Structure and Interpretation of Computer Programs (SICP) worked examples

Mac Radigan

Abstract

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

Contents

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.

```
#!/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

Exercise 1.2. Translate the following expression into prefix form

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

```
#!/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

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

```
#!/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
```

```scheme
  (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

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))

```
#!/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

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.)

```
#!/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*
```

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

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

Eva demonstrates the program for Alyssa:

```
(new-if (= 2 3) 0 5)
  5
(new-if (= 1 1) 0 5)
  0
```

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

```
(new-if (= 1 1) 0 5)
  (define (sqrt-iter guess x)
    (new-if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))
```

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

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

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

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

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

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

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

1.1.6 Conditional Expressions and Predicates

1.1.7 Example: Square Roots by Newton's Method

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

Newton's method:

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

Applied to square root:

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

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

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

```scheme

;;; 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_e1-7.scm

1.1.8 Procedures as Black-Box Abstractions

1.2 Procedures and the Processes They Generate

1.2.1 Linear Recursion and Iteration

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

```
#!/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).
;;; 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
```

```scheme

;; *EOF*
```

```
## ./sicp_ch1_e1-9.scm

a := 4
b := 5
dec
dec
dec
dec
inc
inc
inc
inc
recursive := 9
dec
inc
dec
inc
dec
inc
dec
inc
iterative := 9
```

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

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

What are the values of the following expressions?

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

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

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

```scheme
#!/usr/bin/csi -s
;; sicp_ch1_e1-10.scm
;; Mac Radigan

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

;; Exercise 1.10: The following procedure computes a mathematical function called Ackermann's function.
;;
;;   (define (A x y)
;;     (cond ((= y 0) 0)
;;       ((= x 0) (* 2 y))
;;       ((= y 1) 2)
;;     (else (A (- x 1) (A x (- y 1))))))

;; What are the values of the following expressions?
;;   (A 1 10)
;;   (A 2 4)
;;   (A 3 3)
;;
;; Consider the following procedures, where A is the procedure defined above:
;;   (define (f n) (A 0 n))
;;   (define (g n) (A 1 n))
;;   (define (h n) (A 2 n))
;;   (define (k n) (* 5 n n))

;; Give concise mathematical definitions for the functions computed by the procedures f , g , and h for positive integer values of n. For example, (k n) computes 5n 2 .

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

  (bar)
  (prnvar "(A 1 10)" (A 1 10))
  (prnvar "(A 2 4)"  (A 2  4))
  (prnvar "(A 3 3)"  (A 3  3))
  (hr)
```

```
  (prn "(f n) := 2*n")
  (prn "(g n) := 2")
  (prn "(h n) := ")
  (prn "(k n) := 5*n^2")
  (bar)

;; *EOF*
```

```
## ./sicp_ch1_e1-10.scm

================================================================================
(A 1 10) := 1024
(A 2 4) := 65536
(A 3 3) := 65536
--------------------------------------------------------------------------------
(f n) := 2*n
(g n) := 2
(h n) :=
(k n) := 5*n^2
================================================================================
```

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

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

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

Representing State Space Transitions

Direct Iterative Implementation

$$f(n) \coloneqq s_0 \tag{7}$$

with state transition

$$\begin{array}{c} \mathrm{T} \\ \begin{bmatrix} s_0 \leftarrow s_0 + 2s_1 + 3s_2 \\ s_1 \leftarrow s_0 \\ s_2 \leftarrow s_1 \end{bmatrix} \end{array} \tag{8}$$

and initial conditions

$$\begin{array}{c} S_0 \\ \left[\begin{array}{c} s_0 \coloneqq 2 \\ s_1 \coloneqq 1 \\ s_2 \coloneqq 0 \end{array}\right] \end{array} \tag{9}$$

Linear Feedback Shift Register (LFSR) representation

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

$$x, y \triangleq \sum_k x_k y_k = x_k y^k \tag{11}$$

$$n_k^{th} \triangleq x_k \tag{12}$$

$$\sigma_k\left(\underline{\mathrm{x}}\right) \triangleq x_{(n+k)mod|x|} \forall n \in \underline{\mathrm{x}} \tag{13}$$

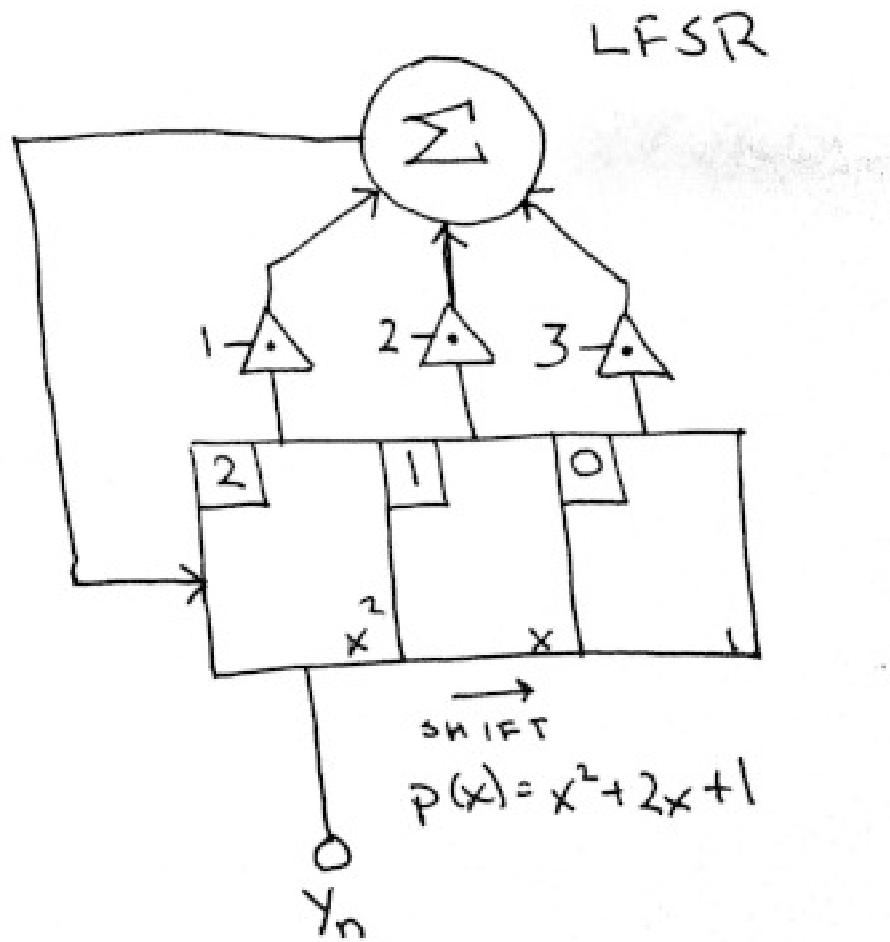

Figure 1: Linear Feedback Shift Register (LFSR)

State Space Representation

$$\mathbf{X}_k = \mathbf{F}\mathbf{X}_{k-1} \tag{14}$$

$$\begin{array}{ccccc} X_k & & F & & X_{k-1} \\ \begin{bmatrix} x_0' \\ x_1' \\ x_2' \end{bmatrix} & = & \begin{bmatrix} 1 & 2 & 3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} & & \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \end{array} \tag{15}$$

where

$$\mathbf{X}_0 = \begin{array}{c} X_0 \\ \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \end{array} \tag{16}$$

so

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

```
#!/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(n-1)+2f(n-2)+3f(n-3) if n>3
;;;
;;; 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(n-1) + 2f(n-2) + 3f(n-3)   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

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 $\left(b^{\frac{n}{2}}\right)^2 = \left(b^2\right)^{\frac{n}{2}}$, 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 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.)

