

# Structure and Interpretation of Computer Programs (SICP)

## worked examples

Mac Radigan

### Abstract

A collection of worked examples from Gerald Sussman's book Structure and Interpretation of Computer Programs (SICP) [?].

## Contents

<b>1</b>	<b>Building Abstractions with Procedures</b>	<b>6</b>
1.1	The Elements of Programming . . . . .	6
1.1.1	Expressions . . . . .	6
1.1.2	Naming and the Environment . . . . .	7
1.1.3	Evaluating Combinations . . . . .	8
1.1.4	Compound Procedures . . . . .	9
1.1.5	The Substitution Model for Procedure Application . . . . .	10
1.1.6	Conditional Expressions and Predicates . . . . .	13
1.1.7	Example: Square Roots by Newton's Method . . . . .	13
1.1.8	Procedures as Black-Box Abstractions . . . . .	15
1.2	Procedures and the Processes They Generate . . . . .	15
1.2.1	Linear Recursion and Iteration . . . . .	15
1.2.2	Tree Recursion . . . . .	24
1.2.3	Orders of Growth . . . . .	24
1.2.4	Exponentiation . . . . .	24
1.2.5	Greatest Common Divisors . . . . .	28
1.2.6	Example: Testing for Primality . . . . .	28
1.3	Formulating Abstractions with Higher-Order Procedures . . . . .	28

1.3.1	Procedures as Arguments . . . . .	28
1.3.2	Constructing Procedures Using Lambda . . . . .	28
1.3.3	Procedures as General Methods . . . . .	28
1.3.4	Procedures as Returned Values . . . . .	28
<b>2</b>	<b>Building Abstractions with Data</b>	<b>30</b>
2.1	Introduction to Data Abstraction . . . . .	30
2.1.1	Example: Arithmetic Operations for Rational Numbers . . . . .	30
2.1.2	Abstraction Barriers . . . . .	32
2.1.3	What Is Meant by Data? . . . . .	35
2.1.4	Extended Exercise: Interval Arithmetic . . . . .	37
2.2	Hierarchical Data and the Closure Property . . . . .	37
2.2.1	Representing Sequences . . . . .	39
2.2.2	Hierarchical Structures . . . . .	39
2.2.3	Sequences as Conventional Interfaces . . . . .	45
2.2.4	Example: A Picture Language . . . . .	45
2.3	Symbolic Data . . . . .	50
2.3.1	Quotation . . . . .	50
2.3.2	Example: Symbolic Differentiation . . . . .	56
2.3.3	Example: Representing Sets . . . . .	69
2.3.4	Example: Huffman Encoding Trees . . . . .	69
2.4	Multiple Representations for Abstract Data . . . . .	69
2.4.1	Representations for Complex Numbers . . . . .	69
2.4.2	Tagged data . . . . .	69
2.4.3	Data-Directed Programming and Additivity . . . . .	69
2.5	Systems with Generic Operations . . . . .	69
2.5.1	Generic Arithmetic Operations . . . . .	69
2.5.2	Combining Data of Different Types . . . . .	69
2.5.3	Example: Symbolic Algebra . . . . .	69
<b>3</b>	<b>Modularity, Objects, and State</b>	<b>70</b>
3.1	Assignment and Local State . . . . .	70
3.1.1	Local State Variables . . . . .	70
3.1.2	The Benefits of Introducing Assignment . . . . .	74

3.1.3	The Costs of Introducing Assignment . . . . .	74
3.2	The Environment Model of Evaluation . . . . .	74
3.2.1	The Rules for Evaluation . . . . .	74
3.2.2	Applying Simple Procedures . . . . .	74
3.2.3	Frames as the Repository of Local State . . . . .	74
3.2.4	Internal Definitions . . . . .	74
3.3	Modeling with Mutable Data . . . . .	74
3.3.1	Mutable List Structure . . . . .	74
3.3.2	Representing Queues . . . . .	74
3.3.3	Representing Tables . . . . .	74
3.3.4	A Simulator for Digital Circuits . . . . .	74
3.3.5	Propagation of Constraints . . . . .	74
3.4	Concurrency: Time Is of the Essence . . . . .	74
3.4.1	The Nature of Time in Concurrent Systems . . . . .	74
3.4.2	Mechanisms for Controlling Concurrency . . . . .	74
3.5	Streams . . . . .	74
3.5.1	Streams Are Delayed Lists . . . . .	74
3.5.2	Infinite Streams . . . . .	76
3.5.3	Exploiting the Stream Paradigm . . . . .	76
3.5.4	Streams and Delayed Evaluation . . . . .	76
3.5.5	Modularity of Functional Programs and Modularity of Objects . . . . .	76
<b>4</b>	<b>Metalinguistic Abstraction</b>	<b>77</b>
4.1	The Metacircular Evaluator . . . . .	78
4.1.1	The Core of the Evaluator . . . . .	78
4.1.2	Representing Expressions . . . . .	78
4.1.3	Evaluator Data Structures . . . . .	78
4.1.4	Running the Evaluator as a Program . . . . .	78
4.1.5	Data as Programs . . . . .	78
4.1.6	Internal Definitions . . . . .	78
4.1.7	Separating Syntactic Analysis from Execution . . . . .	78
4.2	Variations on a Scheme – Lazy Evaluation . . . . .	78
4.2.1	Normal Order and Applicative Order . . . . .	78

4.2.2	An Interpreter with Lazy Evaluation . . . . .	78
4.2.3	Streams as Lazy Lists . . . . .	78
4.3	Variations on a Scheme – Nondeterministic Computing . . . . .	78
4.3.1	Amb and Search . . . . .	78
4.3.2	Examples of Nondeterministic Programs . . . . .	78
4.3.3	Implementing the Amb Evaluator . . . . .	78
4.4	Logic Programming . . . . .	78
4.4.1	Deductive Information Retrieval . . . . .	78
4.4.2	How the Query System Works . . . . .	85
4.4.3	Is Logic Programming Mathematical Logic? . . . . .	85
4.4.4	Implementing the Query System . . . . .	85
<b>5</b>	<b>Computing with Register Machines</b>	<b>87</b>
5.1	Designing Register Machines . . . . .	87
5.1.1	A Language for Describing Register Machines . . . . .	87
5.1.2	Abstraction in Machine Design . . . . .	87
5.1.3	Subroutines . . . . .	87
5.1.4	Using a Stack to Implement Recursion . . . . .	87
5.1.5	Instruction Summary . . . . .	87
5.2	A Register-Machine Simulator . . . . .	87
5.2.1	The Machine Model . . . . .	87
5.2.2	The Assembler . . . . .	87
5.2.3	Generating Execution Procedures for Instructions . . . . .	87
5.2.4	Monitoring Machine Performance . . . . .	87
5.3	Storage Allocation and Garbage Collection . . . . .	87
5.3.1	Memory as Vectors . . . . .	87
5.3.2	Maintaining the Illusion of Infinite Memory . . . . .	87
5.4	The Explicit-Control Evaluator . . . . .	87
5.4.1	The Core of the Explicit-Control Evaluator . . . . .	87
5.4.2	Sequence Evaluation and Tail Recursion . . . . .	87
5.4.3	Conditionals, Assignments, and Definitions . . . . .	87
5.4.4	Running the Evaluator . . . . .	87
5.5	Compilation . . . . .	87

5.5.1	Structure of the Compiler . . . . .	87
5.5.2	Compiling Expressions . . . . .	87
5.5.3	Compiling Combinations . . . . .	87
5.5.4	Combining Instruction Sequences . . . . .	87
5.5.5	An Example of Compiled Code . . . . .	87
5.5.6	Lexical Addressing . . . . .	87
5.5.7	Interfacing Compiled Code to the Evaluator . . . . .	87
<b>6</b>	<b>Appendix A: Modules</b>	<b>88</b>
6.1	util.scm . . . . .	88
<b>7</b>	<b>Appendix B: Installation Notes</b>	<b>96</b>
7.1	Chicken Scheme . . . . .	96
<b>8</b>	<b>Appendix C: Notation</b>	<b>97</b>
8.1	Membership . . . . .	97
8.2	Symbols . . . . .	97
8.3	Access . . . . .	97
8.4	Equality . . . . .	98
8.5	Logic . . . . .	98
<b>9</b>	<b>Appendix D: Y-Combinator</b>	<b>99</b>
9.1	Introduction . . . . .	99
9.2	Canonical Expression . . . . .	99
9.3	Connonical Form in Scheme . . . . .	99
9.3.1	Strict Scheme (Chicken) . . . . .	99
9.3.2	Using Lazy Evaluation (Racket #lang lazy) . . . . .	100
9.4	Normal Order Y Combinator . . . . .	101
9.4.1	Strict Scheme (Chicken) . . . . .	101
9.4.2	Using Strict Evaluation (Racket) . . . . .	102
9.4.3	Using Lazy Evaluation (Racket #lang lazy) . . . . .	102
9.5	Strict (Applicative-Order) Y Combinator . . . . .	103
9.5.1	Strict Scheme (Chicken) . . . . .	103
9.5.2	Using Strict Evaluation (Racket) . . . . .	105
9.5.3	Using Lazy Evaluation (Racket #lang lazy) . . . . .	106

# 1 Building Abstractions with Procedures

## 1.1 The Elements of Programming

### 1.1.1 Expressions

Exercise 1.1. Below is a sequence of expressions. What is the result printed by the interpreter in response to each expression? Assume that the sequence is to be evaluated in the order in which it is presented.

```
1  #!/usr/bin/csi -s
2  ;; sicp_ch1_e1-1.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  ;; Exercise 1.1. Below is a sequence of expressions. What is the result printed by the interpreter in
9  ;; response to each expression? Assume that the sequence is to be evaluated in the order in which it is
10 ;; presented.
11
12 (prn 10 )
13 (prn (+ 5 3 4) )
14 (prn (- 9 1) )
15 (prn (/ 6 2) )
16 (prn (+ (* 2 4) (- 4 6)) )
17
18 (define a 3)
19 (define b (+ a 1))
20
21 (prn (+ a b (* a b)) )
22 (prn (= a b) )
23
24 (prn (if (and (> b a) (< b (* a b)))
25         b
26         a) )
27
28 (prn (cond ((= a 4) 6)
29         ((= b 4) (+ 6 7 a))
30         (else 25)) )
31
```

```

32 (prn (+ 2 (if (> b a) b a)) )
33
34 (prn (* (cond ((> a b) a)
35            ((< a b) b)
36            (else -1))
37        (+ a 1)) )
38
39 ;; *EOF*

```

```
## ./sisp_ch1_e1-1.scm
```

```

10
12
8
3
6
19
#f
4
16
6
16

```

### 1.1.2 Naming and the Environment

Exercise 1.2. Translate the following expression into prefix form

$$\frac{5 + 1/2 + (2 - (3 - (6 + 1/5)))}{3(6 - 2)(2 - 7)} \quad (1)$$

```

1 #!/usr/bin/csi -s
2 ;; sisp_ch1_e1-2.scm
3 ;; Mac Radigan
4
5 (load "../library/util.scm")
6 (import util)
7
8 ;;; Exercise 1.2. Translate the following expression into prefix form
9 ;;;
10 ;;; 5 + 1/2 + (2 - (3 - (6 + 1/5) ) )
11 ;;; -----

```

```

12  ;;      3 * ( 6 - 2 ) * ( 2 - 7 )
13
14
15  (prn
16    (/
17      (+ 5 1/2 (- 2 (- 3 (+ 6 1/5) ) ) )
18      (* 3 (- 6 2) (- 2 7) )
19    )
20  )
21
22  ;; *EOF*

```

```
## ./sisp_ch1_e1-2.scm
```

```
-0.1783333333333333
```

### 1.1.3 Evaluating Combinations

Exercise 1.3. Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

```

1  #!/usr/bin/csi -s
2  ;; sisp_ch1_e1-3.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  ;; Exercise 1.3. Define a procedure that takes three numbers as arguments and returns the sum of the
9  ;; squares of the two larger numbers.
10
11  ;; suares and sum
12  (define (my-square x) (map (lambda (x) (* x x)) x) )
13  (define (my-sum x) (apply + x) )
14
15  ;; two methods for computing the sum of squares
16  (define (ss-1 x) ((compose my-sum my-square) x))
17  (define (ss-2 x) (apply + (map (lambda (x) (* x x )) x) ) )
18
19  ;; selection N elements from a list

```



```

20 (define (take x N)
21   (if (> N 1)
22     (cons (car x) (take (cdr x) (- N 1)))
23     (list (car x))
24   )
25 )
26
27 ;; selection for top N given operand
28 (define (top x pred? N) (take (sort x pred?) N) )
29
30 ;; sum of squares for top 2 largest elements in list
31 (define (topss-1 x) ((compose ss-2 (lambda (x) (top x > 2)) ) x))
32 (define (topss-2 x) (ss-2 (top x > 2)) )
33
34 ;; test solution
35 (define x '(3 5 2 9 1) )
36
37 (prn (topss-1 x) )
38 (prn (topss-2 x) )
39
40 (assert (= (ss-1 x) (ss-2 x) ) )
41 (assert (= (ss-1 x) (ss-2 x) ) )
42 (assert (= (topss-1 x) (topss-2 x) ) )
43 (assert (= (topss-1 x) (topss-2 x) ) )
44
45 ;; *EOF*

```

```
## ./sisp_ch1_e1-3.scm
```

```
106
```

```
106
```

#### 1.1.4 Compound Procedures

Exercise 1.4. Observe that our model of evaluation allows for combinations whose operators are compound expressions. Use this observation to describe the behavior of the following procedure:

```

(define (a-plus-abs-b a b)
  ((if (< b 0) + -) a b))

```

```

1  #!/usr/bin/csi -s
2  ;; sicp_ch1_e1-4.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  ;; Exercise 1.4. Observe that our model of evaluation allows for combinations whose operators are
9  ;; compound expressions. Use this observation to describe the behavior of the following procedure:
10 ;; (define (a-plus-abs-b a b)
11 ;;   ((if (> b 0) + -) a b))
12
13 (define (a-plus-abs-b a b)
14   ((if (> b 0) + -) a b))
15
16 (prn (a-plus-abs-b 5 +2) )
17 (prn (a-plus-abs-b 5 -2) )
18
19 ;; *EOF*

```

```
## ../sicp_ch1_e1-4.scm
```

```
7
7
```

### 1.1.5 The Substitution Model for Procedure Application

Exercise 1.5. Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative-order evaluation or normal-order evaluation. He defines the following two procedures:

```

(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))

```

Then he evaluates the expression

```
(test 0 (p))
```

What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume that the evaluation rule for the special form `if` is the same whether the interpreter is using normal or applicative

order: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.)

```
1  #!/usr/bin/csi -s
2  ;; sicp_ch1_e1-5.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  ;;; Exercise 1.5. Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with
   ↪  is
9  ;;; using applicative-order evaluation or normal-order evaluation. He defines the following two
10 ;;; procedures:
11 ;;; (define (p) (p))
12 ;;; (define (test x y)
13 ;;;   (if (= x 0)
14 ;;;       0
15 ;;;       y))
16 ;;; Then he evaluates the expression
17 ;;; (test 0 (p))
18 ;;; What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What
19 ;;; behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer.
20 ;;; (Assume that the evaluation rule for the special form if is the same whether the interpreter is using
21 ;;; normal or applicative order: The predicate expression is evaluated first, and the result determines
22 ;;; whether to evaluate the consequent or the alternative expression.)
23
24 (define (p) (p)) ; infinite recursion
25
26 (define (test x y)
27   (if (= x 0)
28       0
29       y))
30
31 (prn(test 0 (p)) ) ; infinite loop
32
33 (prn (p) ) ; infinite loop
34
35 ;; *EOF*
```

Exercise 1.6. Alyssa P. Hacker doesn't see why `if` needs to be provided as a special form. "Why can't I just define it as an ordinary procedure in terms of `cond`?" she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of `if`:

```
(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))
```

Eva demonstrates the program for Alyssa:

```
(new-if (= 2 3) 0 5)
5

(new-if (= 1 1) 0 5)
0
```

Delighted, Alyssa uses `new-if` to rewrite the square-root program:

```
(new-if (= 1 1) 0 5)

(define (sqrt-iter guess x)
  (new-if (good-enough? guess x)
    guess
    (sqrt-iter (improve guess x) x)))
```

What happens when Alyssa attempts to use this to compute square roots? Explain.

Calls to the compound procedure `new-if` are applicatively evaluated, that is evaluated first and then passed as arguments to the procedure. The predicates and clauses to `cond`, and thus results in infinite recursion of unintended clause.

