

BU CAS CS 320:
Concepts of Programming Languages
Lecture Notes

Hongwei Xi
Computer Science Department, Boston University
111 Cummington Street, Boston, MA 02215

Chapter 1

Building Abstractions with Procedures

1.1 The Elements of Programming

- **primitive expressions**, which represent the simplest entities the language is concerned with, and
- **means of combination**, by which compound elements are built from simpler ones, and
- **means of abstraction**, by which compound elements can be named and manipulated as units.

1.1.1 Expressions

Here are some examples of arithmetic expressions and their values. Intuitively, we use values for a special form of expressions that cannot be computed furthermore.

expression	value
123	123
(+ 6 23)	29
(- 100 4)	96
(* 25 25)	625
(/ 3 6)	1/2
(+ 1 10 100 1000 10000)	11111
(- 10000 1000 100 10 1)	8889
(* 5 4 3 2 1)	120
(+ (/ 3 6) (/ 6 12))	1
(/ 8 (- 3 (/ 8 3)))	24

1.1.2 Naming and the Environment

A critical aspect of a programming language is the means it provides for using names to refer to values. For instance, the following syntax in Scheme associates the value 2 with the name `size`.

```
(define size 2)
```

After this association, it should be clear that the value of the following expression is 20.

```
(* size 10)
```

Also, it should be clear that the value associated with `area` and `circumference` are 314.159 and 62.8318, respectively.

```
(define pi 3.14159)
(define radius 10)
(define area (* pi radius radius))
(define circumference (* 2 pi radius))
```

1.1.3 Evaluating Combinations

- Evaluate the subexpressions of a combination
- Apply the procedure that is the leftmost subexpression (the operator) to the argument that are the values of the other subexpressions (the operands)

The following gives some illustration of evaluation.

$$\begin{aligned}
 (* (+ 2 \underline{(* 4 6)}) (+ 3 (+ 1 4) 7)) &\rightarrow (* \underline{(+ 2 24)} (+ 3 (+ 1 4) 7)) \\
 &\rightarrow (* 26 (+ 3 \underline{(+ 1 4)} 7)) \\
 &\rightarrow (* 26 \underline{(+ 3 5 7)}) \rightarrow \underline{(* 26 15)} \rightarrow 390
 \end{aligned}$$

1.1.4 Compound Procedures

The general form of a procedure definition is

```
(define (<name> <formal parameters>) <body>)
```

For instance, the square function can be defined as follows.

```
(define (square x) (* x x))
```

After defining `square`, we can now use it.

$$\begin{aligned}
 \underline{(\text{square } 11)} &\rightarrow \underline{(* 11 11)} \rightarrow 121 \\
 (\text{square } \underline{(+ 10 10)}) &\rightarrow \underline{(\text{square } 20)} \rightarrow \underline{(* 20 20)} \rightarrow 400
 \end{aligned}$$

We can also use `square` to define another procedure:

```
(define (sum-of-squares x y) (+ (square x) (square y)))
```

The expression `(sum-of-squares 3 4)` evaluates to `25`.

1.1.5 The Substitution Model for Procedure Application

- applicative (call-by-value) versus normal order (call-by-name)

1.1.6 Conditional Expressions and Predicates

The absolute value function can be defined as follows.

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        (<< x 0) (- x))))
```

The general form of a conditional expression is:

```
(cond (<p1> <e1>)
      (<p2> <e2>)
      ⋮
      (<pn> <en>))
```

Here is another way to write the absolute value procedure:

```
(define (abs x)
  (if (< x 0) (- x) x))
```

The general form of an if expression is

```
(if <predicate> <consequent> <alternative>)
```

In addition to primitive predicates such as `<`, `=` and `>`, here are some boolean operations for forming compound predicates.

- `(and <e1> ⋯ <en>)`
- `(or <e1> ⋯ <en>)`
- `(not <e>)`

The following are some examples.

```
(define (between x y z) (and (<= x y) (<= x z)))
(define (<> x y) (not (= x y)))
(define (<= x y) (not (> x y)))
```

1.1.7 Example: Square Roots By Newton's Method

```
(define epsilon 0.001)

(define (good-enough? guess x)
  (< (/ (abs (- (* guess guess) x)) x) epsilon))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (/ (+ guess (/ x guess)) 2.0) x)))

(define (sqrt x)
  (sqrt-iter (/ x 2.0) x))
```

1.1.8 Procedures as Black-Box Abstractions

- **Local names** One detail of the implementation of a procedure should not matter to the user of the procedure is the implementer's choice of the names for the formal parameters of a procedure. For instance, the following two procedures should not be considered equivalent.

```
(define (square x) (* x x))
(define (square y) (* y y))
```

In mathematics, such formal parameters are called bounded variables. For instance, we have the following in calculus, where x and y are bound variables.

$$\int_0^1 x^2 dx = \int_0^1 y^2 dy = \frac{1}{3}$$

- **Internal definitions and block structure** Sometimes, we may want to hide the definition of some auxiliary procedures. This can be done using the following block structure, which is originated in the design of the programming language Algol 60.

```
(define (sqrt x)
  (define epsilon 0.00001)
  (define (good-enough? guess)
    (< (/ (abs (- (* guess guess) x)) x) epsilon))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (define (improve guess)
    (/ (+ guess (/ x guess)) 2))
  (sqrt-iter (/ x 2.0)))
```

1.2 Procedures and the Processes They Generate

1.2.1 Linear Recursion and Iteration

Let us consider the factorial function, defined by

$$n! = n \cdot (n - 1) \cdots 2 \cdot 1$$

Clearly, for any positive number n , we have the following equation (in the case $n = 1$, we define $0!$ to be 1).

$$n! = n \cdot (n - 1)!$$

This allows us to implement the factorial function in Scheme as follows.

```
(define (factorial n)
  (if (zero? n) 1 (* n (factorial (- n 1)))))
```

With the substitution model, we can watch the procedure in action computing $3!$ as follows.

```
(factorial 3) → (* 3 (factorial 2))
              → (* 3 (* 2 (factorial 1)))
              → (* 3 (* 2 (* 1 (factorial 0))))
              → (* 3 (* 2 (* 1 1)))
              → (* 3 (* 2 1))
              → (* 3 2)
              → 6
```

The factorial function can also be implemented in the following iterative style. The style of recursion involved in the definition of `iter` is called *tail recursion*. Roughly speaking, a procedure is tail-recursively defined if each call to this procedure terminates with a call to the procedure itself.

```
(define (factorial n)
  (define (fact-iter n res)
    (if (zero? n) res (fact-iter (- n 1) (* n res))))
  (iter n 1))

(factorial 3) → (fact-iter 3 1)
              → (fact-iter 2 3)
              → (fact-iter 1 6)
              → (fact-iter 0 6)
              → 6
```

1.2.2 Tree Recursion

Another common pattern of computation is called *tree recursion*. As an example, let us consider the sequence of Fibonacci numbers, in which each number is the sum of the preceding two:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

In general, the n th Fibonacci number $Fib(n)$ is defined as follows.

$$Fib(n) = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ Fib(n-1) + Fib(n-2) & \text{otherwise.} \end{cases}$$

We can immediately translate this definition into a recursive procedure in Scheme for computing Fibonacci numbers:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1)) (fib (- n 2))))))
```

It can be readily shown that the number of calls to `fib` during the computation of `(fib n)` is precisely $Fib(n+1)$. Given that $Fib(n)$ is the closest integer to $\phi^n/\sqrt{5}$ for $\phi = (1 + \sqrt{5})/2$, $Fib(n)$ obviously grows exponentially with the input.

We can also use the following procedure to compute Fibonacci numbers. In this case, the number of calls to `fib-iter` during the computation of `(fib n)` is linear in n .

```
(define (fib n)
  (define (fib-iter a b n)
    (cond ((= n 0) a)
          ((= n 1) b)
          (else (fib-iter b (+ a b) (- n 1)))))
  (fib-iter 0 1 n))
```

The difference in number of steps required by the above two approaches to computing Fibonacci numbers is enormous, even for small inputs.

