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)
- Client-server programming (1 week)
- 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.

• **Unix Shell Programming**

[This topic may be in week 1, week 4, or week 11 depending on the holidays each semester.]

Sometimes the best way to solve a programming problem is not to write a program at all, but instead to glue together existing programs that solve the problem.

As an example, we'll construct a spelling-checker program. It will take a text file as input, and will generate a list of words that are in the file but not in the online dictionary.

For this demonstration I'll use the file named `summary` as the text I want to spell-check.

We are given a file that contains several words per line, including things we don't want to compare against the dictionary, such as spaces and punctuation. Our job will be easier if we transform the input into a file with exactly one word per line, with no spaces or punctuation (except that we'll keep apostrophes, which are part of words — contractions such as "we'll" — rather than word delimiters).

```
tr -d '.,;:"\!\[]()' < summary > nopunct
```

`Tr` is a Unix utility program that translates one character into another. In this case, because of the `-d` switch, we are asking it to delete from the input any periods, commas, semicolons, colons, exclamation points, braces, and parentheses.

The single-quote (`'`) characters are used to tell the shell that the characters between them should not be interpreted according to the shell's usual syntax. For example, a semicolon in a shell command line separates two distinct commands; the shell does one and then does the other. Here we want the semicolon to be passed to the `tr` program just as if it were a letter of the alphabet.

When a program asks to read or print text, without specifying a particular source or destination, its input comes from something called the *standard input*; its output goes to the *standard output.* Ordinarily, the standard input is the keyboard and the standard output is the screen. But you can *redirect* them using the characters `<` and `>` in shell commands. In this case, we are asking the shell to connect `tr`'s standard input to the file named `summary`, and to connect its standard output to a new file named `nopunct`.

Spacing characters in text files include the space character and the tab character. To simplify things, let's translate tabs to spaces.

```
tr '    ' ' ' < nopunct > notab
```

Between the first pair of single-quotes is a tab character. `Tr` will translate every tab in the file to a space.

We really want one word per line, so let's translate spaces to newline characters.

```
tr ' ' '\n' < notab > oneword
```

The notation `\n` is a standard Unix notation for the newline character.

In English text, we capitalize the first word of each sentence. The words in the dictionary aren't capitalized (unless they're proper nouns). So let's translate capital letters to lower case:

```
tr '[A-Z]' '[a-z]' < oneword > lowcase
```

The notation `[A-Z]` means all the characters between the given extremes; it's equivalent to

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

What `tr` does is convert every instance of the nth character of its first argument into the nth character of the second argument. So we are asking it to convert `A` to `a`, `B` to `b`, and so on.

Our plan is to compare each word of the text against the words in the dictionary. But we don't have to read every word of the dictionary for each word of the file; since the dictionary is sorted alphabetically, if we sort the words in our text file, we can just make one pass through both files in parallel.

```
sort < lowcase > sorted
```

The `sort` program can take arguments to do sorting in many different ways; for example, you can ask it to sort the lines of a file based on the third word of each line. But in this case, we want the simplest possible sort: The "sort key" is the entire line, and we're sorting in character-code order (which is the same as alphabetical order since we eliminated capital letters).

Common words like "the" will occur many times in our text. There's no need to spell-check the same word repeatedly. Since we've sorted the file, all instances of the same word are next to each other in the file, so we can ask Unix to eliminate consecutive equal lines:

```
uniq < sorted > nodup
```

`Uniq` has that name because in its output file every line is unique; it eliminates (consecutive) duplicates.

Now we're ready to compare our file against the dictionary. This may seem like a special-purpose operation for which we'll finally have to write our own program, but in fact what we want to do is a common database operation, combining entries from two different databases that have a certain element in common. Our application is a trivial one in which each "database" entry has only one element, and we are just checking for matches.

This combining of databases is called a *join* in the technical terminology of database programming. In our case, what we want is not the join itself, but rather a report of those elements of one database (our text file) that don't occur in the other database (the online dictionary). This is what the `-v1` switch means:

```
join -v1 nodup words > errors
```

That's it! We now have a list of misspelled words that we can correct in Emacs. The user interface of this spelling program isn't fancy, but it gets the job done, and we didn't have to work hard to get it.

It's a little ugly that we've created all these intermediate files. But we don't have to, because the shell includes a "pipe" feature that lets you connect the standard output of one program to the standard input of another. So we can do it this way:

```
tr -d '.,;:"\!\[]()' | tr '    ' ' ' | tr ' ' '\n' \
  | tr '[A-Z]' '[a-z]' | sort | uniq | join -v1 - words
```

The backslash at the end of the first line is the shell's *continuation* character, telling it that the same command continues on the next line. The minus sign (`-`) as the second argument to `join` tells it to take its first database input from its standard input rather than from a named file. (This is a standard Unix convention for programs that read more than one input file.)

We can write a file named `spell` containing this command line, mark the file as executable (a program, rather than data) with the command

```
chmod +x spell
```

and then instead of the sequence of commands I used earlier we can just say

```
spell < summary > errors
```

What we've done is use existing programs, glued together in a "scripting language" (the Unix shell), to solve a new problem with very little effort. This is called "code re-use." The huge collection of Unix utility programs have been written with this technique in mind; that's why almost all of them are what the Unix designers call "filters," meaning programs that take a text stream as input and produce a modified text stream as output. Each filter doesn't care where its input and output are; they can be files, or they can be pipes connected to another filter program.

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 CS 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: Data abstraction, sequences

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

Note: The first midterm is next week.

• 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.

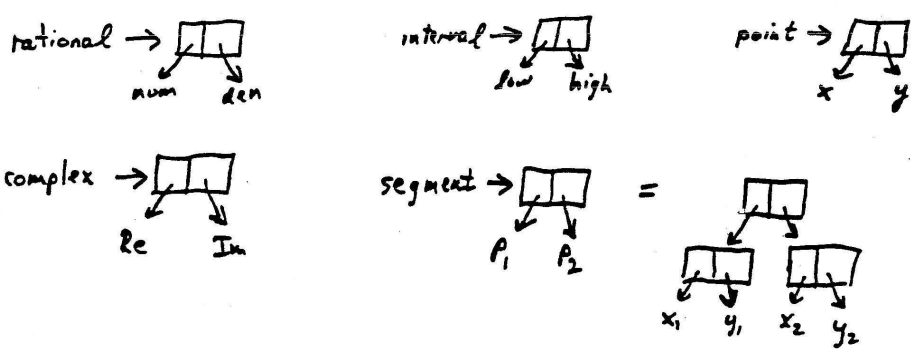
Data abstraction is a really good idea because it helps keep you from getting confused when you're dealing with lots of data types, but don't get religious about it. For example, we have invented the *sentence* data type for this course. We have provided symmetric selectors `first` and `last`, and symmetric 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:

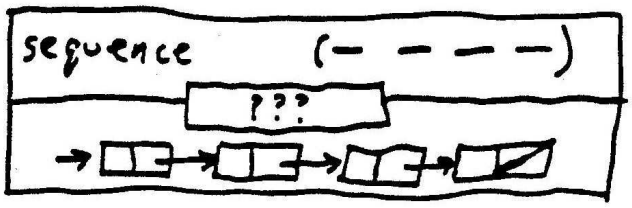
```
;;;;;                          In file cs61a/lectures/2.1/cons.scm
(define (cons x y)
  (lambda (which)
    (cond ((equal? which 'car) x)
          ((equal? which 'cdr) y)
          (else (error "Bad message to CONS" message)) )))


(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.

• List constructors.

Sentences have a very simple structure, so there's just one constructor for them. Lists are more complicated, and have three constructors:

`List` is the simplest to understand. It takes any number of arguments, each of which can be anything, and returns a list containing those arguments as its elements:

```
> (list '(a list) 'word 87 #t)
((a list) word 87 #t)
```

This seems very straightforward, but in practice it's not the most useful constructor, because it can be used only when you know exactly how many elements you want in the list. Most list programming deals with arbitrary sequences, which could be of any length. `List` is useful when you use a fixed-length list to represent an abstract data type, such as a point in three-dimensional space:

```
(define (point x y z)
  (list x y z))
```

`Cons` adds one new element at the front of an existing list:

```
> (cons '(new element) '(the old list))
((new element) the old list)
```

(Of course, `cons` really just takes two arguments and makes a pair containing them, but if you're using that pair as the head of a list, then the effect of `cons` in terms of the list abstraction is to add one new element.) This may seem too specific and arbitrary to be useful, but in fact `cons` is the most commonly used list constructor, because adding one new element is exactly what you want to do in a recursive transformation of a list:

```
(define (map fn seq)
  (if (null? seq)
      '()
      (CONS (fn (car seq))
            (map fn (cdr seq)))))
```

`Append` is used to combine two or more lists in a way that "flattens" some of the structure of the result: It returns a list whose elements are *the elements of* the arguments, which must be lists:

```
> (append '(one list) '(and another list))
(one list and another list)
```

It's most useful when combining results from multiple recursive calls, each of which returns a subset of the overall answer; you want to take the union of those sets, and that's what `append` does.

• 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.

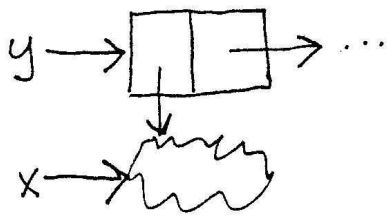
• Box and pointer diagrams.

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.



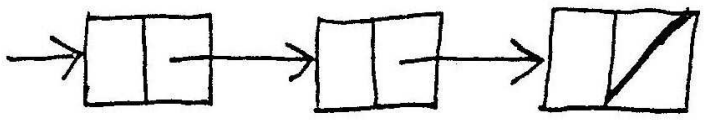
2. The direction of arrows (up, down, left, right) is irrelevant. You can draw them however you want to make the arrangement of pairs neat. That's why it's crucial not to forget the arrowheads!

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.

• MapReduce

In the past, functional programming, and higher-order functions in particular, have been considered esoteric and unimportant by most programmers. But the advent of highly parallel computation is changing that, because functional programming has the very useful property that the different pieces of a program don't interfere with each other, so it doesn't matter in what order they are invoked. Later this semester, when we have more sophisticated functional mechanisms to work with, we'll be examining one famous example of functional programming at work: the `MapReduce` programming paradigm developed by Google that uses higher-order functions to allow a programmer to process large amount of data in parallel on many computers.

Much of the computing done at Google consists of relatively simple algorithms applied to massive amounts of data, such as the entire World Wide Web. It's routine for them to use clusters consisting of many thousands of processors, all running the same program, with a distributed filesystem that gives each processor local access to part of the data.

In 2003 some very clever people at Google noticed that the majority of these computations could be viewed as a `map` of some function over the data followed by an `accumulate` (they use the name `reduce`, which is a synonym for this function) to collect the results. Although each program was conceptually simple, a lot of programmer effort was required to manage the parallelism; every programmer had to worry about things like how to recover from a processor failure (virtually certain to happen when a large computation uses thousands of machines) during the computation. They wrote a library procedure named `MapReduce` that basically takes two functions as arguments, a one-argument function for the `map` part and a two-argument function for the `accumulate` part. (The actual implementation is more complicated, but this is the essence of it.) Thus, only the implementors of `MapReduce` itself had to worry about the parallelism, and application programmers just have to write the two function arguments.*

`MapReduce` is a little more complicated than just

```
(define (mapreduce mapper reducer base-case data)    ; Nope.
  (accumulate reducer base-case (map mapper data)))
```

because of the parallelism. The input data comes in pieces; several computers run the `map` part in parallel, and each of them produces some output. These intermediate results are rearranged into groups, and each computer does a `reduce` of part of the data. The final result isn't one big list, but separate output files for each reducing process.

To make this a little more specific, today we'll see a toy version of the algorithm that just handles small data lists in one processor.

Data pass through the program in the form of *key-value pairs:*

