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

1.1 The Elements of Programming

1.1.1 Expressions

```
#!/usr/bin/csi -s
;; sicp_ch1_e1-1.scm
;; Mac Radigan

  (load "../library/util.scm")
  (import util)

;;; 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.

   (prn 10 )
   (prn (+ 5 3 4) )
   (prn (- 9 1) )
   (prn (/ 6 2) )
   (prn (+ (* 2 4) (- 4 6)) )

   (define a 3)
   (define b (+ a 1))

   (prn (+ a b (* a b)) )
   (prn (= a b) )

   (prn (if (and (> b a) (< b (* a b)))
     b
     a) )

   (prn (cond ((= a 4) 6)
     ((= b 4) (+ 6 7 a))
     (else 25)) )

   (prn (+ 2 (if (> b a) b a)) )

   (prn (* (cond ((> a b) a)
     ((< a b) b)
```

```
      (else -1))
      (+ a 1)) )

;; *EOF*
```

```
## ./sicp_ch1_e1-1.scm

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

1.1.2 Naming and the Environment

```
#!/usr/bin/csi -s
;; sicp_ch1_e1-2.scm
;; Mac Radigan

  (load "../library/util.scm")
  (import util)

;;; Exercise 1.2. Translate the following expression into prefix form
;;;
;;;  5 + 1/2 + (2 - (3 - (6 + 1/5) ) )
;;;  ---------------------------------
;;;     3 * ( 6 - 2 ) * ( 2 - 7 )


  (prn
    (/
      (+ 5 1/2 (- 2 (- 3 (+ 6 1/5) ) ) )
      (* 3 (- 6 2) (- 2 7) )
    )
  )

;; *EOF*
```

```
## ./sicp_ch1_e1-2.scm

-0.178333333333333
```

1.1.3 Evaluating Combinations

```
#!/usr/bin/csi -s
;; sicp_ch1_e1-3.scm
;; Mac Radigan

  (load "../library/util.scm")
  (import util)

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

  ;; suares and sum
  (define (my-square x) (map (lambda (x) (* x x)) x) )
  (define (my-sum x) (apply + x) )

  ;; two methods for computing the sum of squares
  (define (ss-1 x) ((compose my-sum my-square) x))
  (define (ss-2 x) (apply + (map (lambda (x) (* x x )) x) ) )

  ;; selection N elements from a list
  (define (take x N)
    (if (> N 1)
      (cons (car x) (take (cdr x) (- N 1)))
      (list (car x))
    )
  )

  ;; selection for top N given operand
  (define (top x pred? N) (take (sort x pred?) N) )

  ;; sum of sqares for top 2 largest elements in list
  (define (topss-1 x) ((compose ss-2 (lambda (x) (top x > 2)) ) x))
  (define (topss-2 x) (ss-2 (top x > 2)) )

  ;; test solution
```

```
(define x '(3 5 2 9 1) )

(prn (topss-1 x) )
(prn (topss-2 x) )

(assert (= (ss-1 x) (ss-2 x) ) )
(assert (= (ss-1 x) (ss-2 x) ) )
(assert (= (topss-1 x) (topss-2 x) ) )
(assert (= (topss-1 x) (topss-2 x) ) )

;; *EOF*
```

```
## ./sicp_ch1_e1-3.scm

106
106
```

1.1.4 Compound Procedures

```
#!/usr/bin/csi -s
;; sicp_ch1_e1-4.scm
;; Mac Radigan

(load "../library/util.scm")
(import util)

;;; 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))

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

(prn (a-plus-abs-b 5 +2) )
(prn (a-plus-abs-b 5 -2) )

;; *EOF*
```

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

7
7
```

1.1.5 The Substitution Model for Procedure Application

```
#!/usr/bin/csi -s
;; sicp_ch1_e1-5.scm
;; Mac Radigan

  (load "../library/util.scm")
  (import util)

;;; 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.)

  (define (p) (p))                  ; infinite recursion

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

  (prn(test 0 (p)) ) ; infinite loop

  (prn (p) )          ; infinite loop

;; *EOF*
```

```
## sicp_ch1_e1-5.scm

; infinite loop (no output available)
```