Example: Counting Change Suppose that we have four kinds of coins with denominations $1c$, $5c$, $10c$ and $25c$. How many distinct ways do we have to make changes of one dollar? In other words, how many distinct nonnegative integer solutions can we find to the following equation?

$$1 \cdot x_1 + 5 \cdot x_2 + 10 \cdot x_3 + 25 \cdot x_4 = 100$$

Clearly, this number is the sum of the numbers of solutions to the following two equations.

$$\begin{aligned} 1 \cdot x_1 + 5 \cdot x_2 + 10 \cdot x_3 + 25 \cdot x_4 &= 99 \\ 5 \cdot x_2 + 10 \cdot x_3 + 25 \cdot x_4 &= 100 \end{aligned}$$

This observation allows us to define the following procedure for counting the number of ways a sum can be exchanged in terms of a given set of coins.

```
(define (exchange coins sum)
  (cond ((zero? sum) 1)
        ((null? coins) 0)
        (else (let ((c (car coins)))
                  (if (< sum c)
                      (exchange (cdr coins) sum)
                      (+ (exchange (cdr coins) sum)
                         (exchange coins (- sum c))))))))))
```

1.2.3 Orders of Growth

1.2.4 Exponentiation

We now consider the problem of computing the exponential of a given number, defined as follows.

$$x^n = \begin{cases} 1 & \text{if } n = 0; \\ x \cdot x^{n-1} & \text{if } n > 0. \end{cases}$$

The following recursive procedure is a direct translation of the definition of the exponential function. This procedure takes $\Theta(n)$ -time and $\Theta(n)$ -space.

```
(define (expt x n)
  (if (zero? n) 1 (* x (expt x (- n 1)))))
```

As in the case of factorial function, we can implement the factorial function as follows via tail-recursion. This procedure takes $\Theta(n)$ -time and $\Theta(1)$ -space.

```
(define (expt x n)
  (define (expt-iter n res)
    (if (zero? n) res (expt-iter (- n 1) (* x res))))
  (expt-iter n 1))
```

Notice that the exponential function can also be defined as follows.

$$x^n = \begin{cases} 1 & \text{if } n = 0; \\ (x^2)^n & \text{if } n > 0 \text{ and } n \text{ is even;} \\ x \cdot (x^2)^{(n-1)/2} & \text{if } n > 0 \text{ and } n \text{ is odd.} \end{cases}$$

This leads to the following $\Theta(\log n)$ -time and $\Theta(1)$ -space procedure that implements the exponential function.

```
(define (expt x n)
  (define (expt-iter x n res)
    (if (zero? n) res
        (if (even? n)
            (expt-iter (* x x) (/ n 2) res)
            (expt-iter (* x x) (/ (- n 1) 2) (* x res))))))
  (expt-iter x n 1))
```

1.2.5 Greatest Common Divisors

```
(define (gcd a b)
  (if (zero? b) a (gcd b (remainder a b))))
```

1.2.6 Example: Testing for Primality

```
(define (smallest-divisor n)
  (find-divisor n 2 (floor (sqrt n))))

(define (find-divisor n i r)
  (cond ((> i r) n)
        ((divides i n) i)
        (else (find-divisor n (1+ i) r))))

(define (divides i n) (zero? (remainder n i)))
```

Theorem 1 (*Fermat's little theorem*) *If p is a prime number, then we have the following equation for every natural number $1 \leq a < p$.*

$$a^{p-1} \equiv 1 \pmod{p}$$

```
(define (expmod base exp m)
  (cond ((zero? exp) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m)) m))
        (else
         (remainder (* base (expmod base (- exp 1) m)) m))))

(define (fermat-test n)
  (let ((a (1+ (random (- n 1)))))
    (= (expmod a (- n 1) n) 1)))

(define (fast-prime? n times)
  (if (zero? times)
      true
      (and (fermat-test n) (fast-prime? n (- times 1)))))
```

1.3 Formulating Abstractions with Higher-Order Procedures

A higher-order procedure is one that manipulates procedures. Higher-order procedures offer a powerful approach to code reuse.

1.3.1 Procedures as Arguments

$$\sum_{n=a}^b n \qquad \sum_{n=a}^b n^2 \qquad \sum_{n=a}^b \frac{1}{4n \cdot (4n + 2)}$$

The following three procedures calculate the above three sums, respective.

```

(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))

(define (sum-squares a b)
  (if (> a b)
      0
      (+ (square a) (sum-integers (+ a 1) b))))

(define (sum-pi/8 a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2))) (sum-pi/8 (+ a 4) b)))

```

Clearly, there is a lot of common in these procedures. The following higher-order procedure *sum* calculates the sum $f(n)$, where n ranges over all numbers between a and b that are of form $a + next * i$ for some integer i . Note that function f is taken as an argument of the procedure *sum*.

```

(define (sum f next a b)
  (if (> a b)
      0
      (+ (f a) (sum f next a b))))

```

Now the three procedures *sum-integers*, *sum-squares*, and *sum-pi/8* can be implemented as follows.

```

(define (sum-integer a b)
  (define (f x) x)
  (sum f inc a b))

(define (sum-squares a b)
  (sum square inc a b))

(define (sum-pi/8 a b)
  (define (next a) (+ a 4))
  (define (f a)
    (/ 1.0 (* a (a + 2))))
  (sum f next a b))

```

The definite integral of a function f between limits a and b can be approximated numerically using the following formula

$$\int_b^a f = \left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + \dots \right] dx$$

for small values of dx . We can express this directly as a procedure.

```

(define (integral f a b dx)
  (define (add-dx a) (+ a dx))
  (* (sum f add-dx (+ a (/ dx 2.0) b)) dx))

```

1.3.2 Constructing Procedures using lambda

We now introduce a language construct `lambda` for constructing anonymous procedures. The following is the format for forming such a procedure.

$$(\text{lambda } (\langle \text{formal parameters} \rangle) \langle \text{body} \rangle)$$

For instance, `(lambda (x) (+ x 4))` defines a procedure that adds 4 to its argument. Furthermore, it should be clear that

```
(define (add4 x) (+ x 4))
```

is equivalent to

```
(define add4 (lambda (x) (+ x 4)))
```

The following is an example of computation involving an anonymous procedure.

$$\begin{aligned} \underline{((\text{lambda } (x \ y) \ (+ \ (* \ x \ x) \ (* \ y \ y))) \ 3 \ 4)} &\rightarrow (+ \ (\underline{* \ 3 \ 3}) \ (* \ 4 \ 4)) \\ &\rightarrow (+ \ 9 \ (\underline{* \ 4 \ 4})) \rightarrow (+ \ 9 \ 16) \rightarrow 25 \end{aligned}$$

- Use `let` to create local variables

1.3.3 Procedure as General Methods

We define a procedure to find roots of functions by half-interval method.

```
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond ((positive? test-value) (search f neg-point midpoint))
                ((negative? test-value) (search f midpoint pos-point))
                (else midpoint))))))
```

```
(define (close-enough? a b)
  (< (- a b) 0.001))
```

```
(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-value (f b)))
    (cond ((and (negative? a-value) (positive? b-value))
           (search f a b))
          ((and (positive? a-value) (negative? b-value))
           (search f b a))
          (else (error "Values are not of opposite signs")))))
```

We now define a procedure finding fixed points of functions. A point x is a fixed point of a function f if $f(x) = x$. For some functions f , we can locate a fixed point by beginning with an initial guess and then applying f repeatedly,

$$x, f(x), f(f(x)), \dots$$

until the values do not change very much.

```
(define tolerance 0.000001)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
```

```
(let ((next (f guess)))
  (if (close-enough? guess next)
      next
      (try next))))
(try first-guess))
```

1.3.4 Procedures as Returned Values

Here is a procedure that composes two given unary functions.

```
(define (compose f g) (lambda (x) (g (f (x)))))
```

We can also define a function `twice` as follows. Given a function f , `twice` returns function which is essentially equivalent to applying f twice.

```
(define (twice f) (compose f f))
```

Let `succ` be the usual successor function on natural numbers. What is then the value of `((twice twice) succ) 0`?

Continuation Sometimes, we may be in a scenario where we need to return a procedure that roughly indicates what needs to be done after a call to a given procedure is finished.

```
(define (fact-cont n k)
  (if (= n 0)
      (k 1)
      (fact-cont (- n 1) (lambda (res) (k (* n res))))))
```

```
(define (fact n) (fact n (lambda (x) x)))
```

```
(define (fib-cont n k)
  (cond ((= n 0) (k 0))
        ((= n 1) (k 1))
        (else (fib-cont (- n 1)
                        (lambda (res1)
                          (fib-cont (- n 2)
                                    (lambda (res2) (k (+ res1 res2))))))))))
```

```
;;; a lame version
```

```
(define (fib n) (fib-cont n (lambda (x) x)))
```

```
;;; a version using call-with-current-continuation
```

```
(define (fib n) (call-with-current-continuation (lambda (k) (fib-cont n k))))
```

Chapter 2

Building Abstraction with Data

2.1 Introduction to Data Abstraction

2.1.1 Example: Arithmetic Operations for Rational Numbers

We use the following example of implementing a package for rational numbers to illustrate some concepts in data abstraction. Let us begin by assuming that we can already construct a rational number equal to p/q when two integers p and q are given and can also extract both the numerator and the denominator of a rational number. In particular, we have the procedures `make-rat`, `numer` and `denom`.

