CSCI 275: Programming Abstractions

Lecture 30: Exam 2 Review

Fall 2024

Stephen Checkoway
Slides from Molly Q Feldman

Plan for Today

- Brief overview of Exam 2 logistics
- Many (!) Clicker questions for Review
- Some open ended Review Questions
- If there’s time, opportunity to ask questions

Details of the Exam

Open book/notes/Racket – will specify specifically in the assignment

Programming problems & conceptual questions related to programming

Two goals:

- Show you how much you know
- Show me your “intuitive” response to different topics

Logistics

Exam will be available for 24 hours (all day Wednesday)

The exam will be released/submitted via GitHub Classroom

During Wednesday’s class, I will be in my office, feel free to stop by to ask any questions about the exam

SAME deal as Exam 1

In Scope Topics

- Higher-order functions
- MiniScheme
  - Implementation
  - Design
  - Theory
- Scoping
- ***Streams***
- Parameter Passing

Practice Clicker Questions

Consider a new structure to represent a point in 2D:

```
(struct point (x y) #:transparent)
```

If p is a point created via the `point` constructor, how would we create a new point whose fields are the absolute value of the fields in `p`? (The function `(abs x)` returns the absolute value of `x`.)

A. `(map abs p)`

B. `(list* 'point (map abs (rest p)))`

C. `(struct point (abs (point-x p)) (abs (point-y p)))`

D. `(point (abs (point-x p)) (abs (point-y p)))`

E. More than one of the above (which?)

Which of the following, when `(stream->list (stream-take (PROCEDURE 1 2) 10))` is run, produces `'(1 2 1 2 1 2 1 2 1 2)`?

A.

```
(define (sheep a b)
  (stream-cons '(a b) (sheep a b)))
```

B.

```
(define (lamb a b)
  (stream-cons a
   (stream-cons b (lamb a b))))
```

C.

```
(define (ram a b)
  (stream-cons a b (ram a b)))
```

D. More than one of the above

E. None of the above

When parsing a `let` expression, which pieces of information does the parse tree need to store?

A. An extended environment mapping the symbols in the binding list to their values and the body expression

B. A list of binding symbols, list of parse trees for the binding expressions, and the parsed body expression

C. A list of binding symbols, a list of binding values, and the body expression

D. Any of A, B, or C work

E. Either B or C work, but not A

Let’s say we want to implement `let*` in MiniScheme. Which files would need to change?

```
(let* ([x 2]
        [y (+ x 4)]) y)
```

A. `env.rkt`, `parse.rkt`, `interp.rkt`

B. `interp.rkt` and `parse.rkt`

C. `parse.rkt` only

D. `interp.rkt` only

E. Some other combination!

Evaluating a lambda gives a closure. A closure in a language with *dynamic binding* needs to contain which information?

A. The list of parameters

B. The list of parameters and the parsed body

C. The list of parameters, the parsed body, and the environment in which the lambda was evaluated

D. The list of parameters, the parsed body, and the environment in which the closure is to be evaluated

What is the output of the following in Call by Value versus Call by Name?

```
; Always returns an even int when x is an int
(define (double x) (+ x x))

(let ([a 1])
   (double (begin (set! a (add1 a)) a)))
```

A. CBV: 4
   CBN: 4

B. CBV: 3
   CBN: 3

C. CBV: 4
   CBN: 3

D. CBV: 4
   CBN: 5

E. CBV: 5
   CBN: 4

Adapted from Lyn Turbak

Additional Practice

For many primitive procedures, we can have a line like
`[(eq? op '+) (apply + args)]`
in `apply-primitive-op`.

Does `[(eq? op 'lt?) (apply < args)]`
work for our less than procedure?

A. It will work because `<` is Racket's less than

B. It won't work because `lt?` is Racket's less than

C. It won't work because `<` takes two arguments and `apply` allows any number of arguments

D. It won't work because `<` returns `#t` or `#f`

What is the value of the expression assuming lexical binding? What about dynamic binding?

```
(let* ([x 10]
       [f (lambda (z) (* x z))])
  (let ([x 20])
    (f x)))
```

A. Lexical: 100
   Dynamic: 100

B. Lexical: 100
   Dynamic: 200

C. Lexical: 200
   Dynamic: 100

D. Lexical: 200
   Dynamic: 200

E. Lexical: 200
   Dynamic: 400

Write a procedure `power` that, given `n`, returns a stream containing the powers of `n`.

For instance, if `n` = 2, we should get the stream (2,4,8,16,32...).

Why do we have multiple environments?

Why not just have a single environment where we update the bindings for each `let` expression or procedure call?