CS 242

Midterm Review

John Mitchell

Midterm

- Thurs Nov 7, 7-9PM *Terman Aud* Closed book
- Class schedule
  - Thurs Oct 31, Tues Nov 5 – topics not on midterm
  - Thurs Nov 7 – optional review during lecture time
- Homework and Sample Midterm
  - Homework due Tues Nov 5 as usual; no late HW
  - HW 5 solutions available Nov 5
  - Sample exam on web now – previous year's handouts
  - Sample exam solutions on web Tues Nov 5

Topics

- Lisp, 1960
- Fundamentals
  - lambda calculus
  - denotational semantics
  - functional prog
- ML and type systems
- Block structure and activation records
- Exceptions and continuations
- Modularity
- OO concepts
  - encapsulation
  - dynamic lookup
  - subtyping
  - inheritance
- Simula and Smalltalk
- C++
- Java
- Concurrency

Lisp Summary

- Successful language
  - Symbolic computation, experimental programming
- Specific language ideas
  - Expression-oriented: functions and recursion
  - Lists as basic data structures
  - Programs as data, with universal function `eval`
  - Stack implementation of recursion via "public pushdown list"
  - Introduced garbage collection

Differentiation in Lisp

```
(define diff  (lambda (y x)
   (cond (
     (atom y) (if (eq x y) 1 0))  ;; y is a var or constant
    (                             ;; expression y is sum
     (eq (car y) '+)   ;; (A + B)' = A' + B'
     (cons '+ (maplist (cdr y)
                 (lambda (z) (diff (car z) x)))))
    (                             ;; expression is product
     (eq (car y) '*)   ;; (A * B)' = A'B'C + B'A
     (cons  '+  (maplist (cdr y)
                 (lambda (z)  (cons '*  (maplist  ...
    ))))))))))))
```

Fundamentals

- Grammars, parsing
- Lambda calculus
- Denotational semantics
- Functional vs. Imperative Programming
  - Why don't we use functional programming?
  - Is implicit parallelism a good idea?
  - Is implicit *anything* a good idea?

Declarations and reduction

```
function f(x)
    return x+2
end;
f(5);
```

block body declared function

$(\lambda f.\ f(5))\ (\lambda x.\ x+2)$

$\rightarrow (\lambda x.\ x+2)\ (5)$

$\rightarrow 5+2$

$\rightarrow 7$ (if we add reduction rules for addition)

Initialize-before-use semantics

Values = {*OK*, *bad*})

State = { *error* } $\cup$ (Variables $\rightarrow$ {*init*, *uninit*})

E[[c]](s) = OK for c any symbol not a variable

E[[v]](s) = if s(v)= init then OK else bad

C[[x:= e]](s) = if E[[e]](s) = OK
then modify(s,x, init)
else error

C[[x:= 0; y:= x+1]](s) = state s' with s'(x)=s'(y)=init

C[[x:= y; ...]](s) = error if s(y) = uninit

ML

- Typed programming language
- Intended for interactive use
- Combination of Lisp and Algol-like features
  - Expression-oriented
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions
- General purpose non-C-like, non-OO language

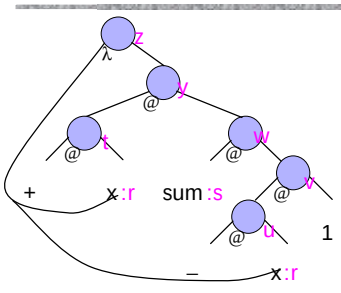
Use of ML in this course

- Survey a modern procedural language
- Discuss general programming languages issues
  - Types and type checking
    - General issues in static/dynamic typing
    - Type inference
    - Polymorphism and Generic Programming
  - Memory management
    - Static scope and block structure
    - Function activation records, higher-order functions
  - Control
    - Force and delay
    - Exceptions
    - Tail recursion and continuations

Types and Type Inference

- Type safety
  - Can use compile-time or run-time typing
  - Cannot have dangling pointers
- Type inference
  - Compute types from the way symbols are used
  - Know how algorithm works (for simple examples)
  - Know what an ML polymorphic type means
    - fun f(g,x) = g(g(x));
    > val it = fn : (t $\rightarrow$ t)*t $\rightarrow$ t
  - Polymorphism different from overloading

fun sum(x) = x + sum(x-1);



r → t = int → (int → int)
r → u = int → (int → int)
u = int → v
s = v → w
t = w → y
z = r → y
s = z

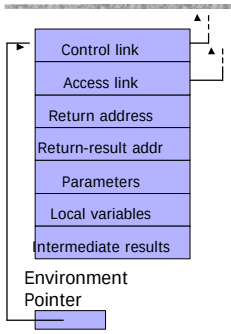
v = int
w = int
...

s = v → w = int → int

Block structure and storage mgmt

- Block-structured languages and stack storage
- In-line Blocks
  - activation records
  - storage for local, global variables
- First-order functions
  - parameter passing
  - tail recursion and iteration
- Higher-order functions
  - deviations from stack discipline
  - language expressiveness => implementation complexity

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

Complex nesting structure

```
function m(...) {
  int x=1;
  ...
  function n( ... ){
    function g(z) = x+z;
    ...
    { ...
      function f(y) {
        int x = y+1;
        return g(y*x) };
      ...
      f(3);  ... }
    ... n( ... ) ...}
... m(...)
```

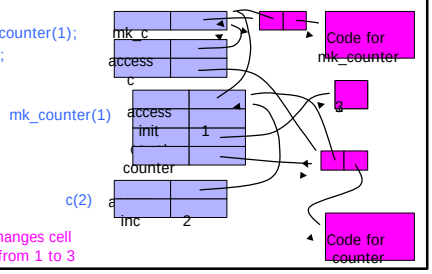
Simplify to

```
int x=1;
function g(z) = x+z;
function f(y) =
    { int x = y+1;
      return g(y*x) };
f(3);
```

Simplified code has same block nesting, if we follow convention that each declaration begins a new block.

Function Results and Closures

```
{int→int mk_counter (int init) {
    int count = init;   int counter(int inc) { return count+=inc;}
    }
 int→int c = mk_counter(1);
 print c(2) + c(2);
}
```



Summary of scope issues

- Block-structured lang uses stack of activ records
  - Activation records contain parameters, local vars, ...
  - Also pointers to enclosing scope
- Several different parameter passing mechanisms
- Tail calls may be optimized
- Function parameters/results require closures
  - Closure environment pointer used on function call
  - Stack deallocation may fail if function returned from call
  - Closures *not* needed if functions not in nested blocks

Control

- Structured Programming
  - Go to considered harmful
- Exceptions
  - "structured" jumps that may return a value
  - dynamic scoping of exception handler
- Continuation
  - Function representing the rest of the program
  - Generalized form of tail recursion

General Suggestions

- Review your notes and the reading assignments
- Look over homework
- Do the practice midterm
- Relax