- `(make-rat n d)` returns a rational number whose value equals n/d .
- `(numer x)` returns the numerator of the rational number x .
- `(denom x)` returns the denominator of the rational number x .

We now implement some arithmetic operations on rational numbers plus a print procedure in Figure 2.1. We can represent a rational number as a pair where the first and the second components of the pair are the numerator and the denominator of the rational number. The invariants we impose on such a pair are

- The denominator is always positive.
- The nominator and the denominator are relatively prime, that is, the greatest common divisor of them is 1.

The following is an implementation of `make-rat`, `numer` and `denom`.

```
(define (make-rat p q)
  (if (zero? q)
      (error "make-rat: denominator is zero")
      (let ((r (gcd p q)))
        (let ((p0 (/ p r))
              (q0 (/ q r)))
          (if (negative? q0)
              (cons (- p0) (- q0))
              (cons p0 q0))))))

(define (numer x) (car x))

(define (denom x) (cdr x))
```

```
(define (neg-rat x)
  (let ((xn (numer x))
        (xd (denom x)))
    (make-rat (- xn) xd)))

(define (add-rat x y)
  (let ((xn (numer x)) (xd (denom x))
        (yn (numer y)) (yd (denom y)))
    (make-rat (+ (* xn yd) (* xd yn)) (* xd yd))))

(define (sub-rat x y)
  (let ((xn (numer x)) (xd (denom x))
        (yn (numer y)) (yd (denom y)))
    (make-rat (- (* xn yd) (* xd yn)) (* xd yd))))

(define (mul-rat x y)
  (let ((xn (numer x)) (xd (denom x))
        (yn (numer y)) (yd (denom y)))
    (make-rat ((* xn yn) (* xd yd)))))

(define (div-rat x y)
  (let ((xn (numer x)) (xd (denom x))
        (yn (numer y)) (yd (denom y)))
    (make-rat ((* xn yd) (* xd yn)))))

(define (print-rat x)
  (format #t "~S/~S" (numer x) (denom x)))
```

Figure 2.1: Arithmetic Operations for Rational Numbers

2.1.2 Abstraction Barriers

2.1.3 What Is Meant by Data?

2.1.4 Extended Exercise: Interval Arithmetic

2.2 Hierarchical Data and the Closure Property

2.2.1 Representing Sequences

We define various list operations as follows.

```

(define (append xs ys)
  (if (null? xs) ys (cons (car xs) (append (cdr xs) ys))))

;;; not tail-recursive
(define (reverse xs)
  (append (reverse (cdr xs)) (list (car xs))))

;;; tail-recursive
(define (revAppend xs ys)
  (if (null? xs) ys (revAppend (cdr xs) (cons (car xs) ys))))

(define (reverse xs) (revAppend xs ()))

;;; not tail-recursive
(define (length xs)
  (if (null? xs) 0 (1 + (length (cdr xs)))))

;;; tail-recursive
(define (length xs)
  (define (len xs n)
    (if (null? xs) n (len (cdr xs) (1+ n))))
  (len xs 0))

;;; a generic version of map function is provided in Scheme
(define (map f xs)
  (if (null? xs) () (cons (f (car xs)) (map f (cdr xs)))))

(define (app f xs)
  (if (null? xs) () (begin (f (car xs)) (app f (cdr xs)))))

```

2.2.2 Hierarchical Structures

2.2.3 Sequence as Conventional Interfaces

```

;;; fold-left f init (a1 a2 ... an) = (f (...(f (f init a1) a2)...)) an)
(define (fold-left f init seq)
  (if (null? seq)
      init
      (fold-left f (f init (car seq)) (cdr seq))))

;;; fold-right f init (a1 a2 ... an) = (f a1 (f a2 (...(f init an)...)))

```

```

(define (fold-right f init seq)
  (if (null? seq)
      init
      (f (car seq) (fold-right f init (cdr seq)))))

(define (interval m n)
  (define (aux m n res)
    (if (< n m) res (aux (1+ m) n (cons m res))))
  (aux m n ()))

(define (factorial-l n)
  (fold-left * 1 (interval 1 n)))

(define (factorial-r n)
  (fold-right * 1 (interval 1 n)))

;;; (permute xs) returns a list of all permutations of xs

(define (permute s)
  (if (null? s)
      (list nil)
      (flatmap (lambda (x)
                 (map (lambda (p) (cons x p))
                     (permute (remove x s))))
                s)))

(define (flatmap f xss)
  (fold-left append nil (map f xss)))

(define (remove item sequence)
  (filter (lambda (x) (not (= x item))) sequence))

(define (permute xs)
  (define (aux xs ys res)
    (if (null? xs)
        res
        (aux (cdr xs)
              (cons (car xs) ys)
              (append (map (lambda (zs) (cons (car xs) zs))
                          (permute (revAppend ys (cdr xs))))
                      res))))
  (if (null? xs) (list nil) (aux xs () ())))

```

2.3 Symbolic Data

2.3.1 Quotation

Expressions in Scheme are simply represented as lists. For instance, we have expressions like the following.

```
(* (+ 1 2) (- 3 4)) and (define (fact n) (if (zero? n) 1 (* n (fact (- n 1)))))
```

However, such expressions are not values and they are to be evaluated if given to the interpreter. To stop such evaluation, we introduce quotation.

For instance, `(+ 1 2)` is an expression that evaluates to `3`. On the other hand, `'(+ 1 2)` evaluates to a list representing the three element sequence `+`, `1` and `2`.

2.3.2 Example: Symbolic Differentiation

<code>(variable? e)</code>	Is <i>e</i> a variable?
<code>(same-variable? v1 v2)</code>	Are <i>v1</i> and <i>v2</i> the same variable?
<code>(sum? e)</code>	Is <i>e</i> a sum?
<code>(addend e)</code>	Addend of the sum <i>e</i> .
<code>(augend e)</code>	Augend of the sum <i>e</i> .
<code>(make-sum e1 e2)</code>	Construct the sum of <i>e1</i> and <i>e2</i> .
<code>(multiplier e)</code>	Multiplier of the product <i>e</i> .
<code>(multiplicand e)</code>	Multiplicand of the product <i>e</i> .
<code>(make-product e1 e2)</code>	Construct the product of <i>e1</i> and <i>e2</i> .
<code>(product? e)</code>	Is <i>e</i> a product?

2.3.3 Example: Representing Sets

- Sets as unordered lists

```
(define (intersect-set set1 set2)
  (if (null? set1)
      nil
      (if (element-of-set? (car set1) set2)
          (cons (car set1) (intersect-set (cdr set1) set2))
          (intersect-set (cdr set1) set2))))
```

- Sets as ordered lists

```
(define (intersect-set set1 set2)
  (if (or (null? set1) (null? set2))
      nil
      (let ((x1 (car set1))
            (x2 (car set2)))
        (cond ((= x1 x2)
               (cons x1 (intersection-set (cdr set1) (cdr set2))))
              ((< x1 x2) (intersect-set (cdr set1) set2))
              ((< x2 x1) (intersect-set set1 (cdr set2)))))))
```

- Sets as binary trees

```
(define (adjoin-set x set)
  (cond ((null? set) (make-tree x nil nil))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree (entry set)
                    (adjoin-set x (left-branch set))
```

```

(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)
         (let ((m1 (multiplier exp))
               (m2 (multiplicand exp)))
           (make-sum (make-product (deriv m1 var) m2)
                     (make-product m1 (deriv m2 var)))))
        (else (error "deriv: unknown expression: " exp))))

(define (variable? e) (symbol? e))

(define (same-variable? e1 e2) (eq? e1 e2))

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

(define (addend e) (cadr e))

(define (augend e) (caddr e))

(define (make-sum e1 e2)
  (cond ((eq? e1 0) e2)
        ((eq? e2 0) e1)
        ((and (number? e1) (number? e2)) (+ e1 e2))
        (else (list '+ e1 e2))))

(define (product? e) (and (pair? e) (eq? '* (car e))))

(define (multiplier e) (cadr e))

(define (multiplicand e) (caddr e))

(define (make-product e1 e2)
  (cond ((eq? e1 0) 0)
        ((eq? e2 0) 0)
        ((eq? e1 1) e2)
        ((eq? e2 1) e1)
        ((and (number? e1) (number? e2)) (* e1 e2))
        (else (list '* e1 e2))))

