CS 61A Lecture Notes Week 1

Topic: Functional programming

Reading: Abelson & Sussman, Section 1.1 (pages 1–31)

Welcome to CS 61A, the world's best computer science course, because we use the world's best CS book as the textbook. The only thing wrong with this course is that all the rest of the CS courses for the rest of your life will seem a little disappointing (and repetitive).

Course overview comes next lecture; now we're going to jump right in so you can get started exploring on your own in the lab.

In 61A we program in Scheme, which is an *interactive* language. That means that instead of writing a great big program and then cranking it through all at once, you can type in a single expression and find out its value. For example:

```
3                            self-evaluating
(+ 2 3)                      function notation
(sqrt 16)                    names don't have to be punctuation
(+ (* 3 4) 5)                composition of functions


+                            functions are things in themselves
'+                           quoting
'hello                       can quote any word
'(+ 2 3)                     can quote any expression
'(good morning)              even non-expression sentences


(first 274)                  functions don't have to be arithmetic
(butfirst 274)               (abbreviation bf)
(first 'hello)               works for non-numbers
(first hello)                reminder about quoting
(first (bf 'hello))          composition of non-numeric functions
(+ (first 23) (last 45))     combining numeric and non-numeric


(define pi 3.14159)          special form
pi                           value of a symbol
'pi                          contrast with quoted symbol
(+ pi 7)                     symbols work in larger expressions
(* pi pi)


(define (square x)
  (* x x))                   defining a function
(square 5)                   invoking the function
(square (+ 2 3))             composition with defined functions
```

Terminology: the *formal parameter* is the name of the argument (`x`); the *actual argument expression* is the expression used in the invocation (`(+ 2 3)`); the *actual argument value* is the value of the argument in the invocation (5). The argument's name comes from the function's definition; the argument's value comes from the invocation.

Examples:

```
(define (plural wd)
  (word wd 's))
```

This simple `plural` works for lots of words (book, computer, elephant) but not for words that end in `y` (fly, spy). So we improve it:

```
;;;;;                        In file cs61a/lectures/1.1/plural.scm
(define (plural wd)
  (if (equal? (last wd) 'y)
      (word (bl wd) 'ies)
      (word wd 's)))
```

`If` is a special form that only evaluates one of the alternatives.

Pig Latin: Move initial consonants to the end of the word and append "ay"; SCHEME becomes EMESCHAY.

```
;;;;;                        In file cs61a/lectures/1.1/pigl.scm
(define (pigl wd)
  (if (pl-done? wd)
      (word wd 'ay)
      (pigl (word (bf wd) (first wd)))))

(define (pl-done? wd)
  (vowel? (first wd)))

(define (vowel? letter)
  (member? letter '(a e i o u)))
```

`Pigl` introduces *recursion*—a function that invokes itself. More about how this works next week.

Another example: Remember how to play Buzz? You go around the circle counting, but if your number is divisible by 7 or has a digit 7 you have to say "buzz" instead:

```
;;;;;                        In file cs61a/lectures/1.1/buzz.scm
(define (buzz n)
  (cond ((equal? (remainder n 7) 0) 'buzz)
        ((member? 7 n) 'buzz)
        (else n)))
```

This introduces the `cond` special form for multi-way choices.

`Cond` is the big exception to the rule about the meaning of parentheses; the clauses aren't invocations.

Course overview:

Computer science isn't about computers (that's electrical engineering) and it isn't primarily a science (we invent things more than we discover them).

CS is partly a form of engineering (concerned with building reliable, efficient mechanisms, but in software instead of metal) and partly an art form (using programming as a medium for creative expression).

Programming is really easy, as long as you're solving small problems. Any kid in junior high school can write programs in BASIC, and not just exercises, either; kids do quite interesting and useful things with computers. But BASIC doesn't scale up; once the problem is so complicated that you can't keep it all in your head at once, you need help, in the form of more powerful ways of thinking about programming. (But in this course we mostly use small examples, because we'd never get finished otherwise, so you have to imagine how you think each technique would work out in a larger case.)

We deal with three big programming styles/approaches/paradigms:

- Functional programming (2 months)
- Object-oriented programming (1 month)
- Logic programming (1 week)

The big idea of the course is *abstraction*: inventing languages that let us talk more nearly in a problem's own terms and less in terms of the computer's mechanisms or capabilities. There is a hierarchy of abstraction:

```
Application programs
High-level language (Scheme)
Low-level language (C)
Machine language
Architecture (registers, memory, arithmetic unit, etc)
circuit elements (gates)
transistors
solid-state physics
quantum mechanics
```

In 61C we look at lower levels; all are important but we want to start at the highest level to get you thinking right.

Style of work: Cooperative learning. No grading curve, so no need to compete. Homework is to learn from; only tests are to test you. Don't cheat; ask for help instead. (This is the *first* CS course; if you're tempted to cheat now, how are you planning to get through the harder ones?)

Functions.

• A function can have any number of arguments, including zero, but must have exactly one return value. (Suppose you want two? You combine them into one, e.g., in a sentence.) It's not a function unless you always get the same answer for the same arguments.

• Why does that matter? If each little computation is independent of the past history of the overall computation, then we can *reorder* the little computations. In particular, this helps cope with parallel processors.

• The function definition provides a formal parameter (a name), and the function invocation provides an actual argument (a value). These fit together like pieces of a jigsaw puzzle. *Don't write a "function" that only works for one particular argument value!*

• Instead of a sequence of events, we have composition of functions, like $f(g(x))$ in high school algebra. We can represent this visually with function machines and plumbing diagrams.

Recursion:

```
;;;;;                        In file cs61a/lectures/1.1/argue.scm
> (argue '(i like spinach))
(i hate spinach)
> (argue '(broccoli is awful))
(broccoli is great)

(define (argue s)
  (if (empty? s)
      '()
      (se (opposite (first s))
          (argue (bf s)))))

(define (opposite w)
  (cond ((equal? w 'like) 'hate)
        ((equal? w 'hate) 'like)
        ((equal? w 'wonderful) 'terrible)
        ((equal? w 'terrible) 'wonderful)
        ((equal? w 'great) 'awful)
        ((equal? w 'awful) 'great)
        ((equal? w 'terrific) 'yucky)
        ((equal? w 'yucky) 'terrific)
        (else w) ))
```

This computes a function (the `opposite` function) of each word in a sentence. It works by dividing the problem for the whole sentence into two subproblems: an easy subproblem for the first word of the sentence, and another subproblem for the rest of the sentence. This second subproblem is just like the original problem, but for a smaller sentence.

We can take `pigl` from last lecture and use it to translate a whole sentence into Pig Latin:

```
(define (pigl-sent s)
  (if (empty? s)
      '()
      (se (pigl (first s))
          (pigl-sent (bf s)))))
```

The structure of `pigl-sent` is a lot like that of `argue`. This common pattern is called *mapping* a function over a sentence.

Not all recursion follows this pattern. Each element of Pascal's triangle is the sum of the two numbers above it:

```
(define (pascal row col)
  (cond ((= col 0) 1)
        ((= col row) 1)
        (else (+ (pascal (- row 1) (- col 1))
                 (pascal (- row 1) col) ))))
```

Normal vs. applicative order.

To illustrate this point we use a modified Scheme evaluator that lets us show the process of applicative or normal order evaluation. We define functions using `def` instead of `define`. Then, we can evaluate expressions using `(applic (...))` for applicative order or `(normal (...))` for normal order. (Never mind how this modified evaluator itself works! Just take it on faith and concentrate on the results that it shows you.)

In the printed results, something like

```
(* 2 3) ==> 6
```

indicates the ultimate invocation of a primitive function. But

```
(f 5 9) ---->
(+ (g 5) 9)
```

indicates the substitution of actual arguments into the body of a function defined with `def`. (Of course, whether actual argument values or actual argument expressions are substituted depends on whether you used `applic` or `normal`, respectively.)

```
> (load "lectures/1.1/order.scm")
> (def (f a b) (+ (g a) b))     ; define a function
f
> (def (g x) (* 3 x))           ; another one
g
> (applic (f (+ 2 3) (- 15 6))) ; show applicative-order evaluation

(f (+ 2 3) (- 15 6))
   (+ 2 3) ==> 5
   (- 15 6) ==> 9
(f 5 9) ---->
(+ (g 5) 9)
   (g 5) ---->
   (* 3 5) ==> 15
(+ 15 9) ==> 24
24
> (normal (f (+ 2 3) (- 15 6))) ; show normal-order evaluation

(f (+ 2 3) (- 15 6)) ---->
(+ (g (+ 2 3)) (- 15 6))
   (g (+ 2 3)) ---->
   (* 3 (+ 2 3))
      (+ 2 3) ==> 5
   (* 3 5) ==> 15
   (- 15 6) ==> 9
(+ 15 9) ==> 24                 ; Same result, different process.
24
```

(continued on next page)

```
> (def (zero x) (- x x))        ; This function should always return 0.
zero
> (applic (zero (random 10)))

(zero (random 10))
   (random 10) ==> 5
(zero 5) ---->
(- 5 5) ==> 0
0                              ; Applicative order does return 0.

> (normal (zero (random 10)))

(zero (random 10)) ---->
(- (random 10) (random 10))
   (random 10) ==> 4
   (random 10) ==> 8
(- 4 8) ==> -4
-4                             ; Normal order doesn't.
```

The rule is that if you're doing functional programming, you get the same answer regardless of order of evaluation. Why doesn't this hold for `(zero (random 10))`? Because it's not a function! Why not?

Efficiency: Try computing

```
(square (square (+ 2 3)))
```

in normal and applicative order. Applicative order is more efficient because it only adds 2 to 3 once, not four times. (But later in the semester we'll see that sometimes normal order is more efficient.)

Note that the reading for next week is section 1.3, skipping 1.2 for the time being.

CS 61A Lecture Notes Week 2

Topic: Higher-order procedures

Reading: Abelson & Sussman, Section 1.3

Note that we are skipping 1.2; we'll get to it later. Because of this, never mind for now the stuff about iterative versus recursive processes in 1.3 and in the exercises from that section.

We're all done teaching you the syntax of Scheme; from now on it's all big ideas!

This week's big idea is *function as object* (that is, being able to manipulate functions as data) as opposed to the more familiar view of function as process, in which there is a sharp distinction between program and data.

The usual metaphor for function as process is a recipe. In that metaphor, the recipe tells you what to do, but you can't eat the recipe; the food ingredients are the "real things" on which the recipe operates. But this week we take the position that a function is just as much a "real thing" as a number or text string is.

Compare the *derivative* in calculus: It's a function whose domain and range are functions, not numbers. The derivative function treats ordinary functions as things, not as processes. If an ordinary function is a meat grinder (put numbers in the top and turn the handle) then the derivative is a "metal grinder" (put meat-grinders in the top...).

• Using functions as arguments.

Arguments are used to generalize a pattern. For example, here is a pattern:

```
;;;;;                          In file cs61a/lectures/1.3/general.scm
(define pi 3.141592654)

(define (square-area r) (* r r))

(define (circle-area r) (* pi r r))

(define (sphere-area r) (* 4 pi r r))

(define (hexagon-area r) (* (sqrt 3) 1.5 r r))
```

In each of these procedures, we are taking the area of some geometric figure by multiplying some constant times the square of a linear dimension (radius or side). Each is a function of one argument, the linear dimension. We can generalize these four functions into a single function by adding an argument for the shape:

```
;;;;;                          In file cs61a/lectures/1.3/general.scm
(define (area shape r) (* shape r r))

(define square 1)
(define circle pi)
(define sphere (* 4 pi))
(define hexagon (* (sqrt 3) 1.5))
```

We define names for shapes; each name represents a constant number that is multiplied by the square of the radius.

In the example about areas, we are generalizing a pattern by using a variable *number* instead of a constant number. But we can also generalize a pattern in which it's a *function* that we want to be able to vary:

```
;;;;;                         In file cs61a/lectures/1.3/general.scm
(define (sumsquare a b)
  (if (> a b)
      0
      (+ (* a a) (sumsquare (+ a 1) b)) ))

(define (sumcube a b)
  (if (> a b)
      0
      (+ (* a a a) (sumcube (+ a 1) b)) ))
```

Each of these functions computes the sum of a series. For example, `(sumsquare 5 8)` computes $5^2 + 6^2 + 7^2 + 8^2$. The process of computing each individual term, and of adding the terms together, and of knowing where to stop, are the same whether we are adding squares of numbers or cubes of numbers. The only difference is in deciding which function of `a` to compute for each term. We can generalize this pattern by making *the function* be an additional argument, just as the shape number was an additional argument to the area function:

```
(define (sum fn a b)
  (if (> a b)
      0
      (+ (fn a) (sum fn (+ a 1) b)) ))
```

Here is one more example of generalizing a pattern involving functions:

```
;;;;;                         In file cs61a/lectures/1.3/keep.scm
(define (evens nums)
  (cond ((empty? nums) '())
        ((= (remainder (first nums) 2) 0)
         (se (first nums) (evens (bf nums))) )
        (else (evens (bf nums))) ))

(define (ewords sent)
  (cond ((empty? sent) '())
        ((member? 'e (first sent))
         (se (first sent) (ewords (bf sent))) )
        (else (ewords (bf sent))) ))

(define (pronouns sent)
  (cond ((empty? sent) '())
        ((member? (first sent) '(I me you he she it him her we us they them))
         (se (first sent) (pronouns (bf sent))) )
        (else (pronouns (bf sent))) ))
```

Each of these functions takes a sentence as its argument and returns a smaller sentence *keep*ing only some of the words in the original, according to a certain criterion: even numbers, words that contain the letter `e`, or pronouns. We can generalize by writing a `keep` function that takes a predicate function as an additional argument.

```
(define (keep pred sent)
  (cond ((empty? sent) '())
        ((pred (first sent)) (se (first sent) (keep pred (bf sent))) )
        (else (keep pred (bf sent))) ))
```

• Unnamed functions.

Suppose we want to compute

$$\sin^2 5 + \sin^2 6 + \sin^2 7 + \sin^2 8$$

We can use the generalized `sum` function this way:

```
> (define (sinsq x) (* (sin x) (sin x)))
> (sum sinsq 5 8)
2.408069916229755
```

But it seems a shame to have to define a named function `sinsq` that (let's say) we're only going to use this once. We'd like to be able to represent the function *itself* as the argument to `sum`, rather than the function's name. We can do this using `lambda`:

```
> (sum (lambda (x) (* (sin x) (sin x))) 5 8)
2.408069916229755
```

`Lambda` is a special form; the formal parameter list obviously isn't evaluated, but the body isn't evaluated *when we see the* `lambda`, either—only when we invoke the function can we evaluate its body.

• First-class data types.

A data type is considered *first-class* in a language if it can be

- the value of a variable (i.e., named)
- an argument to a function
- the return value from a function
- a member of an aggregate

In most languages, numbers are first-class; perhaps text strings (or individual text characters) are first-class; but usually functions are not first-class. In Scheme they are. So far we've seen the first two properties; we're about to look at the third. (We haven't really talked about aggregates yet, except for the special case of sentences, but we'll see in chapter 2 that functions can be elements of aggregates.) It's one of the design principles of Scheme that everything in the language should be first-class. Later, when we write a Scheme interpreter in Scheme, we'll see how convenient it is to be able to treat Scheme programs as data.

• Functions as return values.

```
(define (compose f g) (lambda (x) (f (g x))))
(define (twice f) (compose f f))
(define (make-adder n) (lambda (x) (+ x n)))
```

The derivative is a function whose domain and range are functions.

People who've programmed in Pascal might note that Pascal allows functions as arguments, but *not* functions as return values. That's because it makes the language harder to implement; you'll learn more about this in 164.

• Let.

We write a function that returns a sentence containing the two roots of the quadratic equation $ax^2+bx+c=0$ using the formula

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

(We assume, to simplify this presentation, that the equation has two real roots; a more serious program would check this.)

```
;;;;;                        In file cs61a/lectures/1.3/roots.scm
(define (roots a b c)
  (se (/ (+ (- b) (sqrt (- (* b b) (* 4 a c)))) (* 2 a))
      (/ (- (- b) (sqrt (- (* b b) (* 4 a c)))) (* 2 a)) ))
```

This works fine, but it's inefficient that we have to compute the square root twice. We'd like to avoid that by computing it once, giving it a name, and using that name twice in figuring out the two solutions. We know how to give something a name by using it as an argument to a function:

```
;;;;;                        In file cs61a/lectures/1.3/roots.scm
(define (roots a b c)
  (define (roots1 d)
    (se (/ (+ (- b) d) (* 2 a))
        (/ (- (- b) d) (* 2 a)) ))
  (roots1 (sqrt (- (* b b) (* 4 a c)))) )
```

`Roots1` is an internal helper function that takes the value of the square root in the formula as its argument `d`. `Roots` calls `roots1`, which constructs the sentence of two numbers.

This does the job, but it's awkward having to make up a name `roots1` for this function that we'll only use once. As in the `sum` example earlier, we can use `lambda` to make an unnamed function:

```
;;;;;                        In file cs61a/lectures/1.3/roots.scm
(define (roots a b c)
  ((lambda (d)
     (se (/ (+ (- b) d) (* 2 a))
         (/ (- (- b) d) (* 2 a)) ))
   (sqrt (- (* b b) (* 4 a c))) ))
```

This does exactly what we want. The trouble is, although it works fine for the computer, it's a little hard for human beings to read. The connection between the name `d` and the `sqrt` expression that provides its value isn't obvious from their positions here, and the order in which things are computed isn't the top-to-bottom order of the expression. Since this is something we often want to do, Scheme provides a more convenient notation for it:

```
;;;;;                        In file cs61a/lectures/1.3/roots.scm
(define (roots a b c)
  (let ((d (sqrt (- (* b b) (* 4 a c)))))
    (se (/ (+ (- b) d) (* 2 a))
        (/ (- (- b) d) (* 2 a)) )))
```

Now we have the name next to the value, and we have the value of `d` being computed above the place where it's used. But you should remember that `let` does not provide any new capabilities; it's merely an abbreviation for a `lambda` and an invocation of the unnamed function.

The unnamed function implied by the `let` can have more than one argument:

```
;;;;;                        In file cs61a/lectures/1.3/roots.scm
(define (roots a b c)
  (let ((d (sqrt (- (* b b) (* 4 a c))))
        (-b (- b))
        (2a (* 2 a)))
    (se (/ (+ -b d) 2a)
        (/ (- -b d) 2a) )))
```

Two cautions: (1) These are not long-term "assignment statements" such as you may remember from other languages. The association between names and values only holds while we compute the body of the `let`. (2) If you have more than one name-value pair, as in this last example, they are not computed in sequence! Later ones can't depend on earlier ones. They are all arguments to the same function; if you translate back to the underlying `lambda`-and-application form you'll understand this.