This is not the case with the intrinsic `if` procedure, which performs normal evaluation of the expression.

$$\text{new-if}(\text{predicate}, \text{if-clause}, \text{else-clause}) = \begin{cases} \text{if-clause} & \text{if predicate} \\ \text{then-clause} & \text{otherwise} \end{cases} \quad (2)$$

One possible correction for new-if, here, new-if-im, is to accept quasi-quoted arguments, and then eval them.

$$\text{new-if}_{\text{im}}(\overline{\text{predicate}}, \overline{\text{if-clause}}, \overline{\text{else-clause}}) = \begin{cases} \text{eval}(\text{if-clause}) & \text{if } \text{eval}(\text{predicate}) \\ \text{eval}(\text{then-clause}) & \text{otherwise} \end{cases} \quad (3)$$

where  $\overline{\text{predicate}}$ ,  $\overline{\text{if-clause}}$ , and  $\overline{\text{else-clause}}$  are quasi-quoted expressions.

## 1.1.6 Conditional Expressions and Predicates

### 1.1.7 Example: Square Roots by Newton's Method

Exercise 1.7. The good-enough? test used in computing square roots will not be very effective for finding the square roots of very small numbers. Also, in real computers, arithmetic operations are almost always performed with limited precision. This makes our test inadequate for very large numbers. Explain these statements, with examples showing how the test fails for small and large numbers. An alternative strategy for implementing good-enough? is to watch how guess changes from one iteration to the next and to stop when the change is a very small fraction of the guess. Design a square-root procedure that uses this kind of end test. Does this work better for small and large numbers?

Newton's method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (4)$$

Applied to square root:

$$x_{n+1} = x_n - \frac{x_n^2 - \text{guess}}{2x_n} \quad (5)$$

```

1 |#!/usr/bin/csi -s
2 |;; sicp_ch1_e1-5.scm
3 |;; Mac Radigan
4 |
5 |(load "../library/util.scm")
6 |(import util)

```

```

7
8 ;;; Exercise 1.5. Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with
9 ↪ is
10 ;;; using applicative-order evaluation or normal-order evaluation. He defines the following two
11 ;;; procedures:
12 ;;; (define (p) (p))
13 ;;; (define (test x y)
14 ;;; (if (= x 0)
15 ;;; 0
16 ;;; y))
17 ;;; Then he evaluates the expression
18 ;;; (test 0 (p))
19 ;;; What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What
20 ;;; behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer.
21 ;;; (Assume that the evaluation rule for the special form if is the same whether the interpreter is using
22 ;;; normal or applicative order: The predicate expression is evaluated first, and the result determines
23 ;;; whether to evaluate the consequent or the alternative expression.)
24 (define (p) (p)) ; infinite recursion
25
26 (define (test x y)
27 (if (= x 0)
28 0
29 y))
30
31 ; (prn(test 0 (p)) ) ; infinite loop
32
33 ; (prn (p) ) ; infinite loop
34
35 ;; *EOF*

```

```
## ./sicp_ch1_e1-7.scm
```

### 1.1.8 Procedures as Black-Box Abstractions

## 1.2 Procedures and the Processes They Generate

### 1.2.1 Linear Recursion and Iteration

Exercise 1.9: Each of the following two procedures defines a method for adding two positive integers in terms of the procedures `inc`, which increments its argument by 1, and `dec`, which decrements its argument by 1.

```
1  #!/usr/bin/csi -s
2  ;; sicp_ch1_e1-9.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  ;;; Exercise 1.9: Each of the following two procedures defines a method for adding two positive integers
   ↳ in terms of the procedures inc, which increments its argument by 1, and dec, which decrements its
   ↳ argument by 1.
9
10 (define (my-inc x) (begin (prn "inc") (+ x 1) ) ) ; inc with side effects
11 (define (my-dec x) (begin (prn "dec") (- x 1) ) ) ; dec with side effects
12
13 (define (recursive-+ a b)
14   (if (= a 0) b (my-inc (recursive-+ (my-dec a) b))))
15
16 (define (iterative-+ a b)
17   (if (= a 0) b (iterative-+ (my-dec a) (my-inc b)))) ; proper tail recursion
18
19 ;;; Using the substitution model, illustrate the process generated by each procedure in evaluating (+ 4
   ↳ 5).
20 ;;; Are these processes iterative or recursive?
21
22 (define a 4)
23 (define b 5)
24
25 (prnvar "a" a )
26 (prnvar "b" b )
27 (prnvar "recursive" (recursive-+ a b) ) ; recursive
28 (prnvar "iterative" (iterative-+ a b) ) ; iterative
```

29  
30 ; ; \*EOF\*

```
## ./sicp_ch1_e1-9.scm  
  
a := 4  
b := 5  
dec  
dec  
dec  
dec  
inc  
inc  
inc  
inc  
recursive := 9  
dec  
inc  
dec  
inc  
dec  
inc  
dec  
inc  
iterative := 9
```

Exercise 1.10: The following procedure computes a mathematical function called Ackermann's function.

```
(define (A x y)  
  (cond ((= y 0) 0)  
        ((= x 0) (* 2 y))  
        ((= y 1) 2)  
        (else (A (- x 1) (A x (- y 1))))))
```

What are the values of the following expressions?

```
(A 1 10)  
(A 2 4)  
(A 3 3)
```

Consider the following procedures, where A is the procedure defined above:

```
(define (f n) (A 0 n))  
(define (g n) (A 1 n))  
(define (h n) (A 2 n))  
(define (k n) (* 5 n n))
```



```

1  #!/usr/bin/csi -s
2  ;; sicp_ch1_e1-10.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  ;; Exercise 1.10: The following procedure computes a mathematical function called Ackermann's function.
9  ;;
10 (define (A x y)
11   (cond ((= y 0) 0)
12         ((= x 0) (* 2 y))
13         ((= y 1) 2)
14         (else (A (- x 1) (A x (- y 1))))))
15
16 ;; What are the values of the following expressions?
17 ;; (A 1 10)
18 ;; (A 2 4)
19 ;; (A 3 3)
20 ;;
21 ;; Consider the following procedures, where A is the procedure defined above:
22 ;; (define (f n) (A 0 n))
23 ;; (define (g n) (A 1 n))
24 ;; (define (h n) (A 2 n))
25 ;; (define (k n) (* 5 n n))
26
27 ;; Give concise mathematical definitions for the functions computed by the procedures f , g , and h for
   ⇨ positive integer values of n. For example, (k n) computes 5n2 .
28
29 (define (A x y)
30   (cond ((= y 0) 0)
31         ((= x 0) (* 2 y))
32         ((= y 1) 2)
33         (else (A (- x 1) (A x (- y 1))))))
34
35 (bar)
36 (prnvar "(A 1 10)" (A 1 10))
37 (prnvar "(A 2 4)" (A 2 4))
38 (prnvar "(A 3 3)" (A 3 3))
39 (hr)

```

```

40 (prn "(f n) := 2*n")
41 (prn "(g n) := 2")
42 (prn "(h n) := ")
43 (prn "(k n) := 5*n^2")
44 (bar)
45
46 ;; *EOF*

```

```
## ./sisp_ch1_e1-10.scm
```

```

=====
(A 1 10) := 1024
(A 2 4) := 65536
(A 3 3) := 65536
=====

```

```

(f n) := 2*n
(g n) := 2
(h n) :=
(k n) := 5*n^2
=====

```

Exercise 1.11: A function  $f$  is defined by the rule that

$$f(n) = \begin{cases} n & n < 3 \\ 1f(n-1) + 2f(n-2) + 3f(n-3) & \text{otherwise} \end{cases} \quad (6)$$

Write a procedure that computes  $f$  by means of a recursive process. Write a procedure that computes  $f$  by means of an iterative process.

Representing State Space Transitions

Direct Iterative Implementation

$$f(n) := s_0 \quad (7)$$

with state transition

$$\begin{matrix} T \\ \left[ \begin{array}{l} s_0 \leftarrow s_0 + 2s_1 + 3s_2 \\ s_1 \leftarrow s_0 \\ s_2 \leftarrow s_1 \end{array} \right] \end{matrix} \quad (8)$$

and initial conditions

$$\begin{array}{c} S_0 \\ \left[ \begin{array}{c} s_0 := 2 \\ s_1 := 1 \\ s_2 := 0 \end{array} \right] \end{array} \quad (9)$$

Linear Feedback Shift Register (LFSR) representation

$$f(n, \underline{s}) \leftarrow \begin{cases} n_1^{th} \underline{s} & n = 0 \\ f(n-1, n_1^{th} \sigma_1(\underline{s}), [1, 2, 3]) & \text{otherwise} \end{cases} \quad (10)$$

$$x, y \triangleq \sum_k x_k y_k = x_k y^k \quad (11)$$

$$n_k^{th} \triangleq x_k \quad (12)$$

$$\sigma_k(\underline{x}) \triangleq x_{(n+k) \bmod |x|} \forall n \in \underline{x} \quad (13)$$

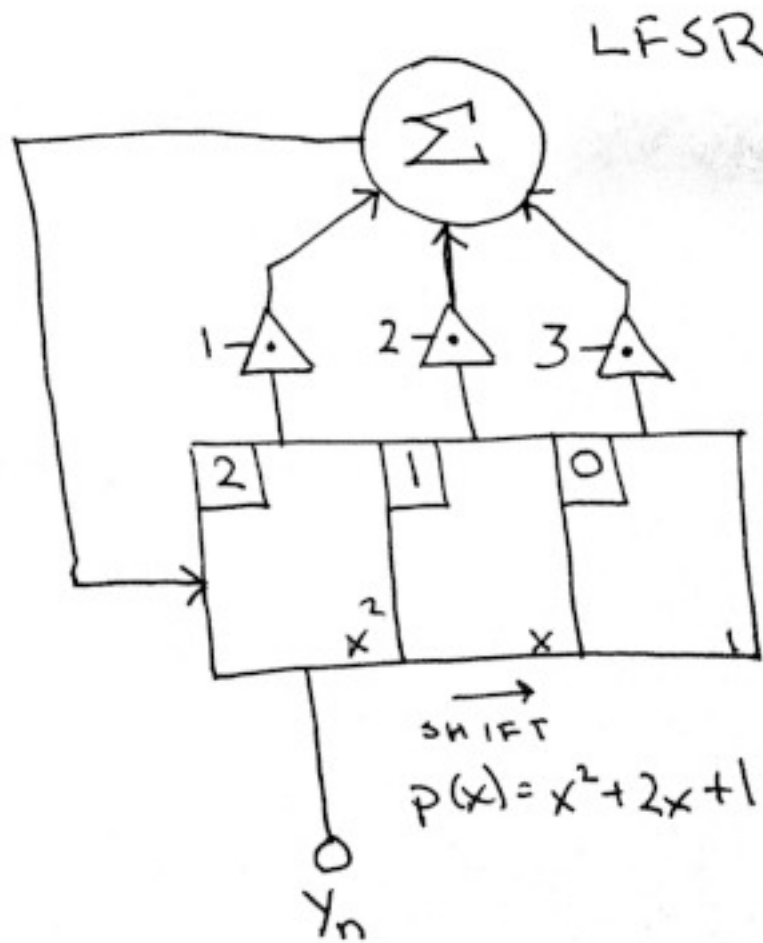


Figure 1: Linear Feedback Shift Register (LFSR)

State Space Representation

$$\mathbf{X}_k = \mathbf{F}\mathbf{X}_{k-1} \quad (14)$$

$$\begin{bmatrix} X_k \\ x'_0 \\ x'_1 \\ x'_2 \end{bmatrix} = \begin{bmatrix} F \\ 1 & 2 & 3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_{k-1} \\ x_0 \\ x_1 \\ x_2 \end{bmatrix} \quad (15)$$

where

$$\mathbf{X}_0 = \begin{bmatrix} X_0 \\ 2 \\ 1 \\ 0 \end{bmatrix} \quad (16)$$

so

$$\mathbf{X}_k = \mathbf{F}\mathbf{X}_{k-1} = \mathbf{F}(\mathbf{F}\mathbf{X}_{k-2}) = \mathbf{F}(\mathbf{F}(\mathbf{F}\mathbf{X}_{k-3})) = \cdots = \mathbf{F}^N \mathbf{X}_0 \quad (17)$$

```

1  #!/usr/bin/csi -s
2  ;; sicp_ch1_e1-1.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  ;; Exercise 1.11: A function f is defined by the rule that
9  ;;
10 ;;      { n                      if n<3,
11 ;; f(n)={ f(n-1)+2f(n-2)+3f(n-3) if n>3
12 ;;
13 ;; Write a procedure that computes f by means of a recursive process. Write a procedure that computes f
14 ;; ↪ by means of an iterative process.
15
16 ;; =====
17 ;; RECURSIVE
18 ;; =====
19
20 ;;      { n                      if n<3
21 ;; f(n)={
22 ;;      { f(n-1) + 2f(n-2) + 3f(n-3) otherwise
23
24 ;; f(n) recursive form
25 (define (f-recursive n)
26   (if (< n 3)
27       n
28       (+ (f-recursive (- n 1)) (* 2 (f-recursive (- n 2)) ) (* 3 (f-recursive (- n 3)) ) )

```

```

29     )
30 )
31
32 ;; =====
33 ;; DIRECT ITERATIVE
34 ;; =====
35
36 ;; NB:  $f(n) = 1*f(n-1) + 2*f(n-2) + 3*f(n-3)$ 
37 ;;
38 ;;       $f(n) = s0$ 
39 ;;      state transition
40 ;;       $s0 \leftarrow s0 + 2*s1 + 3*s2$ 
41 ;;       $s1 \leftarrow s0$ 
42 ;;       $s2 \leftarrow s1$ 
43
44
45 ;;  $f(n)$  direct form
46 (define (f-direct n)
47   (f-direct-iter 2 1 0 n) ; initial state vector [ 0 1 2 ]
48 )
49
50 ;;  $f(n)$  direct form iteration step
51 (define (f-direct-iter s0 s1 s2 n)
52   (if (< n 3)
53     s0
54     (f-direct-iter
55       (+ (* 1 s0) (* 2 s1) (* 3 s2))
56       s0
57       s1
58       (- n 1)
59     ) ; next
60   ) ; iteration test
61 ) ; direct form
62
63 ;; however, in general,  $f(n)$  can be thought of as:
64
65 ;; =====
66 ;; Linear Feedback Shift Register (LFSR)
67 ;; =====
68

```

```

69 ;; 1) Linear Feedback Shift Register (LFSR)
70 ;;
71 ;; f[n] is a Linear Feedback Shift Register (LFSR) operating on the sequence of
72 ;; previous integers up to n with initial register state x0 := [ 0 1 2 ]
73 ;; and polynomial coefficients given by a := [ 1 2 3 ]
74 ;;
75 ;; x[k] = LFSR(x[k-1], a)
76 ;; = program { circshift(x), x_0 = <x,a> }
77 ;;
78 ;; f(n) = CAR of x[n]
79 ;;
80 ;; where
81 ;;
82 ;; x[0] := [ 0 1 2 ]
83 ;;
84 ;; a := [ 1 2 3 ]
85 ;;
86
87
88 ;; f(n) LFSR form
89 (define (f-lfsr n)
90   (let (
91     (a '(1 2 3)) ; coefficients a := [ 1 2 3 ]
92     (x0 '(2 1 0)) ; initial state x0 := [ 0 1 2 ]
93     (k (- n 2)) ; k transitions k := n - 2
94   ) ; bindings
95     (f-lfsr-iter x0 a k)
96   ) ; let
97 )
98
99 ;; f(n) LFSR form iteration step
100 (define (f-lfsr-iter x a k)
101   (if (= k 0)
102     (car x)
103     (f-lfsr-iter (lfsr x a) a (- k 1)))
104   )
105 )
106
107 ;; =====
108 ;; State Space Representation

```

```

109 ;; =====
110
111 ;; 2) State Space Representaiton
112 ;;
113 ;; f[n] is the effect of a system up to time n with a given state space
114 ;; representation F := [ 0 1 0 ; 0 0 1 ; 1 2 3 ], and with
115 ;; initial conditions x0 := [ 0 1 2 ]
116 ;;
117 ;; x[k] = F * x[k-1]
118 ;;      = F * ( F * x[k-2] )
119 ;;      = F * ( F * ( F * x[k-3] ) )
120 ;;      = ...
121 ;;      = F^n * x0
122 ;;
123 ;; f(n) = x[n]
124 ;;
125 ;; where
126 ;;
127 ;; x[0] := [ 2 1 0 ]'
128 ;;
129 ;;      [ 1 2 3 ]
130 ;;      F := [ 1 0 0 ]
131 ;;      [ 0 1 0 ]
132 ;;
133
134 ;; version #1, using Iverson matrix representation
135
136 (define (f-ss n)
137   (let (
138     (t_ref 2) ; reference time relative to state space
139     (x0 '(2 1 0)) ; initial state x0
140     (F '(1 2 3
141          1 0 0
142          0 1 0 )
143       ) ; state transition matrix F
144     (dimF '(3 3)) ; F is MxN = 3x3
145     (dimX '(3 1)) ; X is Nx1 = 3x1
146   ) ; bindings
147   (car (f-ss-iter F dimF x0 dimX (- n t_ref)) )
148 ) ; let