```

Figure 2.2: An implementation of symbolic differentiation

```

                (right-branch set)))
  (> x (entry set))
  (make-tree (entry set)
             (left-branch set)
             (adjoin-set x (right-branch set))))))

```

2.3.4 Example: Hoffman Encoding Trees

Initial leaves	{(A 8) (B 3) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1)}
Merge	{(A 8) (B 3) ({CD} 2) (E 1) (F 1) (G 1) (H 1)}
Merge	{(A 8) (B 3) ({CD} 2) ({EF} 2) (G 1) (H 1)}
Merge	{(A 8) (B 3) ({CD} 2) ({EF} 2) ({GH} 2)}
Merge	{(A 8) (B 3) ({CD} 2) ({EFGH} 4)}
Merge	{(A 8) ({BCD} 5) ({EFGH} 4)}
Merge	{(A 8) ({BCDEFGH} 9)}

```
(define (make-leaf symbol weight) (list 'leaf symbol weight))
```

```
(define (leaf? tree)
  (and (pair? tree) (eq? (car tree) 'left)))
```

```
(define (symbol-leaf leaf) (second leaf))
(define (weight-leaf leaf) (third leaf))
```

```
(define (make-code-tree left right)
  (list left right
        (append (symbols left) (symbols right))
        (+ (weight left) (weight right))))
```

```
(define (symbols tree)
  (if (leaf? tree) (list (symbol-leaf tree)) (third tree)))
```

```
(define (weight tree)
  (if (leaf? tree) (weight-leaf tree) (fourth tree)))
```

2.4 Multiple Representations for Abstract Data

2.4.1 Representations for Complex Numbers

Arithmetic Operations

```
(define (add-complex z1 z2)
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                       (+ (imag-part z1) (imag-part z2))))
```

```
(define (sub-complex z1 z2)
  (make-from-real-imag (- (real-part z1) (real-part z2))
                       (- (imag-part z1) (imag-part z2))))
```

```
(define (mul-complex z1 z2)
```

```

(make-from-mag-ang (* (magnitude z1) (magnitude z2))
                  (+ (angle z1) (angle z2)))

(define (div-complex z1 z2)
  (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
                    (- (angle z1) (angle z2))))

```

To complete the complex number package, we must choose a representation. There are two obvious ways to do this: rectangular form and polar form. Which shall we choose?

Rectangular Coordinates

```

(define (real-part z) (car z))
(define (imag-part z) (cdr z))

(define (magnitude z)
  (sqrt (square (real-part z)) (square (imag-part z))))

(define (angle z) (atan (imag-part z) (real-part z)))

(define (make-from-real-imag x y) (cons x y))
(define (make-from-mag-ang r a)
  (cons (* r (cos a)) (* r (sin a))))

```

Polar Coordinates

```

(define (real-part z) (* (magnitude z) (cos (angle z))))
(define (imag-part z) (* (magnitude z) (sin (angle z))))

(define (magnitude z) (car z))
(define (angle z) (cdr z))

(define (make-from-real-imag x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)))

(define (make-from-mag-ang r a) (cons r a))

```

2.4.2 Tagged Data

Tagging is a common approach that allows certain information on data to be revealed at run-time by associating tags with data.

```

(define (attach-tag type-tag contents)
  (cons type-tag contents))

(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Bad tagged value -- TYPE-TAG" datum)))

```

Here are some constructors and destructors for polar representation.

```

(define (real-part-rectangular z) (car z))

(define (imag-part-rectangular z) (cdr z))

(define (magnitude-rectangular z)
  (sqrt (+ (square (real-part-rectangular z))
           (square (imag-part-rectangular z)))))

(define (angle-rectangular z)
  (atan (imag-part-rectangular z)
        (real-part-rectangular z)))

(define (make-from-real-imag-rectangular x y)
  (attach-tag 'rectangular (cons x y)))

(define (make-from-mag-ang-rectangular r a)
  (attach-tag 'rectangular (cons (* r (cons a) (* r (sin a))))))

```

Here are some constructors and destructors for polar representation.

```

(define (real-part-polar z) (* (magnitude-polar z) (cos (angle-polar z))))

(define (imag-part-polar z) (* (magnitude-polar z) (sin (angle-polar z))))

(define (magnitude-polar z) (car z))

(define (angle-polar z) (cdr z))

(define (make-from-real-imag-polar x y)
  (attach-tag 'polar
              (cons (sqrt (+ (square x) (square y)))
                    (atan y x))))

(define (make-from-mag-ang-polar r a)
  (attach-tag 'polar (cons r a)))

```

Here are some generic constructors and destructors.

```

(define (real-part z)
  (cond ((rectangular? z)
        (real-part-rectangular (contents z)))
        ((polar? z)
        (real-part-polar (contents z)))
        (else (error "Unknown type -- REAL-PART" z))))

(define (imag-part z)
  (cond ((rectangular? z)
        (imag-part-rectangular (contents z)))
        ((polar? z)
        (imag-part-polar (contents z)))
        (else (error "Unknown type -- IMAG-PART" z))))

```

There are some serious drawbacks with this approach.

1. Probably, the most serious drawback is the need for modifying source code if one wants to add a new implementation of complex numbers into the system.
2. Also, the requirement that no two implementations share a common function name may also be a severe limitation, though this seems to be manageable by adopting some naming convention.

We can now implement arithmetic operations on complex numbers as follows.

```
(define (add-complex z1 z2)
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                       (+ (imag-part z1) (imag-part z2))))

(define (make-from-real-imag x y)
  (make-from-real-imag-rectangular x y))
```

2.4.3 Data-Directed Programming and Additivity

```
(load-option 'hash-table)

(define *opname-type-opcode-table* (make-equal-hash-table 23))

(define (put opname type opcode)
  (hash-table/put! *opname-type-opcode-table* (cons opname type) opcode))

(define (get opname type)
  (let ((opcdoe
        (hash-table/get *opname-type-opcode-table*
                        (cons opname type) 'undefined)))
    (if (eq? opcode 'undefined)
        (error "undefined operation -- GET: " (cons opname type))
        opcode)))
```

Here is the code for installing a complex number package that is implemented using rectangular representation.

```
(define (install-rectangular-package)
  ;; internal procedures
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (magnitude z)
    (sqrt (+ (square (real-part z)) (square (imag-part z)))))
  (define (angle z) (atan (imag-part z) (real-part z)))
  ;;; ...
  ;;; ...
  ;;; interface to the rest of the system
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  ;;; ...
```

```
;;; ...
'done)
```

Here is the code for installing a complex number package that is implemented using polar representation.

```
(define (install-polar-package)
  ;;; internal procedures
  (define (magnitude z) (car z))
  (define (angle z) (cdr z))
  (define (real-part z) (* (magnitude z) (cos (angle z))))
  (define (imag-part z) (* (magnitude z) (sin (angle z))))
  ;;; ...
  ;;; ...
  ;;; interface to the rest of the system
  (put 'real-part '(polar) real-part)
  (put 'imag-part '(polar) imag-part)
  (put 'magnitude '(polar) magnitude)
  (put 'angle '(polar) angle)
  ;;; ...
  ;;; ...
'done)
```

Here is the code for defining generic operations.

```
(define (apply-generic opname . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get opname type-tags)))
      (if proc
          (apply proc (map contents args))
          (error "No method for these types -- APPLY-GENERIC"
                 (list op type-tags))))))

(define (real-part z) (apply-generic 'real-part z))
(define (imag-part z) (apply-generic 'imag-part z))
(define (magnitude z) (apply-generic 'magnitude z))
(define (angle z) (apply-generic 'angle z))
;;; ...
```

We have now addressed both of the name sharing problem and the code modification issue.