```
(define make-kv-pair cons)
(define kv-key car)
(define kv-value cdr)
```

A list of key-value pairs is called an *association list* or *a-list* for short. We'll see a-lists in many contexts other than `MapReduce`. Conceptually, the input to `MapReduce` is an a-list, although in practice there are several a-lists, each on a different processor.

Any computation in `MapReduce` involves two function arguments: the *mapper* and the *reducer*. (Note: The Google `MapReduce` paper in the course reader says "the `map` function" to mean the function that the user writes, the one that's applied to each datum; this usage is confusing since everyone else uses "`map`" to mean

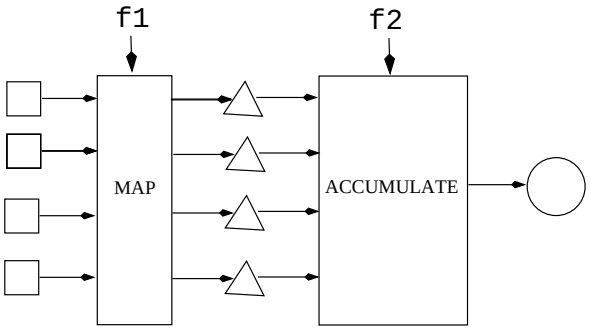
* We are indebted to Google for funding both the development of software and curriculum for this piece of the course and the cost of the cluster of parallel processors you'll use in the lab later. This should not be taken as an endorsement of Google's contemptible business practices, in which their income depends on massive violations of their users' privacy.

the higher-order function that controls the invocation of the user's function, so we're calling the latter the *mapper*:

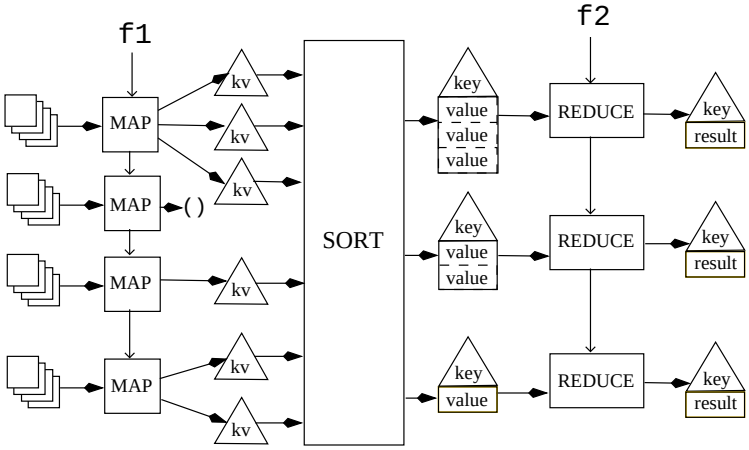
```
(map mapper data)
```

Similarly, we'll use `reduce` to refer to the higher-order function, and `reducer` to mean the user's accumulation function.)

`(accumulate f2 base (map f1 data))`



`(mapreduce f1 f2 base dataname)`



The argument to the mapper is always one kv-pair. Keys are typically used to keep track of where the data came from. For example, if the input consists of a bunch of Web pages, the keys might be their URLs. Another example we'll be using is Project Gutenberg, an online collection of public-domain books; there the key would be the name of a book (more precisely, the filename of the file containing that book). In most uses of a-lists, there will only be one kv-pair with a given key, but that's not true here; for example, each line of text in a book or Web page might be a datum, and every line in the input will have the same key.

The value returned by the mapper must be *a list of* kv-pairs. The reason it's a list instead of a single kv-pair, as you might expect, is twofold. First, a single input may be split into smaller pieces; for example, a line of text might be mapped into a separate kv-pair for each word in the line. Second, the mapper might return an empty list, if this particular kv-pair shouldn't contribute to the result at all; thus, the mapper might also be viewed as a filterer. The mapper is not required to use the same key in its output kv-pairs that it gets in its input kv-pair.

Since `map` handles each datum independently of all the others, the fact that many `map`s are running in parallel doesn't affect the result; we can model the entire process with a single `map` invocation. That's not the case with the `reduce` part, because the data are being combined, so it matters which data end up on which machine. This is where the keys are most important. Between the mapping and the reduction is an intermediate step in which the kv-pairs are sorted based on the keys, and all pairs with the same key are reduced together. Therefore, the reducer doesn't need to look at keys at all; its two arguments are a value and the result of the partial accumulation of values already done. In many cases, just as in the accumulations we've seen earlier, the reducer will be a simple associative and commutative operation such as `+`.

The overall result is an a-list, in which each key occurs only once, and the value paired with that key is the result of the `reduce` invocation that handled that key. The keys are guaranteed to be in order. (This is the result of the 61A version of `MapReduce`; the real Google software has a more complicated interface because each computer in the cluster collects its own `reduce` results, and there are many options for how the reduction tasks are distributed among the processors. You'll learn more details in 61B.) So in today's single-processor simulation, instead of talking about `reduce` we'll use a higher order function called `groupreduce` that takes a *list of a-lists* as argument, with each sublist having kv-pairs with the same key, does a separate reduction for each sublist, and returns an a-list of the results. So a complete `MapReduce` operation works

roughly like this:

```
(define (mapreduce mapper reducer base-case data) ; handwavy approximation
  (groupreduce reducer base-case
               (sort-into-buckets (map mapper data))))

(define (groupreduce reducer base-case buckets)
  (map (lambda (subset) (make-kv-pair
                         (kv-key (car subset))
                         (reduce reducer base-case (map kv-value subset))))
       buckets))
```

As a first example, we'll take some grades from various exams and add up the grades for each student. This example doesn't require `map`. Here's the raw data:

```
(define mt1 '((cs61a-xc . 27) (cs61a-ya . 40) (cs61a-xw . 35)
              (cs61a-xd . 38) (cs61a-yb . 29) (cs61a-xf . 32)))
(define mt2 '((cs61a-yc . 32) (cs61a-xc . 25) (cs61a-xb . 40)
              (cs61a-xw . 27) (cs61a-yb . 30) (cs61a-ya . 40)))
(define mt3 '((cs61a-xb . 32) (cs61a-xk . 34) (cs61a-yb . 30)
              (cs61a-ya . 40) (cs61a-xc . 28) (cs61a-xf . 33)))
```

Each midterm in this toy problem corresponds to the output of a parallel `map` operation in a real problem.

First we combine these into one list, and use that as input to the `sortintobuckets` procedure:

```
> (sort-into-buckets (append mt1 mt2 mt3))
(((cs61a-xb . 40) (cs61a-xb . 32))
 ((cs61a-xc . 27) (cs61a-xc . 25) (cs61a-xc . 28))
 ((cs61a-xd . 38))
 ((cs61a-xf . 32) (cs61a-xf . 33))
 ((cs61a-xk . 34))
 ((cs61a-xw . 35) (cs61a-xw . 27))
 ((cs61a-ya . 40) (cs61a-ya . 40) (cs61a-ya . 40))
 ((cs61a-yb . 29) (cs61a-yb . 30) (cs61a-yb . 30))
 ((cs61a-yc . 32)))
```

In the real parallel context, instead of the `append`, each `map` process would sort its own results into the right buckets, so that too would happen in parallel.

Now we can use `groupreduce` to add up the scores in each bucket separately:

```
> (groupreduce + 0 (sort-into-buckets (append mt1 mt2 mt3)))
((cs61a-xb . 72) (cs61a-xc . 80) (cs61a-xd . 38) (cs61a-xf . 65)
 (cs61a-xk . 34) (cs61a-xw . 62) (cs61a-ya . 120) (cs61a-yb . 89)
 (cs61a-yc . 32))
```

Note that the returned list has the keys in sorted order. This is a consequence of the sorting done by `sort-into-buckets`, and also, in the real parallel `mapreduce`, a consequence of the order in which keys are assigned to processors (the "partitioning function" discussed in the `MapReduce` paper).

Similarly, we could ask *how many* midterms each student took:

```
> (groupreduce (lambda (new old) (+ 1 old)) 0
               (sort-into-buckets (append mt1 mt2 mt3)))
((cs61a-xb . 2) (cs61a-xc . 3) (cs61a-xd . 1) (cs61a-xf . 2) (cs61a-xk . 1)
 (cs61a-xw . 2) (cs61a-ya . 3) (cs61a-yb . 3) (cs61a-yc . 1))
```

We could combine these in the obvious way to get the average score per student, for exams actually taken.

Word frequency counting. A common problem is to look for commonly used words in a document. For starters, we'll count word frequencies in a single sentence. The first step is to turn the sentence into key-value pairs in which the key is the word and the value is always 1:

```
> (map (lambda (wd) (list (make-kv-pair wd 1))) '(cry baby cry))
((cry . 1) (baby . 1) (cry . 1))
```

If we group these by key and add the values, we'll get the number of times each word appears.

```
(define (wordcounts1 sent)
  (groupreduce + 0 (sort-into-buckets (map (lambda (wd) (make-kv-pair wd 1))
                                           sent))))

> (wordcounts1 '(cry baby cry))
((baby . 1) (cry . 2))
```

Now to try the same task with (simulated) files. When we use the real `mapreduce`, it'll give us file data in the form of a key-value pair whose key is the name of the file and whose value is a line from the file, in the form of a sentence. For now, we're going to simulate a file as a list whose car is the "filename" and whose cdr is a list of sentences, representing the lines of the file. In other words, a file is a list whose first element is the filename and whose remaining elements are the lines.

```
(define filename car)
(define lines cdr)
```

Here's some data for us to play with:

```
(define file1 '((please please me) (i saw her standing there) (misery)
                (anna go to him) (chains) (boys) (ask me why)
                (please please me) (love me do) (ps i love you)
                (baby its you) (do you want to know a secret)))
(define file2 '((with the beatles) (it wont be long) (all ive got to do)
                (all my loving) (dont bother me) (little child)
                (till there was you) (roll over beethoven) (hold me tight)
                (you really got a hold on me) (i wanna be your man)
                (not a second time)))
(define file3 '((a hard days night) (a hard days night)
                (i should have known better) (if i fell)
                (im happy just to dance with you) (and i love her)
                (tell me why) (cant buy me love) (any time at all)
                (ill cry instead) (things we said today) (when i get home)
                (you cant do that) (ill be back)))
```

We start with a little procedure to turn a "file" into an a-list in the form `mapreduce` will give us:

```
(define (file->linelist file)
  (map (lambda (line) (make-kv-pair (filename file) line))
       (lines file)))

> (file->linelist file1)
(((please please me) i saw her standing there)
 ((please please me) misery)
 ((please please me) anna go to him)
 ((please please me) chains)
 ((please please me) boys)
 ((please please me) ask me why)
 ((please please me) please please me)
```

```
 ((please please me) love me do)
 ((please please me) ps i love you)
 ((please please me) baby its you)
 ((please please me) do you want to know a secret))
```

Note that `((please please me) misery)` is how Scheme prints the kv-pair `((please please me) . (misery))`.

Now we modify our `wordcounts1` procedure to accept such kv-pairs:

```
(define (wordcounts files)
  (groupreduce + 0 (sort-into-buckets
                    (flatmap (lambda (kv-pair)
                               (map (lambda (wd) (make-kv-pair wd 1))
                                    (kv-value kv-pair)))
                             files))))
```

```
> (wordcounts (append (file->linelist file1)
                      (file->linelist file2)
                      (file->linelist file3)))
((a . 4) (all . 3) (and . 1) (anna . 1) (any . 1) (ask . 1) (at . 1)
 (baby . 1) (back . 1) (be . 3) (beethoven . 1) (better . 1) (bother . 1)
 (boys . 1) (buy . 1) (cant . 2) (chains . 1) (child . 1) (cry . 1)
 (dance . 1) (days . 1) (do . 4) (dont . 1) (fell . 1) (get . 1) (go . 1)
 (got . 2) (happy . 1) (hard . 1) (have . 1) (her . 2) (him . 1) (hold . 2)
 (home . 1) (i . 7) (if . 1) (ill . 2) (im . 1) (instead . 1) (it . 1)
 (its . 1) (ive . 1) (just . 1) (know . 1) (known . 1) (little . 1)
 (long . 1) (love . 4) (loving . 1) (man . 1) (me . 8) (misery . 1) (my . 1)
 (night . 1) (not . 1) (on . 1) (over . 1) (please . 2) (ps . 1) (really . 1)
 (roll . 1) (said . 1) (saw . 1) (second . 1) (secret . 1) (should . 1)
 (standing . 1) (tell . 1) (that . 1) (there . 2) (things . 1) (tight . 1)
 (till . 1) (time . 2) (to . 4) (today . 1) (wanna . 1) (want . 1) (was . 1)
 (we . 1) (when . 1) (why . 2) (with . 1) (wont . 1) (you . 7) (your . 1))
```

(If you count yourself to check, remember that words in the album titles don't count! They're keys, not values.)

Note the call to `flatmap` above. In a real `mapreduce`, each file would be mapped on a different processor, and the results would be distributed to `reduce` processes in parallel. Here, the `map` over files gives us a list of a-lists, one for each file, and we have to append them to form a single a-list. `Flatmap` flattens (appends) the results from calling `map`.

We can postprocess the groupreduce output to get an overall reduction to a single value:

```
(define (mostfreq files)
  (accumulate (lambda (new old)
                (cond ((> (kv-value new) (kv-value (car old)))
                       (list new))
                      ((= (kv-value new) (kv-value (car old)))
                       (cons new old))        ; In case of tie, remember both.
                      (else old)))
              (list (make-kv-pair 'foo 0))        ; Starting value.
              (groupreduce + 0 (sort-into-buckets
                                (flatmap (lambda (kv-pair)
                                           (map (lambda (wd)
                                                  (make-kv-pair wd 1))
                                                (kv-value kv-pair)))
                                         files)))))
```

```
> (mostfreq (append (file->linelist file1)
                    (file->linelist file2)
                    (file->linelist file3)))
((me . 8))
```

(Second place is "you" and "I" with 7 appearances each, which would have made a two-element a-list as the result.) If we had a truly enormous word list, we'd put it into a distributed file and use another `mapreduce` to find the most frequent words of subsets of the list, and then find the most frequent word of those most frequent words.

Searching for a pattern. Another task is to search through files for lines matching a pattern. A *pattern* is a sentence in which the word `*` matches any set of zero or more words:

```
> (match? '(* i * her *) '(i saw her standing there))
#t
> (match? '(* i * her *) '(and i love her))
#t
> (match? '(* i * her *) '(ps i love you))
#f
```

Here's how we look for lines in files that match a pattern:

```
(define (grep pattern files)
  (groupreduce cons '()
               (sort-into-buckets
                (flatmap (lambda (kv-pair)
                           (if (match? pattern (kv-value kv-pair))
                               (list kv-pair)
                               '()))
                         files))))

> (grep '(* i * her *) (append (file->linelist file1)
                               (file->linelist file2)
                               (file->linelist file3)))
(((a hard days night) (and i love her))
 ((please please me) (i saw her standing there)))
```

Summary. The general pattern here is

```
(groupreduce reducer base-case
             (sort-into-buckets
              (map-or-flatmap mapper data)))
```

This corresponds to

```
(mapreduce mapper reducer base-case data)
```

in the truly parallel `mapreduce` exploration we'll be doing later.

CS 61A Lecture Notes Week 5

Topic: Hierarchical data/Scheme interpreter

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

Midterm 1 is this week.

• Example: A Calculator Program

Later in the course we'll be studying several variants of a Scheme interpreter written in Scheme. As a first, small step in that direction, here is an interactive calculator that accepts arithmetic expressions in Scheme notation, but without variables, without the ability to define procedures, and with no data types other than numbers. Here's how it works:

```
STk> (load "~cs61a/lib/calc.scm")
STk> (calc)
calc: (+ 2 3)
5
calc: (+ (* 2 3) (* 4 5))
26
calc: foo
Error: calc: bad expression: foo
```

The last example shows that there are no variables in this language.

The entire program consists of three procedures in 30 lines of code. You should find it easy to understand. And yet these three procedures exactly parallel the core procedures in a real Scheme interpreter:

1. The read-eval-print loop: interact with the user.

2. Eval: `(eval expression)` returns the value of the expression.

3. Apply: `(apply function argument-values)` calls the function and returns the value it returns.

Here's the read-eval-print loop (or REPL, pronounced "rep-uhl"):

