)
  "Pick one element of set, and make a list of it."
  (list (random-elt set)))
```

Defines:
`one-of`, used in chunks 38 and 41.
Uses `random-elt` 40.

40 ⟨*random-elt* 40⟩≡ (38 103)

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

Defines:
`random-elt`, used in chunks 39, 41, 47, 50, 52, and 104.

41 ⟨*src/simple.lisp* 37⟩+≡ ◁38 42▷

```
(defun Adj* ()
  (if (= (random 2) 0)
      nil
    (append (Adj) (Adj*))))

(defun PP* ()
  (if (random-elt '(t nil))
      (append (PP) (PP*))
    nil))

(defun noun-phrase ()
  (append (Article) (Adj*) (Noun) (PP*)))

(defun PP ()
  (append (Prep) (noun-phrase)))

(defun Adj ()
  (one-of '(big little blue green adiabatic)))

(defun Prep ()
  (one-of '(to in by with on)))
```

Defines:
`Adj`, used in chunk 51.
`Adj*`, used in chunk 51.
`noun-phrase`, used in chunks 38, 42, and 51.
`PP`, used in chunk 51.
`PP*`, used in chunk 51.
`Prep`, used in chunk 51.
Uses `Article` 38, `Noun` 38, `one-of` 39, and `random-elt` 40.

A Rule-Based Solution

42 ⟨*src/simple.lisp* 37⟩+≡ ◁41 43▷

```
(defparameter *simple-grammar*
  '((sentence -> (noun-phrase verb-phrase))
    (noun-phrase -> (Article Noun))
    (verb-phrase -> (Verb noun-phrase))
    (Article -> the a)
    (Noun -> man ball woman table)
    (Verb -> hit took saw liked))
  "A grammar for a trivial subset of English.")

(defvar *grammar* *simple-grammar*
  "The grammar used by generate. Initially, this is *simple-grammar*,
   but we can switch to other grammars.")
```

Defines:
`*grammar*`, used in chunks 43 and 51.
Uses `Article` 38, `Noun` 38, `noun-phrase` 41, `sentence` 38, `Verb` 38, and `verb-phrase` 38.

43 ⟨*src/simple.lisp* 37⟩+≡ ◁42 47▷

```
(defun rule-lhs (rule)
  "The left-hand side of a rule."
  (first rule))

(defun rule-rhs (rule)
  "The right-hand side of a rule."
  (rest (rest rule)))

(defun rewrites (category)
  "Return a list of the possible rewrites for this category."
  (rule-rhs (assoc category *grammar*)))
```

Defines:
`rewrites`, used in chunks 47, 49, 50, 52, and 53.
`rule-lhs`, never used.
`rule-rhs`, never used.
Uses `*grammar*` 42.

Exercise 2.1

I prefer treating definitions as immutable, so I'm not a fan of `setf`. I'll do it my way instead, without `cond`.

Because I can't resist leaving a yak unshaved, define `if-let`, too.

44 ⟨*if-let* 44⟩≡ (47)

```
(defmacro if-let ((name test) then &optional else)
  `(let ((,name ,test))
     (if ,name ,then ,else)))
```

Defines:
`if-let`, used in chunks 47, 52, and 53.

45 ⟨*phrase is a list* 45⟩≡ (47 50 52 53)

```
(listp phrase)
```

46 ⟨*generate a phrase* 46⟩≡ (47 50)

```
(mappend #'generate phrase)
```

Uses `generate` 47 and `mappend` 36.

47 ⟨*src/simple.lisp* 37⟩+≡ ◁43 50▷

```
⟨if-let 44⟩

(defun generate (phrase)
  "Generate a random sentence or phrase."
  (if ⟨phrase is a list 45⟩
      ⟨generate a phrase 46⟩
    (if-let (choices (rewrites phrase))
        (generate (random-elt choices))
      (list phrase))))
```

Defines:
`generate`, used in chunks 46 and 50.
Uses `if-let` 44, `random-elt` 40, `rewrites` 43, and `sentence` 38.

Exercise 2.2

48 ⟨*phrase is nonterminal* 48⟩≡ (50)

