CS 4
Fundamentals of Computer Programming

Lecture 5: January 14, 2015

Higher-order functions, part 1

```
f
(f f)
(f (f f))
((f f) (f f))
(f (f f) (f (f f)))
```

```
(define k (lambda (x)(lambda (y) x))
(define s
  (lambda (f)
    (lambda (g)
      (lambda (x) ((f x) (g x))))))
```

Last time

- Asymptotic complexity
  - time and space
  - big-O notation
  - big-Θ notation

Today

- A really big idea:
  - procedures (functions) are *data*
  - can be passed to other procedures as arguments
  - can be *created inside* procedures
  - can be *returned from* procedures
- Provides big increase in abstractive power
  - Not available in most computer languages
- Interlude: Lambda calculus
- *Also:* Some more ideas on how to design recursive procedures

In mathematics…

- Not all operations take in (only) numbers
- `+`, `-`, `*`, `/`, `expt`, `log`, `mod`, …
  - take in numbers, return numbers
- but operations like Σ, `d/dx`, integration ($\int$)
  - take in *functions*
  - return numbers or functions

Math: Functions as Arguments

- You've seen:

$$a = \sum_{n=0}^{6} f(n)$$

a=f(0)+f(1)+f(2)+f(3)+f(4)+f(5)+f(6)

Math: Functions as Arguments

- Σ is a "function"
  - which takes in
    - a function
    - a lower bound (an integer)
    - an upper bound (also an integer)
  - and returns
    - a number

$$\sum_{x=0}^{6} f(x)$$

- We say that Σ is a "higher-order" function
- Can define higher-order functions in Scheme

Recall: design strategy

- First figure out base cases
- Then figure out how to decompose problem in terms of itself

Base case

```
(define (sum f low high)
  (if (> low high) ; base case
      0            ; nothing to sum
      ???))        ; how to decompose?
```