Message Passing

```
(define (make-from-real-imag x y)
  (define (dispatch op)
    (cond ((eq? op 'real-part) x)
          ((eq? op 'imag-part) y)
          ((eq? op 'magnitude)
           (sqrt (+ (square x) (square y))))
          ((eq? op 'angle) (atan y x))
          (else (error "Unknown op -- MAKE-FROM-REAL-IMAG" op))))
  dispatch)

(define (apply-generic op arg) (arg op))
```

2.5 Systems with Generic Operations

2.5.1 Generic Arithmetic Operations

```
(define (add x y) (apply-generic 'add x y))
(define (sub x y) (apply-generic 'sub x y))

(define (install-scheme-number-package)
  (define (tag x) (attach-tag 'scheme-number x))
  (put 'add '(scheme-number scheme-number)
      (lambda (x y) (tag (+ x y))))
  (put 'sub '(scheme-number scheme-number)
      (lambda (x y) (tag (- x y))))
  ...
  (put 'make 'scheme-number (lambda (x) (tag x)))
  'done)

(define (make-scheme-number n)
  ((get 'make 'scheme-number) n))

(define (install-rational-package)
  ;; internal procedures
  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (make-rat n d) ...)
  (define (add-rat x y) ...)
  (define (sub-rat x y) ...)
  ...

  ;; interface to rest of the system
  (define (tag x) (attach-tag 'rational x))
  (put 'add '(rational 'rational) (tag (add-rat x y)))
  (put 'sub '(rational 'rational) (tag (sub-rat x y)))
  ...
  'done)
```

2.5.2 Combining Data of Different Types

```
(define (apply-generic op . args)
  (let ((type-args (map type-arg args)))
    (let ((proc (get op type-args)))
      (if proc
          (apply proc (map contents args))
          (if (= (length args) 2)
              (let ((t1 (first type-args))
                    (t2 (second type-args))
                    (a1 (first args))
                    (a2 (second args)))
                (let ((t1->t2 (get-coercion t1 t2))
                      (t2->t1 (get-coercion t2 t1)))
                  (cond (t1->t2 (apply-generic op (t1->t2 a1) a2))
```

```
(t2->t1 (apply-generic op (t1->t2 a1) a2))
(else (error "No method for these types: "
            (list op type-args))))))
(error "No method for these types"
      (list op type-args))))))
```

This coercion strategy has many advantages over the method of defining explicit cross-type operations. Although we still need to write coercion procedures to relate the types (possibly n^2 procedures for a system with n types), we need to write only one procedure for each pair of types rather than a different procedure for each collection of types and each generic operation. What we are counting on here is the fact that the appropriate transformation between types depends only on the types themselves, not on the operation to be applied.

Hierarchies of Types

Inadequacies of Hierarchies

Chapter 3

Modularity, Objects and State

One powerful design strategy, which is particular appropriate to the construction of programs for modeling physical systems, is to base the structure of our programs on the structure of the system being modeled.

3.1 Assignment and Local State

3.1.1 Local State Variables

```
(define balance 100)

(define (warning msg) (display msg))

(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      (warning "Insufficient funds")))

(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          (warning "Insufficient funds")))))

(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        (warning "Insufficient funds"))))

(define (make-withdraw init)
  (let ((balance init))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          (warning "Insufficient funds")))))
```

```

        balance)
      (warning "Insufficient funds")))))

(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
                balance)
        (warning "Insufficient funds")))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT" m))))))

(define paul-acc (make-account 100))

```

3.1.2 The Benefits of Introducing Assignment

```

(define rand
  (let ((x random-update))
    (lambda ()
      (set! x (rand-update x)) x)))

(define (estimate-pi trials)
  (sqrt (/ 6 monte-carlo trials cesaro-test)))

(define (cesaro-test)
  (= (gcd (rand) (rand)) 1))

(define (monte-carlo trials experiment)
  (define (iter trials-remaining trials-passed)
    (cond ((= trials-remaining 0) (/ trials-passed trials))
          (else (if (experiment)
                     (iter (- trials-remaining 1) (+ trials-passed 1))
                     (iter (- trials-remaining 1) trials-passed)))))
  (iter trials 0))

(define (estimate-pi trials)
  (sqrt (/ 6 (random-gcd-test trials random-init))))

;;; How do we write a cesaro-test?

(define (random-gcd-test trials initial-x)
  (define (iter trials-remaining trials-passed x)
    (let* ((x1 (rand-update x))
           (x2 (rand-update x1)))
      (cond ((= trials-remaining 0)

```

```

        (/ trials-passed trials))
      (= (gcd x1 x2) 1)
      (iter (- trials-remaining 1) (+ trials-passed 1) x2))
      (else
        (iter (- trials-remaining 1) trials-passed x2))))))
(iter trials 0 initial-x))

```

While the program is still simple, it betrays some painful breaches of modularity.

3.1.3 The Cost of Introducing Assignment

Sameness and Change

Please think in terms of pointer comparison!

Pitfalls of Imperative Programming

```

(define (factorial n)
  (define (iter product counter)
    (if (> counter x)
        product
        (iter (* counter product) (1+ counter))))
  (iter 1 1))

```

```

(define (factorial n)
  (let ((product 1)
        (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! product (* counter product))
                  (set! counter (1+ counter))
                  (iter))))
    (iter))

```

3.2 The Environment Model of Evaluation

An environment is a sequence of frames. Each frame is a table (possibly empty) of bindings, which associates variable names with their corresponding values.

3.2.1 The Rules for Evaluation

The low representation of a function is really a pair such that one component is a pointer to code and the other is a pointer to an environment.

3.2.2 Applying Simple Procedures

```

(define square (lambda (x) (* x x)))
(define sum-of-squares (lambda (x y) (+ (square x) (square y))))
(define (f a) (sum-of-squares (+ a 1) (+ a 2)))

```

Suppose we apply f to 5, that is, we evaluate $(f\ 5)$.

3.2.3 Frames as the Repository of Local State

3.2.4 Internal Definition

```
(define (f x y)
  (define (g z) (x + y + z))
  g)
```

3.3 Modelling with Mutable Data

```
(define x (cons 0 '()))
(set-cdr! x x)
```

What is the value stored in x ?

3.3.1 Mutable List Structure

We use the following program to illustrate the point that mutation is just assignment.

```
(define (pair x y)
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          ((eq? m 'set-car!) (lambda (v) (set! x v)))
          ((eq? m 'set-cdr!) (lambda (v) (set! x v)))
          (else (error "unknown message -- PAIR: " m))))
  dispatch)
```

3.3.2 Representing Queues

- A constructor: `(make-queue)`: returns an empty queue.
- Two selectors:
 1. `(empty-queue? <queue>)`: tests whether a queue is empty.
 2. `(front-queue <queue>)`: returns the front of the queue (if it is not empty) or signals an error (if it is empty).
- Two mutators:
 - `(insert-queue! <queue> <item>)`: insert the element at the rear of the queue and return the modified queue as its value.
 - `(delete-queue! <queue> <item>)`: deletes the front element of the queue and returns the modified queue as its value, signaling an error if the queue is empty before the deletion.

3.3.3 Representing Tables

One-dimensional tables

```
(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (if record (cdr record) false)))
```

```

(define (make-queue)
  (define queue '())
  (define rear-ptr '())

  (define last)

  (define (dispatch msg)
    (cond ((eq? 'empty-queue? (car msg))
           (null? queue))
          ((eq? 'front-queue (car msg))
           (if (null? queue)
               (error "FRONT called with an empty queue")
               (car queue)))
          ((eq? 'insert-queue! (car msg))
           (set! last (cons (cadr msg) '()))
           (if (null? queue)
               (begin (set! queue last)
                      (set! rear-ptr last))
               (begin (set-cdr! rear-ptr last)
                      (set! rear-ptr last))))
          ((eq? 'delete-queue! (car msg))
           (if (null? queue)
               (error "DELETE-QUEUE: empty queue")
               (set! queue (cdr queue))))
          ((eq? 'print-queue (car msg))
           (display queue))
          (else (error "MAKE-QUEUE: unknow message: " msg))))
  dispatch)

(define (empty-queue? queue)
  (queue (list 'empty-queue?)))

(define (front-queue queue)
  (queue (list 'front-queue)))

(define (insert-queue! queue item)
  (queue (list 'insert-queue! item))
  queue)

(define (delete-queue! queue)
  (queue (list 'delete-queue!))
  queue)

(define (print-queue queue)
  (queue (list 'print-queue)))

```

Figure 3.1: Implementation Queue in the Message-Passing Style

```

(define (assoc key records)
  (cond ((null? records) false)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records)))))

(define (insert! key value table)
  (let ((record (lookup key table)))
    (if record
        (set-cdr! record value)
        (set-cdr! table (cons (cons key value) (cdr table)))))
  'ok)

```

Two-dimensional tables

3.3.4 A Simulator for Digital Circuits

3.4 Concurrency: Time Is of the Essence

3.4.1 The natural of time in Concurrent Systems

Suppose that Alice and Bob share a bank account. They implement the following withdraw function as two independent processes sharing a common variable `balance`.

```

(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      'insufficient funds'))

```

Let us imagine some scenarios.

