

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) [1].

Contents

1	Building Abstractions with Procedures	6
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	12
1.1.7	Example: Square Roots by Newton's Method	12
1.1.8	Procedures as Black-Box Abstractions	16
1.2	Procedures and the Processes They Generate	16
1.2.1	Linear Recursion and Iteration	16
1.2.2	Tree Recursion	25
1.2.3	Orders of Growth	25
1.2.4	Exponentiation	25
1.2.5	Greatest Common Divisors	30
1.2.6	Example: Testing for Primality	30
1.3	Formulating Abstractions with Higher-Order Procedures	30

1.3.1	Procedures as Arguments	30
1.3.2	Constructing Procedures Using Lambda	30
1.3.3	Procedures as General Methods	30
1.3.4	Procedures as Returned Values	30
2	Building Abstractions with Data	32
2.1	Introduction to Data Abstraction	32
2.1.1	Example: Arithmetic Operations for Rational Numbers	32
2.1.2	Abstraction Barriers	34
2.1.3	What Is Meant by Data?	37
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	39
2.2.4	Example: A Picture Language	39
2.3	Symbolic Data	45
2.3.1	Quotation	45
2.3.2	Example: Symbolic Differentiation	51
2.3.3	Example: Representing Sets	51
2.3.4	Example: Huffman Encoding Trees	51
2.4	Multiple Representations for Abstract Data	51
2.4.1	Representations for Complex Numbers	51
2.4.2	Tagged data	51
2.4.3	Data-Directed Programming and Additivity	51
2.5	Systems with Generic Operations	51
2.5.1	Generic Arithmetic Operations	51
2.5.2	Combining Data of Different Types	51
2.5.3	Example: Symbolic Algebra	51
3	Modularity, Objects, and State	53
3.1	Assignment and Local State	53
3.1.1	Local State Variables	53
3.1.2	The Benefits of Introducing Assignment	53

3.1.3	The Costs of Introducing Assignment	53
3.2	The Environment Model of Evaluation	53
3.2.1	The Rules for Evaluation	53
3.2.2	Applying Simple Procedures	53
3.2.3	Frames as the Repository of Local State	53
3.2.4	Internal Definitions	53
3.3	Modeling with Mutable Data	53
3.3.1	Mutable List Structure	53
3.3.2	Representing Queues	53
3.3.3	Representing Tables	53
3.3.4	A Simulator for Digital Circuits	53
3.3.5	Propagation of Constraints	53
3.4	Concurrency: Time Is of the Essence	53
3.4.1	The Nature of Time in Concurrent Systems	53
3.4.2	Mechanisms for Controlling Concurrency	53
3.5	Streams	53
3.5.1	Streams Are Delayed Lists	53
3.5.2	Infinite Streams	53
3.5.3	Exploiting the Stream Paradigm	53
3.5.4	Streams and Delayed Evaluation	53
3.5.5	Modularity of Functional Programs and Modularity of Objects	53
4	Metalinguistic Abstraction	54
4.1	The Metacircular Evaluator	55
4.1.1	The Core of the Evaluator	55
4.1.2	Representing Expressions	55
4.1.3	Evaluator Data Structures	55
4.1.4	Running the Evaluator as a Program	55
4.1.5	Data as Programs	55
4.1.6	Internal Definitions	55
4.1.7	Separating Syntactic Analysis from Execution	55
4.2	Variations on a Scheme – Lazy Evaluation	55
4.2.1	Normal Order and Applicative Order	55