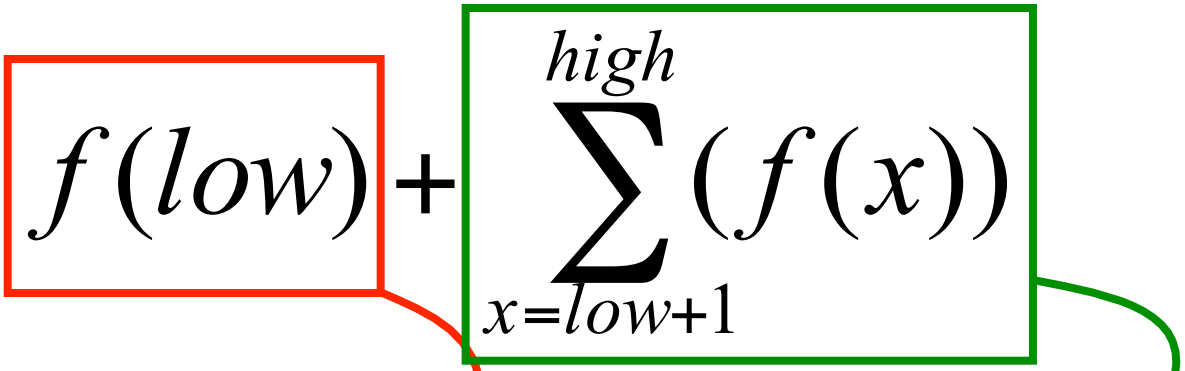
Decomposing summation

$$\sum_{x=low}^{high} f(x)$$

is the same as…

$$f(low) + \sum_{x=low+1}^{high} f(x)$$

Summation in Scheme



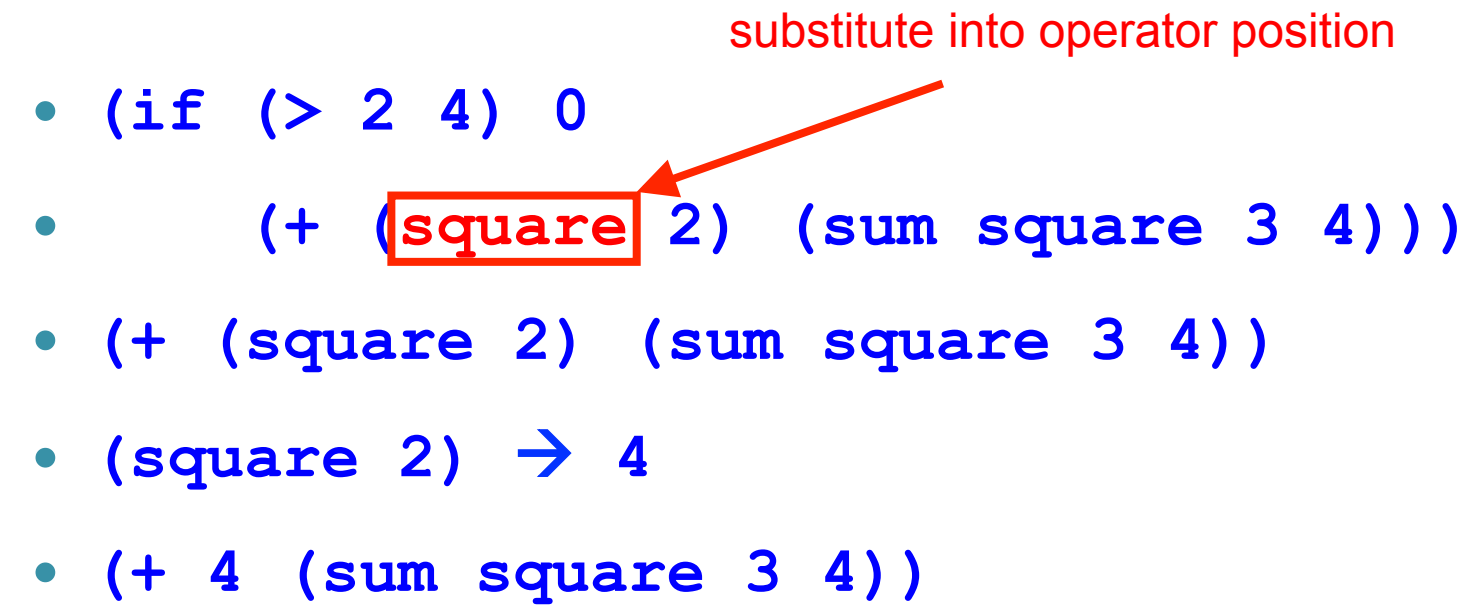

```
(define (sum f low high)
  (if (> low high)
      0
      (+ (f low)
         (sum f (+ low 1) high))))
```

Evaluating summation

[We'll skip some steps for brevity]

- Evaluate: `(sum square 2 4)`
- `((lambda (f low high) ...) square 2 4)`
- Substitute:
  - `square` for `f`
  - `2` for `low`, `4` for `high`

…continuing evaluation



…continuing evaluation

```
(+ 4 (sum square 3 4))
(+ 4 (if (> 3 4)
         0
         (+ (square 3)
            (sum square 4 4))))
(+ 4 (+ (square 3)
         (sum square 4 4)))
(+ 4 (+ 9 (sum square 4 4)))
```

…continuing evaluation

```
(+ 4 (+ 9 (sum square 4 4)))
```

- yadda yadda…

```
(+ 4 (+ 9 (+ 16 (sum square 5 4))))
(+ 4 (+ 9 (+ 16 (if (> 5 4) 0 …)
(+ 4 (+ 9 (+ 16 0)))
```

… `29`

- Pop quiz: what kind of process?
  - linear recursive

Also valid…

```
(sum (lambda (x) (* x x)) 2 4)
```

- This is also a valid call
- Equivalent in this case
- No need to give the `square` function a name if we only use it here

Iterative version

- `sum` generates a recursive process
- Iterative process would use less space
  - no pending operations
- Can we re-write to get an iterative version?