3.4.2 Mechanisms for Controlling Concurrency

Serializing access to shared state

Serializers in Scheme

```

(define x 10)

(parallel-execute
 (lambda () (set! x (* x x)))
 (lambda () (set! x (+ x 1))))

(define s (make-serializer))

(parallel-execute
 (s (lambda () (set! x (* x x))))
 (s (lambda () (set! x (+ x 1)))))

```

Let us now revisit the problem of implementing a bank account.

```

(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
                balance)
        'insufficient funds)))

(define (deposit amount)
  (set! balance (+ balance amount))
  balance)

(let ((protected (make-serializer)))
  (define (dispatch msg)
    (cond ((eq? msg 'withdraw) (protected withdraw))
          ((eq? msg 'deposit) (protected deposit))
          (else (error 'Unknown request -- MAKE-ACCOUNT'))))
  dispatch)
'ok)

```

Implementing Serializers

```

(define (make-serializer)
  (let ((mutex (make-mutex)))
    (lambda (p)
      (define (serialized-p . args)
        (mutex 'acquire)
        (let ((val (apply p args)))
          (mutex 'release)
          val))
        serialized-p)))

(define (make-mutex)
  (let ((cell (list false)))
    (define (the-mutex m)
      (cond ((eq? m 'acquire)
             (if (test-and-set! cell)
                 (the-mutex 'acquire))) ; retry
            ((eq? m 'release) (clear! cell))))
    the-mutex)

(define (clear! cell)
  (set-car! cell false))

(define (test-and-set! cell)
  (if (car cell)
      true
      (begin (set-car! cell true)
              false)))

```

Deadlock**Synchronization****3.5 Streams****3.5.1 Streams Are Delayed Lists**

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count) (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

```
(define (sum-primes a b)
  (folder-left + 0 (filter prime? (enumerate-interval a b))))
```

The latter implementation of `sum-primes` is clear but extremely inefficient. We now introduce the notion of lazy evaluation to address the inefficiency problem. We are to implement a data structure for streams to support lazy evaluation. Clearly, we need to have the following properties on streams, where `stream-car` and `stream-cdr` are two selectors and `cons-stream` is a constructor.

$$\begin{aligned} (\text{stream-car } (\text{cons-stream } x \ y)) &\Rightarrow x \\ (\text{stream-cdr } (\text{cons-stream } x \ y)) &\Rightarrow y \end{aligned}$$

```
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))
```

```
(define (cons-stream x xs)
  (cons x (delay xs)))
```

```
(define (stream-car stream) (car stream))
(define (stream-cdr stream) (force (cdr stream)))
```

The stream implementation in action

```
(stream-car
 (stream-cdr
  (stream-filter prime? (stream-enumerate-interval 1000 10000))))
```

```
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low (stream-enumerate-interval (1+ low) high))))
```

Implementing *delay* and *force*

```
(defmacro (delay exp) (lambda () exp))
(define (force delayed-object) (delayed-object))
```

```
(define (memo-proc proc)
  (let ((already-run? false)
        (result? false))
    (lambda ()
      (if (not already-run)
          (begin (set! result (proc))
                  (set! already-run? true)
                  result)
          result))))

(defmacro (delay exp) (memo-proc (lambda () exp)))
```

3.5.2 Infinite Streams

```
(define (divisible? x y) (= (remainder x y) 0))

(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))

(define naturals (integers-starting-from 0))

(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7))) integers))

(define (fibgen a b)
  (cons-stream a (fibgen (b (+ a b))))))

(define fibs (fibgen 0 1))

(define (fib n) (stream-ref fibs n))
```

Let us now implement the famous sieve of Eratosthenes.

```
(define (sieve stream)
  (cons-stream (car stream)
               (sieve (stream-filter
                       (lambda (x) (> (remainder x (car stream)) 0))
                       (cdr stream)))))

(define primes (sieve (integers-starting-from 2)))
```

Define Streams Implicitly

```
(define (stream-map2 f s1 s2)
  (if (stream-null? s1)
      the-empty-stream
      (cons-stream (f (stream-car s1) (stream-car s2))
                    (stream-map2 f (stream-cdr s1) (stream-cdr s2)))))

(define (add-streams s1 s2) (stream-map2 + s1 s2))
```

```
(define fibs
  (cons-stream 0
    (cons-stream 1
      (add-streams (stream-cdr fibs) fibs))))
```

Here is another implementation of the sieve of Eratosthenes.

```
(define primes
  (cons-stream 2 (stream-filter prime? (integers-starting-from 3))))

(define (prime? n)
  (define (iter ps)
    (cond ((> (square (car ps)) n) true)
          ((divisible n (car ps)) false)
          (else (iter (cdr ps)))))
  (iter primes))
```

Notice the subtlety here. We need the property that $p_{k+1} \leq p_k$ for $k = 1, \dots$, where p_k stands for the k th prime number.

3.5.3 Exploiting the Stream Paradigm

Formulating iterations as stream processes

```
(define (sqrt-improve guess x)
  (average guess (/ x guess)))

(define (sqrt-stream x)
  (define guesses
    (cons-stream 1.0
      (stream-map (lambda (guess)
                    (sqrt-improve guess x) guesses))))
  guesses)
```

The following is the Euler's transform.

$$S_{n+1} = \frac{(S_{n+1} - S_n)^2}{S_{n-1} - 2S_n + S_{n+1}}$$

Let us now implement the Euler's transform.

```
(define (partial-sums s)
  (define (aux sum s)
    (cons-stream sum (aux (+ sum (stream-car s)) (stream-cdr s))))
  (aux (stream-car s) (stream-cdr s)))

(define pi-summand-stream
  (let ((aux (lambda (sign n)
                (cons-stream (/ sign n) (aux (- sign) (+ n 2))))))
    (aux 1 1)))

(define pi-stream
  (cons-stream (partial-sums pi-summand-stream) 4))
```

```

(define (euler-transform s)
  (let ((s0 (steam-ref s 0))
        (s1 (steam-ref s 1))
        (s2 (steam-ref s 2)))
    (cons-stream (- s2 (/ (square (- s2 s1)) (+ s0 (* -2 s1) s2)))
                 (euler-transform (stream-cdr s)))))

(define accelerated-pi-stream (euler-transform pi-stream))

(define (make-tableau transform s)
  (cons-stream s (make-tableau transform (transform s))))

(define accelerated-accelerated-pi-stream
  (stream-map stream-car (make-tableau euler-transform pi-stream)))

```

Infinite Streams of Pairs

```

(define (interleave s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (interleave s2 (stream-cdr s1)))))

(define (pairs s1 s2)
  (cons-stream (list (stream-car s1) (stream-car s2))
               (interleave
                (stream-map (lambda (x) (list (stream-car s1) s2)))
                (pairs (stream-cdr s1) (stream-cdr s2)))))

```

Streams as Signals

```

(define (integral integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (add-streams (scale-stream integrand dt) int)))
  int)

```

3.5.4 Streams and Delayed Evaluation

```

(define (solve f y0 dt)
  (define y (integral dy y0 dt))
  (define dy (stream-map f y))
  y)

(define (integral delayed-integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (add-streams
                  (scale-stream (force delayed-integrand) dt) int)))
  int)

```

```
(define (solve f y0 dt)
  (define y (integral (delay dy) y0 dt))
  (define dy (stream-map f y))
  y)
```

Chapter 4

Metalinguistic Abstraction

4.1 The Metacircular Evaluation

We are to address four issues.

- The evaluator enables us to deal with nested expressions.
- The evaluator allows us to use variables.
- The evaluator allows us to define compound procedure.
- The evaluator provides special forms, which must be evaluated differently from procedure calls.