4.2.2	An Interpreter with Lazy Evaluation	55
4.2.3	Streams as Lazy Lists	55
4.3	Variations on a Scheme – Nondeterministic Computing	55
4.3.1	Amb and Search	55
4.3.2	Examples of Nondeterministic Programs	55
4.3.3	Implementing the Amb Evaluator	55
4.4	Logic Programming	55
4.4.1	Deductive Information Retrieval	55
4.4.2	How the Query System Works	62
4.4.3	Is Logic Programming Mathematical Logic?	62
4.4.4	Implementing the Query System	62
5	Computing with Register Machines	64
5.1	Designing Register Machines	64
5.1.1	A Language for Describing Register Machines	64
5.1.2	Abstraction in Machine Design	64
5.1.3	Subroutines	64
5.1.4	Using a Stack to Implement Recursion	64
5.1.5	Instruction Summary	64
5.2	A Register-Machine Simulator	64
5.2.1	The Machine Model	64
5.2.2	The Assembler	64
5.2.3	Generating Execution Procedures for Instructions	64
5.2.4	Monitoring Machine Performance	64
5.3	Storage Allocation and Garbage Collection	64
5.3.1	Memory as Vectors	64
5.3.2	Maintaining the Illusion of Infinite Memory	64
5.4	The Explicit-Control Evaluator	64
5.4.1	The Core of the Explicit-Control Evaluator	64
5.4.2	Sequence Evaluation and Tail Recursion	64
5.4.3	Conditionals, Assignments, and Definitions	64
5.4.4	Running the Evaluator	64
5.5	Compilation	64

5.5.1	Structure of the Compiler	64
5.5.2	Compiling Expressions	64
5.5.3	Compiling Combinations	64
5.5.4	Combining Instruction Sequences	64
5.5.5	An Example of Compiled Code	64
5.5.6	Lexical Addressing	64
5.5.7	Interfacing Compiled Code to the Evaluator	64
6	Appendix A: Modules	65
6.1	util.scm	65
7	Appendix B: Installation Notes	73
7.1	Chicken Scheme	73
8	Appendix C: Notation	74
8.1	Membership	74
8.2	Symbols	74
8.3	Access	74
8.4	Equality	75
8.5	Logic	75
9	Appendix D: Notes	76
9.1	Y-Combiner	76

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*
```

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:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (2)$$

Applied to square root:

$$x_{n+1} = x_n - \frac{x_n^2 - \text{guess}}{2x_n} \quad (3)$$

```
1 #!/usr/bin/csi -s
2 ;; sicp_ch1_e1-7.scm
3 ;; Mac Radigan
4
5 (load "../library/util.scm")
6 (import util)
7
8 ;; Exercise 1.7. The good-enough? test used in computing square roots will not be very effective for
9 ;; finding the square roots of very small numbers. Also, in real computers, arithmetic operations are
10 ;; almost always performed with limited precision. This makes our test inadequate for very large
11 ;; numbers. Explain these statements, with examples showing how the test fails for small and large
12 ;; numbers. An alternative strategy for implementing good-enough? is to watch how guess changes
13 ;; from one iteration to the next and to stop when the change is a very small fraction of the guess.
14   ↪ Design
15 ;; a square-root procedure that uses this kind of end test. Does this work better for small and large
16 ;; numbers?
17 ;; benchmark implementation from book:
```

```

18
19 (define (bm-sqrt-iter guess x)
20   (new-if (good-enough? guess x)
21     guess
22     (bm-sqrt-iter (improve guess x)
23       x)))
24
25 (define (good-enough? guess x)
26   (< (abs (- (square guess) x)) eps))
27
28 (define (improve guess x)
29   (average guess (/ x guess)))
30
31 (define (average x y)
32   (/ (+ x y) 2))
33
34 ;; =====
35 ;; DIRECT
36 ;; =====
37
38 ;;; Many numerical methods exist for accurate square root computations
39 ;;; with fast convergence. Visit the literature for a complete survey.
40 ;;; Here we are employing Newton's method for root finding, with having
41 ;;; "reasonable" convergence.
42
43
44 ;;;  $f(x) = x^2 - s = 0$ 
45
46 ;;; 
$$\frac{f(x[n])}{f'(x[n])} = \frac{x[n]^2 - s}{2 x[n]}$$

47 ;;;  $x[n+1] = x[n] - \frac{f(x[n])}{f'(x[n])} = x[n] - \frac{x[n]^2 - s}{2 x[n]} = \frac{1}{2} (x[n] + s/x[n])$ 
48 ;;;
49
50 ;;; apply Newton's method:
51 ;;;
52 ;;;  $f(x) = x^2 - s = 0$ 
53 ;;;  $f'(x) = 2x$ 
54 ;;;
55 ;;;  $x$  in  $(a, b)$ 
56 ;;;  $y$  in  $(f(a), f(b))$ 
57 ;;;

```

