CS560

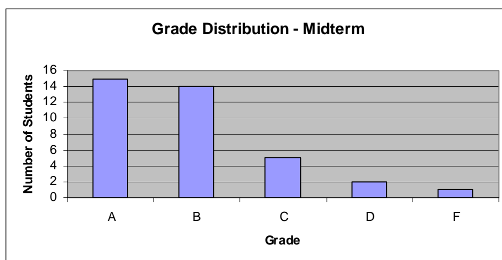
LECTURE 7
Feb 17, 2005

Slides: From John Mitchell, Concepts in Programming Languages, 2003.

Announcements

- ◆Homework 3 is posted; due week9
- ◆Midterm results
- ◆Any questions?

Midterm Results



- Ave: 75 points; Max: 89 points; Min: < 50

Overview of Class Tonight

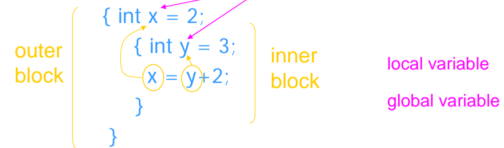
- ◆Scope
  - scope/lifetime
  - activation records
  - parameter passing
  - globals variables
- ◆Control

Topics

- Block-structured languages and stack storage
- In-line Blocks
  - activation records
  - storage for local, global variables
- First-order functions
  - parameter passing

Block-Structured Languages

- Nested blocks, local variables
  - Example

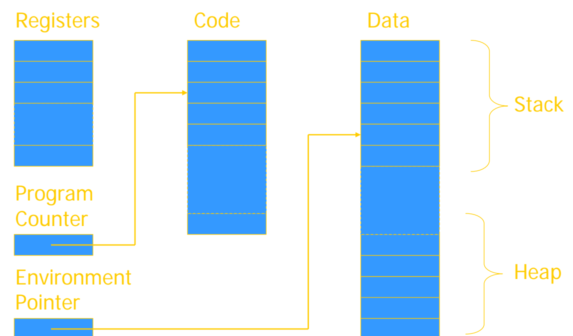


  - Storage management
    - Enter block: allocate space for variables
    - Exits block: some or all space may be deallocated

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

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)

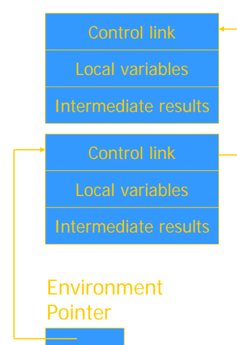
Some basic concepts

- 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.

Activation record for in-line block

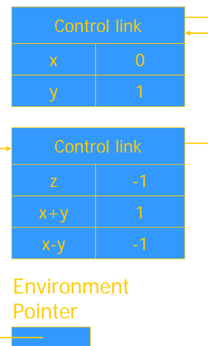


- Control link
  - pointer to previous record on stack
- Push record on stack:
  - Set new control link to point to old env ptr
  - Set env ptr to new record
- Pop record off stack
  - Follow control link of current record to reset environment pointer

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

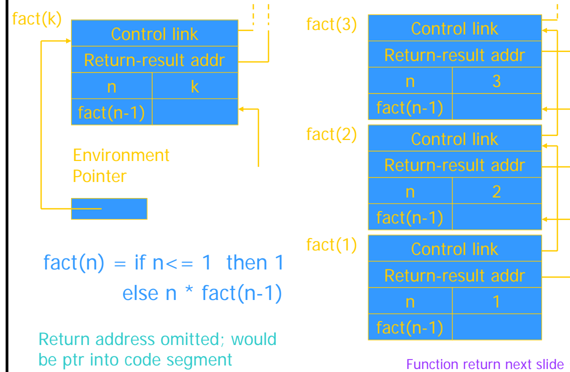
Activation record for function

- 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

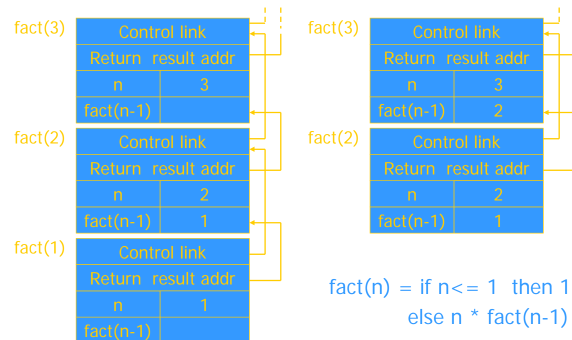


fact(n) = if n<= 1 then 1
else n * fact(n-1)

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

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 →

Standard ML

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

pass-by-value →

```
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);
```

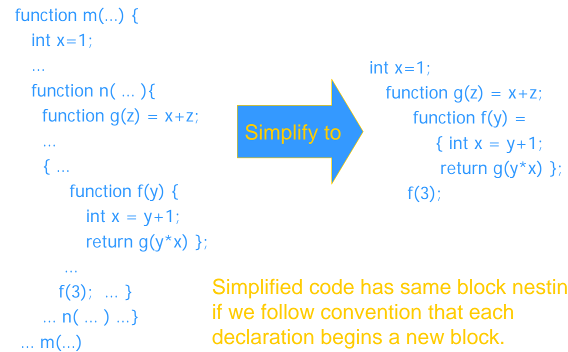
outer block	x	1
f(3)	y	3
	x	4
g(12)	z	12

Which x is used for expression x+z ?