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

may be restructured as

$$f\left(x,n\right)=f_{k}\left(x,n,p\right) \tag{19}$$

where

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

```
#!/usr/bin/csi -s
;; sicp_ch1_e1-16.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
```

```scheme
;;  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) )
```

```scheme
        (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

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, $((compose\ square\ inc)\ 6)$

```
#!/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

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.

```
#!/usr/bin/csi -s
;; sicp_ch2_e2-1.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)
```

```scheme
    (= (* (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_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:

```
#!/usr/bin/csi -s
;; sicp_ch2_e2-2.scm
;; Mac Radigan

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

;;; 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:

  (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_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?

```
#!/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_e2-3.scm
```

2.1.3 What Is Meant by Data?

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

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

$$\begin{aligned} car\,(z) &\triangleq \lambda z.z\lambda pq.p \\ &\rightarrow_\beta \lfloor^{cons(x,y)}/_z\rfloor car\,(z) \\ &= (\lambda m.mxy)\,\lambda pq.p \\ &\rightarrow_\beta (\lambda pq.p)\,xy \\ &\rightarrow_\beta x \end{aligned} \tag{22}$$

$$\begin{aligned} cdr\,(z) &\triangleq \lambda z.z\lambda pq.q \\ &\rightarrow_\beta \lfloor^{cons(x,y)}/_z\rfloor cdr\,(z) \\ &= (\lambda m.mxy)\,\lambda pq.q \\ &\rightarrow_\beta (\lambda pq.q)\,xy \\ &\rightarrow_\beta y \end{aligned} \tag{23}$$

```
#!/usr/bin/csi -s
;; sicp_ch2_e2-4.scm
;; Mac Radigan

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

```scheme
  (import util)

;;; Exercise 2.4. Here is an alternative procedural representation of
;;; pairs. For this representation, verify that (car (cons x y)) yields
;;; x for any objects x and y.

;;; What is the corresponding definition of cdr? (Hint: To verify that
;;; this works, make use of the substitution model of section 1.1.5.)

  ;; alternate cons
  (define (my-cons x y)
    (lambda (m) (m x y)))

  ;; alternate cdr
  (define (my-car z)
    (z (lambda (p q) p)))

  ;; alternate car
  (define (my-cdr z)
    (z (lambda (p q) q)))

;; =======================================================
;; TESTS
;; =======================================================

  (define p (my-cons 'a 'b))

  (bar)
  (prnvar "(cons 'a 'b)" p)
  (prnvar "(car (cons 'a 'b)" (my-car p))
  (prnvar "(cdr (cons 'a 'b)" (my-cdr p))
  (bar)

;; *EOF*
```

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

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

2.1.4 Extended Exercise: Interval Arithmetic

2.2 Hierarchical Data and the Closure Property

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

```
#!/usr/bin/csi -s
;; sicp_ch2_e2-17.scm
;; Mac Radigan

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

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

  (define (last-pair x)
    (if (null? x)
      #f ; case empty list
      (if (null? (cdr x))
         (car x)
         (last-pair (cdr x))
      )
    )
  )

;; =======================================================
;; TESTS
;; =======================================================

  (bar)
  (prnvar "(23 72 149 34)" (last-pair (list 23 72 149 34)) ) ; 34
  (prnvar "(            )" (last-pair (list)) )              ; #f
  (bar)

;; *EOF*
```

```
## ./sicp_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:

```
#!/usr/bin/csi -s
;; sicp_ch2_e2-17.scm
;; Mac Radigan

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

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

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

;; ======================================================
;; TESTS
;; ======================================================

  (bar)
  (prnvar "(1 4 9 16 25)" (my-reverse (list 1 4 9 16 25)) ) ; (25 16 9 4 1)
  (prnvar "(           )" (my-reverse (list)) )             ; #f
  (bar)

;; *EOF*
```

```
## ./sicp_ch2_e2-18.scm

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

2.2.1 Representing Sequences

2.2.2 Hierarchical Structures

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

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

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

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


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

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

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

```
#!/usr/bin/csi -s
;; sicp_ch2_e2-21.scm
;; Mac Radigan

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

```scheme

;;; Exercise 2.21. The procedure square-list takes a list of numbers as argument and returns a list
;;; of the squares of those numbers.
;;;
;;;   (square-list (list 1 2 3 4))
;;;     (1 4 9 16)
;;;
;;; Here are two different definitions of square-list. Complete both of them by filling in the missing
;;; expressions:
;;;
;;;   (define (square-list items)
;;;     (if (null? items)
;;;       nil
;;;     (cons <??> <??>)))
;;;
;;;   (define (square-list items)
;;;     (map <??> <??>))

  (define (square-list-1 items)
    (if (null? items)
      nil
      (cons (expt (car items) 2) (square-list-1 (cdr items)))
    )
  )

  (define (square-list-2 items)
    (map (lambda (x) (expt x 2)) items)
  )

  (define x (list 1 2 3 4))

  (bar)
  (prnvar "x" x)
  (hr)
  (prnvar "(square-list-1 x)" (square-list-1 x))
  (br)
  (prnvar "(square-list-2 x)" (square-list-2 x))
  (bar)

;; *EOF*
```

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

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

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

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

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


57
321
88
```

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

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

```
#!/usr/bin/csi -s
;; sicp_ch2_e2-23.scm
;; Mac Radigan

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

;;; Exercise 2.23. The procedure for-each is similar to map. It takes as arguments a procedure and a
```

```scheme
;;; list of elements. However, rather than forming a list of the results, for-each just applies the
;;; procedure to each of the elements in turn, from left to right. The values returned by applying the
;;; procedure to the elements are not used at all -- for-each is used with procedures that perform an
;;; action, such as printing. For example,
;;;   (for-each (lambda (x) (newline) (display x))
;;;    (list 57 321 88))
;;;
;;;   57
;;;   321
;;;   88
;;;
;;; The value returned by the call to for-each (not illustra above) can be something arbitrary, such
;;; as true. Give an implementation of for-each.


  (define (for-each f x)
    (if (null? x)
      #t
      (begin
        (f (car x))
        (for-each f (cdr x))
      )
    )
  )



