

ERIC BAILEY

PAIP EXERCISES

Contents

<i>Introduction to Common Lisp</i>	5
<i>Overview of Lisp</i>	9
<i>GPS: The General Problem Solver</i>	11
<i>Package</i>	25
<i>Test Runner</i>	27
<i>Chunks</i>	29
<i>Index</i>	31
<i>Bibliography</i>	33

Introduction to Common Lisp

Using Functions

- 1 $\langle \text{titles } 1 \rangle \equiv$ (7)
 (defparameter *titles*
 '(Mr Mrs Miss Ms Sir Madam Dr Admiral Major General)
 "A list of titles that can appear at the start of a name.")
Defines:
 titles, used in chunk 4.
- 2 $\langle \text{abstract first-name } 2 \rangle \equiv$
 ($\langle \text{function first-name(name): } 3 \rangle$
 ($\langle \text{if the first element of name is a title } 4 \rangle$
 $\langle \text{then return the first-name of the rest of the name } 5 \rangle$
 $\langle \text{else return the first element of the name } 6 \rangle$))
- 3 $\langle \text{function first-name(name): } 3 \rangle \equiv$ (2)
 defun first-name (name)
 "Select the first name from a name represented as a list."
- 4 $\langle \text{if the first element of name is a title } 4 \rangle \equiv$ (2)
 if (member (first name) *titles*)
Uses *titles* 1.
- 5 $\langle \text{then return the first-name of the rest of the name } 5 \rangle \equiv$ (2)
 (first-name (rest name))
- 6 $\langle \text{else return the first element of the name } 6 \rangle \equiv$ (2)
 (first name)

Exercises

```

7  <src/intro.lisp 7>≡
    (in-package #:paip)
    (defpackage #:paip.intro
      (:use #:cl #:lisp-unit))
    (in-package #:paip.intro)

    <titles 1>

    ;; Exercise 1.1
    <Exercise 1.1 8>

    ;; Exercise 1.2
    <Exercise 1.2 17>

    ;; Exercise 1.3
    <Exercise 1.3 24>
    Uses use 45.

```

Exercise 1.1

Define a version of **last-name** that handles “Rex Morgan MD,” “Morton Downey, Jr.,” and whatever other cases you can think of.

```

8  <Exercise 1.1 8>≡ (7)
    <suffixes 13>

    <last-name 9>

    <Exercise 1.1 tests 14>

9  <last-name 9>≡ (8)
    (defun last-name (name)
      "Select the last name from a name represented as a list."
      (if <the last element of a name is a suffix 10>
          <then return the last-name of all but the last element of the name 11>
          <else return the last element of the name 12>))

```

Defines:

last-name, used in chunks 11, 15, and 16.

First, we check to see if the last element of the **name** is a suffix, i.e. whether it’s a member of ***suffixes***.

```

10 <the last element of a name is a suffix 10>≡ (9)
    (member (first (last name)) *suffixes*)
    Uses *suffixes* 13.

```

If it is, then drop it from the **name** and return the **last-name** of the result.

```

11 <then return the last-name of all but the last element of the name 11>≡ (9)
    (last-name (butlast name))
    Uses last-name 9.

```

Otherwise, it's the last name, so return it.

12 $\langle \text{else return the last element of the name } 12 \rangle \equiv$ (9)
`(first (last name))`

13 $\langle \text{suffixes } 13 \rangle \equiv$ (8)
`(defparameter *suffixes*`
`'(MD Jr)`
`"A list of suffixes that can appear at the end of a name.")`

Defines:

`*suffixes*`, used in chunk 10.

14 $\langle \text{Exercise 1.1 tests } 14 \rangle \equiv$ (8)
`(define-test test-last-name`
 `$\langle \text{Rex Morgan MD } 15 \rangle$`
 `$\langle \text{Morton Downey, Jr } 16 \rangle$)`

15 $\langle \text{Rex Morgan MD } 15 \rangle \equiv$ (14) Assert that the `last-name` of Rex Morgan MD is Morgan.
`(assert-equal 'Morgan (last-name '(Rex Morgan MD)))`
 Uses last-name 9.

16 $\langle \text{Morton Downey, Jr } 16 \rangle \equiv$ (14)
`(assert-equal 'Downey (last-name '(Morton Downey Jr)))`
 Uses last-name 9.

