These notes are based on the second edition of *Structure and Interpretation of Computer Programs* by Hal Abelson and Gerald Jay Sussman with Julie Sussman.

## Dedication

> This book is dedicated, in respect and admiration, to the spirit that lives in the computer. [@dedication]

> I think that it's extraordinarily important that we in computer science keep fun in computing. [@dedication]

> Don't feel as if the key to successful computing is only in your hands. What's in your hands, I think and hope, is intelligence: the ability to see the machine as more than when you were first led up to it, that you can make it more. [@dedication]

## Foreword

> To appreciate programming as an intellectual activity in its own right you must turn to computer programming; you must read and write computer programs – many of them. [@foreword]

> Every computer program is a model, hatched in the mind, of a real or mental process. These processes, arising from human experience and thought, are huge in number, intricate in detail, and at any time only partially understood. They are modeled to our permanent satisfaction rarely by our computer programs. Thus even though our programs are carefully handcrafted discrete collections of symbols, mosaics of interlocking functions, they continually evolve: we change them as our perception of the model deepens, enlarges, generalizes until the model ultimately attains a metastable place within still another model with which we struggle. The source of the exhilaration associated with computer programming is the continual unfolding within the mind and on the computer of mechanisms expressed as programs and the explosion of perception they generate. [@foreword]

> Since large programs grow from small ones, it is crucial that we develop an arsenal of standard program structures of whose correctness we have become sure – we call them idioms – and learn to combine them into larger structures using organizational techniques of proven value. [@foreword]

> The computers are never large enough or fast enough. Each breakthrough in hardware technology leads to more massive programming enterprises, new organizational principles, and an enrichment of abstract models. Every reader should ask himself periodically "Toward what end, toward what end?" – but do not ask it too often lest you pass up the fun of programming for the constipation of bittersweet philosophy. [@foreword]

> Lisp is for building organisms – imposing, breathtaking, dynamic structures built by squads fitting fluctuating myriads of simpler organisms into place. [@foreword]

> It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures. [@foreword]

## Preface

> A computer language is not just a way of getting a computer to perform operations but rather … it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. [@preface]

> The essential material … is not the syntax of particular programming-language constructs, nor clever algorithms for computing particular functions efficiently, nor even the mathematical analysis of algorithms and the foundations of computing, but rather the techniques used to control the intellectual complexity of large software systems. [@preface]

> Underlying our approach to this subject is our conviction that "computer science" is not a science and that its significance has little to do with computers. [@preface]

> Mathematics provides a framework for dealing precisely with notions of "what is". Computation provides a framework for dealing precisely with notions of "how to". [@preface]

# 1: Building Abstractions with Procedures

- A computational process evolves to manipulate data.
- The evolution is controlled by a program, a pattern of rules.
- Well-designed computational systems are modular.
- This book uses the Scheme dialect of Lisp.
- Lisp represents procedures as data.

## 1.1: The Elements of Programming

There are three mechanisms for combining simple ideas to form more complex ideas found in every powerful programming language:

- primitive expressions, which represent the simplest entities the language is concerned with,
- 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

Programming deals with procedures and data (which are almost the same thing in Lisp). Procedures manipulate data.

## 1.1.1: Expressions

- The REPL reads an expression, evaluates it, prints the result, and repeats.
- A number is one kind of primitive expression.
- An application of a primitive procedure is one kind of compound expression.
- A *combination* denotes procedure application by a list of expressions inside parentheses. The first element is the *operator*; the rest are the *operands*.
- Lisp combinations use prefix notation (the operator comes first).
- Combinations can be nested: an operator or operand can itself be another combination.

## 1.1.2: Naming and the Environment

- Scheme names things with the `define`. This is the simplest means of abstraction.
- The name–value pairs are stored in an *environment*.

## 1.1.3: Evaluating Combinations

- To evaluate a combination:
  1. Evaluate the subexpressions of the combination.
  2. Apply the procedure (value of leftmost subexpression, the operator) to the arguments (values of other subexpressions, the operands).
- Before evaluating a combination, we must first evaluate each element inside it.
- Evaluation is recursive in nature – one of its steps is invoking itself.
- The evaluation of a combination can be represented with a tree.
- Recursion is a powerful technique for dealing with hierarchical, tree-like objects.
- To end the recursion, we stipulate the following:
  1. Numbers evaluate to themselves.
  2. Built-in operators evaluate to machine instruction sequences.
  3. Names evaluate to the values associated with them in the environment.
- `(define x 3)` does not apply `define` to two arguments; this is not a combination.
- Exceptions such as these are *special forms*. Each one has its own evaluation rule.

> In the words of Alan Perlis, "Syntactic sugar causes cancer of the semicolon". [@1.1.fn11]

## 1.1.4: Compound Procedures

- *Procedure definition* is a powerful technique for abstraction.
- A squaring procedure: `(define (square x) (* x x))`.
- This is a compound procedure given the name "square".
- General form of a procedure definition: `(define («name» «formal-parameters») «body»)`.
- If the body contains more than one expression, each is evaluated in sequence and the value of the last one is returned.

## 1.1.5: The Substitution Model for Procedure Application

This is the substitution model:

> To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument. [@1.1.5]

An example of procedure application:

```
(f 5)
(sum-of-squares (+ 5 1) (* 5 2))
(sum-of-squares 6 10)
(+ (square 6) (square 10))
(+ 36 100)
136
```

**Applicative order versus normal order**

- The example above used *applicative order*: evaluate all the subexpressions first, then apply the procedure to the arguments.
- With *normal order*, operands are substituted in the procedure unevaluated. Only when it reaches primitive operators do combinations reduce to values.
- An example of normal-order procedure application:

```
(f 5)
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square (+ 5 1)) (square (* 5 2)))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

- Here, normal order causes some combinations to be evaluated multiple times.

## 1.1.6: Conditional Expressions and Predicates

- To make more useful procedures, we need to be able to conduct tests and perform different operations accordingly.
- We do *case analysis* in Scheme using `cond`.
- `cond` expressions work by testing each predicate. The consequent expression of the first clause with a true predicate is returned, and the other clauses are ignored.
- A predicate is an expression that evaluates to true or false.
- The symbol `else` can be used as the last clause – it will always evaluate to true.
- The `if` conditional can be used when there are only two cases.
- Logical values can be combined with `and`, `or`, and `not`. The first two are special forms, not procedures, because they have short-circuiting behavior.

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

> But there is an important difference between mathematical functions and computer procedures. Procedures must be effective. [@1.1.7]

- In mathematics, you can define square roots by saying, "The square root of $x$ is the nonnegative $y$ such that $y^2 = x$". This is not a procedure.
- Mathematical functions describe things (declarative knowledge); procedures describe how to do things (imperative knowledge).
- Declarative is *what is*, imperative is *how to*.

1.6-8

## 1.1.8: Procedures as Black-Box Abstractions

- Each procedure in a program should accomplish an identifiable task that can be used as a module in defining other procedures.
- When we use a procedure as a "black box", we are concerned with *what* it is doing but not *how* it is doing it.
- This is called procedural abstraction. Its purpose is to suppress detail.

> A user should not need to know how the procedure is implemented in order to use it. [@1.1.8]

### Local names

- The choice of names for the procedure's formal parameters should not matter to the user of the procedure.
- Consequentially, the parameter names must be local to the body of the procedure.
- The name of a formal parameter doesn't matter; it is a *bound variable*. The procedure *binds* its formal parameters.
- If a variable is not bound, it is *free*.
- The expression in which a binding exists is called the *scope* of the name. For parameters of a procedure, this is the body.
- Using the same name for a bound variable and an existing free variable is called *capturing* the variable.
- The names of the free variables *do* matter for the meaning of the procedure.

### Internal definitions and block structure

- Putting a definition in the body of a procedure makes it local to that procedure. This nesting is called *block structure*.
- Now we have two kinds of name isolation: formal parameters and internal definitions.
- By internalizing auxiliary procedures, we can often eliminate bindings by allowing variables to remain free.
- Scheme uses *lexical scoping*, meaning free variables in a procedure refer to bindings in enclosing procedure definitions.

## 1.2: Procedures and the Processes They Generate

> To become experts, we must learn to visualize the processes generated by various types of procedures. Only after we have developed such a skill can we learn to reliably construct programs that exhibit the desired behavior. [@1.2]

- A procedure is a pattern for the *local evolution* of a computation process: how one stage is built on the previous stage.
- The global behavior of a computational process is much harder to reason about.
- Processes governed by different types of procedures generate different "shapes".
- Computational processes consume two important resources: time and space.

### 1.2.1: Linear Recursion and Iteration

- The factorial of $N$ is defined as the product of the integers on the interval $[1, N]$.
- The naive *recursive* implementation creates a curved shape:

```
(factorial 4)
(* 4 (factorial 3))
(* 4 (* 3 (factorial 2)))
(* 4 (* 3 (* 2 (factorial 1))))
(* 4 (* 3 (* 2 1)))
(* 4 (* 3 2))
(* 4 6)
24
```

- The *iterative* implementation maintains a running product and multiplies the numbers from 1 to $N$. This creates a shape with a straight edge:

```
(factorial 4)
(fact-iter 1 1 4)
(fact-iter 1 2 4)
(fact-iter 2 3 4)
(fact-iter 6 4 4)
(fact-iter 24 5 4)
24
```

- Both compute the same mathematical function, but the computational processes evolve very differently.
- The first one is a *linear recursive process*. The chain of deferred operations causes an expansion (as operations are added) and a contraction (as operations are performed).
    - The interpreter must keep track of all these operations.
    - It is a *linear* recursive process because the information it must keep track of (the call stack) grows linearly with $N$.
- The second is a *linear iterative process*. It does not grow and shrink.
    - It is summarized by a fixed number of state variables and a rule to describe how they should update and when the process should terminate.
    - It is a *linear* iterative process because the number of steps grows linearly with $N$.

- In the iterative process, the variables provide a complete description of the state of the process at any point. In the recursive process, there is “hidden” information that makes it impossible to resume the process midway through.
- The longer the chain of deferred operations, the more information must be maintained (in a stack, as we will see).
- A recursive *procedure* is simply a procedure that refers to itself directly or indirectly.
- A recursive *process* refers to the evolution of the process described above.
- A recursive procedure can generate an iterative process in Scheme thanks to *tail-call optimization*. In other languages, special-purpose looping constructs are needed.

1.9-10

## 1.2.2: Tree Recursion

- With tree recursion, the procedure invokes itself more than once, causing the process to evolve in the shape of a tree.
- The naive Fibonacci procedure calls itself twice each time it is invoked, so each branch splits into two at each level.

> In general, the number of steps required by a tree-recursive process will be proportional to the number of nodes in the tree, while the space required will be proportional to the maximum depth of the tree. [@1.2.2]

- The iterative Fibonacci procedure is vastly more efficient in space and in time.

#### Example: Counting change

Let $f(A, N)$ represent the number of ways of changing the amount $A$ using $N$ kinds of coins. If the first kind of coin has denomination $N$, then $f(A, N) = f(A, N - 1) + f(A - D, N)$. In words, there are two situations: where you do not use any of the first kind of coin, and when you do. The value of $f(A, N - 1)$ assumes we don’t use the first kind at all; the value of $f(A - D, N)$ assumes we use one or more of the first kind.

That rule and a few degenerate cases are sufficient to describe an algorithm for counting the number of ways of changing amounts of money. We can define it with the following piecewise function:

$$ f(A,N) = \begin{cases} 1, & \text{if $A = 0$,} \\ 0, & \text{if $A < 0$ or $N = 0$,} \\ f(A,N-1) + f(A-D,N), & \text{if $A > 0$ and $N > 0$.} \end{cases} $$

Like Fibonacci, the easy tree-recursive implementation involves a lot of redundancy. Unlike it, there is no obvious iterative solution (it is possible, just harder). One way to improve the performance of the tree-recursive process is to use *memoization*.

1.11-13

## 1.2.3: Orders of Growth

- Different processes consume different amounts of computational resources.
- We compare this using *order of growth*, a gross measure of the resources required by a process as the inputs becomes larger.

- Let $n$ be a parameter that measures the size of a problem – it could be the input itself, the tolerance, the number of rows in the matrix, etc.
- Let $R(n)$ be the amount of resources the process requires for a problem of size $n$. This could be time, space (amount of memory), number of registers used, etc.
- We say that $R(n)$ has order of growth $\Theta(f(n))$, or $R(n) = \Theta(f(n))$, if there are positive constants $A$ and $B$ independent of $n$ such that $Af(n) \le R(n) \le Bf(n)$ for any sufficiently large value of $n$.
- The value $R(n)$ is sandwiched between $Af(n)$ and $Bf(n)$.
- The linear recursive process for computing factorials had $\Theta(n)$ time and $\Theta(n)$ space (both linear), whereas the linear iterative process had $\Theta(1)$ space (constant).
- The order of growth is a crude description of the behavior of a process.
- Its importance is allowing us to see the *change* in the amount of resources required when you, say, increment $n$ or double $n$.

1.14-15

## 1.2.4: Exponentiation

One way to calculate $b$ to the $n$th power is via the following recursive definition:

$$b^0 = 1, \qquad b^n = b * b^{n-1}.$$

A faster method is to use successive squaring:

$$ b^n = \begin{cases} \left(b^{n/2}\right)^2, & \text{if $n$ is even,} \\ b * b^{n-1}, & \text{if $n$ is odd.} \end{cases} $$

1.16-19

## 1.2.5: Greatest Common Divisors

- The GCD of integers $a$ and $b$ is the largest integer that divides both $a$ and $b$ with no remainder. For example, $\gcd(16, 28) = 4$.
- An efficient algorithm uses $\gcd(a, b) = \gcd(b, a \bmod b)$.
- For example, we can reduce `(gcd 206 40)` as follows:

```
(gcd 206 40)
(gcd 40 6)
(gcd 6 4)
(gcd 4 2)
(gcd 2 0)
2
```

- This always works: you always get a pair where the second number is zero, and the other number is the GCD of the original pair.
- This is called *Euclid’s Algorithm*.
- Lamé’s Theorem: If Euclid’s Algorithm requires $k$ steps to compute the GCD of some pair $(a, b)$, then $\min\{a,b\} ≥ \Fib(k)$.

1.20

### 1.2.6: Example: Testing for Primality

#### Searching for divisors

- One way to test for primality is to find the number's divisors.
- A number is prime if and only if it is its own smallest divisor.

#### The Fermat test

The Fermat test is a $\Theta(log(n))$ primality test based on Fermat's Little Theorem:

> If $n$ is a prime number and $a$ is any positive integer less than $n$, then $a$ raised to the $n$th power is congruent to $a$ modulo $n$. [@1.2.6]

The test works like this:

1. Given a number $n$, pick a random number $a < n$ and calculate $a^n \bmod n$.
2. Fail: If the result is not equal to $a$, then $n$ is not prime.
3. Pass: If the result is equal to $a$, then $n$ is likely prime.
4. Repeat. The more times the number passes the test, the more confident we are that $n$ is prime. If there is a single failure, $n$ is certainly not prime.

#### Probabilistic methods

- Most familiar algorithms compute an answer that is guaranteed to be correct.
- Not so with the Fermat test. If $n$ passes the Fermat test for one random value of $a$, all we know is that there is a better than 50% chance of $n$ being prime.
- A *probabilistic algorithm* does not always give a correct result, but you can prove that the chance of error becomes arbitrarily small.
- We can make the probability error in our primality test as small as we like simply by running more Fermat tests – except for Carmichael numbers.

> Numbers that fool the Fermat test are called Carmichael numbers, and little is known about them other than that they are extremely rare. … In testing primality of very large numbers chosen at random, the chance of stumbling upon a value that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a "correct" algorithm. Considering an algorithm to be inadequate for the first reason but not for the second illustrates the difference between mathematics and engineering. [@1.2.fn47]

1.21-28

## 1.3: Formulating Abstractions with Higher-Order Procedures

> We have seen that procedures are, in effect, abstractions that describe compound operations on numbers independent of the particular numbers. [@1.3]

> One of the things we should demand from a powerful programming language is the ability to build abstractions by assigning names to common patterns and then to work in terms of the abstractions directly. [@1.3]

- Procedures operating on numbers are powerful, but we can go further.
- To abstract more general programming patterns, we need to write procedures that take other procedures as arguments and return new procedures.
- These are called *higher-order* procedures.

## 1.3.1: Procedures as Arguments

Procedures that compute a sum all look the same:

```
(define («name» a b)
  (if (> a b)
      0
      (+ («term» a)
         («name» («next» a) b))))
```

This is a useful abstraction, just as sigma notation is useful in math because the summation of a series is so common.

> The power of sigma notation is that it allows mathematicians to deal with the concept of summation itself rather than only with particular sums. [@1.3.1]

1.29-33

## 1.3.2: Constructing Procedures Using `Lambda`

`lambda` creates anonymous procedures. They are just like the procedures created by `define`, but without a name: `(lambda («formal-parameters») «body»)`.

A lambda expression can be used as the operand in a combination. It will be evaluated to a procedure and applied to the arguments (the evaluated operands). The name comes from the λ-calculus, a formal system invented by Alonzo Church.

### Using `let` to create local variables

We often need local variables in addition to the formal parameters. We can do this with a lambda expression that takes the local variables as arguments, but this is so common that there is a special form `let` to make it easier.

The general form of a let-expression is:

```
(let ((«var1» «exp1»)
      («var2» «exp2»)
      ...
      («varn» «expn»))
  «body»)
```

This is just syntactic sugar for:

```
((lambda («var1» «var2» ... «varn»)
   «body»)
 «exp1»
 «exp2»
 ...
 «expn»)
```

- The scope of a variable in a let-expression is the body. This allows variables to be bound as locally as possible.
- The variables in the let-expression are parallel and independent. They cannot refer to each other, and their order does not matter.
- You can use let-expressions instead of internal definitions.

1.34

## 1.3.3: Procedures as General Methods

So far, we have seen:

- Compound procedures that abstract patterns of numerical operators (mathematical functions), independent of the particular numbers.
- Higher-order procedures that express a more powerful kind of abstraction, independent of the procedures involved.

Now we will take it a bit further.

### Finding roots of equations by the half-interval method

- The *half-interval method*: a simple but powerful technique for solving $f(x) = 0$.
- Given $f(a) < 0 < f(b)$, there must be at least one zero between $a$ and $b$.
- To narrow it down, we let $x$ be the average of $a$ and $b$, and then replace either the left bound or the right bound with it.

```
(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? x y)
  (< (abs (- x y)) 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 (negative? b-value) (positive? a-value))
           (search f b a))
          (else
           (error "values are not of opposite sign" a b)))))
```

We can use `half-interval-method` to approximate $\pi$ as a root of $\sin x = 0$:

```
(half-interval-method sin 2.0 4.0)
→ 3.14111328125
```

**Finding fixed points of functions**

- A number $x$ is a *fixed point* of a function if $f(x) = x$.
- In some cases, repeatedly applying the function to an arbitrary initial guess will converge on the fixed point.
- The procedure we made earlier for finding square roots is actually a special case of the fixed point procedure.

```
(define tolerance 0.00001)
(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))
```

We can use `fixed-point` to approximate the fixed point of the cosine function:

```
(fixed-point cos 1.0)
→ 0.7390822985224023
```

1.35-39

## 1.3.4: Procedures as Returned Values

Passing procedures as arguments gives us expressive power. Returning procedures from functions gives us even more. For example, we can write a procedure that creates a new procedure with average damping:

```
(define (average-damp f)
  (lambda (x) (average x (f x))))
```

If we use `average-damp` on `square`, we get a procedure that calculates the sum of the numbers from 1 to $n$:

```
((average-damp square) 10)
→ 55

(+ 1 2 3 4 5 6 7 8 9 10)
→ 55
```

> In general, there are many ways to formulate a process as a procedure. Experienced programmers know how to choose procedural formulations that are particularly perspicuous, and where useful elements of the process are exposed as separate entities that can be reused in other applications. [@1.3.4]

**Newton's method**

The square-root procedure we wrote earlier was a special case of *Newton's method*. Given a function *f*(*x*), the solution to *f*(*x*) = 0 is given by the fixed point of

$$x \mapsto x - \frac{f(x)}{f'(x)}.$$

Newton's method converges very quickly – much faster than the half-interval method in favorable cases. We need a procedure to transform a function into its derivative (a new procedure). We can use a small *dx* for this:

$$f'(x) = \frac{f(x+dx) - f(x)}{dx}.$$

This translates to the following procedure:

```
(define (deriv f)
  (lambda (x) (/ (- (f (+ x dx)) (f x)) dx)))
```

Now we can do things like this:

```
(define (cube x) (* x x x))
(define dx 0.00001)
((deriv cube) 5)
→ 75.00014999664018
```

**Abstractions and first-class procedures**

- Compound procedures let us express general methods of computing as explicit elements in our programming language.
- Higher-order procedures let us manipulate methods to create further abstractions.
- We should always be on the lookout for underlying abstractions that can be brought out and generalized. But this doesn't mean we should always program in the most abstract form possible; there is a level appropriate to each task.
- Elements with the fewest restrictions are *first-class*:
    - They may be named by variables.
    - They may be passed as arguments to procedures.
    - They may be returned as the results of procedures.
    - They may be included in data structures.
- In Lisp, procedures have first-class status. This gives us an enormous gain in expressive power.

1.40-46

# 2: Building Abstractions with Data

- In we wrote procedures that operated on simple numerical data.
- In this chapter we are going to look at more complex data.

> Just as the ability to define procedures enables us to deal with processes at a higher conceptual level than that of the primitive operations of the language, the ability to construct compound data objects enables us to deal with data at a higher conceptual level than that of the primitive data objects of the language. [@2]

- Consider functions dealing with rational numbers: only being able to return one number is awkward. This is where compound data objects become useful.
- *Data abstraction* separates representation from use. It enables us to construct *abstraction barriers* between different parts of the program.

## 2.1: Introduction to Data Abstraction

- We already know about procedural abstraction, which suppresses implementation details by treating procedures as black boxes.
- Data abstractions allows us to isolate how a compound data object is used from the details of its actual representation.

> That is, our programs should use data in such a way as to make no assumptions about the data that are not strictly necessary for performing the task at hand. [@2.1]

- There is a concrete data representation behind the abstraction.
- The interface is made up of procedures called *constructors* and *selectors*.

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

- We want to add, subtract, multiply, divide, and test equality with our rational numbers.
- Assume we have `(make-rat «n» «d»)`, `(number «x»)`, and `(denom «x»)` available as the constructor and selectors. This is wishful thinking, and it is a good technique.
- Here is the implementation for addition:

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

#### Pairs

- A *pair* is a concrete structure that we create with `cons`.
- We extract the parts of the pair with `car` and `cdr`.

```
(define x (cons 1 2))
```

```
(car x)
→ 1

(cdr x)
→ 2
```

- This is all the glue we need to implement all sorts of complex data structures.
- Data objects constructed from pairs are called *list-structured* data.

**Representing rational numbers**

- Now we can represent rational numbers:

```
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))
```

- To ensure that our rational numbers are always in lowest terms, we need `make-rat` to divide the numerator and the denominator by their greatest common divisor (GCD).

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))
```

2.1

## 2.1.2: Abstraction Barriers

> In general, the underlying idea of data abstraction is to identify for each type of data object a basic set of operations in terms of which all manipulations of data objects of that type will be expressed, and then to use only those operations in manipulating the data. [@2.1.2]

- Details on the other side of an abstraction barrier are irrelevant to the code on this side.
- This makes programs easier to maintain and modify.

> Constraining the dependence on the representation to a few interface procedures helps us design programs as well as modify them, because it allows us to maintain the flexibility to consider alternate implementations. [@2.1.2]

2.2-3

## 2.1.3: What Is Meant by Data?

- Data is defined by some collection of selectors and constructors, together with specified conditions that these procedures must fulfill in order to be a valid representation.
- For rationals, we have the following definition:
    1. We can construct a rational `x` with `(make-rat n d)`.
    2. We can select the numerator with `(numer x)`.

    3. We can select the denominator with `(denom x)`.
    4. For all values of `x`, `(/ (numer x) (denom x))` must equal *n*/*d*.
- For pairs, it is even simpler: we need three operations, which we will call `cons`, `car`, and `cdr`, such that if `z` is `(cons x y)`, then `(car z)` is `x` and `(cdr z)` is `y`.
- Any triple of procedures satisfying this definition can be used to implement pairs. In fact, we can do it with procedures themselves:

```
(define (cons x y)
  (lambda (m)
    (if (= m 0) x y)))
(define (car z) (z 0))
(define (cdr z) (z 1))
```

- This doesn't look like *data*, but it works.
- This is how you implement pairs in the λ-calculus.
- In real Lisp implementations, pairs are implemented directly, for reasons of efficiency. But they *could* be implemented this way and you wouldn't be able to tell the difference.

> The ability to manipulate procedures as objects automatically provides the ability to represent compound data. [@2.1.3]

- This style of programming is often called *message passing*.