```

58   ;;;     $x[n+1] = x[n] - f(x)/f'(x)$ 
59   ;;;
60   ;;;    initial conditions:
61   ;;;     $a = 0$ 
62   ;;;     $b = x$ 
63   ;;;     $c0 = (b-2)/2$ 
64
65   ;;;    C implementation:
66   ;;;
67   ;;;    double c = (x-0)/2;      // Choose any initial conditions
68   ;;;                                // that satisfy Bolzano's theorem.
69   ;;;                                // (b-a)/2 will work just fine
70   ;;;    //
71   ;;;    const double tol = 0.05; // Set a convergence tolerance based
72   ;;;                                // on your own personal tolerance for
73   ;;;                                // numerical error.
74   ;;;    //
75   ;;;                                // Currently I am favoring speed over
76   ;;;                                // precision.
77   ;;;    //
78   ;;;    double r = c*c - x;
79   ;;;    while(r>tol)
80   ;;;    {
81   ;;;        //  $x[n+1] = x[n] - f(x)/f'(x) = x[n] - (1/2) * (x[n] - s/x[n])$ 
82   ;;;         $c = c - 0.5*(c-x/c)$ ;    // update prediction
83   ;;;         $r = c*c - x$ ;            // find residual
84   ;;;    }
85   ;;;    return c;
86
87   (define (my-sqrt-iter x guess residual)
88     ;;  $x[n+1] = x[n] - f(x)/f'(x) = x[n] - (1/2) * (x[n] - s/x[n])$ 
89     (if (< residual tol)
90         guess ; return
91         (let (
92             (c (- guess (* 0.5 (- guess (/ x guess))))) ;  $c = c - 0.5*(c-x/c)$ ;    // update prediction
93             (r (- (* guess guess) x))                    ;  $r = c*c - x$ ;            // find residual
94             ) ; bind
95             (my-sqrt-iter x c r) ; recursion
96         ) ; let
97     ) ; if

```

```

98  ) ; my-sqrt-iter
99
100 (define (my-sqrt-2 x)
101   (my-sqrt-iter x x x)
102 ) ; my-sqrt
103
104 ;; =====
105 ;; NESTED FUNCTION
106 ;; =====
107
108 (define (my-sqrt x)
109   ;;  $x[n+1] = x[n] - f(x)/f'(x) = x[n] - (1/2) * (x[n] - s/x[n])$ 
110   (define (f x guess residual)
111     (if (< residual tol)
112         guess ; return
113         (let (
114             (c (- guess (* 0.5 (- guess (/ x guess))))) ; c = c - 0.5*(c-x/c); // update prediction
115             (r (- (* guess guess) x)) ; r = c*c - x; // find residual
116             ) ; bind
117             (my-sqrt-iter x c r) ; recursion
118             ) ; let
119             ) ; if
120             ) ; iteration
121   (f x x x)
122 ) ; my-sqrt-iter
123
124 ;; =====
125 ;; TESTS
126 ;; =====
127
128 (define tol 0.0001)
129
130 (bar)
131 (prn "intrinsic:")
132 (prn (sqrt 9) ) ; 3.00009155413138
133 (prn (sqrt (+ 100 37)) ) ; 11.704699917758145
134 (prn (sqrt (+ (sqrt 2) (sqrt 3))) ) ; 1.7739279023207892
135 (prn (square (sqrt 1000)) ) ; 1000.000369924366
136 (hr)
137 (fmt "example 1-7 (base/iter): tolerance ~a%" tol ) ; tolerance 0.0001

```