```
(define (calc)
  (display "calc: ")
  (flush)
  (print (calc-eval (read)))
  (calc))
```

The calls to `display` and `flush` print the user prompt. (`Flush` is used when you want to print something that doesn't end with a newline character, but you want it to print right away. Ordinarily, most programs, including STk, save up the characters you print until you finish a line, then send the entire line to the operating system at once. `Flush` tells STk not to wait for the end of the line.)

The most important line is the `(print (calc-eval (read)))`. `Read` is a Scheme primitive that reads a datum from the keyboard. "Datum" here means a word, a list, or any other single Scheme value. In our case, the things we're reading are Scheme expressions, but `read` doesn't know that; as far as `read` is concerned, `(+ 2 3)` is just a list of three elements, not a request to add two numbers.

The fact that a Scheme expression is just a Scheme datum—a list—makes it very easy to write an interpreter. This is why Scheme uses the `(+ 2 3)` notation for function calls rather than, say, `+(2,3)` as in the usual mathematical notation for functions! Even a more complicated expression such as `(+ (* 2 3) (* 4 5))` is one single Scheme datum, a list, in which some of the elements are themselves lists.

What do we want to do with the thing `read` returns? We want to treat it as an expression in this Scheme-

subset language and *evaluate* it. That's the job of `calc-eval`. (We use the names `calc-eval` and `calc-apply` in this program because STk has primitive procedures called `eval` and `apply`, and we don't want to step on those names. The STk procedures have jobs exactly analogous to the ones in `calc`, though; every interpreter for any Lisp-family language has some form of `eval` and `apply`.)

Once we get the value from `eval`, what do we want to do with it? We want to show it to the user by printing it to the display. That's the job of `print`. So now you understand why it's a "read-eval-print" loop! Read an expression, evaluate it, and print its value.

Finally, the procedure ends with a recursive call to itself, so it loops forever; this is the "loop" part of the REPL.

Notice that `read` and `print` are not functional programming; `read` returns a different value each time it's called, and `print` changes something in the world instead of just returning a value. The body of the REPL has more than one expression; Scheme evaluates the expressions in order, and returns the value of the last expression. (In this case, though, it never returns a value at all, since the last expression is a recursive call and there's no base case to end the recursion.) In functional programming, it doesn't make sense to have more than one expression in a procedure body, since a function can return only one value.

(It's also worth noting, in passing, that the REPL is the `only` non-functional part of this program. Even though we're doing something interactive, functional programming techniques are still the best way to do most of the work of the calculator program.)

The job of `eval` is to turn expressions into values. It's very important to remember that those are two different things. Some people lose midterm exam points by thinking that when you type `'foo` into Scheme, it prints out `'foo`. Of course it really prints `foo` without the quotation mark. `Foo` is a possible Scheme value, but `'foo` really makes sense only as an expression. (Of course, as we've seen, every expression is also a possible value; what expression would you type to Scheme to make it print `'foo`?) What confuses people is that *some* things in Scheme are both expressions and values; a number is a Scheme expression whose value is the number itself. But most expressions have a value different from the expression itself.

Part of what makes one programming language different from another is what counts as an expression. In Scheme, certain kinds of expressions have funny notation rules, such as the extra parentheses around clauses in a `cond` expression. The notation used in a language is called its *syntax*. `Eval` is the part of a Lisp interpreter that knows about syntax. Our simplified calculator language has only two kinds of syntax: numbers, which are *self-evaluating* (i.e., the value is the number itself), and lists, which represent function calls:

```
(define (calc-eval exp)
  (cond ((number? exp) exp)
        ((list? exp) (calc-apply (car exp) (map calc-eval (cdr exp))))
        (else (error "Calc: bad expression:" exp))))
```

In real Scheme, there are more kinds of self-evaluating expressions, such as Booleans (`#t` and `#f`) and `"strings in double quotes"`; there are variables, which are expressions; and some lists are special forms instead of procedure calls. So in a Scheme interpreter, `eval` is a little more complicated, but not that much more.

By the way, notice that we're talking about two different programming languages here. I've said that the calculator language doesn't have variables, and yet in `calc-eval` we're using a variable named `exp`. This isn't a contradiction because `calc-eval` isn't itself a program in calculator-language; it's a program in STk, which is a complete Scheme interpreter. The calculator language and Scheme are different enough so that you probably won't be confused about this, but I'm belaboring the point because later on we'll see interpreters for much more complete subsets of Scheme, and you can easily get confused about whether some expression you're looking at is part of the interpreter, and therefore an STk expression, or data given to the interpreter, and therefore a mini-Scheme expression.

The way `calc-eval` handles function calls is the only part of the calculator that is *not* the same as the corresponding feature of real Scheme. That's because in calculator language, numbers are the only data type, and so in particular procedures aren't data. In a real Scheme procedure call expression, the first subexpression, the one whose value provides the procedure itself, has to be evaluated just as much as the argument subexpressions. *Often* the first subexpression is just a variable name, such as `+` or `cdr`, but the expression could be

```
((lambda (x) (+ x 5))  (* 2 3))
```

in which case the first subexpression is a special form, a `lambda` expression. But in calculator language, the first sub-"expression" is always the name of the function, and there are only four possibilities: `+`, `-`, `*`, and `/`. I put "expression" in quotes because these symbols are *not* expressions in calculator language. So the expression in `calc-eval` that handles procedure calls is

```
(calc-apply (car exp) (map calc-eval (cdr exp)))
```

The first argument to `calc-apply` is the *name* of the function we want to call. The rest of the expression `exp` consists of actual argument subexpressions, which we recursively evaluate by calling `calc-eval` for each of them. (Remember that `map` is the list version of `every`; it calls `calc-eval` repeatedly, once for each element of `(cdr exp)`.) When we look at a more-nearly-real Scheme interpreter in another week, the corresponding part of `eval` will look like this:

```
(apply (EVAL (car exp)) (map eval (cdr exp)))
```

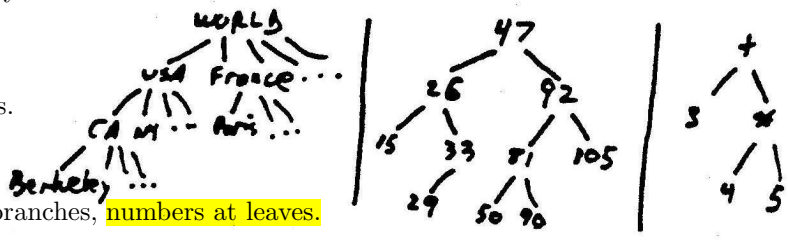
`Eval` is the part of the interpreter that knows about the syntax of the language. By contrast, `apply` works entirely in the world of values; there are no expressions at all in the arguments to `apply`. Our version of `apply` has the four permitted operations built in:

```
(define (calc-apply fn args)
  (cond ((eq? fn '+) (accumulate + 0 args))
        ((eq? fn '-) (cond ((null? args) (error "Calc: no args to -"))
                           ((= (length args) 1) (- (car args)))
                           (else (- (car args) (accumulate + 0 (cdr args))))))
        ((eq? fn '*) (accumulate * 1 args))
        ((eq? fn '/) (cond ((null? args) (error "Calc: no args to /"))
                           ((= (length args) 1) (/ (car args)))
                           (else (/ (car args) (accumulate * 1 (cdr args))))))
        (else (error "Calc: bad operator:" fn))))
```

The associative operations can be done with a single call to the higher order `accumulate` function; the non-associative ones have to deal with the special case of one argument separately. I'm not spending a lot of time on the details because this isn't how real Scheme handles function calls; the real `apply` takes the actual procedure as its first argument, so it doesn't have to have special knowledge of the operators built into itself. Also, the real `apply` handles user-defined procedures as well as the ones built into the language.

• Trees. Big idea: representing a hierarchy of information.

What are trees good for?



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

The name "tree" comes from the branching structure of the pictures, like real trees in nature except that they're drawn with the root at the top and the leaves at the bottom.

A *node* is a point in the tree. In these pictures, each node includes a *datum* (the value shown at the node, such as `France` or `26`) but also includes the entire structure under that datum and connected to it, so the `France` node includes all the French cities, such as `Paris`. Therefore, **each node is itself a tree**—the terms "tree" and "node" mean the same thing! The reason we have two names for it is that we generally use "tree" when we mean the entire structure that our program is manipulating, and "node" when we mean just one piece of the overall structure. Therefore, another synonym for "node" is "subtree."

The *root node* (or just the *root*) of a tree is the node at the top. Every tree has one root node. (A more general structure in which nodes can be arranged more flexibly is called a *graph*; you'll study graphs in 61B and later courses.)

The *children* of a node are the nodes directly beneath it. For example, the children of the `26` node in the picture are the `15` node and the `33` node.

A *branch* node is a node that has at least one child. A *leaf* node is a node that has no children. (The root node is also a branch node, except in the trivial case of a one-node tree.)

• The `Tree` abstract data type

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'll get back to some of these variations later, but first we'll consider a commonly used version of trees, in which every tree has at least one node, every node has a datum, and nodes can have any number of children. Here are the constructor and selectors:

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

The selector `children` should return a *list of trees*, the children of the node. These children are themselves trees. There is a name for a list of trees: a *forest*. It's very important to remember that Tree and Forest are two different data types! A forest is just a sequence, although its elements are required to be trees, and so we can manipulate forests using the standard procedures for sequences (`cons`, `car`, `cdr`, etc.); a tree is *not* a sequence, and should be manipulated only with the tree constructor and selectors.

A leaf node is one with no children, so its `children` list is empty:

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

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

The straightforward implementation is

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

• Mapping over trees

One thing we might want to do with a tree is create another tree, with the same shape as the original, but with each datum replaced by some function of the original. This is the tree equivalent of `map` for lists.

```
;;;;;                          In file cs61a/lectures/2.2/tree1.scm
(define (treemap fn tree)
  (make-tree (fn (datum tree))
             (map (lambda (t) (treemap fn t))
                  (children tree) )))
```

This is a remarkably simple and elegant procedure, especially considering the versatility of the data structures it can handle (trees of many different sizes and shapes). It's one of the more beautiful things you'll see in the course, so spend some time appreciating it.

Every tree node consists of a datum and some children. In the new tree, the datum corresponding to this node should be the result of applying `fn` to the datum of this node in the original tree. What about the children of the new node? There should be the same number of children as there are in the original node, and each new child should be the result of calling `treemap` on an original child. Since a forest is just a list, we can use `map` (not `treemap`!) to generate the new children.

• Mutual recursion

Pay attention to the strange sort of recursion in this procedure. `Treemap` does not actually call itself! `Treemap` calls `map`, giving it a function that in turn calls `treemap`. The result is that each call to `treemap` may give rise to any number of recursive calls, via `map`: one call for every child of this node.

This pattern (procedure `A` invokes procedure `B`, which invokes procedure `A`) is called *mutual recursion*. We can rewrite `treemap` without using `map`, to make the mutual recursion more visible:

```
;;;;;                          In file cs61a/lectures/2.2/tree11.scm
(define (treemap fn tree)
  (make-tree (fn (datum tree))
             (forest-map fn (children tree))))

(define (forest-map fn forest)
  (if (null? forest)
      '()
      (cons (treemap fn (car forest))
            (forest-map fn (cdr forest)))))
```

`Forest-map` is a helper function that takes a forest, not a tree, as argument. `Treemap` calls `forest-map`, which calls `treemap`.

Mutual recursion is what makes it possible to explore the two-dimensional tree data structure fully. In particular, note that reaching the base case in `forest-map` does not mean that the entire tree has been visited! It means merely that one group of sibling nodes has been visited (a "horizontal" base case), or that a node has no children (a "vertical" base case). The entire tree has been seen when every child of the root node has been completed.

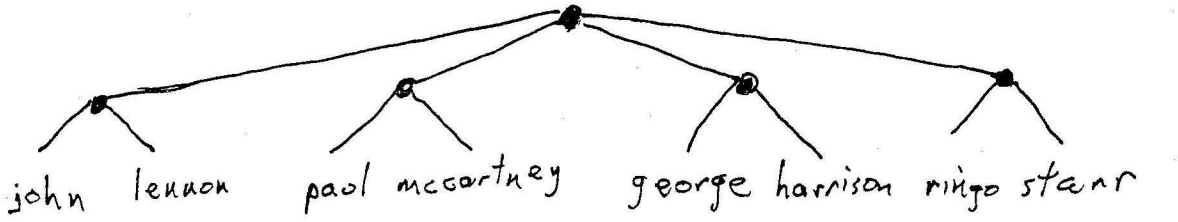
Note that we use `cons`, `car`, and `cdr` when manipulating a forest, but we use `make-tree`, `datum`, and `children` when manipulating a tree. Some students make the mistake of thinking that data abstraction means "always say `datum` instead of `car`"! But that defeats the purpose of using different selectors and constructors for different data types.

• Deep lists

Trees are our first two-dimensional data structure. But there's a sense in which any list that has lists as elements is also two-dimensional, and can be viewed as a kind of tree. We'll use the name *deep lists* for lists that contain lists. For example, the list

```
[[john lennon] [paul mccartney] [george harrison] [ringo starr]]
```

is probably best understood as a sequence of sentences, but instead we can draw a picture of it as a sort of tree:



Don't be confused; this is *not* an example of the Tree abstract data type we've just developed. In this picture, for example, only the "leaf nodes" contain data, namely words. We didn't make this list with `make-tree`, and it wouldn't make sense to examine it with `datum` or `children`.

But we can still use the *ideas* of tree manipulation if we'd like to do something for every word in the list. Compare the following procedure with the first version of `treemap` above:

```
;;;;;                          In file cs61a/lectures/2.2/tree22.scm
(define (deep-map fn lol)
  (if (list? lol)
      (map (lambda (element) (deep-map fn element))
           lol)
      (fn lol)))
```

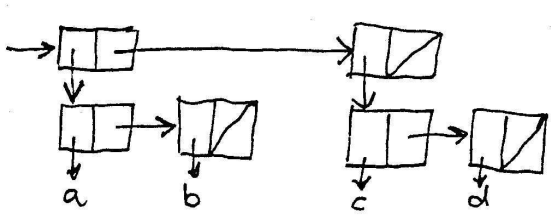
The formal parameter `lol` stands for "list of lists." This procedure includes the two main tasks of `treemap`: applying the function `fn` to one datum, and using `map` to make a recursive call for each child.

But `treemap` applies to the Tree abstract data type, in which every node has both a datum and children, so `treemap` carries out both tasks for each node. In a deep list, by contrast, the "branch nodes" have children but no datum, whereas the "leaf nodes" have a datum but no children. That's why `deep-map` chooses only one of the two tasks, using `if` to distinguish branches from leaves.

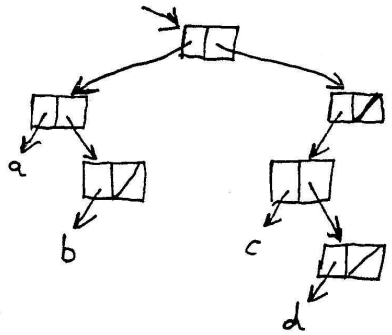
Note: SICP does not define a Tree abstract data type; they use the term "tree" to describe what I'm calling a deep list. So they use the name `tree-map` in Exercise 2.31, page 113, which asks you to write what I've called `deep-map`. (Since I've done it here, you should do the exercise without using `map`.) SICP does define an abstract data type for *binary* trees, in which each node can have a `left-branch` and/or a `right-branch`, rather than having any number of children.

• **Car/cdr recursion**

Consider the deep list `((a b) (c d))`. Ordinarily we would draw its box and pointer diagram with a horizontal spine at the top and the sublists beneath the spine:



But imagine that we grab the first pair of this structure and "shake" it so that the pairs fall down as far as they can. We'd end up with this diagram:



Note that these two diagrams represent the same list! They have the same pairs, with the same links from one pair to another. It's just the position of the pairs on the page that's different. But in this new picture, the structure looks a lot like a binary tree, in which the branch nodes are pairs and the leaf nodes are atoms (non-pairs). The "left branch" of each pair is its `car`, and the "right branch" is its `cdr`. With this metaphor, we can rewrite `deep-map` to look more like a binary tree program:

```
;;;;;                          In file cs61a/lectures/2.2/tree3.scm
(define (deep-map fn xmas)
  (cond ((null? xmas) '())
        ((pair? xmas)
         (cons (deep-map fn (car xmas))
               (deep-map fn (cdr xmas))))
        (else (fn xmas))))
```

(The formal parameter `xmas` reflects the fact that the picture looks kind of like a Christmas tree.)

This procedure strongly violates data abstraction! Ordinarily when dealing with lists, we write programs that treat the car and the cdr differently, reflecting the fact that the car of a pair is a list element, whereas the cdr is a sublist. But here we treat the car and the cdr identically. One advantage of this approach is that it works even for improper lists:

```
> (deep-map square '((3 . 4) (5 6))
((9 . 16) (25 36))
```

• **Tree recursion**

Compare the car/cdr version of `deep-map` with ordinary `map`:

```
(define (map fn seq)
  (if (null? seq)
      '()
      (cons (fn (car seq))
            (map fn (cdr seq)))))
```

Each non-base-case invocation of `map` gives rise to one recursive call, to handle the cdr of the sequence. The car, an element of the list, is not handled recursively.

By contrast, in `deep-map` there are *two* recursive calls, one for the car and one for the cdr. This is what makes the difference between a sequential, one-dimensional process and the two-dimensional process used for deep lists and for the Tree abstraction.

A procedure in which each invocation makes more than one recursive call is given the name *tree recursion* because of the relationship between this pattern and tree structures. It's tree recursion only if each call (other than a base case) gives rise to two or more recursive calls; it's not good enough to have two recursive calls of which only one is chosen each time, as in the following non-tree-recursive procedure:

```
(define (filter pred seq)
  (cond ((null? seq) '())
        ((pred (car seq)) (cons (car seq) (filter pred (cdr seq))))
        (else (filter pred (cdr seq)))))
```

There are two recursive calls to `filter`, but only one of them is actually carried out each time, so this is a sequential recursion, not a tree recursion.

A program can be tree recursive even if there is no actual tree-like data structure used, as in the Fibonacci number function:

```
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

This procedure just handles numbers, not trees, but each non-base-case call adds the results of two recursive calls, so it's a tree recursive program.

• **Tree traversal**

Many problems involve visiting each node of a tree to look for or otherwise process 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 traverse a sequence (left to right), but many ways in which we can traverse a tree.

In the following examples, we "visit" each node by printing the datum at that node. If you apply these procedures to actual trees, you can see the order in which the nodes are visited.

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

```
;;;;;                          In file cs61a/lectures/2.2/search.scm
(define (depth-first-search tree)
  (print (datum tree))
  (for-each depth-first-search (children tree)))
```

This is the easiest way, because the program's structure follows the data structure; each child is traversed in its entirety (that is, including grandchildren, etc.) before looking at the next child.

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

What we want to do is take horizontal slices of the tree. First we look at the root node, then we look at the children of the root, then the grandchildren, and so on. The program is a little more complicated because the order in which we want to visit nodes isn't the order in which they're connected together.

To solve this, we use an extra data structure, called a *queue*, which is just an ordered list of tasks to be carried out. Each "task" is a node to visit, and a node is a tree, so a list of nodes is just a forest. The iterative helper procedure takes the first task in the queue (the car), visits that node, and adds its children at the end of the queue (using `append`).

```
;;;;;                          In file cs61a/lectures/2.2/search.scm
(define (breadth-first-search tree)
  (bfs-iter (list tree)))

(define (bfs-iter queue)
  (if (null? queue)
      'done
      (let ((task (car queue)))
        (print (datum task))
        (bfs-iter (append (cdr queue) (children task))))))
```

Why would we use this more complicated technique? For example, in some situations the same value might appear as a datum more than once in the tree, and we want to find the *shortest* path from the root node to a node containing that datum. To do that, we have to look at nodes near the root before looking at nodes far away from the root.

Another example is a game-strategy program that *generates* a tree of moves. The root node is the initial board position; each child is the result of a legal move I can make; each child of a child is the result of a legal move for my opponent, and so on. For a complicated game, such as chess, the move tree is much too large to generate in its entirety. So we use a breadth-first technique to generate the move tree up to a certain depth (say, ten moves), then we look for desirable board positions at that depth. (If we used a depth-first program, we'd follow one path all the way to the end of the game before starting to consider a different possible first move.)

For binary trees, within the general category of depth-first traversals, there are three possible variants:

Preorder: Look at a node before its children.

```
;;;;;                          In file cs61a/lectures/2.2/print.scm
(define (pre-order tree)
  (cond ((null? tree) '())
        (else (print (entry tree))
              (pre-order (left-branch tree))
              (pre-order (right-branch tree)) )))
```

Inorder: Look at the left child, then the node, then the right child.

```
;;;;;                          In file cs61a/lectures/2.2/print.scm
(define (in-order tree)
  (cond ((null? tree) '())
        (else (in-order (left-branch tree))
              (print (entry tree))
              (in-order (right-branch tree)) )))
```

Postorder: Look at the children before the node.

```
;;;;;                          In file cs61a/lectures/2.2/print.scm
(define (post-order tree)
  (cond ((null? tree) '())
        (else (post-order (left-branch tree))
              (post-order (right-branch tree))
              (print (entry tree)) )))
```

For a tree of arithmetic operations, preorder traversal looks like Lisp; inorder traversal looks like conventional arithmetic notation; and postorder traversal is the HP calculator "reverse Polish notation."

• Path finding

As an example of a somewhat more complicated tree program, suppose we want to look up a place (e.g., a city) in the world tree, and find the path from the root node to that place:

```
> (find-place 'berkeley world-tree)
(world (united states) california berkeley)
```

If a place isn't found, `find-place` will return the empty list.

To find a place within some tree, first we see if the place is the datum of the root node. If so, the answer is a one-element list containing just the place. Otherwise, we look at each child of the root, and see if we can find the place within that child. If so, the path within the complete tree is the path within the child, but with the root datum added at the front of the path. For example, the path to Berkeley within the USA subtree is

```
((united states) california berkeley)
```

so we put `world` in front of that.

Broadly speaking, this program has the same mutually recursive tree/forest structure as the other examples we've seen, but one important difference is that once we've found the place we're looking for, there's no need to visit other subtrees. Therefore, we don't want to use `map` or anything equivalent to handle the children of a node; we want to check the first child, see if we've found a path, and only if we haven't found it should we go on to the second child (if any). This is the reason for the `let` in `find-forest`.

```
;;;;;                          In file cs61a/lectures/2.2/world.scm
(define (find-place place tree)
  (if (eq? place (datum tree))
      (cons (datum tree) '())
      (let ((try (find-forest place (children tree))))
        (if (not (null? try))
            (cons (datum tree) try)
            '()))))

(define (find-forest place forest)
  (if (null? forest)
      '()
      (let ((try (find-place place (car forest))))
        (if (not (null? try))
            try
            (find-forest place (cdr forest))))))
```

(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.)

• The `Scheme-1` interpreter

[This topic may be in week 5 or in week 6 depending on the holiday schedule.]

We're going to investigate a Scheme interpreter written in Scheme. SICP has a rather large and detailed Scheme interpreter in Chapter 4, which we'll get to near the end of the semester. But students often find that program intimidating, so we're going to work up to it with a series of three smaller versions that leave out some details and some of the features of real Scheme. This week we'll use the `Scheme-1` interpreter.

We weren't ready for this investigation until we had the idea of lists that contain sublists, because that's what a Scheme program is – a list. That's the point of all those parentheses; the Scheme language can look at a Scheme program as data, rather than as something different from data.

Here's how we use the interpreter:

```
STk> (load "~cs61a/lib/scheme1.scm")
STk> (scheme-1)
Scheme-1: (+ 2 3)
5
Scheme-1: ((lambda (x) (* x 3)) 4)
12
```

To leave `Scheme-1` and return to the STk prompt, just enter an illegal expression, such as `()`.

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 evaluation models.
- 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*.

This week's interpreter implements the *substitution* model of evaluation that we learned in Chapter 1 of SICP. In Chapter 3, we'll get to a more complicated but more realistic evaluation model, called the *environment* model.

Universality means we can write *one program* that's equivalent to all other programs. We'll talk more about this when we see SICP's full Scheme interpreter in Chapter 4. This week's interpreter, although universal in principle, doesn't make the point clearly because it's quite difficult to write serious programs in it, mainly because it lacks `define`.

Our Scheme interpreter leaves out many 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 a Scheme interpreter, using the substitution model, with most of the details left out:

```
;;;;;                          In file cs61a/lectures/2.2/tiny.scm
(define (scheme)
  (display "> ")
  (print (eval (read)))
  (scheme) )

(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((symbol? exp) (look-up-global-value exp))
        ((special-form? exp) (do-special-form exp))
        (else (apply (eval (car exp))
                     (map eval (cdr exp)) ))))

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

Although the versions we can actually run are bigger, this really does capture the essential structure of every Lisp interpreter, namely, a mutual recursion between `eval` (evaluate an expression) and `apply` (apply a function to arguments). To evaluate a procedure call 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 with the argument values in place of the formal parameters.

The `substitute` procedure is essentially the `substitute2` that you wrote in last week's homework, except that it has to be a little more complicated to avoid substituting for quoted symbols and for the formal parameters of a `lambda` inside the body.

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

The `Scheme-1` interpreter has only three special forms: `quote`, `if`, and `lambda`. In particular, it doesn't have `define`, so there are no global variables, and we can't give procedures global names. If we need a name for a procedure, we have to use it as an argument to another procedure. In particular, if we want to write *recursive* procedures we have to use a trick that was an extra-for-experts in week 2:

```
Scheme-1: ((lambda (n)
             ((lambda (f) (f f n))   ; the "Y combinator"
              (lambda (fact n)
                (if (= n 0)
                    1
                    (* n (fact fact (- n 1))) )) ))
           5)
120
```

CS 61A Lecture Notes Week 6

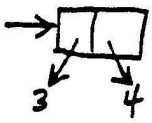
Topic: Generic operators

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

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 the 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.

Note: The second midterm exam is next week.

CS 61A Lecture Notes Week 7

Topic: Object-oriented programming

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

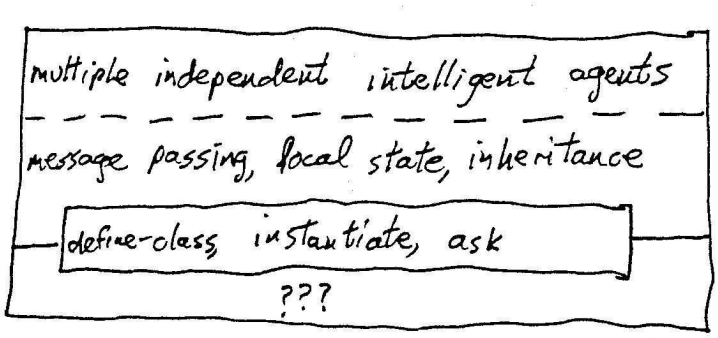
Midterm 2 is this week.

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/demo1.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/demo1.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/demo1.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 8

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`.

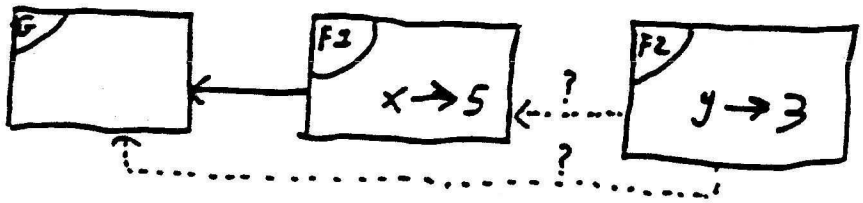
Skip step 2 for a moment and think about step 3. The idea is that we are going to evaluate the expression `(* x x)` but we are refining our notion of what it means to evaluate an expression. Expressions are no longer evaluated in a vacuum, but instead, every evaluation must be done with respect to some 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)
```