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

;; *EOF*
```

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

================================================================================

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

Suppose we evaluate the expression (list 1 (list 2 (list 3 4))). Give the result printed by the interpreter, the corresponding box-and-pointer structure, and the interpretation of this as a tree (as in figure 2.6).

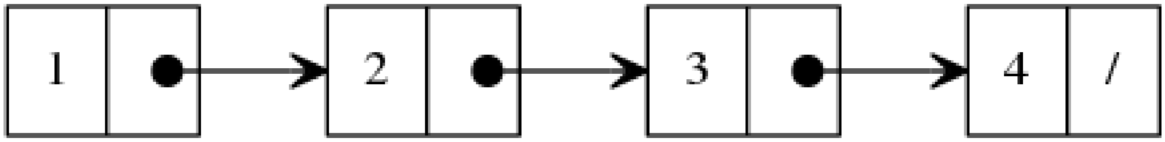

Figure 2: box and pointer representation

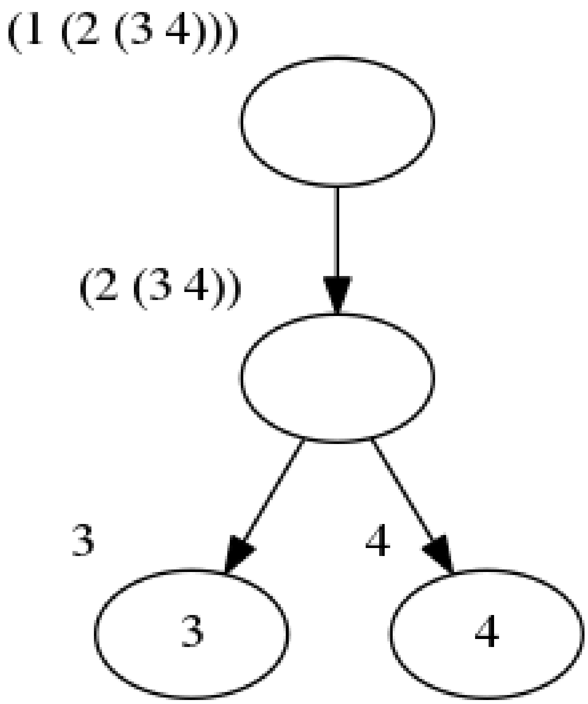

Figure 3: tree representation

```
#!/usr/bin/csi -s
;; sicp_ch2_e2-24.scm
;; Mac Radigan

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

;;; Exercise 2.24. Suppose we evaluate the expression (list 1 (list 2 (list 3 4))).
;;; Give the result printed by the interpreter, the corresponding box-and-pointer
;;; structure, and the interpretation of this as a tree (as in figure 2.6).

```

```scheme

(bar)
(prn (list 1 (list 2 (list 3 4))))
(bar)

;; *EOF*
```

```
## ./sicp_ch2_e2-24.scm

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

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

```
L1: (1 3 (5 7) 9)
L2: ((7))
L3: (1 (2 (3 (4 (5 (6 7))))))
```

```scheme
#!/usr/bin/csi -s
;; sicp_ch2_e2-25.scm
;; Mac Radigan

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

;;; Exercise 2.25. Give combinations of cars and cdrs that will pick 7 from
;;; each of the following lists:
;;;
;;;   (1 3 (5 7) 9)
;;;   ((7))
;;;   (1 (2 (3 (4 (5 (6 7))))))

(define L1 '(1 3 (5 7) 9) )
(define L2 '((7)) )
(define L3 '(1 (2 (3 (4 (5 (6 7)))))) )

(bar)
(prnvar "L1" (car (cdaddr L1)))
(prnvar "L2" (caar L2))
(prnvar "L3" (car (cdr (cadr (cadr (cadr (cadr (cadr L3))))))))
(br)
```

```
  (bar)

;; *EOF*
```

```
## ./sicp_ch2_e2-25.scm

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

==============================================================================
```

2.2.3 Sequences as Conventional Interfaces

2.2.4 Example: A Picture Language

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

```
#!/usr/bin/csi -s
;; sicp_ch2_e2-44.scm
;; Mac Radigan

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

  (use sicp)

;;; 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.

  (define (up-split painter n)
    (if (= n 0)
      painter
      (let
        ( (subimage (up-split painter (- n 1))) )
        (below painter (beside subimage subimage))
      )
    )
  )

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

```scheme
  ;; TEST
  ;; =======================================================

  (bar)
  (prnvar "up-split lena.jpg" "../figures/sicp_ch2_e2-44.png")
    (write-painter-to-png (up-split
      (image->painter "../figures/lena.jpg") 2)
       "../figures/sicp_ch2_e2-44.png")
  (bar)

;; *EOF*
```

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

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

../figures/sicp_ch2_e2-44.png

Figure 4: Up Split 2

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

```scheme
(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.

```scheme
#!/usr/bin/csi -s
;; sicp_ch2_e2-45.scm
;; Mac Radigan

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

  (use sicp)

```

```scheme
;;; 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.

  (define (split dir1 dir2)
    (lambda (painter n)
      (if (= n 0)
        painter
        (let
          ( (subimage ((split dir1 dir2) painter (- n 1))) )
          (dir1 painter (dir2 subimage subimage))
        ) ; let
      ) ; if
    ) ; lambda
  ) ; split

  (define right-split (split beside below))

  (define up-split (split below beside))

  ;; =======================================================
  ;; TEST
  ;; =======================================================

  (bar)
  (prnvar "right-split lena.jpg" "../figures/sicp_ch2_e2-45_right.png")
    (write-painter-to-png (right-split
      (image->painter "../figures/lena.jpg") 2)
       "../figures/sicp_ch2_e2-45_right.png")
  (hr)
  (prnvar "up-split lena.jpg" "../figures/sicp_ch2_e2-45_up.png")
    (write-painter-to-png (up-split
      (image->painter "../figures/lena.jpg") 2)
       "../figures/sicp_ch2_e2-45_up.png")
```

```
  (bar)

;; *EOF*
```

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

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

../figures/sicp_ch2_e2-45_right.png

Figure 5: Right Split 2

../figures/sicp_ch2_e2-45_up.png

Figure 6: Up Split 2

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

```
#!/usr/bin/csi -s
;; sicp_ch2_e2-46.scm
;; Mac Radigan

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

;;; 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:

  (define (make-vect x y)
    (cons x y))

  (define (xcor-vect v)
    (car v))

  (define (ycor-vect v)
    (cdr v))

  (define (scale-vect v s)
    (make-vect (* s (xcor-vect v)) (* s (ycor-vect v)) ) )

  (define (add-vect v1 v2)
    (make-vect
      (+ (xcor-vect v1) (xcor-vect v2))
      (+ (ycor-vect v1) (ycor-vect v2)) ) )

  (define (sub-vect v1 v2)
    (make-vect
      (- (xcor-vect v1) (xcor-vect v2))
      (- (ycor-vect v1) (ycor-vect v2)) ) )

;; =======================================================
;; TESTS
;; =======================================================

  (define v1 (make-vect  1  2))
  (define v2 (make-vect  1  -4))

  (prnvar "v1   " v1)
  (prnvar "v2   " v2)
  (prnvar "v1.x " (xcor-vect v1))
  (prnvar "v1.y " (ycor-vect v1))
```

```
  (prnvar "v1   " (scale-vect v1 2))
  (prnvar "v1+v2" (add-vect v1 v2))
  (prnvar "v1-v2" (sub-vect v1 v2))

;; *EOF*
```

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

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

2.3 Symbolic Data

2.3.1 Quotation

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

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

is true, but

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

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

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

Comparing to structures using $equals$?:

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

where

$$L(a) \wedge L(b) \Longrightarrow \neg(S(a) \wedge S(b))$$

(28)

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

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

(29)