Designing the iterative version

- Need extra state variable to hold sum of terms encountered so far

```
(define (sum-iter f low high subtotal)
  ...)
```

- This will be our helper function
  - need to have another function to call it

Designing the iterative version

```
(define (isum f low high)
  (sum-iter f low high 0))

;; helper function
(define (sum-iter f low high subtotal)
  ...)
```

Base case

```
(define (isum f low high)
  (sum-iter f low high 0))

(define (sum-iter f low high subtotal)
  (if (> low high) ; we're done
      subtotal ; return amount summed so far
      ...)
```

Recursive case

```
(define (isum f low high)
    (sum-iter f low high 0))

(define (sum-iter f low high subtotal)
    (if (> low high)
        subtotal
        (sum-iter ???))) ; no pending operations
```

Recursive case

- Since we don't want pending operations...
- ...all the "action" must happen in the operands to recursive `sum-iter` call
- `f` → stays the same
- `low` → `(+ low 1)`
- `high` → stays the same
- `subtotal` → `(+ subtotal (f low))`

Recursive case

```
(define (sum-iter f low high subtotal)
    (if (> low high)
        subtotal
        (sum-iter f
                  (+ low 1)
                  high
                  (+ subtotal (f low)))))
```

Recursive vs. iterative process

- Recursive process:
  - pending computations
  - when recursive calls return, still work to do
- Iterative process:
  - current state of computation stored in last operand of `sum-iter` procedure (`subtotal`)
  - when recursive calls return, no more work to do

Evaluating iterative version

- "Exercise for the student"
- Easy, based on what you already know

isum

```
(define (isum f low high)
    (sum-iter f low high 0))

(define (sum-iter f low high subtotal)
    (if (> low high)
        subtotal
        (sum-iter f (+ low 1) high
            (+ subtotal (f low)))))
```

`isum` with internal procedure

```
(define (isum f low high)
  (define (sum-iter f low high subtotal)
      (if (> low high)
          subtotal
          (sum-iter f (+ low 1) high
              (+ subtotal (f low)))))
  (sum-iter f low high 0))
```

- Preferable since no other procedure uses `sum-iter` anyway

"Scope" of names

- When evaluating `isum` with internal procedure, Scheme only allows the name `sum-iter` to be looked up from code inside `isum`
- *i.e.* `sum-iter` is only "visible" inside `isum`
- We say that `sum-iter` is in the scope of `isum`

"Scope" of names

- Conversely, names of arguments to **`isum`** are "in scope" inside the definition of **`sum-iter`**
- Can use this fact to simplify the definition of **`sum-iter`**

`isum` with internal procedure

```
(define (isum f low high)
  (define (sum-iter f low high subtotal)
      (if (> low high)
          subtotal
          (sum-iter f (+ low 1) high
              (+ subtotal (f low)))))
  (sum-iter f low high 0))
```