```



```

149 )
150
151 (define (f-ss-iter F dimF x dimX k)
152   (if (< k 1)
153     x
154     ;; x[k] = F * x[k-1]
155     (f-ss-iter F dimF (mat-* F dimF x dimX) dimX (- k 1))
156   ) ; each
157 ) ; ff-ss-iter
158
159 (define n 12)
160
161 (prnvar "recursive f(n)" (f-recursive n) ) ; recursive
162 (prnvar "  direct f(n)" (f-direct n) ) ; direct
163 (prnvar "    LFSR f(n)" (f-lfsr n) ) ; LFSR
164 (prnvar "      SS f(n)" (f-ss n) ) ; state space
165
166 ;; *EOF*

```

```

## ./sicp_ch1_e1-11.scm

recursive f(n) := 10661
  direct f(n) := 10661
    LFSR f(n) := 10661
      SS f(n) := 10661

```

### 1.2.2 Tree Recursion

### 1.2.3 Orders of Growth

### 1.2.4 Exponentiation

Exercise 1.16: Design a procedure that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps, as does fast-expt. (Hint: Using the observation that  $(b^{\frac{n}{2}})^2 = (b^2)^{\frac{n}{2}}$ , keep, along with the exponent  $n$  and the base  $b$ , an additional state variable  $a$ , and define the state transformation in such a way that the product  $a b^n$  is unchanged from state to state. At the beginning of the process  $a$  is taken to be 1, and the answer is given by the value of  $a$  at the end of the process. In general, the technique of defining an invariant quantity that remains unchanged from state to

state is a powerful way to think about the design of iterative algorithms.)

$$f_{\text{benchmark}}(x, n) = \begin{cases} 1 & \text{if } n \text{ is zero} \\ f_{\text{benchmark}}(x, \frac{n}{2})^2 & \text{if } n \text{ is even, nonzero} \\ f_{\text{benchmark}}(x, n-1)^2 & \text{if } n \text{ is odd} \end{cases} \quad (18)$$

may be restructured as

$$f(x, n) = f_k(x, n, p) \quad (19)$$

where

$$f_k(x, n, p) = \begin{cases} p & \text{if } n \text{ is zero} \\ f_k(x, \frac{n}{2}, p) & \text{if } n \text{ is even, nonzero} \\ f_k(x, n-1, p \cdot x) & \text{if } n \text{ is odd} \end{cases} \quad (20)$$

---

```

1  #!/usr/bin/csi -s
2  ;; sicp_ch1_e1-16.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8
9  ;; Exercise 1.16. Design a procedure that evolves an iterative exponentiation process that uses
   ↪ successive squaring and uses a logarithmic number of steps, as does fast-expt. (Hint: Using the
   ↪ observation that (bn/2)2 = (b2)n/2, keep, along with the exponent n and the base b, an additional
   ↪ state variable a, and define the state transformation in such a way that the product a bn is
   ↪ unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is
   ↪ given by the value of a at the end of the process. In general, the technique of defining an invariant
   ↪ quantity that remains unchanged from state to state is a powerful way to think about the design of
   ↪ iterative algorithms.)
10
11  ;; =====
12  ;; benchmark from book:
13  ;;
14  ;;          { 1          if n is zero

```

```

15 ;; f(x,n) = { f(x,n/2)^2      if n is even, nonzero
16 ;;           { f(x,n-1)      if n is odd
17 ;;
18
19 (define (even? n)
20   (= (remainder n 2) 0))
21
22 (define (ref-fast-expt b n)
23   (cond ((= n 0) 1)
24         ((even? n) (square (ref-fast-expt b (/ n 2)) ))
25         (else (* b (ref-fast-expt b (- n 1))))) ))
26
27
28 ;; =====
29 ;; propagating product up through recursion:
30 ;;
31 ;;           { p              if n is zero
32 ;; f(x,n,p) = { f(x,n/2,p)    if n is even, nonzero
33 ;;           { f(x,n-1,x*p)  if n is odd
34 ;;
35
36 (define (fast-expt-iter b n p)
37   (cond ((= n 0) p)
38         ((even? n) (fast-expt-iter (* b b) (/ n 2) p) )
39         (else (fast-expt-iter b (- n 1) (* b p)) )))
40
41 (define (sep-fast-expt b n)
42   (fast-expt-iter b n 1))
43
44
45 ;; =====
46 ;; encapsulated as a single function
47
48 (define (fast-expt b n)
49   ;;           { p              if n is zero
50   ;; f(x,n,p) = { f(x,n/2,p)    if n is even, nonzero
51   ;;           { f(x,n-1,x*p)  if n is odd
52   (define (f b n p)
53     (cond ((= n 0) p)
54           ((even? n) (f (* b b) (/ n 2) p) )

```

```

55     (else (f b (- n 1) (* b p)) )))
56   (f b n 1) ; call
57 )
58
59
60 ;; =====
61 ;; applying self-referencing lambdas
62
63 (define (sr-fast-expt b n)
64   (define f (lambda (@f)
65     (lambda (b n p)
66       (cond ((= n 0) p)
67             ((even? n) ((f f) (* b b) (/ n 2) p))
68             (else ((f f) b (- n 1) (* b p)))))
69     ) ; f(x,n)
70   ) ; self
71   ((f f) b n 1)
72 )
73
74
75 ;; =====
76 ;; with hygienic macros
77
78 (define-syntax call
79   (syntax-rules ()
80     ((_ f)
81      (f f)))
82
83 (define-syntax fn
84   (syntax-rules ()
85     ((_ signature self fn-base fn-iter)
86      (define signature
87        (define self (lambda (@self) fn-iter))
88        fn-base
89      ) )))
90
91 (fn (mac-fast-expt b n) f
92   ;; f(b,n,1)
93   ((call f) b n 1)
94   ;; f(b,n,p)

```

```

95     (lambda (b n p)
96       (cond ((= n 0) p )
97             ((even? n) ((call f) (* b b) (/ n 2) p) )
98             (else ((call f) b (- n 1) (* b p)) ))
99     ) ; f(x,n)
100 )
101
102
103 ;; =====
104 ;; test:
105 (define b 2)
106 (define n 8)
107
108 (bar)
109 (prn "intrinsic:")
110 (prn (expt b n)) ;
111 (hr)
112 (prn "reference:")
113 (prn (ref-fast-expt b n)) ;
114 (hr)
115 (prn "example 1-16: (separate functions)")
116 (prn (sep-fast-expt b n)) ;
117 (hr)
118 (prn "example 1-16: (nested functions)")
119 (prn (fast-expt b n)) ;
120 (hr)
121 (prn "example 1-16 (self-referencing lambdas):")
122 (prn (sr-fast-expt b n)) ;
123 (hr)
124 (prn "example 1-16 (using macros):")
125 (prn (mac-fast-expt b n) )
126 (bar)
127
128 ;; *EOF*

```

```

## ./sicp_ch1_e1-16.scm

=====
intrinsic:
256
-----
reference:
256
-----
example 1-16: (separate functions)
256
-----
example 1-16: (nested functions)
256
-----
example 1-16 (self-referencing lambdas):
256
-----
example 1-16 (using macros):
256
=====

```

### 1.2.5 Greatest Common Divisors

### 1.2.6 Example: Testing for Primality

## 1.3 Formulating Abstractions with Higher-Order Procedures

### 1.3.1 Procedures as Arguments

### 1.3.2 Constructing Procedures Using Lambda

### 1.3.3 Procedures as General Methods

### 1.3.4 Procedures as Returned Values

Exercise 1.42. Let  $f$  and  $g$  be two one-argument functions. The composition  $f$  after  $g$  is defined to be the function  $x \mapsto f(g(x))$ . Define a procedure `compose` that implements composition. For example, if `inc` is a procedure that adds 1 to its argument, `((compose square inc) 6)`

```

1  #!/usr/bin/csi -s
2  ;; sicp_ch1_e1-42.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7

```

```
8  ;;; Exercise 1.42. Let f and g be two one-argument functions. The composition f after g is defined to be
9  ;;; the function x f(g(x)). Define a procedure compose that implements composition. For example, if
10 ;;; inc is a procedure that adds 1 to its argument,
11 ;;; ((compose square inc) 6)
12
13 ;;; from util.scm
14 ; (define (square x) (map (lambda (x) (* x x)) x) )
15 ; (define (inc x) (+ x 1))
16 ; (define ((compose f g) x) (f (g x)))
17
18 (prn ((compose square inc) 6) ) ; 49
19
20 ;; *EOF*
```

```
## ./sicp_ch1_e1-42.scm
```

```
49
```

## 2 Building Abstractions with Data

### 2.1 Introduction to Data Abstraction

#### 2.1.1 Example: Arithmetic Operations for Rational Numbers

Exercise 2.1. Define a better version of `make-rat` that handles both positive and negative arguments. `Make-rat` should normalize the sign so that if the rational number is positive, both the numerator and denominator are positive, and if the rational number is negative, only the numerator is negative.

```
1  #!/usr/bin/csi -s
2  ;; sicp_ch2_e2-1.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  ;; Exercise 2.1. Define a better version of make-rat that handles both positive and negative
9  ;; arguments. Make-rat should normalize the sign so that if the rational number is positive, both the
10 ;; numerator and denominator are positive, and if the rational number is negative, only the numerator is
11 ;; negative.
12
13 (define (add-rat x y)
14   (make-rat (+ (* (numer x) (denom y))
15                 (* (numer y) (denom x)))
16             (* (denom x) (denom y))))
17
18 (define (sub-rat x y)
19   (make-rat (- (* (numer x) (denom y))
20                 (* (numer y) (denom x)))
21             (* (denom x) (denom y))))
22
23 (define (mul-rat x y)
24   (make-rat (* (numer x) (numer y))
25             (* (denom x) (denom y))))
26
27 (define (div-rat x y)
28   (make-rat (* (numer x) (denom y))
29             (* (denom x) (numer y))))
30
31 (define (equal-rat? x y)
```



```

32     (= (* (number x) (denom y))
33        (* (number y) (denom x))))
34
35 (define (signum x)
36   (if (> x 0) +1 -1) )
37
38 (define (make-rat num denom)
39   (cons (* (signum (* num denom)) (abs (/ num (gcd num denom)))) (abs (/ denom (gcd num denom)))) )
40
41 (define (number x)
42   (car x))
43
44 (define (denom x)
45   (cdr x))
46
47 (define x1 (make-rat 1 2)) ; x1 = 1/2
48 (define x2 (make-rat 1 4)) ; x2 = 1/4
49 (define x3 (make-rat 2 4)) ; x3 = 2/4
50 (define x4 (make-rat -1 2)) ; x4 = -1/2
51 (define x5 (make-rat 1 -4)) ; x5 = -1/4
52 (define x6 (make-rat -2 -4)) ; x6 = 2/4
53
54 (prvar "x1 = 1/2 " x1) ; 1/2
55 (prvar "x2 = 1/4 " x2) ; 1/4
56 (prvar "x3 = 2/4 " x3) ; 2/4
57 (prvar "x4 = -1/2 " x4) ; -1/2
58 (prvar "x5 = -1/4 " x5) ; -1/4
59 (prvar "x6 = 2/4 " x6) ; 2/4
60
61 (ck "x1*x2" equal-rat? (mul-rat x1 x2) (make-rat 1 8)) ; 1/2 * 1/4 = 1/8
62 (ck "x1*x4" equal-rat? (mul-rat x1 x4) (make-rat -1 4)) ; 1/2 * -1/2 = -1/4
63 (ck "x4*x5" equal-rat? (mul-rat x4 x5) (make-rat 1 8)) ; -1/2 * -1/4 = 1/8
64 (ck "x5*x6" equal-rat? (mul-rat x5 x6) (make-rat -1 8)) ; -1/4 * 2/4 = -1/8
65
66 ;; *EOF*

```

```
## ./sicp_ch2_e2-1.scm
```

### 2.1.2 Abstraction Barriers

Exercise 2.2. Consider the problem of representing line segments in a plane. Each segment is represented as a pair of points: a starting point and an ending point. Define a constructor `make-segment` and selectors `start-segment` and `end-segment` that define the representation of segments in terms of points. Furthermore, a point can be represented as a pair of numbers: the x coordinate and the y coordinate. Accordingly, specify a constructor `make-point` and selectors `x-point` and `y-point` that define this representation. Finally, using your selectors and constructors, define a procedure `midpoint-segment` that takes a line segment as argument and returns its midpoint (the point whose coordinates are the average of the coordinates of the endpoints). To try your procedures, you'll need a way to print points:

```
1  #!/usr/bin/csi -s
2  ;; sicp_ch2_e2-2.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  ;; Exercise 2.2. Consider the problem of representing line segments in a plane. Each segment is
9  ;; represented as a pair of points: a starting point and an ending point. Define a constructor
10 ;; make-segment and selectors start-segment and end-segment that define the representation
11 ;; of segments in terms of points. Furthermore, a point can be represented as a pair of numbers: the x
12 ;; coordinate and the y coordinate. Accordingly, specify a constructor make-point and selectors
13 ;; x-point and y-point that define this representation. Finally, using your selectors and
14 ;; constructors, define a procedure midpoint-segment that takes a line segment as argument and
15 ;; returns its midpoint (the point whose coordinates are the average of the coordinates of the
   ↪ endpoints).
16 ;; To try your procedures, you'll need a way to print points:
17
18 (define (print-point p)
19   (newline)
20   (display "(")
21   (display (x-point p))
22   (display ",")
23   (display (y-point p))
24   (display ")")
25   (newline)
26 )
27
```

```

28  ;; point construct
29  (define (make-point x y)
30    (cons x y))
31
32  (define (x-point pt)
33    (car pt))
34
35  (define (y-point pt)
36    (cdr pt))
37
38  (define (equal-point? pt1 pt2)
39    (and
40      (= (x-point pt1) (x-point pt2))
41      (= (y-point pt1) (y-point pt2))
42    )
43  )
44
45  ;; segment construct
46  (define (make-segment pt1 pt2)
47    (cons pt1 pt2))
48
49  (define (start-segment seg)
50    (car seg))
51
52  (define (end-segment seg)
53    (cdr seg))
54
55  (define (midpoint-segment seg)
56    (make-point
57      (/ (+ (x-point (start-segment seg)) (x-point (end-segment seg))) 2)
58      (/ (+ (y-point (start-segment seg)) (y-point (end-segment seg))) 2)
59    )
60  )
61
62  ;; test constructs
63  (define pt-00 (make-point 0 0)) ; (0, 0) origin
64  (define pt-10 (make-point 1 0)) ; (1, 0)
65  (define pt-01 (make-point 0 1)) ; (0, 1)
66
67  (define s-x (make-segment pt-00 pt-10)) ; (0,0) -> (1,0)

```

```

68 (define s-y (make-segment pt-00 pt-01)) ; (0,0) -> (1,0)
69 (define s-xy (make-segment pt-10 pt-01)) ; (1,0) -> (1,0)
70
71 (define pt-mid (midpoint-segment s-xy)) ; (0.5,0.5)
72
73 (print-point pt-mid)
74
75 (ck "midpoint" equal-point? pt-mid (make-point 0.5 0.5)) ; -1/4 * 2/4 = -1/8
76
77 ;; *EOF*

```

```
## ./sicp_ch2_e2-2.scm
```

```

(0.5,0.5)
midpoint = (0.5 . 0.5) ; ok: expected (0.5 . 0.5)

```

Exercise 2.3. Implement a representation for rectangles in a plane. (Hint: You may want to make use of exercise 2.2.) In terms of your constructors and selectors, create procedures that compute the perimeter and the area of a given rectangle. Now implement a different representation for rectangles. Can you design your system with suitable abstraction barriers, so that the same perimeter and area procedures will work using either representation?

```

1 #!/usr/bin/csi -s
2 ;; sicp_ch2_e1-1.scm
3 ;; Mac Radigan
4
5 (load "../library/util.scm")
6 (import util)
7
8 ;;; Exercise 2.3. Implement a representation for rectangles in a plane. (Hint: You may want to make use
9 ;;; of exercise 2.2.) In terms of your constructors and selectors, create procedures that compute the
10 ;;; perimeter and the area of a given rectangle. Now implement a different representation for rectangles.
11 ;;; Can you design your system with suitable abstraction barriers, so that the same perimeter and area
12 ;;; procedures will work using either representation?
13
14 ;; *EOF*

