MIT 6.001 1986 Video Notes

Nebhrajani A.V.

February 15, 2022

Contents

1 Prologue

1.1 About

These are my notes of the twenty SICP lectures of June 1986, produced by Hewlett-Packard Television. These videos are available under a Creative Commons license.

These notes aim to be concise and as example-heavy as possible. The language used and referred to as "Lisp" is MIT-Scheme. These notes, however, use the SICP language provided by Racket, a modern Scheme dialect. This is because Racket's integration with Emacs and Org mode is orders of magnitude better than MIT-Scheme's. In general, all "Lisp" code looks exactly the same as in SICP, with the exception of having to prefix some numbers with `#i` to ensure Racket treats them as imprecise.

Note that Racket's SICP language creates mutable pairs (`mpair`'s) by default, as in MIT Scheme. Therefore, `set-car!` and `set-cdr!` are available as primitives.

1.2 License

2 Lecture 1A: Overview and Introduction to Lisp

Computer science isn't really a science, and it isn't really about computers. Computer science is the study of how-to or imperative knowledge (as opposed to declarative knowledge). To illustrate the difference, consider:

$$y = \sqrt{x} \text{ such that } y^2 = x, y \geq 0$$

This is declarative, in that we could recognize if y is the square root of x given x and y, but we're no closer to knowing how to *find* y if we are given x. Imperative knowledge would look like:

To find the square root y of x:

- Make a guess g.
- If g^2 is close enough to x, $y = g$.
- Otherwise, make a new guess equal to the average of g and x/g.

  This method will eventually come up with a g close enough to the actual square root y of x.

  Computer science focuses on this kind of imperative knowledge, and, specifically, how to communicate that knowledge to a computer.

2.1 Managing Complexity: Key Ideas of 6.001

Computer science is also about managing complexity, in that large programs that you can't hold in your head should still be manageable and easy to work with. We explore this theme in 6.001 by learning three key ideas:

- Black-box abstractions
- Conventional interfaces
- Metalinguistic abstraction.

2.2 Let's Learn Lisp

When learning a new language, always ask about its:

- Primitive elements,
- Means of combination, and
- Means of abstraction.

2.2.1 Primitive Elements

These are numbers like 3, 17.4, or 5. Other primitives are discussed later in the course.

```
4
17.4
5
```

```
4
17.4
5
```

2.2.2 Means of Combination

Lisp's numerical primitives can be combined with "operations" such as addition, written in prefix notation.

```
(+ 3 17.4 5)
```

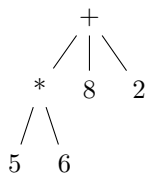
```
25.4
```

Other basic operations are provided by Lisp, such as multiplication and division. Of course, combinations can be combined recursively:

```
(+ 3 (* 5 6) 8 2)
```

```
43
```

This should show you the tree structure inherent in all of Lisp:



In Lisp, () is the application of an operation or function in prefix notation.

2.2.3 Means of Abstraction

Abstraction can simply be done by naming things. Giving complicated things a name prevents us from having to understand how the thing the name refers to *works*, and instead lets us "abstractly" use the name for our purposes.

```
(define a (* 5 5))
a
(* a a)
(define b (+ a (* 5 a)))
b
(+ a (/ b 5))
```

```
25
625
150
55
```

Now, it's often more useful to abstract away imperative how-to knowledge. Consider:

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

```
(square 10)
```

```
100
```

This defines `square` as a function taking a single argument `x`, and returning `(* x x)`. Note that this way of writing a define is actually "syntactic sugar" for:

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

(square 25)
```

```
625
```

`lambda (x)` means "make a procedure that takes argument `x`". The second argument to lambda is the actual procedure body. The `define` names this anonymous procedure `square`.

Just like we can use combinations recursively, so we can abstractions. Consider:

```
(define (average x y)
  (/ (+ x y) 2))
```

```
(define (mean-square x y)
  (average (square x)
           (square y)))

(mean-square 2 3)
```

```
13/2
```

Note the indentation: since Lisp is parenthesis heavy, we use indentation. Good editors like Emacs should do this automatically.

2.3 Case Analysis in Lisp

To represent functions like:

$$abs(x) = \begin{cases} -x & x < 0 \\ 0 & x = 0 \\ x & x > 0 \end{cases}$$

Lisp needs some form of conditional execution. In Lisp, this function would look like:

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

```
(abs -3)
(abs 0)
(abs 5)
```

```
3
0
5
```

`cond` takes any number of arguments. Each argument must be structured as `(predicate) (consequent)`. If `predicate` is true, we do the `consequent`. Otherwise, we don't. Lisp also provides a way to write conditionals that only have two branches (an if-else):

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

```
(abs -11)
(abs 0)
(abs 33)
```

```
11
0
33
```

`cond` and `if` are syntactical sugar for each other. The Lisp implementation picks any one and defines the other in terms of it.

We now know most of Lisp. Lisp doesn't have `do...while` or `for`, since anything a loop can do can be done via recursion.

2.4 Finding Square Roots

Remember our square root finding algorithm?

To find the square root y of x:

- Make a guess g.
- If g^2 is close enough to x, $y = g$.
- Otherwise, make a new guess equal to the average of g and x/g.

Or, in Lisp,

```
(define (try g x)
  (if (good-enough? g x)
      g
      (try (improve g x) x)))
```

This is a form of programming called "wishful thinking": we assume `good-enough?` (good enough predicate) and `improve` are already implemented. Now that we can try a guess and improve it till it's good enough, we can write a simple square root function:

```
(define (sqrt x)
  (try 1 x))
```

This function simply starts the guess at 1, then improves it. Let's now write the functions we don't have:

```
(define (improve g x)
  (average g (/ x g)))
```

```
(define (good-enough? g x)
  (< (abs (- (square g) x))
     0.00001))
```

This tests if g^2 is within 0.0001 of x. Putting it all together, we can finally try to find square roots:

```
(sqrt #i2)
(sqrt #i3)
(sqrt #i4)
```

```
1.4142156862745097
1.7320508100147274
2.0000000929222947
```

> **Note:** The `#i4` is Racket's syntax for using imprecise (decimals) instead of precise (fractions). Ignore it, and treat it as the number `4`.

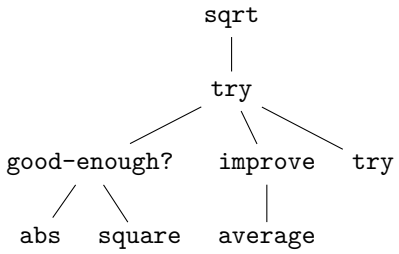
See that `try` actually runs a loop, but does so recursively, calling itself every time the `if` condition fails to improve the guess. Also note that these functions can all be nested inside the square root function to hide them from the outer scope, thus:

```
(define (sqrt x)
  (define (good-enough? g)
    (define (square g)
      (* g g))
    (define (abs y)
      (if (< y 0)
          (- y)
          y))
    (< (abs (- (square g) x))
       0.0001))
  (define (improve g)
    (define (average y z)
      (/ (+ y z) 2))
    (average g (/ x g)))
  (define (try g)
    (if (good-enough? g)
        g
        (try (improve g))))
  (try 1))
```

```
20
21 (sqrt #i2)
```

```
1.4142156862745097
```

This program should also show you a tree-like dependency of the functions, with each function containing the definitions of the functions it depends on. For someone using `sqrt`, all the functions within it are hidden.



This discipline of writing procedures is called lexical scoping.

2.5 Inbuilt/Primitive Procedures Aren't Special

```
1 square
2 +
```

```
#<procedure:square>
#<procedure:+>
```

3 Lecture 1B: Procedures and Processes, Substitution Model

3.1 Substitution Rule/Model

The substitution rule states that,

> To evaluate an application:
>
> - Evaluate the operator to get procedure.
> - Evaluate the operands to get arguments.
> - Apply procedure to arguments.
>   - Copy body of procedure.
>   - Replace formal parameters with actual arguments.
> - Evaluate new body.

Note that this isn't necessarily how the *interpreter* evaluates a Lisp application, but the substitution rule is a "good enough" model for our purposes.

3.1.1 Kinds of Expressions in Lisp

- Numbers (evaluate to "themselves")
- Symbols (represent some procedure)
- Combinations
- λ-expressions (used to build procedures)
- Definitions (used to name symbols)
- Conditionals

  We will focus our use of the substitution rule on the first three. The last three are called "special forms", and we'll worry about them later.

3.1.2 Example

Consider:

```
1 (define (sum-of-squares x y)
2   (+ (square x) (square y)))
3
4 (sum-of-squares 3 4)
```

```
25
```

Let's try to apply the substitution rule to our application,

```
(sum-of-squares 3 4)
(+ (square 3) (square 4))
(+ (square 3) (* 4 4))
(+ (square 3) 16)
(+ (* 3 3) 16)
(+ 9 16)
25
```

3.2 Peano Arithmetic

3.2.1 Simple Peano Addition

Peano arithmetic defines addition as:

```
(define (pa+ x y)
  (if (= x 0)
      y
      (pa+ (dec x) (inc y))))
```

```
(pa+ 3 4)
```

```
7
```

Assume that `inc` and `dec` are primitives available that increment and decrement the argument respectively. How is the procedure `pa+` working? Let's apply the substitution rule.

```
(pa+ 3 4)
(if (= 3 0)
    4
    (pa+ (dec 3) (inc 4)))
(pa+ 2 5)
...
(pa+ 1 6)
...
(pa+ 0 7)
7
```

We're skipping some steps, but the idea is that `x` keeps giving one "unit" to `y` until it reaches zero. Then the sum is `y`. Written with steps skipped:

```
(pa+ 3 4)
(pa+ 2 5)
(pa+ 1 6)
(pa+ 0 7)
7
```

3.2.2 Another Peano Adder

Consider:

```
(define (pb+ x y)
  (if (= x 0)
      y
      (inc (pb+ (dec x) y))))
```

This is also a Peano adder: but it's implemented *slightly* differently syntax-wise, a few characters here and there. Let's use the substitution rule to see how it works.

```
(pb+ 3 4)
(inc (pb+ 2 4))
(inc (inc (pb+ 1 4)))
(inc (inc (inc (pb+ 0 4))))
(inc (inc ((inc 4))))
(inc (inc 5))
(inc 6)
7
```

See that it *does* work:

```
(pb+ 3 4)
```

```
7
```

Now, consider how these two, `pa+` and `pb+`, are different. While the *procedures* do the same thing, the processes are wildly different. Let's discuss their time and space complexity. It should be obvious to you that the time complexity is the vertical axis in the substitution rule application, since the interpreter "executes" these instructions line by line. More lines means more time.

In the case of `pa+`, the number of lines increases by 1 if you increase input `x` by 1. Thus, the time complexity is $O(x)$. Similarly, in the case of `pb+`, the number of lines increases by 2 (once in the expansion, once in the contraction) when you increase `x` by 1. Thus, it is also $O(x)$.

Now, the horizontal axis shows us how much space is being used. In the case of `pa+`, the space used is a constant. Thus, $O(1)$. On the other hand, see that `pb+` first *expands* then *contracts*. The length of the maximum expansion increases by 1 if we increase x by 1, since there's one more increment to do. Thus, $O(x)$.

Now, we call a process like `pa+` *linear iterative* and a process like `pb+` *linear recursive*.

Process	Time Complexity	Space Complexity	Type
`pa+`	$O(x)$	$O(1)$	Linear iterative
`pb+`	$O(x)$	$O(x)$	Linear recursive

Note that the *process* `pa+` being iterative has nothing to do with the implementation/definition of the *procedure*, which is recursive. Iteration refers to the constant space requirement.

3.3 Differentiating Between Iterative and Recursive Processes

One of the primary ways to differentiate between an iterative and recursive process is to imagine what'd happen if you turned the computer off, then resumed the current computation.

In a recursive process, we've lost some important information: how deep into the recursion we are. In the `pb+` example, we wouldn't know how many `inc`'s deep we are (information stored in the RAM by the interpreter, not by the process), meaning that we can't return the right value.

In an iterative process, we can pick up right where we left off, since *all* state information is contained by the process.

3.4 Fibonacci Numbers

Fibonacci numbers are defined as:

$$F(x) = \begin{cases} 0, & x = 0 \\ 1, & x = 1 \\ F(x-1) + F(x-2), & \text{otherwise} \end{cases}$$

The series itself is:

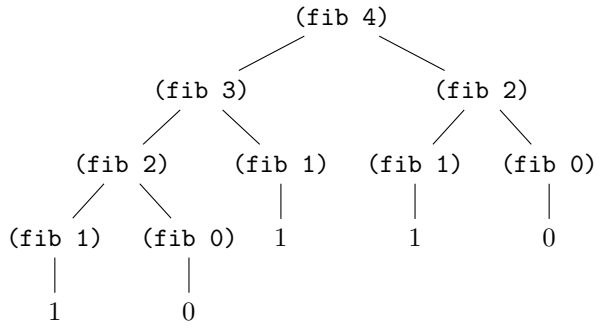
$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 \ldots$$

Let's write a Lisp function to calculate the nth Fibonacci number, assuming 0 is the 0th.

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))
(fib 10)
```

```
55
```

It works, that's true. But how *well* does it work. Let's see. When we call (say) `(fib 4)`, we also call `(fib 3)` and `(fib 2)`, both of which also call ... let's draw it:



A tree! Clearly, this is an exponential-time process, since computing $n + 1$ takes exponentially more effort. Also note that it's a pretty bad process, since we constantly recompute many values. The space complexity is the maximum depth of the tree (depth of recursion), which is at most n. Therefore, the time complexity is $O(\text{fib}(n))$ and space complexity is $O(n)$.

It is useful to try and write an iterative Fibonacci with better performance as an exercise.

3.5 Towers of Hanoi

From Wikipedia:

> The Tower of Hanoi is a mathematical game or puzzle. It consists of three rods and a number of disks of different diameters, which can slide onto any rod. The puzzle starts with the disks stacked on one rod in order of decreasing size, the smallest at the top, thus approximating a conical shape. The objective of the puzzle is to move the entire stack to the last rod, obeying the following simple rules:
>
> - Only one disk may be moved at a time.
> - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or an empty rod.
> - No disk may be placed on top of a disk that is smaller than it.

Let's try to solve Hanoi for 4 disks, from rod A to rod C. Again — "wishful thinking". Let's assume that we know how to solve for 3 disks. To solve, we'd take the top 3 disks, put it on the spare rod B. Then, we'd take the fourth and largest disk, and put it on destination rod C. Finally, we'd move the three disk pile from B to C. Solved!

But wait — to solve the 3 disk case, let's assume we know how to solve the 2 disk case.

To solve the 2 disk case, we should know how to solve the one disk case, which is just moving a disk from a rod to another.

Or, in Lisp,

```
(define (move n from to spare)
  (cond ((= n 1) (display "Move disk at rod ")
                 (display from)
                 (display " to rod ")
                 (display to)
                 (display ".\n"))
        (else
         (move (- n 1) from spare to)
         (move 1 from to spare)
         (move (- n 1) spare to from))))

(move 4 "A" "C" "B")
```

```
Move disk at rod A to rod B.
Move disk at rod A to rod C.
Move disk at rod B to rod C.
Move disk at rod A to rod B.
Move disk at rod C to rod A.
Move disk at rod C to rod B.
Move disk at rod A to rod B.
```

```
Move disk at rod A to rod C.
Move disk at rod B to rod C.
Move disk at rod B to rod A.
Move disk at rod C to rod A.
Move disk at rod B to rod C.
Move disk at rod A to rod B.
Move disk at rod A to rod C.
Move disk at rod B to rod C.
```

Note, of course, that this procedure too, is an exponential time procedure. However, any procedure for Hanoi will be exponential time, since for n disks, Hanoi requires 2^{n-1} moves. Even if you compute every move in $O(1)$ (which we do, since it's just a print), the complexity will be $O(2^n)$.

3.6 Iterative Fibonacci

```
(define (iter-fib n a b)
  (if (= n 1)
      b
      (iter-fib (dec n) b (+ a b))))

(define (fib n)
  (iter-fib n 0 1))
```

```
(fib 10)
```

```
55
```

4 Lecture 2A: Higher-Order Procedures

4.1 Abstracting Procedural Ideas

Consider the functions and their respective (recursive) procedures:

$$\sum_{i=a}^{b} i$$

```
(define (sum-int a b)
  (if (> a b)
      0
      (+ a
         (sum-int (inc a) b))))

(sum-int 0 10)
```

```
55
```

$$\sum_{i=a}^{b} i^2$$

```
(define (sum-sq a b)
  (if (> a b)
      0
      (+ (square a)
         (sum-sq (inc a) b))))

(sum-sq 0 4)
```

```
30
```

$$\sum_{i=a \text{ by } 4}^{b} \frac{1}{i(i+2)}$$

Note that this series estimates $\pi/8$.

```
(define (sum-pi a b)
  (if (> a b)
      0
      (+ (/ 1
            (* a (+ a 2)))
         (sum-pi (+ a 4) b))))

(* 8 (sum-pi #i1 #i1000000))
```

```
3.141590653589793
```

See that the commonality between these procedures comes from the fact that the notion of "summation" from `a` to `b` is the same, but the *function* being summed is different in each case. Or, in general form:

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

The way to solve this is by writing a procedure `sum`, which has available to it two procedures `term` and `next`. We supply these as arguments. Consider:

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

When we call `sum` recursively, see that we pass to it the *same procedures* `term` and `next`, along with `b` and the next value of `a`. Now, it is easy to define `sum-int`, `sum-sq`, and `sum-pi` using `sum`, thus:

```
(define (sum-int a b)
  (define (identity x) x)
  (sum identity
       a
       inc
       b))

(sum-int 0 10)
```

```
55
```

`identity` is the function $p(x) = x$.

```
(define (sum-sq a b)
  (sum square
       a
       inc
       b))

(sum-sq 0 4)
```

```
30
```

```
(define (sum-pi a b)
  (sum (lambda (x)
         (/ 1
            (* x (+ x 2))))
       a
```

```
        (lambda (x) (+ x 4))
        b))

(* 8 (sum-pi #i1 #i1000000))
```

```
3.141590653589793
```

Recall that `lambda` means "make a procedure" that is nameless. In `sum-pi`, we choose to give `sum` anonymous functions as arguments instead of defining our own, because there's no reason to name a procedure we won't later use.

