CS 242

Scope, Function Calls and Storage Management

John Mitchell

Revised class schedule

- Friday Oct 17
  - No lecture; discussion section as usual
- Friday Oct 24
  - No section
- Monday Oct 27
  - Review section during class meeting time, Gates B01
- Wednesday Oct 29
  - No lecture
  - Evening exam: 7PM, Gates B01 and B03

Topics

- 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

Block-Structured Languages

- Nested blocks, local variables
  - Example

new variables declared in nested blocks

outer block

```
{ int x = 2;
    { int y = 3;
        x = y+2;
    }
}
```

inner block

local variable

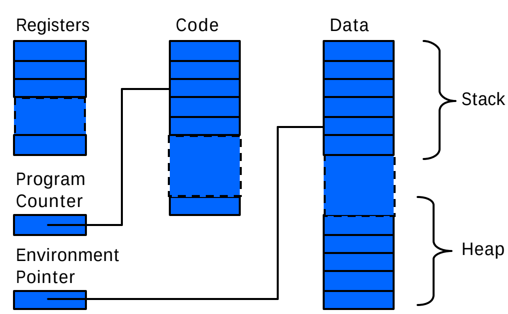
global variable

  - Storage management
    - Enter block: allocate space for variables
    - Exits block: some or all space may be deallocated

Examples

- Blocks in common languages
  - C { ... }
  - Algol begin ... end
  - ML let ... in ... end
- Two forms of blocks
  - In-line blocks
  - Blocks associated with functions or procedures
- Topic: block-based memory management, access to local variables, parameters,global vars

Simplified Machine Model



Interested in Memory Mgmt Only

- Registers, Code segment, Program counter
  - Ignore registers
  - Details of instruction set will not matter
- Data Segment
  - Stack contains data related to block entry/exit
  - Heap contains data of varying lifetime
  - Environment pointer points to current stack position
    - Block entry: add new activation record to stack
    - Block exit: remove most recent activation record

Some basic concepts

- Scope
  - Region of program text where declaration is visible
- Lifetime
  - Period of time when location is allocated to program

```
{ int x = ... ;
  { int y = ... ;
    { int x = ... ;
      ....
    };
  };
};
```

- Inner declaration of x hides outer one.
- Called "hole in scope"
- Lifetime of outer x includes time when inner block is executed
- Lifetime ≠ scope
- Lines indicate "contour model" of scope.

In-line Blocks

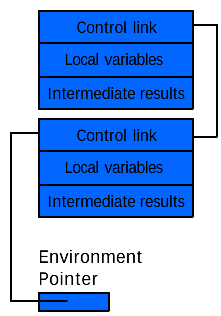
- Activation record
  - Data structure stored on run-time stack
  - Contains space for local variables
- Example

```
{ int x=0;
  int y=x+1;
  { int z=(x+y)*(x-y);
  };
};
```

Push record with space for x, y
Set values of x, y
Push record for inner block
Set value of z
Pop record for inner block
Pop record for outer block

May need space for variables and intermediate results like (x+y), (x-y)

Activation record for in-line block

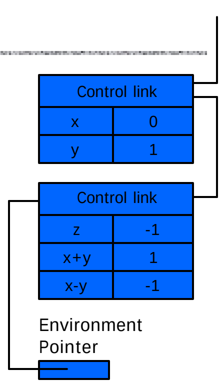


- Control link
  - pointer to previous record on stack
- Push record on stack:
  - Set new control link to point to old env ptr
  - Set env ptr to new record
- Pop record off stack
  - Follow control link of current record to reset environment pointer

Example

```
{ int x=0;
  int y=x+1;
  { int z=(x+y)*(x-y);
  };
};
```

Push record with space for x, y
Set values of x, y
Push record for inner block
Set value of z
Pop record for inner block
Pop record for outer block



Scoping rules

- Global and local variables
  - x, y are local to outer block
  - z is local to inner bock
  - x, y are global to inner block

```
{ int x=0;
  int y=x+1;
  { int z=(x+y)*(x-y);
  };
};
```

- Static scope
  - global refers to declaration in closest enclosing block
- Dynamic scope
  - global refers to most recent activation record

These are same until we consider function calls.