```

138 (prn (my-sqrt-2 9) ) ; 3.0
139 (prn (my-sqrt-2 (+ 100 37)) ) ; 11.7046999107196
140 (prn (my-sqrt-2 (+ (my-sqrt-2 2) (my-sqrt-2 3))) ) ; 1.77377122818687
141 (prn (square (my-sqrt-2 1000)) ) ; 1000.0
142 (hr)
143 (fmt "example 1-7 (nested): tolerance ~a~%" tol ) ; tolerance 0.0001
144 (prn (my-sqrt 9) ) ; 3.0
145 (prn (my-sqrt (+ 100 37)) ) ; 11.7046999107196
146 (prn (my-sqrt (+ (my-sqrt 2) (my-sqrt 3))) ) ; 1.77377122818687
147 (prn (square (my-sqrt 1000)) ) ; 1000.0
148 (bar)
149
150 ;; *EOF*

```

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

```
=====
intrinsic:
```

```
3.0
11.7046999107196
1.77377122818642
1000.0
-----
```

```
example 1-7 (base/iter): tolerance 0.0001
```

```
3.0
11.7046999107196
1.77377122818687
1000.0
-----
```

```
example 1-7 (nested): tolerance 0.0001
```

```
3.0
11.7046999107196
1.77377122818687
1000.0
=====
```

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*

```

```

## ./sisp_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 1.11: A function f is defined by the rule that

$$f(n) = \begin{cases} n & n < 3 \\ 1f(n-1) + 2f(n-2) + 3f(n-3) & \text{otherwise} \end{cases} \quad (4)$$

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

$$f(n) := s_0 \quad (5)$$

with state transition

$$\begin{matrix} T \\ \left[\begin{array}{l} s_0 \leftarrow s_0 + 2s_1 + 3s_2 \\ s_1 \leftarrow s_0 \\ s_2 \leftarrow s_1 \end{array} \right] \end{matrix} \quad (6)$$

and initial conditions

$$\begin{array}{c} S_0 \\ \left[\begin{array}{c} s_0 := 2 \\ s_1 := 1 \\ s_2 := 0 \end{array} \right] \end{array} \quad (7)$$

Linear Feedback Shift Register (LFSR) representation

$$f(n, \underline{s}) \leftarrow \begin{cases} n_1^{th} \underline{s} & n = 0 \\ f(n-1, n_1^{th} \sigma_1(\underline{s}), [1, 2, 3]) & \text{otherwise} \end{cases} \quad (8)$$

$$x, y \triangleq \sum_k x_k y_k = x_k y^k \quad (9)$$

$$n_k^{th} \triangleq x_k \quad (10)$$

$$\sigma_k(\underline{x}) \triangleq x_{(n+k) \bmod |x|} \forall n \in \underline{x} \quad (11)$$

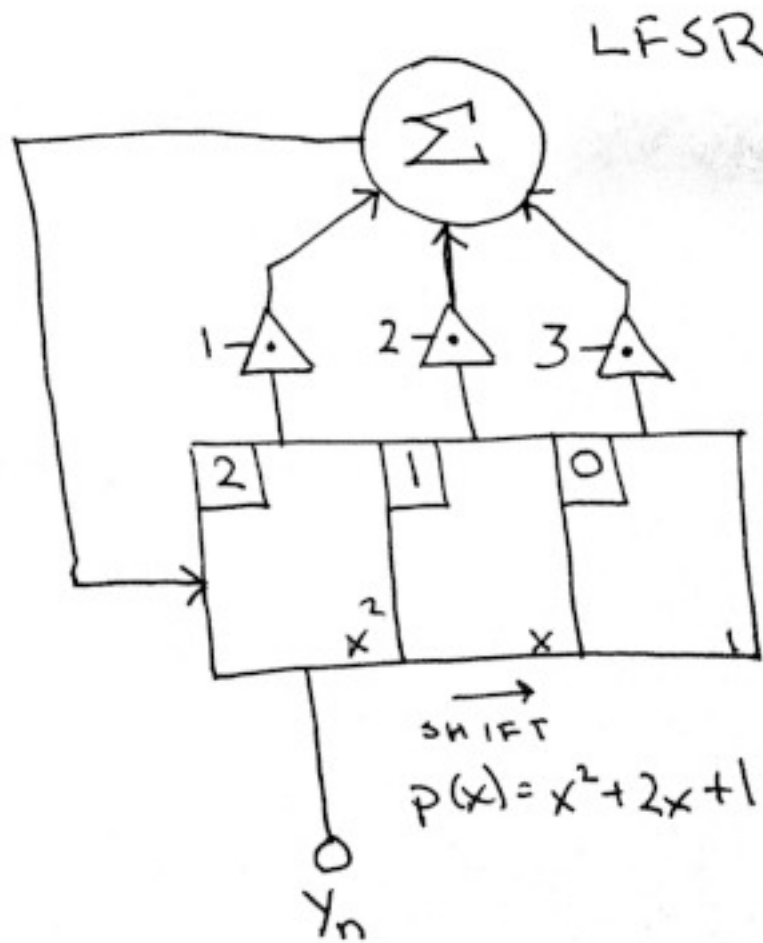


Figure 1: Linear Feedback Shift Register (LFSR)

State Space Representation

$$\mathbf{X}_k = \mathbf{F}\mathbf{X}_{k-1} \quad (12)$$

$$\begin{bmatrix} X_k \\ x'_0 \\ x'_1 \\ x'_2 \end{bmatrix} = \begin{bmatrix} F \\ 1 & 2 & 3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_{k-1} \\ x_0 \\ x_1 \\ x_2 \end{bmatrix} \quad (13)$$

where

$$\mathbf{X}_0 = \begin{bmatrix} X_0 \\ 2 \\ 1 \\ 0 \end{bmatrix} \quad (14)$$

so

$$\mathbf{X}_k = \mathbf{F}\mathbf{X}_{k-1} = \mathbf{F}(\mathbf{F}\mathbf{X}_{k-2}) = \mathbf{F}(\mathbf{F}(\mathbf{F}\mathbf{X}_{k-3})) = \cdots = \mathbf{F}^N \mathbf{X}_0 \quad (15)$$