2.4-6

### 2.1.4: Extended Exercise: Interval Arithmetic

- We want to design a system that allows us to manipulate inexact quantities with known precision (uncertainty).
- We need arithmetic operations for combining intervals (ranges of possible values).

2.7-16

## 2.2: Hierarchical Data and the Closure Property

- Pairs form a primitive "glue" for compound data objects.
- We can visualize pairs with *box-and-pointer* notation. Each pair is a double box, and both sides have an arrow pointing to a primitive data object, or to another pair.
- The *closure property* of `cons` is the ability to make pairs whose elements are pairs. This allows us to create hierarchal structures.
- We have been using closure all along with combinations. Now, we are going to use closure for compound data.

> The use of the word "closure" here comes from abstract algebra, where a set of elements is said to be closed under an operation if applying the operation to elements in the set produces an element that is again an element of the set. The Lisp community also (unfortunately) uses the word "closure" to describe

> a totally unrelated concept: A closure is an implementation technique for representing procedures with free variables. We do not use the word "closure" in this second sense in this book. [@2.2.fn6]

### 2.2.1: Representing Sequences

- One thing we can build with pairs is a *sequence*: an ordered collection of data objects.
- Sequences can be represented by chains of pairs where each `car` points to a value and each `cdr` points to the next pair. The final pair's `cdr` is a special value, `nil`.
- For example, `(cons 1 (cons 2 (cons 3 (cons 4 nil))))` represents a sequence:

```
     .---+---.    .---+---.    .---+---.    .---+--+.
---->| * | *-+--->| * | *-+--->| * | *-+--->| * | / +
     '-|-+---'    '-|-+---'    '-|-+---'    '-|-++--'
       |            |            |            |
       v            v            v            v
     .---.        .---.        .---.        .---.
     | 1 |        | 2 |        | 3 |        | 4 |
     '---'        '---'        '---'        '---'
```

- This way of nesting pairs is called a *list*. We usually represent lists by placing each element one after the other and enclosing the whole thing in parentheses.
- The procedure `car` gives us the first item; `cdr` gives us the sublist containing all items but the first; `cons` returns a list with an item added to the front.
- The `nil` value can be thought of as an empty list.

> In this book, we use *list* to mean a chain of pairs terminated by the end-of-list marker. In contrast, the term *list structure* refers to any data structure made out of pairs, not just to lists. [@2.2.fn8]

#### List operations

- We can get the nth item of a list by `cdr`ing $n - 1$ times, and then taking the `car`.

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
```

- Scheme includes a primitive predicate `null?` which is true if its argument is `nil`.
- We often write recursive procedures that `cdr` all the way through the list.

```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)))))
```

- We can build up lists to return by `cons`ing them up:

```
(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1 (append (cdr list1) list2)))))
```

2.17-20

**Mapping over lists**

- The higher-order procedure `map` applies the same transformation to each element in a list, producing a new list.

```
(define (map f xs)
  (if (null? xs)
      nil
      (cons (f (car xs))
            (map f (cdr xs)))))
```

- For example, `(map abs (list -1 5 -3 0 2 -2))` gives the list `(1 5 3 0 2 2)`.
- `map` establishes a higher level of abstraction for dealing with lists; it suppressive the recursive detail. We think about the process differently when we use `map`.

> This abstraction gives us the flexibility to change the low-level details of how sequences are implemented, while preserving the conceptual framework of operations that transform sequences to sequences. [@2.2.1]

2.21-23

## 2.2.2: Hierarchical Structures

- We can represent lists whose elements themselves are also lists.
- We can also think of these structures as *trees*.
- Recursion is a natural tool for dealing with trees.
- The primitive predicate `pair?` returns true if its argument is a pair.
- We can count the number of leaves in a tree like so:

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                 (count-leaves (cdr x))))))
```

2.24-29

**Mapping over trees**

- We can deal with trees using `map` together with recursion.
- This allows us to apply an operation to all the leaves in a tree, for example.

2.30-32

## 2.2.3: Sequences as Conventional Interfaces

- Abstractions preserves the flexibility to experiment with alternative representations.
- The use of *conventional interfaces* is another powerful design principle.
- To make abstract operations for things other than numbers, we need to have a conventional style in which we manipulate data.

### Sequence operations

- We want to organize programs to reflect signal-flow structure. To do this, we focus on the signals and represent them as lists, and implement sequence operations on them.
- Expressing programs as sequence operations helps us make program designs that are *modular* – made of relatively independent pieces that we can connect in flexible ways. This is a strategy for controlling complexity.
- A surprisingly vast range of operations can be expressed as sequence operations.
- Sequences serve as a conventional interface for the modules of the program.

> One of the reasons for the success of Lisp as a programming language is that lists provide a standard medium for expressing ordered collections so that they can be manipulated using higher-order operations. The programming language APL owes much of its power and appeal to a similar choice. In APL all data are represented as arrays, and there is a universal and convenient set of generic operators for all sorts of array operations. [@2.2.fn15]

2.33-39

### Nested mappings

- For many computations, the sequence paradigm can be used instead of loops.
- Sometimes we need *nested* mappings, where each mapping maps to a second set of mappings. We can use `mapcat` to flatten the result into one list at the end.
- The procedure for computing all permutations of a set is pure magic: this is wishful thinking in action!

```
(define (permutations s)
  (if (null? s)
      (list nil)
      (mapcat (lambda (x)
                (map (lambda (p) (cons x p))
                     (permutations (remove x s))))
              s)))
```

2.40-43

## 2.2.4: Example: A Picture Language

- In this section, we will create a simple language for drawing pictures.
- The data objects are represented as procedures, not as list structure.

**The picture language**

- There is only one kind of element, called a *painter*.
- The painter draws an image transformed into a parallelogram.
- We combine images with operations like `beside` and `below`.
- We transform single images with operations like `flip-vert` and `flip-horiz`.
- We can build up complexity easily thanks to closure: the painters are closed under the language's means of combination. For example:

```
(define (flipped-pairs painter)
  (let ((painter2 (beside painter (flip-vert painter))))
    (below painter2 painter2)))
```

2.44

**Higher-order operations**

- Just as we have higher-order procedures, we can have higher-order painter operations.
- We can manipulate the painter operations rather than manipulating the painters directly.
- Here is an example higher-order painter operation:

```
(define (square-of-four tl tr bl br)
  (lambda (painter)
    (let ((top (beside (tl painter) (tr painter)))
          (bottom (beside (bl painter) (br painter))))
      (below bottom top))))
```

2.45

**Frames**

- Painters paint their contents in *frames*.
- A frame parallelogram can be represented by an origin vector and two edge vectors.
- We will use coordinates in the unit square to specify images.
- We can use basic vector operations to map an image coordinate into a pair of coordinates within the frame.

2.46-47

**Painters**

- A painter is a procedure that takes a frame as an argument and draws its image transformed to fit in the frame.
- The details of primitive painters depend on the characteristics of the graphics system.
- Representing painters as procedures creates a powerful abstraction barrier.

2.48-49

**Transforming and combining painters**

- Operations on painters invoke the original painters with new frames derived from the argument frame.
- They are all based on the procedure `transform-painter`.

```
(define (transform-painter painter origin corner1 corner2)
  (lambda (frame)
    (let ((m (frame-coord-map frame)))
      (let ((new-origin (m origin)))
        (painter
         (make-frame new-origin
                     (sub-vect (m corner1) new-origin)
                     (sub-vect (m corner2) new-origin)))))))
```

2.50-51

**Levels of language for robust design**

- The fundamental data abstractions in the picture language are painters. Representing them as procedures makes all the tools of procedural abstraction available to us.
- This example also uses *stratified design*, the notion that a complex system should be structured as a sequence of levels.
- Each time we start a new level, we treat the old complex things as primitive black boxes and combine them.

> Stratified design helps make programs *robust*, that is, it makes it likely that small changes in a specification will require correspondingly small changes in the program. … In general, each level of a stratified design provides a different vocabulary for expressing the characteristics of the system, and a different kind of ability to change it. [@2.2.4]

2.52

## 2.3: Symbolic Data

- So far our compound data has been made up of numbers.
- Now, we will start working with arbitrary symbols.

### 2.3.1: Quotation

- Lists with symbols look just like Lisp code, except we *quote* them.
- To quote in Lisp, we use the special form `(quote «exp»)`, or the shorthand `'«exp»`.
- For example, `'x` evaluates to the symbol “x” instead of the variable `x`’s value.
- This is like just natural language. If I say, “Say your name”, you will say your name. If I instead say, “Say ‘your name’”, you will literally say the words “your name”.

```
(define a 1)
(define b 2)
```

```
(list a b)
→ (1 2)

(list 'a 'b)
→ (a b)

(list 'a b)
→ (a 2)
```

- Now that we have quotation, we will use `'()` for the empty list instead of `nil`.
- We need another primitive now: `eq?`. This checks if two symbols are the same.

```
(eq? 'a 'b)
→ #f

(eq? 'a 'a)
→ #t
```

2.53-55

## 2.3.2: Example: Symbolic Differentiation

- Let's design a procedure that computes the derivative of an algebraic expression.
- This will demonstrate both symbol manipulation and data abstraction.

> Symbolic differentiation is of special historical significance in Lisp. It was one of the motivating examples behind the development of a computer language for symbol manipulation. [@2.3.2]

### The differentiation program with abstract data

- To start, we will only consider addition and multiplication.
- We need three differentiation rules: constant, sum, and product.
- Let's assume we already have these procedures:

| Procedure | Result |
|---|---|
| `(variable? e)` | Is `e` a variable? |
| `(same-variable? v1 v2)` | Are `v1`and `v2` the same variable? |
| `(sum? e)` | Is `e` a sum? |
| `(addend e)` | Addend of the sum `e` |
| `(augend e)` | Augend of the sum `e` |
| `(make-sum a1 a2)` | Construct the sum of `a1` and `a2` |
| `(product? e)` | Is `e` a product? |
| `(multiplier e)` | Multiplier of the product `e` |
| `(multiplicand e)` | Multiplicand of the product `e` |
| `(make-product m1 m2)` | Construct the product of `m1` and `m2` |

- Then, we can express the differentiation rules like so:

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

**Representing algebraic expressions**

- There are many ways we could represent algebraic expressions.
- The most straightforward way is parenthesized Polish notation: Lisp syntax.
- We can simplify answers in the constructor just like in the rational number example.
- Simplifying the same way a human would is hard, partly because the most “simplified” form is sometimes subjective.

2.56-58

## 2.3.3: Example: Representing Sets

There are a number of possible ways we could represent sets. A set is a collection of distinct objects. Our sets need to work with the following operations:

| Procedure | Result |
|---|---|
| `(element-of-set? x s)` | Is `x` a member of the set `s`? |
| `(adjoin-set x s)` | Set containing `x` and the elements of set `s` |
| `(union-set s t)` | Set of elements that occur in either `s` or `t` |
| `(intersection-set s t)` | Set of elements that occur in both `s` and `t` |

**Sets as unordered lists**

- One method is to just use lists. Each time we add a new element, we have to check to make sure it isn’t already in the set.
- This isn’t very efficient. `element-of-set?` and `adjoin-set` are $\Theta(n)$, while `union-set` and `intersection-set` are $\Theta(n^2)$.
- We can make `adjoin-set` $\Theta(1)$ if we allow duplicates, but then the list can grow rapidly, which may cause an overall decrease in performance.

2.59-60

**Sets as ordered lists**

- A more efficient representation is a sorted list.
- To keep things simple, we will only consider sets of numbers.
- Now, scanning the entire list in `element-of-set?` is a worst-case scenario. On average we should expect to scan half of the list.
- This is still $\Theta(n)$, but now we can make `union-set` and `intersection-set` $\Theta(n)$ too.

2.61-62

**Sets as binary trees**

- An even more efficient representation is a binary tree where left branches contain smaller numbers and right branches contain larger numbers.
- The same set can be represented by such a tree in many ways.
- If the tree is *balanced*, each subtree is about half the size of the original tree. This allows us to implement `element-of-set?` in $\Theta(\log(n))$ time.
- We'll represent each node by `(list number left-subtree right-subtree)`.
- Efficiency hinges on the tree being balanced. We can write a procedure to balance trees, or we could use a difference data structure (B-trees or red–black trees).

2.63-65

**Sets and information retrieval**

- The techniques discussed for sets show up again and again in information retrieval.
- In a data management system, each record is identified by a unique key.
- The simplest, least efficient method is to use an unordered list with $\Theta(n)$ access.
- For fast "random access", trees are usually used.
- Using data abstraction, we can start with unordered lists and then later change the constructors and selectors to use a tree representation.