```

```
## ./sicp_ch2_e2-3.scm
```

### 2.1.3 What Is Meant by Data?

Exercise 2.4. Here is an alternative procedural representation of pairs. For this representation, verify that  $(\text{car } (\text{cons } x \ y))$  yields  $x$  for any objects  $x$  and  $y$ . What is the corresponding definition of  $\text{cdr}$ ? (Hint: To verify that this works, make use of the substitution model of section 1.1.5.)

$$\text{cons}(x, y) \triangleq \lambda m. mxy \quad (21)$$

$$\begin{aligned} \text{car}(z) &\triangleq \lambda z. z \lambda pq. p \\ &\rightarrow_{\beta} [\text{cons}(x, y) / z] \text{car}(z) \\ &= (\lambda m. mxy) \lambda pq. p \\ &\rightarrow_{\beta} (\lambda pq. p) xy \\ &\rightarrow_{\beta} x \end{aligned} \quad (22)$$

$$\begin{aligned} \text{cdr}(z) &\triangleq \lambda z. z \lambda pq. q \\ &\rightarrow_{\beta} [\text{cons}(x, y) / z] \text{cdr}(z) \\ &= (\lambda m. mxy) \lambda pq. q \\ &\rightarrow_{\beta} (\lambda pq. q) xy \\ &\rightarrow_{\beta} y \end{aligned} \quad (23)$$

```
1 | #!/usr/bin/csi -s
2 | ;; sicp_ch2_e2-4.scm
3 | ;; Mac Radigan
4 |
5 | (load "../library/util.scm")
```

```

6  (import util)
7
8  ;; Exercise 2.4. Here is an alternative procedural representation of
9  ;; pairs. For this representation, verify that (car (cons x y)) yields
10 ;; x for any objects x and y.
11
12 ;; What is the corresponding definition of cdr? (Hint: To verify that
13 ;; this works, make use of the substitution model of section 1.1.5.)
14
15 ;; alternate cons
16 (define (my-cons x y)
17   (lambda (m) (m x y)))
18
19 ;; alternate cdr
20 (define (my-car z)
21   (z (lambda (p q) p)))
22
23 ;; alternate car
24 (define (my-cdr z)
25   (z (lambda (p q) q)))
26
27 ;; =====
28 ;; TESTS
29 ;; =====
30
31 (define p (my-cons 'a 'b))
32
33 (bar)
34 (prnvar "(cons 'a 'b)" p)
35 (prnvar "(car (cons 'a 'b))" (my-car p))
36 (prnvar "(cdr (cons 'a 'b))" (my-cdr p))
37 (bar)
38
39 ;; *EOF*

```

```
## ./sicp_ch2_e2-4.scm
```

```

=====
(cons 'a 'b) := #<procedure (? m)>
(car (cons 'a 'b)) := a
(cdr (cons 'a 'b)) := b
=====

```

### 2.1.4 Extended Exercise: Interval Arithmetic

## 2.2 Hierarchical Data and the Closure Property

Exercise 2.17. Define a procedure `last-pair` that returns the list that contains only the last element of a given (nonempty) list:

```
1  #!/usr/bin/csi -s
2  ;; sicp_ch2_e2-17.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  ;; Exercise 2.17. Define a procedure last-pair that returns the list
9  ;; that contains only the last element of a given (nonempty) list:
10
11  (define (last-pair x)
12    (if (null? x)
13        #f ; case empty list
14        (if (null? (cdr x))
15            (car x)
16            (last-pair (cdr x))
17          )
18        )
19  )
20
21  ;; =====
22  ;; TESTS
23  ;; =====
24
25  (bar)
26  (prnvar "(23 72 149 34)" (last-pair (list 23 72 149 34)) ) ; 34
27  (prnvar "(          )" (last-pair (list)) ) ; #f
28  (bar)
29
30  ;; *EOF*
```

```
## ./sisp_ch2_e2-17.scm
```

```
=====
(23 72 149 34) := 34
(           ) := #f
=====
```

Exercise 2.18. Define a procedure reverse that takes a list as argument and returns a list of the same elements in reverse order:

```
1  #!/usr/bin/csi -s
2  ;; sisp_ch2_e2-17.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  ;; Exercise 2.18. Define a procedure reverse that takes a list as
9  ;; argument and returns a list of the same elements in reverse order:
10
11  ;; my-reverse
12  ; (define (my-reverse x)
13  ;   (if (null? x)
14  ;       (list)
15  ;       (append (my-reverse (cdr x)) (list (car x))))
16  ;   )
17  ; )
18
19  ;; =====
20  ;; TESTS
21  ;; =====
22
23  (bar)
24  (prnvar "(1 4 9 16 25)" (my-reverse (list 1 4 9 16 25)) ) ; (25 16 9 4 1)
25  (prnvar "(           )" (my-reverse (list)) )           ; #f
26  (bar)
27
28  ;; *EOF*
```

```
## ./sisp_ch2_e2-18.scm
```

```
=====
(1 4 9 16 25) := (25 16 9 4 1)
(           ) := ()
=====
```



### 2.2.1 Representing Sequences

### 2.2.2 Hierarchical Structures

Exercise 2.21. The procedure `square-list` takes a list of numbers as argument and returns a list of the squares of those numbers.

```
(square-list (list 1 2 3 4))  
  
(1 4 9 16)
```

Here are two different definitions of `square-list`. Complete both of them by filling in the missing expressions:

```
(define (square-list items)  
  (if (null? items)  
      nil  
      (cons <??> <??>)))
```

```
(define (square-list items)  
  (map <??> <??>))
```

$$\ell^2(\text{items}) \triangleq \begin{cases} \text{nil} & , \text{ null? items} \\ \text{items}_A^2 :: \ell^2(\text{items}_D) & \text{o.w.} \end{cases} \quad (24)$$

$$\ell^2(\text{items}) \triangleq \text{map } (\lambda x.x^2) \text{ items} \quad (25)$$

```
1 | #!/usr/bin/csi -s  
2 | ;; sicp_ch2_e2-21.scm  
3 | ;; Mac Radigan  
4 |  
5 | (load "../library/util.scm")  
6 | (import util)  
7 | (use sicp)
```

```

8
9   ;;; Exercise 2.21. The procedure square-list takes a list of numbers as argument and returns a list
10  ;;; of the squares of those numbers.
11  ;;;
12  ;;; (square-list (list 1 2 3 4))
13  ;;; (1 4 9 16)
14  ;;;
15  ;;; Here are two different definitions of square-list. Complete both of them by filling in the missing
16  ;;; expressions:
17  ;;;
18  ;;; (define (square-list items)
19  ;;; (if (null? items)
20  ;;; nil
21  ;;; (cons <??> <??>)))
22  ;;;
23  ;;; (define (square-list items)
24  ;;; (map <??> <??>))
25
26  (define (square-list-1 items)
27    (if (null? items)
28        nil
29        (cons (expt (car items) 2) (square-list-1 (cdr items))))
30    )
31  )
32
33  (define (square-list-2 items)
34    (map (lambda (x) (expt x 2)) items)
35    )
36
37  (define x (list 1 2 3 4))
38
39  (bar)
40  (prnvar "x" x)
41  (hr)
42  (prnvar "(square-list-1 x)" (square-list-1 x))
43  (br)
44  (prnvar "(square-list-2 x)" (square-list-2 x))
45  (bar)
46
47  ;; *EOF*

```

```
## ./sicp_ch2_e2-21.scm

=====
x := (1 2 3 4)
-----
(square-list-1 x) := (1 4 9 16)
(square-list-2 x) := (1 4 9 16)
=====
```

Exercise 2.23. The procedure `for-each` is similar to `map`. It takes as arguments a procedure and a list of elements. However, rather than forming a list of the results, `for-each` just applies the procedure to each of the elements in turn, from left to right. The values returned by applying the procedure to the elements are not used at all – `for-each` is used with procedures that perform an action, such as printing. For example,

```
(for-each (lambda (x) (newline) (display x))
  (list 57 321 88))
```

57

321

88

The value returned by the call to `for-each` (not illustrated above) can be something arbitrary, such as `true`. Give an implementation of `for-each`.

$$\text{for-each}(f, x) \triangleq \begin{cases} \#t & \text{null? } x \\ \text{progn: } fx_A ; \text{for-each}(f, x_D) & \text{o.w.} \end{cases} \quad (26)$$

```
1 #!/usr/bin/csi -s
2 ;; sicp_ch2_e2-23.scm
3 ;; Mac Radigan
4
5 (load "../library/util.scm")
6 (import util)
7
8 ;; Exercise 2.23. The procedure for-each is similar to map. It takes as arguments a procedure and a
```

```

9  ;; list of elements. However, rather than forming a list of the results, for-each just applies the
10 ;; procedure to each of the elements in turn, from left to right. The values returned by applying the
11 ;; procedure to the elements are not used at all -- for-each is used with procedures that perform an
12 ;; action, such as printing. For example,
13 ;;   (for-each (lambda (x) (newline) (display x))
14 ;;     (list 57 321 88))
15 ;;
16 ;;   57
17 ;;   321
18 ;;   88
19 ;;
20 ;; The value returned by the call to for-each (not illustrated above) can be something arbitrary, such
21 ;; as true. Give an implementation of for-each.
22
23
24 (define (for-each f x)
25   (if (null? x)
26       #t
27       (begin
28         (f (car x))
29         (for-each f (cdr x))
30       )
31     )
32 )
33
34
35 (bar)
36 (for-each (lambda (x) (newline) (display x)) (list 57 321 88))
37 (br)
38 (bar)
39
40 ;; *EOF*

```

```
## ./sicp_ch2_e2-23.scm
```

```

=====
57
321
88
=====

```

Suppose we evaluate the expression (list 1 (list 2 (list 3 4))). Give the result printed by the interpreter,

the corresponding box-and-pointer structure, and the interpretation of this as a tree (as in figure 2.6).

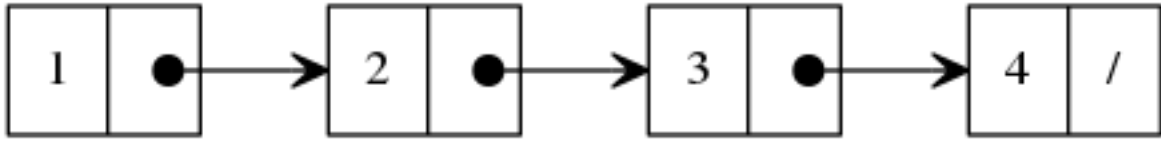


Figure 2: box and pointer representation

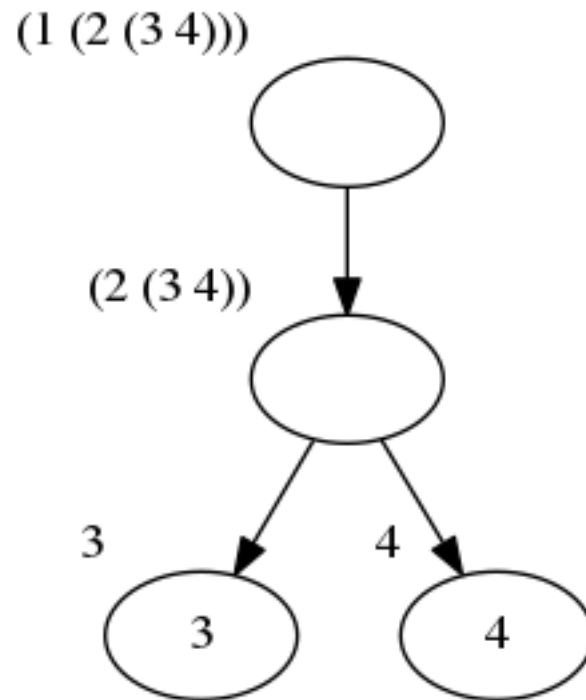


Figure 3: tree representation

```

1 #!/usr/bin/csi -s
2 ;; sicp_ch2_e2-24.scm
3 ;; Mac Radigan
4
5 (load "../library/util.scm")
6 (import util)
7
8 ;; Exercise 2.24. Suppose we evaluate the expression (list 1 (list 2 (list 3 4))).
9 ;; Give the result printed by the interpreter, the corresponding box-and-pointer
10 ;; structure, and the interpretation of this as a tree (as in figure 2.6).
11

```

```

12
13 (bar)
14 (prn (list 1 (list 2 (list 3 4))))
15 (bar)
16
17 ;; *EOF*

```

```
## ./sisp_ch2_e2-24.scm
```

```
=====
(1 (2 (3 4)))
=====
```

Exercise 2.25. Give combinations of cars and cdrs that will pick 7 from each of the following lists:

L1: (1 3 (5 7) 9)

L2: ((7))

L3: (1 (2 (3 (4 (5 (6 7))))))

```

1 #!/usr/bin/csi -s
2 ;; sisp_ch2_e2-25.scm
3 ;; Mac Radigan
4
5 (load "../library/util.scm")
6 (import util)
7
8 ;;; Exercise 2.25. Give combinations of cars and cdrs that will pick 7 from
9 ;;; each of the following lists:
10 ;;;
11 ;;; (1 3 (5 7) 9)
12 ;;; ((7))
13 ;;; (1 (2 (3 (4 (5 (6 7))))))
14
15 (define L1 '(1 3 (5 7) 9) )
16 (define L2 '((7)) )
17 (define L3 '(1 (2 (3 (4 (5 (6 7)))))) )
18
19 (bar)
20 (prnvar "L1" (car (cdaddr L1)))
21 (prnvar "L2" (caar L2))
22 (prnvar "L3" (car (cdr (cadr (cadr (cadr (cadr (cadr L3)))))))
23 (br)

```

```

24  (bar)
25
26  ;; *EOF*

```

```

## ./sisp_ch2_e2-25.scm

=====
L1 := 7
L2 := 7
L3 := 7
=====

```

### 2.2.3 Sequences as Conventional Interfaces

### 2.2.4 Example: A Picture Language

Exercise 2.44. Define the procedure `up-split` used by `corner-split`. It is similar to `right-split`, except that it switches the roles of `below` and `beside`.

```

1  #!/usr/bin/csi -s
2  ;; sisp_ch2_e2-44.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  (use sisp)
9
10 ;; Exercise 2.44. Define the procedure up-split used by corner-split.
11 ;; It is similar to right-split, except that it switches the roles of below ;; and beside.
12
13 (define (up-split painter n)
14   (if (= n 0)
15       painter
16       (let
17         ((subimage (up-split painter (- n 1))) )
18         (below painter (beside subimage subimage))
19       )
20   )
21 )
22
23 ;; =====

```

```

24 ;; TEST
25 ;; =====
26
27 (bar)
28 (prnvar "up-split lena.jpg" "../figures/sicp_ch2_e2-44.png")
29 (write-painter-to-png (up-split
30   (image->painter "../figures/lena.jpg") 2)
31   "../figures/sicp_ch2_e2-44.png")
32 (bar)
33
34 ;; *EOF*

```

```
## ./sicp_ch2_e2-44.scm
```

```
=====
up-split lena.jpg := ../figures/sicp_ch2_e2-44.png
```

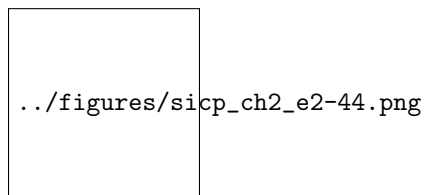


Figure 4: Up Split 2

Exercise 2.45. Right-split and up-split can be expressed as instances of a general splitting operation. Define a procedure `split` with the property that evaluating

```

(define right-split (split beside below))
(define up-split (split below beside))

```

produces procedures `right-split` and `up-split` with the same behaviors as the ones already defined.

```

1 #!/usr/bin/csi -s
2 ;; sicp_ch2_e2-45.scm
3 ;; Mac Radigan
4
5 (load "../library/util.scm")
6 (import util)
7
8 (use sicp)
9

```



```

10  ;; Exercise 2.45. Right-split and up-split can be expressed as
11  ;; instances of a general splitting operation. Define a procedure
12  ;; split with the property that evaluating
13  ;;
14  ;; (define right-split (split beside below))
15  ;; (define up-split (split below beside))
16  ;;
17  ;; produces procedures right-split and up-split with the same
18  ;; behaviors as the ones already defined.
19
20  (define (split dir1 dir2)
21    (lambda (painter n)
22      (if (= n 0)
23          painter
24          (let
25              ( (subimage ((split dir1 dir2) painter (- n 1))) )
26                (dir1 painter (dir2 subimage subimage))
27              ) ; let
28            ) ; if
29          ) ; lambda
30    ) ; split
31
32  (define right-split (split beside below))
33
34  (define up-split (split below beside))
35
36  ;; =====
37  ;; TEST
38  ;; =====
39
40  (bar)
41  (prnvar "right-split lena.jpg" "../figures/sicp_ch2_e2-45_right.png")
42  (write-painter-to-png (right-split
43      (image->painter "../figures/lena.jpg") 2)
44      "../figures/sicp_ch2_e2-45_right.png")
45  (hr)
46  (prnvar "up-split lena.jpg" "../figures/sicp_ch2_e2-45_up.png")
47  (write-painter-to-png (up-split
48      (image->painter "../figures/lena.jpg") 2)
49      "../figures/sicp_ch2_e2-45_up.png")