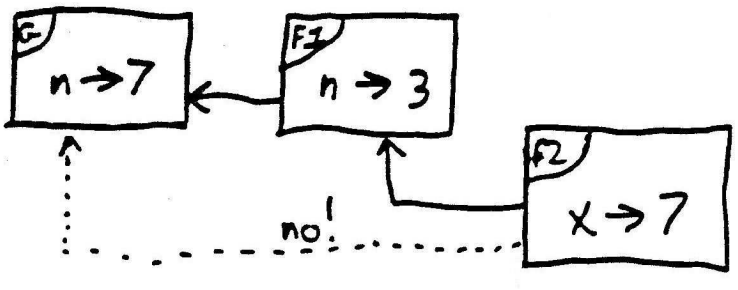


When we invoke `f`, we create a frame (call it F1) in which `x` is bound to `5`. We use that frame to extend the global environment (call it G), creating a new environment E1. Now we evaluate the body of `f`, which contains the internal definition for `g` and the expression `(g 3)`. To invoke `g`, we 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))
```

CS 61A Lecture Notes Week 9

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 node) (car node))
  (define (question node) (cadr node))
  (define (yespart node) (caddr node))
  (define (nopart node) (cadddr node))
  (define (answer node) (cadr node))
  (define (leaf? node) (eq? (type node) 'leaf))
  (define (branch? node) (eq? (type node) '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!`)

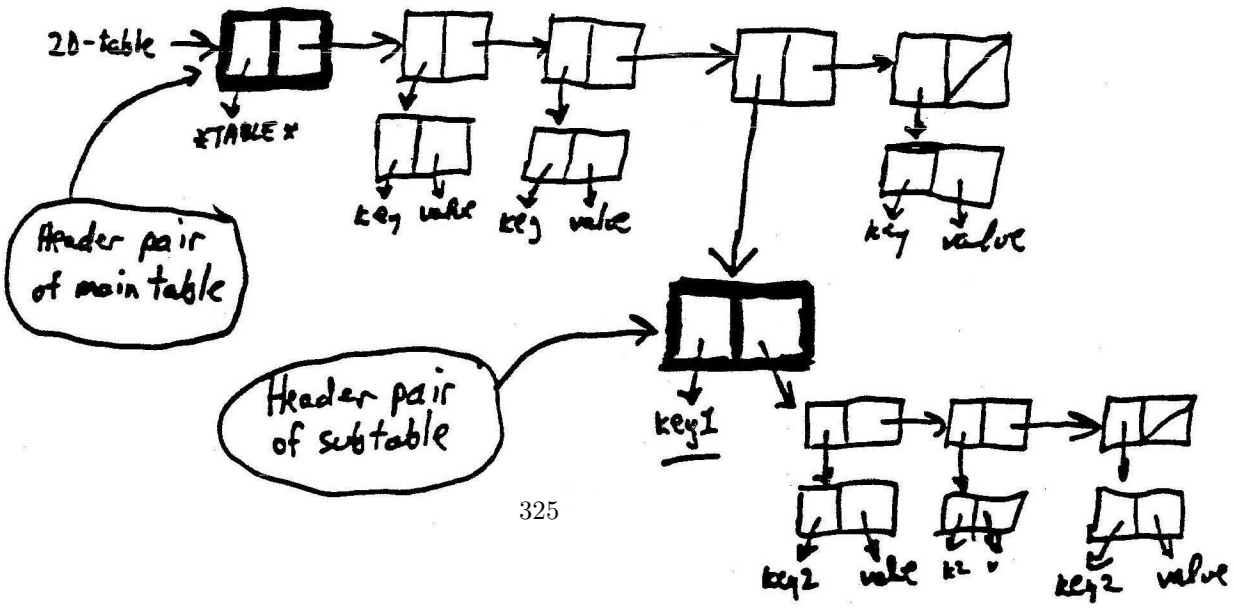
```
;;;;;                          In file cs61a/lectures/3.3/table.scm
(define (get key)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)
        #f
        (cdr record))))

(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.

•• **VECTORS.** So far we have seen one primitive data aggregation mechanism: the pair. We use linked pairs to represent *sequences* (an abstract type) in the form of *lists*.

The list suffers from one important weakness: Finding the nth element of a list takes time $\Theta(n)$ because you have to call `cdr` $n-1$ times. Scheme, like most programming languages, also provides a primitive aggregation mechanism without this weakness. In Scheme it's called a *vector*; in many other languages it's called an *array*, but it's the same idea. Finding the nth element of a vector takes $\Theta(1)$ time.

• **Vector primitives**

Some of the procedures for vectors are exact analogs to procedures for lists:

```
(vector a b c d ...)          (list a b c d ...)
(vector-ref vec n)            (list-ref lst n)
(vector-length vec)           (length lst)
```

Most notably, the *selector* for vectors, `vector-ref`, is just like the selector for lists (except that it's faster).

What about constructors? There's a `vector` procedure, just like the `list` procedure, that's good for situations in which you know exactly how many elements the sequence will have, and all of the element values, all at once. But there are no vector analogs to the list constructors `cons` and `append`, which are useful for *extending* lists. In particular, `cons` is the workhorse of recursive list processing procedures; we'll see that vector processing is done quite differently.

The weakness of vectors is that they can't be extended. You have to know the length of the vector when you create it. So instead of `cons` and `append` we have

```
(make-vector len)
```

which creates a vector of length `len`, in which the element values are unspecified. (You then use mutation, discussed below, to fill in the desired values.) Alternatively, if you want to create a vector in which every element has the same initial value, you can say

```
(make-vector len value)
```

Because vectors are created all at once, rather than one element at a time, *mutation* is crucial to any useful vector program. The primitive mutator for vectors is

```
(vector-set! vec n value)
```

This procedure is comparable to `set-car!` and `set-cdr!` for pairs. (It's interesting to note that Scheme doesn't provide a mutator for the nth element of a list; this is because most list processing is done using functional programming style, and pair mutation is mainly for special cases such as tables.)

The printed format of a vector is

```
#(a b c d)
```

You can quote this to include a constant vector in a program. (Note: In STk, vectors are self-evaluating, so you can omit the quotation mark, but this is a nonstandard extension to Scheme.)

Scheme also provides functions `list->vector` and `vector->list` that let you convert between the two sequence implementations.

• **Vector programming style**

Let's write a mapping function for vectors; it will take a function and a vector as arguments, and return a vector.

For reference, here's the `map` function for lists:

```
(define (map fn lst)
  (if (null? lst)
      '()
      (cons (fn (car lst))
            (map fn (cdr lst)))))
```

To do the same task for vectors, we must first create a new vector of the same length as the argument vector, then fill in the values using mutation:

```
;;;;;                          In file cs61a/lectures/vector.scm
(define (vector-map fn vec)
  (define (loop newvec n)
    (if (< n 0)
        newvec
        (begin (vector-set! newvec n (fn (vector-ref vec n)))
               (loop newvec (- n 1)))))
  (loop (make-vector (vector-length vec)) (- (vector-length vec) 1)))
```

This is a lot more complicated! It requires a helper procedure, and an extra *index variable*, `n`, to keep track of the element number within the vector. By contrast, the list version of `map` never actually knows how long its argument list is.

• **Strengths and weaknesses**

Of course, if we wanted, we could write our own equivalent to `cons` for vectors:

```
;;;;;                          In file cs61a/lectures/vector.scm
(define (vector-cons value vec)
  (define (loop newvec n)
    (if (= n 0)
        (begin (vector-set! newvec n value)
               newvec)
        (begin (vector-set! newvec n (vector-ref vec (- n 1)))
               (loop newvec (- n 1)))))
  (loop (make-vector (+ (vector-length vec) 1)) (vector-length vec)))
```

If we wrote similar procedures `vector-car` and `vector-cdr`, we could then write `vector-map` in a style exactly like `map`. But this would be a bad idea, because our `vector-cons` requires $\Theta(n)$ time to copy the elements from the old vector to the new one.

operation	**lists**	**vectors**
nth element	`list-ref`, $\Theta(n)$	`vector-ref`, $\Theta(1)$
add new element	`cons`, $\Theta(1)$	`vector-cons`, $\Theta(n)$

This is why there isn't one best way to represent sequences. Lists are faster (and allow for cleaner code) at adding elements, but vectors are faster at selecting arbitrary elements.

(Note, though, that if you want to select *all* the elements of a sequence, one after another, then lists are just as fast as arrays. It's only when you want to jump around within the sequence that arrays are faster.)

• **Example: Shuffling**

Suppose we want to shuffle a deck of cards — we want to reorder the cards randomly. We'll look at three solutions to this problem.

First, here's a solution using functional programming with lists. Because we aren't allowing mutation of pairs, this version does a lot of recopying:

```
;;;;;                          In file cs61a/lectures/vector.scm
(define (shuffle1 lst)
  (define (loop in out n)
    (if (= n 0)
        (cons (car in) (shuffle1 (append (cdr in) out)))
        (loop (cdr in) (cons (car in) out) (- n 1))))
  (if (null? lst)
      '()
      (loop lst '() (random (length lst)))))
```

This is a case in which functional programming has few virtues. The code is hard to read, and it takes $\Theta(n^2)$ time to shuffle a list of length n. (There are n recursive calls to `shuffle1`, each of which calls the $\Theta(n)$ primitives `append` and `length` as well as $\Theta(n)$ calls to the helper function `loop`.)

We can improve things using list mutation. Any list-based solution will still be $\Theta(n^2)$, because it takes $\Theta(n)$ time to find one element at a randomly chosen position, and we have to do that n times. But we can improve the constant factor by avoiding the copying of pairs that `append` does in the first version:

```
;;;;;                          In file cs61a/lectures/vector.scm
(define (shuffle2! lst)
  (if (null? lst)
      '()
      (let ((index (random (length lst))))
        (let ((pair ((repeated cdr index) lst))
              (temp (car lst)))
          (set-car! lst (car pair))
          (set-car! pair temp)
          (shuffle2! (cdr lst))
          lst))))
```

(Note: This could be improved still further by calling `length` only once, and using a helper procedure to subtract one from the length in each recursive call. But that would make the code more complicated, so I'm not bothering. You can take it as an exercise if you're interested.)

Vectors allow a more dramatic speedup, because finding each element takes $\Theta(1)$ instead of $\Theta(n)$:

```
;;;;;                          In file cs61a/lectures/vector.scm
(define (shuffle3! vec)
  (define (loop n)
    (if (= n 0)
        vec
        (let ((index (random n))
              (temp (vector-ref vec (- n 1))))
          (vector-set! vec (- n 1) (vector-ref vec index))
          (vector-set! vec index temp)
          (loop (- n 1)))))
  (loop (vector-length vec)))
```

The total time for this version is $\Theta(n)$, because it makes n recursive calls, each of which takes constant time.

- **How it works**

One handwavy paragraph on why vectors have the performance they do:

A pair is two pointers attached to each other in a single block of memory. A vector is similar, but it's a block of n pointers for an arbitrary (but fixed) number n. Since a vector is one contiguous block of memory, if you know the address of the beginning of the block, you can just add k to find the address of the kth element. The downside is that in order to get all the elements in a single block of memory, you have to allocate the block all at once.

If you don't understand that, don't worry about it until 61B.

Note: Programming project 3 starts this week.

CS 61A Lecture Notes Week 10

Topic: Client/server paradigm, Concurrency

Reading: Abelson & Sussman, Section 3.4

• Client/server programming paradigm

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. A 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 Mozilla, Netscape, or Internet Explorer to read most web pages, because they all 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 (Internet Protocol) 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.

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-connection s)
  (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)))))))))
```

• **Concurrency**

To work with the ideas in this section you should first

```
(load "~cs61a/lib/concurrent.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
```

Levels of Abstraction

Computers don't really have instructions quite like `reserve` and `release`, but we'll see that they do provide similar mechanisms. A typical programming environment includes concurrency control mechanisms at three levels of abstraction:

```
SICP name              What's protected          Provided by
---------              ----------------          -----------

serializer             high level abstraction    programming language
                        (procedure, object, ...)

mutex                  critical section          operating system

test-and-set!          one atomic                hardware
                        state transition
```

The serializer and the mutex are, in SICP, abstract data types. There is a constructor `make-serializer` that's implemented using a mutex, and a constructor `make-mutex` that's implemented using `test-and-set!`, which is a (simulated, in our case) hardware instruction.

Serializers

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 easy 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. (It turns out that there is a very tricky software algorithm to generate guaranteed atomic test-and-set, but in practice, there is almost always hardware support for parallelism. Look up "Peterson's algorithm" in Wikipedia if you want to see the software solution.)

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.

CS 61A Lecture Notes Week 11

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 $2 \leq f < n$."

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))
                 (range 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-range 2 (- n 1)))))
```

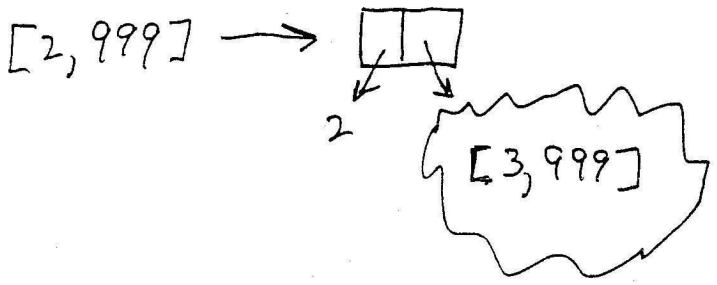
The only changes are `stream-range` instead of `range`, `stream-null?` instead of `null?`, and `stream-filter` instead of `filter`.

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-range` 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-range (+ from 1) to))
```

(inside `stream-range`) 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.)

• Memoization of streams. `Delay` is really slightly more complicated than what's shown above. It returns a procedure of no arguments that *memoizes* the promise; the expression given as `delay`'s argument is evaluated only the first time this promise is forced; after that, the value is remembered and reused. This is another reason why streams are a functional-only technique; any mutation operation in a promise will only happen once even if you cash in the promise repeatedly.

• Infinite streams. Think about the plain old list function

```
(define (range from to)
  (if (> from to)
      '()
      (cons from (range (+ from 1) to)) ))
```

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

```
(define (stream-range from to)
  (if (> from to)
      THE-EMPTY-STREAM
      (cons-STREAM from (stream-range (+ 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-range from)
  (cons-stream from (stream-range (+ 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 third midterm exam is next week.

CS 61A Lecture Notes Week 12

Topic: Metacircular evaluator

Reading: Abelson & Sussman, Section 4.1

Midterm 3 is this week.