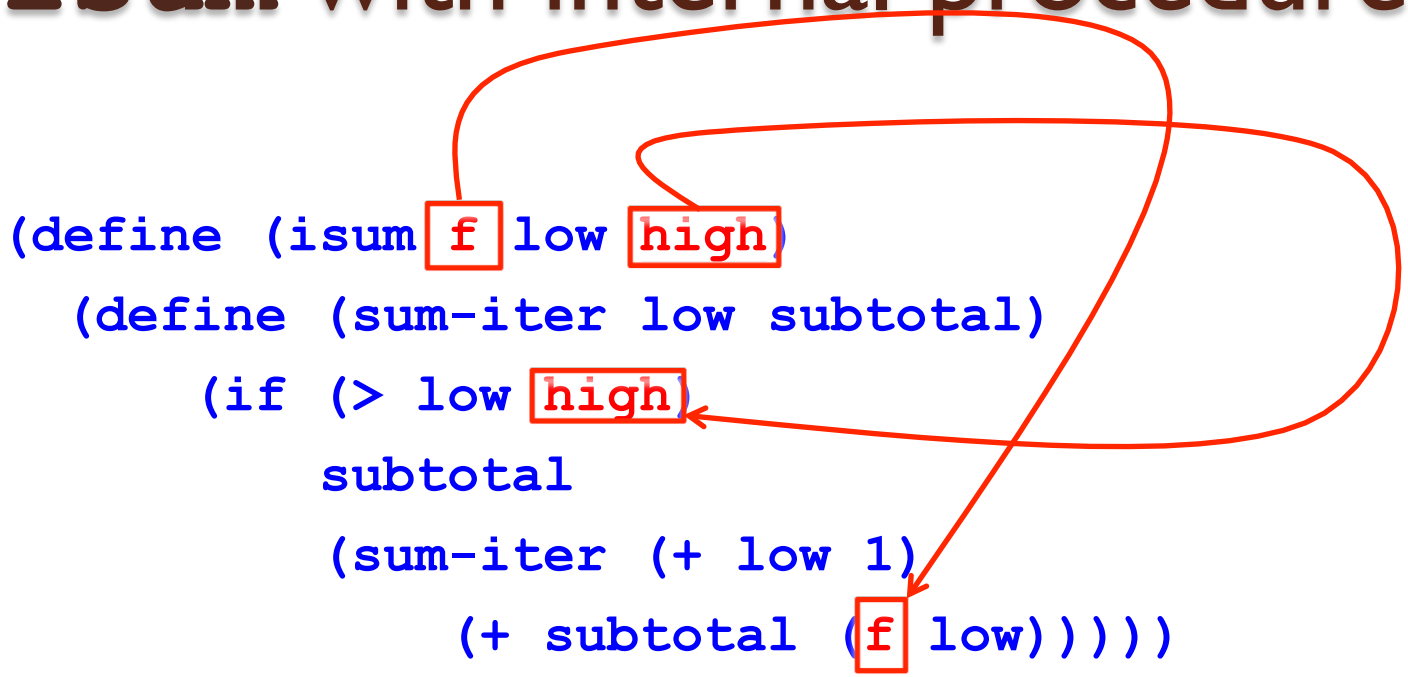
- Previous version

`isum` with internal procedure

```
(define (isum f low high)
  (define (sum-iter low subtotal)
      (if (> low high)
          subtotal
          (sum-iter (+ low 1)
                    (+ subtotal (f low)))))
  (sum-iter low 0))
```

- New version
- **`f`** and **`high`** arguments never change, so use the values in the "outer scope" (arguments of **`isum`**)

`isum` with internal procedure



```
(define (isum f low high)
  (define (sum-iter low subtotal)
      (if (> low high)
          subtotal
          (sum-iter (+ low 1)
              (+ subtotal (f low)))))
  (sum-iter low 0))
```

Historical interlude

Historical interlude

Reactions on first seeing "lambda":

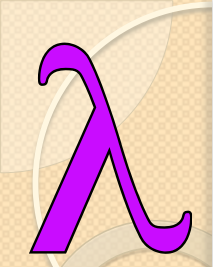
- What is this thing?
- What is it good for?
- Where does it come from?

Now it can be told:

- Where lambda (λ) comes from…

The 1930s

- Rumblings of war spreading through Europe…
- But in Princeton, NJ:
  - Alonzo Church (1903-1995)
  - theory of computation
    - called "lambda calculus"
    - based on “pure functions”

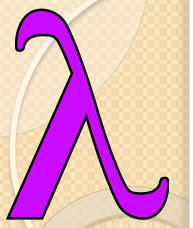


Church's idea

- Get rid of as many kinds of data as possible
  - no arrays, no lists, no compound data types
  - no characters, no strings
  - no booleans
  - no numbers!
- Keep *one* kind of data only:
  - Functions of a single operand (λ expressions)

