CS 242 Midterm Review

Paul Twohey
twohey@cs

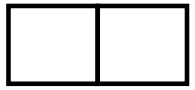
General Advice

- look at the old midterm available on the handouts page
  - ignore denotational semantics
- look over problems in the book
- relax

Topics

- ML
- lambda calculus
- higher order functions
- exceptions
- continuations
- parameter passing

LISP / Scheme

- composed of s-expressions
- basic unit is the cons cell 
- function evaluation is eager, but some special forms are not
  - (if (test) (true-part) (else-part))
- includes an evaluator
  - eval and quote

LISP / Scheme

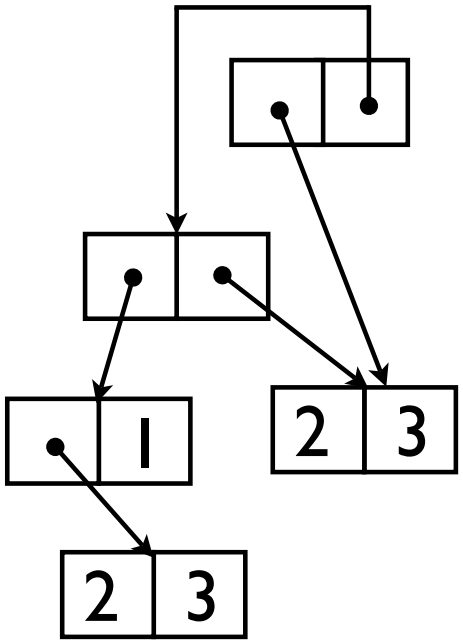
- Question: what does the code do?
- Answer: it depends

```
(eval `(define foo 3))
(print foo)
```

LISP / Scheme

- draw the box and pointer diagram for the following code

```
(((lambda (x)
    (lambda (y)
      (cons x (cons y x))))
  (cons 2 3))
 (cons (cons 2 3) 1))
```



ML

- convert the following ML code into lambda calculus

```
let
        val x = 6
in
        x+4
end
```

$$(\lambda x.\ x + 4)\ 6$$

Lambda Calculus

- a b c = (a b) c
- □ reduction is the basic unit of computation□
- computation stops when we reach normal form
  - Q: do all lambda expressions have a normal form?
- □ reduction renames bound variables

□ reduction

λx. ((λf. λx. f (f x)) (λy. y + x))
λz. ((λf. λx. f (f x)) (λy. y + z))
λz. λx. (λy . y + z) ((λy. y + z) x)
λz. λx. (λy. y + z) (x + z)
λz. λx. x + z + z

Higher Order Functions

- functional languages like ML and Scheme allow programmers to pass and return functions like any other values
- provides a powerful programming tool

Higher Order Functions

- Provide a procedure that takes as input a procedure f of one argument and an integer n and returns a procedure that applies f to it's argument n times

```
(define (repeat f n)
  (lambda (x)
    (if (eq? n 1)
        (f x)
        ((repeat f (- n 1)) (f x)))))
```

Exceptions and Continuations

- exceptions provide non-local flow control
  - mainly used to handle *exceptional* conditions
- continuations are a mechanism to explicitly represent where a value is returned to
- Traditionally `(define (f x) (+ x 1))`
- Continuations

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

Parameter Passing

- Two main styles of parameter passing
- pass by value - the value of the parameter is stored in the location allocated for the function parameters
- pass by reference - a reference to the value of the parameter is stored in the location allocated for the function parameters

Parameter Passing

- pass by value

```
struct foo { int i; char c; };
void bar( struct foo f ) {
    f.i++;
}
```

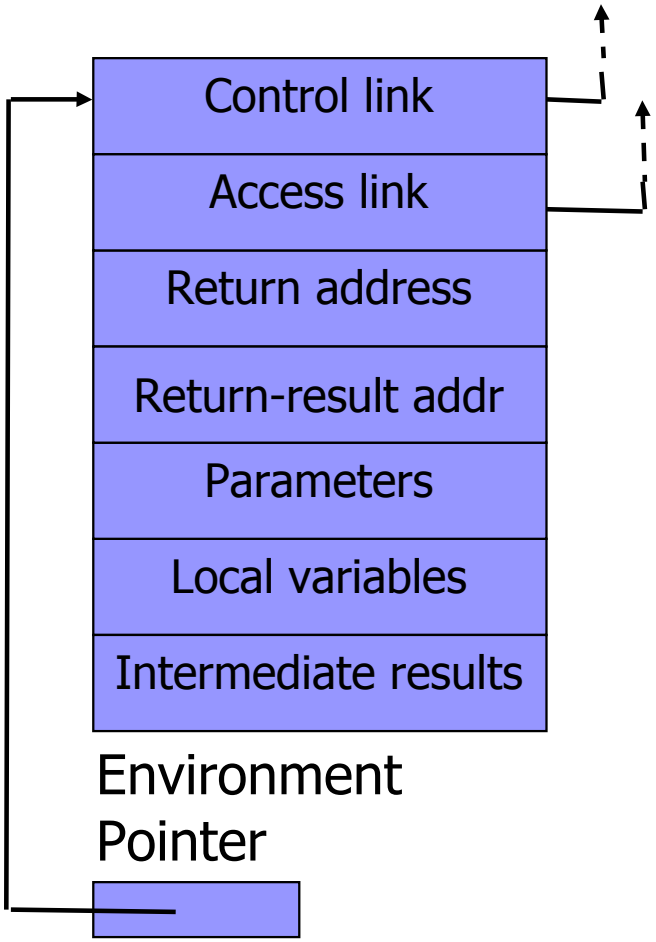
- pass by reference

```
void baz( struct foo &f ) {
    f.i++;
}
```

Activation Records

- Need a way to find variables, return addresses, and exception handlers
- activation records are the answer
- obey a stack discipline if functions are not returned from functions
  - Q: why?
  - A: closures

Activation Record for Static Scope



Control link

Link to previous (calling) block

Access link

Link to record of enclosing block

Return address

Next instruction after return

Return-result address

Place to put return value

Parameters

Set when function called

Local variables

Intermediate results

Other topics

- tail recursion
- lazy vs eager evaluation
- parallel vs sequential evaluation
- be prepared to put ideas together