

Programming Abstractions

Lecture 27: Exam 2 Review

Stephen Checkoway

Exam Format

One 4-part programming problem (40 points)

- ▶ Write code in DrRacket, upload file

Several conceptual problems (60 points)

- ▶ Short answer or multiple choice
- ▶ Possibly short code snippets you have to write

1 extra credit programming problem (10 points)

- ▶ It's significantly more difficult than the other questions; do this last

Exam will be released at 00:01 EDT on Monday

Your solutions are due by 23:59 EDT on Monday

Class time

During Monday's class, I will be in my office, feel free to stop by to ask about the exam

Possible question topics

Programming language issues

- ▶ Backtracking
 - Single solution
 - All solutions
- ▶ Environments
- ▶ Lexical vs. dynamic binding
- ▶ Parameter passing mechanisms
 - Pass by value
 - Pass by reference
 - Pass by name
- ▶ Closures

Possible question topics

Interpreter project

- ▶ Datatypes for various constructs (literals, variables, if-then-else, let, applications)
- ▶ Environment implementation
- ▶ How specific expressions are parsed and evaluated
- ▶ What would happen if we did something differently

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?)

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 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

Recall that application expressions (`proc exp1 ... expn`) work by evaluating the `proc` expression and then each of the argument expressions in order before calling the procedure.

In a language without mutation (e.g., all of MiniSchemes A–E do not have mutation), it doesn't matter what order the expressions are evaluated in; the result will be the same. What about a language that supports `set!`, does order matter then? Why or why not?

- A. Yes it matters (what's an example?)
- B. No it doesn't matter (why not?)
- C. It depends (in which cases does it matter)

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

```
(let* ([x 10]
       [f (λ (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

Consider this Python-like code snippet

```
def foo(x):  
    x += 10  
    return x + 1  
  
def main():  
    y = 1  
    z = foo(y)  
    print(y+z)
```

What is printed by main assuming pass-by-value? Assuming pass-by-reference?

- A. Value: 13
Reference: 13
- B. Value: 13
Reference: 23

- C. Value: 13
Reference: 24
- D. Value: 23
Reference: 24

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?

A latin square is an $n \times n$ array filled with n different symbols, each occurring exactly once in each row and in each column. E.g.,

A	B	C
C	A	B
B	C	A

is a 3×3 latin square.

An $n \times n$ latin square can be found using backtracking. What should the `feasible` procedure do to check if the next cell in a partial solution can (potentially) be set to the next value?

In other words, given a partial solution, e.g.,

A	B	C
C		

and a symbol $s \in \{A, B, C\}$, how would you check if the symbol s could be assigned to the next open cell in the square (the center cell in this example)?

What are the lexical addresses (lexical-depth, index) of each of each use of the highlighted variables?

```
(define foo
  (λ (x lst)
    (foldr (λ (y acc)
            (if (equal? y x)
                (append lst acc)
                acc))
          empty
          lst)))
```

Different variables can have the same lexical address. Why is that not a problem?