Functions and procedures

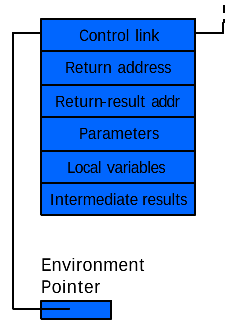
◆ Syntax of procedures (Algol) and functions (C)

```
procedure P (<pars>)          <type> function f(<pars>)
    begin                          {
        <local vars>                   <local vars>
        <proc body>                    <function body>
    end;                           };
```

◆ Activation record must include space for

- parameters
- return address
- return value
  (an intermediate result)
- location to put return value on function exit

Activation record for function

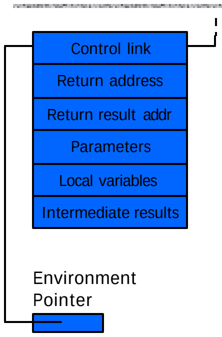


◆ Return address
- Location of code to execute on function return

◆ Return-result address
- Address in activation record of calling block to receive return address

◆ Parameters
- Locations to contain data from calling block

Example



◆ Function

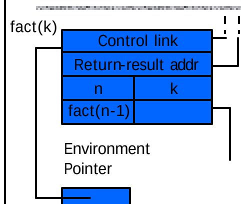
fact(n) = if n<= 1 then 1
else n * fact(n-1)

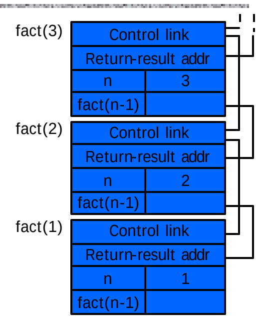
◆ Return result address
- location to put fact(n)

◆ Parameter
- set to value of n by calling sequence

◆ Intermediate result
- locations to contain value of fact(n-1)

Function call



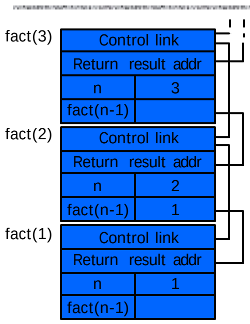


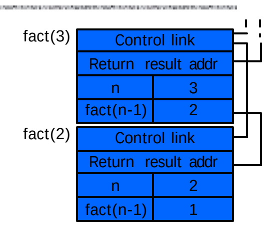
fact(n) = if n<= 1 then 1
else n * fact(n-1)

Return address omitted; would be ptr into code segment

Function return next slide →

Function return





fact(n) = if n<= 1 then 1
else n * fact(n-1)

Topics for first-order functions

◆ Parameter passing
- use ML reference cells to describe pass-by-value, pass-by-reference

◆ Access to global variables
- global variables are contained in an activation record higher “up” the stack

◆ Tail recursion
- an optimization for certain recursive functions

See this yourself: write factorial and run under debugger

ML imperative features (review)

- General terminology: L-values and R-values
  - Assignment y := x+3
    - Identifier on left refers to location, called its L-value
    - Identifier on right refers to contents, called R-value
- ML reference cells and assignment
  - Different types for location and contents

x : int	non-assignable integer value
y : int ref	location whose contents must be integer
!y	the contents
ref x	expression creating new cell initialized to x

  - ML form of assignment

y := x+3	place value of x+3 in location (cell) y
y := !y + 3	add 3 to contents of y and store in location y

Parameter passing

- Pass-by-reference
  - Caller places L-value (address) of actual parameter in activation record
  - Function can assign to variable that is passed
- Pass-by-value
  - Caller places R-value (contents) of actual parameter in activation record
  - Function cannot change value of caller's variable
  - Reduces aliasing (alias: two names refer to same loc)

Example

pseudo-code

```
function f (x) =
   { x := x+1; return x };
var y : int = 0;
print f(y)+y;
```

pass-by-ref

pass-by-value

Standard ML

```
fun f (x : int ref) =
    ( x := !x+1; !x );
y = ref 0 : int ref;
f(y) + !y;
```

```
fun f (z : int) =
    let x = ref z in
       x := !x+1; !x
    end;
y = ref 0 : int ref;
f(!y) + !y;
```

Access to global variables

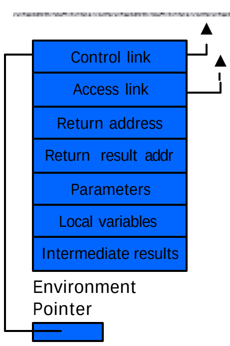
- Two possible scoping conventions
  - Static scope: refer to closest enclosing block
  - Dynamic scope: most recent activation record on stack
- Example

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

outer block	x	1
f(3)	y	3
	x	4
g(12)	z	12

Which x is used for expression x+z ?