The big advantage of abstracting away `sum` this way is that in case we want to implement it in a different way, we merely have to change the implementation of one function (`sum`) and not that of the three functions that use it. In fact, those functions can remain exactly the same.

Here's another implementation of `sum`. See that `sum-pi` still works without changes, because it doesn't care about how `sum` is implemented as long as the argument number and order remains constant.

```
(define (sum term a next b)
  (define (iter j ans)
    (if (> j b)
        ans
        (iter (next j)
              (+ (term j)
                 ans))))
  (iter a 0))

(define (sum-pi a b)
  (sum (lambda (x)
         (/ 1
            (* x (+ x 2))))
       a
       (lambda (x) (+ x 4))
       b))

(* 8 (sum-pi #i1 #i1000000))
```

```
3.1415906535898936
```

4.2 More on Square Roots

Recall our square root procedure. When seen in Lisp code, it's not very clear what it's doing, or how it's working.

```
(define (sqrt x)
  (define (good-enough? g)
    (define (square g)
      (* g g))
```

```
    (define (abs y)
      (if (< y 0)
          (- y)
          y))
    (< (abs (- (square g) x))
       0.0001))
  (define (improve g)
    (define (average y z)
      (/ (+ y z) 2))
    (average g (/ x g)))
  (define (try g)
    (if (good-enough? g)
        g
        (try (improve g))))
  (try 1))
```

```
(sqrt #i2)
```

```
1.4142156862745097
```

Let's use higher-order procedure abstraction to make it clearer.

4.2.1 Fixed Points

Recall that the algorithm itself relies on writing a function

$$f : y \mapsto \frac{y + \frac{x}{y}}{2}$$

Note that this works because $f(\sqrt{x}) = \sqrt{x}$:

$$f(\sqrt{x}) = \frac{\sqrt{x} + \frac{x}{\sqrt{x}}}{2} = \frac{2\sqrt{x}}{2} = \sqrt{x}$$

See that this is *actually* an algorithm for finding a fixed point of a function f, which is defined as finding the point where $f(z) = z$. This algorithm is merely an instance of a function f whose fixed point happens to be the square root.

For some functions, the fixed point can be found by iterating it.

This is the top-level abstraction we'll write a function for. First, let's see how we'd write a square-root function by wishful thinking:

```
(define (sqrt x)
  (fixed-point
   (lambda (y) (average (/ x y)
                        y))
   1))
```

Now writing `fixed-point`:

```
(define (fixed-point f start)
  (define (close-enough-p x y)
    (< (abs (- x y))
       0.00001))
  (define (iter old new)
    (if (close-enough-p old new)
        new
        (iter new (f new))))
  (iter start (f start)))
```

Let's try it out!

```
(sqrt #i2)
```

```
1.4142135623746899
```

4.2.2 Damping Oscillations

A fair question when seeing the function

$$f_1 \colon y \mapsto \frac{y + \frac{x}{y}}{2}$$

is why another function

$$f \colon y \mapsto \frac{x}{y}$$

wouldn't work in its place. This question is best answered by trying to find its fixed point by iteration. Let's try to find it for $x = 2$, starting at $y = 1$. Then,

$$\begin{aligned} f(1) &= \frac{2}{1} = 2 \\ f(2) &= \frac{2}{2} = 1 \\ f(1) &= \frac{2}{1} = 2 \\ f(2) &= \frac{2}{2} = 1 \\ &\dots \end{aligned}$$

It seems that instead of converging, this function is *oscillating* between two values. We know that it's easy to fix this: we have to damp these oscillations. The most natural way to do this is to take the average of successive values y and $f(y)$. A `sqrt` function that uses average damping would be:

```
(define (sqrt x)
  (fixed-point
   (avg-damp (lambda (y) (/ x y)))
   1))
```

The `avg-damp` function takes in a procedure, creates an average damping procedure, and returns it. Or, in Lisp:

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

It is worth discussing how `avg-damp` works. It is defined as a procedure which takes the argument of a function `f`. It then returns an anonymous procedure which takes an argument `x`, and computes the average of $f(x)$ and x. This is finally the highest level of abstraction we can reach for the `sqrt` algorithm — finding the fixed point of a damped oscillating function.

Using the `sqrt` function,

```
(sqrt #i2)
```

```
1.4142135623746899
```

4.3 Newton's Method

Newton's method is used to find the zeros of a function ($y \ni f(y) = 0$). To use it, start with some guess y_0. Then,

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}$$

where

$$f'(y) = \frac{\mathrm{d}f(y)}{\mathrm{d}y}$$

We can, of course, find the zero of the square root finding function $f(y) = x - y^2$ using Newton's method. Note that Newton's method *itself* is based on fixed points, since it aims to find a fixed point where $y_{n+1} \approx y_n$.

Defining `sqrt`:

```
(define (sqrt x)
  (newton (lambda (y) (- x (square y)))
          1))
```

We pass to `newton` a function $f(y) = x - y^2$, since its zero is $x = y^2$.

```
(define (newton f guess)
  (define df (deriv f))
  (fixed-point
   (lambda (x) (- x
                  (/ (f x)
                     (df x))))
   guess))
```

It is important to note that defining `df` to be `(deriv f)` once prevents wasteful recomputation of `df` every time `fixed-point` calls itself.

Of course, we now have to define a derivative function. We can simply use the standard limit definition to find it numerically:

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Or, in Lisp,

```
(define dx 0.0000001)

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

This function returns a function which is the derivative of `f`, and can be used as such. Consider:

```
((deriv (lambda (x) (* x x x))) 2)
```

```
12.000000584322379
```

Which is the expected value of differentiating x^3 w.r.t x ($3x^2$) and evaluating at 2.

Testing out our `sqrt` function:

```
(sqrt #i2)
```

```
1.4142135623747674
```

4.4 Procedures are First-Class Citizens

This means that procedures can be:

- Named using variables.
- Passed as arguments to procedures.
- Returned as values from procedures.
- Included in data structures.

5 Lecture 2B: Compound Data

Consider our `sqrt` function that uses `good-enough?`. What we did while writing `sqrt` is assume the existence of `good-enough?`. That is, we divorced the task of building `sqrt` from the task of implementing its parts.

Let's do this for data.

5.1 Rational Number Arithmetic

Let's design a system which can add fractions:

$$\frac{1}{2} + \frac{1}{4} = \frac{3}{4}$$

and multiply them:

$$\frac{3}{4} \times \frac{2}{3} = \frac{1}{2}$$

The *procedures* for these two tasks are well known to most people:

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_2}{d_1 d_2}$$

and

$$\frac{n_1}{d_1} \times \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

5.1.1 Abstraction

We don't know, however, how to represent this data in a Lisp procedure. Let's use our powerful "wishful thinking" strategy. Assume that we have the following procedures available to us:

- A constructor `(make-rat n d)` which makes a fraction with numerator `n` and denominator `d`.
- Two selectors:
  - `(numer x)` which takes in a fraction `x` and returns its numerator.
  - `(denom x)` which takes in a fraction `x` and returns its denominator.

  Then, our procedures are easy to write:

```
1  (define (+rat x y)
2    (make-rat
3     (+ (* (numer x) (denom y))
4        (* (numer y) (denom x)))
5     (* (denom x) (denom y))))
6
7  (define (*rat x y)
```

```
  (make-rat
   (* (numer x) (numer y))
   (* (denom x) (denom y))))
```

Why do we need this data object abstraction anyway? We could very well define `+rat` to take in four numbers, two numerators and two denominators. But to return, we can't return *both* numerator and denominator. We now have to define two summation functions, one for the numerator and one for the denominator, and somehow keep track of the fact that one of these numbers is the numerator and the other the denominator. Furthermore, when applying more complex operations like:

```
(*rat (+rat x y)
      (+rat s t))
```

The data abstraction helps. If it weren't there, we'd have to maintain some temporary registers to store the numerator and denominator values of the `+rat` operations into, then pass them to `*rat`.

Worse than confusing the program, such a design philosophy would confuse us, the programmers.

5.1.2 Data Object Creation

The glue we use to stick two numbers together is provided by three Lisp primitives:

- A constructor `cons`, which generates an ordered pair.
- Two selectors:
  - `car`, which selects the first element of the pair, and
  - `cdr`, which selects the second element of the pair.

  In use,

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

```
1
2
```

  We can now write the procedures that we'd deferred writing earlier:

```
(define (make-rat x y)
  (cons x y))

(define (numer x)
  (car x))

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

```
(define x (make-rat 1 2))
(define y (make-rat 1 4))
(define z (+rat x y))
(numer z)
(denom z)
```

```
6
8
```

Agh. We forgot to reduce results to the simplest form. We can easily include this in the `make-rat` procedure:[1]

```
(define (make-rat x y)
  (let ((g (gcd x y)))
    (cons (/ x g)
          (/ y g))))

(define (numer x)
  (car x))

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

Note that we could shift the `gcd` bit to functions `numer` and `denom`, which would display the simplest form at access time rather than creation time. Deciding between the two is a matter of system efficiency: a system which displays often should use creation time simplification, while a system which creates many fractions should use access time simplification. We now need a GCD function:

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

We can now use `+rat` in *exactly* the same way, since the interface is the same. This is the advantage of abstraction.

```
(define x (make-rat 1 2))
(define y (make-rat 1 4))
(define z (+rat x y))
(numer z)
(denom z)
```

```
3
4
```

Excellent: we now have a working system. The data abstraction model can be visualised as follows:

[1] `let` is a Lisp primitive which takes as its first argument a series of definitions, and second input a series of applications that may use these definitions. The trick is that these definitions are only valid in the body (second argument) of `let`, effectively creating a local namespace.

+rat, *rat ...
make-rat, numer, denom
gcd
Pairs

At each layer of abstraction, we merely care about the usage of the lower layers and not their implementation or underlying representation.

5.2 Representing Points on a Plane

This is now an easy problem — the code should be self-explanatory.

```
(define (make-vec x y)
  (cons x y))

(define (xcor v)
  (car v))

(define (ycor v)
  (cdr v))
```

We could now define a segment as a pair of vectors:

```
(define (make-seg v w)
  (cons v w))

(define (seg-start s)
  (car s))

(define (seg-end s)
  (cdr s))
```

Some sample operations:

```
(define (midpoint s)
  (let ((a (seg-start s))
        (b (seg-end s)))
    (make-vec
     (average (xcor a) (xcor b))
     (average (ycor a) (ycor b)))))

(define (length s)
  (let ((dx (- (xcor (seg-end s))
               (xcor (seg-start s))))
        (dy (- (ycor (seg-end s))
               (ycor (seg-start s)))))
    (sqrt (+ (square dx)
```

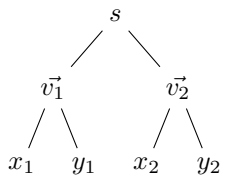
```
              (square dy)))))

(define side-a (make-vec #i3 #i0))
(define side-b (make-vec #i0 #i4))
(define segment (make-seg side-a side-b))

(length segment)

(define mp (midpoint segment))

(xcor mp)
(ycor mp)
```

```
5.000000000053722
1.5
2.0
```

The abstraction layer diagram of this code is:

Segments
Vectors
Pairs

It is interesting to note that segments are pairs of vectors, which are pairs of numbers, so segments are actually pairs of pairs. Represented as a tree:



This property is called *closure* (from abstract algebra[2]): that means of combination can be nested recursively. It's an important and powerful technique.

For instance, a three-dimensional vector can be represented by a pair whose one element is a number and whose other element is a pair of numbers. Or, in Lisp:

```
(define three-d-vec (cons 3 (cons 4 5)))
(car three-d-vec)
(car (cdr three-d-vec))
(cdr (cdr three-d-vec))
```

```
3
4
5
```

[2]For an operation defined on members of a set, the result of that operation is a member of the set. For instance, addition on natural numbers.

5.3 Pairs

Let's go back to when we assumed that `make-rat`, `numer`, and `denom`, were already implemented. The procedures we then wrote were written using *abstract data*, with the only "assured" property being that:

```
if x = (make-rat n d):
```

$$\frac{\texttt{numer x}}{\texttt{denom x}} = \frac{\texttt{n}}{\texttt{d}}$$

Beyond this basic "spec", or the interface contract, we know nothing about its implementation.

Now, it's easy not to appreciate how knowing *merely* the specification of the layer below is sufficient to use it, so let's discuss how pairs work. When we wanted to implement `make-rat`, we kind of "cheated" in that we said, "Okay, Lisp has a primitive to do this so we don't have to implement a pair." Let's now take a look at a possible implementation of a pair that doesn't use data objects at all, and instead mimics them from thin air. Consider:

```
(define (our-cons a b)
  (lambda (pick)
    (cond ((= pick 1) a)
          ((= pick 2) b))))

(define (our-car x) (x 1))
(define (our-cdr x) (x 2))
```

```
(define pair (our-cons 3 4))
(our-car pair)
(our-cdr pair)
```

```
3
4
```

Before thinking about how it works: consider the fact that Lisp's pairs could be implemented this way, and not only would we not know about this while implementing `make-rat` — we wouldn't care, since it's below the level of abstraction we're working at. As long as it behaves the way we expect it to — that is, it follows the "spec", we don't know or care about its implementation[3]. Such is the power of abstraction.

Now, how is this implementation even working? Well:

- `cons` is a procedure that returns a lambda (anonymous procedure) which, by the substitution model, looks like:

```
(lambda (pick)
  (cond ((= pick 1) 3)
        ((= pick 2) 4)))
```

[3]Note that Lisp actually implements pairs using "real" data structures, since using procedures this way is less efficient.

- `car` expects this procedure as an input, and returns the result of supplying this procedure with the value 1. This is naturally the first of the two numbers given to `cons` (`a`).
- `cdr` is identical to `car`, except that *it* supplies the input procedure with argument 2 to get `b`.

We can thus implement a pair "data structure" using only lambdas. In fact, these pairs are closed:

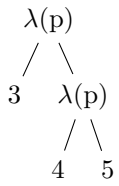
```
(define three-d-vec (our-cons 3 (our-cons 4 5)))
(our-car three-d-vec)
(our-car (our-cdr three-d-vec))
(our-cdr (our-cdr three-d-vec))
(our-cdr three-d-vec)
```

```
3
4
5
#<procedure:...6f_i/ob-2136OZJ.rkt:4:2>
```

It is worth thinking about the structure of `three-d-vec`:

```
(lambda (pick)
  (cond ((= pick 1) 3)
        ((= pick 2) (lambda (pick)
                      (cond ((= pick 1) 4)
                            ((= pick 2) 5))))))
```

Picking `2` in the top-level lambda gives us another lambda, in which we can pick either the first number (4) or the second (5). Note that this is precisely the nested pair structure we were going for.



6 Lecture 3A: Henderson Escher Example

Recall our vector procedures:

```
(define (make-vec x y)
  (cons x y))

(define (xcor v)
  (car v))

(define (ycor v)
  (cdr v))
```

We could define more procedures using these:

```
(define (+vect v1 v2)
  (make-vec
   (+ (xcor v1) (xcor v2))
   (+ (ycor v1) (ycor v2))))

(define (scale v s)
  (make-vec
   (* s (xcor v))
   (* s (ycor v))))
```

Recall that our representation of a line segment was as a pair of vectors, or pair of pairs. That is, we can use the property of closure that pairs have to store any amount of data.

6.1 Lists

Often, we want to store a sequence of data. Using pairs, there are many ways to do this, for instance:

```
(cons (cons 1 2) (cons 3 4))
(cons (cons 1 (cons 2 3)) 4)
```

```
((1 . 2) 3 . 4)
((1 2 . 3) . 4)
```

However, we want to establish a conventional way of dealing with sequences, to prevent having to make ad-hoc choices. Lisp uses a representation called a list:

```
(cons 1 (cons 2 (cons 3 (cons 4 nil))))
```

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

Note that the `nil` represents the null or empty list. Since writing so many `cons` is painful, Lisp provides the primitive `list` which lets us build such a structure.