2.66

## 2.3.4: Example: Huffman Encoding Trees

- A *code* is a system for converting information (such as text) into another form.
- ASCII is a *fixed-length* code: each symbol uses the same number of bits.
- Morse code is *variable-length*: since "e" is the most common letter, it uses a single dot.
- The problem with variable-length codes is that you must have a way of knowing when you've reached the end of a symbol. Morse code uses a temporal pause.
- Prefix codes, like Huffman encoding, ensure that no code for a symbol is a prefix of the code for another symbol. This eliminates any ambiguity.
- A Huffman code can be represented as a binary tree where each leaf contains a symbol and its weight (relative frequency), and each internal node stores the set of symbols and the sum of weights below it. Zero means "go left", one means "go right".

**Generating Huffman trees**

Huffman gave an algorithm for constructing the best code for a given set of symbols and their relative frequencies:

1. Begin with all symbols and their weights as isolated leaves.
2. Form an internal node branching off to the two least frequent symbols.
3. Repeat until all nodes are connected.

**Representing Huffman trees**

- Leaves are represented by `(list 'leaf symbol weight)`.
- Internal nodes are represented by `(list left right symbols weight)`, where `left` and `right` are subtrees, `symbols` is a list of the symbols underneath the node, and `weight` is the sum of the weights of all the leaves beneath the node.

**The decoding procedure**

The following procedure decodes a list of bits using a Huffman tree:

```
(define (decode bits tree)
  (define (decode-1 bits current-branch)
    (if (null? bits)
        '()
        (let ((next-branch
               (choose-branch (car bits) current-branch)))
          (if (leaf? next-branch)
              (cons (symbol-leaf next-branch)
                    (decode-1 (cdr bits) tree))
              (decode-1 (cdr bits) next-branch)))))
  (decode-1 bits tree))

(define (choose-branch bit branch)
  (cond ((= bit 0) (left-branch branch))
        ((= bit 1) (right-branch branch))
        (else (error "bad bit" bit))))
```

**Sets of weighted elements**

- Since the tree-generating algorithm requires repeatedly finding the smallest item in a set, we should represent a set of leaves and trees as a list ordered by weight.
- The implementation of `adjoin-set` is similar to , except that we compare items by weight, and the element being added is never already in the set:

```
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((< (weight x) (weight (car set))) (cons x set))
        (else (cons (car set)
                    (adjoin-set x (cdr set))))))
```

2.67-72

## 2.4: Multiple Representations for Abstract Data

- Data abstraction lets us write programs that work independently of the chosen representation for data objects. This helps control complexity.
- But it's not powerful enough: sometimes we want not just an abstracted underlying representation, but multiple representations.
- In addition to horizontal abstraction barriers, separating high-level from low-level, we need vertical abstraction barriers, allowing multiple design choices to coexist.
- We'll do this by constructing *generic procedures*, which can operate on data that may be represented in more than one way.

### 2.4.1: Representations for Complex Numbers

- We can represent a complex number $z = x + yi = re^{i\theta}$ as a list in two ways: in rectangular form $(x, y)$ or in polar form $(r, \theta)$.
- Rectangular form is convenient for addition and substraction, while polar form is convenient for multiplication and division.
- Our goal is to implement all the operations to work with either representation:

```
(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))))
```

- We can implement the constructors and selectors to use rectangular form or polar form, but how do we allow both?

### 2.4.2: Tagged Data

- One way to view data abstraction is as the "principle of least commitment". We waited until the last minute to choose a concrete representation, retaining maximum flexibility.
- We can take it even further and avoid committing to a single representation at all.
- To do this, we need some way of distinguishing between representations. We will do this with a *type tag* `'rectangular` or `'polar`.
- Each generic selector will strip off the type tag and use case analysis to pass the untyped data object to the appropriate specific selector.

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))
(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "bad tagged datum" datum)))
(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "bad tagged datum" datum)))

(define (rectangular? z)
  (eq? (type-tag z) 'rectangular))
(define (polar? z)
  (eq? (type-tag z) 'polar))
```

- For example, here is how we implement the generic `real-part`:

```
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
        (else (error "unknown type" z))))
```

### 2.4.3: Data-Directed Programming and Additivity

- The strategy of calling a procedure based on the type tag is called *dispatching on type*.
- The implementation in has two significant weaknesses:
    1. The generic procedures must know about all the representations.
    2. We must guarantee that no two procedures have the same name.
- The underlying issue is that the technique is not *additive*.
- The solution is to use a technique known as *data-directed programming*.
- Imagine a table with operations on one axis and types on the other axis:

| | **Polar** | **Rectangular** |
|---|---|---|
| Real part | `real-part-polar` | `real-part-rectangular` |
| Imaginary part | `imag-part-polar` | `imag-part-rectangular` |
| Magnitude | `magntiude-polar` | `magntiude-rectangular` |
| Angle | `angle-polar` | `angle-rectangular` |

- Before, we implemented each generic procedure using case analysis. Now, we will write a single procedure that looks up the operation name and argument type in a table.
- Assume we have the procedure `(put «op» «type» «item»)` which installs `«item»` in the the table, and `(get «op» «type»)` which retrieves it. (We will implement them in .)
- Then, we can define a collection of procedures, or *package*, to install the each representation of complex numbers. For example, the rectangular representation:

```
(define (install-rectangular-package)
  ;; Internal procedures
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (make-from-real-imag x y) (cons x y))
  (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-mag-ang r a)
    (cons (* r (cos a)) (* r (sin a))))

  ;; Interface to the rest of the system
  (define (tag x) (attach-tag 'rectangular x))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  (put 'make-from-real-imag 'rectangular
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'rectangular
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

- Next, we need a way to look up a procedure in the table by operation and arguments:

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

- Using `apply-generic`, we can define our generic selectors as follows:

```
(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))
```

2.73-74

**Message passing**

- Data-directed programming deals explicitly with the operation–type table.
- In , we made “smart operations” that dispatch based on type. This effectively decomposes the table into rows.
- Another approach is to make “smart data objects” that dispatch based on operation. This effectively decomposes the table into columns.
- This is called *message-passing*, and we can accomplish it in Scheme using closures.
- For example, here is how we would implement the rectangular representation:

```
(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" op))))
  dispatch)
```

- Message passing can be a powerful tool for structuring simulation programs.

2.75-76

## 2.5: Systems with Generic Operations

- In we saw how to create operations that work on multiple data representations.
- We can extend this further to create operations that are generic over different kinds of arguments, not just different representations of the same kind of data.
- We will develop a general arithmetic system using data-directed programming that works on several kinds of numbers.

### 2.5.1: Generic Arithmetic Operations

- We want `add` to work for primitive, rational, and complex numbers.
- We will attach a type tag to each kind of number. Complex numbers will have two levels of tagging: a `'complex` tag on top of the `'rectangular` or `'polar` tag.
- The tags get stripped off as the data is passed down through packages to the appropriate specific procedure.

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

- Here is the arithmetic package for Scheme numbers:

```
(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 'mul '(scheme-number scheme-number)
       (lambda (x y) (tag (* x y))))
  (put 'div '(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))
```

- We can implement similar packages for rational and complex numbers.

2.77-80

## 2.5.2: Combining Data of Different Types

- In our unified arithmetic system, we ignored an important issue: we did not consider operations that cross type boundaries, like adding a Scheme number to a rational.
- One solution would be to design a procedure for every possible combination of types. But this is cumbersome, and it violates modularity.

### Coercion

- Often the different data types are not completely independent.
- For example, any real number can be expressed as a complex number with an imaginary part of zero.
- We can make all operations work on combinations of Scheme numbers and complex numbers by first *coercing* the former to the latter.
- Here is a typical coercion procedure:

```
(define (scheme-number->complex n)
  (make-complex-from-real-imag (contents n) 0))
```

- The big advantage of coercion is that, although we may need to write many coercion procedures, we only need to implement each operation once per type.
- We could modify `apply-generic` to try coercion if there is no specific procedure available for the given types.
- But it gets complicated. What if both types can be converted to a third type? What if multiple coercions are required? We end up with a graph of relations among types.

### Hierarchies of types

- We can simplify the problem by using a *type hierarchy* instead of an arbitrary graph.
- In the hiearchy, each type is related to supertypes above it and subtypes below it.
- A *tower* is a hiearchy where each type has at most one supertype and subtype.
- Our numeric tower comprises integers, rationals, real numbers, and complex numbers.
- Coercion then simply becomes a matter of raising the argument whose type is lower in the tower to the level of the other type.
- We can write fewer procedures by allowing types to *inherit* their supertype's operations.
- In some cases we can simplify results by lowering a value down the type tower.

### Inadequacies of hierarchies

- In general, a type may have more than one subtype. This is easy to deal with.

- Allowing multiple super types (known as *multiple inheritance*) is tricky, since the type can be raised via multiple paths to search for a procedure.
- Having large numbers of interrelated types conflicts with modularity.

> Developing a useful, general framework for expressing the relations among different types of entities (what philosophers call "ontology") seems intractably difficult. The main difference between the confusion that existed ten years ago and the confusion that exists now is that now a variety of inadequate ontological theories have been embodied in a plethora of correspondingly inadequate programming languages. For example, much of the complexity of object-oriented programming languages – and the subtle and confusing differences among contemporary object-oriented languages – centers on the treatment of generic operations on interrelated types. [@2.5.fn52]

2.81-86

## 2.5.3: Example: Symbolic Algebra

- Manipulating symbolic algebraic expressions is hard.
- We can view them as trees of operators applied to operands.
- To keep things simple, we'll stick to polynomials.

### Arithmetic on polynomials

- Polynomials are expressed in terms of variables called *indeterminates*.
- A *univariate* polynomial has the form $c_0 + c_1x + c_2x^2 + \cdots + c_nx^n$.
- To avoid thorny issues of identity and sameness, we will consider our polynomials to be syntactic forms, not representations of mathematical functions.
- We will implement addition and multiplication for polynomials in the same variable.

```
(define (add-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                 (add-terms (term-list p1)
                            (term-list p2)))
      (error "polys not in same var" p1 p2)))

(define (mul-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                 (mul-terms (term-list p1)
                            (term-list p2)))
      (error "polys not in same var" p1 p2)))
```

- We will install these procedures in our generic arithmetic system:

```
(define (install-polynomial-package)
  ;; Internal procedures
  (define (make-poly variable term-list)
    (cons variable term-list))
  (define (variable p) (car p))
  (define (term-list p) (cdr p))
  (define (add-poly p1 p2) ...)
  (define (mul-poly p1 p2) ...)

  ;; Interface to rest of the system
  (define (tag p) (attach-tag 'polynomial p))
  (put 'add '(polynomial polynomial)
       (lambda (p1 p2) (tag (add-poly p1 p2))))
  (put 'mul '(polynomial polynomial)
       (lambda (p1 p2) (tag (mul-poly p1 p2))))
  (put 'make 'polynomial
       (lambda (var terms) (tag (make-poly var terms))))
  'done)
```

- Here is the implementation of `add-terms`:

```
(define (add-terms L1 L2)
  (cond ((empty-termlist? L1) L2)
        ((empty-termlist? L2) L1)
        (else
         (let ((t1 (first-term L1)) (t2 (first-term L2)))
           (cond ((> (order t1) (order t2))
                  (adjoin-term
                   t1 (add-terms (rest-terms L1) L2)))
                 ((< (order t1) (order t2))
                  (adjoin-term
                   t2 (add-terms L1 (rest-terms L2))))
                 (else
                  (adjoin-term
                   (make-term (order t1)
                              (add (coeff t1) (coeff t2)))
                   (add-terms (rest-terms L1)
                              (rest-terms L2)))))))))
```

- Since it uses the generic `add` internally, the system automatically works with polynomials whose coefficients are themselves polynomials in a different variable!

**Representing term lists**

- Our procedures `add-terms` and `mul-terms` always access terms sequentially from highest to lowest order, so we need some kind of ordered representation.
- For *dense* polynomials (mostly nonzero coefficients), a simple list is best.
  - *Example*: $x^5 + 2x^4 + 3x^2 - 2x - 5$ becomes `'(1 2 3 -2 -5)`.
- For *sparse* polynomials (mostly zero coefficients), an associative list is best.
  - *Example*: $x^{100} + 2x^2 + 1$ becomes `'((100 1) (2 2) (0 1))`.
- Here is an associative list representation:

```
(define (adjoin-term term term-list)
  (if (=zero? (coeff term))
      term-list
      (cons term term-list)))
```

```
(define (the-empty-termlist) '())
(define (first-term term-list) (car term-list))
(define (rest-terms term-list) (cdr term-list))
(define (empty-termlist? term-list) (null? term-list))
(define (make-term order coeff) (list order coeff))
(define (order term) (car term))
(define (coeff term) (cadr term))
```

2.87-91

### Hierarchies of types in symbolic algebra

- The data types in our polynomial algebra cannot easily be arranged in a tower.
- For example, how would we coerce $(x + 1)y^2$ and $(y + 1)x^2$ to a common type?