Activation record for static scope



- Control link
  - Link to activation record of previous (calling) block
- Access link
  - Link to activation record of closest enclosing block in program text
- Difference
  - Control link depends on dynamic behavior of prog
  - Access link depends on static form of program text

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(...)
```

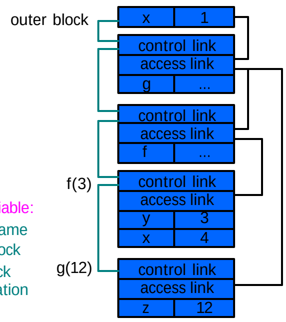
Simplify to

```
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.

Static scope with access links

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



Use access link to find global variable:
- Access link is always set to frame of closest enclosing lexical block
- For function body, this is block that contains function declaration

Tail recursion (first-order case)

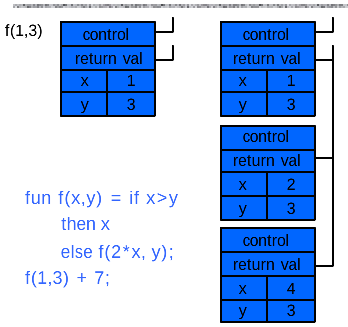
- Function g makes a *tail call* to function f if
  - Return value of function f is return value of g
- Example

tail call   not a tail call

```
fun g(x) = if x>0 then f(x) else f(x)*2
```

- Optimization
  - Can pop activation record on a tail call
  - Especially useful for recursive tail call
    - next activation record has exactly same form

Example Calculate least power of 2 greater than y



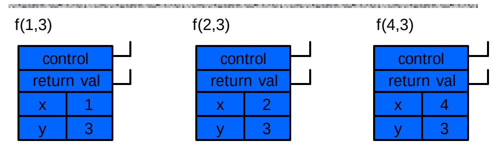

```
fun f(x,y) = if x>y
     then x
     else f(2*x, y);
f(1,3) + 7;
```

Optimization
- Set return value address to that of caller

Question
- Can we do the same with control link?

Optimization
- avoid return to caller

Tail recursion elimination

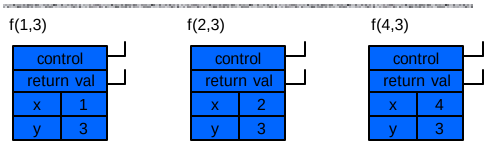



```
fun f(x,y) = if x>y
     then x
     else f(2*x, y);
f(1,3);
```

Optimization
- pop followed by push = reuse activation record in place

Conclusion
- Tail recursive function equiv to iterative loop

Tail recursion and iteration




```
fun f(x,y) = if x>y          test        fun g(y) = {
     then x                                   x := 1;
     else f(2*x, y);         loop body        while not(x>y) do
f(1,y);                                          x := 2*x;
                                              return x;
              initial value                 };
```

Higher-Order Functions

- Language features
  - Functions passed as arguments
  - Functions that return functions from nested blocks
  - Need to maintain environment of function
- Simpler case
  - Function passed as argument
  - Need pointer to activation record "higher up" in stack
- More complicated second case
  - Function returned as result of function call
  - Need to keep activation record of returning function

Example

Why this example here at this point in the lecture????

- Map function

```
fun map (f, nil) = nil  |   map(f, x::xs) = f(x) :: map(f,xs)
```

- Modify repeated elements in list

```
fun modify(l) =
    let  val c = ref (hd l)
         fun f(y) = ((if y = !c then c:=y+1 else c:=y);  !c)
    in
         (hd l) :: map(f, tl l)
    end;
modify [1,2,2,3,4]     =>     [1,2,3,4,5]
```

Exercise: pure functional version of modify

Pass function as argument

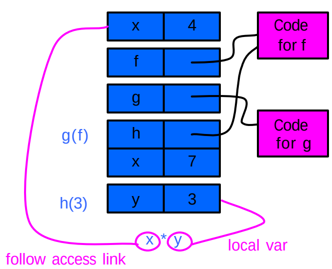
```
val x = 4;
 fun f(y) = x*y;
  fun g(h) = let
        val x=7
        in
        h(3) + x;
  g(f);
```

```
{ int x = 4;
  { int f(int y) {return x*y;}
    { int g(int→int h) {
          int x=7;
          return h(3) + x;
      }
      g(f);
} } }
```

There are two declarations of x
Which one is used for each occurrence of x?