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

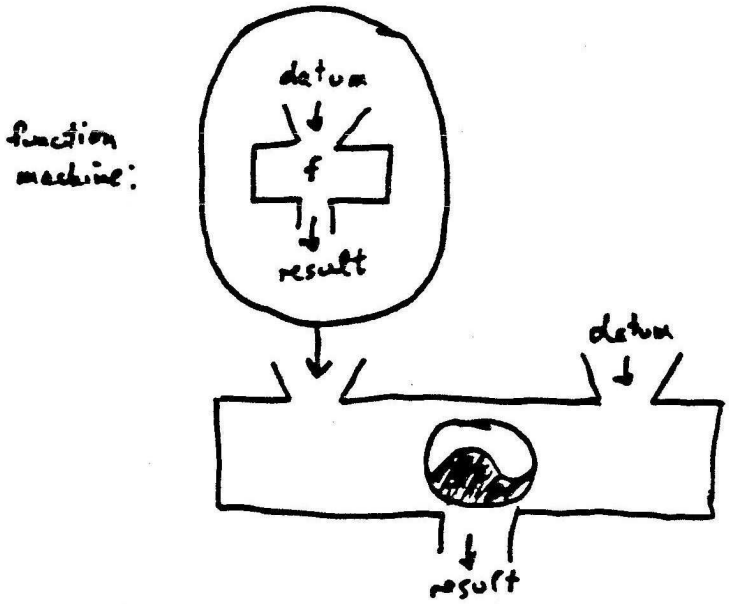
Here's a reminder of the reasons I mentioned in week 6 for studying a Scheme interpreter in Scheme, even though it's obviously not something you'd use in practice:

- 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. You might want to compare it to the one-screenful substitution-model interpreter you saw in week 6.

```
;;;;;                          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.

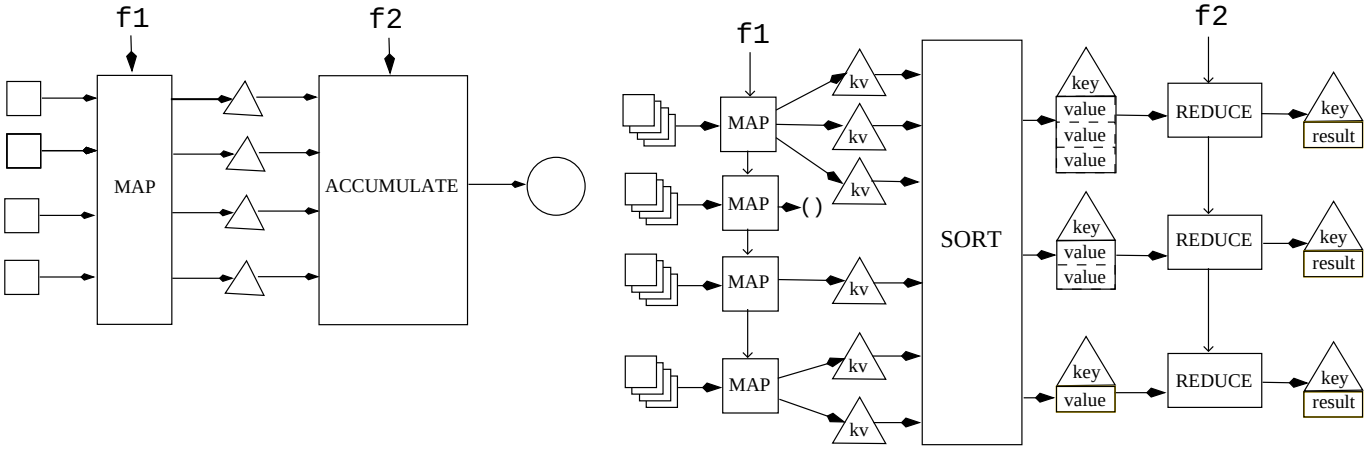
Finally, dynamic scope is useful for allowing "semi-global" variables. Take the metacircular evaluator as an example. Lots of procedures in it require `env` as an argument, but there's nothing special about the value of `env` in any one of those procedures. It's almost always just the current environment, 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.

• **Mapreduce part 2**

Here's the diagram of mapreduce again:

`(accumulate f2 base (map f1 data))` `(mapreduce f1 f2 base dataname)`



The seemingly unpoetic names `f1` and `f2` serve to remind you of two things: `f1` (the mapper) is used before `f2` (the reducer), and `f1` takes one argument while `f2` takes two arguments (just like the functions used with ordinary `map` and `accumulate` respectively).
mapper: `kv-pair` → `list-of-kv-pairs`
reducer: `value`, `partial-result` → `result`

All data are in the form of key-value pairs. Ordinary `map` doesn't care what the elements of the data list argument are, but `mapreduce` works only with data each of which is a key-value pair. In the Scheme interface to the distributed filesystem, a file is a stream. Every element of the file stream represents one line of the file, using a key-value pair whose key is the filename and whose value is a list of words, representing the text of the line. The `cdr` of a file stream is a promise to ask the distributed filesystem for the next line.

Each processor runs a separate `stream-map`. The overlapping squares at the left of the mapreduce picture represent an entire stream. (How is a large distributed file divided among map processes? It doesn't really matter, as far as the `mapreduce` user is concerned; `mapreduce` tries to do it as efficiently as possible given the number of processes and the location of the data in the filesystem.) The entire stream is the input to a `map` process; each element of the stream (a kv-pair) is the input to your mapper function `f1`.

For each key-value pair in the input stream, the mapper returns a list of key-value pairs. In the simplest case, each of these lists will have one element; the code will look something like

```
(define (my-mapper input-kv-pair)
  (list (make-kv-pair ... ...)))
```

The interface requires that you return a list to allow for the non-simplest cases: (1) Each input key-value pair may give rise to more than one output key-value pair. For example, you may want an output key-value pair for each *word* of the input file, whereas the input key-value pair represents an entire line:

```
(define (my-mapper input-kv-pair)
  (map (lambda (wd) (make-kv-pair ... ...))
       (kv-value input-kv-pair)))
```

(2) There are *three* commonly used higher order functions for sequential data, `map`, `accumulate`/`reduce`, and `filter`. The way `mapreduce` handles the sort of problem for which `filter` would ordinarily be used is to allow a mapper to return an empty list if this particular key-value pair shouldn't contribute to the result:

```
(define (my-mapper input-kv-pair)
  (if ...
      (list input-kv-pair)
      '()))
```

Of course it's possible to write mapper functions that combine these three patterns for more complicated tasks.

The keys in the kv-pairs returned by the mapper need not be the same as the key in the input kv-pair.

Instead of one big accumulation, there's a separate accumulation of values for each key. The non-parallel computation in the left half of the picture has two steps, a `map` and an `accumulate`. But the `mapreduce` computation has *three* steps; the middle step sorts all the key-value pairs produced by all the mapper processes by their keys, and combines all the kv-pairs with the same key into a single aggregate structure, which is then used as the input to a `reduce` process.

This is why the use of key-value pairs is important! If the data had no such structure imposed on them, there would be no way for us to tell `mapreduce` which data should be combined in each reduction.

Although it's shown as one big box, the sort is also done in parallel; it's a "bucket sort," in which each `map` process is responsible for sending each of its output kv-pairs to the proper `reduce` process. (Don't be confused; your mapper function doesn't have to do that. The `mapreduce` program takes care of it.)

Since all the data seen by a single `reduce` process have the same key, the reducer doesn't deal with keys at all. This is important because it allows us to use simple reducer functions such as `+`, `*`, `max`, etc. The Scheme interface to `mapreduce` recognizes the special cases of `cons` and `cons-stream` as reducers and does what you intend, even though it wouldn't actually work without this special handling, both because `cons-stream` is a special form and because the iterative implementation of `mapreduce` would do the combining in the wrong order.

In the underlying `mapreduce` software, each `reduce` process leaves its results in a separate file, stored on the particular processor that ran the process. But the Scheme interface to `mapreduce` returns a single value, a stream that effectively merges the results from all the `reduce` processes.

Running mapreduce: The `mapreduce` function is not available on the standard lab machines. You must connect to the machine that controls the parallel cluster. To do this, from the Unix shell you say this:

```
ssh icluster1.eecs.berkeley.edu
```

If you're at home, rather than in the lab, you'll have to provide your class login to the `ssh` command:

```
ssh cs61a-XY@icluster.eecs.berkeley.edu
```

replacing `XY` above with your login account. `Ssh` will ask for your password, which is the same on the parallel cluster as for your regular class account. Once you are logged into `icluster1`, you can run `stk` as usual, but `mapreduce` will be available:

```
(mapreduce mapper reducer reducer-base-case filename-or-special-stream)
```

The first three arguments are the mapper function for the `map` phase, and the reducer function and starting value for the `reduce` phase. The last argument is the data input to the `map`, but it is restricted to be either a distributed filesystem folder, which must be one of these:

`"/beatles-songs"`	This one is small and has all Beatles song names
`"/gutenberg/shakespeare"`	The collected works of William Shakespeare
`"/gutenberg/dickens"`	The collected works of Charles Dickens
`"/sample-emails"`	Some sample email data for the homework
`"/large-emails"`	A much larger sample email dataset. Use this only if you're willing to wait a while.

(the quotation marks above are required), or the stream returned by an earlier call to `mapreduce`. (Streams you make yourself with `cons-stream`, etc., can't be used.) Some problems are solved with two `mapreduce` passes, like this:

```
(define intermediate-result (mapreduce ...))
(mapreduce ... intermediate-result)
```

(Yes, you could just use one `mapreduce` call directly as the argument to the second `mapreduce` call, but in practice you'll want to use `show-stream` to examine the intermediate result first, to make sure the first call did what you expect.)

Here's a sample. We provide a file of key-value pairs in which the key is the name of a Beatles album and the value is the name of a song on that album. Suppose we want to know how many times each word appears in the name of a song:

```
(define (wordcount-mapper document-line-kv-pair)
  (map (lambda (wd-in-line) (make-kv-pair wd-in-line 1))
       (kv-value document-line-kv-pair)))

(define wordcounts (mapreduce wordcount-mapper + 0 "/beatles-songs"))

> (ss wordcounts)
```

The argument to `wordcount-mapper` will be a key-value pair whose key is an album name, and whose value is a song name. (In other examples, the key will be a filename, such as the name of a play by Shakespeare, and the value will be a line from the play.) We're interested only in the song names, so there's no call to `kv-key` in the procedure. For each song name, we generate a list of key-value pairs in which the key is a word in the name and the value is 1. This may seem silly, having the same value in every pair, but it means that in the `reduce` stage we can just use `+` as the reducer, and it'll add up all the occurrences of each word.

You'll find the running time disappointing in this example; since the number of Beatles songs is pretty small, the same computation could be done faster on a single machine. This is because there is a significant setup time both for `mapreduce` itself and for the `stk` interface. Since your mapper and reducer functions

have to work when run on parallel machines, your Scheme environment must be shipped over to each of those machines before the computation begins, so that bindings are available for any free references in your procedures. It's only for large amounts of data (or long computations that aren't data-driven, such as calculating a trillion digits of π, but `mapreduce` isn't really appropriate for those examples) that parallelism pays off.

By the way, if you want to examine the input file, you can't just say

```
(ss "/beatles-songs")        ; NO
```

because a distributed filename isn't a stream, even though the file itself is (when viewed by the `stk` interface to `mapreduce`) a stream. These filenames only work as arguments to `mapreduce` itself. But we can use `mapreduce` to examine the file by applying null transformations in the map and reduce stages:

```
(ss (mapreduce list cons-stream the-empty-stream "/beatles-songs"))
```

The mapper function is `list` because the mapper must always return a list of key-value pairs; in this case, `map` will call `list` with one argument and so it'll return a list of length one.

Now we'd like to find the most commonly used word in Beatle song titles. There are few enough words so that we could really do this on one processor, but as an exercise in parallelism we'll do it partly in parallel. The trick is to have each reduce process find the most common word starting with a particular letter. Then we'll have 26 candidates from which to choose the absolutely most common word on one processor.

```
(define (find-max-mapper kv-pair)
  (list (make-kv-pair (first (kv-key kv-pair))
                      kv-pair)))

(define (find-max-reducer current so-far)
  (if (> (kv-value current) (kv-value so-far))
      current
      so-far))

(define frequent (mapreduce find-max-mapper find-max-reducer
                            (make-kv-pair 'foo 0) wordcounts))

> (ss frequent)

> (stream-accumulate find-max-reducer (make-kv-pair 'foo 0)
                     (stream-map kv-value frequent))
```

This is a little tricky. In the `wordcounts` stream, each key-value pair has a word as the key, and the count for that word as the value: `(back . 3)`. The mapper transforms this into a key-value pair in which the key is the first letter of the word, and the value is *the entire input key-value pair*: `(b . (back . 3))`. Each `reduce` process gets all the pairs with a particular key, i.e., all the ones with the same first letter of the word. The reducer sees only the values from those pairs, but each value is itself a key-value pair! That's why the reducer has to compare the `kv-value` of its two arguments.

As another example, here's a way to count the total number of lines in all of Shakespeare's plays:

```
(define will (mapreduce (lambda (kv-pair) (list (make-kv-pair 'line 1)))
                        + 0 "/gutenberg/shakespeare"))
```

For each line in Shakespeare, we make exactly the same pair `(line . 1)`. Then, in the `reduce` stage, all the ones in all those pairs are added. But this is actually a bad example! Since all the keys are the same (the word `line`), only one `reduce` process is run, so the counting isn't done in parallel. A better way would be to count each play separately, then add those results if desired. You'll do that in lab.

CS 61A Lecture Notes Week 13

Topic: Analyzing evaluator

Reading:

To work with the ideas in this section you should first

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

in order to get the analyzing metacircular evaluator.

Inefficiency in the Metacircular Evaluator

Suppose we've defined the factorial function as follows:

```
(define (fact num)
  (if (= num 0)
      1
      (* num (fact (- num 1)))))
```

What happens when we compute `(fact 3)`?