```
(list 1 2 3 4)
```

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

Note that `list` is merely syntactic sugar for building up using pairs:

```
(define one-to-four (list 1 2 3 4))
```

```
(car one-to-four)
(cdr one-to-four)
(car (cdr one-to-four))
(cdr (cdr one-to-four))
(car (cdr (cdr (cdr one-to-four))))
(cdr (cdr (cdr (cdr one-to-four))))
```

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

Note that the empty list, `nil`, is also represented by `()`. This way of walking down the list for elements is called `cdr`-ing down a list, but it's a bit painful. Thus, when we want to process lists, we write procedures.

6.1.1 Procedures on Lists

Say we wanted to write a procedure `scale-list` which multiplies every element in the list by a certain value. That is, when scale list is called on `one-to-four` with value 10, it returns `(10 20 30 40)`. Here's one possible (recursive) implementation:

```
(define (scale-list l scale)
  (if (null? l)
      nil
      (cons (* scale (car l))
            (scale-list (cdr l) scale))))

(scale-list one-to-four 10)
```

```
(10 20 30 40)
```

`null?` is a predicate which tells us whether the given input is the empty list. This will be the case at the end of the list. Of course, this is *actually* a general method for processing all values of a list and returning another list, so we write a higher-order procedure which applies a procedure to all elements of a list and returns the result as a list, called `map`.

```
(define (map p l)
  (if (null? l)
      nil
      (cons (p (car l))
            (map p (cdr l)))))
```

Now defining `scale-list` in terms of `map`:

```
(define (scale-list l s)
  (map (lambda (x) (* x s))
       l))

(scale-list one-to-four 20)
```

```
(20 40 60 80)
```

We can now square lists:

```
(map square one-to-four)
```

```
(1 4 9 16)
```

Similar to `map`, we define a higher-order procedure `for-each`, which, instead of `cons`-ing a list and returning it, simply applies to procedure to each element of the list.

```
(define (for-each proc l)
  (cond ((null? l) done)
        (else
         (proc (car l))
         (for-each proc (cdr l)))))
```

6.2 Henderson's Picture Language

Let's define a language. As usual, we'll concern ourselves with its primitives, means of combination, and means of abstraction, implementing some of this language in Lisp along the way.

6.2.1 Primitives

This language has only one primitive: "picture", which is a figure scaled to fit a frame.

6.2.2 Means of Combination and Operations

- Rotate, which rotates a picture and returns it.
- Flip, which flips the picture across an axis and returns it.
- Beside, which takes two pictures and a scale, then puts the two next to each other, returning a picture.
- Above, like beside, but above.

  See that the closure property (that an operation on pictures returns a picture)[1] allows us to combine these operations/means of combination to build complex pictures with ease.

  Let's now implement this part of the language.

[1] $p \otimes p = p$

6.2.3 An Implementation

1. Frames

   Three vectors are needed to uniquely identify a frame on the plane. By convention, we take these to be the bottom left corner ("origin"), the bottom right corner ("horizontal") and the top left corner ("vertical"). Their positions can be described relative to the $(0, 0)$ of the display screen. Therefore, frame is implemented by:

   - Constructor `make-frame`.
   - Selectors `origin`, `horiz`, and `vert`, for the three vectors.

     Note that technically, a frame describes a transformation of the unit square, where each point in the unit square:

     $$(x, y) \mapsto \texttt{origin} + x \cdot \texttt{horiz} + y \cdot \texttt{vert}$$

     We can define a procedure which returns a procedure which maps a pair of points (x, y) on the unit square to a given frame:

```
(define (coord-map rect)
  (lambda (point)
    (+vect
     (+vect (scale (xcor point)
                   (horiz rect))
            (scale (ycor point)
                   (vert rect)))
     (origin rect))))
```

     `coord-map` returns a procedure which given a point will map it correctly to `rect`.

2. Pictures

   We can now easily define a procedure which makes a picture:

```
(define (make-picture seglist)
  (lambda (rect)
    (for-each
     (lambda (s)
       (drawline
        ((coord-map rect) (seg-start s))
        ((coord-map rect) (seg-end s))))
     seglist)))
```

   Well, relatively easily. Let's explain what `make-picture` actually does:

   - Takes argument `seglist`, which is a list of line segments (pairs of vectors) that the picture is.
   - Returns a procedure which:
     - Takes the argument of a frame.

- For every element in `seglist`:
  - Draws the segment within frame, by scaling it correctly using `coord-map`.
  - This is done by giving `coord-map` the frame to scale to.
  - The procedure returned by `coord-map` then scales the vectors `(seg-start s)` and `(seg-end s)` to the frame.
  - This can now be drawn by `drawline`, since it has as arguments two points.

    Note that a picture is *actually* a procedure which draws itself inside a given frame, and `make-picture` generates this procedure from a `seglist`. Or, in use:

```
(define R (make-frame ;some vectors
           ))
(define draw-george-in-frame (make-picture ;some seglist
                              ))
(draw-george-in-frame R)
```

3. Beside

   `beside` needs to draw two pictures on the screen, after scaling them correctly (by `a`) and placing them side by side. Thus, `beside` returns a picture which takes in an argument `rect`. `beside` starts drawing the left picture at `(origin rect), (scale a (horiz rect)) (vert rect)` and the right picture at `(+vect (origin rect) (scale a (horiz rect))), (scale (- 1 a) (horiz rect)), (vert rect)`. This places the two pictures side by side and scales them correctly within `rect`. Or, in Lisp,

```
(define (beside p1 p2 a)
  (lambda (rect)
    (p1 (make-frame
         (origin rect)
         (scale a (horiz rect))
         (vert rect)))
    (p2 (make-frame
         (+vect (origin rect)
                (scale a (horiz rect)))
         (scale (-1 a) (horiz rect))
         (vert rect)))))
```

4. Rotate-90

   To rotate a picture by 90 degrees counter-clockwise, all we have to do is make the `origin` shift to where `horiz` is, then draw the new `horiz` and `vert` correctly. With some vector algebra, the procedure in Lisp is:

```
(define (rot90 pict)
  (lambda (rect)
    (pict (make-frame
           (+vect (origin rect)
                  (horiz rect))
```

```
            (vert rect)
            (scale -1 (horiz rect))))))
```

6.2.4 Means of Abstraction

See that the picture language is now embedded in Lisp. We can write recursive procedures to modify a picture:

```
(define (right-push pict n a)
  (if (= n 0)
      pict
      (beside pict
              (right-push pict (dec n) a)
              a)))
```

We can even write a higher order procedure for "pushing":

```
(define (push comb)
  (lambda (pict n a)
    ((repeated
      (lambda (p)
        (comb pict p a))
      n)
     pict)))

(define right-push (push beside))
```

There's a lot to learn from this example:

- We're embedding a language inside Lisp. All of Lisp's power is available to this small language now: including recursion.
- There's no difference between a procedure and data: we're passing pictures around exactly like data, even though it's actually a procedure.
- We've created a layered system of abstractions on top of Lisp, which allows *each layer* to have all of Lisp's expressive power. This is contrasted to a designing such a system bottom-up as a tree, which would mean that:
  - Each node does a very specific purpose and is limited in complexity because a new feature has to be built ground-up at the node.
  - Making a change is near impossible, since there's no higher order procedural abstraction. Making a change that affects more than one node is a nightmare.

7 Lecture 3B: Symbolic Differentiation; Quotation

We saw that robust system design involves insensitivity to small changes, and that embedding a language within Lisp allows this. Let us turn to a somewhat similar thread, solving the problem of symbolic differentiation in Lisp.

This problem is somewhat different from *numerical* differentiation of a function like we did for Newton's method, since we actually want the expressions we work with to be in an algebraic language. Before figuring out how to implement such a thing, let's talk about the operation of differentiation itself.

7.1 Differentiation v. Integration

Why is it so much easier to differentiate than to integrate? Let us look at the basic rules of differentiation:

$$\frac{\mathrm{d}k}{\mathrm{d}x} = 0$$
$$\frac{\mathrm{d}x}{\mathrm{d}x} = 1$$
$$\frac{\mathrm{d}k \cdot a}{\mathrm{d}x} = k \cdot \frac{\mathrm{d}a}{\mathrm{d}x}$$
$$\frac{\mathrm{d}(a+b)}{\mathrm{d}x} = \frac{\mathrm{d}a}{\mathrm{d}x} + \frac{\mathrm{d}b}{\mathrm{d}x}$$
$$\frac{\mathrm{d}(ab)}{\mathrm{d}x} = a \cdot \frac{\mathrm{d}b}{\mathrm{d}x} + \frac{\mathrm{d}a}{\mathrm{d}x} \cdot b$$
$$\frac{\mathrm{d}x^n}{\mathrm{d}x} = nx^{n-1}$$

See that these rules are reduction rules, in that the derivative of some complex thing is the derivative of simpler things joined together by basic operations. Such reduction rules are naturally recursive in nature. This makes the problem of differentiation very easy to solve using simple algorithms.

On the other hand, implementing an integration system is a much harder problem, since such a system would require us to go the other way, combining up simpler expressions to make more complicated ones, which often involves an intrinsically difficult choice to make.

With these simple recursive rules in mind, let's implement a symbolic differentiation system.

7.2 Some Wishful Thinking

```
(define (deriv expr var)
  (cond ((constant? expr var) 0)
        ((same-var? expr var) 1)
        ((sum? expr)
```

```
          (make-sum (deriv (a1 expr) var)
                    (deriv (a2 expr) var)))
         ((product? expr)
          (make-sum
           (make-product (m1 expr)
                         (deriv (m2 expr) var))
           (make-product (deriv (m1 expr) var)
                         (m2 expr))))))
```

That's enough rules for now, we can add more later.

Note that `a1` is a procedure returning the first term of the addition $x + y$ (in this case, x), and `a2` is a procedure returning the second (in this case, y). Similar for multiplication, `m1` and `m2`.

All the `-?` procedures are predicates, and should be self-explanatory. `make-`, as expected, makes the object with given arguments as values and returns it. These are a level of abstraction below `deriv`, and involve the actual representation of algebraic expressions. Let's figure out how to do this.

7.3 Representing Algebraic Expressions

7.3.1 Using Lisp Syntax

One very simple way to represent expressions is to use Lisp's way: expressions that form trees. Consider:

$$ax^2 \mapsto (* \texttt{ a } (* \texttt{ x x}))$$
$$bx + c \mapsto (+ \ (* \texttt{ b x}) \texttt{ c})$$

This has the advantage that representing such expression is just a list. Moreover, finding out the operation is merely the `car` of the list, and the operands are the `cdr`. This effectively eliminates our need for parsing algebraic expressions.

7.3.2 Representation Implementation

Let's start defining our procedures.

```
(define (constant? expr var)
  (and (atom? expr)
       (not (eq? expr var))))

(define (same-var? expr var)
  (and (atom? expr)
       (eq? expr var)))

(define (sum? expr)
  (and (not (atom? expr))
       (eq? (car expr) '+)))
```

```
12
13 (define (product? expr)
14   (and (not (atom? expr))
15        (eq? (car expr) '*)))
```

We see a new form here: `'+` and `'*`. This is called "quoting". Why do we need to do this? Consider:

> "Say your name!"
> "Susanne."
> "Say 'your name'!"
> "Your name."

To differentiate the cases where we mean *literally* say "your name" and the case where we actually ask what "your name" *is*, we use quotation marks in English. Similarly, quoting a symbol in Lisp tells the interpreter to check *literally* for `(car expr)` to be the symbol `+` and not the procedure `+`.

Quotation is actually quite a complicated thing. Following the principle of substituting equals for equals, consider:

> "Chicago" has seven letters.
> Chicago is the biggest city in Illinois.
> "The biggest city in Illinois" has seven letters.

The first two statements are true, and quotation marks are used correctly in the first to show that we're talking about Chicago the word and not Chicago the city. However, the third statement is wrong entirely (although it is the result of changing equals for equals), because the phrase "The biggest city in Illinois" does not have seven letters. That is, we cannot substitute equals for equals in referentially opaque contexts.

Note that the `'` symbol breaks the neat pattern of Lisp where all expressions are delimited by `()`. To resolve this, we introduce the special form `(quote +)`, which does the exactly same thing as `'+`.

Now defining the constructors:

```
1 (define (make-sum a1 a2)
2   (list '+ a1 a2))
3
4 (define (make-product m1 m2)
5   (list '* m1 m2))
```

Finally, we must define the selectors:

```
1 (define a1 cadr)
2 (define a2 caddr)
3
4 (define m1 cadr)
5 (define m2 caddr)
```

`cadr` is the `car` of the `cdr` and `caddr` is the `car` of the `cdr` of the `cdr`. These are forms provided for convenience while programming, since list processing a big part of Lisp.[1]

[1] LISP actually stands for LISt Processing.

Let's try it out:

```
(deriv '(+ (* a (* x x)) (+ (* b x) c)) 'x)
(deriv '(+ (* a (* x x)) (+ (* b x) c)) 'a)
(deriv '(+ (* a (* x x)) (+ (* b x) c)) 'b)
(deriv '(+ (* a (* x x)) (+ (* b x) c)) 'c)
```

```
(+ (+ (* a (+ (* x 1) (* 1 x))) (* 0 (* x x))) (+ (+ (* b 1) (* 0 x)) 0))
(+ (+ (* a (+ (* x 0) (* 0 x))) (* 1 (* x x))) (+ (+ (* b 0) (* 0 x)) 0))
(+ (+ (* a (+ (* x 0) (* 0 x))) (* 0 (* x x))) (+ (+ (* b 0) (* 1 x)) 0))
(+ (+ (* a (+ (* x 0) (* 0 x))) (* 0 (* x x))) (+ (+ (* b 0) (* 0 x)) 1))
```

Note the recursive nature of `deriv`: the process creates results with the same shape even when we differentiate with respect to some other variable. This is because the recursion only ends when an expression is decomposed to either `same-var?` or `constant?`.

7.3.3 Simplification

However, these results are ugly, and we know why — there's no simplification. Technically, it's correct:

$$\begin{aligned} & a(1x+1x)+0x^2+b+0x+0 \\ =& 2ax+b \end{aligned}$$

Note that we've faced this same problem before with fractions, and recall that the solution was to change the constructors so that they'd simplify while creating the lists. Consider:

```
(define (make-sum a1 a2)
  (cond ((and (number? a1)
              (number? a2))
         (+ a1 a2))
        ((and (number? a1)
              (= a1 0))
         a2)
        ((and (number? a2)
              (= a2 0))
         a1)
        (else
         (list '+ a1 a2))))

(define (make-product m1 m2)
  (cond ((and (number? m1)
              (number? m2))
         (* m1 m2))
        ((and (number? m1)
              (= m1 0))
         0)
```

```
        ((and (number? m2)
              (= m2 0))
         0)
        ((and (number? m1)
              (= m1 1))
         m2)
        ((and (number? m2)
              (= m2 1))
         m1)
        (else
         (list '+ m1 m2))))
```

Now trying `deriv`:

```
(deriv '(+ (* a (* x x)) (+ (* b x) c)) 'x)
(deriv '(+ (* a (* x x)) (+ (* b x) c)) 'a)
(deriv '(+ (* a (* x x)) (+ (* b x) c)) 'b)
(deriv '(+ (* a (* x x)) (+ (* b x) c)) 'c)
```

```
(+ (+ a (+ x x)) b)
(* x x)
x
1
```

Excellent, these are much better. Note, of course, that we could simplify the first one further, but, in general, algebraic simplification is a painful problem, since the definition of simplest form varies with application. However, this is good enough.

7.4 On Abstract Syntax

Note that the syntax we used was abstract in the sense that it had its own rules and grammar. However, since it followed Lisp's syntax closely, we needed quotation to allow full expression.

This is a powerful paradigm: not only can we use meta-linguistic abstraction to create languages embedded within Lisp, but we can also use Lisp to interpret any syntax. We'll see more of this in the future.

8 Lecture 4A: Pattern Matching and Rule-Based Substitution

It's a funny technique we used last time, converting the rules of differentiation to Lisp. In fact, if we wanted to explain (say) the rules of algebra to the computer, we'd have to again create a similar program which converts the rules of algebra to Lisp.

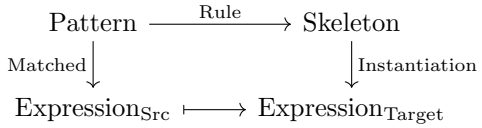
See that there's a higher-order idea here, of explaining rules to Lisp and having the rules applied to an input expression to "simplify" it. Our style of writing a rule-based substitution program is:

Rules → conditional → dispatch

That is, we try the rules on the given expression. If there's a match, we "dispatch" the result to substitute. Now, in general, the application of a rule is:

- Compare LHS of rule to input expression.
- If match, RHS with substituted values is replacement.

  Or, diagrammatically:



  Let us now build a simple language to express these rules, which can then be pattern matched, skeletons created, then instantiated.

8.1 Rule Language

Here's a sample bit of what we want the rule language to look like:

```
(define deriv-rules
  '(
    ((dd (?c c) (? v)) 0)
    ((dd (?v v) (? v)) 1)
    ((dd (?v u) (? v)) 0)