```
(non-terminal-p phrase)
```

Uses `non-terminal-p` 49.

49 ⟨*non-terminal-p* 49⟩≡ (50)

```
(defun non-terminal-p (category)
  "Return true iff this is a category in the grammar."
  (not (null (rewrites category))))
```

Defines:
`non-terminal-p`, used in chunk 48.
Uses `rewrites` 43.

50 ⟨*src/simple.lisp* 37⟩+≡ ◁47 51▷

```
(defun generate-alt (phrase)
  "Generate a random sentence or phrase,
   differentiating between terminal and nonterminal symbols."
  (cond (⟨phrase is a list 45⟩
         ⟨generate a phrase 46⟩)
        (⟨phrase is nonterminal 48⟩
         (generate (random-elt (rewrites phrase))))
        (t (list phrase))))

⟨non-terminal-p 49⟩
```

Defines:
`generate-alt`, never used.
Uses `generate` 47, `random-elt` 40, `rewrites` 43, and `sentence` 38.

Changing the Grammar without Changing the Program

51 ⟨*src/simple.lisp* 37⟩+≡ ◁50 52▷

```
(defparameter *bigger-grammar*
  '((sentence -> (noun-phrase verb-phrase))
    (noun-phrase -> (Article Adj* Noun PP*) (Name) (Pronoun))
    (verb-phrase -> (Verb noun-phrase PP*))
    (PP* -> () (PP PP*))
    (Adj* -> () (Adj Adj*))
    (PP -> (Prep noun-phrase))
    (Prep -> to in by with on)
    (Adj -> big little blue green adiabatic)
    (Article -> the a)
    (Name -> Pat Kim Lee Terry Robin)
    (Noun -> man ball woman table)
    (Verb -> hit took saw liked)
    (Pronoun -> he she they it these those that)))

;; (setf *grammar* *bigger-grammar*)
```

Uses `*grammar*` 42, `Adj` 41, `Adj*` 41, `Article` 38, `Noun` 38, `noun-phrase` 41, `PP` 41, `PP*` 41, `Prep` 41, `sentence` 38, `Verb` 38, and `verb-phrase` 38.

Using the Same Data for Several Programs

52 ⟨*src/simple.lisp* 37⟩+≡ ◁51 53▷

```
(defun generate-tree (phrase)
  "Generate a random sentence or phrase,
   with a complete parse tree."
  (if ⟨phrase is a list 45⟩
      (mapcar #'generate-tree phrase)
    (if-let (choices (rewrites phrase))
        (cons phrase
              (generate-tree (random-elt (rewrites phrase))))
      (list phrase))))
```

Defines:
`generate-tree`, never used.

Uses `if-let` 44, `random-elt` 40, `rewrites` 43, and `sentence` 38.

53 ⟨*src/simple.lisp* 37⟩+≡ ◁52 54▷

```
(defun generate-all (phrase)
  (cond ((null phrase) (list nil))
        (⟨phrase is a list 45⟩
         (combine-all (generate-all (first phrase))
                      (generate-all (rest phrase))))
        (t (if-let (choices (rewrites phrase))
               (mappend #'generate-all choices)
             (list (list phrase))))))

(defun combine-all (xs ys)
  "Return a list of lists formed by appending a y to an x."
  (cross-product #'append xs ys))
```

Defines:
`combine-all`, never used.
`generate-all`, never used.
Uses `cross-product` 55, `if-let` 44, `mappend` 36, and `rewrites` 43.

Exercises

Exercise 2.3

54 ⟨*src/simple.lisp* 37⟩+≡ ◁53 55▷

```
(defparameter *grammática-simple*
  '((sentence -> (frase-sustantiva frase-verbal))
    (frase-sustantiva -> (Artículo Sustantivo))
    (frase-verbal -> (Verbo frase-sustantiva))
    (Artículo -> el la un una)
    (Sustantivo -> hombre pelota mujer mesa)
    (Verbo -> pegó tomó gustó))
  "Una grammática simple para un subconjunto trivial del español.")
```

Uses `sentence` 38.

Exercise 2.4

55 ⟨*src/simple.lisp* 37⟩+≡ ◁54