1.1.6 Conditional Expressions and Predicates

1.1.7 Example: Square Roots by Newton's Method

```
#!/usr/bin/csi -s
;; sicp_ch1_e1-7.scm
;; Mac Radigan

  (load "../library/util.scm")
  (import util)

;;; 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?

;;; benchmark implementation from book:

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

     (define (good-enough? guess x)
       (< (abs (- (square guess) x)) eps))

     (define (improve guess x)
       (average guess (/ x guess)))

     (define (average x y)
       (/ (+ x y) 2))

;;;  Many numerical methods exist for accurate square root computations
;;;  with fast convergence.  Visit the literature for a complete survey.
;;;  Here we are employing Newton's method for root finding, with having
```

```
;;;  "reasonable" convergence.


;;;  f(x) = x^2 - s = 0

;;;                    f(x[n])             x[n]^2 - s
;;;  x[n+1] = x[n] - ---------- = x[n] - -------------- = 1/2 (x[n] + s/x[n])
;;;                    f'(x[n])              2 x[n]

;;;     apply Newton's method:
;;;
;;;     f(x)  = x^2 - s = 0
;;;     f'(x) = 2*x
;;;
;;;     x in (a,b)
;;;     y in (f(a),f(b))
;;;
;;;     x[n+1] = x[n] - f(x)/f'(x)
;;;
;;;     initial conditions:
;;;       a = 0
;;;       b = x
;;;       c0 = (b-2)/2

;;;   C implemention:
;;;
;;;      double c = (x-0)/2;       // Choose any initial conditions
;;;                                // that satisfy Bolzano's theorem.
;;;                                // (b-a)/2 will work just fine
;;;      //
;;;      const double tol = 0.05;  // Set a convergence tolerance based
;;;                                // on your own personal tolerance for
;;;                                // numerical error.
;;;                                //
;;;                                // Currently I am favoring speed over
;;;                                // precision.
;;;      //
;;;      double r = c*c - x;
;;;      while(r>tol)
;;;      {
```

```
;;;        // x[n+1] = x[n] - f(x)/f'(x) = x[n] - (1/2) * (x[n] - s/x[n])
;;;        c = c - 0.5*(c-x/c);    // update prediction
;;;        r = c*c - x;            // find root
;;;      }
;;;      return c;

  (define tol 0.0001)

  (define (my-sqrt-iter x guess root)
    ;; x[n+1] = x[n] - f(x)/f'(x) = x[n] - (1/2) * (x[n] - s/x[n])
    (if (< root tol)
      guess ; return
      (let (
          (c (- guess (* 0.5 (- guess (/ x guess))))) ;; c = c - 0.5*(c-x/c);    // update prediction
          (r (- (* guess guess) x))                   ;; r = c*c - x;            // find root
        ) ; bind
        (my-sqrt-iter x c r) ; recursion
      ) ; let
    ) ; if
  ) ; my-sqrt-iter

  (define (my-sqrt x)
    (my-sqrt-iter x x x)
  ) ; my-sqrt

  (bar)
  (prn "intrinsic:")
  (prn (sqrt 9)                              ) ; 3.00009155413138
  (prn (sqrt (+ 100 37))                     ) ; 11.704699917758145
  (prn (sqrt (+ (sqrt 2) (sqrt 3)))          ) ; 1.7739279023207892
  (prn (square (sqrt 1000))                  ) ; 1000.000369924366
  (hr)
  (fmt "example 1-7: tolerance ~a~%" tol     ) ; tolerance 0.0001
  (prn (my-sqrt 9)                           ) ; 3.0
  (prn (my-sqrt (+ 100 37))                  ) ; 11.7046999107196
  (prn (my-sqrt (+ (my-sqrt 2) (my-sqrt 3))) ) ; 1.77377122818687
  (prn (square (my-sqrt 1000))               ) ; 1000.0
  (bar)

;; *EOF*
```

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

================================================================================
intrinsic:
3.0
11.7046999107196
1.77377122818642
1000.0
--------------------------------------------------------------------------------
example 1-7: 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

```scheme
#!/usr/bin/csi -s
;; sicp_ch1_e1-9.scm
;; Mac Radigan

  (load "../library/util.scm")
  (import util)

;;; 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.

  (define (my-inc x) (begin (prn "inc") (+ x 1) ) ) ; inc with side effects
  (define (my-dec x) (begin (prn "dec") (- x 1) ) ) ; dec with side effects

  (define (recursive-+ a b)
    (if (= a 0) b (my-inc (recursive-+ (my-dec a) b))))

  (define (iterative-+ a b)
    (if (= a 0) b (iterative-+ (my-dec a) (my-inc b)))) ; proper tail recursion

;;; Using the substitution model, illustrate the process gener- ated by each procedure in evaluating (+ 4
↪   5).
```

```scheme
;;; Are these processes iterative or recursive?

  (define a 4)
  (define b 5)

  (prnvar "a" a )
  (prnvar "b" b )
  (prnvar "recursive" (recursive-+ a b) ) ; recursive
  (prnvar "iterative" (iterative-+ a b) ) ; iterative

;; *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
```

Representing State Space Transitions

There are only two hard things in Computer Science: cache invalidation and naming things.

