Programming Abstractions

Lecture 35: Final Exam Review

Stephen Checkoway

Announcements

No class on Friday; extra office hours via Zoom instead

Office hours on Friday 13:30–14:30 and 15:30–16:30

Evaluations are available

- If 90% of the class fills out an evaluation, everybody gets extra credit
- Currently (2022-10-12 at 14:00) at 48% 😞

Exam Format

Combination of problems (some or all of)

- True/false or multiple choice
- Short answer
- Code to write in DrRacket and uploaded to Blackboard

Exam will be available at 11:00 EST on Saturday, January 22, 2022

Your solutions are due by 11:00 EST on Sunday, January 23, 2022

Late exams are *not* allowed by College policy (sorry, it's out of my control)

Final exam time

During the scheduled final exam time (09:00–11:00 EST), I will be in the class's Zoom meeting, feel free to hang out in there

If you have a question, send me a private chat either with the question itself or just say "I have a question" and I'll bring you into a breakout room and you can ask your question privately there

However, it's better to ask private questions on Piazza early instead since the scheduled time is the last two hours.

Possible question topics

Anything we have covered in the course from day 1 until today, including

- Basic Scheme/Racket procedures and special forms
  - cons, first, rest, list, append, empty?, filter, and all the others
  - define, lambda, if, cond, let, let*, letrec, etc.
- map, foldl, foldr
- apply
- Recursion
  - "Normal" recursion
  - Tail recursion
  - "Accumulator-passing style"
  - Continuation-passing style
- Closures
  - What they are, how we create them, and how we use them

Possible question topics

- Backtracking
  - Single solution
  - All solutions
- Environments
  - How and when they're created
- Lexical vs. dynamic binding
- Parameter passing mechanisms
  - Pass by value
  - Pass by reference
  - Pass by name

Possible question topics

- Interpreter project
  - Creating new structs
  - Implementation of the environment
  - Parsing expressions
  - Evaluating parse trees
  - Implementing new features/special forms
- Basic runtimes of procedures O(n), O(n log n), $O(n^2)$, etc.
- Macros
  - How they work
  - How to write new ones
- Promises
- Streams

- Mutation and boxes
  - `set!` vs. `set-box!`
  - `unbox`

What is the run time of `(range1 n)` given the following definition?

```
(define (range1 n)
  (cond [(zero? n) empty]
        [else (append (range1 (sub1 n)) (list n))]))
```

A. O(log n)

B. O(n)

C. O(n log n)

D. $O(n^2)$

E. $O(2^n)$

What is the run time of `(range2 n)` given the following definition?

```
(define (range2 n)
  (letrec ([f (λ (m)
                (cond [(= m n) empty]
                      [else (cons m (f (add1 m)))]))])
    (f 0)))
```

A. O(log n)

B. O(n)

C. O(n log n)

D. $O(n^2)$

E. $O(2^n)$

Practice problems

Implement (range n) using accumulator-passing style

Implement (range n) using continuation-passing style

When you implemented MiniScheme, why was `(if …)` implemented as a special case in the parser and interpreter rather than implemented as a primitive procedure?

What fails if you try to implement it as a primitive procedure?

A. Got it!

Imagine you had implemented support for macros in MiniScheme, could `(if …)` be implemented as a built-in macro (similar to a primitive procedure except it's a macro rather than a procedure) rather than as a special case in the parser/interpreter?

Assume macros in MiniScheme would work similarly to macros in Racket where patterns are matched against expressions and the output of the macro is valid MiniScheme code.

A. Yes, it would be easy

B. Yes, it would be difficult

C. No, MiniScheme could never support macros

D. No, some conditional is needed

MiniScheme (and most programming languages including Racket) is a "strict" language. This means arguments to called functions must be evaluated before the body of the function is executed.

In contrast, a "lazy" language defers evaluation of arguments until the arguments are used.

If we made MiniScheme a lazy language by deferring evaluation of expressions until evaluating primitive procedures, could (if …) be implemented as a primitive procedure?

A. Yes. When evaluating `(if then-expr else-expr)`, only one of `then-expr` or `else-expr` would ever need to be evaluated

B. No. Like with macros, MiniScheme would still need a special case for conditionals

Course evals

Remember to fill out course evals!