```
(defun cross-product (func xlist ylist)
  "Return a list of all (func x y) values."
  (mappend #'(lambda (y)
               (mapcar #'(lambda (x) (funcall func x y))
                       xlist))
           ylist))
;; (setf (fdefinition 'zip-with) #'cross-product)

(define-test test-cross-product
  (assert-equal '(11 12 13
                  21 22 23
                  31 32 33)
                (cross-product #'+ '(1 2 3) '(10 20 30))))
```

Defines:
`cross-product`, used in chunk 53.
Uses `mappend` 36.

Overview of Lisp

56 ⟨*find-all* 56⟩≡ (57)

```
(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 62.

GPS: The General Problem Solver

57 ⟨*src/gps.lisp* 57⟩≡

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

⟨*find-all* 56⟩

⟨*mappend* 36⟩

⟨*A list of available operators* 58⟩

⟨*An operation with preconds, add-list and del-list* 59⟩

⟨*Solve a goal from a state using a list of operators* 60⟩

⟨*Achieve an individual goal* 62⟩

⟨*Achieve all goals* 61⟩

⟨*Decide if an operator is appropriate for a goal* 63⟩

⟨*Apply operator to current state* 64⟩

⟨*Auxiliary Functions* 65⟩

⟨*Nursery School Example* 75⟩

⟨*Monkey and Bananas Example* 76⟩

⟨*The Maze Searching Domain* 77⟩

⟨*Maze Tests* 78⟩

⟨*Convert existing operators* 71⟩

⟨*The Blocks World Domain* 79⟩

⟨*Print debugging information* 83⟩

⟨*GPS Tests* 85⟩

Uses `debug` 83, `GPS` 60, and `use` 72.

58 ⟨*A list of available operators* 58⟩≡ (57)

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

Defines:
`*ops*`, used in chunks 60, 62, and 72.

59 ⟨*An operation with preconds, add-list and del-list* 59⟩≡ (57)

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

Uses `op` 70.

60 ⟨*Solve a goal from a state using a list of operators* 60⟩≡ (57)

```
(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 57 and 77.

Uses `*ops*` 58, `achieve` 62, `achieve-all` 61, and `action-p` 67.

61 ⟨*Achieve all goals* 61⟩≡ (57)

```
(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 60 and 64.
`achieve-each`, never used.
`orderings`, never used.

Uses `achieve` 62.

62 ⟨*Achieve an individual goal* 62⟩≡ (57)

```
(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 60 and 61.
`appropriate-ops`, never used.
Uses `*ops*` 58, `apply-op` 64, `appropriate-p` 63, `dbg-indent` 83, `find-all` 56, `member-equal` 73, and `op` 70.

63 ⟨*Decide if an operator is appropriate for a goal* 63⟩≡ (57)

```
(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 62.
Uses `member-equal` 73 and `op` 70.

64 ⟨*Apply operator to current state* 64⟩≡ (57)

```
(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 62.
Uses `achieve-all` 61, `dbg-indent` 83, `member-equal` 73, and `op` 70.

Auxiliary Functions

65 ⟨*Auxiliary Functions* 65⟩≡ (57)

```
⟨Is a condition an executing form? 66⟩

⟨Is x an action? 67⟩

⟨Is the argument a list that starts with a given atom? 68⟩

⟨Convert an operator to use the executing convention 69⟩

⟨Create an operator 70⟩

⟨Use a list of of operators 72⟩

⟨Test if an element is equal to a member of a list 73⟩
```

66 ⟨*Is a condition an executing form?* 66⟩≡ (65)

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

Defines:
`executing-p`, used in chunks 67 and 69.
Uses `starts-with` 68.

67 ⟨*Is x an action?* 67⟩≡ (65)

```
(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 60.
Uses `executing-p` 66.

68 ⟨*Is the argument a list that starts with a given atom?* 68⟩≡ (65 103)

```
(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 66 and 96.

69 ⟨*Convert an operator to use the executing convention* 69⟩≡ (65)

```
(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 70 and 71.
Uses `executing-p` 66 and `op` 70.