- Phil Karlton

Recursive Definition

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

Direct Iterative Implementation

$$f(n) \coloneqq s_0 \tag{2}$$

with state transition

$$\begin{array}{c} \mathrm{T} \\ \begin{bmatrix} s_0 \leftarrow s_0 + 2s_1 + 3s_2 \\ s_1 \leftarrow s_0 \\ s_2 \leftarrow s_1 \end{bmatrix} \end{array} \tag{3}$$

and initial conditions

$$\begin{array}{c} S_0 \\ \begin{bmatrix} s_0 \coloneqq 2 \\ s_1 \coloneqq 1 \\ s_2 \coloneqq 0 \end{bmatrix} \end{array} \tag{4}$$

Linear Feedback Shift Register (LFSR) representation

$$f(n, \underline{s}) \leftarrow \begin{cases} n_1^{th}\underline{s} & n = 0 \\ f\left(n-1, n_1^{th}\sigma_1(\underline{s}), [1,2,3]\right) & \text{otherwise} \end{cases} \tag{5}$$

$$x, y \triangleq \sum_k x_k y_k = x_k y^k \tag{6}$$

$$n_k^{th} \triangleq x_k \tag{7}$$

$$\sigma_k(\underline{x}) \triangleq x_{(n+k) mod|x|} \forall n \in \underline{x} \tag{8}$$

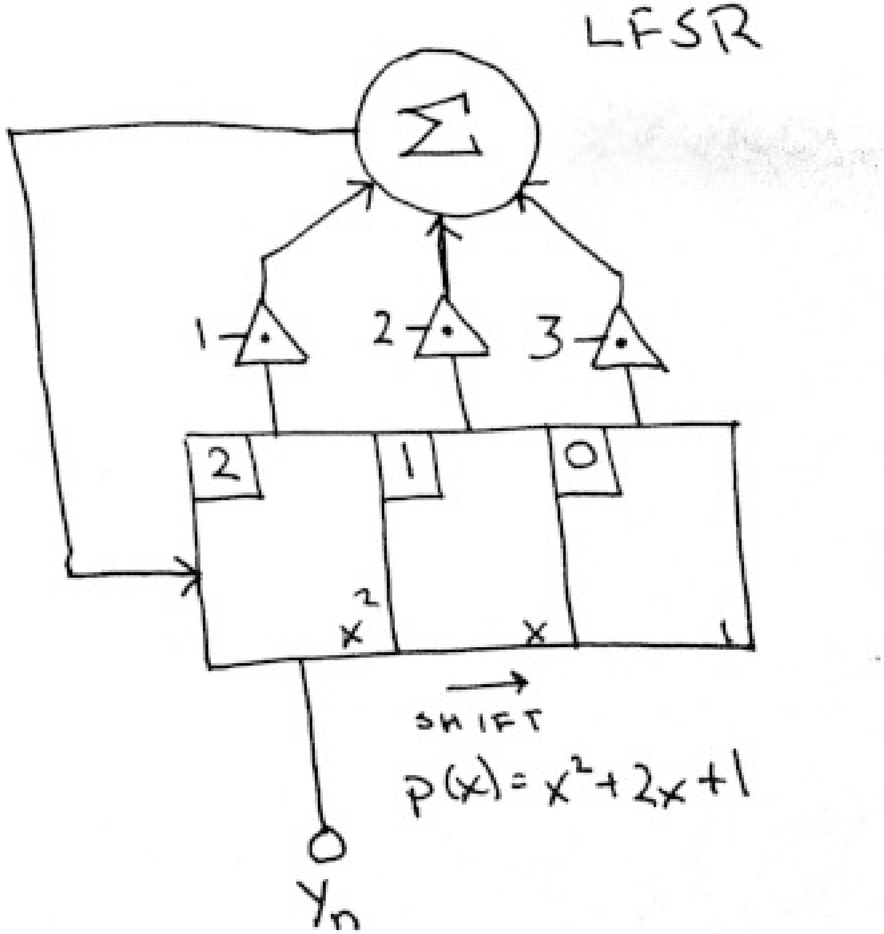

Figure 1: Linear Feedback Shift Register (LFSR)

State Space Representation

$$\mathbf{X}_k = \mathbf{F}\mathbf{X}_{k-1} \tag{9}$$

$$\overset{X_k}{\begin{bmatrix} x_0^{'} \\ x_1^{'} \\ x_2^{'} \end{bmatrix}} = \overset{F}{\begin{bmatrix} 1 & 2 & 3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}} \overset{X_{k-1}}{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}} \tag{10}$$

where

$$\mathbf{X}_0 = \begin{array}{c} X_0 \\ \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \end{array} \tag{11}$$