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

$$(equals?\ \text{a b}) = f(a,b) = f_k(a, b, \top) \tag{30}$$

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

```
#!/usr/bin/csi -s
;; sicp_ch2_e2-54.scm
;; Mac Radigan

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

;;; 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.

  ;; =======================================================
  ;; NOT TAIL-RECURSIVE
  ;; =======================================================

  ;;
  ;;           { #f                                   if    s(a) and s(b)
  ;; f(a,b) = { a =?= b                               if    s(a) xor s(b)
  ;;           { f(car(a),car(b)) ^ f(cdr(a),cdr(b))   if !( s(a) and s(b) )
  ;;
  (define (my-equal-subopt? a b)
```

```
  (define (notlist? x) (not (list? x)))
  (cond
    ;;
    ;; case: null(a) ^ null(b) ->  #t  (null check)
    ;;
    ( (and (null? a) (null? b))
      #t )
    ;;
    ;; case: s(a) ^ s(b)       ->  a =?= b
    ;;
    ( (and (notlist?   a) (notlist?   b))
      (eq? a b) )
    ;;
    ;; case: s(a) xor s(b)     ->  #f
    ;;
    (      (xor (notlist? a) (notlist? b))
      #f )
    ;;
    ;; case: !( s(a)^s(b) )    ->  f( cdr(a), cdr(b) )
    ;;
    ;;       i.e. (not (and (notlist? a) (notlist? b)))
    ;;
    ( else
      (and (eq? (car a) (car b)) (my-equal-subopt? (cdr a) (cdr b))) )
  )
)

;; ======================================================
;; TAIL-RECURSIVE
;; ======================================================

;;
;;               { #f                                    if    s(a) and s(b)
;; f(a,b,p=#t) = { p ^ a =?= b                           if    s(a) xor s(b)
;;               { f(cdr(a), cdr(b), f(car(a),car(a),p))   if !( s(a) and s(b) )
;;
(define (my-equal? a b)
  (define (notlist? x) (not (list? x)))
  (define (f a b p)
    (cond
```

```
    ;;
    ;; case: null(a) ^ null(b) ->  #t  (null check)
    ;;
    (   (and (null? a) (null? b) )
     #t )
    ;;
    ;; case: s(a) ^ s(b)        ->  a =?= b
    ;;
    (   (and (notlist? a) (notlist? b))
     (eq? a b) )
    ;;
    ;; case: s(a) xor s(b)      ->  #f
    ;;
    (   (xor (notlist? a) (notlist? b))
     #f )
    ;;
    ;; case: !( s(a)^s(b) )     ->  f( cdr(a), cdr(b) )
    ;;
    ( else
     (f (cdr a) (cdr b) (f (car a) (car b) p) ) )
   ) ;; cond
  )         ;; <= recur
  (f a b #t) ;; <= base
 )

 (bar)
 (prn "intrinsic:")
 (prn (equal? '(this is a list) '(this is a list))      )
 (prn (equal? '(this is a list) '(this (is a) list))    )
 (hr)
 (prn "example 2-54: (not tail-recursive)")
 (prn (my-equal-subopt? '(this is a list) '(this is a list))   )
 (prn (my-equal-subopt? '(this is a list) '(this (is a) list)) )
; (hr)
; (prn "example 2-54: (tail-recursive)")
; (prn (my-equal? '(this is a list) '(this is a list))   )
; (prn (my-equal? '(this is a list) '(this (is a) list)) )
 (bar)

;; *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.

```scheme
#!/usr/bin/csi -s
;; sicp_ch2_e2-55.scm
;; Mac Radigan

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

;;; Exercise 2.55. Eva Lu Ator types to the interpreter the expression (car 'abracadabra)
;;; To her surprise, the interpreter prints back quote. Explain.

  (bar)
  (prn (car ''abracadabra) )
  (hr)
  (prn "Quote constructs a non-modifiable list, whose contents are the literal arguments to quote.")
  (prn "The second quote is part of the literal quoted list.")
  (prn "Car returns the first element of the list, which itself is quote.")
  (bar)

;; *EOF*
```

```
## ./sicp_ch2_e2-55.scm

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

2.3.2 Example: Symbolic Differentiation

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

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

```
#!/usr/bin/csi -s
;; sicp_ch2_e2-56.scm
;; Mac Radigan

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

;;; Exercise 2.56.  Show how to extend the basic differentiator to handle more
;;; kinds of expressions. For instance, implement the differentiation rule
;;;
;;;   d(u^n)              du
;;;   ------  =  nu^(n-1) --
;;;     dx                dx
;;;

  (define (deriv exp var)
    (cond ((number? exp) 0)
          ((variable? exp)
           (if (same-variable? exp var) 1 0))
          ((sum? exp)
           (make-sum (deriv (addend exp) var)
                     (deriv (augend exp) var)))
          ((product? exp)
           (make-sum
             (make-product (multiplier exp)
                           (deriv (multiplicand exp) var))
             (make-product (deriv (multiplier exp) var)
                           (multiplicand exp))))
          ;
          ;   d(u^n)              du
          ;   ------  =  nu^(n-1) --
```

```scheme
        ;     dx               dx
        ;
        ((exponent? exp)
         (make-product
           (make-product (power exp)
                         (make-exponent (base exp) (- (power exp) 1)))
           (deriv (base exp) var)))
        (else
         (error "unknown expression type -- DERIV" exp))))

(define (variable? x) (symbol? x))

(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))

(define (make-sum a1 a2) (list '+ a1 a2))

(define (=number? exp num) (and (number? exp) (= exp num)))

(define (make-product m1 m2) (list '* m1 m2))

(define (make-exponent b p)
  (cond ((=number? p 0) 1)
        ((=number? p 1) b)
        (else '('^ b p))))

(define (exponent? x) (eq? (car x) '^))

(define (base x) (cadr x))
(define (power x) (caddr x))

(define (sum? x) (and (pair? x) (eq? (car x) '+)))

(define (addend s) (cadr s))

(define (augend s) (caddr s))

(define (product? x)
  (and (pair? x) (eq? (car x) '*)))
```

```scheme
  (define (multiplier p) (cadr p))

  (define (multiplicand p) (caddr p))

  (bar)
  (prnvar "d/dx (2x)^4" (deriv '(^ (* 2 x) 4) 'x))
  (bar)

;; *EOF*
```

```
## ./sicp_ch2_e2-56.scm

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

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

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

 <more rules can be added here>

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

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

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

Constant Rule

$$\frac{\partial c}{\partial x} = 0 \tag{30}$$

Sum Rule

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

Product Rule

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

Exponent Rule

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

Chain Rule

$$\frac{\partial}{\partial x}(f \circ g) = (\frac{\partial f}{\partial x} \circ g) \cdot \frac{\partial g}{\partial x} \tag{30}$$

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

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

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

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

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

What corresponding changes to the derivative system are required?

```
#!/usr/bin/csi -s
;; sicp_ch2_e2-73.scm
;; Mac Radigan

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

;;; Exercise 2.73. Section 2.3.2 described a program that performs symbolic differentiation:
;;;
;;; (define (deriv exp var)
;;;   (cond ((number? exp) 0)
;;;     ((variable? exp) (if (same-variable? exp var) 1 0))
;;;     ((sum? exp)
;;;       (make-sum (deriv (addend exp) var)
;;;                 (deriv (augend exp) var)))
;;;       ((product? exp)
;;;          (make-sum
;;;          (make-product (multiplier exp)
```

```scheme
;;;                          (deriv (multiplicand exp) var))
;;;           (make-product (deriv (multiplier exp) var)
;;;                          (multiplicand exp))))
;;;
;;;  <more rules can be added here>
;;;
;;;        (else (error "unknown expression type -- DERIV" exp))))