70 ⟨*Create an operator* 70⟩≡ (65)

```
(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 59, 62–64, 69, 76, 77, and 79.
Uses `convert-op` 69.

71 ⟨*Convert existing operators* 71⟩≡ (57)

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

Uses `*school-ops*` 75 and `convert-op` 69.

72 ⟨*Use a list of of operators* 72⟩≡ (65)

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

Defines:
`use`, used in chunks 7, 37, 57, 78, 80, 86, 109, and 110.
Uses `*ops*` 58.

73 ⟨*Test if an element is equal to a member of a list* 73⟩≡ (65)

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

Defines:
`member-equal`, used in chunks 62–64.

Nursery School Example

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

74 ⟨*Drive son to school* 74⟩≡ (75)

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

75 ⟨*Nursery School Example* 75⟩≡ (57)

```
(defparameter *school-ops*
  (list
   ⟨Drive son to school 74⟩
   (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 71 and 85.

Monkey and Bananas

76 ⟨*Monkey and Bananas Example* 76⟩≡ (57)

```
(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` 70.

The Maze Searching Domain

77 ⟨*The Maze Searching Domain* 77⟩≡ (57)

```
(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 78.
`make-maze-op`, never used.
`make-maze-ops`, never used.

Uses `GPS` 60, `mappend` 36, and `op` 70.

Tests

78 ⟨*Maze Tests* 78⟩≡ (57)

```
(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` 77 and `use` 72.

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

79 ⟨*The Blocks World Domain* 79⟩≡ (57)

```
(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* 80⟩

Defines:
`make-block-ops`, used in chunk 80.
`move-ons`, never used.
`move-op`, never used.

Uses `op` 70.

80 ⟨*Blocks World Tests* 80⟩≡ (79)

```
(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* 81⟩
Uses `make-block-ops` 79 and `use` 72.

The Sussman Anomaly

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

81 ⟨*Blocks: The Sussman Anomaly* 81⟩≡ (80)

```
(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

82 ⟨*Debugging usage* 82⟩≡

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

;; Turn on debugging
(debug :gps)

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

Uses `dbg` 83, `debug` 83, and `undebug` 83.

83 ⟨*Print debugging information 83*⟩≡ (57)

```
(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 82.
`dbg-indent`, used in chunks 62 and 64.
`debug`, used in chunks 57 and 82.
`undebug`, used in chunk 82.

Exercises

Exercise 4.2

84 ⟨*permutations 84*⟩≡

```
(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

85 ⟨*GPS Tests* 85⟩≡ (57)

```
(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*` 75.

Eliza: *Dialog with a Machine*

86 ⟨*src/eliza.lisp* 86⟩≡

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

⟨ELIZA: Constants 98⟩

⟨ELIZA: Top-Level Function 102⟩

⟨ELIZA: Special Variables 101⟩

⟨ELIZA: Data Types 100⟩

⟨ELIZA: Functions 103⟩
```

Uses `eliza` 102 and `use` 72.

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

Pattern Matching

87 ⟨*ELIZA: Pattern Matching* 87⟩≡ (103)

```
⟨pat-match 89⟩

⟨match-variable 88⟩

⟨segment-match 95⟩

⟨segment-pattern-p 96⟩

⟨variable-p 97⟩
```

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

Matching

88 ⟨*match-variable* 88⟩≡ (87)

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

```
(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 91 and 95.
Uses `binding-val` 99, `extend-bindings` 99, `fail` 98, and `get-binding` 99.

89 ⟨*pat-match* 89⟩≡ (87)

```
(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 90⟩
        ⟨Match a variable 91⟩
        ⟨If pattern equals input, return bindings 92⟩
        ⟨Match a segment 93⟩
        ⟨Call pat-match recursively 94⟩
        (t fail)))
```

Defines:
`pat-match`, used in chunks 94, 95, 98, and 104.
Uses `fail` 98 and `no-bindings` 98.

90 ⟨*Fail if the binding list is `fail`* 90⟩≡ (89)

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

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

Uses `fail` 98.

91 ⟨*Match a variable* 91⟩≡ (89)

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

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

Uses `match-variable` 88 and `variable-p` 97.

92 ⟨*If `pattern` equals `input`, return `bindings`* 92⟩≡ (89)

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

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

93 ⟨*Match a segment* 93⟩≡ (89)

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.

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

Uses `segment-match` 95 and `segment-pattern-p` 96.

94 ⟨*Call `pat-match` recursively* 94⟩≡ (89)

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

Uses `pat-match` 89.

95 ⟨*segment-match* 95⟩≡ (87)

```
(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 93.
Uses `fail` 98, `match-variable` 88, and `pat-match` 89.