CS 61A Lecture Notes Week 3

Topic: Recursion and iteration

Reading: Abelson & Sussman, Section 1.2 through 1.2.4 (pages 31–72)

This week is about efficiency. Mostly in 61A we don't care about that; it becomes a focus of attention in 61B. In 61A we're happy if you can get a program working at all, except for this week, when we introduce ideas that will be more important to you later.

We want to know about the efficiency of algorithms, not of computer hardware. So instead of measuring runtime in microseconds or whatever, we ask about the number of times some primitive (fixed-time) operation is performed. Example:

```
;;;;;                           In file cs61a/lectures/1.2/growth.scm
(define (square x) (* x x))

(define (squares sent)
  (if (empty? sent)
      '()
      (se (square (first sent))
          (squares (bf sent)) )))
```

To estimate the efficiency of this algorithm, we can ask, "if the argument has N numbers in it, how many multiplications do we perform?" The answer is that we do one multiplication for each number in the argument, so we do N altogether. The amount of time needed should roughly double if the number of numbers doubles.

Another example:

```
;;;;;                           In file cs61a/lectures/1.2/growth.scm
(define (sort sent)
  (if (empty? sent)
      '()
      (insert (first sent)
              (sort (bf sent)) )))

(define (insert num sent)
  (cond ((empty? sent) (se num sent))
        ((< num (first sent)) (se num sent))
        (else (se (first sent) (insert num (bf sent)))) ))
```

Here we are sorting a bunch of numbers by comparing them against each other. If there are N numbers, how many comparisons do we do?

Well, if there are K numbers in the argument to `insert`, how many comparisons does it do? K of them. How many times do we call `insert`? N times. But it's a little tricky because each call to `insert` has a different length sentence. The range is from 0 to $N - 1$. So the total number of comparisons is actually

$$0 + 1 + 2 + \cdots + (N - 2) + (N - 1)$$

which turns out to be $\frac{1}{2}N(N - 1)$. For large N, this is roughly equal to $\frac{1}{2}N^2$. If the number of numbers doubles, the time required should quadruple.

That constant factor of $\frac{1}{2}$ isn't really very important, since we don't really know what we're halving—that is, we don't know exactly how long it takes to do one comparison. If we want a very precise measure of how many microseconds something will take, then we have to worry about the constant factors, but for an overall sense of the nature of the algorithm, what counts is the N^2 part. If we double the size of the input to a program, how does that affect the running time?

We use "big Theta" notation to express this sort of approximation. We say that the running time of the `sort` function is $\Theta(N^2)$ while the running time of the `squares` function is $\Theta(N)$. The formal definition is

$$f(x) = \Theta(g(x)) \Leftrightarrow \exists k, N \mid \forall x > N, |f(x)| \leq k \cdot |g(x)|$$

What does all this mean? Basically that one function is always less than another function (e.g., the time for your program to run is less than x^2) except that we don't care about constant factors (that's what the k means) and we don't care about small values of x (that's what the N means).

Why don't we care about small values of x? Because for small inputs, your program will be fast enough anyway. Let's say one program is 1000 times faster than another, but one takes a millisecond and the other takes a second. Big deal.

Why don't we care about constant factors? Because for large inputs, the constant factor will be drowned out by the order of growth—the exponent in the $\Theta(x^i)$ notation. Here is an example taken from the book *Programming Pearls* by Jon Bentley (Addison-Wesley, 1986). He ran two different programs to solve the same problem. One was a fine-tuned program running on a Cray supercomputer, but using an $\Theta(N^3)$ algorithm. The other algorithm was run on a Radio Shack microcomputer, so its constant factor was several million times bigger, but the algorithm was $\Theta(N)$. For small N the Cray was much faster, but for small N both computers solved the problem in less than a minute. When N was large enough for the problem to take a few minutes or longer, the Radio Shack computer's algorithm was faster.

```
;;;;;                           In file cs61a/lectures/1.2/bentley

              t1(N) = 3.0 N^3          t2(N) = 19,500,000 N

        N     CRAY-1 Fortran           TRS-80 Basic

       10     3.0 microsec             200 millisec
      100     3.0 millisec             2.0 sec
     1000     3.0 sec                  20 sec
    10000     49 min                   3.2 min
   100000     35 days                  32 min
  1000000     95 yrs                   5.4 hrs
```

Typically, the algorithms you run across can be grouped into four categories according to their order of growth in time required. The first category is *searching* for a particular value out of a collection of values, e.g., finding someone's telephone number. The most obvious algorithm (just look through all the values until you find the one you want) is $\Theta(N)$ time, but there are smarter algorithms that can work in $\Theta(\log N)$ time or even in $\Theta(1)$ (that is, constant) time. The second category is *sorting* a bunch of values into some standard order. (Many other problems that are not explicitly about sorting turn out to require similar approaches.) The obvious sorting algorithms are $\Theta(N^2)$ and the clever ones are $\Theta(N \log N)$. A third category includes relatively obscure problems such as matrix multiplication, requiring $\Theta(N^3)$ time. Then there is an enormous jump to the really hard problems that require $\Theta(2^N)$ or even $\Theta(N!)$ time; these problems are effectively not solvable for values of N greater than one or two dozen. (Inventing faster computers won't help; if the speed of your computer doubles, that just adds 1 to the largest problem size you can handle!) Trying to find faster algorithms for these *intractable* problems is a current hot research topic in computer science.

• Iterative processes

So far we've been talking about time efficiency, but there is also memory (space) efficiency. This has gotten less important as memory has gotten cheaper, but it's still somewhat relevant because using a lot of memory increases swapping (not everything fits at once) and so indirectly takes time.

The immediate issue for today is the difference between a *linear recursive process* and an *iterative process.*

```
;;;;;                         In file cs61a/lectures/1.2/count.scm
(define (count sent)
  (if (empty? sent)
      0
      (+ 1 (count (bf sent))) ))
```

This function counts the number of words in a sentence. It takes $\Theta(N)$ time. It also requires $\Theta(N)$ space, not counting the space for the sentence itself, because Scheme has to keep track of N pending computations during the processing:

```
(count '(i want to hold your hand))
(+ 1 (count '(want to hold your hand)))
(+ 1 (+ 1 (count '(to hold your hand))))
(+ 1 (+ 1 (+ 1 (count '(hold your hand)))))
(+ 1 (+ 1 (+ 1 (+ 1 (count '(your hand))))))
(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (count '(hand)))))))
(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (count '()))))))))
(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 0))))))
(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 1)))))
(+ 1 (+ 1 (+ 1 (+ 1 2))))
(+ 1 (+ 1 (+ 1 3)))
(+ 1 (+ 1 4))
(+ 1 5)
6
```

When we get halfway through this chart and compute `(count '())`, we aren't finished with the entire problem. We have to remember to add 1 to the result six times. Each of those remembered tasks requires some space in memory until it's finished.

Here is a more complicated program that does the same thing differently:

```
;;;;;                         In file cs61a/lectures/1.2/count.scm
(define (count sent)
  (define (iter wds result)
    (if (empty? wds)
        result
        (iter (bf wds) (+ result 1)) ))
  (iter sent 0) )
```

This time, we don't have to remember uncompleted tasks; when we reach the base case of the recursion, we have the answer to the entire problem:

```
(count '(i want to hold your hand))
(iter '(i want to hold your hand) 0)
(iter '(want to hold your hand) 1)
(iter '(to hold your hand) 2)
(iter '(hold your hand) 3)
(iter '(your hand) 4)
(iter '(hand) 5)
(iter '() 6)
6
```

When a process has this structure, Scheme does not need extra memory to remember all the unfinished tasks during the computation.

This is really not a big deal. For the purposes of this course, you should generally use the simpler linear-recursive structure and not try for the more complicated iterative structure; the efficiency saving is not worth the increased complexity. The reason Abelson and Sussman make a fuss about it is that in other programming languages any program that is recursive in *form* (i.e., in which a function invokes itself) will take (at least) linear space even if it could theoretically be done iteratively. These other languages have special iterative syntax (`for`, `while`, and so on) to avoid recursion. In Scheme you can use the function-calling mechanism and still achieve an iterative process.

• More is less: non-obvious efficiency improvements.

The nth row of Pascal's triangle contains the constant coefficients of the terms of $(a + b)^n$. Each number in Pascal's triangle is the sum of the two numbers above it. So we can write a function to compute these numbers:

```
;;;;;                        In file cs61a/lectures/1.2/pascal.scm
(define (pascal row col)
  (cond ((= col 0) 1)
        ((= col row) 1)
        (else (+ (pascal (- row 1) (- col 1))
                 (pascal (- row 1) col) ))))
```

This program is very simple, but it takes $\Theta(2^n)$ time! [Try some examples. Row 18 is already getting slow.]

Instead we can write a more complicated program that, on the surface, does a lot more work because it computes an *entire row* at a time instead of just the number we need:

```
;;;;;                        In file cs61a/lectures/1.2/pascal.scm
(define (new-pascal row col)
  (nth col (pascal-row row)) )

(define (pascal-row row-num)
  (define (iter in out)
    (if (empty? (bf in))
        out
        (iter (bf in) (se (+ (first in) (first (bf in))) out)) ))
  (define (next-row old-row num)
    (if (= num 0)
        old-row
        (next-row (se 1 (iter old-row '(1))) (- num 1)) ))
  (next-row '(1) row-num) )
```

This was harder to write, and seems to work harder, but it's incredibly faster because it's $\Theta(N^2)$.

The reason is that the original version computed lots of entries repeatedly. The new version computes a few unnecessary ones, but it only computes each entry once.

Moral: When it really matters, think hard about your algorithm instead of trying to fine-tune a few microseconds off the obvious algorithm.

Note: Programming project 1 is assigned this week.

CS 61A Lecture Notes Week 4

Topic: Programming methodology

Reading: none

Did you have trouble writing or debugging the programming project? This week's topic is a collection of ideas to help with the programming process.

Invariants

Exercise 1.16, in the homework due this week, asks you to implement the `fast-expt` algorithm on page 45 of SICP, but avoid using the recursive call as an argument to something else. Here is the code in the book:

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

The hint tells us to use an extra variable, `a`, which will contain part of the result. "Define the state transformation in such a way that the product ab^n is unchanged from state to state." Many students find this problem difficult, partly because they don't understand this talk of state transformations.

Here's a solution:

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

What "unchanged from state to state" means is that we're supposed to keep the quantity ab^n constant in all the invocations of `iter` for any specific problem. If n is even, we square b and divide n by 2, so we have

$$a \cdot (b^2)^{n/2}$$

which is indeed equal to ab^n. If n is odd, we have

$$ab \cdot b^{n-1}$$

which is also equal to ab^n. So each invocation does keep the overall value constant. Since $b^0 = 1$, when we get to $n = 0$ the value of a must be the original value of b^n.

The program might be easier for a reader to understand if the invariant (the term "loop invariant" is often used) is explicitly documented:

```
(define (fast-expt b n)
  (define (iter a b n)
    ;;;;; Invariant: a * b^n
    (cond ((= n 0) a)
          ((even? n) (iter a (square b) (/ n 2)))
          (else (iter (* a b) b (- n 1)))))
  (iter 1 b n))
```

We could improve the readability even further by changing the names of variables so that the same name is not used both in `fast-expt` and in `iter`:

```
(define (fast-expt b n)
  (define (iter a bb nn)
    ;;;;; Invariant:          a * bb^nn = b^n
    ;;;;; Starting condition: a=1, bb=b, nn=n
    ;;;;; Ending condition:   nn=0, therefore bb^nn=1, therefore a=b^n
    (cond ((= nn 0) a)
          ((even? nn) (iter a (square bb) (/ nn 2)))
          (else (iter (* a bb) bb (- nn 1)))))
  (iter 1 b n))
```

This lets us say unambiguously that at the end bb^{nn} is 1, while b^n, the answer we wanted in the first place, is not 1.

This example illustrates two ideas about programming methodology: invariants and documentation. The first is to help you write the program in the first place, while the second is to help you (and other people) understand how your program works later.

Assertions

Loop invariants are a useful mechanism only in a program that loops — that is, an iterative process. But they are a special case of a more general idea: making *assertions* about how you think your program behaves.

For example, in the `iter` subprocedure of `fast-expt`, how do we know that `nn` will ever be zero? (If not, the program will never actually return a value.) We know by the following chain of reasoning:

1. `n` is a positive integer. (This is a requirement for the procedure to work; it doesn't compute negative or fractional powers.)

2. The initial value of `nn` is `n`, therefore a positive integer.

3. If `nn` is an even positive integer, dividing it by 2 gives a strictly smaller, but still positive, integer.

4. If `nn` is a positive integer, subtracting 1 from it gives a strictly smaller nonnegative integer (perhaps zero).

5. If each iteration gives a strictly smaller, nonnegative integer value of `nn`, then within `n` iterations we have to get down to zero! (There are only finitely many nonnegative integers less than `n`.)

We could put a shorter version of this argument into the program:

```
(define (fast-expt b n)
  ;;;;; Requirement:       n is a positive integer.
  (define (iter a bb nn)
    ;;;;; Invariant:          a * bb^nn = b^n
    ;;;;; Starting condition: a=1, bb=b, nn=n
    ;;;;; Ending condition:   nn=0, therefore bb^nn=1, therefore a=b^n
    ;;
    ;;;;; Assertion: nn is always a nonnegative integer.
    ;;;;; Assertion: The end condition nn=0 must happen within n steps.
    (cond ((= nn 0) a)
          ((even? nn) (iter a (square bb) (/ nn 2)))
          (else (iter (* a bb) bb (- nn 1)))))
  (iter 1 b n))
```

Documentation

The last version of `fast-expt` has six lines of actual Scheme code and six lines of comment. This is probably overkill for such a simple procedure! The trouble is that realistic examples are too long to present in a lecture.

The secret to good documentation is to get a lot of mileage out of each comment by focusing on the things that are applicable to more than one procedure. For example, in the twenty-one project a central concept is a particular kind of procedure called a "strategy." So that term can be defined in a comment at the beginning of the program, along with other names for special kinds of data:

```
;; Data types:
;;
;; RANK                One of the following words: A 2 3 4 5 6 7 8 9 10 J Q K
;;                  representing Ace, Two, Three, ... Jack, Queen, King.
;;
;; SUIT                One of the following words: H S D C
;;                  representing Hearts, Spades, Diamonds, Clubs.
;;
;; CARD                A word consisting of a RANK followed by a SUIT
;;                  e.g., 10H for the Ten of Hearts.
;;
;; HAND                A sentence of CARDs
;;
;; STRATEGY        A procedure that takes two arguments:
;;                    a HAND (the player's hand)
;;                    a CARD (the dealer's visible card)
;;                and returns a Boolean:
;;                    #T if the player should take another card
;;                    #F if the player should not take another card
```

Once these definitions are in your program, you can use the data type names as part of your formal parameters, and then you shouldn't need comments about the domain of each procedure. For example, you can write

```
(define (stop-at-17 my-hand dealer-up-card)
  (< (best-total my-hand) 17))
```

and it's clear from the name `my-hand` that the first argument to this procedure must be a hand. Since there is no name comparable to a formal parameter for the procedure's return value, you might want a comment for that:

```
(define (stop-at-17 my-hand dealer-up-card)
  ;; returns a Boolean
  (< (best-total my-hand) 17))
```

but even better is to take advantage of the fact that you've explained both the types and the meanings of the arguments and return value of a strategy, so you can say

```
(define (stop-at-17 my-hand dealer-up-card)
  ;; this is a strategy procedure
  (< (best-total my-hand) 17))
```

But even if you don't put any comments at all in `stop-at-17`, the fact that it takes a hand and a card as its arguments would provide anyone reading your program with a pretty good clue that it's probably a strategy. Another approach would be to name the procedure `stop-at-17-strategy`, except that in this project you were told what name to use.

If you took a programming course in high school, there's a good chance that your teacher insisted that every procedure must have comments, in a particular format, explaining its domain and range, and perhaps other things. I'm a subversive! I think commenting every procedure is way too much. For one thing, since only so many lines fit on your monitor screen at once, lots of comments mean you can only look at a small slice of your program at once. This focuses your attention on small details at the expense of the larger program structure.

One reason for comment-every-procedure requirements is that people tend to write too few procedures, each of which is too long and complicated. (This is especially true when programming in sequential languages, as opposed to functional ones like Scheme.) Each procedure should carry out only one task; each procedure should fit on a screen; you should use names for procedures and variables that make it obvious what they are, without comments. In my own programming I tend to go to the opposite extreme: Whenever I find myself *tempted* to put a comment in a procedure, I take this as a sign that the procedure is too ugly, and that I should rewrite the code instead of commenting it.

It's especially harmful to put a zillion detailed comments in a procedure and then later modify the procedure in a way that makes the comments untrue! For example, suppose we modify our commented `fast-expt` procedure to accept negative exponents:

```
(define (fast-expt b n)
  ;;;;; Requirement:      n is a positive integer.
  (define (iter a bb nn)
    ;;;;; Invariant:          a * bb^nn = b^n
    ;;;;; Starting condition: a=1, bb=b, nn=n
    ;;;;; Ending condition:   nn=0, therefore bb^nn=1, therefore a=b^n
    ;;
    ;;;;; Assertion: nn is always a nonnegative integer.
    ;;;;; Assertion: The end condition nn=0 must happen within n steps.
    (cond ((= nn 0) a)
          ((even? nn) (iter a (square bb) (/ nn 2)))
          (else (iter (* a bb) bb (- nn 1)))))
  (IF (< N 0)
      (/ 1 (ITER 1 B (- N)))
      (iter 1 b n)))
```

What's new is the capitalized `if` expression at the end. But the first comment line, eleven lines earlier, is now a lie. An untrue comment is much worse than no documentation at all. In the abstract this may sound like an unlikely problem, but in fact it's all too common in large real-world programs to find untrue comments. Often the program is modified by someone other than the original author, and the second programmer may not even notice that there's a comment relevant to the change.