so

$$\mathbf{X}_k = \mathbf{F}\mathbf{X}_{k-1} = \mathbf{F}\left(\mathbf{F}\mathbf{X}_{k-2}\right) = \mathbf{F}\left(\mathbf{F}\left(\mathbf{F}\mathbf{X}_{k-3}\right)\right) = \cdots = \mathbf{F}^N\mathbf{X}_0 \tag{12}$$

```
#!/usr/bin/csi -s
;; sicp_ch1_e1-1.scm
;; Mac Radigan

  (load "../library/util.scm")
  (import util)

;;; Exercise 1.11: A function f is defined by the rule that
;;;
;;;       { n                      if n<3,
;;; f(n)={ f(n1)+2f(n2)+3f(n3) if n3
;;;
;;; Write a procedure that computes f by means of a recursive process. Write a procedure that computes f by means of an iterative process.
;;;

  ;; ======================================================
  ;; RECURSIVE
  ;; ======================================================

 ;;       { n                            if n<3
 ;; f(n)={
 ;;       { f(n1) + 2f(n2) + 3f(n3)    otherwise

  ;; f(n) recursive form
  (define (f-recursive n)
    (if (< n 3)
      n
      (+ (f-recursive (- n 1)) (* 2 (f-recursive (- n 2)) ) (* 3 (f-recursive (- n 3)) ) )
```

```
  )
)

;; =======================================================
;; DIRECT ITERATIVE
;; =======================================================

;; NB: f(n) = 1*f(n-1) + 2*f(n-2) + 3*f(n-3)
;;
;;        f(n) = s0
;;          state transition
;;            s0 <- s0 + 2*s1 + 3*s2
;;            s1 <- s0
;;            s2 <- s1


;; f(n) direct form
(define (f-direct n)
  (f-direct-iter 2 1 0 n) ; initial state vector [ 0 1 2 ]
)

;; f(n) direct form iteration step
(define (f-direct-iter s0 s1 s2 n)
   (if (< n 3)
     s0
     (f-direct-iter
       (+ (* 1 s0) (* 2 s1) (* 3 s2) )
       s0
       s1
       (- n 1)
     ) ; next
   ) ; iteration test
) ; direct form

;; however, in general, f(n) can be thought of as:

;; =======================================================
;; Linear Feedback Shift Register (LFSR)
;; =======================================================

```

```
;; 1) Linear Feedback Shift Register (LFSR)
;;
;;   f[n] is a Linear Feedback Shift Register (LFSR) operating on the sequence of
;;           previous integers up to n with initial register state x0 := [ 0 1 2 ]
;;           and polynomial coefficients given by a := [ 1 2 3 ]
;;
;;    x[k] = LFSR(x[k-1], a)
;;         = program { circshift(x), x_0 = <x,a> }
;;
;;    f(n) = CAR of x[n]
;;
;;   where
;;
;;     x[0] := [ 0 1 2 ]
;;
;;        a := [ 1 2 3 ]
;;


;; f(n) LFSR form
(define (f-lfsr n)
  (let (
      (a  '(1 2 3)) ; coefficients    a := [ 1 2 3 ]
      (x0 '(2 1 0)) ; initial state  x0 := [ 0 1 2 ]
      (k  (- n 2))  ; k transitions   k := n - 2
    ) ; bindings
    (f-lfsr-iter x0 a k)
  ) ; let
)

;; f(n) LFSR form iteration step
(define (f-lfsr-iter x a k)
   (if (= k 0)
     (car x)
     (f-lfsr-iter (lfsr x a) a (- k 1))
   )
)

;; =======================================================
;; State Space Representation
```

```
;; ======================================================

;; 2) State Space Representaiton
;;
;;   f[n] is the effect of a system up to time n with a given state space
;;           representation F := [ 0 1 0 ; 0 0 1 ; 1  2 3 ], and with
;;           initial conditions x0 := [ 0 1 2 ]
;;
;;    x[k] = F * x[k-1]
;;         = F * ( F * x[k-2] )
;;         = F * ( F * ( F * x[k-3] ) )
;;         = ...
;;         = F^n * x0
;;
;;    f(n) = x[n]
;;
;;   where
;;
;;     x[0] := [ 2 1 0 ]'
;;
;;               [ 1 2 3 ]
;;        F := [ 1 0 0 ]
;;               [ 0 1 0 ]
;;

;; version #1, using Iverson matrix representation

(define (f-ss n)
  (let (
      (t_ref 2)     ; reference time relative to state space
      (x0 '(2 1 0)) ; initial state x0
      (F  '(1 2 3
            1 0 0
            0 1 0 )
      ) ; state transition matrix F
      (dimF '(3 3)) ; F is MxN = 3x3
      (dimX '(3 1)) ; X is Nx1 = 3x1
    ) ; bindings
    (car (f-ss-iter F dimF x0 dimX (- n t_ref)) )
  ) ; let
```

```
  )

  (define (f-ss-iter F dimF x dimX k)
    (if (< k 1)
      x
      ;; x[k] = F * x[k-1]
      (f-ss-iter F dimF (mat-* F dimF x dimX) dimX (- k 1))
    ) ; each
  ) ; ff-ss-iter

  (define n 12)

  (prnvar "recursive f(n)" (f-recursive n) )  ; recursive
  (prnvar "   direct f(n)" (f-direct n) )     ; direct
  (prnvar "     LFSR f(n)" (f-lfsr n) )       ; LFSR
  (prnvar "       SS f(n)" (f-ss n) )         ; state space

;; *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

```
#!/usr/bin/csi -s
;; sicp_ch1_e1-9.scm
;; Mac Radigan

  (load "../library/util.scm")
  (import util)


```

```
;;; 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.)