    ((dd (* (?c c) (? x)) (? v)) (* (: c) (dd (: x) (: v))))

    ((dd (+ (? x1) (? x2)) (? v))
     (+ (dd (: x1) (: v))
        (dd (: x2) (: v))))

    ((dd (* (? x1) (? x2)) (? v))
     (+ (* (: x1) (dd (: x2) (: v)))
        (* (: x2) (dd (: x1) (: v))))))
```

```
    ; ...
    ))
```

It is worth explaining what this syntax means exactly, because eventually, we want to parse it.

The rules are a list of pairs. The `car` of each pair is the pattern to match (rule LHS), and the `cdr` is the skeleton substitution expression (rule RHS).

8.1.1 Pattern Matching

The idea of the LHS language is to provide a framework where certain constructs can be matched and possibly named. These names will then be passed to the skeleton instantiator.[1]

Syntax	Meaning
`foo`	Matches itself literally.
`(f a b)`	Matches every 3-list whose `car` is `f`, `cadr` is `a`, and `caddr` is `b`.
`(? x)`	Matches any expression, and calls it `x`.
`(?c x)`	Matches an expression which is a constant, and calls it `x`.
`(?v x)`	Matches an expression which is a variable, and calls it `x`.

8.1.2 Skeleton and Instantiation

The RHS language provides a skeleton wherein values provided by the LHS language can be substituted.

Syntax	Meaning
`foo`	Instantiates `foo`.
`(f a b)`	Instantiates each element of the list and returns a list.
`(: x)`	Instantiate the value of `x` provided by the pattern matcher.

8.2 Desired Behaviour

We expect to use this program by calling a procedure called `simplifier`, to which we provide the list of rules. The procedure should return another procedure, which is able to apply the rules to a given input expression. Or, in Lisp:

```
(define dsimp
  (simplifier deriv-rules))

(dsimp '(dd (+ x y) x))

(+ 1 0)
```

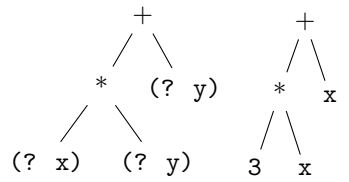
[1] We use "initiate" and "substitute" interchangeably to mean swapping out expressions in the skeleton provided by the RHS of the rules.

8.3 Implementation

We implement a procedure `match`, which takes a pattern, an expression, and a dictionary as arguments. If that pattern matches the expression, it writes the `?` values to the dictionary and returns it. Next, we implement `instantiate`, which takes as arguments a skeleton and a dictionary, and substitutes variables in the skeleton with their dictionary values. Finally, this new expression is returned to the `match`-er to match more patterns. Finally, we implement `simplify`, which takes in a list of the rules and applies these in a match-instantiate cycle until the expression cannot be further simplified (no more change after a round of match-instantiate).

8.3.1 Matcher

Abstractly, the job of the matcher is to do a tree traversal and comparison. Consider the rule LHS: `(+ (* (? x) (? y)) (? y))`, and an expression to match: `(+ (* 3 x) x)` (say). Then, the trees are:



Clearly, these expressions should be matched in the tree traversal. Don't confuse the `(? x)` in the rule LHS with the symbol `x` in the expression: for the rule, it's just a matching variable, but the `x` in the expression goes into the dictionary.

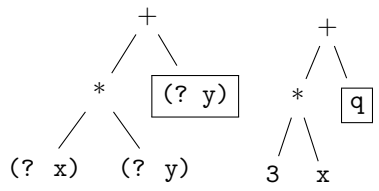
Let's now write our first implementation of `match`:

```
(define (match pat expr dict)
  (cond ((eq? dict 'failed) 'failed)
        ; ... some other cases
        ((atom? expr) 'failed)
        (else
         (match (cdr pat)
           (cdr expr)
           (match (car pat)
             (car expr)
             dict)))))
```

Before we write the entire procedure, let's observe how the general case (`else`) works, because that's the bit which does the tree traversal. It calls `match` on the `car` of the pattern and expression. If they match, we return a dictionary, which is then used to match the `cdr` of the original expression. Why does this do tree traversal? Well, it's basically a depth first search on a tree. Consider what happens when `match` is called on the `car`. After being called for the `caar` and `caaar`…it'll eventually be called for the `cadr` and the `caddr` and so on. That's precisely what a depth first search is. It'll keep going deeper and deeper until it fails, which is when it takes one step back and goes deeper into another branch.

Now, it is important to define the non-general cases, especially the ones that terminate `match`. The

most important of these is when the expression passed is atomic, since that means we're at the leaf of the expression tree. Another possible failure is the insertion into the dictionary failing. How is this possible? Consider:



Just before `match` reaches the last leaf (boxed), our dictionary will look something like the following:

rule-vars	expr-vals
x	3
y	x

However, when it tries to insert `y: q`, the dictionary will `failed` to do so, because `y` already has value `x`, and thus the rule does not match.

Finally, we implement as more cases in the `cond` other things we may need to match, according to the rule language (8.1).

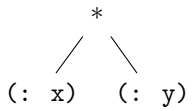
```
(define (match pattern expression dict)
  (cond ((eq? dict 'failed) 'failed)

        ((atom? pattern)
         (if (atom? expression)
             (if (eq? pattern expression)
                 dict
                 'failed)
             'failed))

        ((arbitrary-constant? pattern)
         (if (constant? expression)
             (extend-dict pattern expression dict)
             'failed))

        ((arbitrary-variable? pattern)
         (if (variable? expression)
             (extend-dict pattern expression dict)
             'failed))

        ((arbitrary-expression? pattern)
         (extend-dict pattern expression dict))

        ((atom? expression) 'failed)

```

```
26          (else
27           (match (cdr pattern)
28             (cdr expression)
29             (match (car pattern)
30               (car expression)
31               dict)))))
```

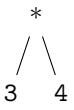
We add one case wherein both the pattern to be matched and the expression are atomic and the same, (the case `foo` in 8.1), in which we simply return the dictionary, since the pattern matches. The other new cases are self-explanatory: they are merely matching `(?c)`, `(?v)`, and `(?)`.

8.3.2 Instantiator

Recall that instantiate must take accept the input of a skeleton and a dictionary, traverse the skeleton tree, and replace rule variables for expression values according to the dictionary. That is, given the tree:



And the dictionary `(x: 3, y: 4)`, it should produce the tree:



This is a fairly simple procedure to write:

```
1  (define (instantiate skeleton dict)
2    (cond ((atom? skeleton) skeleton)
3          ((skeleton-evaluation? skeleton)
4           (evaluate (evaluation-expression skeleton)
5                     dict))
6          (else (cons (instantiate (car skeleton) dict)
7                      (instantiate (cdr skeleton) dict)))))
```

The general case is our usual tree recursion: it first instantiates the `car`, then `cons`' that with the instantiated `cdr`. The depth-first search ends if the leaf is atomic, as usual.

The interesting bit is what we do when we want to evaluate a skeleton of the `(:)` form (predicate `skeleton-evaluation?`). In this case, we call a procedure `evaluate`, to which we pass the expression to be evaluated (the `cadr`, really, since we don't need the `:`).

Now, `evaluate` works in a special way we'll see later, so take on faith that the following `evaluate` function does its job of instantiation the way we want it to:

```
1  (define (evaluation-expression evaluation) (cadr evaluation))
2
3  (define (evaluate form dict)
```

```
  (if (atom? form)
      (lookup form dict)
      (apply (eval (lookup (car form) dict)
                   user-initial-environment)
             (map (lambda (v) (lookup v dict))
                  (cdr form)))))
```

8.3.3 GIGO Simplifier

The GIGO (garbage in, garbage out)[2] simplifier is implemented by stringing together the matcher, the instantiator, and the list of rules we are given as an input. We write it in lexically scoped style, because the procedures within it are merely helpers.

```
(define (simplifier the-rules)
  (define (simplify-expression expression)
    (try-rules
     (if (compound? expression)
         (map simplify-expression expression)
         expression)))
  (define (try-rules expression)
    (define (scan rules)
      (if (null? rules)
          expression
          (let ((dict (match (pattern (car rules))
                             expression
                             (make-empty-dict))))
            (if (eq? dict 'failed)
                (scan (cdr rules))
                (simplify-expression (instantiate
                                      (skeleton (car rules))
                                      dict))))))
    (scan the-rules))
  simplify-expression)
```

Okay, there's a fair amount to break down here.

- `simplify-expression` tries *all* the rules on every node in the given expression tree. It does this using our (by now) standard depth first tree recursion. In this case, the leaf is an atomic expression. The general case is when the expression is compound, in which case we try to simplify the `car`. If this doesn't work, we try the `cdr`, recursively. This fulfils our objective of trying all rules on all nodes. Note that instead of doing the recursion explicitly, we use `map`. This doesn't make a difference, it is merely somewhat easier to write.

[2] "Garbage in, garbage out" means that the simplifier will produce garbage output if the rules supplied to it are garbage. That is, it makes no attempt to fix flaws in logic on the part of the user, much like a computer. This underlying principle of computing was noted by the father of computers, Charles Babbage: "On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."

- `try-rules` accepts an expression as an input, and scans the input list of rules, applying each in turn using `scan`.
- `scan` takes the pattern of the first rule in the list and tries to match it to the expression. If it succeeds, we instantiate the skeleton of this rule with the values from the dictionary. On the other hand, if the `match` failed, we simply try the rest of the rules (`cdr`).
- Finally, `simplifier` returns the `simplify-expression` procedure, which can apply `the-rules` to any input expression. Note that this is what the desired behaviour is (8.2).

8.3.4 Dictionary Implementation

The actual dictionary implementation isn't of much interest to us, since it has much more to do with Lisp primitives we'll study later than our program.

```
(define (make-empty-dict) '())

(define (extend-dict pat dat dictionary)
  (let ((vname (variable-name pat)))
    (let ((v (assq vname dictionary)))
      (cond ((not v)
             (cons (list vname dat) dictionary))
            ((eq? (cadr v) dat) dictionary)
            (else 'failed)))))

(define (lookup var dictionary)
  (let ((v (assq var dictionary)))
    (if (null? v)
        var
        (cadr v))))
```

8.3.5 Predicates

Finally, we must implement the predicates we've used throughout. These are simple to implement and self-explanatory:

```
(define (compound? exp) (pair? exp))
(define (constant? exp) (number? exp))
(define (variable? exp) (atom? exp))
(define (pattern rule) (car rule))
(define (skeleton rule) (cadr rule))

(define (arbitrary-constant? pat)
  (if (pair? pat) (eq? (car pat) '?c) false))

(define (arbitrary-expression? pat)
  (if (pair? pat) (eq? (car pat) '?) false))

(define (arbitrary-variable? pat)
```

```
  (if (pair? pat) (eq? (car pat) '?v) false))

(define (variable-name pat) (cadr pat))

(define (skeleton-evaluation? pat)
  (if (pair? pat) (eq? (car pat) ':) false))
```

8.4 Usage

```
(define dsimp
  (simplifier deriv-rules))

(dsimp '(dd (* x x) x))
```

```
(+ (* x 1) (* x 1))
```

Excellent — it works. Note that there is no algebraic simplification. Witness now the power of abstraction — all we really have to do is define some rules for algebraic simplification, and pass the result of `dsimp` to `algsimp` to get a clean expression.

8.4.1 Algebraic Simplification

Consider the rule set:

```
(define algebra-rules
  '(
    ( ((? op) (?c c1) (?c c2))               (: (op c1 c2))               )
    ( ((? op) (?  e ) (?c c ))               ((: op) (: c) (: e))         )
    ( (+ 0 (? e))                            (: e)                        )
    ( (* 1 (? e))                            (: e)                        )
    ( (* 0 (? e))                            0                            )
    ( (* (?c c1) (* (?c c2) (? e )))         (* (: (* c1 c2)) (: e))      )
    ( (* (?  e1) (* (?c c ) (? e2)))         (* (: c ) (* (: e1) (: e2)))  )
    ( (* (* (? e1) (? e2)) (? e3))           (* (: e1) (* (: e2) (: e3)))  )
    ( (+ (?c c1) (+ (?c c2) (? e )))         (+ (: (+ c1 c2)) (: e))      )
    ( (+ (?  e1) (+ (?c c ) (? e2)))         (+ (: c ) (+ (: e1) (: e2)))  )
    ( (+ (+ (? e1) (? e2)) (? e3))           (+ (: e1) (+ (: e2) (: e3)))  )
    ( (+ (* (?c c1) (? e)) (* (?c c2) (? e))) (* (: (+ c1 c2)) (: e))     )
    ( (* (? e1) (+ (? e2) (? e3)))           (+ (* (: e1) (: e2)))        )
    ))
```

```
(define dsimp
  (simplifier deriv-rules))

(define algsimp
  (simplifier algebra-rules))

(define (derivative x)
```

```
  (algsimp (dsimp x)))

(derivative '(dd (* x x) x))
(derivative '(dd (+ (+ x (* x 5)) (* x x)) x))
```

```
(+ x x)
(+ 6 (+ x x))
```

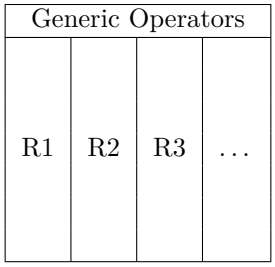
We now have a complete pattern matching and replacement language at our disposal, and we've tested it out on the rules of differentiation and algebraic simplification.

9 Lecture 4B: Generic Operators

We've seen that data abstraction separates the use of data objects from its representation. Unfortunately, this is not sufficient for complex systems.

Consider a situation where there are multiple representation designers and they cannot agree on a single uniform representation. This makes the use complicated, since the user has to use different operators for different representations of the same object, creating clutter.

One of the ways to solve such a problem is to enforce a standard for representations, but this is often impossible. Alternately, we could make a generic set of operators for the user that work on any kind of representation correctly and without complaint. Diagrammatically:



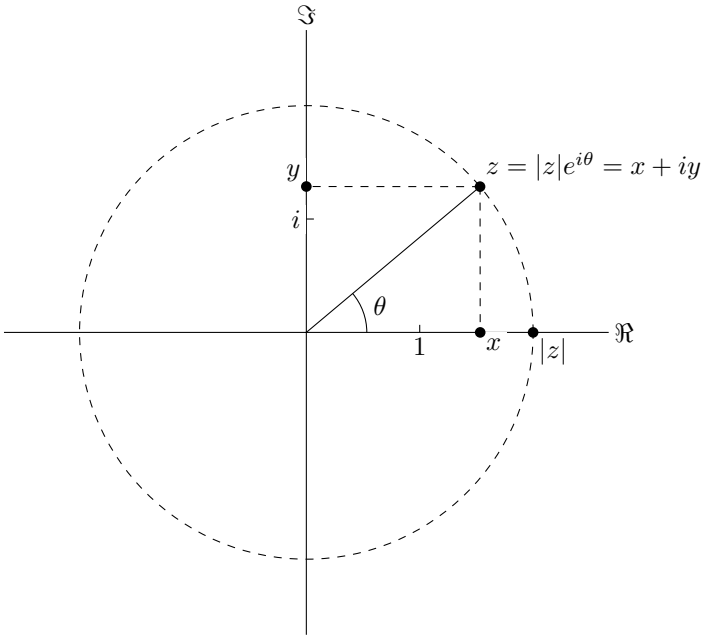
Moreover, note that it won't be too hard to add another representation R4, because all it needs to do is fit in the generic operators architecture.

Throughout the rest of this lecture, we discuss the application of various generic operator techniques on an extended example: first representing complex numbers in rectangular and polar form, and then expanding this to include the entire real number system.

9.1 Dispatch on Type

9.1.1 On Complex Numbers

Complex numbers are represented in the complex plane as:



It should be clear that for a complex number $z = |z|e^{i\theta} = x + iy$:

$$|z| = \sqrt{x^2 + y^2}$$

$$\theta = \arctan \frac{y}{x}$$

$$x = |z| \cos \theta$$

$$y = |z| \sin \theta$$

Let us say we want to add, subtract, multiply, and divide complex numbers. The most natural way to add is using rectangular form:

$$\Re(z_1 + z_2) = \Re(z_1) + \Re(z_2)$$
$$\Im(z_1 + z_2) = \Im(z_1) + \Im(z_2)$$

However, the most natural way to multiply is using polar form:

$$|z_1 z_2| = |z_1||z_2|$$
$$\theta(z_1 z_2) = \theta(z_1) + \theta(z_2)$$

9.1.2 Arithmetic Implementation

Let's implement the four basic arithmetic operations. As in the case of rational numbers, we assume the availability of the constructors and selectors:

```
(make-rect x y)
(make-pol magnitude theta)
(real z)
(img z)
(mag z)
(ang z)
```

Then,

```
(define (+c z1 z2)
  (make-rect
   (+ (real z1) (real z2))
   (+ (img z1) (img z2))))

(define (-c z1 z2)
  (make-rect
   (- (real z1) (real z2))
   (- (img z1) (img z2))))

(define (*c z1 z2)
  (make-pol
   (* (mag z1) (mag z2))
   (+ (ang z1) (ang z2))))