Debugging: Domain and Range

The absolutely most important thing you can do in debugging is to keep straight the domain and range of each procedure you write. For example, in the twenty-one project, one question asked you to write `stop-at`, a procedure that takes a number as argument and returns a strategy. Compare the correct solution with a typical wrong solution:

```
(define (stop-at n)                              ;; correct
  (lambda (my-hand dealer-up-card)
    (< (best-count my-hand) n)))

(define (stop-at n my-hand dealer-up-card)      ;; wrong
  (< (best-count my-hand) n))
```

The thinking that leads to this incorrect solution goes something like this: "I'm supposed to make a strategy, so it has to have a hand and a card as arguments. Also, the problem statement says it has to have a number as an argument. So that makes three altogether." The trouble here is the word "it." In the correct solution there are *two* procedures: `stop-at` itself, and the unnamed strategy that `stop-at` returns. The domain of `stop-at` is numbers—it takes a single number as its argument, and nothing else. The *range* of `stop-at` is *strategies;* in other words, it returns a procedure. It's the returned procedure, not `stop-at`, that requires a hand and a card as arguments. The incorrect version takes too many arguments, and returns a Boolean rather than a procedure.

Thinking about the range of your procedure can avoid mistakes in the base case of a recursion. For example, many students get so much in the habit of seeing procedures like this one:

```
(define (plurals sent)
  (if (empty? sent)
      '()
      (se (plural (first sent))
          (plurals (butfirst sent)))))
```

that they automatically return the empty sentence in the base case without thinking about the particular procedure they're writing, like this:

```
(define (count sent)               ;; wrong!
  (if (empty? sent)
      '()
      (+ 1 (count (butfirst sent)))))
```

The range of `count` is integers, not sentences! So even in the base case, `count` has to return an integer.

Debugging: Read the error message!

The error messages from STk are long, so many people don't bother reading them, other than to notice that *something* went wrong. Here's what happens when we try the incorrect `count` above:

```
STk> (count '(good day sunshine))
*** Error:
    +: not a number
Current eval stack:
__________________
  0    (apply fn (map maybe-num args))
  1    (+ 1 (count (butfirst sent)))
  2    (+ 1 (count (butfirst sent)))
STk>
```

The important part of this message comes right after the `Error` line:

```
    +: not a number
```

This tells you that you're calling the `+` procedure with an argument that isn't a number. Here's the only place where `count` calls `+`:

```
      (+ 1 (count (butfirst sent)))))
```

Since 1 is a number, the problem is clearly in the recursive call. It must be that `count` is returning a non-numeric value. In the recursive case, `count` returns the result of a call to `+`, which has to be a number. What about the base case? Oops, it's returning a sentence, not a number.

Compare this with the error message for a different kind of mistake:

```
(define (count sent)                ;; still wrong!
  (if (empty? sent)
      0
      (+ 1 (count (butfirst snt)))))

STk> (count '(good day sunshine))
*** Error:
    unbound variable: snt
Current eval stack:
-----------------
  0    (butfirst snt)
  1    (count (butfirst snt))
  2    (+ 1 (count (butfirst snt)))
STk>
```

This time the error message was

```
    unbound variable: snt
```

This tells you that somewhere in your program you used the name `snt` (because of a typo, it turns out; you meant to say `sent`). It's easy enough to find the `snt` by eye, or with the Emacs search command, or you can use the rest of the error message to see the context in which the word `snt` appeared.

Debugging: Tracing

Let's go back to the first incorrect version of `count`, the one with the bad base case. Suppose reading the error message wasn't enough to tell you where the error is. You could try this:

```
STk> (trace count)
okay
STk> (count '(good day sunshine))
.. -> count with sent = (good day sunshine)
.... -> count with sent = (day sunshine)
...... -> count with sent = (sunshine)
........ -> count with sent = ()
........ <- count returns ()
*** Error:
    +: not a number
Current eval stack:
-----------------
  0    (apply fn (map maybe-num args))
  1 (let ((res (apply value l))) (format *err-port* " A <-  A returns  S\n"
(indent) symbol res) res)
...
STk>
```

The `trace` procedure takes a procedure as its argument, and modifies the procedure so that it prints messages every time it's called, and every time it returns a value. This trace tells us that our call to `count` successfully made recursive calls with smaller and smaller sentences until finally `count` calls itself with the empty sentence as argument. This call returns the empty sentence, and that's when the error happens — the value returned in the base case causes an error in the call one level up.

Debugging: The Replacement Modeler

Another tool that can help you understand a broken program is the Replacement Modeler. Try this, after defining `count` with the bug as above:

```
STk> (load "~cs61a/lib/modeler.scm")
okay
STK> (model (count '(good day sunshine)))
```

This will bring up a new X window, in which the expression given as argument to the `model` special form will appear. Type the RETURN key repeatedly. Each time, the expression is replaced with another expression that has the same value, but with some procedure calls replaced either by the procedure body with substitution, for defined procedures, or by the possibly quoted returned value, for primitive procedures. You'll see this:

```
(count '(good day sunshine))

(if (empty? '(good day sunshine))
    '()
    (+ 1 (count (butfirst '(good day sunshine)))))

(if #f '() (+ 1 (count (butfirst '(good day sunshine)))))

(+ 1 (count (butfirst '(good day sunshine))))

(+ 1 (count '(day sunshine)))

(+ 1
   (if (empty? '(day sunshine)) '()
       (+ 1 (count (butfirst '(day sunshine))))))

(+ 1 (if #f '() (+ 1 (count (butfirst '(day sunshine))))))

(+ 1 (+ 1 (count (butfirst '(day sunshine)))))

(+ 1 (+ 1 (count '(sunshine))))

(+ 1
   (+ 1 (if (empty? '(sunshine)) '()
            (+ 1 (count (butfirst '(sunshine)))))))

(+ 1 (+ 1 (if #f '() (+ 1 (count (butfirst '(sunshine)))))))

(+ 1 (+ 1 (+ 1 (count (butfirst '(sunshine))))))

(+ 1 (+ 1 (+ 1 (count '()))))

(+ 1 (+ 1 (+ 1 (if (empty? '()) '() (+ 1 (count (butfirst '()))))))))

(+ 1 (+ 1 (+ 1 (if #t '() (+ 1 (count (butfirst '()))))))))

(+ 1 (+ 1 (+ 1 '())))
```

At this point you'll get the STk error message. Seeing the expression

```
(+ 1 '())
```

should make it quite clear what's causing the problem!

You can, by the way, control the order in which the modeler handles subexpressions by clicking inside a subexpression, which will highlight the nearest surrounding expression, and then using the RETURN key (on the main keyboard) or the ENTER key (on the numeric keypad) to process that expression. The RETURN key does one step of processing; the ENTER key replaces the highlighted expression with its simplest form.

Note: The first midterm is next week.

CS 61A   Lecture Notes   Week 5

Topic: Data abstraction

Reading: Abelson & Sussman, Sections 2.1 and 2.2.1 (pages 79–106)

Midterm Wednesday, 7–9pm.

• Big ideas: data abstraction, abstraction barrier.

If we are dealing with some particular type of data, we want to talk about it in terms of its *meaning*, not in terms of how it happens to be represented in the computer.

Example: Here is a function that computes the total point score of a hand of playing cards. (This simplified function ignores the problem of cards whose rank-name isn't a number.)

```
;;;;;                          In file cs61a/lectures/2.1/total.scm
(define (total hand)
  (if (empty? hand)
      0
      (+ (butlast (last hand))
         (total (butlast hand)) )))


> (total '(3h 10c 4d))
17
```

This function calls `butlast` in two places. What do those two invocations mean? Compare it with a modified version:

```
;;;;;                          In file cs61a/lectures/2.1/total.scm
(define (total hand)
  (if (empty? hand)
      0
      (+ (rank (one-card hand))
         (total (remaining-cards hand)) )))

(define rank butlast)
(define suit last)

(define one-card last)
(define remaining-cards butlast)
```

This is more work to type in, but the result is much more readable. If for some reason we wanted to modify the program to add up the cards left to right instead of right to left, we'd have trouble editing the original version because we wouldn't know which `butlast` to change. In the new version it's easy to keep track of which function does what.

The auxiliary functions like `rank` are called *selectors* because they select one component of a multi-part datum.

Actually we're *violating* the data abstraction when we type in a hand of cards as `'(3h 10c 4d)` because that assumes we know how the cards are represented—namely, as words combining the rank number with a one-letter suit. If we want to be thorough about hiding the representation, we need *constructor* functions as well as the selectors:

```
;;;;;                        In file cs61a/lectures/2.1/total.scm
(define (make-card rank suit)
  (word rank (first suit)) )

(define make-hand se)


> (total (make-hand (make-card 3 'heart)
                    (make-card 10 'club)
                    (make-card 4 'diamond) ))
17
```

Once we're using data abstraction we can change the implementation of the data type without affecting the programs that *use* that data type. This means we can change how we represent a card, for example, without rewriting `total`:

```
;;;;;                        In file cs61a/lectures/2.1/total.scm
(define (make-card rank suit)
  (cond ((equal? suit 'heart) rank)
        ((equal? suit 'spade) (+ rank 13))
        ((equal? suit 'diamond) (+ rank 26))
        ((equal? suit 'club) (+ rank 39))
        (else (error "say what?")) ))

(define (rank card)
  (remainder card 13))

(define (suit card)
  (nth (quotient card 13) '(heart spade diamond club)))
```

We have changed the internal *representation* so that a card is now just a number between 1 and 52 (why? maybe we're programming in FORTRAN) but we haven't changed the *behavior* of the program at all. We still call `total` the same way.

Data abstraction is a really good idea because it helps keep you from getting confused when you're dealing with lots of data types, but don't get religious about it. For example, we have invented the *sentence* data type for this course. We have provided symmetric selectors `first` and `last`, and symmetric selectors `butfirst` and `butlast`. You can write programs using sentences without knowing how they're implemented. But it turns out that because of the way they *are* implemented, `first` and `butfirst` take $\Theta(1)$ time, while `last` and `butlast` take $\Theta(N)$ time. If you know that, your programs will be faster.

• Pairs.

To represent data types that have component parts (like the rank and suit of a card), you have to have some way to *aggregate* information. Many languages have the idea of an *array* that groups some number of elements. In Lisp the most basic aggregation unit is the *pair*—two things combined to form a bigger thing. If you want more than two parts you can hook a bunch of pairs together; we'll discuss this more below.

The constructor for pairs is CONS; the selectors are CAR and CDR.

The book uses pairs to represent many different abstract data types: rational numbers (numerator and denominator), complex numbers (real and imaginary parts), points (x and y coordinates), intervals (low and high bounds), and line segments (two endpoints). Notice that in the case of line segments we think of the representation as *one pair* containing two points, not as three pairs containing four numbers. (That's what it means to respect a data abstraction.)

Note: What's the difference between these two:

```
(define (make-rat num den) (cons num den))
(define make-rat cons)
```

They are both equally good ways to implement a constructor for an abstract data type. The second way has a slight speed advantage (one fewer function call) but the first way has a debugging advantage because you can trace `make-rat` without tracing all invocations of `cons`.

• Data aggregation doesn't have to be primitive.

In most languages the data aggregation mechanism (the array or whatever) seems to be a necessary part of the core language, not something you could implement as a user of the language. But if we have first-class functions we can use a function to represent a pair:

```
;;;;;                        In file cs61a/lectures/2.1/cons.scm
(define (cons x y)
  (lambda (which)
    (cond ((equal? which 'car) x)
          ((equal? which 'cdr) y)
          (else (error "Bad message to CONS" message)) )))


(define (car pair)
  (pair 'car))

(define (cdr pair)
  (pair 'cdr))
```

This is like the version in the book except that they use 0 and 1 as the *messages* because they haven't introduced quoted words yet. This version makes it a little clearer what the argument named `which` means.

The point is that we can satisfy ourselves that this version of `cons`, `car`, and `cdr` works in the sense that if we construct a pair with this `cons` we can extract its two components with this `car` and `cdr`. If that's true, we don't need to have pairs built into the language! All we need is `lambda` and we can implement the rest ourselves. (It isn't really done this way, in real life, for efficiency reasons, but it's neat that it could be.)

• Big idea: abstract data type *sequence* (or *list*).

We want to represent an ordered sequence of things. (They can be any kind of things.) We *implement* sequences using pairs, with each `car` pointing to an element and each `cdr` pointing to the next pair.

What should the constructors and selectors be? The most obvious thing is to have a constructor `list` that takes any number of arguments and returns a list of those arguments, and a selector `nth` that takes a number and a list as arguments, returning the *n*th element of the list.

Scheme does provide those, but it often turns out to be more useful to select from a list differently, with a selector for the first element and a selector for all the rest of the elements (i.e., a smaller list). This helps us write recursive functions such as the mapping and filtering ones we saw for sentences earlier.

Since we are implementing lists using pairs, we ought to have specially-named constructors and selectors for lists, just like for rational numbers:

```
(define adjoin cons)
(define first car)
(define rest cdr)
```

Many Lisp systems do in fact provide `first` and `rest` as synonyms for `car` and `cdr`, but the fact is that this particular data abstraction is commonly violated; we just use the names `car`, `cdr`, and `cons` to talk about lists.

This abstract data type has a special status in the Scheme interpreter itself, because lists are read and printed using a special notation. If Scheme knew only about pairs, and not about lists, then when we construct the list `(1 2 3)` it would print as `(1 . (2 .(3 . ())))` instead.

• Lists vs. sentences.

We started out the semester using an abstract data type called *sentence* that looks a lot like a list. What's the difference, and why did we do it that way?

Our goal was to allow you to create aggregates of words without having to think about the structure of their internal representation (i.e., about pairs). We do this by deciding that the elements of a sentence must be words (not sublists), and enforcing that by giving you the constructor `sentence` that creates only sentences.

Example: One of the homework problems this week asks you to reverse a list. You'll see that this is a little tricky using `cons`, `car`, and `cdr` as the problem asks, but it's easy for sentences:

```
(define (reverse sent)
  (if (empty? sent)
      '()
      (se (reverse (bf sent)) (first sent)) ))
```

To give you a better idea about what a sentence is, here's a version of the constructor function:

```
;;;;;                       In file cs61a/lectures/2.2/sentence.scm
(define (se a b)
  (cond ((word? a) (se (list a) b))
        ((word? b) (se a (list b)))
        (else (append a b)) ))

(define (word? x)
  (or (symbol? x) (number? x)) )
```

`Se` is a lot like `append`, except that the latter behaves oddly if given words as arguments. `Se` can accept words or sentences as arguments.

• Box and pointer diagrams.

Here are a few details that people sometimes get wrong about them:

1. An arrow can't point to half of a pair. If an arrowhead touches a pair, it's pointing to the entire pair, and it doesn't matter exactly where the arrowhead touches the rectangle. If you see something like

```
(define x (car y))
```

where `y` is a pair, the arrow for `x` should point to *the thing that the* `car` *of* `y` *points to*, not to the left half of the `y` rectangle.

2. The direction of arrows (up, down, left, right) is irrelevant. You can draw them however you want to make the arrangement of pairs neat. That's why it's crucial not to forget the arrowheads!

3. There must be a top-level arrow to show where the structure you're representing begins.

How do you draw a diagram for a complicated list? Take this example:

```
((a b) c (d (e f)))
```

You begin by asking yourself how many elements the list has. In this case it has three elements: first `(a b)`, then `c`, then the rest. Therefore you should draw a three-pair *backbone*: three pairs with the `cdr` of one pointing to the next one. (The final `cdr` is null.)

Only after you've drawn the backbone should you worry about making the `car`s of your three pairs point to the three elements of the top-level list.

CS 61A   Lecture Notes   Week 6

Topic: Hierarchical data

Reading: Abelson & Sussman, Section 2.2.2–2.2.3, 2.3.1, 2.3.3

• Trees.

Big idea: representing a hierarchy of information.

Definitions: *node*, *root*, *branch*, *leaf*.

A node is a particular point in the tree, but it's also a subtree, just as a pair *is* a list at the same time that it's a pair.

What are trees good for?

- Hierarchy: world, countries, states, cities.
- Ordering: binary search trees.
- Composition: arithmetic operations at branches, numbers at leaves.

Many problems involve tree *search*: visiting each node of a tree to look for some information there. Maybe we're looking for a particular node, maybe we're adding up all the values at all the nodes, etc. There is one obvious order in which to search a sequence (left to right), but many ways in which we can search a tree.

Depth-first search: Look at a given node's children before its siblings.

Breadth-first search: Look at the siblings before the children.

Within the DFS category there are more kinds of orderings:

Preorder: Look at a node before its children.

Postorder: Look at the children before the node.

Inorder (binary trees only): Look at the left child, then the node, then the right child.

For a tree of arithmetic operations, preorder is Lisp, inorder is conventional arithmetic notation, postorder is HP calculator.

(Note: In 61B we come back to trees in more depth, including the study of *balanced* trees, i.e., using special techniques to make sure a search tree has about as much stuff on the left as on the right.)

• Below-the-line representation of trees.

Lisp has one built-in way to represent sequences, but there is no official way to represent trees. Why not?

- Branch nodes may or may not have data.
- Binary vs. n-way trees.
- Order of siblings may or may not matter.
- Can tree be empty?

We can think about a tree ADT in terms of a selector and constructors:

```
(make-tree datum children)
(datum node)
(children node)
```

The selector `children` should return a list (sequence) of the children of the node. These children are themselves trees. A leaf node is one with no children:

```
(define (leaf? node)
  (null? (children node)) )
```

This definition of `leaf?` should work no matter how we represent the ADT.

If every node in your tree has a datum, then the straightforward implementation is

```
;;;;;                         Compare file cs61a/lectures/2.2/tree1.scm
(define make-tree cons)
(define datum car)
(define children cdr)
```

On the other hand, it's also common to think of any list structure as a tree in which the leaves are words and the branch nodes don't have data. For example, a list like

```
(a (b c d) (e (f g) h))
```

can be thought of as a tree whose root node has three children: the leaf `a` and two branch nodes. For this sort of tree it's common not to use formal ADT selectors and constructors at all, but rather just to write procedures that handle the car and the cdr as subtrees. To make this concrete, let's look at mapping a function over all the data in a tree.

First we review mapping over a sequence:

```
;;;;;                         In file cs61a/lectures/2.2/squares.scm
(define (SQUARES seq)
  (if (null? seq)
      '()
      (cons (SQUARE (car seq))
            (SQUARES (cdr seq)) )))
```

The pattern here is that we apply some operation (`square` in this example) to the data, the elements of the sequence, which are in the `car`s of the pairs, and we recur on the sublists, the `cdr`s.

Now let's look at mapping over the kind of tree that has data at every node:

```
;;;;;                         In file cs61a/lectures/2.2/squares.scm
(define (SQUARES tree)
  (make-tree (SQUARE (datum tree))
             (map SQUARES (children tree)) ))
```

Again we apply the operation to every datum, but instead of a simple recursion for the rest of the list, we have to recur for *each child* of the current node. We use `map` (mapping over a sequence) to provide several recursive calls instead of just one.