4.1.1 The Core of the Evaluator

Eval

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-assignment exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                          (lambda-body exp)
                          env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                  (list-of-values (operands exp) env)))
        ((else (error '“Unknown expression type” -- EVAL’ exp))))))
```

Apply

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
```

```

    (apply-primitive-procedure procedure arguments))
  ((compound-procedure? procedure)
   (eval-sequence
    (procedure-body procedure)
    (extend-environment
     (procedure-parameters procedure)
     arguments
     (procedure-environment procedure))))
  (else (error '“Unknown procedure type -- APPLY” procedure))))

```

Procedure Arguments

```

(define (list-of-values exps env)
  (if (no-operands? exps)
      nil
      (cons (eval (first-operand exps) env)
            (list-of-values (cdr exps) env))))

```

Conditional Arguments

```

(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))

```

Sequences

```

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

```

Assignments and Definitions

```

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                        (eval (assignment-value exp) env)
                        env)
  'ok)

```

```

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  'ok)

```

4.1.2 Representing Expressions

```

(define (self-evaluating? exp)
  (cond ((number? exp) true)
        ((string? exp) true)

```

```

    (else false)))

(define (variable? exp) (symbol? exp))

(define (quoted? exp)
  (tagged-list? exp 'quote))

(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))

(define (text-of-quotation exp) (cadr exp))

(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (second exp))
(define (assignment-value exp) (third exp))

```

Representing Procedures

```

(define (make-procedure parameter body env)
  (list 'procedure parameter body env))

```

Operations on Environment

- (lookup-variable-value var env)
- (extended-environment variables values base-env)
- (define-variable! var value env)
- (set-variable-value! var value env)

4.1.3 Separating Syntactic Analysis from Execution

```

(define (analyze exp)
  (cond ((self-evaluating? exp)
        (analyze-self-evaluating exp))
        ((assignment? exp) (analyze-assignment exp))
        ((definition? exp) (analyze-definition exp))
        ((if? exp) (analyze-if exp))
        (else (error 'Unknown expression type -- ANALYZE' exp))))

(define (analyze-variable exp)
  (lambda (env) (lookup-variable-value exp env)))

(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env)

```

```
(execute-application (fproc env)
                    (map (lambda (aproc) (aproc env)) aprocs))))

(define (execute-application proc args)
  (cond ((primitive-procedure? proc)
        (apply-primitive-procedure proc args))
        ((compound-procedure> proc)
         (procedure-body proc)
         (extend-environment (procedure-parameters proc)
                             args
                             (procedure-environment proc)))
        (else (error 'Unknown procedure type -- EXECUTE-APPLICATION' proc))))
```

Chapter 5

Computing with Register Machines

5.1 Designing Register Machines

5.1.1 Instruction Summary

A controller instruction in our register-machine language has one of the following forms, where each $\langle input_i \rangle$ is either `(reg $\langle register-name \rangle$)` or `(reg $\langle constant-value \rangle$)`.

```
(assign  $\langle register-name \rangle$  (reg  $\langle register-name \rangle$ ))
(assign  $\langle register-name \rangle$  (reg  $\langle constant-value \rangle$ ))
(assign  $\langle register-name \rangle$  (op  $\langle operation-name \rangle$ )  $\langle input_1 \rangle \dots \langle input_n \rangle$ )
(perform (op  $\langle operation-name \rangle$ )  $\langle input_1 \rangle \dots \langle input_n \rangle$ )
(test (op  $\langle operation-name \rangle$ )  $\langle input_1 \rangle \dots \langle input_n \rangle$ )
(branch (label  $\langle label-name \rangle$ ))
(goto (label  $\langle label-name \rangle$ ))
(assign  $\langle register-name \rangle$  (label  $\langle label-name \rangle$ ))
(goto (reg  $\langle register-name \rangle$ ))
(save (reg  $\langle register-name \rangle$ ))
(restore (reg  $\langle register-name \rangle$ ))
```

5.1.2 Abstraction in Machine Design

The following is a GCD machine that reads inputs and prints results.

```
(define (gcd a b)
  (if (= b 0) a (gcd b (remainder a b))))

(controller
  (gcd-loop
    (assign a (op read))
    (assign b (op read))
  test-b
    (test (op =) (reg b) (const 0))
    (branch (label gcd-done))
    (assign t (op rem) (reg a) (reg b))
    (assign a (reg b))
    (assign b (reg a))
    (goto (label test-b))
  gcd-done
```

```
(perform (op print) (reg a))
(goto (label gcd-loop)))
```

5.1.3 Subroutines

5.2 A Register-Machine Simulator

Here are some procedures that we need to implement.

```
(make-machine <register-names> <operations> <controller>)
(set-register-contents! <machine-model> <register-name> <value>)
(get-register-contents <machine-model> <register-name>)
(start <machine-model>)
```

The following Scheme program implements a register-machine for computing the greatest common divisors of two given natural numbers.

```
(define gcd-machine
  (make-machine
    '(a b t)
    (list (list 'rem remainder) (list '= =))
    '(test-b
      (test (op =) (reg b) (const 0))
      (branch (label gcd-done))
      (assign t (op rem) (reg a) (reg b))
      (assign a (reg b))
      (assign b (reg t))
      (goto (label test-b))
      gcd-done)))
```

5.2.1 The Machine Model

```
(define (make-machine register-names ops controller-text)
  (let ((machine (make-new-machine)))
    (for-each (lambda (register-name)
                (machine 'allocate-register) register-name)
              register-names)
    ((machine 'install-operations) ops)
    ((machine 'install-instruction-sequence)
     (assemble controller-text machine))
    machine))
```

Registers

```
(define (make-register name)
  (let ((contents '*unassigned*))
    (define (dispatch msg)
      (cond ((eq? msg 'get) contents)
            ((eq? msg 'set)
             (lambda (value) (set! contents value)))
            (else (error "'unknown request -- REGISTER'" msg))))
    dispatch))
```

```
(define (get-contents register) (register 'get))

(define (set-contents register value) ((register 'set) value))
```

The Stack

```
(define (make-stack)
  (let ((s nil))
    (define (push x) (set! s (cons x s)))
    (define (pop)
      (if (null? pop)
          (error '“Empty stack -- POP”’)
          (let ((top (car s))
                (set! s (cdr s))
                top)))
    (define (initialize) (set! s nil) 'done)
    (define (dispatch msg)
      (cond ((eq? msg 'push) push)
            ((eq? msg 'pop) (pop))
            ((eq? msg 'initialize) (initialize))
            (else (error '“Unknown request -- STACK”’ msg))))
    dispatch))
```

The Basic Machine

```
(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        (stack (make-stack))
        (the-instruction-sequence nil))
    (let ((the-ops (list (list 'stack-initialization
                              (lambda () (stack 'initialize))))
                    (register-table (list (list 'pc pc) (list 'flag flag))))
      (define (allocate-register name) ...)
      (define (lookup-register-name name) ...)
      (define (execute)
        (let ((insts (get-contents pc)))
          (if (null? insts)
              'done
              (begin (instruction-execution-proc (car insts))
                     (execute)))))
      (define (dispatch msg) ...)
      dispatch)))
```

5.2.2 The Assembler

```
(define (assemble controller-text machine)
  (extract-labels controller-text
    (lambda (insts labels)
      (update-insts! insts labels machine)
```

```

    insts)))

(define (extract-labels text receive)
  (if (null? text)
      (receive '() '())
      (extract-labels (cdr text)
                      (lambda (insts labels)
                        (let ((next-inst (car text)))
                          (if (symbol? next-inst)
                              (receive insts
                                       (cons (make-label-entry next-inst insts)
                                             labels))
                              (receive (make-instruction next-inst insts)
                                       labels))))))))

```

5.2.3 Generating Execution Procedures for Instructions

```

(define (make-execution-procedure inst labels machine pc flag stack ops)
  (cond ((eq? (car inst) 'assign)
         (make-assign inst machine labels ops pc))
        ...
        ((eq? (car inst) 'goto)
         (make-goto inst machine labels pc))
        ...
        ((eq? (car inst) 'save)
         (make-save inst machine labels pc))
        ...))

(define (make-assign inst machine labels ops pc)
  (let ((target (get-register machine (assign-reg-name inst)))
        (value-exp (assign-value-exp inst)))
    (let ((value-proc
           (if (operation-exp? value-exp)
               (make-operation-exp value-exp machine labels ops)
               (make-primitive-exp
                (car value-exp) machine labels))))
      (lambda ()
        (set-contents! target (value-proc))
        (advance-pc pc)))))

(define (advance-pc pc)
  (set-contents! pc (cdr (get-contents pc))))

(define (make-goto inst machine labels pc)
  (let ((dest (goto-dest inst)))
    (cond ((label-exp? dest)
           (let ((insts (lookup-label labels
                                       (label-exp-label dest))))
             (lambda () (set-contents! pc insts))))
          ((register-exp? dest)
           (register-exp? dest)

```

```

        (let ((reg (get-register machine
                        (register-exp-reg dest))))
          (lambda ()
            (set-contents! pc (get-contents reg))))
        (else (error 'Bad GOTO instruction' -- ASSEMBLE inst))))

(define (make-save inst machine stack pc)
  (let ((reg (get-register machine (stack-inst-reg-name inst))))
    (lambda ()
      (push stack (get-contents reg))
      (advance-pc pc))))

```

5.2.4 Monitoring Machine Performance

```

(define (make-stack)
  (let ((s nil)
        (number-of-pushes 0)
        (max-depth 0)
        (current-depth 0))
    (define (push x)
      (set s (cons x s))
      (set! number-of-pushes (1+ number-of-pushes))
      (set! current-depth (1+ current-depth))
      (set! max-depth (max current-depth max-depth)))
    (define (pop) ...)
    (define (initialize) ...)
    (define (print-statistics) ...)

    (define (dispatch message) ...)
    dispatch))

```

5.3 Storage Allocation and Garbage Collection

5.3.1 Memory as Vectors

- (vector-ref *<vector>* *<n>*)
- (vector-set! *<vector>* *<n>* *<value>*)

Representing List Data

We fix two registers *the-cars* and *the-cdrs*. The following instructions

- (assign *<reg₁>* (op car) (reg *<reg₂>*))
- (assign *<reg₁>* (op cdr) (reg *<reg₂>*))

can now be implemented as follows, respectively.