```

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.)

$$f_{\text{benchmark}}(x, n) = \begin{cases} 1 & \text{if } n \text{ is zero} \\ f_{\text{benchmark}}(x, \frac{n}{2})^2 & \text{if } n \text{ is even, nonzero} \\ f_{\text{benchmark}}(x, n-1)^2 & \text{if } n \text{ is odd} \end{cases} \quad (16)$$

may be restructured as

$$f(x, n) = f_k(x, n, p) \quad (17)$$

where

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

```

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?

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

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  ;; sicp_ch2_e2-44.scm
3  ;; Mac Radigan
4
5  (load "../library/util.scm")
6  (import util)
7
8  (use sicp)
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
Background RRGGBBAA: fffffff00
Area 0:0:256:256 exported to 256 x 256 pixels (90 dpi)
Bitmap saved as: ../figures/sicp_ch2_e2-44.png
=====

```

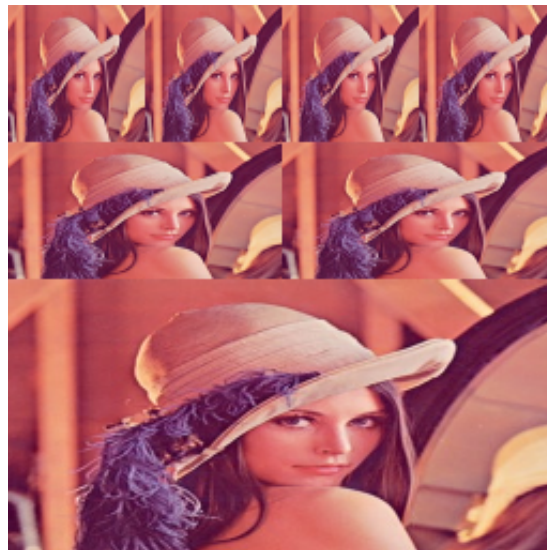


Figure 2: Up Split 2

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
Background RRGGBBAA: ffffff00
Area 0:0:256:256 exported to 256 x 256 pixels (90 dpi)
Bitmap saved as: ../figures/sicp_ch2_e2-45_right.png
-----
up-split lena.jpg := ../figures/sicp_ch2_e2-45_up.png
Background RRGGBBAA: ffffff00
Area 0:0:256:256 exported to 256 x 256 pixels (90 dpi)
Bitmap saved as: ../figures/sicp_ch2_e2-45_up.png
=====

```



Figure 3: Right Split 2



Figure 4: 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*