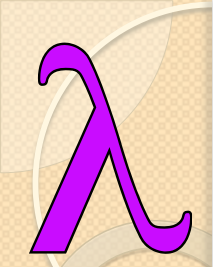
Lambda (λ) Calculus

- Only three types of expressions:
  - variables (*e.g.* **`x`**)
  - functions of one argument
    - *e.g.* **`(lambda (x) x)`**
      - λx.x in Church's notation
  - function applications
    - *e.g.* **`((lambda (x) x) y)`**
      - (λx.x) y in Church's notation



Lambda (λ) Calculus

- What can these expressions represent?
- Here's the cool part:
  - Lambda expressions can represent *anything* that can be computed!
  - Lists, booleans, even numbers!
  - Let's see some examples



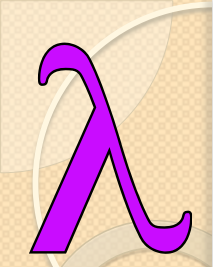
Numbers in λ-calculus

- zero:
- one:
- two:
- three:
- etc.
- These are called Church numerals

- (λ (s) (λ (z) z))
- (λ (s) (λ (z) (s z)))
- (λ (s) (λ (z) (s (s z))))
- (λ (s) (λ (z) (s (s (s z)))))

read s as "successor"

read z as "zero"



Arithmetic in λ-calculus

- f(n) = n + 1 →

```
(λ (n) (λ (s) (λ (z) (s ((n s) z)))))
```

- f(m, n) = m + n →

```
(λ (m) (λ (n) (λ (s) (λ (z)
    ((m s) ((n s) z))))))
```

- Nobody said this was easy! ☺
- but it's *possible*

Significance?

- Lambda calculus is very useful when studying the semantic foundations of programming languages
- Also: it's a model of universal computation (equivalent to a Turing machine)
- And: a direct inspiration for Scheme

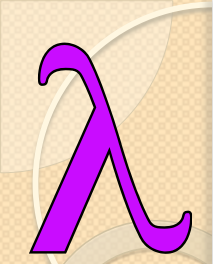
Influence on Scheme

- Gerald J. Sussman and Guy L. Steele, Jr.

  "Scheme: An Interpreter for Extended Lambda Calculus"

  MIT AI Lab. AI Lab Memo AIM-349. December 1975.

- http://library.readscheme.org/page1.html



Other uses...

- Even *recursion* is unnecessary if you have lambda expressions!
- Later, we'll talk about an amazing thing called the *Y combinator* which gives you recursion in a language (like lambda calculus) that doesn't have it built in
- Moral: if you have higher-order procedures (lambda expressions), you have everything!
  - *"lambda the ultimate"*

Back to higher-order procedures…

Generalizing summation

- What if we don't want to go up by 1?
- Supply *another* procedure
  - given current value, finds the next one

```
(define (gsum f low fnext high)
  (if (> low high)
      0
      (+ (f low)
         (gsum f (fnext low) fnext high))))
```

Functions to get next value

```
(define (step1 n) (+ n 1))
(define (sum f low high) ; same as before
  (gsum f low step1 high))

(define (mult2 n) (* n 2))
(define (sum2 f low high)
  (gsum f low mult2 high))
```

Generalized sums

- `(sum square 2 4)`

  $= 2^2 + 3^2 + 4^2$

- `(sum2 square 1 10)`

  $= 1^2 + 2^2 + 4^2 + 8^2$

- `(sum2 (lambda (n) (* n n n)) 1 20)`

  $= 1^3 + 2^3 + 4^3 + 8^3 + 16^3$

Using `lambda`

- `(define (mult2 n) (* n 2))`
- `(define (sum2 f low high)`
  `(gsum f low mult2 high))`
- Can just write this as:
- `(define (sum2 f low high)`
  `(gsum f low (lambda (n) (* n 2)) high))`
- Don't need to name tiny one-shot functions

Really using `lambda`...