;;; We can regard this program as performing a dispatch on the type of the expression to be differentiated.
;;; In this situation the "type tag" of the datum is the algebraic operator symbol (such as +) and the
;;; operation being performed is deriv. We can transform this program into data-directed style by
;;; rewriting the basic derivative procedure as
;;;
;;;   (define (deriv exp var)
;;;    (cond ((number? exp) 0)
;;;      ((variable? exp) (if (same-variable? exp var) 1 0))
;;;      (else ((get 'deriv (operator exp)) (operands exp)
;;;        var))))
;;;   (define (operator exp) (car exp))
;;;   (define (operands exp) (cdr exp))

 (define (deriv expr var)
   (cond ((number? expr) 0)
     ((variable? expr) (if (same-variable? expr var) 1 0))
     (else ((get 'deriv (operator expr)) (operands expr) var))))

 (define (operator expr) (car expr))

 (define (operands expr) (cdr expr))


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


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

```scheme

(define (install-sum-package)
  (define (addend expr) (car expr))
  (define (augend expr) (cadr expr))
  (define (make-sum a b)
    (cond
      ((eq? a 0) b)
      ((eq? b 0) a)
      ((and (number? a) (number? b)) (+ a b))
      (else (list '+ a b))
    )
  )
  ;; sum rule: fg = f' + g'
  (define (deriv-sum expr var)
    (make-sum (deriv (addend expr) var) (deriv (augend expr) var))
  )
  (put 'deriv '+ deriv-sum)
  'done
)

(define (install-product-package)
  (define (multiplier expr) (car expr))
  (define (multiplicand expr) (cadr expr))
  (define (make-product a b)
    (cond
      ((or (eq? a 0) (eq? b 0)) 0)
      ((eq? a 1) b)
      ((eq? b 1) a)
      ((and (number? a) (number? b)) (* a b))
      (else (list '* a b))
    )
  )
  (define (make-sum a b)
    (cond
      ((eq? a 0) b)
      ((eq? b 0) a)
      ((and (number? a) (number? b)) (+ a b))
      (else (list '+ a b))
    )
  )
```

```
    ;; product rule: fg = f g' + f' g
    (define (deriv-product expr var)
      (make-sum
        (make-product (multiplier expr) (deriv (multiplicand expr) var))
        (make-product (deriv (multiplier expr) var) (multiplicand expr) )
      )
    )
    (put 'deriv '* deriv-product)
   'done
  )

  (install-sum-package)
  (install-product-package)

  (define dx 'x)

  (bar)
  (prn " b. Write the procedures for derivatives of sums and products, ")
  (prn "    and the auxiliary code required to install them in the table ")
  (prn "    used by the program above.")
  (hr)
  (define expr-1 '(* x x))
  (prn "e1 := (* x x)")
  (prnvar "d/dx e1" (deriv expr-1 dx))

  (hr)
  (define expr-2 '(+ (* x x) x))
  (prn "e2 := (+ (* x x) x)")
  (prnvar "d/dx e2" (deriv expr-2 dx))

  (hr)
  (define expr-3 '(* (+ (* x x) (* x z) ) (+ (* x y) (* x x)) ) )
  (prn "e3 := (* (+ (* x x) (* x z) ) (+ (* x y) (* x x)) )")
  (prnvar "d/dx e3" (deriv expr-3 dx))

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

  (define (install-exponent-package)
    (define (base expr) (car expr))
```

```
    (define (power expr) (cadr expr))
    (define (make-product a b)
      (cond
        ((or (eq? a 0) (eq? b 0)) 0)
        ((eq? a 1) b)
        ((eq? b 1) a)
        ((and (number? a) (number? b)) (* a b))
        (else (list '* a b))
      )
    )
    (define (make-sum a b)
      (cond
        ((eq? a 0) b)
        ((eq? b 0) a)
        ((and (number? a) (number? b)) (+ a b))
        (else (list '+ a b))
      )
    )
    (define (make-exponent b p)
      (cond ((=number? p 0) 1)
        ((=number? p 1) b)
        (else (list '^ b p))
      )
    )
    ;
    ;   d(u^n)              du
    ;   ------  =  nu^(n-1) --
    ;     dx                dx
    ;
    (define (deriv-exponent expr var)
       (make-product
         (make-product (power expr)
                       (make-exponent (base expr) (- (power expr) 1)))
         (deriv (base expr) var)
       )
    )
    (put 'deriv '^ deriv-exponent)
   'done
  )
```

```
  (install-exponent-package)


  (bar)
  (prn " c. Choose any additional differentiation rule that you like, ")
  (prn "    such as the one for exponents (exercise 2.56), and install ")
  (prn "    it in this data-directed system.")
  (hr)
  (define expr-4 '(^ (* 2 x) 4) )
  (prn "e4 := (^ (* 2 x) 4) )")
  (prnvar "d/dx e4" (deriv expr-4 dx))
  (hr)
  (define expr-5 '(* 2 (^ x 4)) )
  (prn "e5 := (* 2 (^ x 4)) )")
  (prnvar "d/dx e5" (deriv expr-5 dx))

;;; d. In this simple algebraic manipulator the type of an expression is the algebraic operator that binds it
;;;    together. Suppose, however, we indexed the procedures in the opposite way, so that the dispatch line
;;;    in deriv looked like
;;;
;;;   ((get (operator exp) 'deriv) (operands exp) var)
;;;
;;; What corresponding changes to the derivative system are required?

  (define (my-deriv expr var)
    (cond ((number? expr) 0)
      ((variable? expr) (if (same-variable? expr var) 1 0))
      (else ((get (my-operator expr) 'my-deriv) (my-operands expr) var))))

  (define (my-operator expr) (car expr))

  (define (my-operands expr) (cdr expr))

  (define (my-install-sum-package)
    (define (addend expr) (car expr))
    (define (augend expr) (cadr expr))
    (define (make-sum a b)
      (cond
```

```
        ((eq? a 0) b)
        ((eq? b 0) a)
        ((and (number? a) (number? b)) (+ a b))
        (else (list '+ a b))
      )
    )
    ;; sum rule: fg = f' + g'
    (define (deriv-sum expr var)
      (make-sum (my-deriv (addend expr) var) (my-deriv (augend expr) var))
    )
    (put '+ 'my-deriv deriv-sum)
    'done
  )

  (define (my-install-product-package)
    (define (multiplier expr) (car expr))
    (define (multiplicand expr) (cadr expr))
    (define (make-product a b)
      (cond
        ((or (eq? a 0) (eq? b 0)) 0)
        ((eq? a 1) b)
        ((eq? b 1) a)
        ((and (number? a) (number? b)) (* a b))
        (else (list '* a b))
      )
    )
    (define (make-sum a b)
      (cond
        ((eq? a 0) b)
        ((eq? b 0) a)
        ((and (number? a) (number? b)) (+ a b))
        (else (list '+ a b))
      )
    )
    ;; product rule: fg = f g' + f' g
    (define (deriv-product expr var)
      (make-sum
        (make-product (multiplier expr) (my-deriv (multiplicand expr) var))
        (make-product (my-deriv (multiplier expr) var) (multiplicand expr) )
      )
```

```
    )
    (put '* 'my-deriv deriv-product)
   'done
  )

  (bar)
  (prn " d.  Suppose, however, we indexed the procedures in the opposite")
  (prn "     way, so that the dispatch line in deriv looked like")
  (br)
  (prn "    ((get (operator exp) 'deriv) (operands exp) var)")
  (hr)
  (prn "(put '+ 'deriv deriv-sum)")
  (prn "(put '* 'deriv deriv-product)")
  (br)

; (hr)
; (prn "e1 := (* x x)")
; (prnvar "d/dx e1" (my-deriv expr-1 dx))

; (hr)
; (prn "e2 := (+ (* x x) x)")
; (prnvar "d/dx e2" (my-deriv expr-2 dx))

; (hr)
; (prn "e3 := (* (+ (* x x) (* x z) ) (+ (* x y) (* x x)) )")
; (prnvar "d/dx e3" (my-deriv expr-3 dx))
  (bar)

;; *EOF*
```

```
## ./sicp_ch2_e2-73.scm

================================================================================
 b. Write the procedures for derivatives of sums and products,
    and the auxiliary code required to install them in the table
    used by the program above.
--------------------------------------------------------------------------------
e1 := (* x x)
d/dx e1 := (+ x x)
--------------------------------------------------------------------------------
e2 := (+ (* x x) x)
d/dx e2 := (+ (+ x x) 1)
--------------------------------------------------------------------------------
e3 := (* (+ (* x x) (* x z) ) (+ (* x y) (* x x)) )
d/dx e3 := (+ (* (+ (* x x) (* x z)) (+ y (+ x x))) (* (+ (+ x x) z) (+ (* x y) (* x x))))
================================================================================
 c. Choose any additional differentiation rule that you like,
    such as the one for exponents (exercise 2.56), and install
    it in this data-directed system.
--------------------------------------------------------------------------------
e4 := (^ (* 2 x) 4) )
d/dx e4 := (* (* 4 (^ (* 2 x) 3)) 2)
--------------------------------------------------------------------------------
e5 := (* 2 (^ x 4)) )
d/dx e5 := (* 2 (* 4 (^ x 3)))
================================================================================
 d.  Suppose, however, we indexed the procedures in the opposite
     way, so that the dispatch line in deriv looked like

    ((get (operator exp) 'deriv) (operands exp) var)
--------------------------------------------------------------------------------
(put '+ 'deriv deriv-sum)
(put '* 'deriv deriv-product)

================================================================================
```