```

```
## ./sisp_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?*:

$$(equals? \ a \ b) = f(a, b) = \begin{cases} \perp & \text{if } S(a) \oplus S(b) \\ a \stackrel{?}{=} b & \text{if } S(a) \wedge S(b) \\ f(a_A, b_A) f(a_D, b_D) & \text{if } L(a) \wedge L(b) \end{cases} \quad (19)$$

where

$$(20) \quad L(a) \wedge L(b) \implies \neg(S(a) \wedge S(b))$$

To show that all cases have been exhausted (21):

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

(21)

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:

$$(equals? a b) = f(a, b) = f_k(a, b, \top) \quad (22)$$

$$f_k(a, b, p) = \begin{cases} \perp & \text{if } S(a) \oplus S(b) \\ p \wedge (a \stackrel{?}{=} b) & \text{if } S(a) \wedge S(b) \\ f_k(a_D, b_D, f_k(a_A, b_A, p)) & \text{otherwise} \end{cases} \quad (22)$$

```

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   (cond
34     ;;
35     ;; case: null(a) ^ null(b) -> #t (null check)
36     ;;
37     ( (and (null? a) (null? b))
38       #t )
39     ;;
40     ;; case: s(a) ^ s(b) -> #t
41     ;;
42     ( (and (symbol? a) (symbol? b))

```

```

43     #t )
44
45     ;;
46     ;; case: s(a) xor s(b)      ->  a =?= b
47
48     ;;
49     (      (xor (symbol? a) (symbol? b))
50             (eq? a b) )
51
52     ;;
53     ;; case: !( s(a)^s(b) )      ->  f( cdr(a), cdr(b) )
54
55     ;;
56     ;;          i.e. (not (and (symbol? a) (symbol? b)))
57
58     ;;
59     ( else
60         (and (eq? (car a) (car b)) (my-equal-subopt? (cdr a) (cdr b))) )
61     )
62 )
63
64 ;; =====
65 ;; TAIL-RECURSIVE
66 ;; =====
67
68 ;;
69 ;;          { #f                                if      s(a) and s(b)
70 ;; f(a,b,p=#t) = { p ^ a =?= b                    if      s(a) xor s(b)
71 ;;          { f(cdr(a), cdr(b), f(car(a),car(a),p))  if !( s(a) and s(b) )
72
73 ;;
74 (define (my-equal? a b)
75     (define (f a b p)
76         (cond
77             ;;
78             ;; case: null(a) ^ null(b) -> #t (null check)
79
80             ;;
81             ( (and (null? a) (null? b) )
82                 #t )
83
84             ;;
85             ;; case: s(a) ^ s(b)      -> #t
86
87             ;;
88             ( (and (symbol? a) (symbol? b))
89                 #t )
90
91             ;;
92             ;; case: s(a) xor s(b)      ->  a =?= b

```



```

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

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

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

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

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 | ;; sicp_ch4_e4-55.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 "../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*

```

```

## ./sicmp_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  ;; sicmp_ch4_e4-56.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 4.56. 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     Y2
45     yeild
46 )
47 (import scheme chicken)
48 (use extras)
49 (use srfi-1)
50
51 ;;; debug, formatted printing, and assertions
52 (define (br)
53     (format #t "~%"))
54
55 (define (pr x)
56     (format #t "~a" x))
57
58 (define (fmt s x)
59     (format #t s x))
60
61 (define (prn x)
62     (format #t "~a~%" x))
63
64 (define (prnvar name value)
65     (format #t "~a := ~a~%" name value))
66
67 (define (ck name pred? value expect)
68     (cond
69         ( (not (pred? value expect)) (format #t "~a = ~a ; fail expected ~a~%" name value expect) )
70         )
71     (assert (pred? value expect))
72     (format #t "~a = ~a ; ok: expected ~a~%" name value expect)
73 ) ; ck
74
75 ;;; numeric formatting
76 (define (hex x) (format #t "~x~%" x))

```