;; =============================================================================
;; benchmark from book:
;;
;;           {  1              if n is zero
;;  f(x,n) = {  f(x,n/2)^2     if n is even, nonzero
;;           {  f(x,n-1)       if n is odd
;;

  (define (even? n)
    (= (remainder n 2) 0))

  (define (ref-fast-expt b n)
    (cond ((= n 0) 1)
      ((even? n) (square (ref-fast-expt b (/ n 2)) ))
        (else (* b (ref-fast-expt b (- n 1))) )))


;; =============================================================================
;; propagating product up through recursion:
;;
;;             {  p              if n is zero
;;  f(x,n,p) = {  f(x,n/2,p)     if n is even, nonzero
;;             {  f(x,n-1,x*p)   if n is odd
;;

  (define (fast-expt-iter b n p)
    (cond ((= n 0) p)
      ((even? n) (fast-expt-iter (* b b) (/ n 2) p) )
        (else (fast-expt-iter b (- n 1) (* b p)) )))

  (define (sep-fast-expt b n)
```

```
    (fast-expt-iter b n 1))


;; ============================================================================
;; encapsulated as a single function

  (define (fast-expt b n)
    ;;              {  p              if n is zero
    ;;  f(x,n,p) = {  f(x,n/2,p)     if n is even, nonzero
    ;;              {  f(x,n-1,x*p)   if n is odd
    (define (f b n p)
      (cond ((= n 0) p)
        ((even? n) (f (* b b) (/ n 2) p) )
          (else (f b (- n 1) (* b p)) )))
    (f b n 1) ; call
  )


;; ============================================================================
;; applying self-referencing lambdas

  (define (sr-fast-expt b n)
    (define f (lambda (@f)
        (lambda (b n p)
          (cond ((= n 0) p )
                ((even? n) ((f f) (* b b) (/ n 2) p) )
                (else ((f f) b (- n 1) (* b p)) ))
        ) ; f(x,n)
    )) ; self
    ((f f) b n 1)
  )


;; ============================================================================
;; with hygenic macros

 (define-syntax call
   (syntax-rules ()
     ((_ f)
       (f f))))
```

```
(define-syntax fn
  (syntax-rules ()
    ((_ signature self fn-base fn-iter)
      (define signature
        (define self (lambda (@self) fn-iter))
        fn-base
      ) )))

(fn (mac-fast-expt b n) f
  ;; f(b,n,1)
  ((call f) b n 1)
  ;; f(b,n,p)
  (lambda (b n p)
    (cond ((= n 0) p )
          ((even? n) ((call f) (* b b) (/ n 2) p) )
          (else ((call f) b (- n 1) (* b p)) ))
  ) ; f(x,n)
)


;; ============================================================================
;; test:
(define b 2)
(define n 8)

(bar)
(prn "intrinsic:")
(prn (expt b n)) ;
(hr)
(prn "reference:")
(prn (ref-fast-expt b n)) ;
(hr)
(prn "example 1-16: (separate functions)")
(prn (sep-fast-expt b n)) ;
(hr)
(prn "example 1-16: (nested functions)")
(prn (fast-expt b n)) ;
(hr)
(prn "example 1-16 (self-referencing lambdas):")
```

```
  (prn (sr-fast-expt b n)) ;
  (hr)
  (prn "example 1-16 (using macros):")
  (prn (mac-fast-expt b n) )
  (bar)

;; *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

```
#!/usr/bin/csi -s
;; sicp_ch1_e1-42.scm
;; Mac Radigan

```

```
  (load "../library/util.scm")
  (import util)

;;; Exercise 1.42. Let f and g be two one-argument functions. The composition f after g is defined to be
;;; the function x 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)

  ;;; from util.scm
  ; (define (square x) (map (lambda (x) (* x x)) x) )
  ; (define (inc x) (+ x 1))
  ; (define ((compose f g) x) (f (g x)))

  (prn ((compose square inc) 6) ) ; 49

;; *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

```
#!/usr/bin/csi -s
;; sicp_ch2_e1-1.scm
;; Mac Radigan

  (load "../library/util.scm")
  (import util)

;;; 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.

  (define (add-rat x y)
    (make-rat (+ (* (numer x) (denom y))
                 (* (numer y) (denom x)))
              (* (denom x) (denom y))))

  (define (sub-rat x y)
    (make-rat (- (* (numer x) (denom y))
                 (* (numer y) (denom x)))
              (* (denom x) (denom y))))

  (define (mul-rat x y)
    (make-rat (* (numer x) (numer y))
              (* (denom x) (denom y))))

  (define (div-rat x y)
    (make-rat (* (numer x) (denom y))
              (* (denom x) (numer y))))

  (define (equal-rat? x y)
    (= (* (numer x) (denom y))
       (* (numer y) (denom x))))

  (define (signum x)
```

```
    (if (> x 0) +1 -1) )

  (define (make-rat num denom)
    (cons (* (signum (* num denom)) (abs (/ num (gcd num denom)))) (abs (/ denom (gcd num denom)))) )

  (define (numer x)
    (car x))

  (define (denom x)
    (cdr x))