2.3.3 Example: Representing Sets

2.3.4 Example: Huffman Encoding Trees

2.4 Multiple Representations for Abstract Data

2.4.1 Representations for Complex Numbers

2.4.2 Tagged data

2.4.3 Data-Directed Programming and Additivity

2.5 Systems with Generic Operations

2.5.1 Generic Arithmetic Operations

2.5.2 Combining Data of Different Types

2.5.3 Example: Symbolic Algebra

3 Modularity, Objects, and State

3.1 Assignment and Local State

3.1.1 Local State Variables

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

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


(A 10)

15


(A 10)

25
```

```
#!/usr/bin/csi -s
;; sicp_ch3_e3-1.scm
;; Mac Radigan

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

  (use sicp sicp-eval sicp-eval-anal sicp-streams)
  (load "./ch3support.scm")
; (load "./ch3.scm")


  ;; Exercise 3.1: An accumulator is a procedure that is called repeatedly
  ;; with a single numeric argument and accumu- lates its arguments into a sum.
  ;; Each time it is called, it returns the currently accumulated sum. Write a
  ;; procedure make-accumulator that generates accumulators, each main- taining
  ;; an independent sum. The input to make-accumulator should specify the initial
  ;; value of the sum; for example
  ;;
  ;; (define A (make-accumulator 5))
```

```
  ;;
  ;; (A 10)
  ;; 15
  ;;
  ;; (A 10)
  ;; 25

  (define (make-accumulator n)
    (let ((acc n))
      (lambda (f) (set! acc (+ acc f)) acc)
    )
  )

  (bar)
  (define A (make-accumulator 5))
  (prnvar "acc+10" (A 10))
  (hr)
  (prnvar "acc+10" (A 10))
  (bar)

;; *EOF*
```

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

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


(s 100)
10


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

1

```scheme
#!/usr/bin/csi -s
;; sicp_ch3_e3-1.scm
;; Mac Radigan

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

  (use sicp sicp-eval sicp-eval-anal sicp-streams)
  (load "./ch3support.scm")
; (load "./ch3.scm")


  ;; Exercise 3.2: In so\ue039ware-testing applications, it is useful
  ;; to be able to count the number of times a given procedure
  ;; is called during the course of a computation. Write a pro-
  ;; cedure make-monitored that takes as input a procedure, f ,
  ;; that itself takes one input. \ue049e result returned by make-
  ;; monitored is a third procedure, say mf , that keeps track
  ;; of the number of times it has been called by maintaining
  ;; an internal counter. If the input to mf is the special symbol
  ;; how-many-calls? , then mf returns the value of the counter.
  ;; If the input is the special symbol reset-count , then mf re-
  ;; sets the counter to zero. For any other input, mf returns the
  ;; result of calling f on that input and increments the counter.
  ;; For instance, we could make a monitored version of the
  ;; sqrt procedure:
  ;;
  ;; (define s (make-monitored sqrt))
  ;;
  ;; (s 100)
  ;; 100
  ;;
  ;; (s 'how-many-calls?)
  ;; 1

  (define (make-monitored f)
    (let ((count 0))
      (lambda (arglist)
```

```
        (cond
          ((equal? arglist 'how-many-calls?) count)
          ((eq? arglist 'reset-count) (set! count 0))
          (else
            (begin
              (set! count (+ count 1))
              (f arglist)
            )
          )
        )
      )
    )
  )

  (bar)
  (define s (make-monitored sqrt))
  (prnvar "(s 100)" (s 100))
  (hr)
  (prnvar "(s 'how-many-calls?)" (s 'how-many-calls?))
  (hr)
  (prn "(s 'reset-count?)")
  (s 'reset-count)
  (prnvar "(s 'how-many-calls?)" (s 'how-many-calls?))
  (hr)
  (bar)

;; *EOF*
```

3.1.2 The Benefits of Introducing Assignment

3.1.3 The Costs of Introducing Assignment

3.2 The Environment Model of Evaluation

3.2.1 The Rules for Evaluation

3.2.2 Applying Simple Procedures

3.2.3 Frames as the Repository of Local State

3.2.4 Internal Definitions

3.3 Modeling with Mutable Data

3.3.1 Mutable List Structure

3.3.2 Representing Queues

3.3.3 Representing Tables

3.3.4 A Simulator for Digital Circuits

3.3.5 Propagation of Constraints

3.4 Concurrency: Time Is of the Essence

3.4.1 The Nature of Time in Concurrent Systems

3.4.2 Mechanisms for Controlling Concurrency

3.5 Streams

3.5.1 Streams Are Delayed Lists

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

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

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

```
#!/usr/bin/csi -s
;; sicp_ch3_e3-50.scm
;; Mac Radigan

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

  (use sicp sicp-eval sicp-eval-anal sicp-streams)
  (load "./ch3support.scm")
  (load "./ch3.scm")

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

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

  (define (stream-map proc . argstreams)
    (if (stream-null? (car argstreams))
        the-empty-stream
        (cons-stream
         (apply proc (map car argstreams))
         (apply stream-map
                (cons proc (map stream-cdr argstreams))))))

;;; NB Error: unbound variable: get-new-pair
;;; (constructor for the empty list)

;; *EOF*
```

3.5.2 Infinite Streams

3.5.3 Exploiting the Stream Paradigm

3.5.4 Streams and Delayed Evaluation

3.5.5 Modularity of Functional Programs and Modularity of Objects

4 Metalinguistic Abstraction

```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

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

4.4.1 Deductive Information Retrieval

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

(a) all people supervised by Ben Bitdiddle;