Static Scope for Function Argument

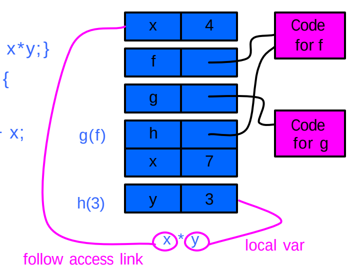
```
val x = 4;
 fun f(y) = x*y;
  fun g(h) =
     let
       val x=7
     in
       h(3) + x;
  g(f);
```



How is access link for h(3) set?

Static Scope for Function Argument

```
{ int x = 4;
  { int f(int y) {return x*y;}
    { int g(int→int h) {
          int x=7;
          return h(3) + x;
      }
      g(f);
} } }
```



How is access link for h(3) set?

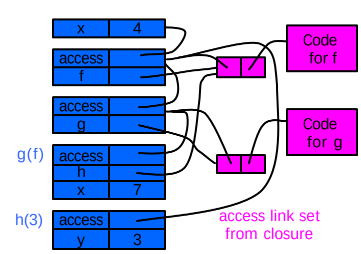
Closures

- Function value is pair *closure* = ⟨*env*, *code*⟩
- When a function represented by a closure is called,
  - Allocate activation record for call (as always)
  - Set the access link in the activation record using the environment pointer from the closure

Function Argument and Closures

Run-time stack with access links

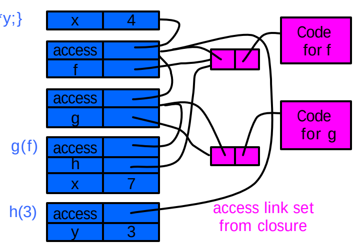
```
val x = 4;
  fun f(y) = x*y;
    fun g(h) =
       let
         val x=7
       in
         h(3) + x;
    g(f);
```



Function Argument and Closures

Run-time stack with access links

```
{ int x = 4;
  { int f(int y){return x*y;}
    { int g(int→int h) {
        int x=7;
        return h(3)+x;
      }
      g(f);
}}}
```



Summary: Function Arguments

- Use closure to maintain a pointer to the static environment of a function body
- When called, set access link from closure
- All access links point "up" in stack
  - May jump past activ records to find global vars
  - Still deallocate activ records using stack (lifo) order

Return Function as Result

- Language feature
  - Functions that return "new" functions
  - Need to maintain environment of function
- Example
  - fun compose(f,g) = (fn x => g(f x));
- Function "created" dynamically
  - expression with free variables
    - values are determined at run time
  - function value is closure = ⟨env, code⟩
  - code *not* compiled dynamically (in most languages)

Example: Return fctn with private state

```
fun mk_counter (init : int) =
   let  val count = ref init
        fun counter(inc:int) =
            (count := !count + inc; !count)
   in
        counter
   end;
val c = mk_counter(1);
c(2) + c(2);
```

- Function to "make counter" returns a closure
- How is correct value of count determined in c(2) ?

Example: Return fctn with private state

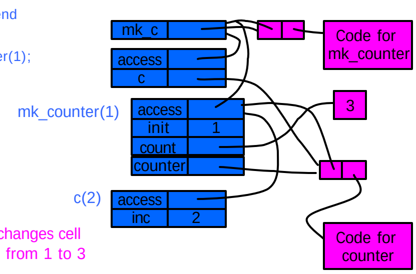
```
{int→int mk_counter (int init) {
      int count = init;
      int counter(int inc) { return count += inc;}
      return counter}
  int→int c  = mk_counter(1);
  print c(2) + c(2);
}
```

Function to "make counter" returns a closure

How is correct value of count determined in call c(2) ?

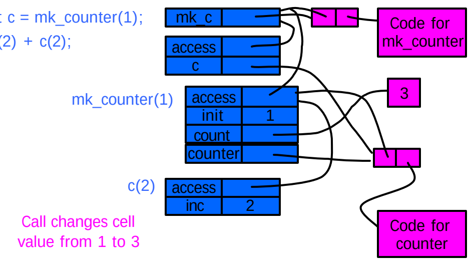
Function Results and Closures

```
fun mk_counter (init : int) =
  let val count = ref init
     fun counter(inc:int) = (count := !count + inc; !count)
     in  counter end
  end;
val c = mk_counter(1);
c(2) + c(2);
```



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: Return Function Results

- Use closure to maintain static environment
- May need to keep activation records after return
  - Stack (lifo) order fails!
- Possible "stack" implementation
  - Forget about explicit deallocation
  - Put activation records on heap
  - Invoke garbage collector as needed
  - Not as totally crazy as is sounds
    - May only need to search reachable data

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