If the data are only at the leaves, we just treat each pair in the structure as containing two subtrees:

```
;;;;;                         In file cs61a/lectures/2.2/squares.scm
(define (SQUARES tree)
  (cond ((null? tree) '())
        ((atom? tree) (SQUARE tree))
        (else (cons (SQUARES (car tree))
                    (SQUARES (cdr tree)) )) ))
```

The hallmark of tree recursion is to recur for both the `car` and the `cdr`.

CS 61A Lecture Notes Week 7

Topic: Representing abstract data

Reading: Abelson & Sussman, Sections 2.4 through 2.5.2 (pages 169–200)

Note: The second midterm exam is next week.

The overall problem we're addressing this week is to control the complexity of large systems with many small procedures that handle several types of data. We are building toward the idea of *object-oriented programming*, which many people see as the ultimate solution to this problem, and which we discuss for two weeks starting next week.

Big ideas:

- tagged data
- data-directed programming
- message passing

The first problem is keeping track of types of data. If we see a pair whose `car` is 3 and whose `cdr` is 4, does that represent $\frac{3}{4}$ or does it represent $3 + 4i$?

The solution is *tagged data*: Each datum carries around its own type information. In effect we do `(cons 'rational (cons 3 4))` for the rational number $\frac{3}{4}$, although of course we use an ADT.

Just to get away from the arithmetic examples in the text, we'll use another example about geometric shapes. Our data types will be squares and circles; our operations will be area and perimeter.

We want to be able to say, e.g., `(area circle3)` to get area of a particular (previously defined) circle. To make this work, the function `area` has to be able to tell which type of shape it's seeing. We accomplish this by attaching a type tag to each shape:

```
;;;;;                           In file cs61a/lectures/2.4/geom.scm
(define pi 3.141592654)

(define (make-square side)
  (attach-tag 'square side))

(define (make-circle radius)
  (attach-tag 'circle radius))

(define (area shape)
  (cond ((eq? (type-tag shape) 'square)
         (* (contents shape) (contents shape)))
        ((eq? (type-tag shape) 'circle)
         (* pi (contents shape) (contents shape)))
        (else (error "Unknown shape -- AREA"))))

(define (perimeter shape)
  (cond ((eq? (type-tag shape) 'square)
         (* 4 (contents shape)))
        ((eq? (type-tag shape) 'circle)
         (* 2 pi (contents shape)))
        (else (error "Unknown shape -- PERIMETER"))))

;; some sample data
(define square5 (make-square 5))
(define circle3 (make-circle 3))
```

• Orthogonality of types and operators.

The next problem to deal with is the proliferation of functions because you want to be able to apply every operation to every type. In our example, with two types and two operations we need four algorithms.

What happens when we invent a new type? If we write our program in the *conventional* (i.e., old-fashioned) style as above, it's not enough to add new functions; we have to modify all the operator functions like `area` to know about the new type. We'll look at two different approaches to organizing things better: *data-directed programming* and *message passing.*

The idea in DDP is that instead of keeping the information about types versus operators inside functions, as `cond` clauses, we record this information in a data structure. A&S provide tools `put` to set up the data structure and `get` to examine it:

```
> (get 'foo 'baz)
#f
> (put 'foo 'baz 'hello)
> (get 'foo 'baz)
hello
```

Once you `put` something in the table, it stays there. (This is our first departure from functional programming. But our intent is to set up the table at the beginning of the computation and then to treat it as *constant* information, not as something that might be different the next time you call `get`, despite the example above.) For now we take `put` and `get` as primitives; we'll see how to build them in section 3.3 in three weeks.

The code is mostly unchanged from the conventional version; the tagged data ADT and the two shape ADTs are unchanged. What's different is how we represent the four algorithms for applying some operator to some type:

```
;;;;;                          In file cs61a/lectures/2.4/geom.scm

(put 'square 'area (lambda (s) (* s s)))
(put 'circle 'area (lambda (r) (* pi r r)))
(put 'square 'perimeter (lambda (s) (* 4 s)))
(put 'circle 'perimeter (lambda (r) (* 2 pi r)))
```

Notice that the entry in each cell of the table is a *function*, not a symbol. We can now redefine the six generic operators ("generic" because they work for any of the types):

```
;;;;;                          In file cs61a/lectures/2.4/geom.scm

(define (area shape)
  (operate 'area shape))

(define (perimeter shape)
  (operate 'perimeter shape))

(define (operate op obj)      ;; like APPLY-GENERIC but for one operand
  (let ((proc (get (type-tag obj) op)))
    (if proc
        (proc (contents obj))
        (error "Unknown operator for type"))))
```

Now if we want to invent a new type, all we have to do is a few `put` instructions and the generic operators just automatically work with the new type.

Don't get the idea that DDP just means a two-dimensional table of operator and type names! DDP is a very general, great idea. It means putting the details of a system into data, rather than into programs, so you can write general programs instead of very specific ones.

In the old days, every time a company got a computer they had to hire a bunch of programmers to write things like payroll programs for them. They couldn't just use someone else's program because the details would be different, e.g., how many digits in the employee number. These days you have general business packages and each company can "tune" the program to their specific purpose with a data file.

Another example showing the generality of DDP is the *compiler compiler.* It used to be that if you wanted to invent a new programming language you had to start from scratch in writing a compiler for it. But now we have formal notations for expressing the syntax of the language. (See section 7.1, page 38, of the *Scheme Report* at the back of the course reader.) A single program can read these formal descriptions and compile any language. [The Scheme BNF is in `cs61a/lectures/2.4/bnf`.]

• Message-passing.

In conventional style, the operators are represented as functions that know about the different types; the types themselves are just data. In DDP, the operators and types are all data, and there is one universal `operate` function that does the work. We can also stand conventional style on its head, representing the *types* as functions and the operations as mere data.

In fact, not only are the types functions, but so are the individual data themselves. That is, there is a function (`make-circle` below) that represents the circle type, and when you invoke that function, it returns *a function* that represents the particular circle you give it as its argument. Each circle is an *object* and the function that represents it is a *dispatch procedure* that takes as its argument a *message* saying which operation to perform.

```
;;;;;                        In file cs61a/lectures/2.4/geom.scm

(define (make-square side)
  (lambda (message)
    (cond ((eq? message 'area)
           (* side side))
          ((eq? message 'perimeter)
           (* 4 side))
          (else (error "Unknown message")))))

(define (make-circle radius)
  (lambda (message)
    (cond ((eq? message 'area)
           (* pi radius radius))
          ((eq? message 'perimeter)
           (* 2 pi radius))
          (else (error "Unknown message")))))

(define square5 (make-square 5))
(define circle3 (make-circle 3))
```

The `define`s that produce the individual shapes look no different from before, but the results are different: Each shape is a function, not a list structure. So to get the area of the shape `circle3` we invoke that shape with the proper message: `(circle3 'area)`. That notation is a little awkward so we provide a little "syntactic sugar" that allows us to say `(area circle3)` as in the past:

```
;;;;;                          In file cs61a/lectures/2.4/msg.scm
(define (operate op obj)
  (obj op))

(define (area shape)
  (operate 'area shape))

(define (perimeter shape)
  (operate 'perimeter shape))
```

Message passing may seem like an overly complicated way to handle this problem of shapes, but we'll see next week that it's one of the key ideas in creating object-oriented programming. Message passing becomes much more powerful when combined with the idea of *local state* that we'll learn next week.

We seem to have abandoned tagged data; every shape type is just some function, and it's hard to tell which type of shape a given function represents. We could combine message passing with tagged data, if desired, by adding a `type` message that each object understands.

```
(define (make-square side)
  (lambda (message)
    (cond ((eq? message 'area)
           (* side side))
          ((eq? message 'perimeter)
           (* 4 side))
          ((EQ? MESSAGE 'TYPE) 'SQUARE)
          (else (error "Unknown message")))))
```

• Dyadic operations.

Our shape example is easier than the arithmetic example in the book because our operations only require one operand, not two. For arithmetic operations like `+`, it's not good enough to connect the operation with a type; the two operands might have two different types. What should you do if you have to add a rational number to a complex number?

There is no perfect solution to this problem. For the particular case of arithmetic, we're lucky in that the different types form a sequence of larger and larger sets. Every integer is a rational number; every rational is a real; every real is a complex. So we can deal with type mismatch by *raising* the less-complicated operand to the type of the other one. To add a rational number to a complex number, raise the rational number to complex and then you're left with the problem of adding two complex numbers. So we only need N addition algorithms, not N^2 algorithms, where N is the number of types.

Do we need N^2 raising algorithms? No, because we don't have to know directly how to raise a rational number to complex. We can raise the rational number to the next higher type (real), and then raise that real number to complex. So if we want to add $\frac{1}{3}$ and $2+5i$ the answer comes out $2.3333+5i$.

As this example shows, nonchalant raising can lose information. It would be better, perhaps, if we could get the answer $\frac{7}{3}+5i$ instead of the decimal approximation. Numbers are a rat's nest full of traps for the unwary. You will live longer if you only write programs about integers.

CS 61A   Lecture Notes   Week 8

Topic: Object-oriented programming

Reading: OOP Above-the-line notes in course reader

Midterm Wednesday, 7–9pm.

OOP is an abstraction. Above the line we have the metaphor of multiple independent intelligent agents; instead of one computer carrying out one program we have hordes of *objects* each of which can carry out computations. To make this work there are three key ideas within this metaphor:

- Message passing: An object can ask other objects to do things for it.
- Local state: An object can remember stuff about its own past history.
- Inheritance: One object type can be just like another except for a few differences.

We have invented an OOP language as an extension to Scheme. Basically you are still writing Scheme programs, but with the vocabulary extended to use some of the usual OOP buzzwords. For example, a *class* is a type of object; an *instance* is a particular object. “Complex number” is a class; $3 + 4i$ is an instance. Here’s how the message-passing complex numbers from last week would look in OOP notation:

```
;;;;;                          In file cs61a/lectures/3.0/demo.scm
(define-class (complex real-part imag-part)
  (method (magnitude)
    (sqrt (+ (* real-part real-part)
             (* imag-part imag-part))))
  (method (angle)
    (atan (/ imag-part real-part))) )

> (define c (instantiate complex 3 4))
> (ask c 'magnitude)
5
> (ask c 'real-part)
3
```

This shows how we define the *class* `complex`; then we create the *instance* `c` whose value is $3 + 4i$; then we send `c` a message (we `ask` it to do something) in order to find out that its magnitude is 5. We can also ask `c` about its *instantiation variables*, which are the arguments used when the class is instantiated.

When we send a message to an object, it responds by carrying out a *method*, i.e., a procedure that the object associates with the message.

So far, although the notation is new, we haven’t done anything different from what we did last week in chapter 2. Now we take the big step of letting an object remember its past history, so that we are no longer doing functional programming. The result of sending a message to an object depends not only on the arguments used right now, but also on what messages we’ve sent the object before:

```
;;;;;                          In file cs61a/lectures/3.0/demo.scm
(define-class (counter)
  (instance-vars (count 0))
  (method (next)
    (set! count (+ count 1))
    count) )

> (define c1 (instantiate counter))
> (ask c1 'next)
1
```

```
> (ask c1 'next)
2
> (define c2 (instantiate counter))
> (ask c2 'next)
1
> (ask c1 'next)
3
```

Each counter has its own *instance variable* to remember how many times it's been sent the `next` message.

Don't get confused about the terms *instance* variable versus *instantiation* variable. They are similar in that each instance has its own version; the difference is that instantiation variables are given values when an instance is created, using extra arguments to `instantiate`, whereas the initial values of instance variables are specified in the class definition and are generally the same for every instance (although the values may change as the computation goes on.)

Methods can have arguments. You supply the argument when you `ask` the corresponding message:

```
;;;;;                          In file cs61a/lectures/3.0/demo.scm
(define-class (doubler)
  (method (say stuff) (se stuff stuff)))

> (define dd (instantiate doubler))
> (ask dd 'say 'hello)
(hello hello)
> (ask dd 'say '(she said))
(she said she said)
```

Besides having a variable for each instance, it's also possible to have variables that are shared by every instance of the same class:

```
;;;;;                          In file cs61a/lectures/3.0/demo2.scm
(define-class (counter)
  (instance-vars (count 0))
  (class-vars (total 0))
  (method (next)
    (set! total (+ total 1))
    (set! count (+ count 1))
    (list count total)))

> (define c1 (instantiate counter))
> (ask c1 'next)
(1 1)
> (ask c1 'next)
(2 2)
> (define c2 (instantiate counter))
> (ask c2 'next)
(1 3)
> (ask c1 'next)
(3 4)
```

Now each `next` message tells us both the count for this particular counter and the overall count for all counters combined.

To understand the idea of inheritance, we'll first define a `person` class that knows about talking in various ways, and then define a `pigger` class that's just like a `person` except for talking in Pig Latin:

```
;;;;;                         In file cs61a/lectures/3.0/demo2.scm
(define-class (person name)
  (method (say stuff) stuff)
  (method (ask stuff) (ask self 'say (se '(would you please) stuff)))
  (method (greet) (ask self 'say (se '(hello my name is) name))) )

> (define marc (instantiate person 'marc))
> (ask marc 'say '(good morning))
(good morning)
> (ask marc 'ask '(open the door))
(would you please open the door)
> (ask marc 'greet)
(hello my name is marc)
```

Notice that an object can refer to itself by the name `self`; this is an automatically-created instance variable in every object whose value is the object itself. (We'll see when we look below the line that there are some complications about making this work.)

```
;;;;;                         In file cs61a/lectures/3.0/demo2.scm
(define-class (pigger name)
  (parent (person name))
  (method (pigl wd)
    (if (member? (first wd) '(a e i o u))
        (word wd 'ay)
        (ask self 'pigl (word (bf wd) (first wd))) ))
  (method (say stuff)
    (if (atom? stuff)
        (ask self 'pigl stuff)
        (map (lambda (w) (ask self 'pigl w)) stuff))) )

> (define porky (instantiate pigger 'porky))
> (ask porky 'say '(good morning))
(oodgay orningmay)
> (ask porky 'ask '(open the door))
(ouldway ouyay easeplay openay ethay oorday)
```

The crucial point here is that the `pigger` class doesn't have an `ask` method in its definition. When we ask `porky` to `ask` something, it uses the `ask` method in its parent (`person`) class.

Also, when the parent's `ask` method says `(ask self 'say ...)` it uses the `say` method from the `pigger` class, not the one from the `person` class. So Porky speaks Pig Latin even when asking something.

What happens when you send an object a message for which there is no method defined in its class? If the class has no parent, this is an error. If the class does have a parent, and the parent class understands the message, it works as we've seen here. But you might want to create a class that follows some rule of your own devising for unknown messages:

```
;;;;;                         In file cs61a/lectures/3.0/demo2.scm
(define-class (squarer)
  (default-method (* message message))
  (method (7) 'buzz) )

> (define s (instantiate squarer))
> (ask s 6)                > (ask s 7)                > (ask s 8)
36                         buzz                       64
```

Within the default method, the name `message` refers to whatever message was sent. (The name `args` refers to a list containing any additional arguments that were used.)

Let's say we want to maintain a list of all the instances that have been created in a certain class. It's easy enough to establish the list as a class variable, but we also have to make sure that each new instance automatically adds itself to the list. We do this with an `initialize` clause:

```
;;;;;                          In file cs61a/lectures/3.0/demo2.scm
(define-class (counter)
  (instance-vars (count 0))
  (class-vars (total 0) (counters '()))
  (initialize (set! counters (cons self counters)))
  (method (next)
    (set! total (+ total 1))
    (set! count (+ count 1))
    (list count total)))

> (define c1 (instantiate counter))
> (define c2 (instantiate counter))
> (ask counter 'counters)
(#<procedure> #<procedure>)
```

There was a bug in our `pigger` class definition; Scheme gets into an infinite loop if we ask Porky to `greet`, because it tries to translate the word `my` into Pig Latin but there are no vowels `aeiou` in that word. To get around this problem, we can redefine the `pigger` class so that its `say` method says every word in Pig Latin except for the word `my`, which it'll say using the usual method that `person`s who aren't `pigger`s use:

```
;;;;;                          In file cs61a/lectures/3.0/demo2.scm
(define-class (pigger name)
  (parent (person name))
  (method (pigl wd)
    (if (member? (first wd) '(a e i o u))
        (word wd 'ay)
        (ask self 'pigl (word (bf wd) (first wd))) ))
  (method (say stuff)
    (if (atom? stuff)
        (if (equal? stuff 'my) (usual 'say stuff) (ask self 'pigl stuff))
        (map (lambda (w) (ask self 'say w)) stuff))) )

> (define porky (instantiate pigger 'porky))
> (ask porky 'greet)
(ellohay my amenay isay orkypay)
```

(Notice that we had to create a new instance of the new class. Just doing a new define-class doesn't change any instances that have already been created in the old class. Watch out for this while you're debugging the OOP programming project.)

We invoke `usual` in the `say` method to mean "say this stuff in the usual way, the way that my parent class would use."

The OOP above-the-line section in the course reader talks about even more capabilities of the system, e.g., *multiple inheritance* with more than one parent class for a single child class.

CS 61A   Lecture Notes   Week 9

Topic: Local state variables, environments

Reading: Abelson & Sussman, Section 3.1, 3.2; OOP below the line

We said the three big ideas in the OOP interface are message passing, local state, and inheritance. You know from section 2.4 how message passing is implemented below the line in Scheme, i.e., with a dispatch function that takes a message as argument and returns a method. This week we're talking about how local state works.

A *local* variable is one that's only available within a particular part of the program; in Scheme this generally means within a particular procedure. We've used local variables before; `let` makes them. A *state* variable is one that remembers its value from one invocation to the next; that's the new part.

First of all let's look at *global* state—that is, let's try to remember some information about a computation but not worry about having separate versions for each object.

```
;;;;;                       In file cs61a/lectures/3.1/count1.scm
(define counter 0)

(define (count)
  (set! counter (+ counter 1))
  counter)

> (count)
1
> (count)
2
```

What's new here is the special form `set!` that allows us to change the value of a variable. This is not like `let`, which creates a temporary, local binding; this makes a permanent change in some variable that must have already existed. The syntax is just like `define` (but not the abbreviation for defining a function): it takes an unevaluated name and an expression whose value provides the new value.