> It should not be surprising that controlling coercion is a serious problem in the design of large-scale algebraic-manipulation systems. Much of the complexity of such systems is concerned with relationships among diverse types. Indeed, it is fair to say that we do not yet completely understand coercion. In fact, we do not yet completely understand the concept of a data type. Nevertheless, what we know provides us with powerful structuring and modularity principles to support the design of large systems. [@2.5.3]

2.92

### Extended exercise: Rational functions

- Rational functions are "fractions" with polynomial numerators and denominators.
- We can use our rational package from , but we need to change a few things.
- To reduce rational functions, we need to be able to compute the GCD of polynomials, which in turn requires computing the remainder after polynomial division.
- To avoid fractional coefficients, we can multiply the result by an *integerizing factor*.
- Here is our algorithm for reducing a rational function to lowest terms:
  1. Compute the integerized GCD of the numerator and denominator.
  2. Multiply the numerator and denominator by an integerizing factor: the leading coefficient of the GCD raised to the power $1 + \max\{O_N, O_D\} - O_G$, where those constants refer to the order of the numerator, denominator, and GCD, respectively.
  3. Divide the results of (2) by the GCD from (1).
  4. Divide the results of (3) by the GCD of all their coefficients.
- The GCD algorithm is at the heart of every system that does operations on rational functions. Ours is very slow. Probabilistic algorithms like Zippel's are faster.

2.93-97

# 3: Modularity, Objects, and State

- So far, we've learned how to build abstractions with procedures and data.
- Abstraction is crucial to deal with complexity, but it's not the whole story. We also need organizational principles to guide the overall design of the program.
- We need strategies to help us build *modular* systems, which naturally divide into coherent parts that can be separately maintained.

> If we have been successful in our system organization, then to add a new feature or debug an old one we will have to work on only a localized part of the system. [@3]

- In this chapter we will investigate two prominent organizational strategies.
- The first views the system as a collection of distinct *objects* that change over time.
- The second focuses on *streams* of information that flow in the system.
- Both approaches raise significant linguistic issues in our programming.

## 3.1: Assignment and Local State

- The world is populated by independent objects possessing individual, changing state.
- When we say an object "has state", we mean its behavior is influenced by its history.
- Instead of remembering an object's history, we can summarize it with *state variables*.
- For example, the state variable for a bank account would record its current balance.

> Indeed, the view that a system is composed of separate objects is most useful when the state variables of the system can be grouped into closely coupled subsystems that are only loosely coupled to other subsystems. [@3.1]

- To model state variables in our programming language, we need an *assignment operator* that changes the value associated with a name.

### 3.1.1: Local State Variables

- Let's model the situation of withdrawing money from a bank account.
- `(withdraw «amount»)` should withdraw the given amount and return the new balance.
- If you try to withdraw too much, it should return the string "Insufficient funds".
- Suppose we begin with $100:

```
(withdraw 25)
→ 75

(withdraw 25)
→ 50

(withdraw 60)
```

```
→ "Insufficient funds"

(withdraw 15)
→ 35
```

- Notice that `(withdraw 25)` was called twice, but returned different values.
- This is a new kind of behavior. Up until now, the returned value of a procedure depended only on the arguments, like a mathematical function.
- Here's how we implement `withdraw`:

```
(define balance 100)

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

- This uses the special form `(set! «name» «new-value»)` to change the value of `balance`.
- The expression `(begin «exp1» «exp2» ... «expn»)` evaluates all the expressions in sequence and returns the value of the last one.
- Instead of defining `balance` globally, we can *encapsulate* it within `withdraw`:

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

- Unfortunately, the substitution model of evaluation from is no longer adequate once we have assignment in our procedures.
- For now, we technically have no way to understand how these procedures work. We will develop a new model soon.
- The following procedure creates “withdraw processors”:

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

(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))

(W1 50)
→ 50

(W2 70)
→ 30
```

```
(W2 40)
→ "Insufficient funds"

(W1 40)
→ 10
```

- `W1` and `W2` are completely independent objects, each with its own local state variable.
- We can also create a “bank-account object” that responds to multiple messages, all operating on the same local state:

```
(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)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "unknown request" m))))
  dispatch)
```

3.1-4

### 3.1.2: The Benefits of Introducing Assignment

> Introducing assignment into our programming language leads us into a thicket of difficult conceptual issues. Nevertheless, viewing systems as collections of objects with local state is a powerful technique for maintaining a modular design. [@3.1.2]

- Consider the procedure `rand`, which returns an integer chosen at random.
- It’s not clear what “random” means, but we can make a *pseudo-random* sequence with a seed value `random-init` and a procedure `rand-update` that produces the next value:

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

> The relation between “real randomness” and so-called pseudo-random sequences, which are produced by well-determined computations and yet have suitable statistical properties, is a complex question involving difficult issues in mathematics and philosophy. [@3.1.fn6]

- One use for randomness is the *Monte Carlo method*, which finds an approximate solution to a numerical problem by studying random numbers in a probabilistic model.

- We can use the Monte Carlo method to approximate $\pi$, knowing that the probability that two randomly chosen integers have 1 as their GCD is $6/\pi^2$.

```
(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))
          ((experiment)
           (iter (- trials-remaining 1) (+ trials-passed 1)))
          (else
           (iter (- trials-remaining 1) trials-passed))))
  (iter trials 0))
```

- If we had to use `rand-update` directly, our Monte Carlo program would betray some painful breaches of modularity.

> The general phenomenon illustrated by the Monte Carlo example is this: From the point of view of one part of a complex process, the other parts appear to change with time. They have hidden time-varying local state. If we wish to write computer programs whose structure reflects this decomposition, we make computational objects (such as bank accounts and random-number generators) whose behavior changes with time. We model state with local state variables, and we model the changes of state with assignments to those variables. [@3.1.2]

3.5-6

## 3.1.3: The Costs of Introducing Assignment

- The advantages of local state and assignment come at a price: the substitution model of procedure application from breaks down. We need a more complex model.
- Programming without the use of assignment, as we did in the first two chapters, is called *functional programming*.
- Consider a simplified version of the `make-withdraw` procedure from :

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
```

- Now observe what happens when we try to apply the substitution model to it:

```
((make-simplified-withdraw 25) 20)
((lambda (amount) (set! balance (- 25 amount)) 25) 20)
```

```
(set! balance (- 25 20)) 25
25
```

- This is the wrong answer. We shouldn't have substituted 25 for `balance` everywhere, because the assignment changed it.
- Before, a variable was simply a name for a value. Now, with assignment, it somehow refers to a place where a value can be stored, and this value can be changed.

### Sameness and change

- A language that supports "equals can be substituted for equals" is *referentially transparent*. Our language was referentially transparent until we introduced `set!`.
- By introducing change into our computational models, many previously straightforward notions become problematic, such as the concept of two things being "the same".
- If we have `(make-withdraw 25)` and `(make-withdraw 25)`, are they the same? No, because they can have different local state.
- If we have `(define peter-acc (make-account 100))`, there is a big difference between `(define paul-acc (make-account 100))` and `(define paul-acc peter-acc)`.
- In the first case, they have distinct accounts. In the second, both names are *aliased* to the same account, so withdrawing from either one will affect the other.

> Bugs can occur in our programs if we forget that a change to an object may also, as a "side effect", change a "different" object because the two "different" objects are actually a single object appearing under different aliases. These so-called side-effect bugs are so difficult to locate and to analyze that some people have proposed that programming languages be designed in such a way as to not allow side effects or aliasing. [@3.1.fn10]

### Pitfalls of imperative programming

- Programming that makes heavy use of assignment is called *imperative programming*.
- Imperative programs are susceptible to bugs that cannot occur in functional programs.
- Things will get even worse when we throw concurrency into the mix in .

> In general, programming with assignment forces us to carefully consider the relative orders of the assignments to make sure that each statement is using the correct version of the variables that have been changed. This issue simply does not arise in functional programs. [@3.1.3]
>
> In view of this, it is ironic that introductory programming is most often taught in a highly imperative style. This may be a vestige of a belief, common throughout the 1960s and 1970s, that programs that call procedures must inherently be less efficient than programs that perform assignments. … Alternatively it may reflect a view that step-by-step assignment is easier for beginners to visualize than procedure call. Whatever the reason, it often

> saddles beginning programmers with “should I set this variable before or after that one” concerns that can complicate programming and obscure the important ideas. [@3.1.fn11]

3.7-8

## 3.2: The Environment Model of Evaluation

- Recall the substitution model:

  > To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument. [@3.2]

- This is no longer adequate once we allow assignment.
- Variables are no longer merely names for values; rather, a variable designates a “place” in which values can be stored.
- These places will be maintained in structures called *environments*.
- An environment is a sequence of *frames*. A frame is a table of *bindings*. A binding associates a variable name with one value.
- Each frame also has a pointer to its enclosing environment, unless it is considered to be global.
- The value of a variable with respect to an environment is the value given by the binding in the first frame of the environment that contains a binding for that variable.
- If no frame in the sequence specifies a binding for the variable, then the variable is *unbound* in the environment.
- A binding *shadows* another (of the same variable) if the other is in a frame that is further in the sequence.
- The environment determines the context in which an expression should be evaluated. An expression has no meaning otherwise.
- The global environment consists of a single frame, and it is implicitly used in interactions with the interpreter.

### 3.2.1: The Rules for Evaluation

- To evaluate a combination:
  1. Evaluate the subexpressions of the combination.
  2. Apply the value of the operator subexpression to the values of the operand subexpressions.
- The environment model redefines the meaning of “apply”.
- A procedure is created by evaluating a λ-expression relative to a given environment.
- The resulting procedure object is a pair consisting of the text of the λ-expression and a pointer to the environment in which the procedure was created.
- To apply a procedure to arguments, create a new environment whose frame binds the parameters to the values of the arguments and whose enclosing environment is specified by the procedure.
- Then, within the new environment, evaluate the procedure body.
- `(lambda (x) (* x x))` evaluates a to pair: the parameters and the procedure body as one item, and a pointer to the global environment as the other.

- `(define square (lambda (x) (* x x)))` associates the symbol `square` with that procedure object in the global frame.
- Evaluating `(define «var» «val»)` creates a binding in the current environment frame to associate `«var»` with `«val»`.
- Evaluating `(set! «var» «val»)` locates the binding of `«var»` in the current environment (the first frame that has a binding for it) and changes the bound value to `«val»`.
- We use `define` for variables that are currently unbound, and `set!` for variables that are already bound.

## 3.2.2: Applying Simple Procedures

- Let's evaluate `(f 5)`, given the following procedures:

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

- These definitions create bindings for `square`, `sum-of-squares`, and `f` in the global frame.
- To evaluate `(f 5)`, we create an environment $E_1$ with a frame containing a single binding, associating `a` with `5`.
- In $E_1$, we evaluate `(sum-of-squares (+ a 1) (* a 2))`.
- We must evaluate the subexpressions of this combination.
- We find the value associated with `sum-of-squares` not in $E_1$ but in the global environment.
- Evaluating the operand subexpressions yields `6` and `10`.
- Now we create $E_2$ with a frame containing two bindings: `x` is bound to `6`, and `y` is bound to `10`.
- In $E_2$, we evaluate `(+ (square x) (square y))`.
- The process continues recursively. We end up with `(+ 36 100)`, which evaluates to `136`.

3.9

## 3.2.3: Frames as the Repository of Local State

- Now we can see how the environment model makes sense of assignment and local state.
- Consider the "withdrawal processor":

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

- This places a single binding in the global environment frame.

- Consider now `(define W1 (make-withdraw 100))`.
    - We set up $E_1$ where `100` is bound to the formal parameter `balance`, and then we evaluate the body of `make-withdraw`.
    - This returns a lambda procedure whose environment is $E_1$, and this is then bound to `W1` in the global frame.
- Now, we apply this procedure: `(W1 50)`.
    - We construct a frame in $E_2$ that binds `amount` to `50`, and then we evaluate the body of `W1`.
    - The enclosing environment of $E_2$ is $E_1$, *not* the global environment.
    - Evaluating the body results in the `set!` rebinding `balance` in $E_1$ to the value `(- 100 50)`, which is `50`.
    - After calling `W1`, the environment $E_2$ is irrelevant because nothing points to it.
    - Each call to `W1` creates a new environment to hold `amount`, but uses the same $E_1$ (which holds `balance`).
- `(define W2 (make-withdraw 100))` creates another environment with a `balance` binding.
    - This is independent from $E_1$, which is why the `W2` object and its local state is independent from `W1`.
    - On the other hand, `W1` and `W2` share the same code.

3.10

### 3.2.4: Internal Definitions

- With block structure, we nested definitions using `define` to avoid exposing helper procedures.
- Internal definitions work according to the environmental model.
- When we apply a procedure that has internal definitions, there are `define` forms at the beginning of the body.
- We are in $E_1$, so evaluating these adds bindings to the first frame of $E_1$, right after the arguments.
- When we apply the internal procedures, the formal parameter environment $E_n$ is created, and its enclosing environment is $E_1$ because that was where the procedure was defined.
- This means each internal procedure has access to the arguments of the procedure they are defined within.
- The names of local procedures don’t interfere with names external to the enclosing procedure, due to $E_1$.

3.11

## 3.3: Modeling with Mutable Data

- We previously looked at compound data and data abstraction.
- To model stateful objects, we need *mutators* in addition to constructors and selectors.
- A mutator is a procedure that modifies the data object.

## 3.3.1: Mutable List Structure

- The primitive mutators for pairs are `set-car!` and `set-cdr!`.
- `(set-car! p x)` changes the `car` of the pair `p`, making it point to `x` instead.
- The old `car` is unreachable garbage. We will see later how Lisp recycles this memory.
- We could implement `cons` in terms of these two procedures in addition to a `get-new-part` procedure.