  (define x1 (make-rat  1  2))  ; x1 =  1/2
  (define x2 (make-rat  1  4))  ; x2 =  1/4
  (define x3 (make-rat  2  4))  ; x3 =  2/4
  (define x4 (make-rat -1  2))  ; x4 = -1/2
  (define x5 (make-rat  1 -4))  ; x5 = -1/4
  (define x6 (make-rat -2 -4))  ; x6 =  2/4

  (prvar "x1 = 1/2 " x1)  ;  1/2
  (prvar "x2 = 1/4 " x2)  ;  1/4
  (prvar "x3 = 2/4 " x3)  ;  2/4
  (prvar "x4 = -1/2 " x4) ; -1/2
  (prvar "x5 = -1/4 " x5) ; -1/4
  (prvar "x6 = 2/4 " x6)  ;  2/4

  (ck "x1*x2" equal-rat? (mul-rat x1 x2) (make-rat 1 8))  ;  1/2 *  1/4 = 1/8
  (ck "x1*x4" equal-rat? (mul-rat x1 x4) (make-rat -1 4)) ;  1/2 * -1/2 = 1/4
  (ck "x4*x5" equal-rat? (mul-rat x4 x5) (make-rat 1 8))  ; -1/2 * -1/4 = 1/8
  (ck "x5*x6" equal-rat? (mul-rat x5 x6) (make-rat -1 8)) ; -1/4 *  2/4 = -1/8

;; *EOF*
```

./sicp_ch2_e1-1.scm

2.1.2 Abstraction Barriers

```
#!/usr/bin/csi -s
;; sicp_ch2_e1-1.scm
;; Mac Radigan
```

```scheme

(load "../library/util.scm")
(import util)

;;; 2.1.2 Abstraction Barriers
;;; Before continuing with more examples of compound data and data abstraction, let us consider some of
;;; the issues raised by the rational-number example. We defined the rational-number operations in terms
;;; of a constructor make-rat and selectors numer and denom. In general, the underlying idea of data
;;; abstraction is to identify for each type of data object a basic set of operations in terms of which
↪ all
;;; manipulations of data objects of that type will be expressed, and then to use only those operations
↪ in
;;; manipulating the data.
;;; We can envision the structure of the rational-number system as shown in figure 2.1. The horizontal
;;; lines represent abstraction barriers that isolate different levels of the system. At each level, the
;;; barrier separates the programs (above) that use the data abstraction from the programs (below) that
;;; implement the data abstraction. Programs that use rational numbers manipulate them solely in terms of
;;; the procedures supplied for public use by the rational-number package: add-rat, sub-rat,
;;; mul-rat, div-rat, and equal-rat?. These, in turn, are implemented solely in terms of the
;;; constructor and selectors make-rat, numer, and denom, which themselves are implemented in
;;; terms of pairs. The details of how pairs are implemented are irrelevant to the rest of the
;;; rational-number package so long as pairs can be manipulated by the use of cons, car, and cdr. In
;;; effect, procedures at each level are the interfaces that define the abstraction barriers and connect
↪ the
;;; different levels.

;;; Figure 2.1: Data-abstraction barriers in the rational-number package.
;;; This simple idea has many advantages. One advantage is that it makes programs much easier to
;;; maintain and to modify. Any complex data structure can be represented in a variety of ways with the
;;; primitive data structures provided by a programming language. Of course, the choice of representation
;;; influences the programs that operate on it; thus, if the representation were to be changed at some
↪ later
;;; time, all such programs might have to be modified accordingly. This task could be time-consuming
;;; and expensive in the case of large programs unless the dependence on the representation were to be
;;; confined by design to a very few program modules.
;;; For example, an alternate way to address the problem of reducing rational numbers to lowest terms is
;;; to perform the reduction whenever we access the parts of a rational number, rather than when we
;;; construct it. This leads to different constructor and selector procedures:
;;; (define (make-rat n d)
;;; (cons n d))
```

```scheme
;;; (define (numer x)
;;; (let ((g (gcd (car x) (cdr x))))
;;; (/ (car x) g)))
;;; (define (denom x)
;;; (let ((g (gcd (car x) (cdr x))))
;;; (/ (cdr x) g)))
;;; The difference between this implementation and the previous one lies in when we compute the gcd. If
;;; in our typical use of rational numbers we access the numerators and denominators of the same rational
;;; numbers many times, it would be preferable to compute the gcd when the rational numbers are
;;; constructed. If not, we may be better off waiting until access time to compute the gcd. In any case,
;;; when we change from one representation to the other, the procedures add-rat, sub-rat, and so on
;;; do not have to be modified at all.
;;; Constraining the dependence on the representation to a few interface procedures helps us design
;;; programs as well as modify them, because it allows us to maintain the flexibility to consider alternate
;;; implementations. To continue with our simple example, suppose we are designing a rational-number
;;; package and we cant decide initially whether to perform the gcd at construction time or at selection
;;; time. The data-abstraction methodology gives us a way to defer that decision without losing the ability
;;; to make progress on the rest of the system.

