

# Procedural abstraction and recursion

6.S184 - Zombies drink caffeinated 6.001

Alex Vandiver (alexmv)

Massachusetts Institute of Technology

Lecture 1 - January 10th, 2011

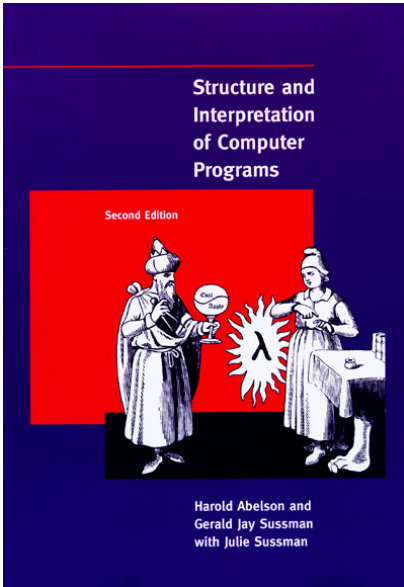
- TR, 7-9PM, through the 2nd
- <http://web.mit.edu/alexmv/6.S184/>
- Email: [6.001-zombies@mit.edu](mailto:6.001-zombies@mit.edu)
- Four projects: due on the 17th, 24th, 31st, and 7th.
- Space for ~30 graded students.
- Graded P/D/F

## Project 1

- Project 1 is out today
- Due one week from today; at 7pm on the 17th
- Start on the first half before the next lecture
- Mail to [6.S184-projects@mit.edu](mailto:6.S184-projects@mit.edu) if for credit, [6.S184-projects-no-credit@mit.edu](mailto:6.S184-projects-no-credit@mit.edu) if not

## Prerequisites

- Some programming clue
- A copy of Racket (Formerly PLT Scheme / DrScheme)  
<http://www.racket-lang.org/>
- Free time



- Structure and Interpretation of Computer Programs by Harold Abelson and Gerald Jay Sussman
- <http://mitpress.mit.edu/sicp/>
- Not required reading
- Useful as study aid and reference
- Roughly one lecture per chapter

- Lisp invented in 1959 by John McCarthy
- Scheme invented in 1975 by Guy Steele and Gerald Sussman
- Hardware Lisp machines, around 1978
- 6.001 first taught in 1980
- SICP published in 1984 and 1996
- R<sup>6</sup>RS in 2007
- 6.001 last taught in 2007
- 6.S184 first taught in 2009

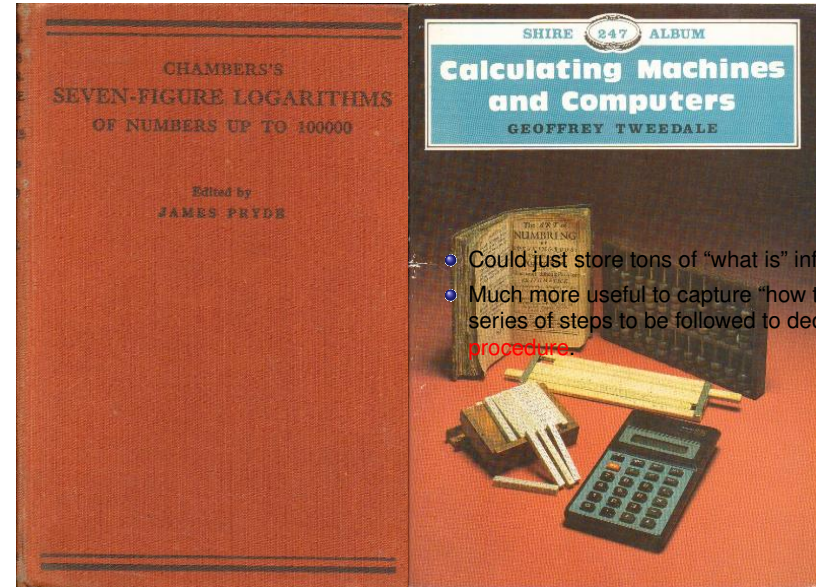
Harold Abelson and Gerald Jay Sussman with Julie Sussman

- This is not a class to teach Scheme
- Nor really a class about programming at all
- This is a course about **Computer Science**
- ...which isn't about computers
- ...nor actually a science
- This is actually a class in **computation**

