# MIT 6.001 1986 Video Notes

Nebhrajani A.V.

June 8, 2021

# Contents

# 1 Introduction

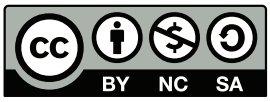
## 1.1 About

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.

## 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)
```

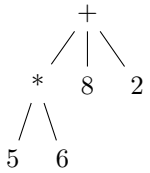
```
25.4
```

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
```

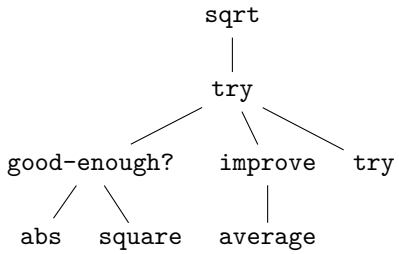
> **Note:** The `#i4` is Racket's syntax for using imprecise (decimals) instead of precise (fractions). Ignore it, and treat it as the number `4`.

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))

(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

```
square
+
```

```
#<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
- $\lambda$-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:

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

(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:

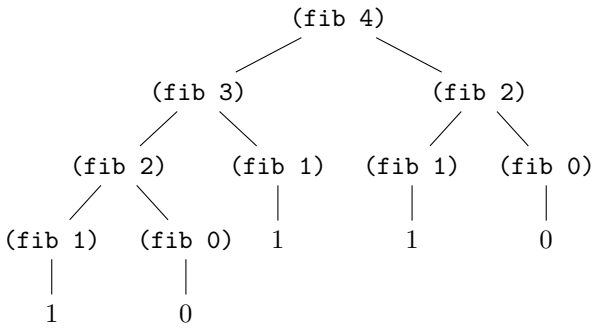
$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 \ldots$$

Let's write a Lisp function to calculate the $n$th 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 it 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

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

is why another function

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

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,

$$f(1) = \frac{2}{1} = 2$$
$$f(2) = \frac{2}{2} = 1$$
$$f(1) = \frac{2}{1} = 2$$
$$f(2) = \frac{2}{2} = 1$$
$$\dots$$

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:

```
(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))))
```

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 number 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)
```

[1] `let` is a Lisp primitive which takes as its first argument a list of definitions, and second input a list 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.

```
  (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:

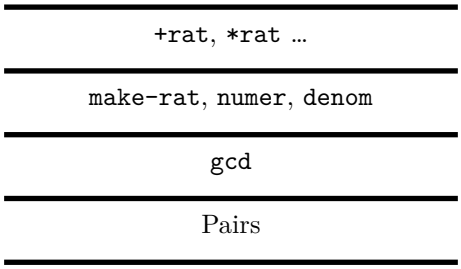
```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

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:




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:

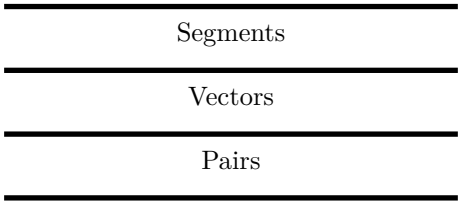
```
(define (make-seg v w)
  (cons v w))

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

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

Some sample operations:

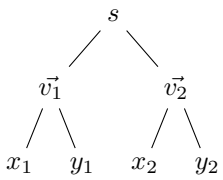
```
(define (midpoint s)
  (let ((a (seg-start s))
        (b (seg-end s)))
    (make-vec
     (average (xcor a) (xcor b))
     (average (ycor a) (ycor b)))))

(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:




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
```

## 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
```

[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.

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)))
  ```

- `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:

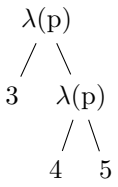
```
(define three-d-vec (our-cons 3 (our-cons 4 5)))
(our-car three-d-vec)
(our-car (our-cdr three-d-vec))
(our-cdr (our-cdr three-d-vec))
(our-cdr three-d-vec)
```

```
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.