(define (/c z1 z2)
  (make-pol
   (/ (mag z1) (mag z2))
   (- (ang z1) (ang z2))))
```

9.1.3 George's Representation

George loves the rectangular form of complex numbers. He therefore implements the constructors and selectors in the following way:

```
(define (make-rect x y)
  (cons x y))

(define (real z)
  (car z))

(define (img z)
  (cdr z))

(define (make-pol r a)
```

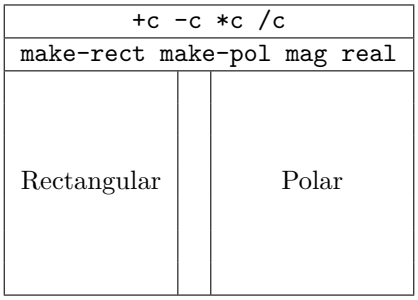
```
  (cons
   (* r (cos a))
   (* r (sin a))))

(define (mag z)
  (sqrt (+ (square (car z))
           (square (cdr z)))))

(define (ang z)
  (atan (cdr z) (car z)))
```

9.1.4 Martha's Representation

Martha, on the other hand, much prefers the polar representation. She implements the constructors and selectors as:

```
(define (make-pol r a)
  (cons r a))

(define (mag z)
  (car z))

(define (ang z)
  (cdr z))

(define (real z)
  (* (car z)
     (cos (cdr z))))

(define (img z)
  (* (car z)
     (sin (cdr z))))

(define (make-rect x y)
  (cons
   (sqrt (square x) (square y))
   (atan y x)))
```

Naturally, George's representation is better if we want to use `+c` a lot, while Martha's is better if we want to use `*c` a lot. Often, it's impossible to choose between two equally good representations. The solution is to use generic selectors and constructors. Diagrammatically:



9.1.5 Type Dispatch Manager

One of our chief problems is that when we have a pair that's a complex number, we have no way of knowing whether it's rectangular or polar. By attaching this pair to a "type label", we can easily tell by reading the label which type it is, and accordingly call George's or Martha's procedures. Consider:

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

(define (type datum)
  (car datum))

(define (contents datum)
  (cdr datum))
```

Now, George redefines his procedures as:

```
(define (make-rect x y)
  (attach-type 'rectangular (cons x y)))

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

(define (img-rectangular z)
  (cdr z))

(define (mag-rectangular z)
  (sqrt (+ (square (car z))
           (square (cdr z)))))

(define (ang-rectangular z)
  (atan (cdr z) (car z)))
```

Note that he changes his procedure names so that they don't conflict with the generic `make-rect`, `real`, and so on. The major change is that he rectangular complex numbers. Moreover, there's no need for George to have a `make-pol`, since that functionality is already provided by Martha, who implements her procedures as:

```
(define (make-pol r a)
  (attach-type 'polar (cons r a)))

(define (mag-polar z)
  (car z))

(define (ang-polar z)
  (cdr z))

(define (real-polar z)
  (* (car z)
     (cos (cdr z))))

(define (img-polar z)
  (* (car z)
     (sin (cdr z))))
```

Now, we can implement the generic procedures as:

```
(define (real z)
  (cond ((rectangular? z)
         (real-rectangular (contents z)))
        ((polar? z)
         (real-polar (contents z)))))

(define (img z)
  (cond ((rectangular? z)
         (img-rectangular (contents z)))
        ((polar? z)
         (img-polar (contents z)))))

(define (mag z)
  (cond ((rectangular? z)
         (mag-rectangular (contents z)))
        ((polar? z)
         (mag-polar (contents z)))))

(define (ang z)
  (cond ((rectangular? z)
         (ang-rectangular (contents z)))
        ((polar? z)
         (ang-polar (contents z)))))
```

We call procedures written this way "managers", since they decide whom to dispatch the data object to for processing. The predicates are simple label-checkers:

```
(define (rectangular? z)
  (eq? 'rectangular (type z)))
```

```
3
4  (define (polar? z)
5    (eq? 'polar (type z)))
```

9.1.6 Usage

```
1  (define a (make-rect 2 3))
2  (define b (make-pol 5 (tan (/ 3 4))))
3
4  (define result (+c a b))
5  (real result)
6  (img result)
7
8  (car a)
9  (car b)
10 (car result)
11 (cadr result)
12 (cddr result)
```

```
4.98276733430013
7.012866684732013
rectangular
polar
rectangular
4.98276733430013
7.012866684732013
```

Note that complex numbers are now a pair of a label and a pair.

9.1.7 Review

There are two major issues with this way of engineering the complex number system ("dispatch on type"):

- George and Martha having to change their procedure names to prevent namespace violation. This issue, however, can be fixed by methods explored in later lectures.
- A larger issue is that when an additional representation is added (say, Harry's), the manager's `cond` has to be changed. This isn't ideal, since it's a set of centralized procedures which all of George, Martha, and Harry have to be able to edit. This centralization and causes a bottleneck, and is annoying, especially because all the manager does, really, is a conditional dispatch — it's a bureaucrat.

  Our next efforts will be directed at eliminating the manager procedures.

9.2 Data-Directed Programming

Abstractly, what does the manager do? It acts as a lookup table, of shape and form:

Generic	Polar	Rectangular
`real`	`real-polar`	`real-rectangular`
`img`	`img-polar`	`img-rectangular`
`mag`	`mag-polar`	`mag-rectangular`
`ang`	`ang-polar`	`ang-rectangular`

We could decentralize the manager by simply constructing a globally accessible table like this, and letting George and Martha put their procedures into the correct place in the table. They must each set up their own columns, then they can continue to define their own "versions" of all the generic operators. They do this using two procedures:

```
(put KEY1 KEY2 value)
(get KEY1 KEY2)
```

`get` returns the value we'd `put` in the table. The precise way this table is represented is not of interest to us, and we defer its implementation to a later lecture.

9.2.1 Putting George's Procedures

```
(put 'rectangular 'real real-rectangular)
(put 'rectangular 'img img-rectangular)
(put 'rectangular 'mag mag-rectangular)
(put 'rectangular 'ang ang-rectangular)
```

Note carefully that the value inserted into the table cell is actually a *lambda*, it's not a symbol. This means that the table literally contains within it the procedures, not their names. Technically, George could have given the third argument of `put` as a λ directly instead of first defining it. This is desirable, since George can safely debug and change his procedures, and only "commit" them when he puts them in the table. Moreover, it greatly simplifies lookups: to apply a procedure, one must only grab the correct cell of the table.

9.2.2 Putting Martha's Procedures

Similarly:

```
(put 'polar 'real real-polar)
(put 'polar 'img img-polar)
(put 'polar 'mag mag-polar)
(put 'polar 'ang ang-polar)
```

Now, if Harry wants to add his procedures, he doesn't have to do anything except `put` them in the table.

9.2.3 Manager Automation

The procedures hanging around in the table isn't enough: when `+c` calls (say) `real`, it needs to `get` these values from the table. This is done by:

```
(define (operate op object)
  (let ((proc (get (type object) op)))
```

```
    (if (not (null? proc))
        (proc (contents object))
        (error "Operation undefined.")))))
```

It is now simple to define the generic operators:

```
(define (real obj)
  (operate 'real obj))

(define (img obj)
  (operate 'img obj))

(define (mag obj)
  (operate 'mag obj))

(define (ang obj)
  (operate 'ang obj))
```

9.2.4 Usage

The usage is the same as last time, since the difference is in the implementation:

```
(define a (make-rect 2 3))
(define b (make-pol 5 (tan (/ 3 4))))

(define result (+c a b))
(real result)
(img result)

(car a)
(car b)
(car result)
(cadr result)
(cddr result)
```

```
4.98276733430013
7.012866684732013
rectangular
polar
rectangular
4.98276733430013
7.012866684732013
```

> Note that an alternative to keeping the operations in a lookup table is to include the operations with the data object. This allows the data object to carry independently all the information required to work with it. This style of programming is called “message passing”.

We can see by the substitution rule how this system is working:

```
(real z)
(operate 'real z)
((get 'polar 'real) (contents z))
(real-polar (contents z))
```

9.3 Advanced Example: Semi-Complete Arithmetic System

9.3.1 Architecture

Let us now embed our complex number system "package" in a more complex arithmetic system which, diagrammatically, looks like the following:

<table>
<tr><td colspan="4">add sub mult div</td></tr>
<tr><td colspan="2">+complex -complex *complex /complex</td><td rowspan="3">Rational
+rat *rat</td><td rowspan="3">Lisp Numbers
+ - * /</td></tr>
<tr><td colspan="2">Complex
+c -c *c /c</td></tr>
<tr><td>Rectangular</td><td>Polar</td></tr>
</table>

Our goal is to embed our complex number, rational number, and standard Lisp number code into a single system. This will also be done with tags and a lookup table, with data being tagged as `complex`, `rational`, or `number`. The top-level operations `add`, `sub`, `mul`, and `div` are defined in terms of an `operate` which is binary rather than unary. Later, we expand this system by adding a way to add polynomials.

9.3.2 Implementation

The rational number procedures in this system are implemented thus:

```
(define (make-rat n d)
  (attach-type 'rational (cons n d)))

(put 'rational 'add +rat)
(put 'rational 'mul *rat)
```

The definitions of `+rat` and `*rat` are the same as last time:

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

(define (*rat x y)
  (make-rat
```

```
    (* (numer x) (numer y))
    (* (denom x) (denom y))))
```

Similarly, we can implement constructors and `put` procedures for complex numbers and standard Lisp numbers in this system:

```
(define (make-complex z)
  (attach-type 'complex z))

(define (+complex z1 z2)
  (make-complex (+c z1 z2)))

(define (-complex z1 z2)
  (make-complex (-c z1 z2)))

(define (*complex z1 z2)
  (make-complex (*c z1 z2)))

(define (/complex z1 z2)
  (make-complex (/c z1 z2)))

(put 'complex 'add +complex)
(put 'complex 'sub -complex)
(put 'complex 'mul *complex)
(put 'complex 'div /complex)
```

Note that we define additional procedures `+complex`, `-complex`, and so on to add the `complex` tag to results of operating on complex numbers. Moving to standard numbers:

```
(define (make-number n)
  (attach-type 'number n))

(define (+number x y)
  (make-number (+ x y)))

(define (-number x y)
  (make-number (- x y)))

(define (*number x y)
  (make-number (* x y)))

(define (/number x y)
  (make-number (/ x y)))

(put 'number 'add +number)
(put 'number 'sub -number)
(put 'number 'mul *number)
(put 'number 'div /number)
```

Finally, we must implement `add`, `sub`, `mul`, and `div` themselves:

```
(define (add x y)
  (operate-2 'add x y))

(define (sub x y)
  (operate-2 'sub x y))

(define (mul x y)
  (operate-2 'mul x y))

(define (div x y)
  (operate-2 'div x y))
```

Finally, we implement the binary `operate`, `operate-2`:

```
(define (operate-2 op arg1 arg2)
  (if (eq? (type arg1) (type arg2))
      (let ((proc (get (type arg1) op)))
        (if (not (null? proc))
            (proc (contents arg1)
                  (contents arg2))
            (error "Operation undefined.")))
      (error "Argument type mismatch.")))
```

9.3.3 Usage

We're now ready to test this system:

```
(define p (make-complex (make-pol 1 2)))
(define q (make-complex (make-pol 3 4)))
(mul q p)

(define r (make-rat 2 4))
(define s (make-rat 1 4))
(add r s)

(sub (make-number 65) (make-number 3))
```

```
(complex polar 3 . 6)
(rational 12 . 16)
(number . 62)
```

See from the output the structure: labels followed by actual value of the datum.

9.3.4 Adding Polynomials

To do arithmetic on polynomials, we must first figure out how to represent them. Consider the polynomial:

$$x^{15} + 2x^7 + 5$$

One simple way to represent a polynomial is a list of coefficients, with the rightmost coefficient being the constant term, the second last being the linear term, followed by quadratic, and so on. However, for "sparse" polynomials like the one in the example, this will result in a list with a lot of zeros. We therefore choose a slightly different representation: a list of pairs. The `car` of the pair is the order, and the `cdr` is the coefficient. Thus, the polynomial in the example is represented as `((15 1) (7 2) (0 5))`. We call this the "term list" of the polynomial. In our final representation, we'll `cons` this with the last bit of missing information: the variable the polynomial is in.

Thus, to construct a polynomial:

```
(define (make-poly var term-list)
  (attach-type 'polynomial (cons var term-list)))

(define (var poly) (car poly))
(define (term-list poly) (cdr poly))
```

Now, implementing `+poly`:

```
(define (+poly p1 p2)
  (if (eq? (var p1) (var p2))
      (make-poly
       (var p1)
       (+term (term-list p1)
              (term-list p2)))
      (error "Polynomials not in same variable.")))

(put 'polynomial 'add +poly)
```

Of course, this just pushed the real work onto `+term`, which is defined as:

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

```
                    (make-term (order t1)
                               (add (coeff t1)
                                    (coeff t2)))
                    (+term (rest-terms L1) (rest-terms L2)))))))))
```

Clearly, we need to implement some predicates, constructors, and selectors for `term`'s and `term-list`'s:

```
(define (empty-termlist? tl)
  (null? tl))

(define (first-term tl)
  (car tl))

(define (rest-terms tl)
  (cdr tl))

(define (adjoin-term term tl)
  (cons term tl))

(define (make-term o c)
  (cons o c))

(define (order t)
  (car t))

(define (coeff t)
  (cdr t))
```

Usage is obvious:

```
(define p1-tl
  (list
   (cons 15 (make-number 1))
   (cons 7 (make-number 2))
   (cons 0 (make-number 5))))
(define poly1 (make-poly 'x p1-tl))

(define p2-tl
  (list
   (cons 25 (make-number 1))
   (cons 15 (make-number 8))
   (cons 9 (make-number 4))
   (cons 7 (make-number 4))
   (cons 0 (make-number 15))))
(define poly2 (make-poly 'x p2-tl))

(add poly1 poly2)
```

```
(polynomial x (25 number . 1) (15 number . 9)
```

```
(9 number . 4) (7 number . 6) (0 number . 20))
```

Although somewhat hard to read, this is just the polynomial:

$$x^{25} + 9x^{15} + 4x^9 + 6x^7 + 20$$

which is the correct result of summing the polynomials:

$$(x^{15} + 2x^7 + 5) + (x^{25} + 8x^{15} + 4x^9 + 4x^7 + 15)$$

9.3.5 Recursive Data Directed Programming

It's easy to miss one *very* crucial point in the implementation of procedure `+term`: that when we actually add the coefficients of two same-order terms, we use the generic operator `add`. Or, instead of doing this:

```
(+ (coeff t1)
   (coeff t2))
```

we do:

```
(add (coeff t1)
     (coeff t2))
```

What this does is that it lets our polynomial have any coefficients supported by the overall system. We therefore, "for free", have rational and complex coefficients. Therefore, expressions like the following are fully supported:

$$\frac{3}{2}x^2 + \frac{17}{7}$$
$$(3 + 2i)x^5 + (4 + 7i)$$

This works by calling `add` on the coefficients, which knows how to add fractions and complex numbers.