- How about:
  - sum of n^4 for n = 1 to 100, stepping by 5?
-
```
(gsum (lambda (n) (* n n n n))
      1
      (lambda (n) (+ n 5))
      100)
```
- NOTE: the `n`'s in the lambdas are independent of each other
  - (`n`-dependent)

The point

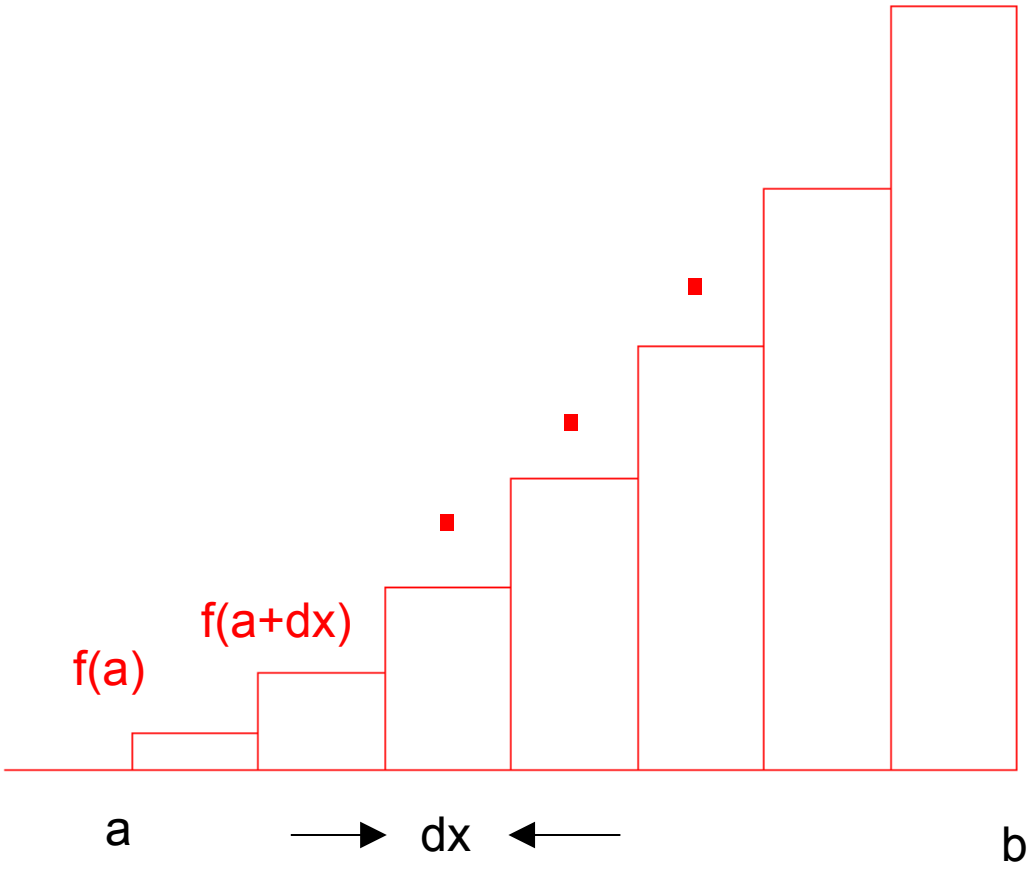
- Using **`gsum`**, we are able to define a large *class* of summation functions
  - summing up arbitrary functions
  - going up by 1, multiplying by 2, or whatever
- This is possible only because functions are "first class" *i.e.* data
- We abstract the part that changes (function to sum, **`fnext`** function) from the part that is always the same (summation)

Yet another function

$$\int_a^b f(x)dx$$

- Definite integral of *f(x)* from *a* to *b*
- If this was a sum, we could do it...

