

CS1: Introduction to Computation

Midterm Review



Midterm

- ... on the web site Real Soon Now
- Find linked from CS 1 home page
 - and in "schedule" section
- Due date: Tuesday, November 4, 5 PM
 - Lab 5 also due Thurs night 2am (Nov. 6)
 - yes, it sucks

Midterm

- Five hour time limit
 - (it shouldn't *really* take five hours)
- Closed book, closed interpreter
- No references except:
 - your own labs and TA's comments on them
 - electronic version of lecture slides
 - The Scheme standard (R5RS)
- No collaboration!

Midterm

- Five hour time limit
 - OK if not consecutive five hours, BUT
 - cannot go to CS 1 lecture, recitation, etc. between start and finish
 - shouldn't take more than 4 hours

Submission

- Submit electronically, **plain text format**
- Submit to CS1man as an assignment called "midterm"
- Don't submit e.g. PDF, Word doc
- Make fit into 80 columns
 - we have to print it out to grade it
- Penalties imposed for violating these rules!
- OK to use editor which matches parentheses
 - ...as long as it's not DrScheme...
 - e.g. emacs, gedit

Grading

- Score between 0.0 and 6.0
- No min grading

Midterm structure

Four parts to midterm:

- 1) Substitution model
 - 2) Asymptotic time complexity
 - 3) Recursion and function design
 - 4) Higher-order procedures and abstraction
-
- Only material through Day 7.
 - No cons/car/cdr stuff.

Substitution model

- Thought process:
- 1) Is this a special form?
 - `define`, `lambda`, `if`, `cond`, `and`, `or`
 - if so, use evaluation rule for that special form

Substitution model

- Thought process:
- 2) Otherwise:
 - evaluate operands
 - evaluate operator
 - apply operator to operands

Apply operator to operands

- If built-in procedure (+, * etc.)
 - just do it
- Otherwise,
 - procedure is a **lambda** expression
 - substitute values for variables in argument list of **lambda** expression, into the body
 - evaluate the resulting expression

Exception

- When do we not substitute values for variables in **lambda** expression?
- *Lambda shielding*

define

- Expect that you evaluate all **defines**
- Even *e.g.* **(define x 2)** should be evaluated
- Be aware of whether **define** is sugared or not!
 - if sugared, you need to explicitly desugar it
 - if first operand has parens, it's sugared

define

`(define x 2)`

- Obviously not sugared

`(define f (lambda (x) (* x (+ x 3))))`

- First operand doesn't have parentheses...
- Not a sugared lambda expression

define

```
(define (f x) (lambda (y) (* x (+ y 3))))
```

– Definitely sugared! Desugars to:

```
(define f
```

```
  (lambda (x)
```

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

– This function returns another function

define

- Evaluate: `(define (f x) (+ x (* 2 x)))`
 - Desugar to `(define f (lambda (x) (+ x (* 2 x))))`
 - `(lambda (x) ...)` evaluates to itself
 - Then bind name `f` to `lambda` expression
- And you're done

Example

```
(define (f x) (+ (* x x) 1))
```

```
(define (g x) (f (f x)))
```

```
(g 2)
```

Example (2)

`(define (f x) (+ (* x x) 1))`

- Desugar to `(define f (lambda (x) (+ (* x x) 1)))`
- `define` is special form; evaluate 2nd, bind to 1st
 - `(lambda (x) (+ (* x x) 1))` evaluates to itself
 - Bind `lambda` to name `f`

Example (3)

```
(define (f x) (+ (* x x) 1))
```

– ...

```
(define (g x) (f (f x)))
```

- Desugar to `(define g (lambda (x) (f (f x))))`
- `define` is special form; evaluate 2nd, bind to 1st
 - `(lambda (x) (f (f x)))` evaluates to itself
 - Bind `lambda` to name `g`

Example (4)

```
(define (f x) (+ (* x x) 1))
```

– ...

```
(define (g x) (f (f x)))
```

– ...

```
(g 2)
```

- $2 \rightarrow 2$
- $g \rightarrow (\text{lambda } (x) (f (f x)))$
- Apply **lambda** to 2
 - Substitute $x = 2$ into body of lambda $\rightarrow (f (f 2))$
 - Evaluate $(f (f 2))$

Important Note:

This lambda was already evaluated in the **define** statement! Here we simply apply it to the operands.

Example (5)

```
(define (f x) (+ (* x x) 1))  
(define (g x) (f (f x)))  
(g 2)
```

- ...
- Evaluate (f (f 2))
 - Evaluate (f 2)
 - Evaluate 2 → 2
 - Evaluate f → (lambda (x) (+ (* x x) 1))
 - Apply lambda to 2
 - » Substitute x = 2 in body of lambda → (+ (* 2 2) 1)
 - » etc.
 - Result: 5
 - Evaluate f → (lambda (x) (+ (* x x) 1))
 - etc.