(b) the names and jobs of all people in the accounting division;

(c) the names and addresses of all people who live in Slumerville.

```scheme
#!/usr/bin/csi -s
;; sicp_ch4_e4-55.scm
;; Mac Radigan

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

  (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)

;;; 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.

;; =======================================================
;; QUERY PROCESSOR
;; =======================================================

  (define (eval-query query)
    (let ((q (query-syntax-process query)))
      (cond ((assertion-to-be-added? q)
             (add-rule-or-assertion! (add-assertion-body q))
             (newline)
             (display "Assertion added to data base.")
             )
            (else
             (newline)
             (display output-prompt)
             ;; [extra newline at end] (announce-output output-prompt)
             (display-stream
              (stream-map
               (lambda (frame)
                 (instantiate q
                              frame
                              (lambda (v f)
```

```
                                (contract-question-mark v))))
                (qeval q (singleton-stream '())))))
              ))))

;; =======================================================
;; a. all people supervised by Ben Bitdiddle:
;; =======================================================
  (define query-a
    '(supervisor ?person (Bitdiddle Ben)) )

;; =======================================================
;; b. the names and jobs of all people in the accounting division;
;; =======================================================
  (define query-b
    '(job ?person (accounting . ?title)) )

;; =======================================================
;; c. the names and addresses of all people who live in Slumerville.
;; =======================================================
  (define query-c '(address ?person (Slumerville . ?address)) )

;; =======================================================
;; TESTS
;; =======================================================

  (bar)
  (prn "Query A. all people supervised by Ben Bitdiddle:")
  (eval-query query-a) (br) (hr)
  (prn "Query B. the names and jobs of all people in the accounting division:")
  (eval-query query-b) (br) (hr)
  (prn "Query C. the names and addresses of all people who live in Slumerville:")
  (eval-query query-c) (br)
  (bar)

;; *EOF*
```

```
## ./sicp_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.

```
#!/usr/bin/csi -s
;; sicp_ch4_e4-56.scm
;; Mac Radigan

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

  (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)

;;; 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.

;; ======================================================
;; QUERY PROCESSOR
;; ======================================================

  (define (eval-query query)
    (let ((q (query-syntax-process query)))
      (cond ((assertion-to-be-added? q)
             (add-rule-or-assertion! (add-assertion-body q))
             (newline)
             (display "Assertion added to data base.")
             )
            (else
             (newline)
             (display output-prompt)
             ;; [extra newline at end] (announce-output output-prompt)
             (display-stream
              (stream-map
               (lambda (frame)
                 (instantiate q
                              frame
                              (lambda (v f)
                                (contract-question-mark v))))
               (qeval q (singleton-stream '()))))
             ))))

;; ======================================================
;; a. the names of all people who are supervised by
;;    Ben Bitdiddle, together with their addresses
;; ======================================================
  (define query-a
    '(and (supervisor (Bitdiddle Ben) ?person)
          (address ?person ?address)
     ) ; conjunction
  ) ; query A
```

```
;; =======================================================
;; b. all people whose salary is less than Ben Bitdiddle's,
;;    together with their salary and Ben Bitdiddle's salary
;; =======================================================

;;; TODO

  (define query-b
    '(and (salary (Bitdiddle Ben) ?max-salary)
          (salary ?person ?salary)
     ) ; conjunction
  ) ; query B

; (define query-b
;   '(and (salary (Bitdiddle Ben) ?max-salary)
;          (salary ?person ?salary)
;          (lisp-value < ?salary ?max-salary)
;    ) ; conjunction
; ) ; query B
;
;; =======================================================
;; c. all people who are supervised by someone who is not
;;    in the computer division, together with the
;;    supervisor's name and job.
;; =======================================================
  (define query-c
    '(and (supervisor ?supervisor ?person)
          (not (job ?supervisor (computer . ?supervisor-title)))
          (job ?supervisor ?supervisor-job)
     ) ; conjunction
  ) ; query C

;; =======================================================
;; TESTS
;; =======================================================

  (bar)
  (prn "Query A. the names of all people who are supervised by Ben Bitdiddle, together with their
↪  addresses")
```

```scheme
  (eval-query query-a) (br) (hr)
  (prn "[TODO] Query B. all people whose salary is less than Ben Bitdiddle’s, together with their salary and Ben Bitdiddle’s salary")
  (eval-query query-b) (br) (hr)
  (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.")
  (eval-query query-c) (br)
  (bar)

;; *EOF*
```

```
## ./sicp_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

Algorithm 1 Driver Loop

procedure DRIVER
 {Main Query REPL}
 let $q \leftarrow$ **READ**
 if Assertion-To-Be-Added? q **then**
  Add-Rule-Or-Assertion! $\leftarrow q$
 else
  Display-Stream Stream-Map \
  λf.**Instantiate**(q,f,$\lambda v f$.ccm(v)) \
  qeval(q,the-stream)
 end if
end procedure

```
(define (query-driver-loop)
  (prompt-for-input input-prompt)
  (let ((q (query-syntax-process (read))))
    (cond ((assertion-to-be-added? q)
           (add-rule-or-assertion! (add-assertion-body q))
           (newline)
           (display "Assertion added to data base.")
           (query-driver-loop))
          (else
           (newline)
           (display output-prompt)
           ;; [extra newline at end] (announce-output output-prompt)
           (display-stream
            (stream-map
             (lambda (frame)
               (instantiate q
                            frame
                            (lambda (v f)
                              (contract-question-mark v))))
             (qeval q (singleton-stream '()))))
           (query-driver-loop)))))
```

```
(define (instantiate exp frame unbound-var-handler)
  (define (copy exp)
    (cond ((var? exp)
           (let ((binding (binding-in-frame exp frame)))
```

Algorithm 2 Instantiate

---

function INSTANTIATE(e:Expression, f:Frame h:Handler)
  {Instantiate Binding Variables}
  function COPY(e:Expression)
    if var? e **then**
      let b := f [e]
      if b **then**
        $Copy(b_D)$
        {Contract Question Mark (ccm) with Handler h}
        $ccm(e)$
      end if
    end if
    if pair? e **then**
      [$Copy(e_A)$ $Copy(e_D)$]
    end if
    if Else: **then**
      $Copy(e)$
    end if
  end function
  $Copy(e)$
end function

```
            (if binding
                (copy (binding-value binding))
                (unbound-var-handler exp frame))))
         ((pair? exp)
          (cons (copy (car exp)) (copy (cdr exp))))
         (else exp)))
  (copy exp))
```

Algorithm 3 Qeval

function QEVAL(q:Query, s:FrameStream)
  {Evaluate Query}
  let f := **getProc**(q)
  if f **then**
    f(q_D, s)
  else
    SimpleQuery(q_D, s)
  end if
end function

```
(define (qeval query frame-stream)
  (let ((qproc (get (type query) 'qeval)))
    (if qproc
        (qproc (contents query) frame-stream)
        (simple-query query frame-stream))))
```

Algorithm 4 SimpleQuery

function SIMPLEQUERY(p:QueryPattern, s:FrameStream)
  {Evaluate Simple Query}
  stream-flatmap \
  λf.**StreamAppendDelayed**(**FindAssertions**(p, f) , **Delay**(**ApplyRules**(p, s))) \
  $f s$
end function

```
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
   (lambda (frame)
     (stream-append-delayed
      (find-assertions query-pattern frame)
      (delay (apply-rules query-pattern frame))))
```

```
    frame-stream))
```