```
(define (cons x y)
  (let ((new (get-new-pair)))
    (set-car! new x)
    (set-cdr! new y)
    new))
```

3.12-14

### Sharing and identity

- Consider `(define x (list 'a 'b))` and `(define z1 (cons x x))`.
- `z1` is a pair whose `car` and `cdr` both point to the same `x`.
- In contrast: `(define z2 (cons (list 'a 'b)) (list 'a 'b))`.
- In `z2`, the two `(a b)` lists are distinct, although the actual symbols are shared.
- Before assignment, we would think `z1` and `z2` were "the same".
- The sharing is undetectable without mutators on list structure.
- It *is* detectable in our environmental model.
- If we `set-car!` on the `car`, this will change both `a` symbols in `z1` but only the first in `z2`.
- We can use the predicate `eq?` to test for sameness in the sense of identity.
- `(eq? x y)` tests whether `x` and `y` point to the same object.
- We can exploit sharing for good, but it can be dangerous.

3.15-19

### Mutation is just assignment

- Earlier we said we can represent pairs purely in terms of procedures:

```
(define (cons x y)
  (lambda (sel)
    (sel x y)))
(define (car p)
  (p (lambda (x y) x)))
(define (cdr p)
  (p (lambda (x y) y)))
```

- The same is true of mutable data. We can implement mutators with procedures and assignment alone:

```
(define (cons x y)
  (define (set-x! v) (set! x v))
  (define (set-y! v) (set! y v))
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          ((eq? m 'set-car!) set-x!)
          ((eq? m 'set-cdr!) set-y!)
          (else (error "Undefined operation: CONS" m))))
  dispatch)
```

- Assignment and mutation are equipotent: each can be implemented in terms of the other.

3.20

## 3.3.2: Representing Queues

- The mutators allow us to construct new data structures.
- A *queue* is a sequence in which items are inserted at one end (the rear) and deleted from the other end (the front).
- It is also called a FIFO buffer (first in, first out).
- We define the following operations for data abstraction:
    - a constructor `(make-queue)`,
    - a predicate `(empty-queue? q)`,
    - a selector `(front-queue q)`,
    - two mutators: `(insert-queue! q x)` and `(delete-queue! q)`.
- A simple list representation is inefficient because we have to scan to get to one end.
- Scanning a list takes $\Theta(n)$ operations.
- A simply modification lets us implement all the operations with $\Theta(1)$ time complexity: keep a pointer to the end as well.
- A queue is a pair formed by `cons`ing the front-pointer and the rear-pointer of a normal list.

3.21-23

## 3.3.3: Representing Tables

- In a one-dimensional table, each value is indexed by one key.
- We can implement it as a simple list of *records*.
- A record is a pair consisting of a key and an associated value.
- The first record in the table is a dummy, and it hold the arbitrarily chosen symbol `'*table*`.
    - If a table was just a pointer to the first actual record, then when we wouldn't be able to write a mutator to add a record to the front.
    - We would need to change the table to point to the new front, but `set!` on a formal parameter doesn't work as desired.
    - It would only change the parameter in $E_1$, not the value in the calling environment.
    - We didn't need to worry about this with sets because a set was a pair of two pointers a therefore we could mutate the `car` and `cdr` – but we couldn't change the set *itself*, since it was effectively a pointer to the pair, *copied* on application.
    - We are essentially using a pointer; we are using one cell of the pair. Some schemes provide `box`, `unbox`, and `set-box!` for this purpose.
- The `lookup` procedure returns the value associated with a key in a table, or `false` if it cannot be found.

- It uses `assoc`, which returns the whole record rather than just the associated value.

**Two-dimensional tables**

- Two-dimensional tables are indexed by two keys.
- In some cases, we could just use a one dimensional table whose keys are pairs of keys.
- We can implement a two-dimensional table as a one-dimensional table whose values are themselves one-dimensional tables.
- We could just use them like that without any specific procedures. However, insertion with two keys is complex enough to merit a convenient two-dimensional procedure.
- We don't need to box the subtables. They key of the record serves the purpose of `'*table*`.

**Creating local tables**

- With our `lookup` and `insert!` procedures taking the table as an argument, we can manage multiple tables.
- Another approach is to have separate procedures for each table.
- We could use the message-passing style with the `dispatch` procedure that we've seen a few times already.
- We could also take the λ-calculus approach:

```
(define (make-table) (lambda (k) false))
(define (lookup key table) (table key))
(define (insert! key value table)
  (lambda (k)
    (if (eq? k key)
        value
        (table k))))
```

3.24-27

## 3.3.4: A Simulator for Digital Circuits

- Digital circuits are made up of simple elements.
- Networks of these simple elements can have very complex behavior.
- We will design a system to simulator digital logic. This type of program is called an *event-driven simulation*.
- Our computational model is based on the physical components.
    - A *wire* carries a *digital signal*, which is either 0 or 1.
    - A *function box* connects to input wires and output wires.
    - The output signal is delayed by a time that depends on the type of function box.
    - Our primitive function boxes are the *inverter*, *and-gate*, and *or-gate*. Each has its own delay.
    - We construct complex functions by connecting primitives.
    - Multiple outputs are not necessarily generated at the same time.
- We construct wires with `(make-wire)`.
- Evaluating `(define a (make-wire))` and `(define b (make-wire))` followed by `(inverter a b)` connects `a` and `b` with an inverter.

- The primitive functions are our primitive elements; wiring is the means of combination; specifying wiring patterns as procedures is the means of abstraction.

### Primitive function boxes

- The primitives boxes implement “forces” by which changes in the signal of one wire influence the signal of another.
- We have the following operations on wires:
  - `(get-signal «wire»)` returns the current value of the signal.
  - `(set-signal! «wire» «value»)` changes the value of the signal.
  - `(add-action! «wire» «procedure-of-no-arguments»)` asserts that the given procedure should be run whenever the signal on the wire changes value.
- The procedure `after-delay` executes a procedure after a given time delay.

3.28-30

### Representing wires

- Our wires will be computational objects each having two local state variables: `signal-value` and `action-procedures`.
- We use the message-passing style as before.
- Wires have time-varying signals and can be incrementally attached to devices. They are a good example of when you need to use a mutable object in the computational model.
- A wire is shared between the devices connected to it. If one changes it, the rest see the change.
- This would be impossible if you didn’t model the wire as an identity, separate from its signal value.

> The truth of the matter is that, in a language in which we can deal with procedures as objects, there is no fundamental difference between “procedures” and “data”, and we can choose our syntactic sugar to allow us to program in whatever style we choose. [@3.3.fn27]

### The agenda

- The only thing left is `after-delay`.
- An *agenda* is a data structure that schedules things to do.
- For an agenda `(define a (make-agenda))`, the operations `(empty-agenda? a)`, `(first-agenda-item a)`, `(remove-first-agenda-item! a)`, and `(current-time a)` are self-explanatory.
- We schedule new items with `(add-to-agenda «time» «action» a)`.
- We call the global agenda `the-agenda`.
- The simulation is driven by `propagate`, which executes each item on the agenda in sequence.

### A sample simulation

3.31

### Implementing the agenda

- The agenda is a pair: the current time, and a list of *time segments* sorted in increasing order of time.
- A time segment is a pair: a number (the time) and a queue of procedures that are scheduled to run during that time segment.
- To add an action to the agenda, we scan its segments, examining their times. If we find the right time, we add the action to that segment's queue. Otherwise, we create a new segment.
- To remove the first agenda item, we delete the first item in the first queue, and if this makes the queue empty, we delete the first time segment as well.
- Whenever we extract the first item with `first-agenda-item`, we also update the current time.

3.32

## 3.3.5: Propagation of Constraints

- We often organize computer programs in one direction: from input to output.
- On the other hand, we often model systems in terms of relations among quantities.
- The equation $dAE = FL$ is not one-directional.
- We will create a language to work in terms of the relations themselves, so that we don't have to write five procedures.
- The primitive elements are *primitive constraints*.
    - `(adder a b c)` specifies $a + b = c$.
    - `(multiplier x y z)` specifies $xy = z$.
    - `(constant 3.14 x)` says that the value of `x` must be 3.14.
- The means of combination are constructing *constraint networks* in which constraints are joined by *connectors*.
- The means of abstraction are procedures.

### Using the constraint system

- We create connectors with `(make-connector)`, just like wires.
- We use `(probe "name" «connector»)`, again just like wires.
- `(set-value! «connector» «value» 'user)` assigns a value to the connector, and this information propagates through the network.
- This will give an error if the new value causes a contradiction.
- `(forget-value! «connector» 'user)` undoes the assignment.

### Implementing the constraint system

- The overall system is simpler than the digital circuit system because there are no propagation delays.
- There are five basic operations on a connector `c`: `(has-value? c)`, `(get-value c)`, `(set-value! c «value» «informant»)`, `(forget-value! c «retractor»)`, and `(connect c «constraint»)`.
- The procedures `inform-about-value` and `inform-about-no-value` tells the constraint that a connector has (lost) a value.
- Whenever an adder gets a new value, it checks if it has two and can calculate the third.

### Representing connectors

- A connector is a procedural object with local state variables – again, just like a wire.
- Each time the connector's value is set, it remembers the informant. This could be a constraint, or a symbol like `'user`.
- `for-each-except` is used to notify all *other* constraints.

3.33-37

# 3.4: Concurrency: Time Is of the Essence

- The power of stateful computational objects comes at a price: the loss of referential transparency.
- This gives rise to a thicket of questions about sameness and change, and we had to create a more intricate evaluation model.
- The central issue is that by introducing assignment we are forced to admit *time* in the computational model.
- We can go further in structuring the model to match the world: in the physical world, we perceive simultaneous changes.
- We want computational processes executing *concurrently*.
- Writing programs this way forces us to avoid inessential timing constraints, making the program more modular.
- It can also provide a speed advantage on multicore computers.
- The complexities introduced by assignment become even more problematic in the presence of concurrency.

## 3.4.1: The Nature of Time in Concurrent Systems

- On the surface, time is straightforward: it is an ordering.
- Two events either occur in one order, or the other, or simultaneously.
- Consider `(set! balance (- balance amount))`. There are three steps: accessing the value of `balance`, computing the new balance, and setting `balance` to this value.
- Two such expressions executed concurrently on the same `balance` variable could have their three steps interleaved.
- The general problem is that, when concurrent processes share a state variable, they may try to change it at the same time.

> To quote some graffiti seen on a Cambridge building wall: "Time is a device that was invented to keep everything from happening at once". [@3.4.fn35]

### Correct behavior of concurrent programs

- We already know we have to be careful about order with `set!`.
- With concurrent programs we must be especially careful.
- A very stringent restriction to ensure correctness: disallow changing more than one state variable at a time.
- A less stringent restriction: to ensure that the system produces the same result as if the processes had run sequentially in some order (we don't specify a particular order).

- Concurrent programs are inherently *nondeterministic*, because we don't what order of execution its result is equivalent to, so there is a set of possible values it could take.

3.38

## 3.4.2: Mechanisms for Controlling Concurrency

- If one process has three ordered events $(a, b, c)$ and another, running concurrently, has three ordered events $(x, y, z)$, then there are twenty ways of interleaving them.
- The programmer would have to consider the results in all twenty cases to be confident in the program.
- A better approach is to use mechanisms to constrain interleaving to ensure correct behavior.

### Serializing access to shared state

- Serialization groups procedures into sets such and prevents multiple procedures in the same set from executing concurrently.
- We can use this to control access to shared variables.
- Before, assignments based on a state variables current value were problematic. We could solve this with the set {`get-value`, `set-value!`, and `swap-value!`} where the latter is defined like so:

```
(define (swap-value! f)
  (set-value! (f (get-value))))
```

### Serializers in Scheme

- Suppose we have a procedure `parallel-execute` that takes a variable number of arguments that are procedures of no arguments, and executes them all concurrently.
- We construct *serializers* with `(make-serializer)`.
- A serializer takes a procedure as its argument and returns a serialized procedure that behaves like the original.
- Calls to the same serializer return procedures in the same set.

3.39-42

### Complexity of using multiple shared resources

- Serializers are powerful, and easy to use for one resource.
- Things get much more difficult with multiple shared resources.
- Suppose we want to swap the balances in two bank accounts:

```
(define (exchange acc1 acc2)
  (let ((diff (- (acc1 'balance) (acc2 'balance))))
    ((acc1 'withdraw) diff)
    ((acc2 'deposit) diff)))
```

- Serializing deposits and withdrawals themselves is not enough to ensure correctness.

- The exchange comprises four individually serialized steps, and these may interleave with a concurrent process.
- One solution is to expose the serializer from `make-account`, and use that to serialize the entire exchanging procedure.
- We would have to manually serialize deposits, but this would give us the flexibility to serialize the exchanging procedure.

3.43-45

### Implementing serializers

- Serializers are implemented in terms of the primitive *mutex*.
- A mutex can be *acquired* and it can be *released*.
- Once acquired, no other acquire operations can proceed until the mutex is released.
- Each serializer has an associated mutex.
- A serialized procedure (created with `(s proc)` where `s` is a serializer) does the following when it is run:
    - acquire the mutex,
    - run the procedure `proc`,
    - release the mutex.
- The mutex is a mutable object, represented by a *cell* (a one-element list). It holds a boolean value, indicating whether or not it is currently locked.
- To acquire the mutex, we test the cell. We wait until it is false, then we set it to true and proceed.
- To release the mutex, we set its contents to false.

3.46-47

### Deadlock

- Even with a proper implementation of mutexes and serializers, we still have a problem with the account exchanging procedure.
- We serialize the whole procedure with both accounts so that an account may only participate in one exchange at a time.
- There are two mutexes, so it is possible for something to happen in between acquiring the first and the second.
- If we exchange `a1` with `a2` and concurrently do the reverse exchange, it is possible for the first process to lock `a1` and the second process to lock `a2`.
- Now both need to lock the other, but they can't. This situation is called *deadlock*.
- In this case, we can fix the problem by locking accounts in a particular order based on a unique identifier.
- In some cases, it is not possible to avoid deadlock, and we simply have to "back out" and try again.

3.48-49

### Concurrency, time, and communication

- Concurrency can be tricky because it's not always clear what is meant by "shared state".
- It also becomes more complicated in large, distributed systems.

- The notion of time in concurrency control must be intimately linked to *communication*.
- There are some parallels with the theory of relativity.

## 3.5: Streams

- We've used assignment as a powerful tool and dealt with some of the complex problems it raises.
- Now we will consider another approach to modeling state, using data structures called *streams*.
- We want to avoid identifying time in the computer with time in the modeled world.
- If $x$ is a function of time $x(t)$, we can think of the identity $x$ as a history of values (and these don't change).
- With time measured in discrete steps, we can model a time function as a sequence of values.
- Stream processing allows us to model state without assignments.

### 3.5.1: Streams Are Delayed Lists

- If we represent streams as lists, we get elegance at the price of severe inefficiency (time *and* space).
- Consider adding all the primes in an interval. Using `filter` and `reduce`, we waste a lot of space storing lists.
- Streams are lazy lists. They are the clever idea of using sequence manipulations without incurring the cost.
- We only construct an item of the stream when it is needed.
- We have `cons-stream`, `stream-car`, `stream-cdr`, `the-empty-stream`, and `stream-null?`.
- The `cons-stream` procedure must not evaluate its second argument until it is accessed by `stream-cdr`.
- To implement streams, we will use *promises*. `(delay «exp»)` does not evaluate the argument but returns a promise. `(force «promise»)` evaluates a promise and returns the value.
- `(cons-stream a b)` is a special form equivalent to `(cons a (delay b))`.

#### The stream implementation in action

> In general, we can think of delayed evaluation as "demand-driven" programming, we herby each stage in the stream process is activated only enough to satisfy the next stage. [@3.5.1]

#### Implementing `delay` and `force`

- Promises are quite straightforward to implement.
- `(delay «exp»)` is syntactic sugar for `(lambda () «exp»)`.
- `force` simply calls the procedure. We can optimize it by saving the result and not calling the procedure a second time.
- The promise stored in the `cdr` of the stream is also known as a *thunk*.

3.50-52

## 3.5.2: Infinite Streams

- With lazy sequences, we can manipulate infinitely long streams!
- We can define Fibonacci sequence explicitly with a generator:

```
(define (fibgen a b) (cons-stream a (fibgen b (+ a b))))
(define fibs (fibgen 0 1))
```

- As long as we don't try to display the whole sequence, we will never get stuck in an infinite loop.
- We can also create an infinite stream of primes.

### Defining streams implicitly

- Instead of using a generator procedure, we can define infinite streams implicitly, taking advantage of the laziness.

```
(define fibs
  (cons-stream
   0
   (cons-stream 1 (stream-map + fibs (stream-cdr fibs)))))
(define pot
  (cons-stream 1 (stream-map (lambda (x) (* x 2)) pot)))
```

- This implicit technique is known as *corecursion*. Recursion works backward towards a base case, but corecursion works from the base and creates more data in terms of itself.

3.53-62

## 3.5.3: Exploiting the Stream Paradigm

- Streams can provide many of the benefits of local state and assignment while avoiding some of the theoretical tangles.
- Using streams allows us to have different module boundaries.
- We can focus on the whole stream/series/signal rather than on individual values.

### Formulating iterations as stream processes

- Before, we made iterative processes by updating state variables in recursive calls.
- To compute the square root, we improved a guess until the values didn't change very much.
- We can make a stream that converges on the square root of `x`, and a stream to approximate $\pi$.
- One neat thing we can do with these streams is use sequence accelerators, such as Euler's transform.

3.63-65

### Infinite streams of pairs

- Previously, we handled traditional nested loops as processes defined on sequences of pairs.
- We can find all pairs $(i,j)$ with $i \le j$ such that $i + j$ is prime like this:

```
(stream-filter
 (lambda (pair) (prime? (+ (car pair) (cadr pair))))
 int-pairs)
```

- We need some way of producing a stream of all integer pairs.
- More generally, we can combined two streams to get a two-dimensional grid of pairs, and we want to reduce this to a one-dimensional stream.
- One way to do this is to use `interleave` in the recursive definition, in order to handle infinite streams.

3.66-72

### Streams as signals

- We can use streams to model signal-processing systems.
- For example, taking the integral of a signal:

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

3.73-76

## 3.5.4: Streams and Delayed Evaluation

- The use of `delay` in `cons-stream` is crucial to defining streams with feedback loops.
- However, there are cases where we need further, explicit uses of `delay`.
- For example, solving the differential equation $dy/dt = f(y)$ where $f$ is a given function.
- The problem here is that `y` and `dy` will depend on each other.
- To solve this, we need to change `integral` to take a delayed integrand.

3.77-80

### Normal-order evaluation

- Explicit use of `delay` and `force` is powerful, but makes our programs more complex.
- It creates two classes of procedures: normal, and ones that take delayed arguments.
- (This is similar to the sync vs. async divide in modern programming languages.)
- This forces us to define separate classes of higher-order procedures, unless we make everything delayed, equivalent to normal-order evaluation (like Haskell).

- Mutability and delayed evaluation do not mix well.

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

- Modularity through encapsulation is a major benefit of introducing assignment.
- Stream models can provide equivalent modularity without assignment.
- For example, we can reimplement the Monte Carlo simulation with streams.

3.81-82

#### A functional-programming view of time

- Streams represent time explicitly, decoupling the simulated world from evaluation.
- They produce stateful-seeming behavior but avoid all the thorny issues of state.
- However, the issues come back when we need to merge streams together.
- Immutability is a key pillar of *functional programming languages*.

> We can model the world as a collection of separate, time-bound, interacting objects with state, or we can model the world as a single, timeless, stateless unity. Each view has powerful advantages, but neither view alone is completely satisfactory. A grand unification has yet to emerge. [@3.5.5]
>
> The object model approximates the world by dividing it into separate pieces. The functional model does not modularize along object boundaries. The object model is useful when the unshared state of the "objects" is much larger than the state that they share. An example of a place where the object viewpoint fails is quantum mechanics, where thinking of things as individual particles leads to paradoxes and confusions. Unifying the object view with the functional view may have little to do with programming, but rather with fundamental epistemological issues. [@3.5.fn76]

# 4: Metalinguistic Abstraction

- Expert programmers build up abstractions from simpler concepts to higher-level ones, and preserve modularity by adopting appropriate large-scale views of system structure.
- However, with increasingly complex problems we will find that Lisp, or any programming language, is not sufficient.

> We must constantly turn to new languages in order to express our ideas more effectively. Establishing new languages is a powerful strategy for controlling complexity in engineering design; we can often enhance our ability to deal with a complex problem by adopting a new language that enables us to describe (and hence to think about) the problem in a different way, using [terms] that are particularly well suited to the problem at hand. [@4]

- *Metalinguistic abstraction* means establishing new languages.
- An *evaluator* (or *interpreter*) is a procedure that implements a programming language.

> It is no exaggeration to regard this as the most fundamental idea in programming: The evaluator, which determines the meaning of expressions in a programming language, is just another program. [@4]

- Lisp is particularly well suited to metalinguistic abstraction.

## 4.1: The Metacircular Evaluator

- We will implement a Lisp evaluator as a Lisp program.
- The metacircular evaluator implements the environment model of evaluation:
    1. To evaluate a combination, evaluate subexpressions and then apply the operator subexpression to the operand subexpressions.
    2. To apply a procedure to arguments, evaluate the body of the procedure in a new environment. To construct the new environment, extend the environment part of the procedure object by a frame in which the formal parameters of the procedure are bound to the arguments to which the procedure is applied.
- This embodies the interplay between two critical procedures, `eval` and `apply`.

### 4.1.1: The Core of the Evaluator

#### `Eval`

- `eval` classifies an expression and directs its evaluation in an environment.
- We use *abstract syntax* to avoid committing to a particular syntax in the evaluator.

```
(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-definition 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`

- `apply` classifies a procedure and directs its application to a list of arguments.
- If compound, it evaluates the procedure body in an extended environment.

```
(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**

**Conditionals**

**Sequences**

**Assignments and definitions**

4.1

### 4.1.2: Representing Expressions

- The evaluator is reminiscent of the symbolic differentiator: both make recursive computations on compound expressions, and both use data abstraction.
- The syntax of the language is determined solely by procedures that classify and extract pieces of expressions. For example:

```
(define (quoted? exp) (tagged-list? exp 'quote))
(define (text-of-quotation exp) (cadr exp))
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))
```

**Derived expressions**

- Some special forms can be defined in terms of others.
- For example, we can reduce `cond` to an `if` expression:

```
(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))
(define (cond-else-clause? clause) (eq? (cond-predicate clause)
'else))
(define (cond-predicate clause) (car clause))
(define (cond-actions clause) (cdr clause))
(define (cond->if exp) (expand-clauses (cond-clauses exp)))

(define (expand-clauses clauses)
  (if (null? clauses)
      'false ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
```

```
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last: COND->IF"
clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))
```

- Practical Lisp systems allow the user to define new derived expressions by syntactic transformation. These are called *macros*.
- There is much research on avoiding name-conflict problems in macro definition languages.

4.2-10

## 4.1.3: Evaluator Data Structures

### Testing of predicates

- Anything other than `false` is considered "truthy".