Example: Recursion

```
(define (h a b)
  (if (< b 1)
      a
      (h (* a a) (/ b 2))))
(h 2 4)
```

Example: Returning Lambdas

```
(define (f z)
  (lambda (x y) (+ x y z)))
((f 2) 3 4)
```

```
(define (f y)
  (lambda (x y z) (+ x y z)))
((f 2) 3 4 5)
```

lambda shielding...

Example: `let` expressions

```
(define (f x y)
  (let ((z (+ x y))
        (w (- x y)))
    (* z w)))
(f 2 3)
```

Note: Desugar `let` into `lambdas` before substituting operands into body of `f`

```
(let ((z (+ x y)) (w (- x y))) (* z w)) →
((lambda (z w) (* z w)) (+ x y) (- x y))
```

Asymptotic time complexity

- Note that this is *worst-case* complexity
- Most important rule of thumb with time complexity is:
 - find the variable which determines the termination of the procedure
 - let's say it's x
 - complexity will be $O(f(x))$
 - where f is some function

Rule of thumb 2

- Second most important rule:
- How is the "termination variable" (e.g. x) changing between calls of the function?
- If it's increasing/decreasing by some fixed amount (e.g. 1) then probably linear complexity
- If it's going down by division (e.g. $x \rightarrow x/2$) then probably...
 - logarithmic (e.g. 16, 8, 4, 2, 1)

Rule of thumb 3

- If there is a "process within a process"
 - where for every iteration of process A, have to execute process B in its entirety
 - probably $\text{complexity}(A) * \text{complexity}(B)$
 - e.g. if both A and B are linear ($O(N)$)
 - whole function is quadratic ($O(N^2)$)

Rule of thumb 4

- If there is are multiple recursive calls per pass through the function (*i.e.* tree recursion)
- then depth of tree will determine complexity
- If depth is $O(N)$, then overall function will be $O(a^N)$ where a is the branching factor
 - e.g. two recursive calls per invocation $\rightarrow O(2^N)$
- If depth is $O(\log N)$, then overall function will be $O(N)$ (harder to see this)

Rule of thumb 5

- Functions which use iterative helpers must be examined carefully
- Must express complexity in terms of only the original function's parameters
- Example:
 - $f(x)$ calls $g(y)$
 - Time complexity of g is $O(y)$
 - You need to figure out how the size of y relates to the size of x
 - Final answer will be in terms of x
 - Usually pretty straightforward; just be aware of this

Examples

```
(define (f x y z)
  (if (< y 0)
      0
      (+ x z (f x (- y 1) z))))
```

- time complexity?
- $O(y)$

Examples

```
(define (f x y z)
  (if (< z 1)
      0
      (+ x z (f x y (/ z 2))))))
```

- time complexity?
- $O(\log z)$

Examples

```
(define (f x y)
  (if (= x 0) 0
      (+ y (f (- x 1) y))))
```

```
(define (g x y)
  (if (= x 0) 1
      (+ (f x y) (g (- x 1) y))))
```

- time complexity?
- $f: O(x)$ $g: O(x^2)$

Examples

```
(define (f x y)
  (if (< x 1) 0
      (+ y (f (/ x 2) y))))
```

```
(define (g x y)
  (if (< x 1) 1
      (+ (f x y) (g (/ x 2) y))))
```

- time complexity?
- f: $O(\log x)$ g: $O(\log^2 x)$

Examples

```
(define (f x y)
  (if (< x 1) 0
      (+ y (f (/ x 2) y))))
```

```
(define (g x y)
  (if (< x 1) 1
      (+ (g (/ x 2) y) (g (/ x 2) y))))
```

- time complexity?
- $f: O(\log x)$ $g: O(x)$

Examples

```
(define (f x y z)
  (define (f-iter cur_y xz total)
    (if (= cur_y y)
        total
        (f-iter (+ cur_y 1) xz (+ total xz))))
  (f-iter 0 (+ x z) 0))
```

- time complexity?
- $O(y)$
 - inner loop is linear in `cur_y`, `cur_y` is at most `y`

Iterative/Recursive Process

- How do computations unfold?
 - Not “does the function call itself”
- Linear recursive process
 - $O(N)$ operations
 - **Chain of deferred operations**
- Linear iterative process
 - Helper functions, internal “state” values
 - **No deferred operations**

Example

- Sum the results of function f over $[a, b]$
 - Step-size of 1
- Linear recursive process:

```
(define (sum f a b)
  (if (> a b)
      0
      (+ (f a) (sum f (+ a 1) b)) ))
```
- Chain of deferred additions

Example

- Turn this into a linear iterative process
 - Need some state
 - Adding up total was cause of deferred operations before...
 - Need a helper function now too

Example

- Linear iterative process:

```
(define (sum f a b)
```

```
  (define (iter a b total)
```

```
    (if (> a b)
```

```
        total
```

```
        (iter (+ a 1) b (+ total (f a))))
```

```
  (iter a b 0))
```

Update state values before
recursive call to iter

- Helper function keeps a running total
 - No more deferred operations

Recursion

- Design pattern:
- Is extra variable needed?
 - if so, need helper function
- Identify base cases!
 - that you can solve immediately
- Write an **if** or **cond** statement skeleton
- Put in the base cases
 - should require almost no work

Recursion

- Design pattern:
- Handle recursive cases:
 - What variable should be changing between recursive calls to the function?
 - How should it change?
 - *e.g.* down by 1, up by 1, down by 50%
 - How to combine results of recursive call with previous values to get result?