- **Procedural** and data **abstraction**
- Conventional interfaces & programming paradigms
  - Type systems
  - Streams
  - Object-oriented programming
- Metalinguistic abstraction
  - Creating new languages
  - Evaluators

- “How to” knowledge
- To approximate  $\sqrt{x}$  (Heron’s Method):
  - Make a guess  $G$
  - Improve the guess by averaging  $G$  and  $\frac{x}{G}$
  - Keep improving until it is good enough

$$\begin{aligned}
 x = 2 \quad G &= 1 \\
 \frac{x}{G} = 2 \quad G &= \frac{(1+2)}{2} = 1.5 \\
 \frac{x}{G} = \frac{4}{3} \quad G &= \frac{(\frac{3}{2} + \frac{4}{3})}{2} = 1.4166 \\
 \frac{x}{G} = \frac{24}{17} \quad G &= \frac{(\frac{17}{12} + \frac{24}{17})}{2} = 1.4142
 \end{aligned}$$



- Could just store tons of “what is” information
- Much more useful to capture “how to” knowledge – a series of steps to be followed to deduce a value – a **procedure**.

Need a language for describing processes:

- Vocabulary – **basic primitives**
- Rules for writing compound expressions – **syntax**
- Rules for assigning meaning to constructs – **semantics**
- Rules for capturing process of evaluation – **procedures**

- Numbers
  - As floating point values
  - In IEEE 754 format
  - Stored in binary
  - In registers
  - Made up of bits
  - Stored in flip-flops
  - Made of inverters
  - Implemented by transistors
  - On silicon
  - With electrical potential
  - Of individual electrons
  - Made of leptons
  - ...

- We assume that our language provides us with a basic set of data elements:
  - Numbers
  - Characters
  - Booleans
- It also provides a basic set of operations on these primitive elements
- We can then focus on using these basic elements to construct more complex processes

- Legal expressions have rules for constructing from simpler pieces – the **syntax**.
- (Almost) every expression has a value, which is “returned” when an expression is “evaluated.”
- Every value has a type.
- The latter two are the **semantics** of the language.

## Language elements – primitives

Self-evaluating primitives – value of expression is just object itself:

**Numbers** 29, -35, 1.34, 1.2e5

**Strings** “this is a string” “odd # \$ % # \$ things number 35”

**Booleans** #t, #f

## Language elements – primitives

Built-in procedures to manipulate primitive objects:

**Numbers** +, -, \*, /, >, <, >=, <=, =

**Strings** string-length, string=?

**Booleans** and, or, not

## Names for built-in procedures

- +, -, \*, /, =, ...
- What is the **value** of them?
- + → #<procedure:+>
- Evaluate by looking up value associated with the name in a special table – the **environment**.

- How do we create expressions using these procedures?
- (+ 2 3)
  - Open paren
  - Expression whose value is a procedure
  - Other expressions
  - Close paren
- This type of expression is called a combination
- Evaluate it by getting values of sub-expressions, then applying operator to values of arguments.
- You now know all there is to know about Scheme syntax! (almost)

- Note the recursive definition – can use combinations as expressions to other combinations:
  - (+ (\* 2 3) 4) → 10
  - (\* (+ 3 4) (- 8 2)) → 42

- In order to abstract an expression, need a way to give it a name  
(define score 23)
- This is a **special form**
  - Does not evaluate the second expression
  - Rather, it pairs the name with the value of the third expression
- The return value is **unspecified**

- To get the value of a name, just look up pairing in the environment

```
(define score 23)           →      undefined
score                       →      23
(define total (+ 12 13))   →      undefined
(* 100 (/ score total))   →      92
```

Rules for evaluation:

- If **self-evaluating**, return value
- If a **name**, return value associated with name in environment
- If a **special form**, do something special
- If a **combination**, then
  - Evaluate all of the sub-expressions, in any order
  - Apply the operator to the values of the operands and return the result

## Mathematical operators are just names

```
(+ 3 5)           →      8
(define fred +)   →      undefined
(fred 3 6)        →      9
```

- + is just a name
- + is bound to a value which is a procedure
- line 2 binds the name `fred` to that same value

## All names are names

```
(+ 3 5)           →      8
(define + *)       →      undefined
(+ 3 5)           →      15
```

- There's nothing "special" about the operators you take for granted, either!
- Their values can be changed using `define` just as well