```
(define p1-tl
  (list
   (cons 15 (make-rat 1 2))
   (cons 7 (make-rat 2 17))
   (cons 0 (make-rat 5 4))))
(define poly1 (make-poly 'x p1-tl))

(define p2-tl
  (list
   (cons 25 (make-rat 1 3))
   (cons 15 (make-rat 8 7))
   (cons 9 (make-rat 4 13))
   (cons 7 (make-rat 14 7))
```

```
    (cons 0 (make-rat 15 1))))
(define poly2 (make-poly 'x p2-tl))

(add poly1 poly2)
```

```
(polynomial x (25 rational 1 . 3) (15 rational 23 . 14)
              (9 rational 4 . 13) (7 rational 252 . 119) (0 rational 65 . 4))
```

Even cooler is the fact that we have support for polynomials with polynomial coefficients, such as:

$$(x^2 + 1)y^2 + (x^3 - 2x)y + (x^4 - 7)$$

This is because `add` also knows, recursively, how to add polynomials. We can, theoretically, nest this as far as we want to, having polynomials whose coefficients are polynomials whose coefficients are polynomials whose…

Note that we could rewrite `+rat` to use `add` instead of `+`, and we'd be able to support expressions such as

$$\frac{3x + 7}{x^2 + 1}$$

This exposes a powerful technique: once we have the basic implementation of any operation, say matrix addition, in the form:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} p & q \\ r & s \end{bmatrix} = \begin{bmatrix} a+p & b+q \\ c+r & d+s \end{bmatrix}$$

If we use generic operators, it doesn't matter what a, b, and the other operands are: they can be any type the generic operators support, including matrices.

9.4 Review

We started our implementation of generic operators by writing a manager to dispatch the correct procedure according to type using conditional statements. We saw that this created a bottleneck at the manager. We then eliminated the manager by using a lookup table, which the generic operators accessed and got the correct λ, effectively decentralising the manager's role, since any new representation can be added to the table without any interaction with the others. Finally, we saw how to use generic operators to allow the recursive use of operations on types within the system.

One feature, however, missing from the system, is the idea of "type coercion". Our system fails when it tries to add, say,

$$3 + \frac{2}{5}$$

by reporting a type mismatch (it cannot add a `number` and a `rational`). However, it won't complain if we ask it to add

$$\frac{3}{1} + \frac{2}{5}$$

Despite the meanings being mathematically the same. We then need to implement a way to "coerce" types into other types when it is mathematically meaningful to do so, for instance, interpreting a pure real number r as complex number $r + 0i$. The implementation of "type coercion" is shown in the SICP book.

10 Lecture 5A: Assignment, State, and Side-Effects

Up until now, we've used no assignment statements in our use of Lisp. If we add one, there must be a good reason to do so, and the creation of significant advantage. We will first consider how we'd go about dealing with assignment in Lisp, then the benefits it brings to our programming practice.

10.1 Functional v. Imperative Style

10.1.1 Functional Programming

Functional programs encode mathematical truth in a procedure. For instance,

```
(define (fact n)
  (cond ((= n 1) 1)
        (else (* n (fact (dec n))))))

(fact 4)
```

```
24
```

This is almost precisely the mathematical definition of a factorial in Lisp:

$$f(n) = \begin{cases} 1 & n = 1 \\ n \times f(n-1) & n > 1 \end{cases}$$

Moreover, such a function can be fully understood by substituting recursively:

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

Furthermore, we can differentiate between procedures with different processes. Consider our Peano adders:

```
(define (pa+ x y)
  (if (= x 0)
      y
      (pa+ (dec x) (inc y))))
```

and

```
(define (pb+ x y)
  (if (= x 0)
```

```
      y
      (inc (pb+ (dec x) y))))
```

10.1.2 Imperative Programming

Imperative programming involves writing statements which change the overall *state* of the program. Lisp does this using assignment statements called `set!`. `set!` is called by giving it two arguments. The second is a value, and the first is the variable to assign it to. Or:

```
(set! <var> <value>)
```

Consider what this assignment does. It creates an instant in time. Before this instant, `<var>` had no value, or some other value. *After* this `set!`, `<var>` has value `<value>`, which means that the `set!` changed something in the program state. Before the `set!` the state was (say) A, and after, it was a different state, (say) B. This is entirely different from any of our previous programs, all of which used functional programming style.

It should be obvious that procedures which use assignment are not bijective, that is, for the same input, they may have different outputs. Consider:

```
(define count 1)

(define (demo x)
  (set! count (inc count))
  (+ x count))

(demo 3)
(demo 3)
(demo 3)
```

```
5
6
7
```

`demo` does not compute any mathematical function. Instead, the successive values of `count` are `set!` and remembered somewhere in the environment each time `demo` is called.

Clearly, assignment introduces difficulties. Even our previous substitution model is now insufficient, since it is a static phenomenon which cannot account for changing values in the environment. Symbols now can refer to some place where the values of variables are stored. While this certainly makes our job harder, assignment is worth the trouble, as we'll see later.

10.1.3 Direct Comparison

Here's an iterative implementation of a functional factorial procedure:

```
(define (fact n)
  (define (iter m i)
    (cond ((> i n) m)
          (else
           (iter (* i m) (inc i)))))
```

```
  (iter 1 1))

(fact 4)
```

```
24
```

And an iterative implementation of an imperative-style factorial procedure:

```
(define (fact n)
  (let ((i 1) (m 1))
    (define (loop)
      (cond ((> i n) m)
            (else
             (set! m (* i m))
             (set! i (inc i))
             (loop))))
    (loop)))

(fact 4)
```

```
24
```

Note the difference between the two. The first passes successive values of `i` and `m` as parameters into the next call of `iter`. However, in the imperative style, the very values of `i` and `m` are changed to their new values, and the procedure simply called again without any arguments.

Note also that swapping the `set!` statements:

```
(set! i (inc i))
(set! m (* i m))
```

Will mess up the execution of the program, since `m` depends on `i` having the correct value *at the time* it's reassigned to `(* i m)`. The time instance creation inherent in assignment creates these time-based dependencies, which are worth being aware of.

10.2 Environment Model

As we observed, assignment invalidates our substitution model of evaluating Lisp programs. We therefore work on the creation of a new model which supports assignment.

10.2.1 Bound and Free Variables

> We say that a variable `v` is "bound in an expression" `E` if the meaning of `E` is unchanged by the uniform replacement of `v` with a variable `w` (not occurring in `E`) for every occurrence of `v` in `E`.

Bound variables exist naturally in mathematics. Consider the expressions:

$$\forall x \exists y P(x, y)$$

$$\int_0^1 \frac{\mathrm{d}x}{1 + x^2}$$

have no difference in meaning when x is substituted:

$$\forall w \exists y P(w, y)$$

$$\int_0^1 \frac{\mathrm{d}w}{1 + w^2}$$

This is because, in both cases, x is a bound variable. Now,

> A quantifier is a symbol which binds a variable.

Thus, in these cases, $\forall$, $\exists$, and $\int$ were the quantifiers of bound variable x. Similarly, consider a Lisp expression:

```
(lambda (y) ((lambda (x) (* x y)) 3))
```

Here, `x` and `y` are bound variables, and the λ are quantifiers for them. This can be illustrated by swapping out the variables and having no change in meaning:

```
(lambda (v) ((lambda (w) (* w v)) 3))
```

However, it is not necessary that all variables be bound. Consider the Lisp expression:

```
(lambda (x) (* x y))
```

`x` is bound, with quantifier λ. However, `y` is not bound (it is a free variable). This is because the definition of `y` does not come from the formal parameter list of a λ-expression or any other known location within the expression. Instead, it comes from somewhere outside. We cannot replace `y` with another variable `v`, since we don't know what value `y` may have. Formally,

> We say that a variable `v` is "free in an expression" `E` if the meaning of `E` is changed by the uniform replacement of `v` with a variable `w` (not occurring in `E`) for every occurrence of `v` in `E`.

In fact, in the expression:

```
(lambda (y) ((lambda (x) (* x y)) 3))
```

The symbol `*` is a free variable, since the expression's meaning will change if we replace it with the symbol `+`.

10.2.2 Scope

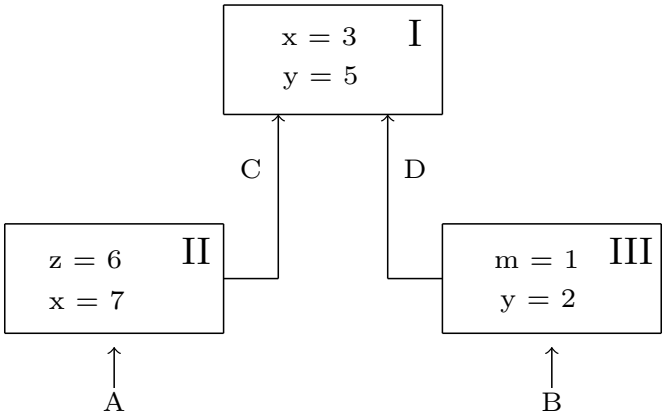
Formally,

> If `x` is a bound variable in `E`, then there is a λ-expression where it is bound. We call the list of formal parameters of this λ-expression the "bound variable list" and we say that the λ-expression "binds" the variable defined in the bound variable list. Moreover, those parts of the expression where a variable has a value defined by the λ-expression which binds it is called the "scope" of the variable.

Or, for example:

$$(\lambda\ (\texttt{y})\ \underbrace{((\lambda\ (\texttt{x})\ \overbrace{(*\ \texttt{x}\ \texttt{y})}^{\text{Scope of } \texttt{x}}\ )\ 3)}_{\text{Scope of } \texttt{y}})$$

10.2.3 Frames and Environments



An environment is made up of a linked chain of frames. Shown in the figure are three frames, I, II, and III. Now, each of A, B, C, and D are different environments, which view the frames in different ways.

Environment	x	y	z	m
A	7	5	6	-
B	3	2	-	1
C	3	5	-	-
D	3	5	-	-

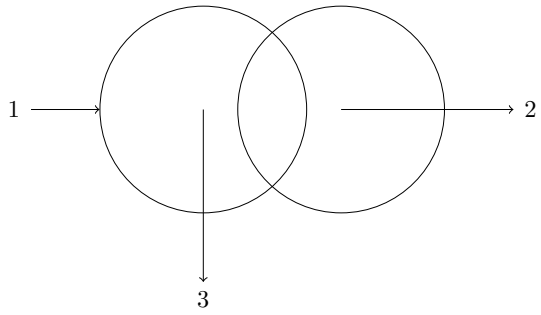
Environments can be considered as different "vantage points" of the frame chain. From A, we see the values of the frame II and the frames behind II, in this case I. Therefore, we get the values of x, y, and z. However, there are two possible values of x. In this case, we say that the value of x in frame II "shadows" the value of x in all previous frames, and we consider that value to be the correct one. Note that this is actually making the choice to use the innermost scope's variables over ones in outer scopes. Other environments are similarly analyzed. Note that C and D are the same environment.

10.2.4 Procedure Objects

A procedure objects consists of:

1. A pointer to the procedure object.
2. A pointer to the environment the procedure will execute in.
3. The actual body (code) of the procedure.

   Diagrammatically:



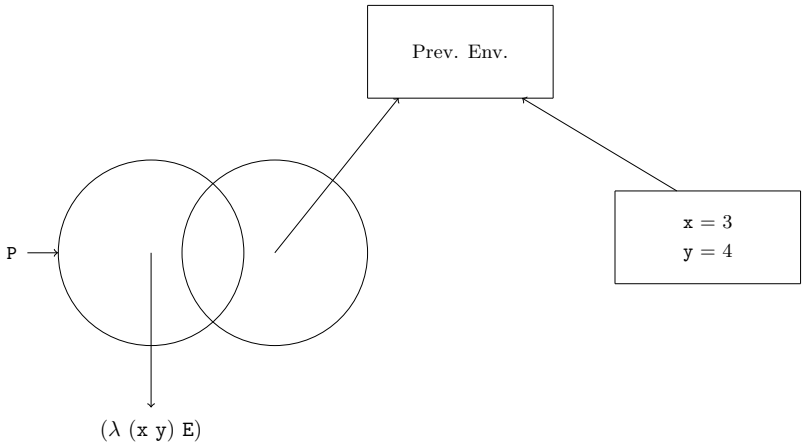
10.2.5 Rules for Environment Model

1. A procedure object is applied to a set of arguments by constructing a frame binding the formal parameters of the procedure to the actual arguments of the call, then evaluating the body of the procedure *in the context* of the new environment thus constructed. The new frame has as its enclosing environment the environment part of the procedure being applied.

   For example, for a procedure `P`, evaluated at `(P 3 4)`, the actual arguments to `P` are "appended" to the environment `P` is called in. The procedure object itself has three pointers:

   - The first is to the procedure itself, `P`.
   - The second points to the extended environment.
   - The third point points to the actual λ.

   Diagrammatically:



2. A λ-expression is evaluated relative to a given environment as follows: a new procedure object is formed, the combining the code of the λ-expression with a pointer to the environment of evaluation.

   This rule is self-explanatory.

   Our most important take-away is that an environment is a linked chain of frames, going all

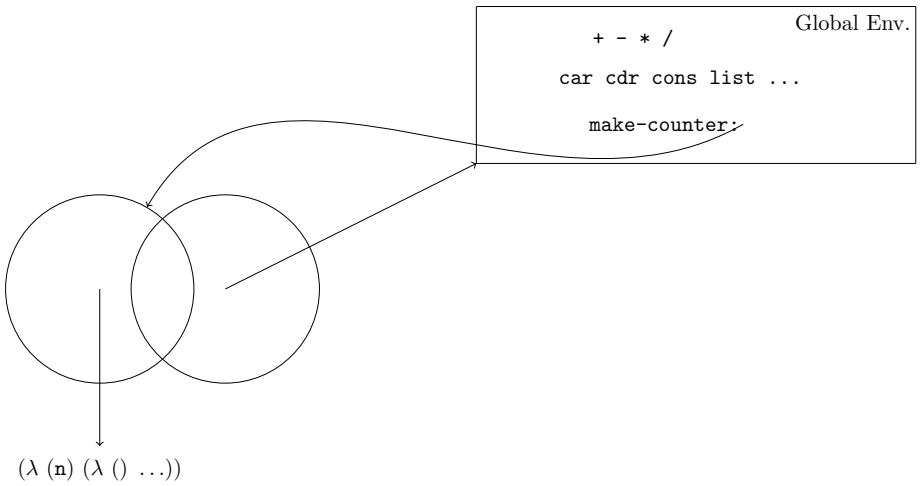
the way back to the first (global) frame. We now have a new working model, which works with assignment, as we'll soon see.

10.2.6 Assignment

Let's play around with assignment in the environment model. Consider the following procedure:

```
(define make-counter
  (lambda (n)
    (lambda ()
      (set! n (inc n))
      n)))
```

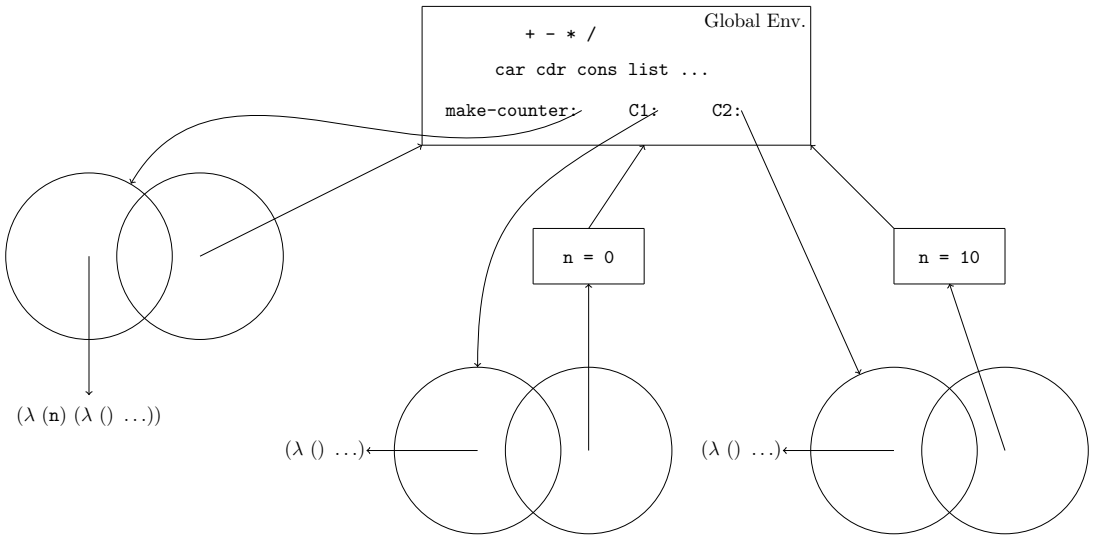
What does the environment look like once we define `make-counter`? Something like this:



Let us now use `make-counter` to define two counters, each of which start at different values of `n`:

```
(define C1 (make-counter 0))
(define C2 (make-counter 10))
```

The environment now looks like:



See that there is no *possible* confusion between which `n` `C1` uses and which `n` `C2` uses. Moreover, neither `C1` nor `C2` would care about a globally defined `n`, or any `n` in a higher level frame, since it'd get shadowed by their own `n`'s.

We can now see how the following would work:

```
1 (C1)
2 (C2)
3 (C1)
4 (C2)
```

```
1
11
2
12
```

The first call to `(C1)` sees that `n` is zero in its environment, and sets `n` to `0+1=1`, and returns `1`. The first call to `(C2)` sees that *its* `n` is `10`, sets it to `11`, and returns `11`. The second call to `(C1)` sees that its `n` is `1`, then sets and returns `2`. The final call to `(C2)` sees that its `n` is `11`, which is then sets to `12` and returned.

10.2.7 Some Philosophy

We have some questions to answer about sameness and change of objects. Consider that in the previous example, `C1`'s `n` and `C2`'s `n` are both called `n`, and yet refer to different objects. However, `C1` refers to only one object, `C1`. How then, can we tell if an identifier refers to an object? This issue is almost philosophical: if the only way know of an object is its identifier, what is an object anyway?

We avoid these issues in Lisp's environment model by defining action, change, and identity as follows:

Action A has an effect on object X (A changes X) if property P true of X before A is false of X after A.

and

X and Y are the same object if any action which changes X changes Y.

The (Lisp) world is thus made up of many individual objects, all with some local state.

10.3 Advantages of Assignment

We've seen that introducing assignment makes our pure functional programming somewhat "murky". One of the reasons to do this is that assignment often greatly enhances modularity, as we will show in an example. A word of caution, however: do not use assignment unless it is necessary.

10.3.1 Cesàro's Pi Finder

Cesàro's theorem tells us that:

$$P(\gcd(a,b)=1)=\frac{6}{\pi^2}$$

Of course, we can estimate probabilities using the Monte Carlo method (do lots of tests, and divide the number of successful tests by the total number of tests conducted). Using assignment, the program looks like:

```
(define (estimate-pi n)
  (sqrt (/ 6 (monte-carlo n cesaro))))

(define (cesaro)
  (= (gcd (random 4294967087) (random 4294967087)) 1))