Exercise 1.2

Write a function to exponentiate, or raise a number to an integer power.
 For example $(\text{power } 3 \ 2) = 3^2 = 9$.

17 $\langle \text{Exercise 1.2 } 17 \rangle \equiv$ (7)
 $\langle \text{square } 22 \rangle$
 $\langle \text{power } 18 \rangle$
 $\langle \text{Exercise 1.2 tests } 23 \rangle$

18 $\langle \text{power } 18 \rangle \equiv$ (17)
`(defun power (x n)`
`"Raise x to the power of n."`
`(cond $\langle \text{if n is zero return } 1 \ 19 \rangle$`
 `$\langle \text{if n is even return x to the power of n over two, squared } 20 \rangle$`
 `$\langle \text{otherwise return x times x to the power of n minus one } 21 \rangle$))`

Defines:

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

19 $\langle \text{if n is zero return } 1 \ 19 \rangle \equiv$ (18) $x^0 = 1$
`((zerop n) 1)`

20 $\langle \text{if n is even return x to the power of n over two, squared } 20 \rangle \equiv$ (18)
`((evenp n) (square (power x (/ n 2))))`
 Uses power 18 and square 22.

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

21 $\langle \text{otherwise return } x \text{ times } x \text{ to the power of } n \text{ minus one } 21 \rangle \equiv$ (18)
 (t (* x (power x (- n 1))))

Uses power 18.

22 $\langle \text{square } 22 \rangle \equiv$ (17) $\text{square}(x) = x^2$
 (defun square (x) (expt x 2))

Defines:

square, used in chunk 20.

23 $\langle \text{Exercise 1.2 tests } 23 \rangle \equiv$ (17)
 (define-test test-power
 (assert-equal 9 (power 3 2)))

Uses power 18.

Exercise 1.3

Write a function that counts the number of atoms in an expression.

For example: (count-atoms '(a (b) c)) = 3. Notice that there is something of an ambiguity in this: should (a nil c) count as three atoms, or as two, because it is equivalent to (a () c)?

24 $\langle \text{Exercise 1.3 } 24 \rangle \equiv$ (7)
 (defun count-atoms (exp)
 "Return the total number of non-nil atoms in the expression."
 (cond $\langle \text{if exp is nil there are no atoms } 25 \rangle$
 $\langle \text{if exp is an atom there is only one } 26 \rangle$
 $\langle \text{otherwise add the count of the atoms in the first and rest of exp } 27 \rangle$))

Defines:

count-atoms, used in chunk 27.

25 $\langle \text{if exp is nil there are no atoms } 25 \rangle \equiv$ (24)
 ((null exp) 0)

26 $\langle \text{if exp is an atom there is only one } 26 \rangle \equiv$ (24)
 ((atom exp) 1)

27 $\langle \text{otherwise add the count of the atoms in the first and rest of exp } 27 \rangle \equiv$ (24)
 (t (+ (count-atoms (first exp))
 (count-atoms (rest exp))))

Uses count-atoms 24.

Higher-Order Functions

28 $\langle \text{mappend } 28 \rangle \equiv$ (30)
 (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 chunk 50.

Overview of Lisp

29 $\langle find-all\ 29 \rangle \equiv$ (30)

```
(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

```
30  <src/gps.lisp 30>≡
    (in-package #:paip)
    (defpackage #:paip.gps
      (:use #:cl #:lisp-unit)
      (:shadow #:debug)
      (:export #:GPS))
    (in-package #:paip.gps)

    <find-all 29>

    <mappend 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>

    <Nursery School Example 48>

    <Monkey and Bananas Example 49>

    <The Maze Searching Domain 50>

    <Maze Tests 51>

    <Convert existing operators 44>

    <The Blocks World Domain 52>
```

⟨Print debugging information 56⟩

⟨GPS Tests 57⟩

Uses debug 56, GPS 33, and use 45.

31 *⟨A list of available operators 31⟩*≡ (30)
 (defvar *ops* nil "A list of available operators.")

Defines:

ops, used in chunks 33, 35, and 45.

32 *⟨An operation with preconds, add-list and del-list 32⟩*≡ (30)
 (defstruct op
 "An operation"
 (action nil)
 (preconds nil)
 (add-list nil)
 (del-list nil))

Uses op 43.

33 *⟨Solve a goal from a state using a list of operators 33⟩*≡ (30)
 (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 30 and 50.

Uses *ops* 31, achieve 35, achieve-all 34, and action-p 40.

34 *⟨Achieve all goals 34⟩*≡ (30)
 (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 33 and 37.

achieve-each, never used.

orderings, never used.

Uses achieve 35.

35 *⟨Achieve an individual goal 35⟩*≡ (30)

```
(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 33 and 34.

appropriate-ops, never used.