```

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

```

```

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

```

```

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

```

```

197     (N_cols (cadr dim) ) ; N_cols
198   ) ; local bindings
199   (choose A (range start stride M_rows))
200 ) ; let
201 ) ; mat-col
202
203 ;; selects the kth column from a column-major Iverson matrix
204 ;; NB: dim is a pair ( M_rows , N_cols )
205 (define (mat-row A dim k)
206   (let (
207     (start (* k (cadr dim)) ) ; start := (kth row -1) * M_rows
208     (stride 1 ) ; stride := 1
209     (M_rows (car dim) ) ; M_rows
210     (N_cols (cadr dim) ) ; N_cols
211   ) ; local bindings
212   (choose A (range start stride N_cols))
213 ) ; let
214 ) ; mat-col
215
216 ;; flatmap (map flattened by one level)
217 (define (flatmap f x)
218   (apply append (map f x))
219 ) ; flatmap
220
221 ;; Kroneker combination of vectors a and b
222 (define (kron-comb a b)
223   (flatmap (lambda (ak) (map (lambda (bk) (list ak bk)) a)) b)
224 ) ; kron-comb
225
226 ;; returns a list with elements of x taken from positions ns
227 (define (choose x ns)
228   (map (lambda (k) (list-ref x k)) ns )
229 )
230
231 ;; range sequence generator
232 (define (range start step n)
233   (range-iter '() start step n)
234 )
235
236 ;; Iverson's iota: zero-based sequence of integers from 0..N

```

```

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

```

```
277  ;; hello.scm
278
279  ;; *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

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

(22)

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

(23)

8.2 Symbols

$$\top \triangleq \#t$$

(24)

$$\perp \triangleq \#f$$

(25)

8.3 Access

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

(26)

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

(27)

8.4 Equality

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

(28)

8.5 Logic

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

(29)

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

(30)

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

(31)

$$x \oplus y \triangleq (\neg xy) \vee (x\neg y)$$

(32)

9 Appendix D: Notes

9.1 Y-Combiner

Curry's Y-Combiner [2] is defined as:

$$\mathbf{Y} = \lambda f. (\lambda x. f (xx)) (\lambda x. f (xx)) \quad (33)$$

When applied to a function g , the expansion follows [2]

$$\begin{aligned} \mathbf{Y}g &= (\lambda f. (\lambda x. f (xx)) (\lambda x. f (xx))) g \\ &= (\lambda x. g (xx)) (\lambda x. g (xx)) \\ &= g ((\lambda x. g (xx)) (\lambda x. g (xx))) \\ &= g (\mathbf{Y}g) \end{aligned} \quad (34)$$

References

- [1] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs, 2nd Edition*. Cambridge, MA, USA: MIT Press, 1996.
- [2] Wikipedia, “Fixed-point combinator — Wikipedia, the free encyclopedia,” 2011, [Online; accessed 11-July-2016]. [Online]. Available: http://en.wikipedia.org/Fixed-point_combinator
- [3] J. Bender, “Read scheme,” 2009, [Online; accessed 11-July-2016]. [Online]. Available: <http://readscheme.org>
- [4] —, “Read scheme,” 2009, [Online; accessed 11-July-2016]. [Online]. Available: <http://repository.readscheme.org/ftp>
- [5] K. E. Iverson, “Notation as a tool of thought,” 2006, [Online; accessed 11-July-2016]. [Online]. Available: <http://www.eecg.toronto.edu/~jzhu/csc326/readings/iverson.pdf>
- [6] P. Michaux, “peter.michaux.ca,” 2006, [Online; accessed 11-July-2016]. [Online]. Available: <http://peter.michaux.ca>
- [7] —, “bootstrap-scheme,” 2010, [Online; accessed 11-July-2016]. [Online]. Available: <https://github.com/petermichaux/bootstrap-scheme>