A crucial thing to note about `set!` is that the substitution model no longer works. We can't substitute the value of `counter` wherever we see the name `counter`, or we'll end up with

```
(set! 0 (+ 0 1))
0
```

which doesn't make any sense. From now on we use a model of variables that's more like what you learned in 7th grade, in which a variable is a shoebox in which you can store some value. The difference from the 7th grade version is that we can have several shoeboxes with the same name (the instance variables in the different objects, for example) and we have to worry about how to keep track of that. Section 3.2 of A&S explains the *environment* model that keeps track for us.

Another new thing is that a procedure body can include more than one expression. In functional programming, the expressions don't *do* anything except compute a value, and a function can only return one value, so it doesn't make sense to have more than one expression in it. But when we invoke `set!` there is an *effect* that lasts beyond the computation of that expression, so now it makes sense to have that expression and then another expression that does something else. When a body has more than one expression, the expressions are evaluated from left to right (or top to bottom) and the value returned by the procedure is the value computed by the last expression. All but the last are just *for effect.*

We've seen how to have a global state variable. We'd like to try for *local* state variables. Here's an attempt that doesn't work:

```
;;;;;                         In file cs61a/lectures/3.1/count.lose
(define (count)
  (let ((counter 0))                      > (count)
    (set! counter (+ counter 1))          1
    counter))                             > (count)
                                          1
                                          > (count)
                                          1
```

It was a good idea to use `let`, because that's a way we know to create local variables. But `let` creates a *new* local variable each time we invoke it. Each call to `count` creates a new `counter` variable whose value is 0.

The secret is to find a way to call `let` only once, when we *create* the `count` function, instead of calling `let` every time we *invoke* `count`. Here's how:

```
;;;;;                         In file cs61a/lectures/3.1/count2.scm
(define count
  (let ((result 0))
    (lambda ()
      (set! result (+ result 1))
      result)))
```

Notice that there are no parentheses around the word `count` on the first line! Instead of

```
(define count (lambda () (let ...)))
```

(which is what the earlier version means) we have essentially interchanged the `lambda` and the `let` so that the former is inside the latter:

```
(define count (let ... (lambda () ...)))
```

We'll have to examine the environment model in detail before you can really understand why this works. A handwavy explanation is that the `let` creates a variable that's available to things in the body of the `let`; the `lambda` is in the body of the `let`; and so the variable is available to the function that the `lambda` creates.

The reason we wanted local state variables was so that we could have more than one of them. Let's take that step now. Instead of having a single procedure called `count` that has a single local state variable, we'll write a procedure `make-count` that, each time you call it, makes a new counter.

```
;;;;;                         In file cs61a/lectures/3.1/count3.scm

(define (make-count)                   > (define dracula (make-count))
  (let ((result 0))                    > (dracula)
    (lambda ()                         1
      (set! result (+ result 1))       > (dracula)
      result)))                        2
                                       > (define monte-cristo (make-count))
                                       > (monte-cristo)
                                       1
                                       > (dracula)
                                       3
```

Each of `dracula` and `monte-cristo` is the result of evaluating the expression `(lambda () ...)` to produce a procedure. Each of those procedures has access to its own local state variable called `result`. `Result` is temporary with respect to `make-count` but permanent with respect to `dracula` or `monte-cristo`, because the `let` is inside the `lambda` for the former but outside the `lambda` for the latter.

• Environment model of evaluation.

For now we're just going to introduce the central issues about environments, leaving out a lot of details. You'll get those next time.

The question is, what happens when you invoke a procedure? For example, suppose we've said

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

and now we say `(square 7)`; what happens? The substitution model says

1. Substitute the actual argument value(s) for the formal parameter(s) in the body of the function;

2. Evaluate the resulting expression.

In this example, the substitution of 7 for `x` in `(* x x)` gives `(* 7 7)`. In step 2 we evaluate that expression to get the result 49.

We now forget about the substitution model and replace it with the environment model:

1. Create a *frame* with the formal parameter(s) *bound to* the actual argument values;

2. Use this frame to extend the lexical environment;

3. Evaluate the body (without substitution!) in the resulting environment.

A frame is a collection of name-value associations or *bindings*. In our example, the frame has one binding, from `x` to `7`.

Skip step 2 for a moment and think about step 3. The idea is that we are going to evaluate the expression `(* x x)` but we are refining our notion of what it means to evaluate an expression. Expressions are no longer evaluated in a vacuum, but instead, every evaluation must be done with respect to some environment—that is, some collection of bindings between names and values. When we are evaluating `(* x x)` and we see the symbol `x`, we want to be able to look up `x` in our collection of bindings and find the value `7`.

Looking up the value bound to a symbol is something we've done before with global variables. What's new is that instead of one central collection of bindings we now have the possibility of *local* environments. The symbol `x` isn't always `7`, only during this one invocation of `square`. So, step 3 means to evaluate the expression in the way that we've always understood, but looking up names in a particular place.

What's step 2 about? The point is that we can't evaluate `(* x x)` in an environment with nothing but the `x`/`7` binding, because we also have to look up a value for the symbol `*` (namely, the multiplication function). So, we create a new frame in step 1, but that frame isn't an environment by itself. Instead we use the new frame to *extend* an environment that already existed. That's what step 2 says.

Which old environment do we extend? In the `square` example there is only one candidate, the *global* environment. But in more complicated situations there may be several environments available. For example:

```
(define (f x)
  (define (g y)
    (+ x y))
  (g 3))

> (f 5)
```

When we invoke `f`, we create a frame (call it F1) in which `x` is bound to `5`. We use that frame to extend the global environment (call it G), creating a new environment E1. Now we evaluate the body of `f`, which contains the internal definition for `g` and the expression `(g 3)`. To invoke `g`, we create a frame in which `y` is bound to 3. (Call this frame F2.) We are going to use F2 to extend some old environment, but which? G or E1? The body of `g` is the expression `(+ x y)`. To evaluate that, we need an envoironment in which we can look up all of + (in G), x (in F1), and y (in F2). So we'd better make our new environment by extending E1, not by extending G.

The example with f and g shows, in a very simple way, why the question of multiple environments comes up. But it still doesn't show us the full range of possible rules for choosing an environment. In the f and g example, the environment where g is defined is the same as the environment from which it's invoked. But that doesn't always have to be true:

```
(define (make-adder n)
  (lambda (x) (+ x n)))

(define 3+ (make-adder 3))

(define n 7)

> (3+ n)
```

When we invoke make-adder, we create the environment E1 in which n is bound to 3. In the global environment G, we bind n to 7. When we evaluate the expression (3+ n), what environment are we in? What value does n have in this expression? Surely it should have the value 7, the global value. So we evaluate expressions that you type in G. When we invoke 3+ we create the frame F2 in which x is bound to 7. (Remember, 3+ is the function that was created by the lambda inside make-adder.

We are going to use F2 to extend some environment, and in the resulting environment we'll evaluate the body of 3+, namely (+ x n). What value should n have in this expression? It had better have the value 3 or we've defeated the purpose of make-adder. Therefore, the rule is that we do *not* extend the *current* environment at the time the function is invoked, which would be G in this case. Rather, we extend the environment in which the function was *created*, i.e., the environment in which we evaluated the lambda expression that created it. In this case that's E1, the environment that was created for the invocation of make-adder.

Scheme's rule, in which the procedure's defining environment is extended, is called *lexical* scope. The other rule, in which the current environment is extended, is called *dynamic* scope. We'll see in project 4 that a language with dynamic scope is possible, but it would have different features from Scheme.

Remember why we needed the environment model: We want to understand local state variables. The mechanism we used to create those variables was

```
(define some-procedure
  (let ((state-var initial-value))
    (lambda (...) ...)))
```

Roughly speaking, the let creates a frame with a binding for state-var. Within that environment, we evaluate the lambda. This creates a procedure within the scope of that binding. Every time that procedure is invoked, the environment where it was created—that is, the environment with state-var bound—is extended to form the new environment in which the body is evaluated. These new environments come and go, but the state variable isn't part of the new frames; it's part of the frame in which the procedure was defined. That's why it sticks around.

• Here are the complete rules for the environment model:

Every expression is either an atom or a list.

At any time there is a *current frame*, initially the global frame.

I. Atomic expressions.

   A. Numbers, strings, `#T`, and `#F` are self-evaluating.

   B. If the expression is a symbol, find the *first available* binding. (That is, look in the current frame; if not found there, look in the frame "behind" the current frame; and so on until the global frame is reached.)

II. Compound expressions (lists).

If the car of the expression is a symbol that names a special form, then follow its rules (II.B below). Otherwise the expression is a procedure invocation.

   A. Procedure invocation.

   Step 1: Evaluate all the subexpressions (using these same rules).

   Step 2: Apply the procedure (the value of the first subexpression) to the arguments (the values of the other subexpressions).

   (a) If the procedure is compound (user-defined):

      a1: Create a frame with the formal parameters of the procedure bound to the actual argument values.

      a2: Extend the procedure's defining environment with this new frame.

      a3: Evaluate the procedure body, using the new frame as the current frame.

      *** ONLY COMPOUND PROCEDURE INVOCATION CREATES A FRAME ***

   (b) If the procedure is primitive:

      Apply it by magic.

   B. Special forms.

   1. `Lambda` creates a procedure. The left circle points to the text of the `lambda` expression; the right circle points to the defining environment, i.e., to the current environment at the time the `lambda` is seen.

      *** ONLY `LAMBDA` CREATES A PROCEDURE ***

   2. `Define` adds a *new* binding to the *current frame.*

   3. `Set!` changes the *first available* binding (see I.B for the definition of "first available").

   4. `Let` = `lambda` (II.B.1) + invocation (II.A)

   5. `(define (...)  ...)` = `lambda` (II.B.1) + `define` (II.B.2)

   6. Other special forms follow their own rules (`cond`, `if`).

• Environments and OOP.

Class and instance variables are both local state variables, but in different environments:

```
;;;;;                        In file cs61a/lectures/3.2/count4.scm
(define make-count
  (let ((glob 0))
    (lambda ()
      (let ((loc 0))
        (lambda ()
          (set! loc (+ loc 1))
          (set! glob (+ glob 1))
          (list loc glob))))))
```

The class variable `glob` is created in an environment that surrounds the creation of the outer `lambda`, which represents the entire class. The instance variable `loc` is created in an environment that's inside the class `lambda`, but outside the second `lambda` that represents an instance of the class.

The example above shows how environments support state variables in OOP, but it's simplified in that the instance is not a message-passing dispatch procedure. Here's a slightly more realistic version:

```
;;;;;                        In file cs61a/lectures/3.2/count5.scm
(define make-count
  (let ((glob 0))
    (lambda ()
      (let ((loc 0))
        (lambda (msg)
          (cond ((eq? msg 'local)
                 (lambda ()
                   (set! loc (+ loc 1))
                   loc))
                ((eq? msg 'global)
                 (lambda ()
                   (set! glob (+ glob 1))
                   glob))
                (else (error "No such method" msg)) ))))))
```

The structure of alternating `let`s and `lambda`s is the same, but the inner `lambda` now generates a dispatch procedure. Here's how we say the same thing in OOP notation:

```
;;;;;                        In file cs61a/lectures/3.2/count6.scm
(define-class (count)
  (class-vars (glob 0))
  (instance-vars (loc 0))
  (method (local)
    (set! loc (+ loc 1))
    loc)
  (method (global)
    (set! glob (+ glob 1))
    glob))
```

Note: Programming project 3 starts this week.

CS 61A Lecture Notes Week 10

Topic: Mutable data, queues, tables

Reading: Abelson & Sussman, Section 3.3.1–3

Play the animal game:

```
> (load "lectures/3.3/animal.scm")
#f
> (animal-game)
Does it have wings? no
Is it a rabbit? no

I give up, what is it? gorilla

Please tell me a question whose answer is YES for a gorilla
and NO for a rabbit.
Enclose the question in quotation marks.
"Does it have long arms?"
"Thanks.  Now I know better."
> (animal-game)
Does it have wings? no
Does it have long arms? no
Is it a rabbit? yes
"I win!"
```

The crucial point about this program is that its behavior changes each time it learns about a new animal. Such *learning* programs have to modify a data base as they run. We represent the animal game data base as a tree; we want to be able to splice a new branch into the tree (replacing what used to be a leaf node).

Changing what's in a data structure is called *mutation*. Scheme provides primitives `set-car!` and `set-cdr!` for this purpose.

They aren't special forms! The pair that's being mutated must be located by computing some expression. For example, to modify the second element of a list:

```
(set-car! (cdr lst) 'new-value)
```

They're different from `set!`, which changes the binding of a variable. We use them for different purposes, and the syntax is different. Still, they are connected in two ways: (1) Both make your program non-functional, by making a permanent change that can affect later procedure calls. (2) Each can be implemented in terms of the other; the book shows how to use local state variables to simulate mutable pairs, and later we'll see how the Scheme interpreter uses mutable pairs to implement environments, including the use of `set!` to change variable values.

The only purpose of mutation is efficiency. In principle we could write the animal game functionally by recopying the entire data base tree each time, and using the new one as an argument to the next round of the game. But the saving can be quite substantial.

Identity. Once we have mutation we need a subtler view of the idea of equality. Up to now, we could just say that two things are equal if they look the same. Now we need *two* kinds of equality, that old kind plus a new one: Two things are *identical* if they are the very same thing, so that mutating one also changes the other. Example:

```
> (define a (list 'x 'y 'z))
> (define b (list 'x 'y 'z))
> (define c a)
> (equal? b a)
#T
> (eq? b a)
#F
> (equal? c a)
#T
> (eq? c a)
#T
```

The two lists `a` and `b` are equal, because they print the same, but they're not identical. The lists `a` and `c` are identical; mutating one will change the other:

```
> (set-car! (cdr a) 'foo)
> a
(X FOO Z)
> b
(X Y Z)
> c
(X FOO Z)
```

If we use mutation we have to know what shares storage with what. For example, `(cdr a)` shares storage with `a`. `(Append a b)` shares storage with `b` but not with `a`. (Why not? Read the `append` procedure.)

The Scheme standard says you're not allowed to mutate quoted constants. That's why I said `(list 'x 'y 'z)` above and not `'(x y z)`. The text sometimes cheats about this. The reason is that Scheme implementations are allowed to share storage when the same quoted constant is used twice in your program.

Here's the animal game:

```
;;;;;                        In file cs61a/lectures/3.3/animal.scm
(define (animal node)
  (define (type l) (car l))
  (define (question l) (cadr l))
  (define (yespart l) (caddr l))
  (define (nopart l) (cadddr l))
  (define (answer l) (cadr l))
  (define (leaf? l) (eq? (type l) 'leaf))
  (define (branch? l) (eq? (type l) 'branch))
  (define (set-yes! node x)
    (set-car! (cddr node) x))
  (define (set-no! node x)
    (set-car! (cdddr node) x))

  (define (yorn)
    (let ((yn (read)))
      (cond ((eq? yn 'yes) #t)
            ((eq? yn 'no) #f)
            (else (display "Please type YES or NO")
                  (yorn)))))
  (display (question node))
  (display " ")
  (let ((yn (yorn)) (correct #f) (newquest #f))
    (let ((next (if yn (yespart node) (nopart node))))
      (cond ((branch? next) (animal next))
            (else (display "Is it a ")
                  (display (answer next))
                  (display "? ")
                  (cond ((yorn) "I win!")
                        (else (newline)
                              (display "I give up, what is it? ")
                              (set! correct (read))
                              (newline)
                               (display "Please tell me a question whose answer ")
                              (display "is YES for a ")
                              (display correct)
                              (newline)
                              (display "and NO for a ")
                              (display (answer next))
                              (display ".")
                              (newline)
                              (display "Enclose the question in quotation marks.")
                              (newline)
                              (set! newquest (read))
                              (if yn
                                  (set-yes! node (make-branch newquest
                                                              (make-leaf correct)
                                                              next))
                                  (set-no! node (make-branch newquest
                                                             (make-leaf correct)
                                                             next)))
                              "Thanks.  Now I know better.")))))))

(define (make-branch q y n)
  (list 'branch q y n))

(define (make-leaf a)
  (list 'leaf a))

(define animal-list
  (make-branch "Does it have wings?"
               (make-leaf 'parrot)
               (make-leaf 'rabbit)))


(define (animal-game) (animal animal-list))
```

Things to note: Even though the main structure of the program is sequential and BASIC-like, we haven't abandoned data abstraction. We have constructors, selectors, and *mutators*—a new idea—for the nodes of the game tree.

• Tables. We're now ready to understand how to implement the `put` and `get` procedures that A&S used at the end of chapter 2. A table is a list of key-value pairs, with an extra element at the front just so that adding the first entry to the table will be no diffferent from adding later entries. (That is, even in an "empty" table we have a pair to `set-cdr!`)

```
;;;;;                        In file cs61a/lectures/3.3/table.scm
(define (get key)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)
        #f
        (cdr record))))

(define (put key value)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)
        (set-cdr! the-table
                  (cons (cons key value)
                        (cdr the-table)))
        (set-cdr! record value)))
  'ok)

(define the-table (list '*table*))
```

`Assoc` is in the book:

```
(define (assoc key records)
  (cond ((null? records) #f)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records))) ))
```

In chapter 2, A&S provided a single, global table, but we can generalize this in the usual way by taking an extra argument for which table to use. That's how `lookup` and `insert!` work.

One little detail that always confuses people is why, in creating two-dimensional tables, we don't need a `*table*` header on each of the subtables. The point is that `lookup` and `insert!` don't pay any attention to the `car` of that header pair; all they need is to represent a table by *some* pair whose cdr points to the actual list of key-value pairs. In a subtable, the key-value pair from the top-level table plays that role. That is, the entire subtable is a value of some key-value pair in the main table. What it means to be "the value of a key-value pair" is to be the `cdr` of that pair. So we can think of that pair as the header pair for the subtable.

• Memoization. Exercise 3.27 is a pain in the neck because it asks for a very complicated environment diagram, but it presents an extremely important idea. If we take the simple Fibonacci number program:

```
;;;;;                         In file cs61a/lectures/3.3/fib.scm
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1))
         (fib (- n 2)) )))
```

we recall that it takes $\Theta(2^n)$ time because it ends up doing a lot of subproblems redundantly. For example, if we ask for `(fib 5)` we end up computing `(fib 3)` twice. We can fix this by *remembering* the values that we've already computed. The book's version does it by entering those values into a local table. It may be simpler to understand this version, using the global get/put:

```
;;;;;                         In file cs61a/lectures/3.3/fib.scm
(define (fast-fib n)
  (if (< n 2)
      n                               ; base case unchanged
      (let ((old (get 'fib n)))
        (if (number? old)             ; do we already know the answer?
            old
            (begin                    ; if not, compute and learn it
             (put 'fib n (+ (fast-fib (- n 1))
                            (fast-fib (- n 2))))
             (get 'fib n))))))
```

Is this functional programming? That's a more subtle question than it seems. Calling `memo-fib` makes a permanent change in the environment, so that a second call to `memo-fib` with the same argument will carry out a very different (and much faster) process. But the new process will get the same answer! If we look inside the box, `memo-fib` works non-functionally. But if we look only at its input-output behavior, `memo-fib` *is* a function because it always gives the same answer when called with the same argument.