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
      xor
      Y
      Y-normal
      Y2
      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
```

```scheme
(define (rotate-right x n)
  (if (< n 1)
    x
    (rotate-right
      (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 (my-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)
  )

```

```scheme
;; 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
  )
)

;; exclusive or
(define (xor a b)
  (or (and (not a) b) (and a (not b)))
)

;; 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)

;; Y combiner
;;   strict-order
(define Y
  (lambda (f)
    ((lambda (x) (x x))
     (lambda (x) (f (lambda (y) ((x x) y)))))))
;;   normal-order
(define (Y-normal f)
   ((lambda (x) (x x))
     (lambda (x) (f (x x)))))
(define Y2
```

```
      (lambda (h)
        (lambda args (apply (h (Y h)) args))))


  ) ; 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*
```

8 Appendix C: Notation

8.1 Membership

$$S\left(x\right) \triangleq x \in \mathrm{SYMBOL}$$

(30)

$$L\left(x\right) \triangleq x \in \mathrm{LIST}$$

(31)

8.2 Symbols

$$\top \triangleq \#\mathrm{t}$$

(32)

$$\bot \triangleq \#\mathrm{f}$$

(33)

8.3 Access

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

(34)

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

(35)

8.4 Equality

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

(36)

8.5 Logic

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

(37)

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

(38)

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

(39)

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

(40)

9 Appendix D: Y-Combinator

9.1 Introduction

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

9.2 Cannonical Expression

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

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

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

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

9.3 Connonical Form in Scheme

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

9.3.1 Strict Scheme (Chicken)

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

```scheme
#!/usr/bin/csi -s
;; y-combinator.scm
;; Mac Radigan
;;
;; copied from http://mvanier.livejournal.com/2897.html

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


  ;;; Eliminating (most) explicit recursion (lazy version)
```

```

(define Y
  (lambda (f)
    (f (Y f))))

(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define factorial (Y almost-factorial))

(prn (factorial 6)) ; infinite loop

;; *EOF*
```

9.3.2 Using Lazy Evaluation (Racket #lang lazy)

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

```
#!/usr/bin/racket
;; y-combinator.scm
;; Mac Radigan
;;
;; copied from http://mvanier.livejournal.com/2897.html

  #lang lazy

  ;;; Eliminating (most) explicit recursion (lazy version)

  (define Y
    (lambda (f)
      (f (Y f))))

  (define almost-factorial
    (lambda (f)
      (lambda (n)
        (if (= n 0)
            1
```

```
            (* n (f (- n 1)))))))

  (define factorial (Y almost-factorial))

  (println (factorial 6)) ; 720

;; *EOF*
```

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

720
```

9.4 Normal Order Y Combinator

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

9.4.1 Strict Scheme (Chicken)

```
#!/usr/bin/csi -s
;; y-combinator-normal.scm
;; Mac Radigan
;;
;; copied from http://mvanier.livejournal.com/2897.html

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


  ;;; The lazy (normal-order) Y combinator

  (define Y
    (lambda (f)
      ((lambda (x) (f (x x)))
       (lambda (x) (f (x x))))))

  (define almost-factorial
    (lambda (f)
      (lambda (n)
        (if (= n 0)
            1
            (* n (f (- n 1)))))))
```

```

  (define factorial (Y almost-factorial))

  (prn (factorial 6)) ; infinite loop

;; *EOF*
```

9.4.2 Using Strict Evaluation (Racket)

```
#!/usr/bin/racket
;; y-combinator-normal.scm
;; Mac Radigan
;;
;; copied from http://mvanier.livejournal.com/2897.html

  #lang lazy

  ;;; The lazy (normal-order) Y combinator

  (define Y
    (lambda (f)
      ((lambda (x) (f (x x)))
       (lambda (x) (f (x x))))))

  (define almost-factorial
    (lambda (f)
      (lambda (n)
        (if (= n 0)
            1
            (* n (f (- n 1)))))))

  (define factorial (Y almost-factorial))

  (println (factorial 6)) ; 720

;; *EOF*
```

9.4.3 Using Lazy Evaluation (Racket #lang lazy)

However, it will work under lazy evaluation.

```
#!/usr/bin/racket
;; y-combinator-normal.scm
;; Mac Radigan
;;
;; copied from http://mvanier.livejournal.com/2897.html

 #lang lazy

 ;;; The lazy (normal-order) Y combinator

 (define Y
   (lambda (f)
     ((lambda (x) (f (x x)))
      (lambda (x) (f (x x))))))

 (define almost-factorial
   (lambda (f)
     (lambda (n)
       (if (= n 0)
           1
           (* n (f (- n 1)))))))

 (define factorial (Y almost-factorial))

 (println (factorial 6)) ; 720

;; *EOF*
```

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

720
```

9.5 Strict (Applicative-Order) Y Combinator

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

9.5.1 Strict Scheme (Chicken)

```
#!/usr/bin/csi -s
;; y-combinator-struct.scm
;; Mac Radigan
;;
;; copied from http://mvanier.livejournal.com/2897.html

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



  ;;; The strict (applicative-order) Y combinator

  (define Y
    (lambda (f)
      ((lambda (x) (x x))
       (lambda (x) (f (lambda (y) ((x x) y)))))))

  (define almost-factorial
    (lambda (f)
      (lambda (n)
        (if (= n 0)
            1
            (* n (f (- n 1)))))))

  (define (part-factorial self)
     (let ((f (lambda (y) ((self self) y))))
       (lambda (n)
         (if (= n 0)
           1
           (* n (f (- n 1)))))))

  (define factorial (Y almost-factorial))

  (prn (factorial 6)) ; 720

;; *EOF*
```

./y-combinator-strict.scm

9.5.2 Using Strict Evaluation (Racket)

```
#!/usr/bin/racket
;; y-combinator-struct.scm
;; Mac Radigan
;;
;; copied from http://mvanier.livejournal.com/2897.html

  #lang racket

  ;;; The strict (applicative-order) Y combinator

  (define Y
    (lambda (f)
      ((lambda (x) (x x))
       (lambda (x) (f (lambda (y) ((x x) y)))))))

  (define almost-factorial
    (lambda (f)
      (lambda (n)
        (if (= n 0)
            1
            (* n (f (- n 1)))))))

  (define (part-factorial self)
     (let ((f (lambda (y) ((self self) y))))
       (lambda (n)
         (if (= n 0)
           1
           (* n (f (- n 1)))))))

  (define factorial (Y almost-factorial))

  (println (factorial 6)) ; 720

;; *EOF*
```

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

720
```

9.5.3 Using Lazy Evaluation (Racket #lang lazy)

```
#!/usr/bin/racket
;; y-combinator-struct.scm
;; Mac Radigan
;;
;; copied from http://mvanier.livejournal.com/2897.html

  #lang lazy

  ;;; The strict (applicative-order) Y combinator

  (define Y
    (lambda (f)
      ((lambda (x) (x x))
       (lambda (x) (f (lambda (y) ((x x) y)))))))

  (define almost-factorial
    (lambda (f)
      (lambda (n)
        (if (= n 0)
            1
            (* n (f (- n 1)))))))

  (define (part-factorial self)
     (let ((f (lambda (y) ((self self) y))))
       (lambda (n)
         (if (= n 0)
           1
           (* n (f (- n 1)))))))

  (define factorial (Y almost-factorial))

  (println (factorial 6)) ; 720

;; *EOF*
```

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

720
```