```

```
50 (bar)
51
52 ;; *EOF*
```

```
## ./sicp_ch2_e2-45.scm
```

```
=====
right-split lena.jpg := ../figures/sicp_ch2_e2-45_right.png
```

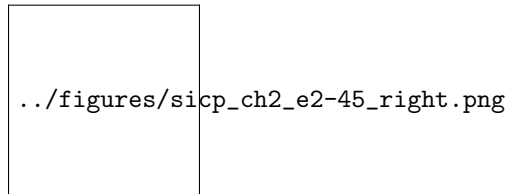


Figure 5: Right Split 2

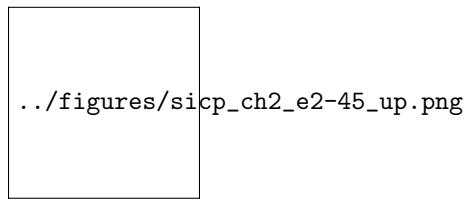


Figure 6: Up Split 2

Exercise 2.46. A two-dimensional vector  $v$  running from the origin to a point can be represented as a pair consisting of an x-coordinate and a y-coordinate. Implement a data abstraction for vectors by giving a constructor `make-vect` and corresponding selectors `xcor-vect` and `ycor-vect`. In terms of your selectors and constructor, implement procedures `add-vect`, `sub-vect`, and `scale-vect` that perform the operations vector addition, vector subtraction, and multiplying a vector by a scalar:

```
1 #!/usr/bin/csi -s
2 ;; sicp_ch2_e2-46.scm
3 ;; Mac Radigan
4
5 (load "../library/util.scm")
6 (import util)
7
8 ;;; Exercise 2.46. A two-dimensional vector v running from the
9 ;;; origin to a point can be represented as a pair consisting
10 ;;; of an x-coordinate and a y-coordinate. Implement a data
```

```

11  ;;; abstraction for vectors by giving a constructor make-vect
12  ;;; and corresponding selectors xcor-vect and ycor-vect. In
13  ;;; terms of your selectors and constructor, implement procedures
14  ;;; add-vect, sub-vect, and scale-vect that perform the operations
15  ;;; vector addition, vector subtraction, and multiplying a vector
16  ;;; by a scalar:
17
18  (define (make-vect x y)
19    (cons x y))
20
21  (define (xcor-vect v)
22    (car v))
23
24  (define (ycor-vect v)
25    (cdr v))
26
27  (define (scale-vect v s)
28    (make-vect (* s (xcor-vect v)) (* s (ycor-vect v)) ) )
29
30  (define (add-vect v1 v2)
31    (make-vect
32      (+ (xcor-vect v1) (xcor-vect v2))
33      (+ (ycor-vect v1) (ycor-vect v2)) ) )
34
35  (define (sub-vect v1 v2)
36    (make-vect
37      (- (xcor-vect v1) (xcor-vect v2))
38      (- (ycor-vect v1) (ycor-vect v2)) ) )
39
40  ;; =====
41  ;; TESTS
42  ;; =====
43
44  (define v1 (make-vect 1 2))
45  (define v2 (make-vect 1 -4))
46
47  (prnvar "v1" " v1)
48  (prnvar "v2" " v2)
49  (prnvar "v1.x" " (xcor-vect v1))
50  (prnvar "v1.y" " (ycor-vect v1))

```

```

51 (prnvar "v1" (scale-vect v1 2))
52 (prnvar "v1+v2" (add-vect v1 v2))
53 (prnvar "v1-v2" (sub-vect v1 v2))
54
55 ;; *EOF*

```

```
## ./sicp_ch2_e2-46.scm
```

```

v1      := (1 . 2)
v2      := (1 . -4)
v1.x    := 1
v1.y    := 2
v1      := (2 . 4)
v1+v2   := (2 . -2)
v1-v2   := (0 . 6)

```

## 2.3 Symbolic Data

### 2.3.1 Quotation

Exercise 2.54. Two lists are said to be equal? if they contain equal elements arranged in the same order. For example,

```
(equal? '(this is a list) '(this is a list))
```

is true, but

```
(equal? '(this is a list) '(this (is a) list))
```

is false. To be more precise, we can define equal? recursively in terms of

the basic eq? equality of symbols by saying that a and b are equal? if they are both symbols and the symbols are eq?, or if they are both lists such that (car a) is equal? to (car b) and (cdr a) is equal? to (cdr b). Using this idea, implement equal? as a procedure.

Comparing to structures using *equals?*:

$$(equals? \ a \ b) = f(a, b) = \begin{cases} \perp & \text{if } S(a) \oplus S(b) \\ a \stackrel{?}{=} b & \text{if } S(a) \wedge S(b) \\ f(a_A, b_A) f(a_D, b_D) & \text{if } L(a) \wedge L(b) \end{cases} \quad (27)$$

where

$$L(a) \wedge L(b) \implies \neg(S(a) \wedge S(b)) \quad (28)$$

To show that all cases have been exhausted (29):

$S(a)$	$S(b)$	$S(a) \wedge S(b)$	$S(a) \oplus S(b)$	$\neg(S(a) \wedge S(b))$
$F$	$F$	$F$	$F$	$T$
$F$	$T$	$F$	$T$	$F$
$T$	$F$	$F$	$T$	$F$
$T$	$T$	$T$	$F$	$F$

(29)

At this point, the recursive form is functional, but it is not expressed in tail-recursive form, and as such is not subject to tail-call optimization. The following is a conversion to tail-recursive form:

$$(equals? \ a \ b) = f(a, b) = f_k(a, b, \top) \quad (30)$$

$$f_k(a, b, p) = \begin{cases} \perp & \text{if } S(a) \oplus S(b) \\ p \wedge (a \stackrel{?}{=} b) & \text{if } S(a) \wedge S(b) \\ f_k(a_D, b_D, f_k(a_A, b_A, p)) & \text{otherwise} \end{cases} \quad (30)$$

```

1  #!/usr/bin/csi -s
2  ;; sicp_ch2_e2-54.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  ;; Exercise 2.54. Two lists are said to be equal? if they contain equal
9  ;; elements arranged in the same order. For example,
10 ;;
11 ;; (equal? '(this is a list) '(this is a list))
12 ;;
13 ;; is true, but
14 ;;
15 ;; (equal? '(this is a list) '(this (is a) list))
16 ;;
17 ;; is false. To be more precise, we can define equal? recursively in terms of
18 ;; the basic eq? equality of symbols by saying that a and b are equal? if
19 ;; they are both symbols and the symbols are eq?, or if they are both lists
20 ;; such that (car a) is equal? to (car b) and (cdr a) is equal? to (cdr b).
21 ;; Using this idea, implement equal? as a procedure.
22
23 ;; =====
24 ;; NOT TAIL-RECURSIVE
25 ;; =====
26
27 ;;
28 ;;      { #f                                if s(a) and s(b)
29 ;; f(a,b) = { a =?= b                        if s(a) xor s(b)
30 ;;      { f(car(a),car(b)) ^ f(cdr(a),cdr(b)) if !( s(a) and s(b) )
31 ;;
32 (define (my-equal-subopt? a b)

```

```

33 (define (notlist? x) (not (list? x)))
34 (cond
35   ;;
36   ;; case: null(a) ^ null(b) -> #t (null check)
37   ;;
38   ( (and (null? a) (null? b))
39     #t )
40   ;;
41   ;; case: s(a) ^ s(b) -> a == b
42   ;;
43   ( (and (notlist? a) (notlist? b))
44     (eq? a b) )
45   ;;
46   ;; case: s(a) xor s(b) -> #f
47   ;;
48   ( (xor (notlist? a) (notlist? b))
49     #f )
50   ;;
51   ;; case: !( s(a)^s(b) ) -> f( cdr(a), cdr(b) )
52   ;;
53   ;; i.e. (not (and (notlist? a) (notlist? b)))
54   ;;
55   ( else
56     (and (eq? (car a) (car b)) (my-equal-subopt? (cdr a) (cdr b))) )
57   )
58 )
59
60 ;; =====
61 ;; TAIL-RECURSIVE
62 ;; =====
63
64 ;;
65 ;; { #f if s(a) and s(b)
66 ;; f(a,b,p=#t) = { p ^ a == b if s(a) xor s(b)
67 ;; { f(cdr(a), cdr(b), f(car(a),car(a),p)) if !( s(a) and s(b) )
68 ;;
69 (define (my-equal? a b)
70   (define (notlist? x) (not (list? x)))
71   (define (f a b p)
72     (cond

```

```

73      ;;
74      ;; case: null(a) ^ null(b) -> #t (null check)
75      ;;
76      (      (and (null? a) (null? b) )
77              #t )
78      ;;
79      ;; case: s(a) ^ s(b)          -> a == b
80      ;;
81      (      (and (notlist? a) (notlist? b))
82              (eq? a b) )
83      ;;
84      ;; case: s(a) xor s(b)        -> #f
85      ;;
86      (      (xor (notlist? a) (notlist? b))
87              #f )
88      ;;
89      ;; case: !( s(a)^s(b) )      -> f( cdr(a), cdr(b) )
90      ;;
91      ( else
92          (f (cdr a) (cdr b) (f (car a) (car b) p) ) )
93      ) ;; cond
94      )      ;; <= recur
95      (f a b #t) ;; <= base
96      )
97
98      (bar)
99      (prn "intrinsic:")
100      (prn (equal? '(this is a list) '(this is a list))      )
101      (prn (equal? '(this is a list) '(this (is a) list))      )
102      (hr)
103      (prn "example 2-54: (not tail-recursive)")
104      (prn (my-equal-subopt? '(this is a list) '(this is a list))      )
105      (prn (my-equal-subopt? '(this is a list) '(this (is a) list))      )
106      ; (hr)
107      ; (prn "example 2-54: (tail-recursive)")
108      ; (prn (my-equal? '(this is a list) '(this is a list))      )
109      ; (prn (my-equal? '(this is a list) '(this (is a) list))      )
110      (bar)
111
112      ;; *EOF*

```



```

## ./sicp_ch2_e2-54.scm

=====

intrinsic:
#t
#f
-----

example 2-54: (not tail-recursive)
#t
#f
=====

```

Exercise 2.55. Eva Lu Ator types to the interpreter the expression (car 'abracadabra)

To her surprise, the interpreter prints back quote. Explain.

```

1  #!/usr/bin/csi -s
2  ;; sicp_ch2_e2-55.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  ;;; Exercise 2.55. Eva Lu Ator types to the interpreter the expression (car 'abracadabra)
9  ;;; To her surprise, the interpreter prints back quote. Explain.
10
11  (bar)
12  (prn (car 'abracadabra) )
13  (hr)
14  (prn "Quote constructs a non-modifiable list, whose contents are the literal arguments to quote.")
15  (prn "The second quote is part of the literal quoted list.")
16  (prn "Car returns the first element of the list, which itself is quote.")
17  (bar)
18
19  ;; *EOF*

```

```

## ./sicp_ch2_e2-55.scm

=====

quote
-----

Quote constructs a non-modifiable list, whose contents are the literal arguments to quote.
The second quote is part of the literal quoted list.
Car returns the first element of the list, which itself is quote.
=====

```

### 2.3.2 Example: Symbolic Differentiation

Exercise 2.56. Show how to extend the basic differentiator to handle more kinds of expressions. For instance, implement the differentiation rule

$$\frac{\partial d(u^n)}{\partial u} = nu^{n-1} \frac{\partial u}{\partial x} \quad (30)$$

```
1  #!/usr/bin/csi -s
2  ;; sicp_ch2_e2-56.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  ;; Exercise 2.56. Show how to extend the basic differentiator to handle more
9  ;; kinds of expressions. For instance, implement the differentiation rule
10 ;;
11 ;;      d(u^n)          du
12 ;;      ----- = nu^(n-1) --
13 ;;      dx              dx
14 ;;
15
16 (define (deriv exp var)
17   (cond ((number? exp) 0)
18         ((variable? exp)
19          (if (same-variable? exp var) 1 0))
20         ((sum? exp)
21          (make-sum (deriv (addend exp) var)
22                     (deriv (augend exp) var)))
23         ((product? exp)
24          (make-sum
25           (make-product (multiplier exp)
26                         (deriv (multiplicand exp) var))
27           (make-product (deriv (multiplier exp) var)
28                         (multiplicand exp))))
29         ;
30         ;      d(u^n)          du
31         ;      ----- = nu^(n-1) --
```

```

32      ;      dx      dx
33      ;
34      ((exponent? exp)
35       (make-product
36        (make-product (power exp)
37                       (make-exponent (base exp) (- (power exp) 1)))
38        (deriv (base exp) var)))
39      (else
40       (error "unknown expression type -- DERIV" exp)))
41
42      (define (variable? x) (symbol? x))
43
44      (define (same-variable? v1 v2)
45        (and (variable? v1) (variable? v2) (eq? v1 v2)))
46
47      (define (make-sum a1 a2) (list '+ a1 a2))
48
49      (define (=number? exp num) (and (number? exp) (= exp num)))
50
51      (define (make-product m1 m2) (list '* m1 m2))
52
53      (define (make-exponent b p)
54        (cond ((=number? p 0) 1)
55              ((=number? p 1) b)
56              (else ' (^ b p))))
57
58      (define (exponent? x) (eq? (car x) '^))
59
60      (define (base x) (cadr x))
61      (define (power x) (caddr x))
62
63      (define (sum? x) (and (pair? x) (eq? (car x) '+)))
64
65      (define (addend s) (cadr s))
66
67      (define (augend s) (caddr s))
68
69      (define (product? x)
70        (and (pair? x) (eq? (car x) '*)))
71

```

```

72 (define (multiplier p) (cadr p))
73
74 (define (multiplicand p) (caddr p))
75
76 (bar)
77 (prnvar "d/dx (2x)^4" (deriv ' (^ (* 2 x) 4) 'x))
78 (bar)
79
80 ;; *EOF*

```

```
## ./sisp_ch2_e2-56.scm
```

```
=====
d/dx (2x)^4 := (* (* 4 ((quote ^) b p)) (+ (* 2 1) (* 0 x)))
=====
```

Exercise 2.73. Section 2.3.2 described a program that performs symbolic differentiation:

```

(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                     (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
          (make-product (multiplier exp)
                        (deriv (multiplicand exp) var))
          (make-product (deriv (multiplier exp) var)
                        (multiplicand exp))))

```

<more rules can be added here>

```

      (else (error "unknown expression type -- DERIV" exp)))

```

We can regard this program as performing a dispatch on the type of the expression to be differentiated. In this situation the "type tag" of the datum is the algebraic operator symbol (such as `+`) and the operation being performed is `deriv`. We can transform this program into data-directed style by rewriting the basic

derivative procedure as

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        (else ((get 'deriv (operator exp)) (operands exp)
                var))))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
```

Constant Rule

$$\frac{\partial c}{\partial x} = 0 \quad (30)$$

Sum Rule

$$\frac{\partial}{\partial x} (u + v) = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial x} \quad (30)$$

Product Rule

$$\frac{\partial}{\partial x} (f \cdot g) = v \cdot \frac{\partial u}{\partial x} + u \cdot \frac{\partial v}{\partial x} \quad (30)$$

Exponent Rule

$$\frac{\partial d(u^n)}{\partial u} = nu^{n-1} \frac{\partial u}{\partial x} \quad (30)$$

Chain Rule

$$\frac{\partial}{\partial x} (f \circ g) = \left( \frac{\partial f}{\partial x} \circ g \right) \cdot \frac{\partial g}{\partial x} \quad (30)$$

a. Explain what was done above. Why can't we assimilate the predicates `number?` and `same-variable?` into the data-directed dispatch?

b. Write the procedures for derivatives of sums and products, and the auxiliary code required to install them in the table used by the program above.

c. Choose any additional differentiation rule that you like, such as the one for exponents (exercise 2.56), and install it in this data-directed system.

d. In this simple algebraic manipulator the type of an expression is the algebraic operator that binds it together. Suppose, however, we indexed the procedures in the opposite way, so that the dispatch line in `deriv` looked like

```
((get (operator exp) 'deriv) (operands exp) var)
```

What corresponding changes to the derivative system are required?