Predicates

96 ⟨*segment-pattern-p* 96⟩≡ (87)

```
(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 93.
Uses `starts-with` 68.

97 ⟨*variable-p* 97⟩≡ (87)

```
(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 91.

Constants

98 ⟨*ELIZA: Constants* 98⟩≡ (86)

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

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

Defines:
`fail`, used in chunks 88–90, 95, and 104.
`no-bindings`, used in chunks 89 and 99.
Uses `pat-match` 89.

99 ⟨*ELIZA: Binding Functions* 99⟩≡ (103)

```
(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 88.
`extend-bindings`, used in chunk 88.
`get-binding`, used in chunk 88.
`lookup`, never used.
Uses `no-bindings` 98.

Rules

100 ⟨*ELIZA: Data Types* 100⟩≡ (86)

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

Defines:
`rule-pattern`, used in chunk 104.
`rule-responses`, used in chunk 104.

101 ⟨*ELIZA: Special Variables* 101⟩≡ (86)

```
(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*

102 ⟨*ELIZA: Top-Level Function* 102⟩≡ (86)

```
(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 chunks 86 and 110.
Uses `flatten` 106 and `use-eliza-rules` 104.

103 ⟨*ELIZA: Functions* 103⟩≡ (86)

⟨*use-eliza-rules* 104⟩

⟨*switch-viewpoint* 105⟩

⟨*ELIZA: Pattern Matching* 87⟩

⟨*ELIZA: Binding Functions* 99⟩

⟨*flatten* 106⟩

⟨*random-elt* 40⟩

⟨*mappend* 36⟩

⟨*Is the argument a list that starts with a given atom?* 68⟩

104 ⟨*use-eliza-rules* 104⟩≡ (103)

```
(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 102.
Uses `fail` 98, `pat-match` 89, `random-elt` 40, `rule-pattern` 100, `rule-responses` 100, and `switch-viewpoint` 105.

105 ⟨*switch-viewpoint* 105⟩≡ (103)

```
(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 104.

106 ⟨*flatten* 106⟩≡ (103)

```
(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 102.
`mklist`, never used.
Uses `mappend` 36.

Build Software Tools

An Interactive Interpreter Tool

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

107 ⟨*interactive-interpreter* 107⟩≡ (109)

```
  (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 108⟩
```

Defines:
`interactive-interpreter`, never used.

108 ⟨*prompt-generator* 108⟩≡ (107)

```
  (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

109 ⟨*src/tools.lisp* 109⟩≡

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

⟨interactive-interpreter 107⟩
```

Uses `use` 72.

Package

110 ⟨*paip.asd* 110⟩≡

```
;;;; paip.asd

(asdf:defsystem #:paip
  :description "Paradigms of Artificial Intelligence Programming exercises"
  :author "Eric Bailey <eric@ericb.me>"
  :license "BSD-3"
  :depends-on (#:lisp-unit)
  :serial t
  :components ((:module "src"
                :serial t
                :components
                ((:file "intro")
                 (:file "simple")
                 (:file "gps")
                 (:file "eliza")
                 (:file "tools")))))

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

Uses `eliza` 102 and `use` 72.

Test Runner

111 ⟨*bin/runtests* 111⟩≡

```
#! /usr/bin/env bash

# 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))"
```

112 ⟨*init.lisp* 112⟩≡

```
#-quicklisp
(let ((quicklisp-init "quicklisp/setup.lisp"))
  (when (probe-file quicklisp-init)
    (load quicklisp-init)))

(push (concatenate 'string (sb-posix:getcwd) "/")
      asdf:*central-registry*)

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

Chunks

Index

Bibliography