- To capture a way of doing things, create a procedure:  
`(lambda (x) (* x x))`
- `(x)` is the list of **parameters**
- `(* x x)` is the **body**
- `lambda` is a special form: create a procedure and returns it

- Use this anywhere you would use a built-in procedure like `+`:  
`( (lambda (x) (* x x)) 5 )`
- **Substitute** the value of the provided arguments into the body:  
`(* 5 5)`
- Can also give it a name:  
`(define square (lambda(x) (* x x)))`  
`(square 5) → 25`
- This creates a loop in our system, where we can create a complex thing, name it, and treat it as a primitive like `+`

### Rules for evaluation:

- If self-evaluating, return value
- If a name, return value associated with name in environment
- If a special form, do something special.
- If a combination, then
  - Evaluate all of the sub-expressions, in any order
  - Apply the operator to the values of the operands and return the result

### Rules for applying:

- If primitive, just do it
- If a compound procedure, then substitute each formal parameter with the corresponding argument value, and evaluate the body

```
(lambda (x) (* x x))
=> #<procedure>
(define square (lambda (x) (* x x)))
=> undefined
(square 4)
=> (* 4 4)
=> 16
```

### “Syntactic sugar”:

```
(define (square x) (* x x))
=> undefined
```

- **Syntax:** `(lambda (x y) (/ (+ x y) 2))`
- 1st operand is the **parameter list:** `(x y)`
  - a list of names (perhaps empty)
  - determines the number of operands required
- 2nd operand is the **body:** `(/ (+ x y) 2)`
  - may be any expression
  - not evaluated when the lambda is evaluated
  - evaluated when the procedure is applied

```
(define x (lambda () (+ 3 2)))
x
(x)
```

```
→ undefined
→ #<procedure>
→ 5
```

The value of a lambda expression is a procedure

## What does a procedure describe?

Capturing a common pattern:

- `(* 3 3)`
- `(* 25 25)`
- `(* foobar foobar)`

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

Name for the thing that changes Common pattern to capture

## Modularity of common patterns

Here is a common pattern:

- `(sqrt (+ (* 3 3) (* 4 4)))`
- `(sqrt (+ (* 9 9) (* 16 16)))`
- `(sqrt (+ (* 4 4) (* 4 4)))`

Here is one way to capture this pattern:

```
(define square (lambda (x) (* x x)))
(define pythagoras
  (lambda (x y)
    (sqrt (+ (* x x) (* y y)))))
```

## Why?

- Breaking computation into modules that capture commonality
- Enables reuse in other places (e.g. `square`)
- Isolates (abstracts away) details of computation within a procedure from use of the procedure
- May be many ways to divide up:

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

```
(define pythagoras  
  (lambda (x y)  
    (sqrt (+ (square x) (square y)))))
```

```
(define square (lambda (x) (* x x)))  
(define sum-squares  
  (lambda (x y) (+ (square x) (square y))))  
(define pythagoras  
  (lambda (x y)  
    (sqrt (sum-squares x y))))
```

Recitation time!

## A more complex example

To approximate  $\sqrt{x}$ :

- 1 Make a guess  $G$
- 2 Improve the guess by averaging  $G$  and  $\frac{x}{G}$ :
- 3 Keep improving until it is good enough

Sub-problems:

- When is “close enough”?
- How do we create a new guess?
- How do we control the process of using the new guess in place of the old one?

## Procedural abstractions

“When the square of the guess is within 0.001 of the value”

```
(define close-enough?  
  (lambda (guess x)  
    (< (abs (- (square guess) x))  
      0.001)))
```

Note the use of the `square` procedural abstraction from earlier!

```
(define average
  (lambda (a b) (/ (+ a b) 2)))

(define improve
  (lambda (guess x)
    (average guess (/ x guess))))
```

- `average` is something we are likely to want to use again
- Abstraction lets us separate implementation details from use
  - Originally:
 

```
(define average
  (lambda (a b) (/ (+ a b) 2)))
```
  - Could redefine as:
 

```
(define average
  (lambda (x y) (* (+ x y) 0.5)))
```
  - There's actually a difference between those in Racket (exact vs inexact numbers)
  - No other changes needed to procedures that use `average`
  - Also note that parameters are internal to the procedure – cannot be referred to by name outside of the lambda

- Given `x` and `guess`, want `(improve guess x)` as new guess
- But only if the guess isn't good enough already
- We need to make a decision – for this, we need a new **special form**

```
(if predicate consequent alternative)
```

```
(if predicate consequent alternative)
```

- Evaluator first evaluates the `predicate` expression
- If it returns a true value (`#t`), then the evaluator evaluates and returns the value of the `consequent` expression
- Otherwise, it evaluates and returns the value of the `alternative` expression
- Why must this be a **special form**? Why can't it be implemented as a regular lambda procedure?