- (assign *<reg₁>* (op vector-ref) (reg the-cars) (reg *<reg₂>*))
- (assign *<reg₁>* (op vector-ref) (reg the-cdrs) (reg *<reg₂>*))

Similarly, the following instructions

- (perform $\langle reg_1 \rangle$ (op set-car!) (reg $\langle reg_2 \rangle$))
- (perform $\langle reg_1 \rangle$ (op set-cdr!) (reg $\langle reg_2 \rangle$))

can now be implemented as follows, respectively.

- (perform (op vector-set!) (reg the-cars) (reg $\langle reg_1 \rangle$) (reg $\langle reg_2 \rangle$))
- (perform (op vector-set!) (reg the-cdrs) (reg $\langle reg_1 \rangle$) (reg $\langle reg_2 \rangle$))

For the following instruction,

- (assign $\langle reg_1 \rangle$ (op cons) (reg $\langle reg_2 \rangle$) ($\langle reg_1 \rangle$))

we implement it using a sequence instructions.

```
(perform (op vector-set! (reg the-cars) (reg free) (reg reg2)))
(perform (op vector-set! (reg the-cdrs) (reg free) (reg reg3)))
(assign (reg reg1) (reg free))
(assign (reg free) (op 1+) (reg free))
```

5.3.2 Maintaining the Illusion of Infinite Memory

For instance, the following computation involves allocating two lists: the enumeration and the result of filtering.

```
(fold-left + 0 (filter odd? (enumerate-interval 0 n)))
```

After the computation, the memory allocated for the intermediate results can be reclaimed. If we can arrange to collect all the garbage periodically, and if it turns out to recycle memory at about the same rate at which we allocate memory, we will have preserved the illusion that there is an infinite amount of memory.

Implementation of a stop-and-copy garbage collector

```
begin-gabaga-collection
  (assign free (cons 0))
  (assign scan (cons 0))
  (assign old (reg root))
  (assign relocate-continue (label reassign-root))
  (goto (label relocate-old-result-in-new))
reassign-root
  (assign root (reg new))
  (goto (label gc-loop))

gc-loop
  (test (op =) (reg scan) (reg free))
  (branch (label gc-flip))
  (assign old (op vector-ref) (reg new-cars) (reg scan))
  (assign relocate-continue (op vector-ref) (reg new-cars) (reg scan))
  (goto (label relocate-old-result-in-new))

update-car
```

```

(perform (op vector-set!) (reg new-cars) (reg scan) (reg new))
(assign old (op vector-ref) (reg new-cdrs) (reg scan))
(assign relocate-continue (label update-cdr))
(goto (label relocate-old-result-in-new))

```

update-cdr

```

(perform (op vector-set!) (reg new-cdrs) (reg scan) (reg new))
(assign scan (op 1+) (reg scan))
(goto (label gc-loop))

```

relocate-old-result-in-new

```

(test (op pointer-to-pair?) (reg old))
(branch (label pair))
(assign new (reg old))
(goto (reg relocate-continue))

```

pair

```

(assign oldcr (op vector-ref) (reg the cars) (reg old))
(test (op broken-heart?) (reg oldcr))
(branch already-moved)
(assign new (reg free)) ; new location for pair
(assign free (op 1+) (reg free)) ; update the free pointer

```

; Copy the car and cdr to new memory

```

(perform (op vector-set!)
         (reg new-cars) (reg new) (reg oldcr))
(assign oldcr (op vector-ref) (reg the-cdrs) (reg old))
(perform (op vector-set!)
         (reg new-cdrs) (reg new) (reg oldcr))

```

; Construct the broken heart

```

(perform (op vector-set!)
         (reg the-cars) (reg old) (const broken-heart))
(perform (op vector-set!) (reg the-cdrs) (reg old) (reg new))
(goto (reg relocate-continue))

```

already-moved

```

(assign new (op vector-ref) (reg the-cdrs) (reg old))
(goto (reg relocate-continue))

```

gc-flip

```

(assign temp (reg the-cdrs))
(assign the-cdrs (reg new-cdrs))
(assign new-cdrs temp)
(assign temp (reg the-cars))
(assign the-cars (reg new-cars))
(assign new-cars temp)

```

5.4 The Explicit-Control Evaluator

```

(define (fact n)
  (if (= n 0) 1 (* n (fact (- n 1)))))

;;; arg for storing the argument
;;; val for storing the result
;;; continue for storing the countination
fact
  (assign r1 (reg arg))
  (test (op =) (reg r1) (const 0))
  (branch (label finish))
if-else
  (assign arg (op -) (reg r1) (const 1))
  (save continue)
  (save r1)
  (assign continue (label rest))
  (goto fact)
rest
  restore r1
  restore continue
  (assign val (op *) val r1)
  goto continue
if-then
  (assign val (const 1))
  (goto continue)

eval-dispatch
  (test (op self-evaluating?) (reg exp))
  (branch (label ev-self-eval))
  (test (op variable?) (reg exp))
  (branch (label ev-variable))
  (test (op quoted?) (reg exp))
  (branch (label ev-quoted))
  (test (op assignment?) (reg exp))
  (branch (label ev-assignment))
  (test (op lambda?) (reg exp))
  (branch (label ev-lambda))
  ...

;;; registers: exp, env, val, continue, proc, argl and unev
ev-self-eval
  (assign val (reg exp))
  (goto continue)
ev-variable
  (assign val (op lookup-variable-value) (reg exp) (reg env))
  (goto continue)
ev-quoted
  (assign val (op text-of-quotation) (reg exp))
  (goto continue)

```

```
ev-lambda
  (assign unev (op lambda-parameters) (reg exp))
  (assign exp (op lambda-body) (reg exp))
  (assign val (op make-procedure)
              (reg unev) (reg exp) (reg env))
  (goto continue)

ev-application
  (save continue)
  (save env)
  (assign unev (op operands) (reg exp))
  (save unev)
  (assign exp (op operator) (reg exp))
  (assign continue (ev-appl-did-operator))
  (goto (label eval-dispatch))

ev-appl-did-operator
  (restore unev)
  (restore env)
  (assign argl (op empty-arglist))
  (assign proc (reg val))
  (test (op no-operands?) (reg unev))
  (branch (apply-dispatch))
  (save proc)

ev-appl-operand-loop
  (save argl)
  (assign exp (op first-operand) (reg unev))
  (test (op last-operand?) (reg unev))
  (branch (ev-appl-last-arg))
  (save env)
  (save unev)
  (assign continue (label ev-app-accumulate-arg))
  (goto eval-dispatch)

ev-appl-accumulate-arg
  (restore unev)
  (restore env)
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (assign unev (op rest-operands) (reg unenv))
  (goto (label eval-appl-operand-loop))

ev-appl-last-arg
  (assign continue (label ev-appl-accum-last-arg))
  (goto (label eval-dispatch))

ev-appl-accum-last-arg
  (restore argl)
```

```
(assign arg1 (op adjoin-arg) (reg val) (reg arg1))
(restore proc)
(goto (label apply-dispatch))
```

```
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (goto (label unknow-procedure-type))
```

```
primitive-apply
  (assign val (op apply-primitive-procedure) (reg proc) (reg arg1))
  (restore coontinue)
  (goto (reg continue))
```

```
compound-apply
  (assign unev (op procedure-parameters) (reg proc))
  (assign env (op procedure-environment) (reg proc))
  (assign env (op extend-environment) (reg unev) (reg arg1) (reg env))
  (assign unev (op procedure-body) (reg proc))
  (goto (label ev-sequence))
```

```
ev-begin
  (assign unev (op begin-actions) (reg exp))
  (save continue)
  (goto (label ev-sequence))
```

```
ev-sequence
  (assign exp (op first-exp) (reg unev))
  (test (op last-exp?) (reg unev))
  (branch (label ev-sequence-last-exp))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
```

```
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
```

```
ev-sequence-last-exp
  (restore continue)
  (goto (label eval-dispatch))
```

5.5 Compilation