Activation record for static scope

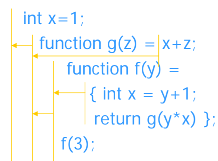
- Control link
  - Link to activation record of previous (calling) block
- Access link
  - Link to activation record of closest enclosing block in program text
- Difference
  - Control link depends on dynamic behavior of prog
  - Access link depends on static form of program text

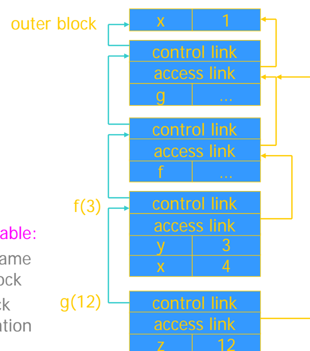
Complex nesting structure



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

Static scope with access links





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

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

◆ Map function

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

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);
```

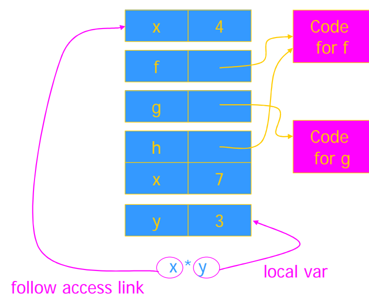
```
{ int x = 4;
  { int f(int y) {return x*y;}
    { int g(int→int h) {
          int x=7;
          return h(3) + x;
      }
      g(f);
} } }
```

There are two declarations of x

Which one is used for each occurrence of x?

Static Scope for Function Argument

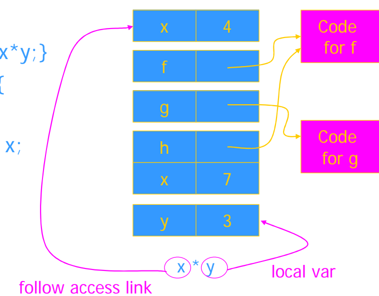
```
val x = 4;
 fun f(y) = x*y;
  fun g(h) =
   let
     val x=7
   in
     h(3) + x;
  g(f);
```



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?

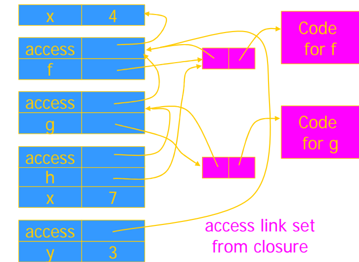
Closures

- 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

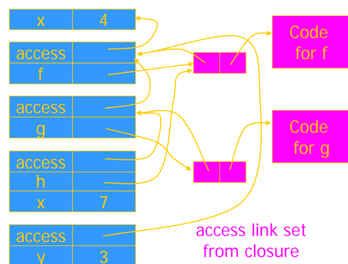
```
val x = 4;
  fun f(y) = x*y;
    fun g(h) =
        let
            val x=7
        in
            h(3) + x;
    g(f);
```



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) ?

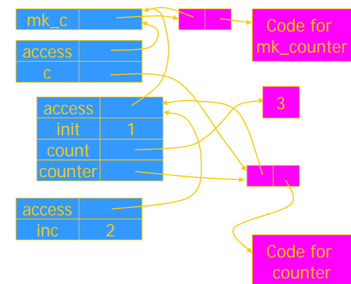
Example: Return fctn with private state

```
{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

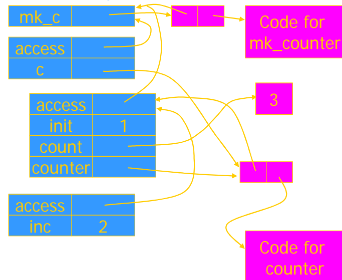
```
fun mk_counter (init : int) =
   let val count = ref init
      fun counter(inc:int) = (count := !count + inc; !count)
      in  counter end
   end;
val c = mk_counter(1);
c(2) + c(2);
```



Call changes cell value from 1 to 3

Function Results and Closures

```
{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);
}
```



Call changes cell value from 1 to 3

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
- 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

Exercise☺

```
val m = 3;
fun f(y) = (m+y) -2;
fun g(h) = let val m = 4 in h(m) end;
let val m = 5 in g(f) end;
```

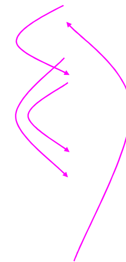
1. What are the types of m, f, g
2. What is the output?
3. Draw the ARs corresponding to the execution.

Topics → Chapter 8

- Structured Programming
  - Go to considered harmful
- Exceptions
  - "structured" jumps that may return a value
  - dynamic scoping of exception handler

Fortran Control Structure

```
10 IF (X .GT. 0.000001) GO TO 20
11 X = -X
   IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.00001) GO TO 30
   X = X-Y-Y
30 X = X+Y
   ...
50 CONTINUE
   X = A
   Y = B-A
   GO TO 11
   ...
```



Similar structure may occur in assembly code

GO TO Statement Considered Harmful --Dijkstra

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

Historical Debate

- Dijkstra, Go To Statement Considered Harmful
  - Letter to Editor, *C ACM*, March 1968
  - Google: paper title
- Knuth, Structured Prog. with go to Statements
  - You can use goto, but do so in structured way ...
- General questions
  - Do syntactic rules force good programming style?
  - Can they help?

Advance in Computer Science

- Standard constructs that structure jumps
  - if … then … else … end
  - while … do … end
  - for … { … }
  - case …
- Modern style
  - Group code in logical blocks
  - Avoid explicit jumps except for function return
  - Cannot jump *into* middle of block or function body

Exceptions: Structured Exit

- Terminate part of computation
  - Jump out of construct
  - Pass data as part of jump
  - Return to most recent site set up to handle exception
  - Unnecessary activation records may be deallocated
    - May need to free heap space, other resources
- Two main language constructs
  - Declaration to establish exception *handler*
  - Statement or expression to *raise* or *throw* exception

Often used for unusual or exceptional condition, but not necessarily

Questions