- So the heart of the process should be:

```
(define (sqrt-loop guess x)
  (if (close-enough? guess x)
      guess
      (sqrt-loop (improve guess x) x)))
```

- But somehow we need to use the value returned by `improve` as the new `guess`, keep the same `x`, and repeat the process
- Call the `sqrt-loop` function again and reuse it!

Now we just need to kick the process off with an initial guess:

```
(define sqrt
  (lambda (x)
    (sqrt-loop 1.0 x)))

(define (sqrt-loop guess x)
  (if (close-enough? guess x)
      guess
      (sqrt-loop (improve guess x) x)))
```

- How do we know it works?
- Fall back to **rules for evaluation** from earlier

Rules for evaluation:

- If self-evaluating, return value
- If a name, return value associated with name in environment
- If a special form, do something special.
- If a combination, then
  - Evaluate all of the sub-expressions, in any order
  - Apply the operator to the values of the operands and return the result

Rules for applying:

- If primitive, just do it
- If a compound procedure, then **substitute** each formal parameter with the corresponding argument value, and evaluate the body

The **substitution model** of evaluation

... is a lie and a simplification, but a useful one!

```

(sqrt 2)
((lambda (x) (sqrt-loop 1.0 x)) 2)
(sqrt-loop 1.0 2)
((lambda (guess x)
  (if (close-enough? guess x)
      guess
      (sqrt-loop (improve guess x) x))) 1.0 2)
(if (close-enough? 1.0 2)
    1.0
    (sqrt-loop (improve 1.0 2) 2))
(sqrt-loop (improve 1.0 2) 2)
(sqrt-loop ((lambda (a b) (/ (+ a b) 2)) 1.0 2) 2)
(sqrt-loop (/ (+ 1.0 2) 2) 2)
(sqrt-loop 1.5 2)
...
(sqrt-loop 1.4166 2)
...

```

- Compute  $n$  factorial, defined as:  
 $n! = n(n-1)(n-2)(n-3)\dots 1$
- How can we capture this in a procedure, using the idea of finding a common pattern?

## Recursive algorithms

- 1 Wishful thinking
- 2 Decompose the problem
- 3 Identify non-decomposable (smallest) problems

### Wishful thinking

- Assume the desired procedure exists
- Want to implement `factorial`? Assume it exists.
- **But**, it only solves a smaller version of the problem
- This is just finding the common pattern; but here, solving the bigger problem involves the same pattern in a smaller problem

### Decompose the problem

- Solve a smaller instance
- Convert that solution into desired solution  
 $n! = n(n-1)(n-2)\dots = n[(n-1)(n-2)\dots] = n * (n-1)!$

```
(define fact (lambda (n) (* n (fact (- n 1)))))
```

### Identify non-decomposable problems

- Must identify the “smallest” problems and solve explicitly
- Define `1!` to be `1`

## Minor difficulty

```

(define fact
  (lambda (n) (* n (fact (- n 1)))))

(fact 2)
(* 2 (fact 1))
(* 2 (* 1 (fact 0)))
(* 2 (* 1 (* 0 (fact -1))))
(* 2 (* 1 (* 0 (* -1 (fact -2)))))
:
:

```

- 1 Wishful thinking
- 2 Decompose the problem
- 3 Identify non-decomposable (smallest) problems

## Wishful thinking

- Assume the desired procedure exists
- Want to implement `factorial`? Assume it exists.
- **But**, it only solves a smaller version of the problem
- This is just finding the common pattern; but here, solving the bigger problem involves the same pattern in a smaller problem

## Decompose the problem

- Solve a smaller instance
  - Convert that solution into desired solution
- $$n! = n(n-1)(n-2)\dots = n[(n-1)(n-2)\dots] = n * (n-1)!$$

```
(define fact (lambda (n) (* n (fact (- n 1)))))
```