```
eval (fact 3)
  self-evaluating?  ==>  #f     if-alternative  ==>  (* num (fact (- num 1)))
  variable?  ==>  #f               eval (* num (fact (- num 1)))
  quoted?  ==>  #f                   self-evaluating?  ==>  #f
  assignment?  ==>  #f               ...
  definition?  ==>  #f               list-of-values (num (fact (- num 1)))
  if?  ==>  #f                         ...
  lambda?  ==>  #f                     eval (fact (- num 1))
  begin?  ==>  #f                        ...
  cond?  ==>  #f                         apply <procedure fact> (2)
  application?  ==>  #t                    eval (if (= num 0) ...)
  eval fact
    self-evaluating?  ==>  #f
    variable?  ==>  #t
    lookup-variable-value  ==>  <procedure fact>
    list-of-values (3)
      eval 3  ==> 3
    apply <procedure fact> (3)
      eval (if (= num 0) ...)
        self-evaluating?  ==>  #f
        variable?  ==>  #f
        quoted?  ==>  #f
        assignment?  ==>  #f
        definition?  ==>  #f
        if?  ==>  #t
          eval-if (if (= num 0) ...)
            if-predicate  ==>  (= num 0)
              eval (= num 0)
                self-evaluating?  ==>  #f
                ...
```

Four separate times, the evaluator has to examine the procedure body, decide that it's an `if` expression, pull out its component parts, and evaluate those parts (which in turn involves deciding what type of expression each part is).

This is one reason why interpreted languages are so much slower than compiled languages: The interpreter does the syntactic analysis of the program over and over again. The compiler does the analysis once, and the compiled program can just do the part of the computation that depends on the actual values of variables.

Separating Analysis from Execution

`Eval` takes two arguments, an expression and an environment. Of those, the expression argument is (obviously!) the same every time we revisit the same expression, whereas the environment will be different each time. For example, when we compute `(fact 3)` we evaluate the body of `fact` in an environment in which `num` has the value 3. That body includes a recursive call to compute `(fact 2)`, in which we evaluate the same body, but now in an environment with `num` bound to 2.

Our plan is to look at the evaluation process, find those parts which depend only on `exp` and not on `env`, and do those only once. The procedure that does this work is called `analyze`.

What is the result of `analyze`? It has to be something that can be combined somehow with an environment in order to return a value. The solution is that `analyze` returns a procedure that takes only `env` as an argument, and does the rest of the evaluation.

Instead of

```
(eval exp env)  ==>  value
```

we now have

```
1.  (analyze exp)  ==>  exp-procedure
2.  (exp-procedure env)  ==>  value
```

When we evaluate the same expression again, we only have to repeat step 2. What we're doing is akin to memoization, in that we remember the result of a computation to avoid having to repeat it. The difference is that now we're remembering something that's only part of the solution to the overall problem, instead of a complete solution.

We can duplicate the effect of the original `eval` this way:

```
(define (eval exp env)
  ((analyze exp) env))
```

The Implementation Details

`Analyze` has a structure similar to that of the original eval:

```
(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ((variable? exp)
         (lookup-var-val exp env))
        ...
        ((foo? exp) (eval-foo exp env))
        ...))
```

```
(define (analyze exp)
  (cond ((self-evaluating? exp)
         (analyze-self-eval exp))
        ((variable? exp)
         (analyze-var exp))
        ...
        ((foo? exp) (analyze-foo exp))
        ...))
```

The difference is that the procedures such as `eval-if` that take an expression and an environment as arguments have been replaced by procedures such as `analyze-if` that take only the expression as argument.

How do these analysis procedures work? As an intermediate step in our understanding, here is a version of `analyze-if` that exactly follows the structure of `eval-if` and doesn't save any time:

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

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

This version of `analyze-if` returns a procedure with `env` as its argument, whose body is exactly the same as the body of the original `eval-if`. Therefore, if we do

```
((analyze-if some-if-expression) some-environment)
```

the result will be the same as if we'd said

```
(eval-if some-if-expression some-environment)
```

in the original metacircular evaluator.

But we'd like to improve on this first version of `analyze-if` because it doesn't really avoid any work. Each time we call the procedure that `analyze-if` returns, it will do all of the work that the original `eval-if` did.

The first version of `analyze-if` contains three calls to `eval`. Each of those calls does an analysis of an expression and then a computation of the value in the given environment. What we'd like to do is split each of those `eval` calls into its two separate parts, and do the first part only once, not every time:

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env)
      (if (true? (pproc env))
          (cproc env)
          (aproc env)))))
```

In this final version, the procedure returned by `analyze-if` doesn't contain any analysis steps. All of the components were already analyzed before we call that procedure, so no further analysis is needed.

The biggest gain in efficiency comes from the way in which `lambda` expressions are handled. In the original metacircular evaluator, leaving out some of the data abstraction for clarity here, we have

```
(define (eval-lambda exp env)
  (list 'procedure exp env))
```

The evaluator does essentially nothing for a `lambda` expression except to remember the procedure's text and the environment in which it was created. But in the analyzing evaluator we analyze the body of the procedure; what is stored as the representation of the procedure does not include its text! Instead, the evaluator represents a procedure in the metacircular Scheme as a procedure in the underlying Scheme, along with the formal parameters and the defining environment.

Level Confusion

The analyzing evaluator turns an expression such as

```
(if A B C)
```

into a procedure

```
(lambda (env)
  (if (A-execution-procedure env)
      (B-execution-procedure env)
      (C-execution-procedure env)))
```

This may seem like a step backward; we're trying to implement `if` and we end up with a procedure that does an `if`. Isn't this an infinite regress?

No, it isn't. The `if` in the execution procedure is handled by the underlying Scheme, not by the metacircular Scheme. Therefore, there's no regress; we don't call `analyze-if` for that one. Also, the `if` in the underlying Scheme is much faster than having to do the syntactic analysis for the `if` in the meta-Scheme.

So What?

The syntactic analysis of expressions is a large part of what a compiler does. In a sense, this analyzing evaluator is a compiler! It compiles Scheme into Scheme, so it's not a very useful compiler, but it's really not that much harder to compile into something else, such as the machine language of a particular computer.

A compiler whose structure is similar to this one is called a *recursive descent* compiler. Today, in practice, most compilers use a different technique (called a stack machine) because it's possible to automate the writing of a parser that way. (I mentioned this earlier as an example of data-directed programming.) But if you're writing a parser by hand, it's easiest to use recursive descent.

- **Software reliability: the Therac failures**

  - 6 accidents, 4 deaths
    - ▷ but 100s of lives saved

  - no bad guys (cf. Ford Pinto case)

  - Software doesn't degrade like hardware
    - ▷ but it rots anyway
    - ▷ but it has much greater complexity

    cf. Star Wars (birth of CPSR)

  - Continuum of life-or-deathness: Clearly Therac yes, clearly video game no.
    But what about OS, spreadsheet, etc.?

  - Therac bugs
    - ▷ no atomic test and set
    - ▷ hardware interlocks removed
    - ▷ UI problems:
      - ⋆ cursor position
      - ⋆ defaults
      - ⋆ too many error messages
    - ▷ documentation
    - ▷ organizational response
      easy to see after the fact, but problems are inherent in organizations (esp. ones that can be sued)

  - Solutions
    - ▷ redundancy
    - ▷ fail soft (work despite bugs)
    - ▷ audit trail
    - ▷ Software Engineering (an attitude about programming)
      - ⋆ Design techniques
        - ○ modularization (cf. OOP)
        - ○ understand concurrency (semaphores)
        - ○ analyze invariants
      - ⋆ Verification techniques
        - ○ correctness proofs
          (can't be perfect because of halting theorem but still useful)
        - ○ automatic analysis in compiler
      - ⋆ Debugging techniques
        - ○ black box vs. glass box
        - ○ don't break old code with new fix
        - ○ introduce bugs on purpose to analyze results downstream
        - ○ debug by subtraction, not addition

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

CS 61A Lecture Notes Week 14

Topic: Lazy evaluator, Nondeterministic evaluator

Reading: Abelson & Sussman, Section 4.2, 4.3

• **Lazy evaluator.** To load the lazy metacircular evaluator, say

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

Streams require careful attention

To make streams of pairs, the text uses this procedure:

```
;;;;;                          In file cs61a/lectures/4.2/pairs.scm
(define (pairs s t)
  (cons-stream
   (list (stream-car s) (stream-car t))
   (interleave
    (stream-map (lambda (x) (list (stream-car s) x))
                (stream-cdr t))
    (pairs (stream-cdr s) (stream-cdr t)))))
```

In exercise 3.68, Louis Reasoner suggests this simpler version:

```
(define (pairs s t)
  (interleave
   (stream-map (lambda (x) (list (stream-car s) x)) t)
   (pairs (stream-cdr s) (stream-cdr t))))
```

Of course you know because it's Louis that this doesn't work. But why not? The answer is that `interleave` is an ordinary procedure, so its arguments are evaluated right away, including the recursive call. So there is an infinite recursion before any pairs are generated. The book's version uses `cons-stream`, which is a special form, and so what looks like a recursive call actually isn't—at least not right away.

But in principle Louis is right! His procedure does correctly specify what the desired result should contain. It fails because of a detail in the implementation of streams. In a perfect world, a mathematically correct program such as Louis's version ought to work on the computer.

In section 3.5.4 they solve a similar problem by making the stream programmer use explicit `delay` invocations. (You skipped over that section because it was about calculus.) Here's how Louis could use that technique:

```
(define (interleave-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2)
      (cons-stream
       (stream-car s1)
       (interleave-delayed (force delayed-s2)
                           (delay (stream-cdr s1))))))

(define (pairs s t)
  (interleave-delayed
   (stream-map (lambda (x) (list (stream-car s) x)) t)
   (delay (pairs (stream-cdr s) (stream-cdr t)))))
```

This works, but it's far too horrible to contemplate; with this technique, the stream programmer has to check carefully every procedure to see what might need to be delayed explicitly. This defeats the object of an abstraction. The user should be able to write a stream program just as if it were a list program, without any idea of how streams are implemented!

Lazy evaluation: delay everything automatically

Back in chapter 1 we learned about *normal order evaluation*, in which argument subexpressions are not evaluated before calling a procedure. In effect, when you type

```
(foo a b c)
```

in a normal order evaluator, it's equivalent to typing

```
(foo (delay a) (delay b) (delay c))
```

in ordinary (applicative order) Scheme. If every argument is automatically delayed, then Louis's `pairs` procedure will work without adding explicit delays.

Louis's program had explicit calls to `force` as well as explicit calls to `delay`. If we're going to make this process automatic, when should we automatically force a promise? The answer is that some primitives need to know the real values of their arguments, e.g., the arithmetic primitives. And of course when Scheme is about to print the value of a top-level expression, we need the real value.

How do we modify the evaluator?

What changes must we make to the metacircular evaluator in order to get normal order?

We've just said that the point at which we want to automatically delay a computation is when an expression is used as an argument to a procedure. Where does the ordinary metacircular evaluator evaluate argument subexpressions? In this excerpt from `eval`:

```
(define (eval exp env)
  (cond ...
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        ...))
```

It's `list-of-values` that recursively calls `eval` for each argument. Instead we could make thunks:

```
(define (eval exp env)
  (cond ...
        ((application? exp)
         (apply (ACTUAL-VALUE (operator exp) env)
                (LIST-OF-DELAYED-VALUES (operands exp) env)))
        ...))
```

Two things have changed:

1. To find out what procedure to invoke, we use `actual-value` rather than `eval`. In the normal order evaluator, what `eval` returns may be a promise rather than a final value; `actual-value` forces the promise if necessary.

2. Instead of `list-of-values` we call `list-of-delayed-values`. The ordinary version uses `eval` to get the value of each argument expression; the new version will use `delay` to make a list of thunks. (This isn't quite true, and I'll fix it in a few paragraphs.)

When do we want to force the promises? We do it when calling a primitive procedure. That happens in `apply`:

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

We change it to force the arguments first:

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure (MAP FORCE ARGUMENTS)))
        ...))
```

Those are the crucial changes. The book gives a few more details: Some special forms must force their arguments, and the read-eval-print loop must force the value it's about to print.

Reinventing `delay` and `force`

I said earlier that I was lying about using `delay` to make thunks. The metacircular evaluator can't use Scheme's built-in `delay` because that would make a thunk in the underlying Scheme environment, and we want a thunk in the metacircular environment. (This is one more example of the idea of level confusion.) Instead, the book uses procedures `delay-it` and `force-it` to implement metacircular thunks.

What's a thunk? It's an expression and an environment in which we should later evaluate it. So we make one by combining an expression with an environment:

```
(define (delay-it exp env)
  (list 'thunk exp env))
```

The rest of the implementation is straightforward.

Notice that the `delay-it` procedure takes an environment as argument; this is because it's part of the implementation of the language, not a user-visible feature. If, instead of a lazy evaluator, we wanted to add a `delay` special form to the ordinary metacircular evaluator, we'd do it by adding this clause to `eval`:

```
((delay? exp) (delay-it (cadr exp) env))
```

Here `exp` represents an expression like `(delay foo)` and so its `cadr` is the thing we really want to delay.

The book's version of `eval` and `apply` in the lazy evaluator is a little different from what I've shown here. My version makes thunks in `eval` and passes them to `apply`. The book's version has `eval` pass the argument expressions to `apply`, without either evaluating or thunking them, and also passes the current environment as a third argument. Then `apply` either evaluates the arguments (for primitives) or thunks them (for non-primitives). Their way is more efficient, but I think this way makes the issues clearer because it's more nearly parallel to the division of labor between `eval` and `apply` in the vanilla metacircular evaluator.

Memoization

Why didn't we choose normal order evaluation for Scheme in the first place? One reason is that it easily leads to redundant computations. When we talked about it in chapter 1, I gave this example:

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

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

In a normal order evaluator, this adds 2 to 3 four times!