```
1  #!/usr/bin/csi -s
2  ;; sicp_ch2_e2-73.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7  (use sicp)
8
9  ;; Exercise 2.73. Section 2.3.2 described a program that performs symbolic differentiation:
10 ;;
11 ;; (define (deriv exp var)
12 ;;   (cond ((number? exp) 0)
13 ;;         ((variable? exp) (if (same-variable? exp var) 1 0))
14 ;;         ((sum? exp)
15 ;;          (make-sum (deriv (addend exp) var)
16 ;;                    (deriv (augend exp) var)))
17 ;;         ((product? exp)
18 ;;          (make-sum
19 ;;            (make-product (multiplier exp)
```

```

20 ;;;          (deriv (multiplicand exp) var))
21 ;;;          (make-product (deriv (multiplier exp) var)
22 ;;;          (multiplicand exp))))
23 ;;;
24 ;;; <more rules can be added here>
25 ;;;
26 ;;;          (else (error "unknown expression type -- DERIV" exp))))
27
28
29 ;;; We can regard this program as performing a dispatch on the type of the expression to be
    ↪ differentiated.
30 ;;; In this situation the "type tag" of the datum is the algebraic operator symbol (such as +) and the
31 ;;; operation being performed is deriv. We can transform this program into data-directed style by
32 ;;; rewriting the basic derivative procedure as
33 ;;;
34 ;;; (define (deriv exp var)
35 ;;; (cond ((number? exp) 0)
36 ;;;       ((variable? exp) (if (same-variable? exp var) 1 0))
37 ;;;       (else ((get 'deriv (operator exp)) (operands exp)
38 ;;;             var))))
39 ;;; (define (operator exp) (car exp))
40 ;;; (define (operands exp) (cdr exp))
41
42 (define (deriv expr var)
43   (cond ((number? expr) 0)
44         ((variable? expr) (if (same-variable? expr var) 1 0))
45         (else ((get 'deriv (operator expr)) (operands expr) var))))
46
47 (define (operator expr) (car expr))
48
49 (define (operands expr) (cdr expr))
50
51
52 ;;; a. Explain what was done above. Why can't we assimilate the predicates number? and
53 ;;; same-variable? into the data-directed dispatch?
54
55
56 ;;; b. Write the procedures for derivatives of sums and products, and the auxiliary code required to
    ↪ install
57 ;;; them in the table used by the program above.

```

```

58
59 (define (install-sum-package)
60   (define (addend expr) (car expr))
61   (define (augend expr) (cadr expr))
62   (define (make-sum a b)
63     (cond
64       ((eq? a 0) b)
65       ((eq? b 0) a)
66       ((and (number? a) (number? b)) (+ a b))
67       (else (list '+ a b))
68     )
69   )
70   ;; sum rule: fg = f' + g'
71   (define (deriv-sum expr var)
72     (make-sum (deriv (addend expr) var) (deriv (augend expr) var))
73   )
74   (put 'deriv '+ deriv-sum)
75   'done
76 )
77
78 (define (install-product-package)
79   (define (multiplier expr) (car expr))
80   (define (multiplicand expr) (cadr expr))
81   (define (make-product a b)
82     (cond
83       ((or (eq? a 0) (eq? b 0)) 0)
84       ((eq? a 1) b)
85       ((eq? b 1) a)
86       ((and (number? a) (number? b)) (* a b))
87       (else (list '* a b))
88     )
89   )
90   (define (make-sum a b)
91     (cond
92       ((eq? a 0) b)
93       ((eq? b 0) a)
94       ((and (number? a) (number? b)) (+ a b))
95       (else (list '+ a b))
96     )
97   )

```



```

98      ;; product rule: fg = f g' + f' g
99      (define (deriv-product expr var)
100        (make-sum
101          (make-product (multiplier expr) (deriv (multiplicand expr) var))
102          (make-product (deriv (multiplier expr) var) (multiplicand expr) )
103        )
104      )
105      (put 'deriv '* deriv-product)
106      'done
107    )
108
109    (install-sum-package)
110    (install-product-package)
111
112    (define dx 'x)
113
114    (bar)
115    (prn " b. Write the procedures for derivatives of sums and products, ")
116    (prn "      and the auxiliary code required to install them in the table ")
117    (prn "      used by the program above.")
118    (hr)
119    (define expr-1 '(* x x))
120    (prn "e1 := (* x x)")
121    (prnvar "d/dx e1" (deriv expr-1 dx))
122
123    (hr)
124    (define expr-2 '(+ (* x x) x))
125    (prn "e2 := (+ (* x x) x)")
126    (prnvar "d/dx e2" (deriv expr-2 dx))
127
128    (hr)
129    (define expr-3 '(* (+ (* x x) (* x z) ) (+ (* x y) (* x x)) ) )
130    (prn "e3 := (* (+ (* x x) (* x z) ) (+ (* x y) (* x x)) )")
131    (prnvar "d/dx e3" (deriv expr-3 dx))
132
133    ;; c. Choose any additional differentiation rule that you like, such as the one for exponents
134    ;;      (exercise 2.56), and install it in this data-directed system.
135
136    (define (install-exponent-package)
137      (define (base expr) (car expr))

```

```

138 (define (power expr) (cadr expr))
139 (define (make-product a b)
140   (cond
141     ((or (eq? a 0) (eq? b 0)) 0)
142     ((eq? a 1) b)
143     ((eq? b 1) a)
144     ((and (number? a) (number? b)) (* a b))
145     (else (list '* a b))
146   )
147 )
148 (define (make-sum a b)
149   (cond
150     ((eq? a 0) b)
151     ((eq? b 0) a)
152     ((and (number? a) (number? b)) (+ a b))
153     (else (list '+ a b))
154   )
155 )
156 (define (make-exponent b p)
157   (cond ((=number? p 0) 1)
158         ((=number? p 1) b)
159         (else (list '^ b p))
160   )
161 )
162 ;
163 ;  $d(u^n)$   $du$ 
164 ;  $----- = nu^{(n-1)} --$ 
165 ;  $dx$   $dx$ 
166 ;
167 (define (deriv-exponent expr var)
168   (make-product
169     (make-product (power expr)
170                   (make-exponent (base expr) (- (power expr) 1)))
171     (deriv (base expr) var)
172   )
173 )
174 (put 'deriv '^ deriv-exponent)
175 'done
176 )
177

```

```

178 (install-exponent-package)
179
180
181 (bar)
182 (prn " c. Choose any additional differentiation rule that you like, ")
183 (prn "      such as the one for exponents (exercise 2.56), and install ")
184 (prn "      it in this data-directed system.")
185 (hr)
186 (define expr-4 ' (^ (* 2 x) 4) )
187 (prn "e4 := (^ (* 2 x) 4) ")
188 (prnvar "d/dx e4" (deriv expr-4 dx))
189 (hr)
190 (define expr-5 ' (* 2 (^ x 4)) )
191 (prn "e5 := (* 2 (^ x 4)) ")
192 (prnvar "d/dx e5" (deriv expr-5 dx))
193
194 ;;; d. In this simple algebraic manipulator the type of an expression is the algebraic operator that binds
195 → it
196 ;;; together. Suppose, however, we indexed the procedures in the opposite way, so that the dispatch
197 → line
198 ;;; in deriv looked like
199 ;;; ((get (operator exp) 'deriv) (operands exp) var)
200 ;;; What corresponding changes to the derivative system are required?
201
202 (define (my-deriv expr var)
203   (cond ((number? expr) 0)
204         ((variable? expr) (if (same-variable? expr var) 1 0))
205         (else ((get (my-operator expr) 'my-deriv) (my-operands expr) var))))
206
207 (define (my-operator expr) (car expr))
208
209 (define (my-operands expr) (cdr expr))
210
211 (define (my-install-sum-package)
212   (define (addend expr) (car expr))
213   (define (augend expr) (cadr expr))
214   (define (make-sum a b)
215     (cond

```

```

216     ((eq? a 0) b)
217     ((eq? b 0) a)
218     ((and (number? a) (number? b)) (+ a b))
219     (else (list '+ a b))
220   )
221 )
222 ;; sum rule: fg = f' + g'
223 (define (deriv-sum expr var)
224   (make-sum (my-deriv (addend expr) var) (my-deriv (augend expr) var))
225 )
226 (put '+ 'my-deriv deriv-sum)
227 'done
228 )
229
230 (define (my-install-product-package)
231   (define (multiplier expr) (car expr))
232   (define (multiplicand expr) (cadr expr))
233   (define (make-product a b)
234     (cond
235       ((or (eq? a 0) (eq? b 0)) 0)
236       ((eq? a 1) b)
237       ((eq? b 1) a)
238       ((and (number? a) (number? b)) (* a b))
239       (else (list '* a b))
240     )
241   )
242   (define (make-sum a b)
243     (cond
244       ((eq? a 0) b)
245       ((eq? b 0) a)
246       ((and (number? a) (number? b)) (+ a b))
247       (else (list '+ a b))
248     )
249   )
250   ;; product rule: fg = f g' + f' g
251   (define (deriv-product expr var)
252     (make-sum
253       (make-product (multiplier expr) (my-deriv (multiplicand expr) var))
254       (make-product (my-deriv (multiplier expr) var) (multiplicand expr) )
255     )

```

```

256     )
257     (put '* 'my-deriv deriv-product)
258     'done
259 )
260
261 (bar)
262 (prn " d.  Suppose, however, we indexed the procedures in the opposite")
263 (prn "      way, so that the dispatch line in deriv looked like")
264 (br)
265 (prn "      ((get (operator exp) 'deriv) (operands exp) var)")
266 (hr)
267 (prn "(put '+ 'deriv deriv-sum)")
268 (prn "(put '* 'deriv deriv-product)")
269 (br)
270
271 ; (hr)
272 ; (prn "e1 := (* x x)")
273 ; (prnvar "d/dx e1" (my-deriv expr-1 dx))
274
275 ; (hr)
276 ; (prn "e2 := (+ (* x x) x)")
277 ; (prnvar "d/dx e2" (my-deriv expr-2 dx))
278
279 ; (hr)
280 ; (prn "e3 := (* (+ (* x x) (* x z) ) (+ (* x y) (* x x) ) )")
281 ; (prnvar "d/dx e3" (my-deriv expr-3 dx))
282 (bar)
283
284 ;; *EOF*

```

```

## ./sicp_ch2_e2-73.scm

=====
b. Write the procedures for derivatives of sums and products,
and the auxiliary code required to install them in the table
used by the program above.
-----

e1 := (* x x)
d/dx e1 := (+ x x)
-----

e2 := (+ (* x x) x)
d/dx e2 := (+ (+ x x) 1)
-----

e3 := (* (+ (* x x) (* x z)) (+ (* x y) (* x x)))
d/dx e3 := (+ (* (+ (* x x) (* x z)) (+ y (+ x x))) (* (+ (+ x x) z) (+ (* x y) (* x x))))
=====

c. Choose any additional differentiation rule that you like,
such as the one for exponents (exercise 2.56), and install
it in this data-directed system.
-----

e4 := (^ (* 2 x) 4)
d/dx e4 := (* (* 4 (^ (* 2 x) 3)) 2)
-----

e5 := (* 2 (^ x 4))
d/dx e5 := (* 2 (* 4 (^ x 3)))
=====

d. Suppose, however, we indexed the procedures in the opposite
way, so that the dispatch line in deriv looked like

((get (operator exp) 'deriv) (operands exp) var)
-----

(put '+ 'deriv deriv-sum)
(put '* 'deriv deriv-product)
=====

```

**2.3.3 Example: Representing Sets**

**2.3.4 Example: Huffman Encoding Trees**

**2.4 Multiple Representations for Abstract Data**

**2.4.1 Representations for Complex Numbers**

**2.4.2 Tagged data**

**2.4.3 Data-Directed Programming and Additivity**

**2.5 Systems with Generic Operations**

**2.5.1 Generic Arithmetic Operations**

**2.5.2 Combining Data of Different Types**

**2.5.3 Example: Symbolic Algebra**

## 3 Modularity, Objects, and State

### 3.1 Assignment and Local State

#### 3.1.1 Local State Variables

Exercise 3.1: An accumulator is a procedure that is called repeatedly with a single numeric argument and accumulates its arguments into a sum. Each time it is called, it returns the currently accumulated sum. Write a procedure `make-accumulator` that generates accumulators, each maintaining an independent sum. The input to `make-accumulator` should specify the initial value of the sum; for example:

```
(define A (make-accumulator 5))
```

```
(A 10)
```

```
15
```

```
(A 10)
```

```
25
```

---

```
1  #!/usr/bin/csi -s
2  ;; sicmp_ch3_e3-1.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  (use sicmp sicmp-eval sicmp-eval-anal sicmp-streams)
9  (load "../ch3support.scm")
10 ; (load "../ch3.scm")
11
12
13 ;; Exercise 3.1: An accumulator is a procedure that is called repeatedly
14 ;; with a single numeric argument and accumulates its arguments into a sum.
15 ;; Each time it is called, it returns the currently accumulated sum. Write a
16 ;; procedure make-accumulator that generates accumulators, each main-
17 ;; taining an independent sum. The input to make-accumulator should specify the initial
18 ;; value of the sum; for example
19 ;;
20 ;; (define A (make-accumulator 5))
```



```

21  ;;
22  ;; (A 10)
23  ;; 15
24  ;;
25  ;; (A 10)
26  ;; 25
27
28  (define (make-accumulator n)
29    (let ((acc n))
30      (lambda (f) (set! acc (+ acc f)) acc)
31    )
32  )
33
34  (bar)
35  (define A (make-accumulator 5))
36  (prnvar "acc+10" (A 10))
37  (hr)
38  (prnvar "acc+10" (A 10))
39  (bar)
40
41  ;; *EOF*

```

Exercise 3.2: In so039ware-testing applications, it is useful to be able to count the number of times a given procedure is called during the course of a computation. Write a procedure `make-monitored` that takes as input a procedure, `f`, that itself takes one input. The result returned by `make-monitored` is a third procedure, say `mf`, that keeps track of the number of times it has been called by maintaining an internal counter. If the input to `mf` is the special symbol `how-many-calls?`, then `mf` returns the value of the counter. If the input is the special symbol `reset-count`, then `mf` resets the counter to zero. For any other input, `mf` returns the result of calling `f` on that input and increments the counter. For instance, we could make a monitored version of the `sqrt` procedure:

```
(define s (make-monitored sqrt))
```

```
(s 100)
```

```
10
```

```
(s 'how-many-calls?)
```

```

1  #!/usr/bin/csi -s
2  ;; sicp_ch3_e3-1.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  (use sicp sicp-eval sicp-eval-anal sicp-streams)
9  (load "../ch3support.scm")
10 ; (load "../ch3.scm")
11
12
13 ;; Exercise 3.2: In so\ware-testing applications, it is useful
14 ;; to be able to count the number of times a given procedure
15 ;; is called during the course of a computation. Write a pro-
16 ;; cedure make-monitored that takes as input a procedure, f ,
17 ;; that itself takes one input. \e result returned by make-
18 ;; monitored is a third procedure, say mf , that keeps track
19 ;; of the number of times it has been called by maintaining
20 ;; an internal counter. If the input to mf is the special symbol
21 ;; how-many-calls? , then mf returns the value of the counter.
22 ;; If the input is the special symbol reset-count , then mf re-
23 ;; sets the counter to zero. For any other input, mf returns the
24 ;; result of calling f on that input and increments the counter.
25 ;; For instance, we could make a monitored version of the
26 ;; sqrt procedure:
27 ;;
28 ;; (define s (make-monitored sqrt))
29 ;;
30 ;; (s 100)
31 ;; 100
32 ;;
33 ;; (s 'how-many-calls?)
34 ;; 1
35
36 (define (make-monitored f)
37   (let ((count 0))
38     (lambda (arglist)

```

```

39     (cond
40       ((equal? arglist 'how-many-calls?) count)
41       ((eq? arglist 'reset-count) (set! count 0))
42       (else
43         (begin
44           (set! count (+ count 1))
45           (f arglist)
46         )
47       )
48     )
49   )
50 )
51 )
52
53 (bar)
54 (define s (make-monitored sqrt))
55 (prnvar "(s 100)" (s 100))
56 (hr)
57 (prnvar "(s 'how-many-calls?)" (s 'how-many-calls?))
58 (hr)
59 (prn "(s 'reset-count)")
60 (s 'reset-count)
61 (prnvar "(s 'how-many-calls?)" (s 'how-many-calls?))
62 (hr)
63 (bar)
64
65 ;; *EOF*

```

### 3.1.2 The Benefits of Introducing Assignment

### 3.1.3 The Costs of Introducing Assignment

## 3.2 The Environment Model of Evaluation

### 3.2.1 The Rules for Evaluation

### 3.2.2 Applying Simple Procedures

### 3.2.3 Frames as the Repository of Local State

### 3.2.4 Internal Definitions

## 3.3 Modeling with Mutable Data

### 3.3.1 Mutable List Structure

### 3.3.2 Representing Queues

### 3.3.3 Representing Tables

### 3.3.4 A Simulator for Digital Circuits

### 3.3.5 Propagation of Constraints

## 3.4 Concurrency: Time Is of the Essence

### 3.4.1 The Nature of Time in Concurrent Systems

### 3.4.2 Mechanisms for Controlling Concurrency

## 3.5 Streams

### 3.5.1 Streams Are Delayed Lists

Exercise 3.50. Complete the following definition, which generalizes stream-map to allow procedures that take multiple arguments, analogous to map in section 2.2.3, footnote 12.