## Identify non-decomposable problems

- Must identify the “smallest” problems and solve explicitly
- Define 1! to be 1

- Have a **test**, a **base case**, and a **recursive case**

```
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1))))))
```

- More complex algorithms may have multiple base cases or multiple recursive cases

```
(define fact (lambda (n)
  (if (= n 1) 1 (* n (fact (- n 1)))))
```

```
(fact 3)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(if #f 1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(* 3 (fact 2))
(* 3 (if (= 2 1) 1 (* 2 (fact (- 2 1)))))
(* 3 (if #f 1 (* 2 (fact (- 2 1)))))
(* 3 (* 2 (fact (- 2 1))))
(* 3 (* 2 (fact 1)))
(* 3 (* 2 (if (= 1 1) 1 (* 1 (fact (- 1 1)))))
(* 3 (* 2 (if #t 1 (* 1 (fact (- 1 1)))))
(* 3 (* 2 1))
(* 3 2)
6
```

## Effects of recursive algorithms

Recursive algorithms consume more **space** with bigger operands!

```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 1)))
...
24 (fact 8)
(* 8 (fact 7))
(* 8 (* 7 (fact 6)))
(* 8 (* 7 (* 6 (fact 5))))
...
(* 8 (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 (fact 1))))))))
(* 8 (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 1))))))
(* 8 (* 7 (* 6 (* 5 (* 4 (* 3 2))))))
...
40320
```

- Try computing  $101!$   
 $101 * 100 * 99 * 98 * 97 * 96 * \dots * 2 * 1$
- How much space do we consume with pending operations?
- Better idea: count down, doing one multiplication at a time
  - Start with 1 as the answer
  - Multiply by 101, store 101 as the current answer, remember 100 is next
  - Multiply the those two, store 10100 as the current answer, remember 99 is next
  - Multiply those two, store 999900 as the current answer, remember 98 is next next
  - ...
  - Multiply by one, get 9425947759838359420851623124482936749562  
 312794702543768327889353416977599316221476503087  
 861591808346911623490003549599583369706302603264 0000000000000000000000, and stop.
- This is an **iterative** algorithm – it uses constant space

product	next
1	101
101	100
10100	99
999900	98
97990200	97

- **First row** handles 0! cleanly
- **product** becomes **product \* next**
- **next** becomes **next - 1**
- The answer is **product** when **next = 1**

```
(define (ifact n) (ifact-helper 1 n))
```

```
(define (ifact-helper product next)
  (if (= next 1)
      product
      (ifact-helper (* product next)
                    (- next 1))))
```

- The helper has **one argument per column**
- Which is **called by ifact**
- Which provides the **values for the first row**
- The recursive call to `ifact-helper` computes the **next row**
- And the `if` statement checks the **end condition** and **output value**

```
(define (ifact-helper product next)
  (if (= next 1)
      product
      (ifact-helper (* product next)
                    (- next 1))))
```

```
(ifact 4)
(ifact-helper 1 4)
(if (= 4 1) 1 (ifact-helper (* 1 4) (- 4 1)))
(ifact-helper 4 3)
(if (= 3 1) 4 (ifact-helper (* 4 3) (- 3 1)))
(ifact-helper 12 2)
(if (= 2 1) 12 (ifact-helper (* 12 2) (- 2 1)))
(ifact-helper 24 1)
(if (= 1 1) 24 (ifact-helper (* 24 1) (- 1 1)))
24
```

- Recursive factorial:

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

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

- Pending operations make the expression grow continuously.

- Iterative factorial:

```
(define (ifact-helper product next)
  (if (= next 1)
      product
      (ifact-helper (* product next)
                    (- next 1))))
```

```
(ifact-helper 1 4)
(ifact-helper 4 3)
(ifact-helper 12 2)
(ifact-helper 24 1)
```

- Fixed space because no pending operations

## Iterative processes

- Iterative algorithms have constant space
- To develop an iterative algorithm:
  - Figure out a way to accumulate partial answers
  - Write out a table to analyze:
    - initialization of first row
    - update rules for other rows
    - how to know when to stop
  - Translate rules into Scheme
- Iterative algorithms have no pending operations

Recitation time!

- Lambdas allow us to create procedures which capture processes
- Procedural abstraction creates building blocks for complex processes
- Recursive algorithms capitalize on “wishful thinking” to reduce problems to smaller subproblems
- Iterative algorithms similarly reduce problems, but based on progressive rules