(define (monte-carlo trials experiment)
  (define (iter remaining passed)
    (cond ((= remaining 0)
           (/ passed trials))
          ((experiment)
           (iter (dec remaining)
                 (inc passed)))
          (else
           (iter (dec remaining)
                 passed))))
  (iter trials 0))

(estimate-pi 10000000)
```

```
3.141330429791511
```

Note that we use Racket's built in `random` to generate a random number. However, if we had to implement a PRNG on our own, it'd look something like:

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

Assignment is used most naturally here because we want to give `x` the value of computing some function with input `x` to use the next time `rand` is called. This function is evaluated within a local scope where `x` at first has some constant seed value `random-init`.

10.3.2 Functional Cesàro's Pi Finder

Since we can't use assignment to keep track of the "state" of the PRNG, we must write our program like:

```
(define (estimate-pi n)
  (sqrt (/ 6 (random-gcd-test n))))

(define (random-gcd-test trials)
  (define (iter remaining passed x)
    (let ((x1 (rand-update x))
          (x2 (rand-update x1)))
      (cond ((= remaining 0)
             (/ passed trials))
            ((= (gcd x1 x2) 1)
             (iter (dec remaining)
                   (inc passed)
                   x))
            (else
             (iter (dec remaining)
                   passed
                   x2)))))
  (iter trials 0 random-seed))
```

The state of the PRNG has "leaked" out into our program, that is, `iter` has to keep track of successive values of the seed. Worse still, this makes `monte-carlo` and `cesaro` non-general, reducing modularity. It is applications like these where assignment is incredibly useful, and helps keep our programs neat and modular.

11 Lecture 5B: Computational Objects

Now that we have local state, let's see what we can do with it. As we saw earlier, real world objects also have local state. As such, we can now make a model of the real world in our computer, taking advantage of its inherent modularity to built a model with modular objects.

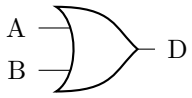
We explore this possibility in this lecture by writing a simulator for digital circuits in Lisp. Along the way, we'll have to implement certain data structures which support local state.

11.1 Digital Circuit Language

We build a language embedded in Lisp (in the same style as the picture language) to simulate digital circuits. Here's the expected usage:

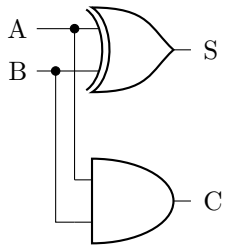
```
(define a (make-wire))
(define b (make-wire))
(define c (make-wire))
(define d (make-wire))
(define e (make-wire))
(define s (make-wire))

(or-gate a b d)
(and-gate a b c)
(inverter c e)
(and-gate d e s)
```

The `or-gate` takes two input wires `a` and `b`, and returns the `OR` of the signals on wire `d`, very much like a real OR gate:



We could use these primitives to write a half adder:

```
(define (half-adder a b s c)
  (let ((d (make-wire))
        (e (make-wire)))
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)
    'OK))
```



Note carefully that the procedure `half-adder` doesn't actually return anything: it merely makes calls to the gate procedures. We'll see why later. Since the language is hierarchical, we could now write a full adder:

```
(define (full-adder a b c-in sum c-out)
  (let ((s (make-wire))
        (c1 (make-wire))
        (c2 (make-wire)))
    (half-adder b c-in s c1)
    (half-adder a s sum c2)
    (or-gate c1 c2 c-out)))
```

11.2 Gates

Let us now implement the primitive gates of this language. Here's an `inverter`:

```
(define (inverter in out)
  (define (invert-in)
    (let ((new
           (logical-not (get-signal in))))
      (after-delay inverter-delay
                   (lambda ()
                     (set-signal! out new)))))
  (add-action! in invert-in))
```

`inverter` defines a procedure called `invert-in`, and adds that action to the wire `in`. `invert-in` itself takes the logical not of `in` and sets the signal on `out` to this value, after some delay. See that this is precisely how a digital inverter in the real world is expected to work: wires have certain signals (`get-signal` and `set-signal`), these signals change according to how the wires are routed `add-action!`, and there are certain delays before the signal changes.

Of course, the primitive `logical-not` is easy to define:

```
(define (logical-not s)
  (cond ((= s 0) 1)
        ((= s 1) 0)
        (else
         (error "Invalid signal" s))))
```

Similarly, `and-gate` can be defined as:

```
(define (and-gate a1 a2 out)
  (define (and-action-proc)
    (let ((new-value
           (logical-and (get-signal a1)
                        (get-signal a2))))
      (after-delay and-gate-delay
                   (lambda ()
                     (set-signal! out new-value)))))
  (add-action! a1 and-action-proc)
  (add-action! a2 and-action-proc))
```

The only difference is in that we need to `get-signal` two signals and `add-action!`'s to two signals. `logical-and` can be implemented as:

```
(define (logical-and a b)
  (cond ((= a 0)
         (cond ((= b 0) 0)
               ((= b 1) 0)))
        ((= a 1)
         (cond ((= b 0) 0)
               ((= b 1) 1)))))
```

Finally, an `or-gate` is implemented as:

```
(define (or-gate r1 r2 out)
  (define (or-action-proc)
    (let ((new-value
           (logical-or (get-signal r1)
                       (get-signal r2))))
      (after-delay or-gate-delay
                   (lambda ()
                     (set-signal! out new-value)))))
  (add-action! r1 or-action-proc)
  (add-action! r2 or-action-proc))
```

`logical-or` is defined similar to `logical-and`:

```
(define (logical-or a b)
  (cond ((= a 0)
         (cond ((= b 0) 0)
               ((= b 1) 1)))
        ((= a 1)
         (cond ((= b 0) 1)
               ((= b 1) 1)))))
```

11.3 Wires

We define wires as:

```
(define (make-wire)
  (let ((signal 0)
        (action-procs '()))
    (define (set-my-signal! new)
      (cond ((= signal new) 'DONE)
            (else
             (set! signal new)
             (call-each action-procs))))
    (define (accept-action-proc proc)
      (set! action-procs
            (cons proc action-procs))
      (proc))
    (define (dispatch m)
      (cond ((eq? m 'get-signal) signal)
            ((eq? m 'set-signal!) set-my-signal!)
            ((eq? m 'add-action!) accept-action-proc)
            (else
             (error "Bad message" m))))
    dispatch))
```

Clearly, a wire is defined by two local-scope variables, `signal` and `action-procs`, which are initially set to `0` and `'()` (the empty list, `nil`) respectively. Internally defined procedure `set-my-signal!` does the obvious thing and sets the signal to the new signal. It then calls all the action procedures of this wire. `accept-action-proc` adds a new procedure to the front of the existing action procedure list (`action-procs`) a new procedure. It then calls this procedure for reasons we'll see later. Finally, `dispatch` does the right thing according to its argument.

`call-each` is a simple `cdr` down a list and execution — note the extra parenthesis around `((car procs))`:

```
(define (call-each procs)
  (cond ((null? procs) 'DONE)
        (else
         ((car procs))
         (call-each (cdr procs)))))
```

Naturally, we can now define `get-signal`, `set-signal`, and `add-action` using the wire's `dispatch`:

```
(define (get-signal wire)
  (wire 'get-signal))

(define (set-signal! wire new-val)
  ((wire 'set-signal!) new-val))

(define (add-action! wire proc)
  ((wire 'add-action!) proc))
```

11.4 Delays and Propagation

We define the `after-delay` procedure as:

```
(define (after-delay delay action)
  (add-to-agenda!
   (+ delay (current-time the-agenda))
   action
   the-agenda))
```

`after-delay` calls a function called `add-to-agenda!`. This function takes three arguments to add to a data structure called an agenda: the time at which to add an action, the action, and the agenda object to add to.

To "run" the agenda, we define `propagate`:

```
(define (propagate)
  (cond ((empty-agenda? the-agenda) 'DONE)
        (else
         ((first-agenda-item the-agenda))
         (remove-first-item! the-agenda)
         (propagate))))
```

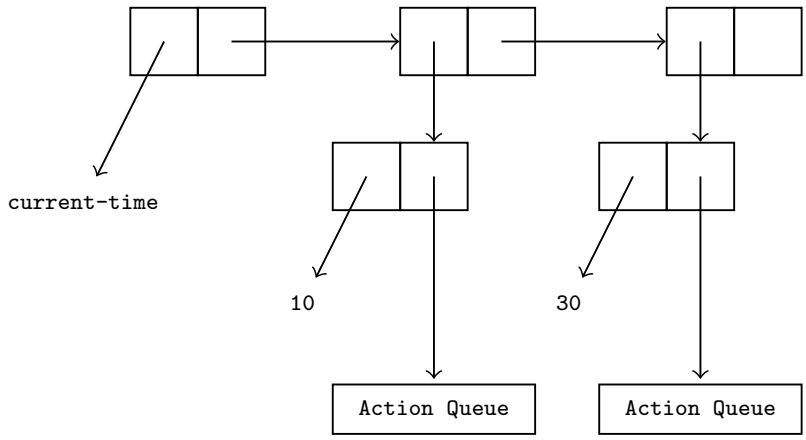
`propagate` simply executes the first item off the agenda, removes it, then executes the rest of the agenda until there's nothing else left in the agenda.

11.5 The Agenda

We've the use of the agenda. We need the following primitives:

```
(make-agenda)
(current-time agenda)
(empty-agenda? agenda)
(add-to-agenda! time action agenda)
(first-agenda-item agenda)
(remove-first-item! agenda)
```

Here's a data structure which can do the job:



In essence, the agenda is a sorted list. The `car` of the agenda is the current time. The `cdr` of this agenda is a list of pairs, each of which we'll call a "time segment". The `cdr` of each time segment is a queue of actions to execute, and the `car` is the time at which all these actions are scheduled.

11.5.1 Time Segments

These are trivial to implement, since they're just pairs:

```
(define (make-time-segment time q)
  (cons time q))

(define (segment-time s) (car s))

(define (segment-queue s) (cdr s))
```

11.5.2 Agenda Implementation

The constructors, selectors, and mutators are:

```
(define (make-agenda) (list 0))

(define (current-time agenda) (car agenda))

(define (set-current-time! agenda time)
  (set-car! agenda time))

(define (segments agenda) (cdr agenda))

(define (set-segments! agenda segments)
  (set-cdr! agenda segments))

(define (first-segment agenda)
  (car (segments agenda)))

```

```
(define (rest-segments agenda)
  (cdr (segments agenda)))
```

There's only one predicate:

```
(define (empty-agenda? agenda)
  (null? (segments agenda)))
```

`add-to-agenda` is defined as:

```
(define (add-to-agenda! time action agenda)
  (define (belongs-before? segments)
    (or (null? segments)
        (< time (segment-time (car segments)))))
  (define (make-new-time-segment time action)
    (let ((q (make-queue)))
      (insert-queue! q action)
      (make-time-segment time q)))
  (define (add-to-segments! segments)
    (if (= (segment-time (car segments)) time)
        (insert-queue! (segment-queue (car segments))
                       action)
        (let ((rest (cdr segments)))
          (if (belongs-before? rest)
              (set-cdr!
               segments
               (cons (make-new-time-segment time action)
                     (cdr segments)))
              (add-to-segments! rest)))))
  (let ((segments (segments agenda)))
    (if (belongs-before? segments)
        (set-segments!
         agenda
         (cons (make-new-time-segment time action)
               segments))
        (add-to-segments! segments))))
```

This procedure wants to add an action A at time t to the agenda. Several cases present themselves:

1. t is less than the earliest time in the agenda. In this case, a new time segment must be created, and the `cdr` of the agenda must point to it. The `cdr` of this new time segment should be the rest of the segments.
2. t is equal to one of the times in the agenda. In this case, A must be inserted into the action queue of the correct time segment.
3. t is any other value. In this case, a new time segment must be created and placed in the correct place. The `cdr` of the previous segment must be the new segment, and the `cdr` of the new segment must be the rest of the segments.

   This procedure does exactly this. We first define some helper functions: `belongs-before?`

simply compares the required time of A with the first segment of a list of input segments, or an empty list. `make-new-time-segment` simply creates a queue, inserts A into it, and creates a time segment ready for insertion. `add-to-segments!` is a recursively written procedure. The base cases are:

(a) If t is equal to the `car` of the segments, add it to the action queue of this segment.

(b) If t is less than the `car` of the segments, put the segment before it. Finally, the actual function checks if t is less than the earliest time in the agenda, in which case `set-segments!` is called. Otherwise, `add-to-segments!` is called to do the right thing. In essence, this procedure is an insertion sort.

We now define how to remove the first time of the agenda:

```
(define (remove-first-item! agenda)
  (let ((q (segment-queue (first-segment agenda))))
    (delete-queue! q)
    (if (empty-queue? q)
        (set-segments! agenda (rest-segments agenda)))))
```

`delete-queue!` (albeit poorly named) is a procedure which deletes the first element of a queue. If this causes the entire queue to be empty, we get rid of the time segment.

Finally, a procedure for getting the first agenda item and setting the time counter to its time is:

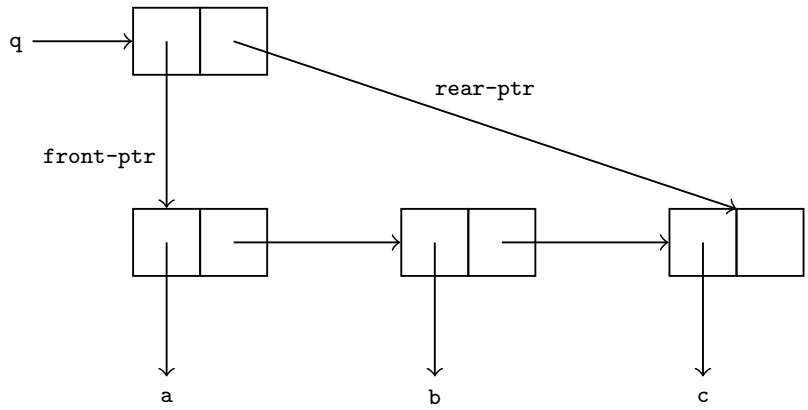
```
(define (first-agenda-item agenda)
  (if (empty-agenda? agenda)
      (error "Agenda is empty --- FIRST-AGENDA-ITEM")
      (let ((first-seg (first-segment agenda)))
        (set-current-time! agenda (segment-time first-seg))
        (front-queue (segment-queue first-seg)))))
```

11.6 Queues

Our only missing piece now is implementing the action queues time segments use. These are FIFO (first in, first out) buffers, and thus the following constructors, selectors, and mutators are required:

```
(make-queue)
(empty-queue? q)
(front-queue q)
(insert-queue! q item)
(delete-queue! q)
```

It is natural to represent a queue as a standard list, since it's just a bunch of items. However, this will cause the `insert-queue!` mutator to run in $O(n)$ time. This is because linked lists by default store only a pointer to the first element, and there's no way to get a pointer to the last element without traversing the list. We avoid this problem by implementing a structure which looks like this:



The `rear-ptr` always points to the last element of the queue, giving us an $O(1)$ way to insert elements to the end of the queue. Queues are thus a pair of "pointers".

We initialize an empty queue as a pair of empty lists. An empty queue is thus one with the `front-ptr` equal to `null`. Thus:

```
(define (make-queue) (cons '() '()))

(define (empty-queue? queue)
  (null? (front-ptr queue)))

(define (front-ptr queue) (car queue))

(define (rear-ptr queue) (cdr queue))

(define (set-front-ptr! queue item)
  (set-car! queue item))

(define (set-rear-ptr! queue item)
  (set-cdr! queue item))

(define (front-queue queue)
  (if (empty-queue? queue)
      (error "FRONT: empty queue" queue)
      (car (front-ptr queue))))
```

Now, the core implementation of a queue: the insertion of an item.

```
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
```

```
            (set-cdr! (rear-ptr queue) new-pair)
            (set-rear-ptr! queue new-pair)
            queue))))
```

First, create a new queue item object. Then, two cases present themselves:

1. If the queue was empty, set both the `front-ptr` and `rear-ptr` to the new item, and return the queue.
2. Otherwise, go to the last item of the queue. Set its `cdr` to the new item, and set the `rear-ptr` to the new item. Return the queue.

Finally, we must implement deletion of the first item.