```
(define (stream-map proc . argstreams)
  (if (<??> (car argstreams))
      the-empty-stream
      (<??>
       (apply proc (map <??> argstreams))
       (apply stream-map
```

```
(cons proc (map <??> argstreams))))))
```

```
1 #!/usr/bin/csi -s
2 ;; sicp_ch3_e3-50.scm
3 ;; Mac Radigan
4
5 (load "../library/util.scm")
6 (import util)
7
8 (use sicp sicp-eval sicp-eval-anal sicp-streams)
9 (load "../ch3support.scm")
10 (load "../ch3.scm")
11
12 ;;; Exercise 3.50. Complete the following definition, which generalizes stream-map
13 ;;; to allow procedures that take multiple arguments, analogous to map in
14 ;;; section 2.2.3, footnote 12.
15
16 ;;; (define (stream-map proc . argstreams)
17 ;;; (if (<??> (car argstreams))
18 ;;; the-empty-stream
19 ;;; (<??>
20 ;;; (apply proc (map <??> argstreams))
21 ;;; (apply stream-map
22 ;;; (cons proc (map <??> argstreams))))))
23
24 (define (stream-map proc . argstreams)
25   (if (stream-null? (car argstreams))
26       the-empty-stream
27       (cons-stream
28         (apply proc (map car argstreams))
29         (apply stream-map
30           (cons proc (map stream-cdr argstreams))))))
31
32 ;;; NB Error: unbound variable: get-new-pair
33 ;;; (constructor for the empty list)
34
35 ;; *EOF*
```

### **3.5.2 Infinite Streams**

### **3.5.3 Exploiting the Stream Paradigm**

### **3.5.4 Streams and Delayed Evaluation**

### **3.5.5 Modularity of Functional Programs and Modularity of Objects**

## 4 Metalinguistic Abstraction

```
1  #!/usr/bin/csi -s
2  ;; run-query.scm
3  ;; Mac Radigan
4
5  (use sicp sicp-eval sicp-eval-anal sicp-streams)
6  (load "./ch4-query.scm")
7  (define false #f)
8  (define true #t)
9  (initialize-data-base microshaft-data-base)
10 (query-driver-loop)
11
12 ;; *EOF*
```

## **4.1 The Metacircular Evaluator**

### **4.1.1 The Core of the Evaluator**

### **4.1.2 Representing Expressions**

### **4.1.3 Evaluator Data Structures**

### **4.1.4 Running the Evaluator as a Program**

### **4.1.5 Data as Programs**

### **4.1.6 Internal Definitions**

### **4.1.7 Separating Syntactic Analysis from Execution**

## **4.2 Variations on a Scheme – Lazy Evaluation**

### **4.2.1 Normal Order and Applicative Order**

### **4.2.2 An Interpreter with Lazy Evaluation**

### **4.2.3 Streams as Lazy Lists**

## **4.3 Variations on a Scheme – Nondeterministic Computing**

### **4.3.1 Amb and Search**

### **4.3.2 Examples of Nondeterministic Programs**

### **4.3.3 Implementing the Amb Evaluator**

## **4.4 Logic Programming**

An excellent discussion of logic programming can be found in chapter 19 of Paul Graham's *On Lisp* [?].

### **4.4.1 Deductive Information Retrieval**

Exercise 4.55. Give simple queries that retrieve the following information from the data base:

- (a) all people supervised by Ben Bitdiddle;
- (b) the names and jobs of all people in the accounting division;
- (c) the names and addresses of all people who live in Slumerville.



```

1  #!/usr/bin/csi -s
2  ;; sicmp_ch4_e4-55.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  (use sicmp sicmp-eval sicmp-eval-anal sicmp-streams)
9  (load "../ch4-query.scm")
10 (define false #f)
11 (define true #t)
12
13 (initialize-data-base microshaft-data-base)
14
15 ;; Exercise 4.55. Give simple queries that retrieve the following information from the data base:
16 ;;   a. all people supervised by Ben Bitdiddle;
17 ;;   b. the names and jobs of all people in the accounting division;
18 ;;   c. the names and addresses of all people who live in Slumerville.
19
20 ;; =====
21 ;; QUERY PROCESSOR
22 ;; =====
23
24 (define (eval-query query)
25   (let ((q (query-syntax-process query)))
26     (cond ((assertion-to-be-added? q)
27            (add-rule-or-assertion! (add-assertion-body q))
28            (newline)
29            (display "Assertion added to data base.")
30            )
31            (else
32             (newline)
33             (display output-prompt)
34             ;; [extra newline at end] (announce-output output-prompt)
35             (display-stream
36              (stream-map
37               (lambda (frame)
38                 (instantiate q
39                             frame
40                             (lambda (v f)

```

```

41         (contract-question-mark v))))
42     (qeval q (singleton-stream '()))))
43     ))))
44
45 ;; =====
46 ;; a. all people supervised by Ben Bitdiddle:
47 ;; =====
48 (define query-a
49   '(supervisor ?person (Bitdiddle Ben)) )
50
51 ;; =====
52 ;; b. the names and jobs of all people in the accounting division;
53 ;; =====
54 (define query-b
55   '(job ?person (accounting . ?title)) )
56
57 ;; =====
58 ;; c. the names and addresses of all people who live in Slumerville.
59 ;; =====
60 (define query-c '(address ?person (Slumerville . ?address)) )
61
62 ;; =====
63 ;; TESTS
64 ;; =====
65
66 (bar)
67 (prn "Query A. all people supervised by Ben Bitdiddle:")
68 (eval-query query-a) (br) (hr)
69 (prn "Query B. the names and jobs of all people in the accounting division:")
70 (eval-query query-b) (br) (hr)
71 (prn "Query C. the names and addresses of all people who live in Slumerville:")
72 (eval-query query-c) (br)
73 (bar)
74
75 ;; *EOF*

```

```

## ./sisp_ch4_e4-55.scm

=====
Query A. all people supervised by Ben Bitdiddle:

;;; Query results:
(supervisor (Tweakit Lem E) (Bitdiddle Ben))
(supervisor (Fect Cy D) (Bitdiddle Ben))
(supervisor (Hacker Alyssa P) (Bitdiddle Ben))
=====

Query B. the names and jobs of all people in the accounting division:

;;; Query results:
(job (Cratchet Robert) (accounting scrivener))
(job (Scrooge Eben) (accounting chief accountant))
=====

Query C. the names and addresses of all people who live in Slumerville:

;;; Query results:
(address (Aull DeWitt) (Slumerville (Onion Square) 5))
(address (Reasoner Louis) (Slumerville (Pine Tree Road) 80))
(address (Bitdiddle Ben) (Slumerville (Ridge Road) 10))
=====

```

Exercise 4.56. Formulate compound queries that retrieve the following information:

- (a) the names of all people who are supervised by Ben Bitdiddle, together with their addresses;
- (b) all people whose salary is less than Ben Bitdiddle's, together with their salary and Ben Bitdiddle's salary;
- (c) all people who are supervised by someone who is not in the computer division, together with the supervisor's name and job.

```

1  #!/usr/bin/csi -s
2  ;; sisp_ch4_e4-56.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  (use sisp sisp-eval sisp-eval-anal sisp-streams)
9  (load "../ch4-query.scm")
10 (define false #f)
11 (define true #t)
12
13 (initialize-data-base microshaft-data-base)
14
15 ;; Exercise 4.56. Formulate compound queries that retrieve the following information:

```

```

16 ;;      a. the names of all people who are supervised by Ben Bitdiddle, together with their addresses;
17 ;;      b. all people whose salary is less than Ben Bitdiddle's, together with their salary and Ben
   ↪      Bitdiddle's salary;
18 ;;      c. all people who are supervised by someone who is not in the computer division, together with
   ↪      the supervisor's name and job.
19
20 ;; =====
21 ;; QUERY PROCESSOR
22 ;; =====
23
24 (define (eval-query query)
25   (let ((q (query-syntax-process query)))
26     (cond ((assertion-to-be-added? q)
27            (add-rule-or-assertion! (add-assertion-body q))
28            (newline)
29            (display "Assertion added to data base.")
30            )
31            (else
32             (newline)
33             (display output-prompt)
34             ;; [extra newline at end] (announce-output output-prompt)
35             (display-stream
36              (stream-map
37               (lambda (frame)
38                 (instantiate q
39                             frame
40                             (lambda (v f)
41                               (contract-question-mark v))))
42              (qeval q (singleton-stream '()))))
43             ))))
44
45 ;; =====
46 ;; a. the names of all people who are supervised by
47 ;;      Ben Bitdiddle, together with their addresses
48 ;; =====
49 (define query-a
50   '(and (supervisor (Bitdiddle Ben) ?person)
51         (address ?person ?address)
52         ) ; conjunction
53   ) ; query A

```

```

54
55 ;; =====
56 ;; b. all people whose salary is less than Ben Bitdiddle's,
57 ;;    together with their salary and Ben Bitdiddle's salary
58 ;; =====
59
60 ;;; TODO
61
62 (define query-b
63   '(and (salary (Bitdiddle Ben) ?max-salary)
64         (salary ?person ?salary)
65         ) ; conjunction
66   ) ; query B
67
68 ; (define query-b
69 ;   '(and (salary (Bitdiddle Ben) ?max-salary)
70 ;         (salary ?person ?salary)
71 ;         (lisp-value < ?salary ?max-salary)
72 ;         ) ; conjunction
73 ; ) ; query B
74 ;
75 ;; =====
76 ;; c. all people who are supervised by someone who is not
77 ;;    in the computer division, together with the
78 ;;    supervisor's name and job.
79 ;; =====
80 (define query-c
81   '(and (supervisor ?supervisor ?person)
82         (not (job ?supervisor (computer . ?supervisor-title)))
83         (job ?supervisor ?supervisor-job)
84         ) ; conjunction
85   ) ; query C
86
87 ;; =====
88 ;; TESTS
89 ;; =====
90
91 (bar)
92 (prn "Query A. the names of all people who are supervised by Ben Bitdiddle, together with their
    ↪ addresses")

```

```

93 (eval-query query-a) (br) (hr)
94 (prn "[TODO] Query B. all people whose salary is less than Ben Bitdiddle's, together with their salary
    ↪ and Ben Bitdiddle's salary")
95 (eval-query query-b) (br) (hr)
96 (prn "Query C. all people who are supervised by someone who is not in the computer division, together
    ↪ with the supervisor's name and job.")
97 (eval-query query-c) (br)
98 (bar)
99
100 ;; *EOF*

```

```

## ./sisp_ch4_e4-56.scm

=====
Query A. the names of all people who are supervised by Ben Bitdiddle, together with their addresses

;;; Query results:
(and (supervisor (Bitdiddle Ben) (Warbucks Oliver)) (address (Warbucks Oliver) (Swellesley (Top Heap
    ↪ Road))))

-----

[TODO] Query B. all people whose salary is less than Ben Bitdiddle's, together with their salary and Ben
    ↪ Bitdiddle's salary

;;; Query results:
(and (salary (Bitdiddle Ben) 60000) (salary (Aull DeWitt) 25000))
(and (salary (Bitdiddle Ben) 60000) (salary (Cratchet Robert) 18000))
(and (salary (Bitdiddle Ben) 60000) (salary (Scrooge Eben) 75000))
(and (salary (Bitdiddle Ben) 60000) (salary (Warbucks Oliver) 150000))
(and (salary (Bitdiddle Ben) 60000) (salary (Reasoner Louis) 30000))
(and (salary (Bitdiddle Ben) 60000) (salary (Tweakit Lem E) 25000))
(and (salary (Bitdiddle Ben) 60000) (salary (Fect Cy D) 35000))
(and (salary (Bitdiddle Ben) 60000) (salary (Hacker Alyssa P) 40000))
(and (salary (Bitdiddle Ben) 60000) (salary (Bitdiddle Ben) 60000))

-----

Query C. all people who are supervised by someone who is not in the computer division, together with the
    ↪ supervisor's name and job.

;;; Query results:
(and (supervisor (Aull DeWitt) (Warbucks Oliver)) (not (job (Aull DeWitt) (computer . ?supervisor-title))))
    ↪ (job (Aull DeWitt) (administration secretary)))
(and (supervisor (Cratchet Robert) (Scrooge Eben)) (not (job (Cratchet Robert) (computer .
    ↪ ?supervisor-title))) (job (Cratchet Robert) (accounting scrivener)))
(and (supervisor (Scrooge Eben) (Warbucks Oliver)) (not (job (Scrooge Eben) (computer .
    ↪ ?supervisor-title))) (job (Scrooge Eben) (accounting chief accountant)))

=====

```

**4.4.2 How the Query System Works**

**4.4.3 Is Logic Programming Mathematical Logic?**

**4.4.4 Implementing the Query System**





## 5 Computing with Register Machines

### 5.1 Designing Register Machines

#### 5.1.1 A Language for Describing Register Machines

#### 5.1.2 Abstraction in Machine Design

#### 5.1.3 Subroutines

#### 5.1.4 Using a Stack to Implement Recursion

#### 5.1.5 Instruction Summary

### 5.2 A Register-Machine Simulator

#### 5.2.1 The Machine Model

#### 5.2.2 The Assembler

#### 5.2.3 Generating Execution Procedures for Instructions

#### 5.2.4 Monitoring Machine Performance

### 5.3 Storage Allocation and Garbage Collection

#### 5.3.1 Memory as Vectors

#### 5.3.2 Maintaining the Illusion of Infinite Memory

### 5.4 The Explicit-Control Evaluator

#### 5.4.1 The Core of the Explicit-Control Evaluator

#### 5.4.2 Sequence Evaluation and Tail Recursion

#### 5.4.3 Conditionals, Assignments, and Definitions

#### 5.4.4 Running the Evaluator

### 5.5 Compilation

#### 5.5.1 Structure of the Compiler

#### 5.5.2 Compiling Expressions

#### 5.5.3 Compiling Combinations

#### 5.5.4 Combining Instruction Sequences

#### 5.5.5 An Example of Compiled Code

#### 5.5.6 Lexical Addressing

## 6 Appendix A: Modules

### 6.1 util.scm

---

```
1  #!/usr/bin/csi -s
2  ;; util.scm
3  ;; Mac Radigan
4
5  (module util (
6      bind
7      bar
8      bin
9      br
10     but-last
11     ck
12     compose
13     dec
14     dotprod
15     flatmap
16     fmt
17     hr
18     hex
19     inc
20     my-iota
21     join
22     kron-comb
23     lfsr
24     mat-*
25     mat-col
26     mat-row
27     mod
28     my-last
29     nth
30     oct
31     permute
32     pr
33     prn
34     prnvar
35     my-reverse
36     range
```

```

37     rotate-right
38     rotate-left
39     rotate
40     square
41     sum
42     xor
43     Y
44     Y-normal
45     Y2
46     yeild
47 )
48 (import scheme chicken)
49 (use extras)
50 (use srfi-1)
51
52 ;;; debug, formatted printing, and assertions
53 (define (br)
54     (format #t "%"))
55
56 (define (pr x)
57     (format #t "~a" x))
58
59 (define (fmt s x)
60     (format #t s x))
61
62 (define (prn x)
63     (format #t "~a%" x))
64
65 (define (prnvar name value)
66     (format #t "~a := ~a%" name value))
67
68 (define (ck name pred? value expect)
69     (cond
70         ( (not (pred? value expect)) (format #t "~a = ~a ; fail expected ~a%" name value expect) )
71         )
72     (assert (pred? value expect))
73     (format #t "~a = ~a ; ok: expected ~a%" name value expect)
74 ) ; ck
75
76 ;;; numeric formatting

```