Uses *ops* 31, apply-op 37, appropriate-p 36, dbg-indent 56, find-all 29,
member-equal 46, and op 43.

36 *⟨Decide if an operator is appropriate for a goal 36⟩*≡ (30)

```
(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 46 and op 43.

37 *⟨Apply operator to current state 37⟩*≡ (30)

```
(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 56, member-equal 46, and op 43.

Auxiliary Functions

38 *⟨Auxiliary Functions 38⟩*≡ (30)
⟨Is a condition an executing form? 39⟩

⟨Is x an action? 40⟩

⟨Is the argument a list that starts with a given atom? 41⟩

⟨Convert an operator to use the executing convention 42⟩

⟨Create an operator 43⟩

⟨Use a list of operators 45⟩

⟨Test if an element is equal to a member of a list 46⟩

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 chunks 40 and 42.

Uses starts-with 41.

40 *⟨Is x an action? 40⟩*≡ (38)
 (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 33.

Uses executing-p 39.

41 *⟨Is the argument a list that starts with a given atom? 41⟩*≡ (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.

42 *⟨Convert an operator to use the executing convention 42⟩*≡ (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 43 and 44.

Uses executing-p 39 and op 43.

43 *⟨Create an operator 43⟩*≡ (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, 42, 49, 50, and 52.

Uses convert-op 42.

44 *⟨Convert existing operators 44⟩*≡ (30)

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

Uses *school-ops* 48 and convert-op 42.

45 *⟨Use a list of operators 45⟩*≡ (38)

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

Defines:

use, used in chunks 7, 30, 51, 53, and 58.

Uses *ops* 31.

46 *⟨Test if an element is equal to a member of a list 46⟩*≡ (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.

47 *⟨Drive son to school 47⟩*≡ (48)

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

48 \langle Nursery School Example 48 $\rangle \equiv$ (30)

```
(defparameter *school-ops*
  (list
     $\langle$ Drive son to school 47 $\rangle$ 
    (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 44 and 57.

Monkey and Bananas

49 $\langle \text{Monkey and Bananas Example 49} \rangle \equiv$ (30)

```

(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 43.

The Maze Searching Domain

```

50 <The Maze Searching Domain 50>≡ (30)
  (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 51.
 make-maze-op, never used.
 make-maze-ops, never used.

Uses GPS 33, mappend 28, and op 43.

Tests

```

51  <Maze Tests 51>≡ (30)
    (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 50 and use 45.

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

```

52 <The Blocks World Domain 52>≡ (30)
  (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 53>

Defines:

make-block-ops, used in chunk 53.

move-ons, never used.

move-op, never used.

Uses op 43.

```

53  <Blocks World Tests 53>≡ (52)
    (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 slightly-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 54⟩

Uses `make-block-ops` 52 and `use` 45.

The Sussman Anomaly

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

```
54 ⟨Blocks: The Sussman Anomaly 54⟩≡ (53)
    (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

```
55 ⟨Debugging usage 55⟩≡
    ;; Example call
    (dbg :gps "The current goal is: ~a" goal)

    ;; Turn on debugging
    (debug :gps)

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

Uses `dbg` 56, `debug` 56, and `undebug` 56.

```

56  <Print debugging information 56>≡                                     (30)
    (defvar *dbg-ids* nil
      "Identifiers used by dbg")

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

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

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

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

```

Defines:

dbg-ids, never used.
 dbg, used in chunk 55.
 dbg-indent, used in chunks 35 and 37.
 debug, used in chunks 30 and 55.
 undebg, used in chunk 55.