What if we tried to memoize `random`? It would be a disaster; instead of getting a random number each time, we'd get the same number repeatedly! Memoization only makes sense if the underlying function really *is* functional.

This idea of using a non-functional implementation for something that has functional behavior will be very useful later when we look at streams.

CS 61A Lecture Notes Week 11

Topic: Networks, client/server

Reading: none

There have always been short-distance networks in which all the machines were of the same type — things like Appletalk for the Mac or Novell for the PC. But it's much harder to build a long-distance network (because the connections will be unreliable) with many different kinds of computers.

The first large-scale attack on this problem was the Arpanet, so called because it was funded by the Advanced Research Projects Agency of the US Defense Department. It was established in 1969. The difficult issues of heterogeneous machines and unreliable links were handled by a kind of abstraction layer in hardware. Instead of connecting computers directly to the net, and having to program each of them to deal with the interconnection issues, each computer was connected to another machine, a special-purpose computer called an IMP (Interface Message Processor). The IMP handled the routing problems, the reliability problems, and the heterogeneity problems. The computers attached to the network only had to deal with the simplified interface provided by the IMP. Routing information (who has direct links to whom) was centrally maintained and distributed automatically to all the IMPs. This worked fine while there were only a few hundred computers on the net.

The Arpanet was designed in the days of very expensive computers. Each Arpanet site was likely to have only a few computers, and each computer could be an Arpanet machine. But soon there were less expensive computers, and each site had several machines. The solution was that a central computer at each site would be on the Arpanet, and would also be attached to a local network. The central machine served as a *gateway* through which users of the other machines could access the network. The Arpanet didn't know about those secondary machines, so users had to know about the gateway machines explicitly. For example, electronic mail could be sent using the addressing format `person@host` for hosts directly attached to the Arpanet, but for users on secondary machines you had to use a kludgy notation such as `person%host@gateway`. The Arpanet treated this as mail for a person named `person%host`; the gateway machine would then resend the mail on its local network.

By the time personal workstations came around, it was common to have network wiring three levels deep: One machine on campus would connect to the Arpanet; a network would connect that machine to a gateway in every department; that low-level gateway would connect to a local network within the department. The goal was that people outside the organization shouldn't have to know how its networks are wired, so the Arpanet was replaced with the Internet — a network of networks.

Names and addresses

The Internet is designed around the idea that no central authority knows all the machines on the net. This is why locating a particular machine uses a *hierarchical* structure.

Human beings refer to computers by name, such as `quasar.cs.berkeley.edu` or `www.microworlds.com`. The periods inside each name separate pieces of the name that are looked up separately. The rightmost piece (such as `edu` or `com`) specifies a *top-level domain*. The `edu` domain is for educational institutions, while `com` is for commercial enterprises. There are *top-level nameservers* that know where to find the nameservers for all of the top-level domains. (Each top-level domain has more than one nameserver, so that the network doesn't collapse if one machine is temporarily unreachable.) The `edu` nameservers know where to find nameservers for second level domains such as `berkeley.edu`; the Berkeley nameservers know how to find the `cs.berkeley.edu` domain's nameservers. Finally, the Computer Science Division nameservers know how to find individual computers such as `quasar.cs.berkeley.edu`. This means that adding a new machine in the Computer Science Division to the Internet can be done by our system administrators, without anybody outside of CS having to approve, or even know about, the new machine.

What does it mean to know "how to find" a computer? Each computer has a *host address*, also known as an *IP address*. ("IP" stands for "Internet Protocol," which will be discussed later.) The host address is four numbers separated by periods. For example, the host address for `quasar` is `128.32.42.25`. Each of the numbers is between 0 and 255 (so they fit in one byte of computer memory). Every address that starts `128.32` is at Berkeley. Local networks that are physically nearby don't necessarily have similar IP addresses (by contrast with US Postal Service Zip Codes, for example); the web server for the City of Berkeley is at `209.232.44.21`.

Who organizes the assignment of names and addresses? In the Arpanet days, this was simple; one government agency, ARPA, was in charge of the entire network. Today it's much more complicated, partly because current US government policies favor the idea that everything should be run privately and through competition, rather than by government. (Consider the fact that your home telephone was built by one company, is connected through wires owned by another company, with local telephone service provided by a third company and long distance service from a fourth. Until 1982, all four of these were provided by a single organization, AT&T, under tight government regulation.) Today, overall management of Internet names and numbers is controlled by the Internet Corporation for Assigned Names and Numbers (ICANN). ICANN authorizes other organizations to control IP addresses within a region, such as the American Registry for Internet Numbers (ARIC) in the Americas and part of Africa. It also authorizes the creation of top-level domains, each of which is controlled by yet another organization, such as Verisign, Inc., which controls the `com`, `org`, and `net` domains. Yet other organizations are licensed to "retail" names within these domains.

Client/server architecture

Before networks, most programs ran on a single computer. Today it's common for programs to involve cooperation between computers. The usual reason is that you want to run a program on your computer that uses data located elsewhere. An example is the use of nameservers to translate host names to IP addresses, discussed above. Another common example is using a browser on your computer to read a web page stored somewhere else.

To make this cooperation possible, *two* programs are actually required: the *client* program on your personal computer and the *server* program on the remote computer. Sometimes the client and the server are written by a single group, but often someone publishes a *standard* document that allows any client to work with any server that follows the same standard. For example, you can use Netscape or Internet Explorer to read most web pages, because they both follow standards set by the World Wide Web Consortium.

For this course we provide a sample client/server system, implementing a simple Instant Message protocol. The files are available in

```
~cs61a/lib/im-client.scm
~cs61a/lib/im-server.scm
```

To use them, you must first start a server. Load `im-server.scm` and call the procedure `im-server-start`; it will print the IP address of the machine you're using, along with another number, the *port* assigned to the server. Clients will use these numbers to connect to the server. Port numbers are important because there might be more than one server program running on the same computer, and also to keep track of connections from more than one client.

(Why don't you need these numbers when using "real" network software? You don't need to know the IP address because your client software knows how to connect to nameservers to translate the host names you give it into addresses. And most client/server protocols use fixed, *registered* port numbers that are built into the software. For example, web browsers use port 80, while the `ssh` protocol you may use to connect to your class account from home uses port 22. But our sample client/server protocol doesn't have a registered port number, so the operating system assigns a port to the server when you start it.)

To connect to the server, load `im-client.scm` and call `im-enroll` with the IP address and port number as arguments. (Details are in this week's lab assignment.) Then use the `im` procedure to send a message to other people connected to the same server.

This simple implementation uses the Scheme interpreter as its user interface; you send messages by typing

Scheme expressions. Commercial Instant Message clients have a more ornate user interface, that accept mouse clicks in windows listing other clients to specify the recipient of a message. But our version is realistic in the way it uses the network; the IM client on your home computer connects to a particular port on a particular server in order to use the facility. (The only difference is that a large commercial IM system will have more than one server; your client connects to the one nearest you, and the servers send messages among themselves to give the illusion of one big server to which everyone is connected.)

In the news these days, client/server protocols are sometimes contrasted with another approach called *peer-to-peer* networking, such as file-sharing systems like Napster and Kazaa. The distinction is social rather than strictly technical. In each individual transaction using a peer-to-peer protocol, one machine is acting as a server and the other as a client. What makes it peer-to-peer networking is that any machine using the protocol can play either role, unlike the more usual commercial networking idea in which rich companies operate servers and ordinary people operate clients.

Layers of abstraction

Hardware network connections have several undesirable properties:

- You can only connect to machines on the same physical net.
- Messages may get lost.
- Messages may be garbled.
- Messages may arrive out of order.

User-level programs that use the network (ssh, scp, Netscape, etc.) shouldn't have to cope with this complexity. We'd like to be able to present user programs with an abstract network in which messages can be sent reliably to anywhere in the world. Several levels of abstraction are involved:

application layer	user-level program	meaningful messages
transport layer	Transmission Control Protocol (TCP)	reliable streams
network layer	Internet Protocol (IP)	forwarding worldwide
datalink layer	device driver	talk to neighbors

And below the device driver, of course, is the network interface hardware. The device driver provides only the capabilities of the hardware connection, namely, the possibility of sending a message to another machine on the same immediate network, perhaps unreliably.

The Internet Protocol specifies how messages can be forwarded through gateway machines (today they are special-purpose machines, called *routers*, that do nothing but forward network traffic) to faraway computers. The difficulty in this protocol is that no single machine knows the entire map of connections in the Internet; hundreds of new machines are added every day. So the protocol must allow each router to pass the message one step closer to its destination, knowing only its immediate situation. For example, in Soda Hall there are local networks on every floor. A router in Soda might know that messages for any machine on network number 128.32.153 should be sent to the seventh floor router, and similar information for the other local building nets; a message for any other network should be sent over to Evans Hall, where the campus central computing facility will figure out whether to send it to some other campus building or out to Qwest or to calren2.net for connection to the rest of the world.

The Transmission Control Protocol specifies how individual host-to-host unreliable messages can be used to simulate a steady, reliable, in order, program-to-program connection. The protocol involves assigning sequence numbers to messages, acknowledging the receipt of each message, remembering which messages have already come in, and so on.

Finally, the application program can be written in terms of the abstraction provided by the TCP/IP layers, so that writing to a network connection is no harder than writing to a file. (Some applications don't use the TCP abstraction, because it's significantly slower than using raw IP.)

There are many details not presented here; you'll learn more about it in CS 162 or EE 122. For example, TCP must worry about overloading the input buffer capacity on the receiving end, and also about overloading the carrying capacity of the network links between here and there.

Internet primitives in STk

STk defines sockets, on systems which support them, as first class objects. Sockets permit processes to communicate even if they are on different machines. Sockets are useful for creating client-server applications.

`(make-client-socket hostname port-number)`

`make-client-socket` returns a new socket object. This socket establishes a link between the running application listening on port `port-number` of `hostname`.

`(socket? socket)`

Returns `#t` if `socket` is a socket, otherwise returns `#f`.

`(socket-host-name socket)`

Returns a string which contains the name of the distant host attached to `socket`. If `socket` was created with `make-client-socket`, this procedure returns the official name of the distant machine used for connection. If `socket` was created with `make-server-socket`, this function returns the official name of the client connected to the socket. If no client has yet used the socket, this function returns `#f`.

`(socket-host-address socket)`

Returns a string which contains the IP number of the distant host attached to `socket`. If `socket` was created with `make-client-socket`, this procedure returns the IP number of the distant machine used for connection. If `socket` was created with `make-server-socket`, this function returns the address of the client connected to the socket. If no client has yet used the socket, this function returns `#f`.

`(socket-local-address socket)`

Returns a string which contains the IP number of the local host attached to `socket`.

`(socket-port-number socket)`

Returns the integer number of the port used for `socket`.

`(socket-input socket)`
`(socket-output socket)`

Returns the port associated for reading or writing with the program connected with `socket`. If no connection has been established, these functions return `#f`. The following example shows how to make a client socket. Here we create a socket on port 13 of the machine `kaolin.unice.fr`. [Port 13 is generally used for testing: making a connection to it returns the distant system's idea of the time of day.]

```
(let ((s (make-client-socket "kaolin.unice.fr" 13)))
  (format #t "Time is: ~A\n" (read-line (socket-input s)))
  (socket-shutdown s))
```

`(make-server-socket)`
`(make-server-socket port-number)`

`make-server-socket` returns a new socket object. If `port-number` is specified, the socket listens on the specified port; otherwise, the communication port is chosen by the system.

(socket-accept-connection socket)

`socket-accept-connection` waits for a client connection on the given socket. If no client is already waiting for a connection, this procedure blocks its caller; otherwise, the first connection request on the queue of pending connections is connected to socket. This procedure must be called on a server socket created with `make-server-socket`. The return value of `socket-accept-connection` is undefined. The following example is a simple server which waits for a connection on the port 1234. Once the connection with the distant program is established, we read a line on the input port associated to the socket and we write the length of this line on its output port. [Under Unix, you can simply connect to listening socket with the telnet command. With the given example, this can be achieved by typing the command

```
telnet localhost 1234
```

in a shell window.]

```
(let ((s (make-server-socket 1234)))
  (socket-accept-connections)
  (let ((l (read-line (socket-input s))))
    (format (socket-output s) "Length is: ~A\n" (string-length l))
    (flush (socket-output s)))
  (socket-shutdown s))
```

(socket-shutdown socket)
(socket-shutdown socket close)

`Socket-shutdown` shuts down the connection associated to `socket`. `Close` is a boolean; it indicates if the socket must be closed or not, when the connection is destroyed. Closing the socket forbids further connections on the same port with the `socket-accept-connection` procedure. Omitting a value for `close` implies closing the socket. The return value of `socket-shutdown` is undefined. The following example shows a simple server: when there is a new connection on the port number 1234, the server displays the first line sent to it by the client, discards the others and goes back waiting for further client connections.

```
(let ((s (make-server-socket 1234)))
  (let loop ()
    (socket-accept-connections)
    (format #t "I've read: ~A\n" (read-line (socket-input s)))
    (socket-shutdown s #f)
    (loop)))
```

(socket-down? socket)

Returns `#t` if socket has been previously closed with socket-shutdown. It returns `#f` otherwise.

(socket-dup socket)

Returns a copy of `socket`. The original and the copy socket can be used interchangeably. However, if a new connection is accepted on one socket, the characters exchanged on this socket are not visible on the other socket. Duplicating a socket is useful when a server must accept multiple simultaneous connections. The following example creates a server listening on port 1234. This server is duplicated and, once two clients are present, a message is sent on both connections.

```
(define s1 (make-server-socket 1234))
(define s2 (socket-dup s1))
(socket-accept-connection s1)
(socket-accept-connection s2) ;; blocks until two clients are present
(display "Hello,\n" (socket-output s1))
(display "world\n" (socket-output s2))
(flush (socket-output s1))
(flush (socket-output s2))
```

`(when-socket-ready socket handler)`
`(when-socket-ready socket)`

Defines a handler for `socket`. The handler is a thunk which is executed when a connection is available on socket. If the special value `#f` is provided as `handler`, the current handler for socket is deleted. If a handler is provided, the value returned by `when-socket-ready` is undefined. Otherwise, it returns the handler currently associated to `socket`. This procedure, in conjunction with `socket-dup`, permits building multiple-client servers which work asynchronously. Such a server is shown below.

```
(define p (make-server-socket 1234))
(when-socket-ready p
  (let ((count 0))
    (lambda ()
      (set! count (+ count 1))
      (register-connection (socket-dup p) count))))
(define register-connection
  (let ((sockets '()))
    (lambda (s cnt)
      ;; Accept connection
      (socket-accept-connection s)
      ;; Save socket somewhere to avoid GC problems
      (set! sockets (cons s sockets))
      ;; Create a handler for reading inputs from this new connection
      (let ((in (socket-input s))
            (out (socket-output s)))
        (when-port-readable in
          (lambda ()
            (let ((l (read-line in)))
              (if (eof-object? l)
                  ;; delete current handler
                  (when-port-readable in #f)
                  ;; Just write the line read on the socket
                  (begin (format out "On #~A --> ~A\n" cnt l)
                         (flush out))))))))))
```

Note: Next week is the third midterm.

CS 61A Lecture Notes Week 12

Topic: Metacircular evaluator

Reading: Abelson & Sussman, Section 4.1.1–6

Midterm Wednesday, 7–9pm.

We're going to investigate a Scheme interpreter written in Scheme. This interpreter implements the environment model of evaluation.

Why bother? What good is an interpreter for Scheme that we can't use unless we already have another interpreter for Scheme?

- It helps you understand the environment model.
- It lets us experiment with modifications to Scheme (new features).
- Even real Scheme interpreters are largely written in Scheme.
- It illustrates a big idea: *universality.*

Universality means we can write *one program* that's equivalent to all other programs. At the hardware level, this is the idea that made general-purpose computers possible. It used to be that they built a separate machine, from scratch, for every new problem. An intermediate stage was a machine that had a *patchboard* so you could rewire it, effectively changing it into a different machine for each problem, without having to re-manufacture it. The final step was a single machine that accepted a program *as data* so that it can do any problem without rewiring.

Instead of a function machine that computes a particular function, taking (say) a number in the input hopper and returning another number out the bottom, we have a *universal* function machine that takes *a function machine* in one input hopper, and a number in a second hopper, and returns whatever number the input machine would have returned. This is the ultimate in data-directed programming.

Our Scheme interpreter leaves out some of the important components of a real one. It gets away with this by taking advantage of the capabilities of the underlying Scheme. Specifically, we don't deal with storage allocation, tail recursion elimination, or implementing any of the Scheme primitives. All we *do* deal with is the evaluation of expressions. That turns out to be quite a lot in itself, and pretty interesting.

Here is a one-screenful version of the metacircular evaluator with most of the details left out:

```
;;;;;                        In file cs61a/lectures/4.1/micro.scm
(define (scheme)
  (display "> ")
  (print (eval (read) the-global-environment))
  (scheme) )

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((symbol? exp) (lookup-in-env exp env))
        ((special-form? exp) (do-special-form exp env))
        (else (apply (eval (car exp) env)
                     (map (lambda (e) (eval e env)) (cdr exp)) ))))

(define (apply proc args)
  (if (primitive? proc)
      (do-magic proc args)
      (eval (body proc)
            (extend-environment (formals proc)
                                args
                                (proc-env proc) ))))
```

Although the version in the book is a lot bigger, this really does capture the essential structure, namely, a mutual recursion between `eval` (evaluate an expression relative to an environment) and `apply` (apply a function to arguments). To evaluate a compound expression means to evaluate the subexpressions recursively, then apply the `car` (a function) to the `cdr` (the arguments). To apply a function to arguments means to evaluate the body of the function in a new environment.

What's left out? Primitives, special forms, and a lot of details.

In that other college down the peninsula, they wouldn't consider you ready for an interpreter until junior or senior year. At this point in the introductory course, they'd still be teaching you where the semicolons go. How do we get away with this? We have two big advantages:

- The *source language* (the language that we're interpreting) is simple and uniform. Its entire formal syntax can be described in one page, as we did in week 7. There's hardly anything to implement!