Approximate as a sum



Approximate as a sum

$$\int_a^b f(x)dx \approx \left(\sum_{x=a,+dx}^{b} f(x) \right) * dx$$

- Sum is from *a* to *b* in steps of *dx*
- OK if *dx* is small enough

in Scheme...

$$\int_a^b f(x)dx \approx \left(\sum_{x=a,+dx}^{b} f(x) \right) * dx$$

```
(define (integral f a b dx)
  (* dx
     (gsum f a (lambda (x) (+ x dx)) b)))
```

Example

$$\int_0^1 x^2 dx = 1/3$$

```
(integral (lambda (x) (* x x)) 0 1 0.0001)
```

→ `0.333383349999416`

```
(integral (lambda (x) (* x x)) 0 1 0.000001)
```

→ takes a long time…

Slight change of topic...

Analysis vs. synthesis

- So far have dealt mainly with *analysis*
  - given procedure, show how it evaluates
- Not what programmers mainly do
- Time to look at *synthesis* for a change
  - Given a problem, how to *design* a procedure to solve the problem?
- This is majority of programmer's job

Example problem

- Contrived (dumb) problem:
- Find the sum of all prime factors of a positive integer
- *e.g.*

  6 → 2 + 3 = 5

  35 → 5 + 7 = 12

  - etc.

Divide and conquer

- How to divide problem up into smaller chunks?
  - Imagine: what would you need to make this problem much easier to solve?
  - `prime?` procedure
    - returns `#t` if input number is prime
  - `smallest-prime-factor` procedure
    - returns smallest prime factor of input number
- "Wishful thinking"

Idea for a solution

- For any number
  - find the smallest prime factor
  - divide the number by this to get new number
    - find *its* smallest prime factor
    - divide it by this
      - etc. ...
- then add the prime factors together

Skeleton solution

- Write a skeleton of the solution
-
```
(define (sum-of-prime-factors n)
  ... )  ; ... is where rest of code goes
```
- Now fill in details
- First important detail is to identify the...

Base cases

- When do we know the answer immediately?
- When **`n`** is prime, answer is just **`n`**
- Update the skeleton:

```
(define (sum-of-prime-factors n)
  (if (prime? n) ; base case
      n
      ...))
```

- Rule: *always* handle base cases first!

Other cases

- If `n` is not prime, must have smallest prime factor `m`
- How to use *that* to make problem smaller?
  - add smallest prime factor (`m`) to ...
  - sum of prime factors of ...
    - `n / m`
- Update skeleton...

Filling in the blanks

```
(define (sum-of-prime-factors n)
  (if (prime? n)
      n
      (+ (smallest-prime-factor n)
         ... )))
```

- Still need to compute sum of prime factors of `n / m`
  - where `m` is `(smallest-prime-factor n)`

Filling in the blanks

```
(define (sum-of-prime-factors n)
  (if (prime? n)
      n
      (+ (smallest-prime-factor n)
         (sum-of-prime-factors
           (/ n (smallest-prime-factor n))))))
```

- Almost done
  - Still need to write **`prime?`** and **`smallest-prime-factor`**

Note (efficiency)

- Inefficient to compute

  `(smallest-prime-factor n)` twice

- We'll soon learn a way to avoid doing this

Copping out

- `prime?` done in book
- `smallest-prime-factor` will be an exercise in lab 2
- So we're done!
- Rule 1: *Always avoid as much work as possible!*
  - (also known as: *Don't reinvent the wheel!*)

Extensions

- Exercise 1: rewrite using helper procedure
  - linear iterative process instead of linear recursive
- Exercise 2: rewrite with "sum" changed to "product"
  - extremely easy ☺ (trick question)
- Exercise 3: abstract into a higher-order procedure
  - replace `sum`, `product` with arbitrary procedure

Summary

- Procedures (functions) are *data*
- We can abstract operations around functions as well as numbers
- Provides great power
  - expression, abstraction
  - high-level formulation of techniques

Summary

- When designing procedures,
  - identify the base case
  - write that code immediately
  - for recursive case, figure out:
    - how to decompose problem in terms of smaller version of itself
    - how to combine recursive result with other values (if any) to get final result
  - use "wishful thinking" to decompose problems

Next time

- Returning functions from functions
  - (the other half of the higher-order function story)