Example

- Find integer x in range $[\text{start}, \text{stop}]$ that maximizes $f(x)$

(define (largest-result start stop)
...)

(define (f x) ...) ; given to us

Example

- Any extra variables needed?
- The x corresponding to largest result
 - Could also pass largest result itself, but not strictly required

```
(define (largest-result start stop)
  (define (iter largest-x current-x)
    ...)
  (iter start start))
```

Example

- Base cases?

```
(define (largest-result start stop)
  (define (iter largest-x current-x)
    (cond ((> current-x stop) largest-x)
          ...))
  (iter start start))
```

Example

- Recursive case: what changes and how?

```
(define (iter largest-x current-x)
  (cond ((> current-x stop) largest-x)
        (<???> (iter <???> (+ current-x 1))))
  ...))
```

Example

- Recursive case: what to do with current value?

```
(define (iter largest-x current-x)
  (cond ((> current-x stop) largest-x)
        ((> (f current-x) (f largest-x))
         (iter current-x (+ current-x 1)))
        ...))
```

Example

- Yes, this is not optimal... but it's simple. ☺

```
(define (iter largest-x current-x)
  (cond ((> current-x stop) largest-x)
        ((> (f current-x) (f largest-x))
         (iter current-x (+ current-x 1)))
        ...))
```

- (Just add another state argument to iter...)

Example

- Recursive case: other cases

```
(define (iter largest-x current-x)
  (cond ((> current-x stop) largest-x)
        ((> (f current-x) (f largest-x))
         (iter current-x (+ current-x 1)))
        (else (iter largest-x (+ current-x 1)))))
```

Abstraction

- Extract general function from more specific function
- If specific function uses another function **f** in its body, then can...
 - make **f** an argument to a more general function

Abstraction

```
(define (largest-result start stop)
  (define (iter largest-x current-x)
    (cond ((> current-x stop) largest-x)
          ((> (f current-x) (f largest-x))
           (iter current-x (+ current-x 1)))
          (else (iter largest-x (+ current-x 1)))))
  (iter start start))
```

- How to abstract this to more general function?

Abstraction

```
(define (largest-result start stop)
  (define (iter largest-x current-x)
    (cond ((> current-x stop) largest-x)
          ((> (f current-x) (f largest-x))
           (iter current-x (+ current-x 1)))
          (else (iter largest-x (+ current-x 1)))))
  (iter start start))
```

- What functions used in `largest-result`?
- `>`, `+`, `f`

Abstraction

```
(define (optimum start stop binary-pred?)  
  (define (iter optimal-x current-x)  
    (cond ((> current-x stop) optimal-x)  
          ((binary-pred? current-x optimal-x)  
           (iter current-x (+ current-x 1)))  
          (else (iter optimal-x (+ current-x 1)))))  
  (iter start start))
```

- Now can easily define:

```
(define (largest-result start stop)  
  (optimum start stop  
    (lambda (current best) (> (f current) (f best)))))
```

More abstraction

```
(define (optimum start stop binary-pred? next)
  (define (iter optimal-x current-x)
    (cond ((> current-x stop) optimal-x)
          ((binary-pred? current-x optimal-x)
           (iter current-x (next current-x)))
          (else (iter optimal-x (next current-x)))))
  (iter start start))
```

- Now can easily define:

```
(define (largest-result start stop)
  (optimum start stop
    (lambda (current best) (> (f current) (f best)))
    (lambda (x) (+ x 1)) ))
```

Even more abstraction

```
(define (optimum start stop bin-pred? next cmp init)
  (define (iter optimal-x current-x)
    (cond ((cmp current-x stop) largest-x)
          ((bin-pred? current-x optimal-x)
           (iter current-x (next current-x)))
          (else (iter optimal-x (next current-x)))))
  (iter init start))
```

- Now can easily define:

```
(define (largest-result start stop)
  (optimum start stop
    (lambda (current best) (> (f current) (f best)))
    (lambda (x) (+ x 1)) > start))
```

Even more abstraction

- Took very specific function, made into very general function
- General function will have much wider applicability
- Redefining the specific function is very easy
- Often use **lambda** expressions as arguments to define behavior

Functions returning functions

- You should be very comfortable with turning *e.g.*

```
(define (f a b c) (+ a b c))
```

- into

```
(define (f2 a)
```

```
  (lambda (b c) (+ a b c)))
```

```
(f 10 20 30) → 60
```

```
((f2 10) 20 30) → 60
```