```

77 (define (hex x) (format #t "~x%" x))
78 (define (bin x) (format #t "~b%" x))
79 (define (oct x) (format #t "~o%" x))
80
81 ;;; delimiters
82 (define (bar) (format #t "~a~%" (make-string 80 #\=)))
83 (define (hr) (format #t "~a~%" (make-string 80 #\-)))
84
85 ;;; returns the nth element of list x
86 (define (nth x n)
87   (if (= n 1)
88       (car x)
89       (nth (cdr x) (- n 1)))
90   ) ; if last iter
91 ) ; nth
92
93 ;;; returns the inner product <u,v>
94 (define (dotprod u v)
95   (apply + (map * u v))
96 )
97
98 ;;; returns x mod n
99 (define (mod x n)
100   (- x (* n (floor (/ x n))))
101 )
102
103 ;;; the permutation x by p
104 (define (permute x p)
105   (map (lambda (pk) (nth x pk)) p)
106 )
107
108 ;;; circular shift (left) of x by n
109 (define (rotate-left x n)
110   (if (< n 1)
111       x
112       (rotate-left (append (cdr x) (list (car x))) (- n 1)))
113   ) ; if last iter
114 ) ; rotate-left
115
116 ;;; circular shift (right) of x by n

```

```

117 (define (rotate-right x n)
118   (if (< n 1)
119     x
120     (rotate-right
121      (append (list (my-last x)) (but-last x))
122      (- n 1)
123      ) ; call
124     ) ; if last iter
125   ) ; rotate-right
126
127   ;;; circular shift of x by n
128   (define (rotate x n)
129     (cond
130      ((= n 0) x)
131      ((> n 0) (rotate-right x n))
132      ((< n 0) (rotate-left x (abs n)))
133      )
134     )
135
136   ;;; return all but last element in list
137   (define (but-last x)
138     (if (null? x)
139       (list)
140       (if (null? (cdr x))
141         (list)
142         (cons (car x) (but-last (cdr x)))
143         ) ; end if list contains only one element
144       ) ; end if list null
145     )
146
147   ;;; return the last element in list
148   (define (my-last x)
149     (if (null? x)
150       #f
151       (if (null? (cdr x))
152         (car x)
153         (my-last (cdr x))
154         ) ; end if list contains only one element
155       ) ; end if list null
156     )

```

```

157
158 ;; composition
159 (define ((compose f g) x) (f (g x)))
160
161 ;; my-reverse
162 (define (my-reverse x)
163   (if (null? x)
164       (list)
165       (append (my-reverse (cdr x)) (list (car x)))))
166 )
167 )
168
169 ;; Linear Feedback Shift Register (LFSR)
170 ;; given initial state x[k-1] and coefficients a
171 ;; return next state x[k]
172 (define (lfsr x a)
173   (append (list (dotprod x a)) (cdr (rotate x +1)) ) ; next state x[k]
174 ) ; lfsr
175
176 ;; matrix multiplication of column-major Iverson matrices
177 (define (mat-* A dimA B dimB)
178   (let (
179     ;  $A_{m \times n} * B_{n \times k} = C_{m \times k}$ 
180     (M_rows (cadr dimA) ) ; M_rows
181     (N_cols (cadr dimB) ) ; N_cols
182     ) ; local bindings
183     (map (lambda (rc)
184       (dotprod (mat-row A dimA (car rc)) (mat-col B dimB (cadr rc)) )
185     )
186     (kron-comb (my-iota N_cols) (my-iota M_rows)))
187   )
188   ) ; let
189 ) ; mat-*
190
191 ;; selects the kth column from a column-major Iverson matrix
192 ;; NB: dim is a pair ( M_rows , N_cols )
193 (define (mat-col A dim k)
194   (let (
195     (start k ) ; start := kth column
196     (stride (cadr dim) ) ; stride := N_cols

```

```

197     (M_rows (car dim) ) ; M_rows
198     (N_cols (cadr dim) ) ; N_cols
199   ) ; local bindings
200   (choose A (range start stride M_rows))
201   ) ; let
202   ) ; mat-col
203
204   ;; selects the kth column from a column-major Iverson matrix
205   ;; NB: dim is a pair ( M_rows , N_cols )
206   (define (mat-row A dim k)
207     (let (
208       (start (* k (cadr dim)) ) ; start := (kth row -1) * M_rows
209       (stride 1 ) ; stride := 1
210       (M_rows (car dim) ) ; M_rows
211       (N_cols (cadr dim) ) ; N_cols
212     ) ; local bindings
213     (choose A (range start stride N_cols))
214     ) ; let
215     ) ; mat-col
216
217   ;; flatmap (map flattened by one level)
218   (define (flatmap f x)
219     (apply append (map f x))
220   ) ; flatmap
221
222   ;; Kroneker combination of vectors a and b
223   (define (kron-comb a b)
224     (flatmap (lambda (ak) (map (lambda (bk) (list ak bk)) a)) b)
225   ) ; kron-comb
226
227   ;; returns a list with elements of x taken from positions ns
228   (define (choose x ns)
229     (map (lambda (k) (list-ref x k)) ns )
230   )
231
232   ;; range sequence generator
233   (define (range start step n)
234     (range-iter '() start step n)
235   )
236

```

```

237  ;; Iverson's iota: zero-based sequence of integers from 0..N
238  (define (my-iota n)
239    (range 0 1 n)
240  )
241
242  ;; local scope: range sequence generator helper
243  (define (range-iter x val step n)
244    (if (< n 1)
245        x
246        (range-iter (append x (list val)) (+ val step) step (- n 1) ) ; x << val + step
247    )
248  )
249
250  ;; exclusive or
251  (define (xor a b)
252    (or (and (not a) b) (and a (not b)))
253  )
254
255  ;; square, sum, inc, and dec
256  (define (square x) (* x x))
257  (define (sum x) (apply + x) )
258  (define (inc x) (+ x 1))
259  (define (dec x) (- x 1))
260
261  ;;; data transformations: bind, join, yeild
262  (define (bind f x) (join (map f x)))
263  (define (join x) (apply append '() x))
264  (define yeild list)
265
266  ;; Y combiner
267  ;;  strict-order
268  (define Y
269    (lambda (f)
270      ((lambda (x) (x x))
271       (lambda (x) (f (lambda (y) ((x x) y)))))))
272  ;;  normal-order
273  (define (Y-normal f)
274    ((lambda (x) (x x))
275     (lambda (x) (f (x x)))))
276  (define Y2

```



```
277     (lambda (h)
278       (lambda args (apply (h (Y h)) args))))
279
280
281 ) ; module util
282
283 ;; hello.scm
284
285 ;; *EOF*
```

---

## 7 Appendix B: Installation Notes

### 7.1 Chicken Scheme

---

```
1  #!/bin/bash
2  ## apt-install.sh
3  ## Mac Radigan
4  apt install chicken-bin -y
5  ## *EOF*
```

---

---

```
1  #!/bin/bash
2  ## yum-install.sh
3  ## Mac Radigan
4  yum -y install chicken
5  ## *EOF*
```

---

---

```
1  #!/bin/bash
2  ## brew-install.sh
3  ## Mac Radigan
4  brew install chicken
5  ## *EOF*
```

---

---

```
1  #!/bin/bash
2  ## chicken-install.sh
3  ## Mac Radigan
4  chicken-install sicp
5  # *EOF*
```

---

## 8 Appendix C: Notation

### 8.1 Membership

$$S(x) \triangleq x \in \text{SYMBOL}$$

(30)

$$L(x) \triangleq x \in \text{LIST}$$

(31)

### 8.2 Symbols

$$\top \triangleq \#t$$

(32)

$$\perp \triangleq \#f$$

(33)

### 8.3 Access

$$x_A \triangleq (\text{car } x)$$

(34)

$$x_D \triangleq (\text{cdr } x)$$

(35)

## 8.4 Equality

$$x \stackrel{?}{=} y \triangleq (\text{eq? } x \ y)$$

(36)

## 8.5 Logic

$$\neg x \triangleq (\text{not } x)$$

(37)

$$x \wedge y \triangleq (\text{and } x \ y)$$

(38)

$$x \vee y \triangleq (\text{or } x \ y)$$

(39)

$$x \oplus y \triangleq (\neg x \wedge y) \vee (x \wedge \neg y) = (\text{or } (\text{and } (\text{not } x) \ (y)) \ (\text{and } (x) \ (\text{not } y)))$$

(40)

## 9 Appendix D: Y-Combinator

### 9.1 Introduction

Description of the Y Combinator based on Mike Mvanier's blog post [?]. see <http://mvanier.livejournal.com/2897.html>

### 9.2 Canonical Expression

Curry's Y Combinator [?] is defined as:

$$\mathbf{Y} = \lambda f. (\lambda x. f (xx)) (\lambda x. f (xx)) \quad (41)$$

When applied to a function  $g$ , the expansion follows [?]

$$\begin{aligned} \mathbf{Y}g &= (\lambda f. (\lambda x. f (xx)) (\lambda x. f (xx))) g \\ &= (\lambda x. g (xx)) (\lambda x. g (xx)) \\ &= g ((\lambda x. g (xx)) (\lambda x. g (xx))) \\ &= g (\mathbf{Y}g) \end{aligned} \quad (42)$$

### 9.3 Connonical Form in Scheme

Direct implementation of the above expression for the Y Combinator will not terminate during applicative order [?].

#### 9.3.1 Strict Scheme (Chicken)

Chicken scheme is a strict scheme, and evaluates in applicative order.

```
1  #!/usr/bin/csi -s
2  ;; y-combinator.scm
3  ;; Mac Radigan
4  ;;
5  ;; copied from http://mvanier.livejournal.com/2897.html
6
7  (load "../library/util.scm")
8  (import util)
9
10
11  ;;; Eliminating (most) explicit recursion (lazy version)
```

```

12
13 (define Y
14   (lambda (f)
15     (f (Y f))))
16
17 (define almost-factorial
18   (lambda (f)
19     (lambda (n)
20       (if (= n 0)
21           1
22           (* n (f (- n 1)))))))
23
24 (define factorial (Y almost-factorial))
25
26 (prn (factorial 6)) ; infinite loop
27
28 ;; *EOF*

```

### 9.3.2 Using Lazy Evaluation (Racket #lang lazy)

This will work in a lazy language, as shown using the lazy extension in Racket.

```

1 #!/usr/bin/racket
2 ;; y-combinator.scm
3 ;; Mac Radigan
4 ;;
5 ;; copied from http://muanier.livejournal.com/2897.html
6
7 #lang lazy
8
9 ;;; Eliminating (most) explicit recursion (lazy version)
10
11 (define Y
12   (lambda (f)
13     (f (Y f))))
14
15 (define almost-factorial
16   (lambda (f)
17     (lambda (n)
18       (if (= n 0)
19           1

```

```

20      (* n (f (- n 1)))))
21
22      (define factorial (Y almost-factorial))
23
24      (println (factorial 6)) ; 720
25
26      ;; *EOF*

```

```
## ./y-combinator.rkt-lazy
```

```
720
```

## 9.4 Normal Order Y Combinator

The Normal Order Y Combinator will not terminate during applicative order [?].

### 9.4.1 Strict Scheme (Chicken)

```

1  #!/usr/bin/csi -s
2  ;; y-combinator-normal.scm
3  ;; Mac Radigan
4  ;;
5  ;; copied from http://mvanier.livejournal.com/2897.html
6
7  (load "../library/util.scm")
8  (import util)
9
10
11  ;;; The lazy (normal-order) Y combinator
12
13  (define Y
14    (lambda (f)
15      ((lambda (x) (f (x x)))
16       (lambda (x) (f (x x))))))
17
18  (define almost-factorial
19    (lambda (f)
20      (lambda (n)
21        (if (= n 0)
22            1
23            (* n (f (- n 1)))))))

```

```

24
25 (define factorial (Y almost-factorial))
26
27 (prn (factorial 6)) ; infinite loop
28
29 ;; *EOF*

```

---

## 9.4.2 Using Strict Evaluation (Racket)

---

```

1 #!/usr/bin/racket
2 ;; y-combinator-normal.scm
3 ;; Mac Radigan
4 ;;
5 ;; copied from http://mvanier.livejournal.com/2897.html
6
7 #lang lazy
8
9 ;;; The lazy (normal-order) Y combinator
10
11 (define Y
12   (lambda (f)
13     ((lambda (x) (f (x x)))
14      (lambda (x) (f (x x))))))
15
16 (define almost-factorial
17   (lambda (f)
18     (lambda (n)
19       (if (= n 0)
20           1
21           (* n (f (- n 1)))))))
22
23 (define factorial (Y almost-factorial))
24
25 (println (factorial 6)) ; 720
26
27 ;; *EOF*

```

---

## 9.4.3 Using Lazy Evaluation (Racket #lang lazy)

However, it will work under lazy evaluation.



```

1  #!/usr/bin/racket
2  ;; y-combinator-normal.scm
3  ;; Mac Radigan
4  ;;
5  ;; copied from http://mvanier.livejournal.com/2897.html
6
7  #lang lazy
8
9  ;;; The lazy (normal-order) Y combinator
10
11  (define Y
12    (lambda (f)
13      ((lambda (x) (f (x x)))
14       (lambda (x) (f (x x))))))
15
16  (define almost-factorial
17    (lambda (f)
18      (lambda (n)
19        (if (= n 0)
20            1
21            (* n (f (- n 1)))))))
22
23  (define factorial (Y almost-factorial))
24
25  (println (factorial 6)) ; 720
26
27  ;; *EOF*

```

```
## ./y-combinator-normal.rkt-lazy
```

```
720
```

## 9.5 Strict (Applicative-Order) Y Combinator

The Strict (Applicative-Order) Y Combinator can be used with both applicative order and lazy evaluation [?].

### 9.5.1 Strict Scheme (Chicken)

```

1  #!/usr/bin/csi -s
2  ;; y-combinator-struct.scm
3  ;; Mac Radigan
4  ;;
5  ;; copied from http://mvanier.livejournal.com/2897.html
6
7  (load "../library/util.scm")
8  (import util)
9
10
11  ;;; The strict (applicative-order) Y combinator
12
13  (define Y
14    (lambda (f)
15      ((lambda (x) (x x))
16       (lambda (x) (f (lambda (y) ((x x) y)))))))
17
18  (define almost-factorial
19    (lambda (f)
20      (lambda (n)
21        (if (= n 0)
22            1
23            (* n (f (- n 1)))))))
24
25  (define (part-factorial self)
26    (let ((f (lambda (y) ((self self) y))))
27      (lambda (n)
28        (if (= n 0)
29            1
30            (* n (f (- n 1)))))))
31
32  (define factorial (Y almost-factorial))
33
34  (prn (factorial 6)) ; 720
35
36  ;; *EOF*

```

```
## ./y-combinator-strict.scm
```

### 9.5.2 Using Strict Evaluation (Racket)

```
1 #!/usr/bin/racket
2 ;; y-combinator-struct.scm
3 ;; Mac Radigan
4 ;;
5 ;; copied from http://muanier.livejournal.com/2897.html
6
7 #lang racket
8
9 ;;; The strict (applicative-order) Y combinator
10
11 (define Y
12   (lambda (f)
13     ((lambda (x) (x x))
14      (lambda (x) (f (lambda (y) ((x x) y)))))))
15
16 (define almost-factorial
17   (lambda (f)
18     (lambda (n)
19       (if (= n 0)
20           1
21           (* n (f (- n 1)))))))
22
23 (define (part-factorial self)
24   (let ((f (lambda (y) ((self self) y))))
25     (lambda (n)
26       (if (= n 0)
27           1
28           (* n (f (- n 1)))))))
29
30 (define factorial (Y almost-factorial))
31
32 (println (factorial 6)) ; 720
33
34 ;; *EOF*
```

```
## ./y-combinator-strict.rkt
```

```
720
```

### 9.5.3 Using Lazy Evaluation (Racket #lang lazy)

```
1 #!/usr/bin/racket
2 ;; y-combinator-struct.scm
3 ;; Mac Radigan
4 ;;
5 ;; copied from http://muanier.livejournal.com/2897.html
6
7 #lang lazy
8
9 ;;; The strict (applicative-order) Y combinator
10
11 (define Y
12   (lambda (f)
13     ((lambda (x) (x x))
14      (lambda (x) (f (lambda (y) ((x x) y)))))))
15
16 (define almost-factorial
17   (lambda (f)
18     (lambda (n)
19       (if (= n 0)
20           1
21           (* n (f (- n 1)))))))
22
23 (define (part-factorial self)
24   (let ((f (lambda (y) ((self self) y))))
25     (lambda (n)
26       (if (= n 0)
27           1
28           (* n (f (- n 1)))))))
29
30 (define factorial (Y almost-factorial))
31
32 (println (factorial 6)) ; 720
33
34 ;; *EOF*
```

```
## ./y-combinator-strict.rkt-lazy
720
```