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

# 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 rectangle.

### 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.

### 6.2.3 An Implementation

1. Rectangles

   Three vectors are needed to uniquely identify a rectangle 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, rectangle is implemented by:

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

   Note that technically, a rectangle 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 rectangle:

[1] $p \otimes p = p$

```
(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 rectangle.
     - For every element in `seglist`:
       * Draws the segment within rectangle, by scaling it correctly using `coord-map`.
       * This is done by giving `coord-map` the rectangle to scale to.
       * The procedure returned by `coord-map` then scales the vectors `(seg-start s)` and `(seg-end s)` to the rectangle.
       * 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 rectangle, and `make-picture` generates this procedure from a `seglist`. Or, in use:

```
(define R (make-rect ;some vectors
           ))
(define draw-george-in-rectangle (make-picture ;some seglist
                  ))
(draw-george-in-rectangle 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-rect
         (origin rect)
         (scale a (horiz rect))
         (vert rect)))
    (p2 (make-rect
         (+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-rect
           (+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/s 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) '+)))

(define (product? expr)
  (and (not (atom? expr))
       (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:

```
(define (make-sum a1 a2)
  (list '+ a1 a2))

(define (make-product m1 m2)
  (list '* m1 m2))
```

Finally, we must define the selectors:

```
(define a1 cadr)
(define a2 caddr)

(define m1 cadr)
(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]

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?`.

[1] LISP actually stands for LISt Processing.

### 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.

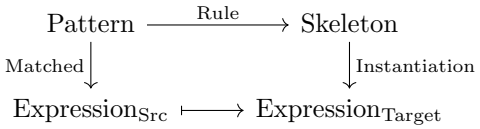
See that there's a higher-order idea here, of explaining rules to Lisp and having the rules applied to an input expression to "simplify" it. Our style of writing a rule-based substitution program is:

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)
```

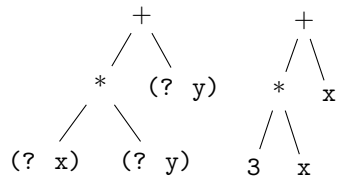
## 8.3 Implementation

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).

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

### 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:



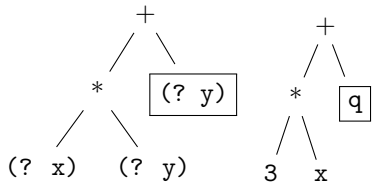

Clearly, these expressions should be matched in the tree traversal. Don't confuse the `(? x)` in the rule LHS with the symbol `x` in the expression: for the rule, it's just a matching variable, but the `x` in the expression goes into the dictionary.

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.

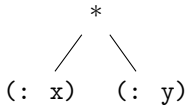
Finally, we implement as more cases in the `cond` other things we may need to match, according to the rule language (8.1).

```
(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)

        (else
         (match (cdr pattern)
                (cdr expression)
                (match (car pattern)
                       (car expression)
                       dict)))))
```

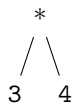
We add one case wherein both the pattern to be matched and the expression are atomic and the same, (the case `foo` in 8.1), in which we simply return the dictionary, since the pattern matches. The other new cases are self-explanatory: they are merely matching `(?c)`, `(?v)`, and `(?)`.

### 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:

```
(define (instantiate skeleton dict)
  (cond ((atom? skeleton) skeleton)
        ((skeleton-evaluation? skeleton)
         (evaluate (evaluation-expression skeleton)
                   dict))
        (else (cons (instantiate (car skeleton) dict)
                    (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:

```
(define (evaluation-expression evaluation) (cadr evaluation))

(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))
```

[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."

```
                    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.
- `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 e1) (?c e2))          (: (op e1 e2))                )
    ( ((? op) (? e1) (?c e2))           ((: op) (: e2) (: e1))        )
    ( (+ 0 (? e))                       (: e)                         )
    ( (* 1 (? e))                       (: e)                         )
    ( (* 0 (? e))                       0                             )
    ( (* (?c e1) (* (?c e2) (? e3)))    (* (: (* e1 e2)) (: e3))      )
    ( (* (? e1) (* (?c e2) (? e3)))     (* (: e2) (* (: e1) (: e3))) )
    ( (* (* (? e1) (? e2)) (? e3))      (* (: e1) (* (: e2) (: e3))) )
    ( (+ (?c e1) (+ (?c e2) (? e3)))    (+ (: (+ e1 e2)) (: e3))      )
    ( (+ (? e1) (+ (?c e2) (? e3)))     (+ (: e2) (+ (: e1) (: e3))) )
    ( (+ (+ (? e1) (? e2)) (? e3))      (+ (: e1) (+ (: e2) (: e3))) )
    ( (+ (* (?c c) (? a)) (* (?c d) (? a)))
      (* (: (+ c d)) (: a))         )
    ( (* (? c) (+ (? d) (? e)))         (+ (* (: c) (: d))
```

```
                                        (* (: c) (: e)))          )
    ))

(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))
```

Excellent. 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.