;;; 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, youll need a way to print points:

 (define (print-point p)
   (newline)
   (display "(")
   (display (x-point p))
   (display ",")
   (display (y-point p))
   (display ")")
   (newline)
```

```scheme
)

;; point construct
(define (make-point x y)
  (cons x y))

(define (x-point pt)
  (car pt))

(define (y-point pt)
  (cdr pt))

(define (equal-point? pt1 pt2)
  (and
    (= (x-point pt1) (x-point pt2))
    (= (y-point pt1) (y-point pt2))
  )
)

;; segment construct
(define (make-segment pt1 pt2)
  (cons pt1 pt2))

(define (start-segment seg)
  (car seg))

(define (end-segment seg)
  (cdr seg))

(define (midpoint-segment seg)
  (make-point
    (/ (+ (x-point (start-segment seg)) (x-point (end-segment seg)) ) 2)
    (/ (+ (y-point (start-segment seg)) (y-point (end-segment seg)) ) 2)
  )
)

;; test constructs
(define pt-00 (make-point 0 0)) ; (0, 0) origin
(define pt-10 (make-point 1 0)) ; (1, 0)
(define pt-01 (make-point 0 1)) ; (0, 1)
```

```
  (define s-x  (make-segment pt-00 pt-10)) ; (0,0) -> (0,1)
  (define s-y  (make-segment pt-00 pt-01)) ; (0,0) -> (1,0)
  (define s-xy (make-segment pt-10 pt-01)) ; (1,0) -> (1,0)

  (define pt-mid (midpoint-segment s-xy)) ; (0.5,0.5)

  (print-point pt-mid)

  (ck "midpoint" equal-point? pt-mid (make-point 0.5 0.5)) ; -1/4 *  2/4 = -1/8

;; *EOF*
```

```
## ./sicp_ch2_e1-2.scm


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

```
#!/usr/bin/csi -s
;; sicp_ch2_e1-1.scm
;; Mac Radigan

  (load "../library/util.scm")
  (import util)

;;; 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?

;; *EOF*
```

```
## ./sicp_ch2_e1-3.scm
```

2.1.3 What Is Meant by Data?

2.1.4 Extended Exercise: Interval Arithmetic

2.2 Hierarchical Data and the Closure Property

2.2.1 Representing Sequences

2.2.2 Hierarchical Structures

2.2.3 Sequences as Conventional Interfaces

2.2.4 Example: A Picture Language

2.3 Symbolic Data

2.3.1 Quotation

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

```scheme
#!/usr/bin/csi -s
;; run-query.scm
;; Mac Radigan

  (use sicp sicp-eval sicp-eval-anal sicp-streams)
  (load "./ch4-query.scm")
  (define false #f)
  (define true #t)
  (initialize-data-base microshaft-data-base)
  (query-driver-loop)

;; *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

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

```
#!/usr/bin/csi -s
;; util.scm
;; Mac Radigan

  (module util (
      bind
      bar
      bin
      br
      but-last
      ck
      compose
      dec
      dotprod
      flatmap
      fmt
      hr
      hex
      inc
      my-iota
      join
      kron-comb
      lfsr
      mat-*
      mat-col
      mat-row
      mod
      my-last
      nth
      oct
      permute
      pr
      prn
      prnvar
      my-reverse
      range
```

```
  rotate-right
  rotate-left
  rotate
  square
  sum
  yeild
)
(import scheme chicken)
(use extras)
(use srfi-1)

;;; debug, formatted printing, and assertions
(define (br)
  (format #t "~%"))

(define (pr x)
  (format #t "~a" x))

(define (fmt s x)
  (format #t s x))

(define (prn x)
  (format #t "~a~%" x))

(define (prnvar name value)
  (format #t "~a := ~a~%" name value))

(define (ck name pred? value expect)
  (cond
    ( (not (pred? value expect)) (format #t "~a = ~a   ; fail expected ~a~%" name value expect) )
  )
  (assert (pred? value expect))
  (format #t "~a = ~a   ; ok: expected ~a~%" name value expect)
) ; ck

;;; numeric formatting
(define (hex x)  (format #t "~x~%" x))
(define (bin x)  (format #t "~b~%" x))
(define (oct x)  (format #t "~o~%" x))

```

```
;;; delimiters
(define (bar)    (format #t "~a~%" (make-string 80 #\=)))
(define (hr)     (format #t "~a~%" (make-string 80 #\-)))

;;; returns the nth element of list x
(define (nth x n)
  (if (= n 1)
    (car x)
    (nth (cdr x) (- n 1))
  ) ; if last iter
) ; nth

;;; returns the inner product <u,v>
(define (dotprod u v)
  (apply + (map * u v))
)

;;; returns x mod n
(define (mod x n)
  (- x (* n (floor (/ x n))))
)

;;; the permutation x by p
(define (permute x p)
  (map (lambda (pk) (nth x pk)) p)
)

;;; circular shift (left) of x by n
(define (rotate-left x n)
  (if (< n 1)
    x
    (rotate-left (append (cdr x) (list (car x))) (- n 1))
  ) ; if last iter
) ; rotate-left

;;; circular shift (right) of x by n
(define (rotate-right x n)
  (if (< n 1)
    x
    (rotate-right
```