```
(square (square (+ 2 3)))  ==>
(* (square (+ 2 3)) (square (+ 2 3)))  ==>
(* (* (+ 2 3) (+ 2 3)) (* (+ 2 3) (+ 2 3)))
```

The solution is memoization. If we force the same thunk more than once, the thunk should remember its value from the first time and not have to repeat the computation. (The four instances of `(+ 2 3)` in the last line above are all the same thunk forced four times, not four separate thunks.)

The details are straightforward; you can read them in the text.

• **Nondeterministic evaluator**

To load the nondeterministic metacircular evaluator, say

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

Solution spaces, streams, and backtracking

Many problems are of the form "Find all A such that B" or "find an A such that B." For example: Find an even integer that is not the sum of two primes; find a set of integers a, b, c, and n such that $a^n + b^n = c^n$ and $n > 2$. (These problems might not be about numbers: Find all the states in the United States whose first and last letters are the same.)

In each case, the set A (even integers, sets of four integers, or states) is called the *solution space*. The condition B is a predicate function of a potential solution that's true for actual solutions.

One approach to solving problems of this sort is to represent the solution space as a stream, and use `stream-filter` to select the elements that satisfy the predicate:

```
(stream-filter sum-of-two-primes? even-integers)

(stream-filter Fermat? (pairs (pairs integers integers)
                              (pairs integers integers)))

(stream-filter (lambda (x) (equal? (first x) (last x))) states)
```

The stream technique is particularly elegant for infinite problem spaces, because the program seems to be generating the entire solution space A before checking the predicate B. (Of course we know that really the steps of the computation are reordered so that the elements are tested as they are generated.)

This week we consider a different way to express the same sort of computation, a way that makes the sequence of events in time more visible. In effect we'll say:

• Pick a possible solution.

• See if it's really a solution.

• If so, return it; if not, try another.

Here's an example of the notation:

```
> (let ((a (amb 2 3 4))
        (b (amb 6 7 8)))
    (require (= (remainder b a) 0))
    (list a b))
(2 6)
> try-again
(2 8)
> try-again
(3 6)
> try-again
(4 8)
> try-again
There are no more solutions.
```

The main new thing here is the special form `amb`. This is not part of ordinary Scheme! We are adding it as a new feature in the metacircular evaluator. `Amb` takes any number of argument expressions and returns the value of one of them. You can think about this using either of two metaphors:

• The computer clones itself into as many copies as there are arguments; each clone gets a different value.

• The computer magically knows which argument will give rise to a solution to your problem and chooses that one.

What really happens is that the evaluator chooses the first argument and returns its value, but if the computation later *fails* then it tries again with the second argument, and so on until there are no more to try. This introduces another new idea: the possibility of the failure of a computation. That's not the same thing as an error! Errors (such as taking the `car` of an empty list) are handled the same in this evaluator as in ordinary Scheme; they result in an error message and the computation stops. A failure is different; it's what happens when you call `amb` with no arguments, or when all the arguments you gave have been tried and there are no more left.

In the example above I used `require` to cause a failure of the computation if the condition is not met. `Require` is a simple procedure in the metacircular Scheme-with-`amb`:

```
(define (require condition)
  (if (not condition) (amb)))
```

So here's the sequence of events in the computation above:

```
a=2
    b=6; 6 is a multiple of 2, so return (2 6)

[try-again]
    b=7; 7 isn't a multiple of 2, so fail.
    b=8; 8 is a multiple of 2, so return (2 8)

[try-again]
    No more values for b, so fail.
a=3
    b=6; 6 is a multiple of 3, so return (3 6)

[try-again]
    b=7; 7 isn't a multiple of 3, so fail.
    b=8; 8 isn't a multiple of 3, so fail.
    No more values for b, so fail.
a=4
    b=6; 6 isn't a multiple of 4, so fail.
    b=7; 7 isn't a multiple of 4, so fail.
    b=8; 8 is a multiple of 4, so return (4 8)

[try-again]
    No more values for b, so fail.
No more values for a, so fail.
(No more pending AMBs, so report failure to user.)
```

Recursive `Amb`

Since `amb` accepts any argument expressions, not just literal values as in the example above, it can be used recursively:

```
(define (an-integer-between from to)
  (if (> from to)
      (amb)
      (amb from (an-integer-between (+ from 1) to))))
```

or if you prefer:

```
(define (an-integer-between from to)
  (require (>= to from))
  (amb from (an-integer-between (+ from 1) to)))
```

Further, since `amb` is a special form and only evaluates one argument at a time, it has the same delaying effect as `cons-stream` and can be used to make infinite solution spaces:

```
(define (integers-from from)
  (amb from (integers-from (+ from 1))))
```

This `integers-from` computation never fails—there is always another integer—and so it won't work to say

```
(let ((a (integers-from 1))
      (b (integers-from 1)))
  ...)
```

because `a` will never have any value other than 1, because the second `amb` never fails. This is analogous to the problem of trying to append infinite streams; in that case we could solve the problem with `interleave` but it's harder here.

Footnote on order of evaluation

In describing the sequence of events in these examples, I'm assuming that Scheme will evaluate the arguments of the unnamed procedure created by a `let` from left to right. If I wanted to be sure of that, I should use `let*` instead of `let`. But it matters only in my description of the sequence of events; considered abstractly, the program will behave correctly regardless of the order of evaluation, because all possible solutions will eventually be tried—although maybe not in the order shown here.

Success or failure

In the implementation of `amb`, the most difficult change to the evaluator is that any computation may either succeed or fail. The most obvious way to try to represent this situation is to have `eval` return some special value, let's say the symbol `=failed=`, if a computation fails. (This is analogous to the use of `=no-value=` in the Logo interpreter project.) The trouble is that if an `amb` fails, we don't want to continue the computation; we want to "back up" to an earlier stage in the computation. Suppose we are trying to evaluate an expression such as

```
(a (b (c (d 4))))
```

and suppose that procedures `b` and `c` use `amb`. Procedure `d` is actually invoked first; then `c` is invoked with the value `d` returned as argument. The `amb` inside procedure `c` returns its first argument, and `c` uses that to compute a return value that becomes the argument to `b`. Now suppose that the `amb` inside `b` fails. We don't want to invoke `a` with the value `=failed=` as its argument! In fact we don't want to invoke `a` at all; we want to re-evaluate the body of `c` but using the second argument to its `amb`.

A&S take a different approach. If an `amb` fails, they want to be able to jump right back to the previous `amb`, without having to propagate the failure explicitly through several intervening calls to `eval`. To make this

work, intuitively, we have to give `eval` two different places to return to when it's finished, one for a success and the other for a failure.

Continuations

Ordinarily a procedure doesn't think explicitly about where to return; it returns to its caller, but Scheme takes care of that automatically. For example, when we compute

```
(* 3 (square 5))
```

the procedure `square` computes the value 25 and Scheme automatically returns that value to the `eval` invocation that's waiting to use it as an argument to the multiplication. But we could tell `square` explicitly, "when you've figured out the answer, pass it on to be multiplied by 3" this way:

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

> (square 5 (lambda (y) (* y 3)))
75
```

A *continuation* is a procedure that takes your result as argument and says what's left to be done in the computation.

Continuations for success and failure

In the case of the nondeterministic evaluator, we give `eval` *two* continuations, one for success and one for failure. Note that these continuations are part of the implementation of the evaluator; the user of `amb` doesn't deal explicitly with continuations.

Here's a handwavy example. In the case of

```
(a (b (c (d 4))))
```

procedure `b`'s success continuation is something like

```
(lambda (value) (a value))
```

but its failure continuation is

```
(lambda () (a (b (redo-amb-in-c))))
```

This example is handwavy because these "continuations" are from the point of view of the user of the metacircular Scheme, who doesn't know anything about continuations, really. The true continuations are written in underlying Scheme, as part of the evaluator itself.

If a computation fails, the most recent `amb` wants to try another value. So a continuation failure will redo the `amb` with one fewer argument. There's no information that the failing computation needs to send back to that `amb` except for the fact of failure itself, so the failure continuation procedure needs no arguments.

On the other hand, if the computation succeeds, we have to carry out the success continuation, and that continuation needs to know the value that we computed. It also needs to know what to do if the continuation itself fails; most of the time, this will be the same as the failure continuation we were given, but it might not be. So a success continuation must be a procedure that takes two arguments: a value and a failure continuation.

The book bases the nondeterministic evaluator on the analyzing one, but I'll use a simplified version based on plain old eval (it's in `cs61a/lib/vambeval.scm`).

Most kinds of evaluation always succeed, so they invoke their success continuation and pass on the failure one. I'll start with a too-simplified version of `eval-if` in this form:

```
(define (eval-if exp env succeed fail)           ; WRONG!
  (if (eval (if-predicate exp) env succeed fail)
      (eval (if-consequent exp) env succeed fail)
      (eval (if-alternative exp) env succeed fail)))
```

The trouble is, what if the evaluation of the predicate fails? We don't then want to evaluate the consequent or the alternative. So instead, we just evaluate the predicate, giving it a success continuation that will evaluate the consequent or the alternative, supposing that evaluating the predicate succeeds.

In general, wherever the ordinary metacircular evaluator would say

```
(define (eval-foo exp env)
   (eval step-1 env)
   (eval step-2 env))
```

using `eval` twice for part of its work, this version has to `eval` the first part with a continuation that `eval`s the second part:

```
(define (eval-foo exp env succeed fail)
  (eval step-1
        env
        (lambda (value-1 fail-1)
          (eval step-2 env succeed fail-1))
        fail))
```

(In either case, `step-2` presumably uses the result of evaluating `step-1` somehow.)

Here's how that works out for `if`:

```
(define (eval-if exp env succeed fail)
  (eval (if-predicate exp)            ; test the predicate
        env
        (lambda (pred-value fail2)    ; with this success continuation
          (if (true? pred-value)
              (eval (if-consequent exp) env succeed fail2)
              (eval (if-alternative exp) env succeed fail2)))
        fail))                        ; and the same failure continuation
```

What's `fail2`? It's the failure continuation that the evaluation of the predicate will supply. Most of the time, that'll be the same as our own failure continuation, just as `eval-if` uses `fail` as the failure continuation to pass on to the evaluation of the predicate. But if the predicate involves an `amb` expression, it will generate a new failure continuation. Think about an example like this one:

```
> (if (amb #t #f)
      (amb 1)
      (amb 2))
1

> try-again
2
```

(A more realistic example would have the predicate expression be some more complicated procedure call that had an `amb` in its body.) The first thing that happens is that the first `amb` returns `#t`, and so `if` evaluates its second argument, and that second `amb` returns 1. When the user says to try again, there are no more values for that `amb` to return, so it fails. What we must do is re-evaluate the first `amb`, but this time returning its second argument, `#f`. By now you've forgotten that we're trying to work out what `fail2` is for in `eval-if`, but this example shows why the failure continuation when we evaluate `if-consequent` (namely the `(amb 1)` expression) has to be different from the failure continuation for the entire `if` expression. If the entire `if`

fails (which will happen if we say `try-again` again) then its failure continuation will tell us that there are no more values. That continuation is bound to the name `fail` in `eval-if`. What ends up bound to the name `fail2` is the continuation that re-evaluates the predicate `amb`.

How does `fail2` get that binding? When `eval-if` evaluates the predicate, which turns out to be an `amb` expression, `eval-amb` will evaluate whatever argument it's up to, but with a new failure continuation:

```
(define (eval-amb exp env succeed fail)
  (if (null? (cdr exp))             ; (car exp) is the word AMB
      (fail)                        ; no more args, call failure cont.
      (eval (cadr exp)              ; Otherwise evaluate the first arg
            env
            succeed                 ; with my same success continuation
            (lambda ()              ; but with a new failure continuation:
              (eval-amb (cons 'amb (cddr exp))    ; try the next argument
                        env
                        succeed
                        fail)))))
```

Notice that `eval-if`, like most other cases, provides a new success continuation but passes on the same failure continuation that it was given as an argument. But `eval-amb` does the opposite: It passes on the same success continuation it was given, but provides a new failure continuation.

Of course there are a gazillion more details, but the book explains them, once you understand what a continuation is. The most important of these complications is that anything involving mutation is problematic. If we say

```
(define x 5)
(set! x (+ x (amb 2 3)))
```

it's clear that the first time around `x` should end up with the value 7 $(5+2)$. But if we try again, we'd like `x` to get the value 8 $(5+3)$, not 10 $(7+3)$. So `set!` must set up a failure continuation that undoes the change in the binding of `x`, restoring its original value of 5, before letting the `amb` provide its second argument.

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:

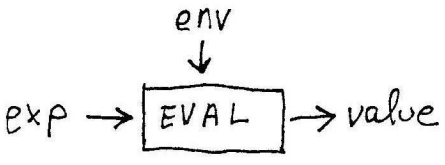
```
;;;;;                          In file cs61a/lectures/4.4/reverse.scm
(assert! (rule (backward (?a . ?x) ?y)
               (and (append ?z (?a) ?y)
                    (backward ?x ?z) )))

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

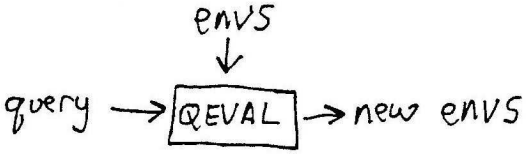
But it's much harder to write one that works both ways. Even as we speak, logic programming fans are trying to push the limits of the idea, but right now, you still have to understand something about the below-the-line algorithm to be confident that your logic program won't loop.

• 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.



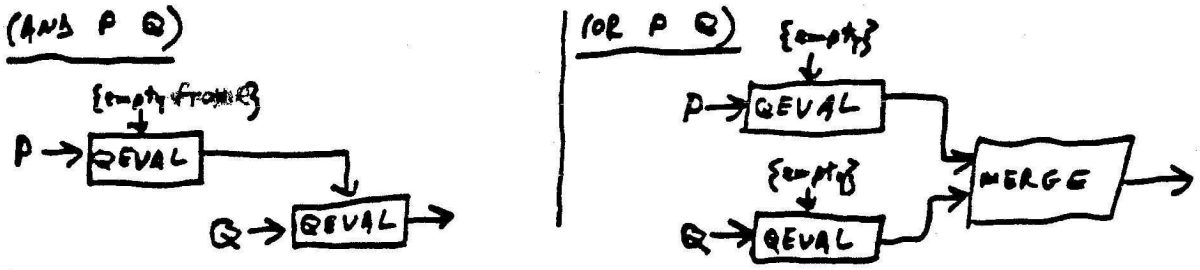


The "collection" of environments we're talking about here is represented as a stream. That's because there might be infinitely many of them! We use the stream idea to reorder the computation; what really happens is that we take one potential set of satisfying values and work it all the way through; then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

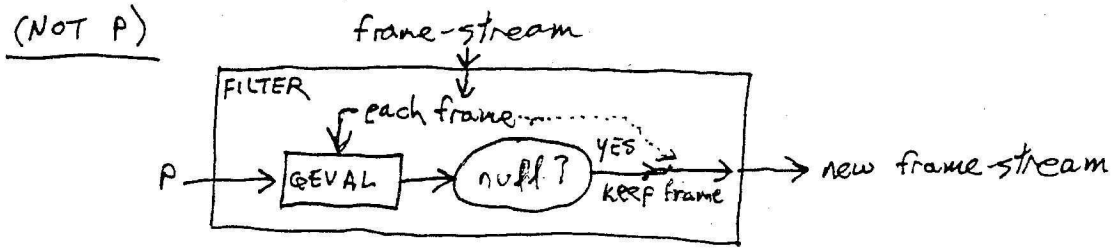
If the query is `(or p q)` then we use `qeval` to evaluate each of the pieces independently, starting in both cases with the empty-environment stream. Then we *merge* the two result streams to get a stream of bindings that satisfy either `p` or `q`.



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)

Client/Server Programming:
event-driven process (idle if nothing to do)
callback from operating system for events
cooperation among separate computers
(vocabulary: client, server, IP address, port, socket, thread)

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!