Tests

```

57 <GPS Tests 57>≡ (30)
  (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*` 48.

Package

```
58 <paip.asd 58>≡  
    ;;;; paip.asd  
  
    (asdf:defsystem #:paip  
      :description "Paradigms of Artificial Intelligence Programming exercises"  
      :author "Eric Bailey <eric@ericb.me>"  
      ;; TODO :license "Specify license here"  
      :depends-on (#:lisp-unit)  
      :serial t  
      :components ((:module "src"  
                     :serial t  
                     :components  
                       ((:file "intro")  
                        (:file "gps")))))  
  
    (defpackage #:paip  
      (:use #:cl))  
    (in-package #:paip)  
    Uses use 45.
```


Test Runner

```
59 <bin/runtests 59>≡
    #! /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))"

60 <init.lisp 60>≡
    #-quicklisp
    (let ((quicklisp-init (merge-pathnames "quicklisp/setup.lisp"
                                           (user-homedir-pathname))))
        (when (probe-file quicklisp-init)
            (load quicklisp-init)))

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

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


Chunks

<A list of available operators 31>
<abstract first-name 2>
<Achieve all goals 34>
<Achieve an individual goal 35>
<An operation with preconds, add-list and del-list 32>
<Apply operator to current state 37>
<Auxiliary Functions 38>
<bin/runtests 59>
<Blocks World Tests 53>
<Blocks: The Sussman Anomaly 54>
<Convert an operator to use the executing convention 42>
<Convert existing operators 44>
<Create an operator 43>
<Debugging usage 55>
<Decide if an operator is appropriate for a goal 36>
<Drive son to school 47>
<else return the first element of the name 6>
<else return the last element of the name 12>
<Exercise 1.1 8>
<Exercise 1.1 tests 14>
<Exercise 1.2 17>
<Exercise 1.2 tests 23>
<Exercise 1.3 24>
<find-all 29>
<function first-name(name): 3>
<GPS Tests 57>
<if exp is an atom there is only one 26>
<if exp is nil there are no atoms 25>
<if n is even return x to the power of n over two, squared 20>
<if n is zero return 1 19>
<if the first element of name is a title 4>
<init.lisp 60>
<Is a condition an executing form? 39>
<Is the argument a list that starts with a given atom? 41>

<Is *x* an action? 40>
 <last-name 9>
 <mappend 28>
 <Maze Tests 51>
 <Monkey and Bananas Example 49>
 <Morton Downey, Jr 16>
 <Nursery School Example 48>
 <otherwise add the count of the atoms in the first and rest of exp 27>
 <otherwise return *x* times *x* to the power of *n* minus one 21>
 <paip.asd 58>
 <power 18>
 <Print debugging information 56>
 <Rex Morgan MD 15>
 <Solve a goal from a state using a list of operators 33>
 <square 22>
 <src/gps.lisp 30>
 <src/intro.lisp 7>
 <suffixes 13>
 <Test if an element is equal to a member of a list 46>
 <The Blocks World Domain 52>
 <the last element of a name is a suffix 10>
 <The Maze Searching Domain 50>
 <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 45>

Index

dbg-ids: [56](#)
ops: [31](#), [33](#), [35](#), [45](#)
school-ops: [44](#), [48](#), [57](#)
suffixes: [10](#), [13](#)
titles: [1](#), [4](#)
achieve: [33](#), [34](#), [35](#)
achieve-all: [33](#), [34](#), [37](#)
achieve-each: [34](#)
action-p: [33](#), [40](#)
apply-op: [35](#), [37](#)
appropriate-ops: [35](#)
appropriate-p: [35](#), [36](#)
convert-op: [42](#), [43](#), [44](#)
count-atoms: [24](#), [27](#)
dbg: [55](#), [56](#)
dbg-indent: [35](#), [37](#), [56](#)
debug: [30](#), [55](#), [56](#)
destination: [50](#)
executing-p: [39](#), [40](#), [42](#)
find-all: [29](#), [35](#)
find-path: [50](#), [51](#)
GPS: [30](#), [33](#), [50](#)
last-name: [9](#), [11](#), [15](#), [16](#)
make-block-ops: [52](#), [53](#)
make-maze-op: [50](#)
make-maze-ops: [50](#)
mappend: [28](#), [50](#)
member-equal: [35](#), [36](#), [37](#), [46](#)
move-ons: [52](#)
move-op: [52](#)
op: [32](#), [35](#), [36](#), [37](#), [42](#), [43](#), [49](#), [50](#), [52](#)
orderings: [34](#)
power: [18](#), [20](#), [21](#), [23](#)
square: [20](#), [22](#)

starts-with: 39, 41

undebbug: 55, 56

use: 7, 30, 45, 51, 53, 58

Bibliography