- The *implementation language* (the one in which the interpreter itself is written) is powerful enough to handle a program as data, and to let us construct data structures that are both hierarchical and circular.

The amazing thing is that the simple source language and the powerful implementation language are both Scheme! You might think that a powerful language has to be complicated, but it's not so.

• Introduction to Logo. For the programming project you're turning the metacircular evaluator into an interpreter for a *different* language, Logo. To do that you should know a little about Logo itself.

Logo is a dialect of Lisp, just as Scheme is, but its design has different priorities. The goal was to make it as natural-seeming as possible for kids. That means things like getting rid of all those parentheses, and that has other syntactic implications.

(To demonstrate Logo, run `~cs61a/logo` which is Berkeley Logo.)

Commands and operations: In Scheme, every procedure returns a value, even the ones for which the value is unspecified and/or useless, like `define` and `print`. In Logo, procedures are divided into operations, which return values, and commands, which don't return values but are called for their effect. You have to start each instruction with a command:

```
print sum 2 3
```

Syntax: If parentheses aren't used to delimit function calls, how do you know the difference between a function and an argument? When a symbol is used without punctuation, that means a function call. When you want the value of a variable to use as an argument, you put colon in front of it.

```
make "x 14
print :x
print sum :x :x
```

Words are quoted just as in Scheme, except that the double-quote character is used instead of single-quote. But since expressions aren't represented as lists, the same punctuation that delimits a list also quotes it:

```
print [a b c]
```

(Parentheses *can* be used, as in Scheme, if you want to give extra arguments to something, or indicate infix precedence.)

```
print (sum 2 3 4 5)
print 3*(4+5)
```

No special forms: Except `to`, the thing that defines a new procedure, all Logo primitives evaluate their arguments. How is this possible? We "proved" back in chapter 1 that `if` has to be a special form. But instead we just quote the arguments to `ifelse`:

```
ifelse 2=3 [print "hi] [print "bye]
```

You don't notice the quoting since you get it for free with the list grouping.

Functions not first class: In Logo every function has a name; there's no `lambda`. Also, the namespace for functions is separate from the one for variables; a variable can't have a function as its value. (This is convenient because we can use things like `list` or `sentence` as formal parameters without losing the functions by those names.) That's another reason why you need colons for variables.

So how do you write higher-order functions like `map`? Two answers. First, you can use the *name* of a function as an argument, and you can use that name to construct an expression and eval it with `run`. Second, Logo has first-class *expressions*; you can `run` a list that you get as an argument. (This raises issues about the scope of variables that we'll explore later this week.)

```
print map "first [the rain in spain]
print map [? * ?] [3 4 5 6]
```

• Data abstraction in the evaluator. Here is a quote from the Instructor's Manual, regarding section 4.1.2:

"Point out that this section is boring (as is much of section 4.1.3), and explain why: Writing the selectors, constructors, and predicates that implement a representation is often uninteresting. It is important to say explicitly what you expect to be boring and what you expect to be interesting so that students don't ascribe their boredom to the wrong aspect of the material and reject the interesting ideas. For example, data abstraction isn't boring, although writing selectors is. The details of representing expressions (as given in section 4.1.2) and environments (as given in section 4.1.3) are mostly boring, but the evaluator certainly isn't."

I actually think they go overboard by having a separate ADT for every kind of homogeneous sequence. For example, instead of `first-operand` and `rest-operands` I'd just use `first` and `rest` for all sequences. But things like `operator` and `operands` make sense.

• Dynamic scope. Logo uses dynamic scope, which we discussed in Section 3.2, instead of Scheme's lexical scope. There are advantages and disadvantages to both approaches.

Summary of arguments for lexical scope:

- Allows local state variables (OOP).
- Prevents name "capture" bugs.
- Faster compiled code.

Summary of arguments for dynamic scope:

- Allows first-class expressions (WHILE).
- Easier debugging.
- Allows "semi-global" variables.

Lexical scope is required in order to make possible Scheme's approach to local state variables. That is, a procedure that has a local state variable must be defined within the scope where that variable is created, and must carry that scope around with it. That's exactly what lexical scope accomplishes.

On the other hand, (1) most lexically scoped languages (e.g., Pascal) don't have `lambda`, and so they can't give you local state variables despite their lexical scope. And (2) lexical scope is needed for local state variables only if you want to implement the latter in the particular way that we've used. Object Logo, for example, provides OOP without relying on `lambda` because it includes local state variables as a primitive feature.

Almost all computer scientists these days hate dynamic scope, and the reason they give is the one about name captures. That is, suppose we write procedure P that refers to a global variable V. Example:

```
(define (area rad)
  (* pi rad rad))
```

This is intended as a reference to a global variable `pi` whose value, presumably, is 3.141592654. But suppose we invoke it from within another procedure like this:

```
(define (mess-up pi)
  (area (+ pi 5)))
```

If we say `(mess-up 4)` we intend to find the area of a circle with radius 9. But we won't get the right area if we're using dynamic scope, because the name `pi` in procedure `area` suddenly refers to the local variable in `mess-up`, rather than to the intended global value.

This argument about naming bugs is particularly compelling to people who envision a programming project in which 5000 programmers work on tiny slivers of the project, so that nobody knows what anyone else is doing. In such a situation it's entirely likely that two programmers will happen to use the same name for different purposes. But note that we had to do something pretty foolish—using the name `pi` for something that isn't π at all—in order to get in trouble.

Lexical scope lets you write compilers that produce faster executable programs, because with lexical scope you can figure out during compilation exactly where in memory any particular variable reference will be. With dynamic scope you have to defer the name-location correspondence until the program actually runs. This is the real reason why people prefer lexical scope, despite whatever they say about high principles.

As an argument for dynamic scope, consider this Logo implementation of the `while` control structure:

```
to while :condition :action
if not run :condition [stop]
run :action
while :condition :action
end

to example :x
while [:x > 0] [print :x make "x :x-1]
end

? example 3
3
2
1
```

This wouldn't work with lexical scope, because within the procedure `while` we couldn't evaluate the argument expressions, because the variable `x` is not bound in any environment lexically surrounding `while`. Dynamic scope makes the local variables of `example` available to `while`. That in turn allows first-class expressions. (That's what Logo uses in place of first-class functions.)

There are ways to get around this limitation of lexical scope. If you wanted to write `while` in Scheme, basically, you'd have to make it a special form that turns into something using thunks. That is, you'd have to make

```
(while cond act)
```

turn into

```
(while-helper (lambda () cond) (lambda () act))
```

But the Logo point of view is that it's easier for a beginning programmer to understand first-class expressions than to understand special forms and thunks.

Most Scheme implementations include a debugger that allows you to examine the values of variables after an error. But, because of the complexity of the scope rules, the debugging language isn't Scheme itself. Instead you have to use a special language with commands like "switch to the environment of the procedure that called this one." In Logo, when an error happens you can *pause* your program and type ordinary Logo expressions in an environment in which all the relevant variables are available. For example, here is a Logo program:

```
;;;;;                        In file cs61a/lectures/4.1/bug.logo
to assq :thing :list
if equalp :thing first first :list [op last first :list]
op assq :thing bf :list
end

to spell :card
pr (se assq bl :card :ranks "of assq last :card :suits)
end

to hand :cards
if emptyp :cards [stop]
spell first :cards
hand bf :cards
end

make "ranks [[a ace] [2 two] [3 three] [4 four] [5 five] [6 six] [7 seven]
             [8 eight] [9 nine] [10 ten] [j jack] [q queen] [k king]]
make "suits [[h hearts] [s spades] [d diamonds] [c clubs]]

? hand [10h 2d 3s]
TEN OF HEARTS
TWO OF DIAMONDS
THREE OF SPADES
```

Suppose we introduce an error into `hand` by changing the recursive call to

```
hand first bf :cards
```

The result will be an error message in `assq`—two procedure calls down—complaining about an empty argument to `first`. Although the error is caught in `assq`, the real problem is in `hand`. In Logo we can say `pons`, which stands for "print out names," which means to show the values of *all* variables accessible at the moment of the error. This will include the variable `cards`, so we'll see that the value of that variable is a single card instead of a list of cards.

Finally, dynamic scope is useful for allowing "semi-global" variables. Take the metacircular evaluator as an example. Lots of procedures in it require `env` as an argument, but there's nothing special about the value of `env` in any one of those procedures. It's almost always just the current environment, whatever that happens to be. If Scheme had dynamic scope, `env` could be a parameter of `eval`, and it would then automatically be available to any subprocedure called, directly or indirectly, by `eval`. (This is the flip side of the name-capturing problem; in this case we *want* `eval` to capture the name `env`.)

• Environments as circular lists. When we first saw circular lists in chapter 2, they probably seemed to be an utterly useless curiosity, especially since you can't print one. But in the MC evaluator, every environment is a circular list, because the environment contains procedures and each procedure contains a pointer to the environment in which it's defined. So, moral number 1 is that circular lists are useful; moral number 2 is not to try to trace a procedure in the evaluator that has an environment as an argument! The tracing mechanism will take forever to try to print the circular argument list.

CS 61A Lecture Notes Week 13

Topic: Concurrency

Reading: Abelson & Sussman, Section 3.4

To work with the ideas in this section you should first

```
(load "~cs61a/lib/serial.scm")
```

in order to get the necessary Scheme extensions.

Parallelism

Many things we take for granted in ordinary programming become problematic when there is any kind of parallelism involved. These situations include

- multiple processors (hardware) sharing data
- software multithreading (simulated parallelism)
- operating system input/output device handlers

This is the most important topic in CS 162, the operating systems course; here in 61A we give only a brief introduction, in the hope that when you see this topic for the second time it'll be clearer as a result.

To see in simple terms what the problem is, think about the Scheme expression

```
(set! x (+ x 1))
```

As you'll learn in more detail in 61C, Scheme translates this into a sequence of instructions to your computer. The details depend on the particular computer model, but it'll be something like this:

```
   lw   $8, x          ; Load a Word from memory location x
                       ; into processor register number 8.
   addi $8, 1          ; Add the Immediate value 1 to the register.
   sw   $8, x          ; Store the Word from register 8 back
                       ; into memory location x.
```

Ordinarily we would expect this sequence of instructions to have the desired effect. If the value of `x` was 100 before these instructions, it should be 101 after them.

But imagine that this sequence of three instructions can be interrupted by other events that come in the middle. To be specific, let's suppose that someone else is also trying to add 1 to `x`'s value. Now we might have this sequence:

```
my process                             other process
----------                             -------------

lw   $8, x   [value is 100]
addi $8, 1   [value is 101]
                                       lw   $9, x   [value is 100]
                                       addi $9, 1   [value is 101]
                                       sw   $9, x   [stores 101]
sw   $8, x   [stores 101]
```

The ultimate value of `x` will be 101, instead of the correct 102.

The general idea we need to solve this problem is the *critical section*, which means a sequence of instructions that mustn't be interrupted. The three instructions starting with the load and ending with the store are a critical section.

Actually, we don't have to say that these instructions can't be interrupted; the only condition we must enforce is that they can't be interrupted by another process that uses the variable `x`. It's okay if another process wants to add 1 to `y` meanwhile. So we'd like to be able to say something like

```
reserve x
lw      $8, x
addi    $8, 1
sw      $8, x
release x
```

Serializers

Computers don't really have instructions quite like `reserve` and `release`, but we'll see that they do provide similar mechanisms. For now, let's look at how this idea can be expressed at a higher level of abstraction, in a Scheme program.

```
(define x-protector (make-serializer))

(define protected-increment-x (x-protector (lambda () (set! x (+ x 1)))))

> x
100
> (protected-increment-x)
> x
101
```

We introduce an abstraction called a *serializer*. This is a procedure that takes as its argument another procedure (call it `proc`). The serializer returns a new procedure (call it `protected-proc`). When invoked, `protected-proc` invokes `proc`, but only if the *same* serializer is not already in use by another protected procedure. `Proc` can have any number of arguments, and `protected-proc` will take the same arguments and return the same value.

There can be many different serializers, all in operation at once, but each one can't be doing two things at once. So if we say

```
(define x-protector (make-serializer))
(define y-protector (make-serializer))

(parallel-execute (x-protector (lambda () (set! x (+ x 1))))
                  (y-protector (lambda () (set! y (+ y 1)))))
```

then both tasks can run at the same time; it doesn't matter how their machine instructions are interleaved. But if we say

```
(parallel-execute (x-protector (lambda () (set! x (+ x 1))))
                  (x-protector (lambda () (set! x (+ x 1)))))
```

then, since we're using the same serializer in both tasks, the serializer will ensure that they don't overlap in time.

I've introduced a new primitive procedure, `parallel-execute`. It takes any number of arguments, each of which is a procedure of no arguments, and invokes them them, in parallel rather than in sequence. (This isn't a standard part of Scheme, but an extension for this section of the textbook.)

You may be wondering about the need for all those `(lambda ()...)` notations. Since a serializer isn't a special form, it can't take an expression as argument. Instead we must give it a procedure that it can invoke.

Programming Considerations

Even with serializers, it's not easy to do a good job of writing programs that deal successfully with concurrency. In fact, all of the operating systems in widespread use today have bugs in this area; Unix systems, for example, are expected to crash every month or two because of concurrency bugs.

To make the discussion concrete, let's think about an airline reservation system, which serves thousands of simultaneous users around the world. Here are the things that can go wrong:

• **Incorrect results.** The worst problem is if the same seat is reserved for two different people. Just as in the case of adding 1 to `x`, the reservation system must first find a vacant seat, then mark that seat as occupied. That sequence of reading and then modifying the database must be protected.

• **Inefficiency.** One very simple way to ensure correct results is to use a single serializer to protect the entire reservation database, so that only one person could make a request at a time. But this is an unacceptable solution; thousands of people are waiting to reserve seats, mostly not for the same flight.

• **Deadlock.** Suppose that someone wants to travel to a city for which there is no direct flight. We must make sure that we can reserve a seat on flight A and a seat on connecting flight B on the same day, before we commit to either reservation. This probably means that we need to use *two* serializers at the same time, one for each flight. Suppose we say something like

```
(serializer-A (serializer-B (lambda () ...))))
```

Meanwhile someone else says

```
(serializer-B (serializer-A (lambda () ...))))
```

The timing could work out so that we get serializer A, the other person gets serializer B, and then we are each stuck waiting for the other one.

• **Unfairness.** This isn't an issue in every situation, but sometimes you want to avoid a solution to the deadlock problem that always gives a certain process priority over some other one. If the high-priority process is greedy, the lower-priority process might never get its turn at the shared data.

Implementing Serializers

A serializer is a high-level abstraction. How do we make it work? Here is an *incorrect* attempt to implement serializers:

```
;;;;;                         In file cs61a/lectures/3.4/bad-serial.scm
(define (make-serializer)
  (let ((in-use? #f))
    (lambda (proc)
      (define (protected-proc . args)
        (if in-use?
            (begin
             (wait-a-while)                 ; Never mind how to do that.
             (apply protected-proc args))   ; Try again.
            (begin
             (set! in-use #t)        ; Don't let anyone else in.
             (apply proc args)       ; Call the original procedure.
             (set! in-use #f))))     ; Finished, let others in again.
      protected-proc)))
```

This is a little complicated, so concentrate on the important parts. In particular, never mind about the *scheduling* aspect of parallelism—how we can ask this process to wait a while before trying again if the serializer is already in use. And never mind the stuff about `apply`, which is needed only so that we can serialize procedures with any number of arguments.

The part to focus on is this:

```
        (if in-use?
            .......         ; wait and try again
            (begin
             (set! in-use #t)        ; Don't let anyone else in.
             (apply proc args)       ; Call the original procedure.
             (set! in-use #f)))      ; Finished, let others in again.
```

The intent of this code is that it first checks to see if the serializer is already in use. If not, we claim the serializer by setting `in-use` true, do our job, and then release the serializer.

The problem is that this sequence of events is subject to the same parallelism problems as the procedure we're trying to protect! What if we check the value of `in-use`, discover that it's false, and right at that moment another process sneaks in and grabs the serializer? In order to make this work we'd have to have another serializer protecting this one, and a third serializer protecting the second one, and so on.

There is no way to avoid this problem by clever programming tricks within the competing processes. We need help at the level of the underlying machinery that provides the parallelism: the hardware and/or the operating system. That underlying level must provide a *guaranteed atomic* operation with which we can test the old value of `in-use` and change it to a new value with no possibility of another process intervening.

The textbook assumes the existence of a procedure called `test-and-set!` with this guarantee of atomicity. Although there is a pseudo-implementation on page 312, that procedure won't really work, for the same reason that my pseudo-implementation of `make-serializer` won't work. What you have to imagine is that `test-and-set!` is a single instruction in the computer's hardware, comparable to the Load Word instructions and so on that I started with. (This is a realistic assumption; modern computers do provide some such hardware mechanism, precisely for the reasons we're discussing now.)

The Mutex

The book uses an intermediate level of abstraction between the serializer and the atomic hardware capability, called a *mutex*. What's the difference between a mutex and a serializer? The serializer provides, as an abstraction, a protected operation, without requiring the programmer to think about the mechanism by which it's protected. The mutex exposes the sequence of events. Just as my incorrect implementation said

```
    (set! in-use #t)
    (apply proc args)
    (set! in-use #f)
```

the correct version uses a similar sequence

```
    (mutex 'acquire)
    (apply proc args)
    (mutex 'release)
```

By the way, all of the versions in these notes have another bug; I've simplified the discussion by ignoring the problem of return values. We want the value returned by `protected-proc` to be the same as the value returned by the original `proc`, even though the call to `proc` isn't the last step. Therefore the correct implementation is

```
    (mutex 'acquire)
    (let ((result (apply proc args)))
      (mutex 'release)
      result)
```

as in the book's implementation on page 311.

Note: The first part of programming project 4 is this week.

CS 61A Lecture Notes Week 14

Topic: Streams

Reading: Abelson & Sussman, Section 3.5.1–3, 3.5.5

Streams are an abstract data type, not so different from rational numbers, in that we have constructors and selectors for them. But we use a clever trick to achieve tremendously magical results. As we talk about the mechanics of streams, there are three big ideas to keep in mind:

- Efficiency: Decouple order of evaluation from the form of the program.
- Infinite data sets.
- Functional representation of time-varying information (versus OOP).

You'll understand what these all mean after we look at some examples.

How do we tell if a number n is prime? Never mind computers, how would you express this idea as a mathematician? Something like this: "N is prime if it has no factors in the range $2 \le f < n$."

