



# CS1 Recitation

Week 2

# Sum of Squares

- Write a function that takes an integer  $n$ 
  - $n$  must be at least 0
  - Function returns the sum of the square of each value between 0 and  $n$ , inclusive
- Code:

```
(define (square x) (* x x))
(define (sum-squares n)
  (if (= 0 n)
      0
      (+ (square n) (sum-squares (- n 1)))))
```

# Sum of Squares (2)

- Code:

```
(define (sum-squares n)
  (if (= 0 n)
      0
      (+ (square n) (sum-squares (- n 1)))))
```

- What kind of process is this?

- Linear recursive
- Long chain of deferred operations

- `(sum-squares 5)` →  
`(+ 25 (+ 16 (+ 9 (+ 4 (+ 1 0)))))`

- Why?

- Recursive call is part of an operand in this function
- Computation for “this step” can’t complete until recursion ends

# Iterative Sum of Squares

- Need some “state” to convert to linear iterative process
  - Will still have a recursive function though!
- State to maintain?
  - Sum computed so far
  - Maximum value to process

- Code:

```
(define (isum-squares n) (iss-helper 0 n))
(define (iss-helper sum n)
  (if (= 0 n)
      sum
      (iss-helper (+ (square n) sum) (- n 1))))
```

# Iterative Sum of Squares (2)

- Code:

```
(define (isum-squares n) (iss-helper 0 n))  
(define (iss-helper sum n)  
  (if (= 0 n)  
      sum  
      (iss-helper (+ (square n) sum) (- n 1))))
```

- Any deferred operations?

- No (why?)
- Recursive call is in the operator, not in an operand
- All work for this step is completed before recursive call is made

- Linear iterative process

- Space requirement is constant

# Linear Recursive vs. Linear Iterative

- Both linear recursive process and linear iterative process use recursive procedures
  - Term is about how the computation unfolds, not whether the function calls itself
- Linear recursive process
  - Produces deferred operations
- Linear iterative process
  - Uses state variables to avoid deferred ops.
  - Often implemented with a helper function
    - Helper function includes the state vars, is called by a wrapper

# Linear Recursive vs. Linear Iterative (2)

- Linear recursive sum-of-squares:

```
(define (sum-squares n)
  (if (= 0 n)
      0
      (+ (square n) (sum-squares (- n 1)))))
```

Produces deferred operations

- Linear iterative sum-of-squares:

```
(define (isum-squares n) (iss-helper 0 n))
(define (iss-helper sum n)
  (if (= 0 n)
      sum
      (iss-helper (+ (square n) sum) (- n 1))))
```

State variable, updated before recursion

# Increment/Decrement Function

- Write a function to increment or decrement the input value, based on a flag
  - Select inc/dec with an input flag
  - #t = increment, #f = decrement
- Code:

```
(define (incdec val flag)
  (if flag (+ val 1) (- val 1)))
```
- A bit repetitive...

# Functions As Results

- In Scheme, an expression's result can also be a *function*, not just a value
  - + and – are Scheme functions...
- Original `incdec` code:

```
(define (incdec val flag)
  (if flag (+ val 1) (- val 1)))
```
- New `incdec` code:

```
(define (incdec val flag)
  ((if flag + -) val 1))
```

  - When expression for operator is evaluated, *flag* value selects + or – function to apply to operands

# Functions As Results (2)

- Evaluate `(incdec 3 #t)`
  - Evaluate `3`  $\rightarrow$  `3`, evaluate `#t`  $\rightarrow$  `#t`
  - Evaluate `incdec`  $\rightarrow$   
`(lambda (val flag) ((if flag + -) val 1))`
  - Apply lambda expression to `3`, `#t`
    - Substitute `3` for `val`, `#t` for `flag` in `((if flag + -) val 1)`  $\rightarrow$   
`((if #t + -) 3 1)`
    - Evaluate `((if #t + -) 3 1)`
      - Evaluate `3`  $\rightarrow$  `3`, evaluate `1`  $\rightarrow$  `1`
      - Evaluate `(if #t + -)`
        - `if` is a special form. Evaluate first argument.
        - Evaluate `#t`  $\rightarrow$  `#t`
        - `#t` is true so evaluate second argument: `+`  $\rightarrow$  `+`
    - Apply `+` to `3`, `1`  $\rightarrow$  `4`

# cond Expressions

- Form:

```
(cond (test1 exp1)  
      (test2 exp2)  
      ...  
      (else expN))
```

- If first test evaluates to #t then first expression is evaluated
- Else if second test evaluates to #t then second expression is evaluated
- *etc.*
- If no tests evaluate to true then `else` expression is evaluated (if present)

# Fibonacci Series with `cond`

- Definition:

- $\text{fib}(0) = 0$

- $\text{fib}(1) = 1$

- $\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$

- Scheme code:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1)) (fib (- n 2))))))
```

# Pascal's Triangle

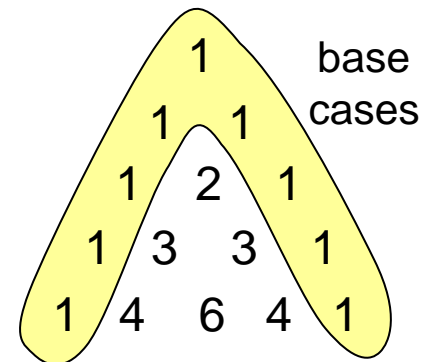
- Write a function to compute values in Pascal's Triangle

- (`pascal-coefficient level n`)

- First level is 0

- First value in a level is index 0

- Indexes in a level  $L$  are  $0..L$



- Follow pattern for designing recursive operations

- Identify base case(s)

- Figure out out decomposition step

- How to compute a given level and index, using recursive function calls and simple operations?

# Coding Style

- “Good code, bad code” focuses on naming
  - Short function/variable names may save typing up front...
  - They waste more time during maintenance/bug-fixing!
- Ideally, code is *self-documenting*
  - Names should clearly describe what is going on
    - `factor?` vs. `fac?` or `f?`
  - Don't be over-verbose!
    - `is-arg-one-a-factor-of-arg-two?`
- Some short names are OK because they're widely used
  - `x` is usually fine for a single decimal value, e.g. (`square x`)
  - `n` for an integer maximum value, e.g. (`sum-ints n`)
  - `i` for a variable that iterates over a sequence of integer values