```
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE-QUEUE!: called with empty queue" queue))
        (else
         (set-front-ptr! queue (cdr (front-ptr queue)))
         queue)))
```

Which simply makes `front-ptr` point to the second element of the queue, and returns the queue. If the queue is empty, it throws an error since you can't delete the first element of an empty queue.

11.7 Using the Digital Circuit Language

Before using our (now complete) circuit language, we define a procedure called `probe`, which shows the simulator in action. That is, it adds an action to a wire which prints some wire information whenever the signal changes. Consider:

```
(define (probe name wire)
  (add-action! wire
               (lambda ()
                 (display name)
                 (display ": TIME = ")
                 (display (current-time the-agenda))
                 (display ": NEW-VAL = ")
                 (display (get-signal wire))
                 (newline))))
```

```
(define the-agenda (make-agenda))
(define inverter-delay 2)
(define and-gate-delay 3)
(define or-gate-delay 5)

(define input-1 (make-wire))
(define input-2 (make-wire))
(define sum (make-wire))
(define carry (make-wire))

```

```
(probe 'SUM sum)
(probe 'CAR carry)

(half-adder input-1 input-2 sum carry)

(set-signal! input-1 1)
(propagate)

(set-signal! input-2 1)
(propagate)
```

```
SUM: TIME = 0: NEW-VAL = 0
CAR: TIME = 0: NEW-VAL = 0
OK
DONE
SUM: TIME = 8: NEW-VAL = 1
DONE
DONE
CAR: TIME = 11: NEW-VAL = 1
SUM: TIME = 16: NEW-VAL = 0
DONE
```

11.8 Church's `set-car!` and `set-cdr!`

Although `set-car!` and `set-cdr!` are primitives for efficiency reasons, it is theoretically possible to define them in terms of `set!`. That is, once we have one way to do assignment, it is unnecessary to define more primitives, similar to how `cons`, `car`, and `cdr` could be implemented by λ-expressions. Consider:

```
(define (cons x y)
  (lambda (m)
    (m x y)))

(define (car x)
  (x (lambda (a d) a)))

(define (cdr x)
  (x (lambda (a d) d)))

(define a (cons 1 2))
(car a)
(cdr a)
```

```
1
2
```

`cons` defines a pair as a λ which takes as an argument a procedure `m`, which it applies to arguments `x` and `y`. `car` takes as an input a pair (a λ which takes a λ argument), and gives it as an argument a

λ which takes two argument and returns the first. `cdr` does the same thing, except its λ argument returns the second argument. Since these are purely functional, we can apply the substitution rule to see what really goes on:

```
(define a (cons 1 2))
(define a (lambda (m) (m 1 2)))

(car a)
(car (lambda (m) (m 1 2)))
((lambda (a d) a) 1 2)
1
```

We can also define `cons`, `car`, and `cdr` such that they support `set-car!` and `set-cdr!`:

```
(define (cons x y)
  (lambda (m)
    (m x
       y
       (lambda (n) (set! x n))
       (lambda (n) (set! y n)))))

(define (car x)
  (x (lambda (a d sa sd) a)))

(define (cdr x)
  (x (lambda (a d sa sd) d)))

(define (set-car! x n)
  (x (lambda (a d sa sd) (sa n))))

(define (set-cdr! x n)
  (x (lambda (a d sa sd) (sd n))))

(define a (cons 1 2))
(car a) (cdr a)
(set-car! a 10)
(set-cdr! a 20)
(car a) (cdr a)
```

```
1
2
10
20
```

All we need to do is add two arguments to `m`: λ's which set `x` and `y` to the argument. `car` and `cdr` are nearly identical to their previous forms, with the only difference being the number of arguments the λ passed to the pair (`m`) takes. `set-car!` and `set-cdr!` need only call the `set!` λ's in `m` with the value they are to be set to.

Thus, once we have a single assignment procedure (`set!`), we can create any other forms of assignment functionally. Note, however, that this is not necessarily how `set-car!` and `set-cdr!` are *actually* implemented, since this way is less efficient.

12 Lecture 6A: Streams, Part I

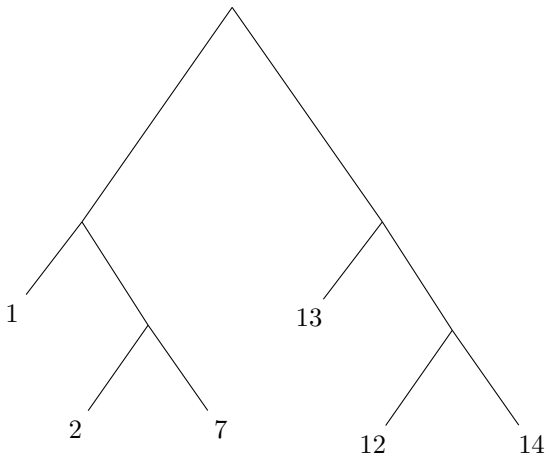
In the last lecture, we discussed assignment. We also saw that this one addition required a complete change of perspective from our simple substitution model to an environment based model, which was required since assignment introduces the ideas of state and of time. Pairs and variables are actually *objects* with identity instead of mere mathematical variables. Two pairs with the same `car` and `cdr` may still be different objects. In this sense, assignment causes the programming language to be philosophically harder, and can create state-related bugs. Why did we choose to incur this cost? To build modular systems, such as the psuedorandom number generator we built. Assignment often helps modularity, since state is a part of the real world, as in the case of digital circuits.

In this lecture, we will attempt to shift our perspective of the world in order to remove state, by looking at it from a "signal processing" style system: with signals coming in, being processed, and some output being generated.

12.1 Stream Processing

Consider the following two problems:

1. Given a tree that looks like:



   We want to sum all the squares of all the odd nodes. A conventional solution to this problem is as follows:

```
(define (sum-odd-squares tree)
  (if (leaf-node? tree)
      (if (odd? tree)
          (square tree)
          0)
      (+ (sum-odd-squares
          (left-branch tree))
         (sum-odd-squares
```

```
            (right-branch tree)))))

(define my-tree (make-tree (make-tree 1 (make-tree 2 7)) (make-tree 13
↪  (make-tree 12 14))))
(sum-odd-squares my-tree)
```

```
219
```

2. Given an input k, we want returned a list of all the odd Fibonacci numbers till the n^{th} Fibonacci number.

```
(define (odd-fibs n)
  (define (next k)
    (if (> k n)
        '()
        (let ((f (fib k)))
          (if (odd? f)
              (cons f (next (inc k)))
              (next (inc k))))))
  (next 1))

(odd-fibs 10)
```

```
(1 1 3 5 13 21 55)
```

Now, we can see that these procedures are actually doing the following:

1. Square odd nodes in a tree:
   (a) Enumerate the leaves.
   (b) Filter for `odd?`-ness.
   (c) Map `square`.
   (d) Accumulate using `+` starting at 0.
2. Find odd Fibonacci numbers till the n^{th} one:
   (a) Enumerate numbers from 1 to n.
   (b) Map `fib`.
   (c) Filter for `odd?`-ness.
   (d) Accumulate using `cons`, starting with the empty list `'()`.

   We've rewritten the procedures in terms of enumerating, mapping, filtering, and accumulating. Note that this is similar to how signals are processed by passing them through filters and the like. However, our Lisp procedures in their current state do not explicitly have these distinct parts.

   Let us therefore invent a language for writing procedures in this signal processing way. The objects passed between maps, filters, etc. are called "streams", which are defined by the following selectors and constructors:

- Constructor: `(cons-stream x y)`.
- Selectors:
  - `(head s)`
  - `(tail s)`

The behavior is such that for any `x` and `y`:

```
(define x 1)
(define y 2)
(head (cons-stream x y))
(tail (cons-stream x y))
```

```
1
2
```

We also have `the-empty-stream`. Note that this description makes a stream look a lot like a pair, and, for now, it's alright to pretend streams *are* pairs. We can now define our stream processing procedures:

```
(define (map-stream proc s)
  (if (empty-stream? s)
      the-empty-stream
      (cons-stream
       (proc (head s))
       (map-stream proc (tail s)))))

(define (filter-stream pred s)
  (cond
    ((empty-stream? s) the-empty-stream)
    ((pred (head s))
     (cons-stream (head s)
                  (filter-stream pred (tail s))))
    (else (filter-stream pred (tail s)))))

(define (accumulate-stream combiner init-val s)
  (if (empty-stream? s)
      init-val
      (combiner (head s)
                (accumulate-stream combiner init-val (tail s)))))

(define (enumerate-tree tree)
  (if (leaf-node? tree)
      (cons-stream tree the-empty-stream)
      (append-streams
       (enumerate-tree
        (left-branch tree))
       (enumerate-tree
```

```
        (right-branch tree)))))

(define (append-streams s1 s2)
  (if (empty-stream? s1)
      s2
      (cons-stream (head s1)
                   (append-streams (tail s1) s2))))

(define (enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low
                   (enumerate-interval (inc low) high))))

(define (singleton s)
  (cons-stream s the-empty-stream))
```

These should be relatively intuitive. We can now write our procedures using stream processing:

```
(define (sum-odd-squares tree)
  (accumulate-stream +
                     0
                     (map-stream square
                                 (filter-stream odd?
                                                (enumerate-tree tree)))))

(define (odd-fibs n)
  (accumulate-stream cons
                     '()
                     (filter-stream odd?
                                    (map-stream fib
                                                (enumerate-interval 1 n)))))

(define my-tree (make-tree (make-tree 1 (make-tree 2 7)) (make-tree 13
↪  (make-tree 12 14))))
(sum-odd-squares my-tree)

(odd-fibs 10)
```

```
219
(1 1 3 5 13 21 55)
```

This brings modularity into our programs: we're establishing conventional interfaces via streams. This way of looking at programs (mapping, filtering, etc.) is very general and can be used to write all sorts of procedures.

12.2 Some More Complex Examples

12.2.1 Flattening

Suppose we have a "stream of streams", or a data structure that looks like the following:

```
((1 2 3) (4 5 6) (7 8 9))
```

That we want to process into a single stream:

```
(1 2 3 4 5 6 7 8 9)
```

That is, we want to "flatten" the stream of streams. We can do so like this:

```
(define (flatten stream-of-streams)
  (accumulate-stream append-streams
                     the-empty-stream
                     stream-of-streams))
```

Now, consider a function f that returns a stream. We want to call f on a stream (map f to it), then take its output stream-of-streams and flatten it. This is also simple:

```
(define (flatmap f s)
  (flatten (map-stream f s)))
```

12.2.2 Prime Sum Pairs

Our first problem is: given N, find all pairs i, j $(0 < j < i \leq N)$ such that $i + j$ is prime. For instance, given $N = 6$, our expected output is something like:

i	j	$i+j$
2	1	3
3	2	5
4	1	5
⋮	⋮	⋮

We can therefore write:

```
(#%require math/number-theory) ;; for prime?
(define (prime-sum-pairs n)
  (map-stream (lambda (p)
                (list (car p) (cadr p) (+ (car p) (cadr p))))
              (filter-stream (lambda (p)
                               (prime? (+ (car p) (cadr p))))
                             (flatmap (lambda (i)
                                        (map-stream (lambda (j) (list i j))
                                                    (enumerate-interval 1 (dec
                                                    ↪  i))))
                                      (enumerate-interval 1 n)))))

(head (prime-sum-pairs 6))
```

```
(head (tail (prime-sum-pairs 6)))
(head (tail (tail (prime-sum-pairs 6))))
```

```
(2 1 3)
(3 2 5)
(4 1 5)
```

How does this work? Let's read it inside out, starting with the `flatmap`. The first argument to `flatmap` is a function which, given an input `i`, creates a stream of lists of `i` and `j` for all values of `j` between `1` and `i`. The second argument is simply all `i`'s from `1` to `n`. `flatmap` therefore creates a stream that looks like the following:

$$\left[\binom{2}{1}\ \binom{3}{1}\ \binom{3}{2}\ \binom{4}{1}\ \binom{4}{2}\ \binom{4}{3}\ \binom{5}{1}\ \cdots\right]$$

All that's left to do is filter this stream for the `prime?`-ness of $i + j$ in each list, then create our output in the expected format. This is done using `filter-stream` with the right λ, and then mapping this stream to our output format.

12.2.3 TODO Backtracking: The Eight Queens Problem

12.3 The Catch: On Efficiency

By treating streams the way we have, simply as lists, introduces a form of inefficiency. Consider the problem of finding the second prime in the interval $[10^4, 10^{15}]$. `enumerate-interval` would never terminate, since it'll spend forever generating an incredibly long list of numbers. Assuming this can be done, we'll spend forever and a day filtering and checking primality, only to ignore most of this work and pick up the second element of the list. This is clearly outrageous, since most of the work has gone to waste, assuming a computer could do it in the first place. It seems the "traditional" approach, while less mathematically pretty, is better, since it would stop as soon as it found the second prime.

Now, this would indeed be true if streams were lists. But — streams are streams, so we can indeed do:

```
(#%require math/number-theory)
(define (get-prime low high)
  (filter-stream prime?
                 (enumerate-interval low high)))
(head (tail (get-prime 10000 1000000000000000)))
```

```
10009
```

And get a result. It seems that we can have our cake and eat it too. What is this magic? How are we able to both have the expressiveness of stream-style processing *and* the efficiency of traditional programming? What is this magical black box called a stream?

12.4 Implementing Streams

Streams work on the principle of "laziness". That is, when we declare a stream, it figures out *only* what the first element is, and promises to do the rest of the work "later". If, and *only if* we need the second element is it computed, by us calling on the promise. An alternate way to visualize this is by imagining how you'd thread a string through a series of tubes: we only pull out of the far end as much string as we need.[1]

Streams are an on-demand data structure. These are easy to implement in our language using procedures as first-class citizens. Here's how `cons-stream`, `head`, and `tail` are defined:

```
(define-syntax cons-stream
  (syntax-rules ()
    ((cons-stream a e) (cons a (delay e)))))

(define (head s)
  (car s))

(define (tail s)
  (force (cdr s)))
```

`define-syntax` defines `cons-stream` to be a macro. All that this means is that any instances of `(cons-stream a e)` are replaced by `(cons a (delay e))` *before* we pass the program to the interpreter. Why we define it as a macro rather than a procedure will become clear momentarily. Note that by this token, streams are created as the `cons` of the first input and the second input passed to something called `delay`. `head` is the same as `car`, and `tail` calls `force` on the `cdr` of the stream.

Clearly, there's some magic going on in `delay` and `force` which makes streams lazy: but there's really none. Here's how they're defined:

```
(define-syntax delay
  (syntax-rules ()
    ((delay expr)
     (lambda () expr))))

(define (force p)
  (p))
```

Again `delay` is simply a macro that replaces any instance of `(delay expr)` with `(lambda () expr)`. That is, it hides the expression inside a λ. `force` calls this λ, recovering the expression.

Simply doing this creates the behavior we need. Consider our previous example, that of finding primes (`get-prime`). Here's what happens:

```
(define (get-prime low high)
  (filter-stream prime?
                 (enumerate-interval low high)))
(head (tail (get-prime 10000 1000000000000000)))
```

[1] This is indeed how the video lectures explain it, see Lecture 6A at timestamp 47:39.

We can use the substitution model since everything is purely functional:

```
(enumerate-interval 10000 10000000000000000)
```

How is `enumerate-interval` defined?

```
(define (enumerate-interval 10000 10000000000000000)
  (if (> 10000 10000000000000000)
      the-empty-stream
      (cons-stream 10000
                   (enumerate-interval (inc 10000) 10000000000000000))))
```

It does a `cons-stream` on `10000` and a *promise* to compute `(enumerate-interval 10001 10000000000000000)`. Or, explicitly written:

```
(cons 10000 (delay (enumerate-interval (inc 10000) 10000000000000000)))
```

Expanding out the `delay` and checking its output:

```
(cons 10000 (lambda () (enumerate-interval (inc 10000) 10000000000000000)))
```

```
(10000 . #<procedure:...CDEkB/ob-p1kSJ7.rkt:91:12>)
```

So our "stream" really is just a pair of the first element and some procedure that delay created. Calling `head` would just give us:

```
(head (cons 10000 (lambda () (enumerate-interval (inc 10000) 10000000000000000))))
```

```
10000
```

What would `tail` give us? It'll simply call the λ, giving us another pair:

```
(tail (cons 10000 (lambda () (enumerate-interval (inc 10000) 10000000000000000))))
```

```
(10001 . #<procedure:...CDEkB/ob-zfZJq9.rkt:25:4>)
```

Clearly, `enumerate-interval` was only called the second time *when* we called tail, and not before. This is where the efficiency is created.

Note that we're filtering the stream for `prime?`-ness. How does `filter-stream` work? Here's the definition:

```
(define (filter-stream pred s)
  (cond
    ((empty-stream? s) the-empty-stream)
    ((pred (head s))
     (cons-stream (head s)
                  (filter-stream pred (tail s))))
    (else (filter-stream pred (tail s)))))
```

Since it also uses `cons-stream`, the output it generates is also a promise-style pair. `filter-stream` will keep "tugging" on the stream (keep calling on promises, as in the `else` clause), until it finds a prime.

```
(#%require math/number-theory)
(filter-stream prime? (cons 10000 (lambda () (enumerate-interval (inc 10000)
↪  1000000000000000))))
```

```
(10007 . #<procedure:...CDEkB/ob-Mvek4B.rkt:25:4>)
```

Since we want to find the second prime, when we call `tail` on the stream, the `delay` is `force`'d, and we get the prime `10009`.

Note that we define `delay` and `cons-stream` as macros because we *don't* want their arguments evaluated: i.e., we want them to be lazy. This is only possible if we define them as macros and directly substitute the source before compilation.

12.4.1 A Small Optimization

Often, calling `tail` repeatedly will result in the recomputation of certain procedures that may already have been computed. This can get expensive, so we add memoization by slightly changing `delay`:

```
(define-syntax delay
  (syntax-rules ()
    ((delay expr)
     (memo-proc (lambda () expr)))))
```

`memo-proc` itself just remembers if it has already computed a procedure by using a flag (`already-run?`) and storing the value in `result`:

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

Streams, clearly, make life simpler and bring more modularity into our code.