```
(define (true? x) (not (eq? x false)))
(define (false? x) (eq? x false))
```

### Representing procedures

- `(apply-primitive-procedure «proc» «args»)` applies a primitive procedure to `«args»`.
- `(primitive-procedure? «proc»)` tests whether `«proc»` is a primitive procedure.
- Compound procedures are represented by the following data structure:

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
(define (compound-procedure? p) (tagged-list? p 'procedure))
(define (procedure-parameters p) (cadr p)) (define (procedure-body
p) (caddr p))
(define (procedure-environment p) (cadddr p))
```

### Operations on environments

- `(lookup-variable-value «var» «env»)` returns the value bound to a variable.
- `(extend-environment «variables» «values» «base-env»)` returns a new environment extended with a frame containing the given bindings.
- `(define-variable! «var» «value» «env»)` adds a binding to the first frame of `«env»`.
- `(set-variable-value! «var» «value» «env»)` changes a binding in `«env»`.
- Here is a partial implementation:

```
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())

(define (make-frame variables values) (cons variables values))
(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))

(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame) (frame-values frame)))))
  (env-loop env))
```

- This representation is simple, but inefficient, since the evaluator may have to search through many frames to find a binding. This approach is called *deep binding*.

4.11-13

## 4.1.4: Running the Evaluator as a Program

- We can run our evaluator as a program: Lisp within Lisp.
- The evaluator ultimately reduces expressions to applications of primitive procedures, so we need the evaluator to map these to the underlying Lisp's primitive procedures.
- We set up a global environment mapping primitive procedures, `true`, and `false`:

```
(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    (define-variable! 'true true initial-env)
    (define-variable! 'false false initial-env)
    initial-env))
(define the-global-environment (setup-environment))
```

- We define a list of primitive procedures `car`, `cdr`, `cons`, and `null?`:

```
(define (primitive-procedure? proc) (tagged-list? proc
'primitive))
(define (primitive-implementation proc) (cadr proc))

(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        «more-primitives»))
(define (primitive-procedure-names)
  (map car primitive-procedures))
(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))
```

- Here, `apply-in-underlying-scheme` refers to the built-in `apply`, not the one we defined:

```
(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme (primitive-implementation proc)
args))
```

- Finally, we make a simple *driver loop*, or REPL:

```
(define input-prompt ";;; M-Eval input:")
(define output-prompt ";;; M-Eval value:")
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))
(define (announce-output string)
  (newline) (display string) (newline))
(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '«procedure-env»))
      (display object)))
```

4.14

## 4.1.5: Data as Programs

- A program can be viewed as a description of an abstract machine.
- The evaluator is a machine that emulates another machine given its description. In other words, the evaluator is a *universal machine*.

- Deep idea: any evaluator can emulate any other. This gets to the heart of *computability*.
- Just as the evaluator can emulate any Lisp-described machine, a *universal Turing machine* can emulate any other Turing machine.
- The existence of a universal machine is a deep and wonderful property of computation.
- The user's programs are the evaluator's data. Lisp takes advantage of this and provides a primitive `eval` procedure for evaluating data as programs.

> We can regard the evaluator as a very special machine that takes as input a description of a machine. Given this input, the evaluator configures itself to emulate the machine described. …
>
> From this perspective, our evaluator is seen to be a *universal machine*. It mimics other machines when these are described as Lisp programs. This is striking. Try to imagine an analogous evaluator for electrical circuits. This would be a circuit that takes as input a signal encoding the plans for some other circuit, such as a filter. Given this input, the circuit evaluator would then behave like a filter with the same description. Such a universal electrical circuit is almost unimaginably complex. It is remarkable that the program evaluator is a rather simple program. [@4.1.5]
>
> Some people find it counterintuitive that an evaluator, which is implemented by a relatively simple procedure, can emulate programs that are more complex than the evaluator itself. The existence of a universal evaluator machine is a deep and wonderful property of computation. *Recursion theory*, a branch of mathematical logic, is concerned with logical limits of computation. Douglas Hofstadter's beautiful book *Gödel, Escher, Bach* (1979) explores some of these ideas. [@4.1.fn20]

4.15

## 4.1.6: Internal Definitions

- Global definitions have *sequential scoping*: they are defined one at a time.
- Internal definitions should have *simultaneous scoping*, as if defined all at once.
- The Scheme standard requires internal definitions to come first in the body and not use each other during evaluation. Although this restriction makes sequential and simultaneous scoping equivalent, simultaneous scoping makes compiler optimization easier.
- To achieve simultaneous scoping, we "scan out" internal definitions:

```
(lambda «vars»
  (define u «e1»)
  (define v «e2»)
  «e3»)
```

- Transforming them into a `let` with assignments:

```
(lambda «vars»
  (let ((u '*unassigned*)
        (v '*unassigned*))
```

```
    (set! u «e1»)
    (set! v «e2») «e3»))
```

- Here, `'*unassigned*` is a special symbol causing an error upon variable lookup.

4.16-21

## 4.1.7: Separating Syntactic Analysis from Execution

- Our evaluator is inefficient because it interleaves syntactic analysis with execution.
- For example, given a recursive procedure:

```
(define (factorial n)
  (if (= n 1) 1 (* (factorial (- n 1)) n)))
```

- When evaluating `(factorial 4)`, on all four recursive calls the evaluator must determine anew that the body is an `if` expression by reaching the `if?` test.
- We can arrange the evaluator to analyze syntax only once by splitting `eval` into two parts:
  - `(analyze exp)` performs syntactic analysis and returns an *execution procedure*.
  - `((analyze exp) env)` completes the evaluation.
- `analyze` is similar to the original `eval`, except it only performs analysis, not full evaluation:

```
(define (analyze exp)
  (cond ((self-evaluating? exp) (analyze-self-evaluating exp))
        ((quoted? exp) (analyze-quoted exp))
        ((variable? exp) (analyze-variable exp))
        ((assignment? exp) (analyze-assignment exp))
        ((definition? exp) (analyze-definition exp))
        ((if? exp) (analyze-if exp))
        ((lambda? exp) (analyze-lambda exp))
        ((begin? exp) (analyze-sequence (begin-actions exp)))
        ((cond? exp) (analyze (cond->if exp)))
        ((application? exp) (analyze-application exp))
        (else (error "Unknown expression type: ANALYZE" exp))))
```

- Here is one of the helper procedures, `analyze-lambda`. It provides a major gain in efficiency because we only analyze the lambda body once, no matter how many times the procedure is called.

```
(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env) (make-procedure vars bproc env))))
```

4.22-24

## 4.2: Variations on a Scheme – Lazy Evaluation

- We can experiment with different language design just by modifying the evaluator.
- This is often how new languages are invented. It’s easy to iterate on a high level evaluator, and it also allows stealing features from the underlying language.

### 4.2.1: Normal Order and Applicative Order

- In , we noted that Scheme is an *applicative-order* language.
- *Normal-order* languages use *lazy evaluation* to delay evaluation as long as possible.
- Consider this procedure:

```
(define (try a b) (if = a 0) 1 b)
```

- In Scheme, `(try 0 (/ 1 0))` causes a division-by-zero error. With lazy evaluation, it does not because the value of `b` is never needed.
- In a lazy language, we can implement `if` as an ordinary procedure.

4.25-26

### 4.2.2: An Interpreter with Lazy Evaluation

- In this section, we will modify the interpreter to support lazy evaluation.
- The lazy evaluator *delays* certain arguments, transforming them into *thunks*.
- The expression in a thunk does not get evaluated until the thunk is *forced*.
- A thunk gets forced when its value is needed:
    - when passed to a primitive procedure;
    - when it is the predicate of conditional;
    - when it is an operator about to be applied as a procedure.
- This is similar to streams, but uniform and automatic throughout the language.
- For efficiency, we’ll make our interpreter *memoize* thunks.
    - This is called *call-by-need*, as opposed to non-memoized *call-by-name*.
    - It raises subtle and confusing issues in the presence of assignments.

> The word *thunk* was invented by an informal working group that was discussing the implementation of call-by-name in Algol 60. They observed that most of the analysis of (“thinking about”) the expression could be done at compile time; thus, at run time, the expression would already have been “thunk” about. [@4.2.fn34]

#### Modifying the evaluator

- The main change required is the procedure application logic in `eval` and `apply`.
- The `application?` clause of `eval` becomes:

```
((application? exp)
 (apply (actual-value (operator exp) env)
        (operands exp)
        env))
```

- Whenever we need the actual value of an expression, we force in addition to evaluating:

```
(define (actual-value exp env) (force-it (eval exp env)))
```

- We change `apply` to take `env`, and use `list-of-arg-values` and `list-of-delayed-args`:

```
(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values arguments env)))  ; changed
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env) ; changed
           (procedure-environment procedure))))
        (else (error "Unknown procedure type: APPLY" procedure))))
```

- We also need to change `eval-if` to use `actual-value` on the predicate:

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

**Representing thunks**

- To force a thunk, we evaluate it in its environment. We use `actual-value` instead of `eval` so that it recursively forces if the result is another thunk:

```
(define (force-it obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj) (thunk-env obj))
      obj))
```

- We can represent thunks simply by a list containing the expression and environment:

```
(define (delay-it exp env) (list 'thunk exp env))
(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))
```

- To memoize thunks, `force-it` becomes a bit more complicated.

4.27-31

### 4.2.3: Streams as Lazy Lists

- Before introducing lazy evaluation, we implemented streams as delayed lists.
- We can instead formulate streams as *lazy* lists. There are two advantages:
  1. No need for special forms `delay` and `cons-stream`.
  2. No need for separate list and stream operations.
- All we need to do is make `cons` lazy, either by introducing non-strict primitives or by defining `cons`, `car`, and `cdr` as compound procedures.
- Now we can write code without distinguishing normal lists from infinite ones:

```
(define ones (cons 1 ones))
(define (scale-list items factor) (map (lambda (x) (* x factor))
items))
```

- These lazy lists are even lazier than our original streams, since the `car` is delayed too.
- This also eliminates the problems we had earlier around having to explicitly delay and force some arguments when computing integrals.

4.32-34

## 4.3: Variations on a Scheme – Nondeterministic Computing

- *Nondeterministic computing*

### 4.3.1: `Amb` and Search

#### Driver loop

4.35-37

### 4.3.2: Examples of Nondeterministic Programs

#### Logic puzzles

4.38-44

#### Parsing natural language

4.45-49

### 4.3.3: Implementing the `Amb` Evaluator

#### Execution procedures and continuations

#### Structure of the evaluator

#### Simple expressions

#### Conditionals and sequences

#### Definitions and assignments

#### Procedure applications

#### Evaluating `amb` expressions

#### Driver loop

4.50-54

## 4.4: Logic Programming

### 4.4.1: Deductive Information Retrieval

### 4.4.2: How the Query System Works

### 4.4.3: Is Logic Programming Mathematical Logic?

### 4.4.4: Implementing the Query System

# 5: Computing with Register Machines

## 5.1: Designing Register Machines

5.1

### 5.1.1: A Language for Describing Register Machines

5.2

#### Actions

### 5.1.2: Abstraction in Machine Design

5.3

### 5.1.3: Subroutines

### 5.1.4: Using a Stack to Implement Recursion

#### A double recursion

5.4-6

### 5.1.5: Instruction Summary

## 5.2: A Register-Machine Simulator

5.7

### 5.2.1: The Machine Model

#### Registers

#### The stack

#### The basic machine

### 5.2.2: The Assembler

5.8

### 5.2.3: Generating Execution Procedures for Instructions

#### `Assign` instructions

#### `Test`, `branch`, and `goto` instructions

#### Other instructions

#### Execution procedures for subexpressions

5.9-13

### 5.2.4: Monitoring Machine Performance

5.14-19

## 5.3: Storage Allocation and Garbage Collection

### 5.3.1: Memory as Vectors

#### Representing Lisp data

#### Implementing the primitive list operations

#### Implementing stacks

5.20-22

### 5.3.2: Maintaining the Illusion of Infinite Memory

#### Implementation of a stop-and-copy garbage collector

## 5.4: The Explicit-Control Evaluator

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

#### Evaluating simple expressions

#### Evaluating procedure applications

#### Procedure application

### 5.4.2: Sequence Evaluation and Tail Recursion

#### Tail recursion

### 5.4.3: Conditionals, Assignments, and Definitions

#### Assignments and definitions

5.23-25

### 5.4.4: Running the Evaluator

#### Monitoring the performance of the evaluator

5.26-30

## 5.5: Compilation

### 5.5.1: Structure of the Compiler

#### Targets and linkages

#### Instruction sequences and stack usage

5.31-32

### 5.5.2: Compiling Expressions

#### Compiling linkage code

#### Compiling simple expressions

#### Compiling conditional expressions

#### Compiling sequences

#### Compiling `lambda` expressions

### 5.5.3: Compiling Combinations

#### Applying procedures

#### Applying compiled procedures

### 5.5.4: Combining Instruction Sequences

### 5.5.5: An Example of Compiled Code

5.33-38

### 5.5.6: Lexical Addressing

5.39-44

### 5.5.7: Interfacing Compiled Code to the Evaluator