So, to implement this on a computer, we should

- Get all the numbers in the range $[2, n-1]$.
- See which of those are factors of n.
- See if the result is empty.

```
;;;;;                         In file cs61a/lectures/3.5/prime1.scm
(define (prime? n)
  (null? (filter (lambda (x) (= (remainder n x) 0))
                 (enumerate-interval 2 (- n 1)))))
```

But we don't usually program it that way. Instead, we write a *loop*:

```
;;;;;                         In file cs61a/lectures/3.5/prime0.scm
(define (prime? n)
  (define (iter factor)
    (cond ((= factor n) #t)
          ((= (remainder n factor) 0) #f)
          (else (iter (+ factor 1)))))
  (iter 2))
```

(Never mind that we can make small optimizations like only checking for factors up to $\sqrt{n}$. Let's keep it simple.)

Why don't we write it the way we expressed the problem in words? The problem is one of efficiency. Let's say we want to know if 1000 is prime. We end up constructing a list of 998 numbers and testing *all* of them as possible factors of 1000, when testing the first possible factor would have given us a false result quickly.

The idea of streams is to let us have our cake and eat it too. We'll write a program that *looks like* the first version, but *runs like* the second one. All we do is change the second version to use the stream ADT instead of the list ADT:

```
;;;;;                          In file cs61a/lectures/3.5/prime2.scm
(define (prime? n)
  (stream-null? (stream-filter (lambda (x) (= (remainder n x) 0))
                               (stream-enumerate-interval 2 (- n 1)))))
```

The only changes are `stream-enumerate-interval` instead of `enumerate-interval`, `stream-null?` instead of `null?`, and `stream-filter` instead of `filter`.

How does it work? A list is implemented as a pair whose `car` is the first element and whose `cdr` is the rest of the elements. A stream is almost the same: It's a pair whose `car` is the first element and whose `cdr` is a *promise* to compute the rest of the elements later.

For example, when we ask for the range of numbers $[2, 999]$ what we get is a single pair whose `car` is 2 and whose `cdr` is a promise to compute the range $[3, 999]$. The function `stream-enumerate-interval` returns that single pair. What does `stream-filter` do with it? Since the first number, 2, does satisfy the predicate, `stream-filter` returns a single pair whose `car` is 2 and whose `cdr` is a promise *to filter* the range $[3, 999]$. `Stream-filter` returns that pair. So far no promises have been "cashed in." What does `stream-null?` do? It sees that its argument stream contains the number 2, and maybe contains some more stuff, although maybe not. But at least it contains the number 2, so it's not empty. `Stream-null?` returns `#f` right away, without computing or testing any more numbers.

Sometimes (for example, if the number we're checking *is* prime) you do have to cash in the promises. If so, the stream program still follows the same order of events as the original loop program; it tries one number at a time until either a factor is found or there are no more numbers to try.

Summary: What we've accomplished is to decouple the form of a program—the order in which computations are presented—from the actual order of evaluation. This is one more step on the long march that this whole course is about, i.e., letting us write programs in language that reflects the problems we're trying to solve instead of reflecting the way computers work.

• Implementation. How does it work? The crucial point is that when we say something like

```
(cons-stream from (stream-enumerate-interval (+ from 1) to))
```

(inside `stream-enumerate-interval`) we can't actually evaluate the second argument to `cons-stream`. That would defeat the object, which is to defer that evaluation until later (or maybe never). Therefore, `cons-stream` has to be a special form. It has to `cons` its first argument onto a promise to compute the second argument. The expression

```
(cons-stream a b)
```

is equivalent to

```
(cons a (delay b))
```

`Delay` is itself a special form, the one that constructs a promise. Promises could be a primitive data type, but since this is Scheme, we can represent a promise as a function. So the expression

```
(delay b)
```

really just means

```
(lambda () b)
```

We use the promised expression as the body of a function with no arguments. (A function with no arguments is called a *thunk*.)

Once we have this mechanism, we can use ordinary functions to redeem our promises:

```
(define (force promise) (promise))
```

and now we can write the selectors for streams:

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

Notice that forcing a promise doesn't compute the entire rest of the job at once, necessarily. For example, if we take our range $[2, 999]$ and ask for its tail, we don't get a list of 997 values. All we get is a pair whose `car` is 3 and whose `cdr` is a new promise to compute $[4, 999]$ later.

The name for this whole technique is *lazy evaluation* or *call by need*.

• Reordering and functional programming. Suppose your program is written to include the following sequence of steps:

```
...
(set! x 2)
...
(set! y (+ x 3))
...
(set! x 7)
...
```

Now suppose that, because we're using some form of lazy evaluation, the actual sequence of events is reordered so that the third `set!` happens before the second one. We'll end up with the wrong value for `y`. This example shows that we can only get away with below-the-line reordering if the above-the-line computation is functional.

(Why isn't it a problem with `let`? Because `let` doesn't mutate the value of one variable in one environment. It sets up a local environment, and any expression within the body of the `let` has to be computed within that environment, even if things are reordered.)

• Infinite streams. Think about the plain old list function

```
(define (enumerate-interval from to)
  (if (> from to)
      '()
      (cons from (enumerate-interval (+ from 1) to)) ))
```

When we change this to a stream function, we change very little in the appearance of the program:

```
(define (stream-enumerate-interval from to)
  (if (> from to)
      THE-EMPTY-STREAM
      (cons-STREAM from (stream-enumerate-interval (+ from 1) to)) ))
```

but this tiny above-the-line change makes an enormous difference in the actual behavior of the program.

Now let's cross out the second argument and the end test:

```
(define (stream-enumerate-interval from)
  (cons-stream from (stream-enumerate-interval (+ from 1))) )
```

This is an *enormous* above-the-line change! We now have what looks like a recursive function with no base case—an infinite loop. And yet there is hardly any difference at all in the actual behavior of the program. The old version computed a range such as $[2, 999]$ by constructing a single pair whose `car` is 2 and whose `cdr` is a promise to compute $[3, 999]$ later. The new version computes a range such as $[2, \infty]$ by constructing a single pair whose `car` is 2 and whose `cdr` is a promise to compute $[3, \infty]$ later!

This amazing idea lets us construct even some pretty complicated infinite sets, such as the set of all the prime numbers. (Explain the sieve of Eratosthenes. The program is in the book so it's not reproduced here.)

• Time-varying information. Functional programming works great for situations in which we are looking for a timeless answer to some question. That is, the same question always has the same answer regardless of events in the world. We invented OOP because functional programming didn't let us model changing state. But with streams we *can* model state functionally. We can say

```
(define (user-stream)
  (cons-stream (read) (user-stream)) )
```

and this gives us *the stream of everything the user is going to type* from now on. Instead of using local state variables to remember the effect of each thing the user types, one at a time, we can write a program that computes the result of the (possibly infinite) collection of user requests all at once! This feels really bizarre, but it does mean that purely functional programming languages can handle user interaction. We don't *need* OOP.

Note: The second part of programming project 4 is this week.

CS 61A   Lecture Notes   Week 15

Topic: Logic programming

Reading: Abelson & Sussman, Section 4.4.1–3

This week's big idea is *logic programming* or *declarative programming.*

It's the biggest step we've taken away from expressing a computation in hardware terms. When we discovered streams, we saw how to express an algorithm in a way that's independent of the *order* of evaluation. Now we are going to describe a computation in a way that has no (visible) algorithm at all!

We are using a logic programming language that A&S implemented in Scheme. Because of that, the notation is Scheme-like, i.e., full of lists. Standard logic languages like Prolog have somewhat different notations, but the idea is the same.

All we do is assert facts:

```
> (load "~cs61a/lib/query.scm")
> (query)

;;; Query input:
(assert! (Brian likes potstickers))
```

and ask questions about the facts:

```
;;; Query input:
(?who likes potstickers)

;;; Query results:
(BRIAN LIKES POTSTICKERS)
```

Although the assertions and the queries take the form of lists, and so they look a little like Scheme programs, they're not! There is no application of function to argument here; an assertion is just data.

This is true even though, for various reasons, it's traditional to put the verb (the *relation*) first:

```
(assert! (likes Brian potstickers))
```

We'll use that convention hereafter, but that makes it even easier to fall into the trap of thinking there is a *function* called `likes`.

• Rules. As long as we just tell the system isolated facts, we can't get extraordinarily interesting replies. But we can also tell it *rules* that allow it to infer one fact from another. For example, if we have a lot of facts like

```
(mother Eve Cain)
```

then we can establish a rule about grandmotherhood:

```
(assert! (rule (grandmother ?elder ?younger)
               (and (mother ?elder ?mom)
                    (mother ?mom ?younger) )))
```

The rule says that the first part (the conclusion) is true *if* we can find values for the variables such that the second part (the condition) is true.

Again, resist the temptation to try to do composition of functions!

```
(assert! (rule (grandmother ?elder ?younger)         ;; WRONG!!!!
               (mother ?elder (mother ?younger)) ))
```

`Mother` isn't a function, and you can't ask for the mother of someone as this incorrect example tries to do. Instead, as in the correct version above, you have to establish a variable (`?mom`) that has a value that satisfies the two motherhood relationships we need.

In this language the words `assert!`, `rule`, `and`, `or`, and `not` have special meanings. Everything else is just a word that can be part of assertions or rules.

Once we have the idea of rules, we can do real magic:

```
;;;;;                         In file cs61a/lectures/4.4/logic-utility.scm
(assert! (rule (append (?u . ?v) ?y (?u . ?z))
               (append ?v ?y ?z)))

(assert! (rule (append () ?y ?y)))
```

(The actual online file uses a Scheme procedure `aa` to add the assertion. It's just like saying `assert!` to the query system, but you say it to Scheme instead. This lets you `load` the file. Don't get confused about this small detail—just ignore it.)

```
;;; Query input:
(append (a b) (c d e) ?what)

;;; Query results:
(APPEND (A B) (C D E) (A B C D E))
```

So far this is just like what we could do in Scheme.

```
;;; Query input:
(append ?what (d e) (a b c d e))

;;; Query results:
(APPEND (A B C) (D E) (A B C D E))

;;; Query input:
(append (a) ?what (a b c d e))

;;; Query results:
(APPEND (A) (B C D E) (A B C D E))
```

The new thing in logic programming is that we can run a "function" backwards! We can tell it the answer and get back the question. But the real magic is...

```
;;; Query input:
(append ?this ?that (a b c d e))

;;; Query results:
(APPEND () (A B C D E) (A B C D E))
(APPEND (A) (B C D E) (A B C D E))
(APPEND (A B) (C D E) (A B C D E))
(APPEND (A B C) (D E) (A B C D E))
(APPEND (A B C D) (E) (A B C D E))
(APPEND (A B C D E) () (A B C D E))
```

We can use logic programming to compute multiple answers to the same question! Somehow it found all the possible combinations of values that would make our query true.

How does the `append` program work? Compare it to the Scheme `append`:

```
(define (append a b)
  (if (null? a)
      b
      (cons (car a) (append (cdr a) b)) ))
```

Like the Scheme program, the logic program has two cases: There is a base case in which the first argument is empty. In that case the combined list is the same as the second appended list. And there is a recursive case in which we divide the first appended list into its `car` and its `cdr`. We reduce the given problem into a problem about appending `(cdr a)` to `b`. The logic program is different in form, but it says the same thing. (Just as, in the grandmother example, we had to give the mother a name instead of using a function call, here we have to give `(car a)` a name—we call it `?u`.)

Unfortunately, this "working backwards" magic doesn't always work.

```
;;;;;                        In file cs61a/lectures/4.4/reverse.scm
(assert! (rule (reverse (?a . ?x) ?y)
               (and (reverse ?x ?z)
                    (append ?z (?a) ?y) )))

(assert! (reverse () ()))
```

This works for `(reverse (a b c) ?what)` but not the other way around; it gets into an infinite loop. We can also write a version that works *only* backwards:

```
;;;;;                        In file cs61a/lectures/4.4/reverse.scm
(assert! (rule (backward (?a . ?x) ?y)
               (and (append ?z (?a) ?y)
                    (backward ?x ?z) )))

(assert! (backward () ()))
```

But it's much harder to write one that works both ways. Even as we speak, logic programming fans are trying to push the limits of the idea, but right now, you still have to understand something about the below-the-line algorithm to be confident that your logic program won't loop.

• Below-the-line implementation.

Think about `eval` in the MC evaluator. It takes two arguments, an expression and an environment, and it returns the value of the expression.

In logic programming, there's no such thing as "the value of the expression." What we're given is a query, and there may or may not be some number of variable bindings that make the query true. The query evaluator `qeval` is analogous to `eval` in that it takes two arguments, something to evaluate and a context in which to work. But the thing to evaluate is a query, not an expression; the context isn't just one environment but a whole collection of environments—one for each set of variable values that satisfy some previous query. And the result returned by `qeval` isn't a value. It's a new collection of environments! It's as if `eval` returned an environment instead of a value.

The "collection" of environments we're talking about here is represented as a stream. That's because there might be infinitely many of them! We use the stream idea to reorder the computation; what really happens is that we take one potential set of satisfying values and work it all the way through; then we try another potential set of values. But the program looks as if we compute all the satisfying values at once for each stage of a query.

Just as every top-level Scheme expression is evaluated in the global environment, every top-level query is evaluated in a stream containing a single empty environment. (No variables have been assigned values yet.)

If we have a query like `(and p q)`, what happens is that we recursively use `qeval` to evaluate `p` in the empty-environment stream. The result is a stream of variable bindings that satisfy `p`. Then we use `qeval` to evaluate `q` in that result stream! The final result is a stream of bindings that satisfy `p` and `q` simultaneously.

If the query is `(or p q)` then we use `qeval` to evaluate each of the pieces independently, starting in both cases with the empty-environment stream. Then we *merge* the two result streams to get a stream of bindings that satisfy either `p` or `q`.

If the query is `(not q)`, we can't make sense of that unless we already have a stream of environments to work with. That's why we can only use `not` in a context such as `(and p (not q))`. We take the stream of environments that we already have, and we `stream-filter` that stream, using as the test predicate the function

```
(lambda (env) (stream-null? (qeval q env)))
```

That is, we keep only those environments for which we *can't* satisfy `q`.

That explains how `qeval` reduces compound queries to simple ones. How do we evaluate a simple query? The first step is to *pattern match* the query against every assertion in the data base. Pattern matching is just like the recursive `equal?` function, except that a variable in the pattern (the query) matches anything in the assertion. (But if the same variable appears more than once, it must match the same thing each time. That's why we need to keep an environment of matches so far.)

The next step is to match the query against the *conclusions* of rules. This is tricky because now there can be variables in both things being matched. Instead of the simple pattern matching we have to use a more complicated version called *unification.* (See the details in the text.) If we find a match, then we take the condition part of the rule (the body) and use that as a new query, to be satisfied within the environment(s) that `qeval` gave us when we matched the conclusion. In other words, first we look at the conclusion to see whether this rule can possibly be relevant to our query; if so, we see if the conditions of the rule are true.

Here's an example, partly traced:

```
;;; Query input:
(append ?a ?b (aa bb))

(unify-match (append ?a ?b (aa bb))         ; MATCH ORIGINAL QUERY
             (append () ?1y ?1y)            ; AGAINST BASE CASE RULE
             ())                            ; WITH NO CONSTRAINTS

RETURNS: ((?1y . (aa bb)) (?b . ?1y) (?a . ()))
PRINTS:  (append () (aa bb) (aa bb))
```

Since the base-case rule has no body, once we've matched it, we can print a successful result. (Before printing, we have to look up variables in the environment so what we print is variable-free.)

Now we unify the original query against the conclusion of the other rule:

```
(unify-match (append ?a ?b (aa bb))                ; MATCH ORIGINAL QUERY
             (append (?2u . ?2v) ?2y (?2u . ?2z))  ; AGAINST RECURSIVE RULE
             ())                                   ; WITH NO CONSTRAINTS

RETURNS: ((?2z . (bb)) (?2u . aa) (?b . ?2y) (?a . (?2u . ?2v)))
         [call it F1]
```

This was successful, but we're not ready to print anything yet, because we now have to take the body of that rule as a new query. Note the indenting to indicate that this call to `unify-match` is within the pending rule.

```
    (unify-match (append ?2v ?2y ?2z)   ; MATCH BODY OF RECURSIVE RULE
                 (append () ?3y ?3y)    ; AGAINST BASE CASE RULE
                 F1)                    ; WITH CONSTRAINTS FROM F1

    RETURNS: ((?3y . (bb)) (?2y . ?3y) (?2v . ()) [plus F1])
    PRINTS:  (append (aa) (bb) (aa bb))

    (unify-match (append ?2v ?2y ?2z)                  ; MATCH SAME BODY
                 (append (?4u . ?4v) ?4y (?4u . ?4z))  ; AGAINST RECURSIVE RULE
                 F1)                                   ; WITH F1 CONSTRAINTS

    RETURNS: ((?4z . ()) (?4u . bb) (?2y . ?4y) (?2v . (?4u . ?4v))
              [plus F1])  [call it F2]

        (unify-match (append ?4v ?4y ?4z)   ; MATCH BODY FROM NEWFOUND MATCH
                     (append () ?5y ?5y)    ; AGAINST BASE CASE RULE
                     F2)                    ; WITH NEWFOUND CONSTRAINTS

        RETURNS: ((?5y . ()) (?4y . ?5y) (?4v . ()) [plus F2])
        PRINTS:  (append (aa bb) () (aa bb))

        (unify-match (append ?4v ?4y ?4z)                  ; MATCH SAME BODY
                     (append (?6u . ?6v) ?6y (?6u . ?6z))  ; AGAINST RECUR RULE
                     F2)                                   ; SAME CONSTRAINTS

        RETURNS: ()                                        ; BUT THIS FAILS
```

CS 61A Lecture Notes Week 16

Topic: Review

Reading: No new reading; study for the final.

• Go over first-day handout about abstraction; show how each topic involves an abstraction barrier and say what's above and what's below the line.

• Go over the big ideas within each programming paradigm:

Functional Programming:
composition of functions
first-class functions (function as object)
higher-order functions
recursion
delayed (lazy) evaluation
(vocabulary: parameter, argument, scope, iterative process)

Object-Oriented Programming:
actors
message passing
local state
inheritance
identity vs. equal value
(vocabulary: dispatch procedure, delegation, mutation)

Logic Programming:
focus on ends, not means
multiple solutions
running a program backwards
(vocabulary: pattern matching, unification)

• Review where 61A fits into the curriculum. (See the CS abstraction hierarchy in week 1.)

Please, please, don't forget the ideas of 61A just because you're not programming in Scheme!