```scheme
        (append (list (my-last x)) (but-last x))
        (- n 1)
      ) ; call
    ) ; if last iter
  ) ; rotate-right

  ;;; circular shift of x by n
  (define (rotate x n)
    (cond
      ((= n 0) x)
      ((> n 0) (rotate-right x n))
      ((< n 0) (rotate-left x (abs n)))
    )
  )

  ;;; return all but last element in list
  (define (but-last x)
    (if (null? x)
      (list)
      (if (null? (cdr x))
        (list)
        (cons (car x) (but-last (cdr x)))
      ) ; end if list contains only one element
    ) ; end if list null
  )

  ;;; return the last element in list
  (define (my-last x)
    (if (null? x)
      #f
      (if (null? (cdr x))
        (car x)
        (my-last (cdr x))
      ) ; end if list contains only one element
    ) ; end if list null
  )

  ;; composition
  (define ((compose f g) x) (f (g x)))
```

```
;; my-reverse
(define (my-reverse x)
  (if (null? x)
    (list)
    (append (reverse (cdr x)) (list (car x)))
  )
)

;; Linear Feedback Shift Register (LFSR)
;;   given initial state x[k-1] and coefficients a
;;   return next state x[k]
(define (lfsr x a)
  (append (list (dotprod x a)) (cdr (rotate x +1)) ) ; next state x[k]
) ; lfsr

;; matrix multiplication of column-major Iverson matrices
(define (mat-* A dimA B dimB)
  (let (
      ; A_mxn * B_nxk = C_nxk
      (M_rows (cadr dimA) ) ; M_rows
      (N_cols (cadr dimB) ) ; N_cols
    ) ; local bindings
    (map (lambda (rc)
        (dotprod (mat-row A dimA (car rc)) (mat-col B dimB (cadr rc)) )
      )
      (kron-comb (my-iota N_cols) (my-iota M_rows))
    )
  ) ; let
) ; mat-*

;; selects the kth column from a column-major Iverson matrix
;;   NB:  dim is a pair ( M_rows , N_cols )
(define (mat-col A dim k)
  (let (
      (start  k          ) ; start  := kth column
      (stride (cadr dim) ) ; stride := N_cols
      (M_rows (car dim)  ) ; M_rows
      (N_cols (cadr dim) ) ; N_cols
    ) ; local bindings
    (choose A (range start stride M_rows))
```

```
    ) ; let
  ) ; mat-col

  ;; selects the kth column from a column-major Iverson matrix
  ;;   NB:  dim is a pair ( M_rows , N_cols )
  (define (mat-row A dim k)
    (let (
        (start  (* k (cadr dim))      ) ; start  := (kth row -1) * M_rows
        (stride 1                     ) ; stride := 1
        (M_rows (car dim)             ) ; M_rows
        (N_cols (cadr dim)            ) ; N_cols
      ) ; local bindings
      (choose A (range start stride N_cols))
    ) ; let
  ) ; mat-col

  ;; flatmap (map flattened by one level)
  (define (flatmap f x)
    (apply append (map f x))
  ) ; flatmap

  ;; Kroneker combination of vectors a and b
  (define (kron-comb a b)
    (flatmap (lambda (ak) (map (lambda (bk) (list ak bk)) a)) b)
  ) ; kron-comb

  ;; returns a list with elements of x taken from positions ns
  (define (choose x ns)
    (map (lambda (k) (list-ref x k)) ns )
  )

  ;; range sequence generator
  (define (range start step n)
    (range-iter '() start step n)
  )

  ;; Iverson's iota: zero-based sequence of integers from 0..N
  (define (my-iota n)
    (range 0 1 n)
  )
```

```

    ;; local scope: range sequence generator helper
    (define (range-iter x val step n)
      (if (< n 1)
        x
        (range-iter (append x (list val)) (+ val step) step (- n 1) ) ; x << val + step
      )
    )

    ;; square, sum, inc, and dec
    (define (square x) (* x x))
    (define (sum x) (apply + x) )
    (define (inc x) (+ x 1))
    (define (dec x) (- x 1))

    ;;; data transformations: bind, join, yeild
    (define (bind f x) (join (map f x)))
    (define (join x) (apply append '() x))
    (define yeild list)

  ) ; module util

  ;; hello.scm

;; *EOF*
```

7 Appendix B: Installation Notes

7.1 Chicken Scheme

```bash
#!/bin/bash
## apt-install.sh
## Mac Radigan
apt install chicken-bin -y
## *EOF*
```

```bash
#!/bin/bash
## yum-install.sh
## Mac Radigan
yum -y install chicken
## *EOF*
```

```bash
#!/bin/bash
## brew-install.sh
## Mac Radigan
brew install chicken
## *EOF*
```

```bash
#!/bin/bash
## chicken-install.sh
## Mac Radigan
chicken-install sicp
# *EOF*
```

References

[1] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs, 2nd Edition.* Cambridge, MA, USA: MIT Press, 1996.