Activation Records

February 12, 2001
CS 132: Compiler Design

Scope and Extent

- The *scope* of a variable is the region of a program where that variable can be referred to.
  - Scope determines, e.g., "which variable named `x` is this expression referring to?"
- The *extent* of a data item (a.k.a. *object*) is the time interval where the object can be referred to.
  - Extent is an issue of memory allocation

Extent in Fortran 77

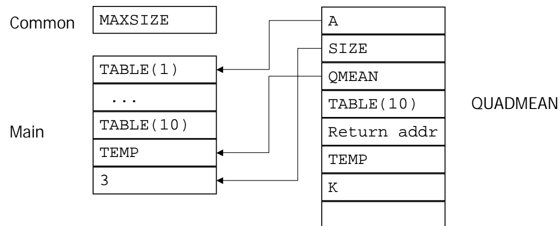
- Memory for *every* variable is statically allocated
  - Global variables (integers, arrays, etc.)
  - Subroutine variables
    - Formal parameters
    - Actual arguments
    - Local variables
    - Scratch space
    - Return addresses

Fortran Sample

```
      PROGRAM TEST
      COMMON MAXSIZE
      INTEGER MAXSIZE
      REAL TABLE(10), TEMP
      MAXSIZE = 10
      READ *, TABLE(1), TABLE(2), TABLE(3)
      CALL QUADMEAN(TABLE, 3, TEMP)
      PRINT *, TEMP
      END

      SUBROUTINE QUADMEAN(A,SIZE,QMEAN)
      COMMON MAXSIZE
      INTEGER MAXSIZE, SIZE
      REAL A(SIZE), QMEAN, TEMP
      INTEGER K
      TEMP = 0.0
      IF ((SIZE.GT.MAXSIZE).OR.(SIZE.LT.1)) GOTO 99
      DO 10 K = 1, SIZE
         TEMP = TEMP + A(K) * A(K)
10    CONTINUE
99    QMEAN = SQRT(TEMP/SIZE)
      RETURN
      END
```

Memory Layout



Activation Records

- The segment of memory used by a single procedure is called an *activation record.*
  - Contains arguments, return address, local data, etc.
- In Fortran, activation records are statically allocated.
  - Consequences?

Stack Frames

- Idea: allocate activation records on a stack
  - These records are then called *stack frames*
- Advantages
  - Permits multiple instances of a procedure to be simultaneously active (i.e., recursion)
  - Permits some dynamic memory allocation
- Consequences
  - Memory accesses must be sp-relative in general, rather than absolute

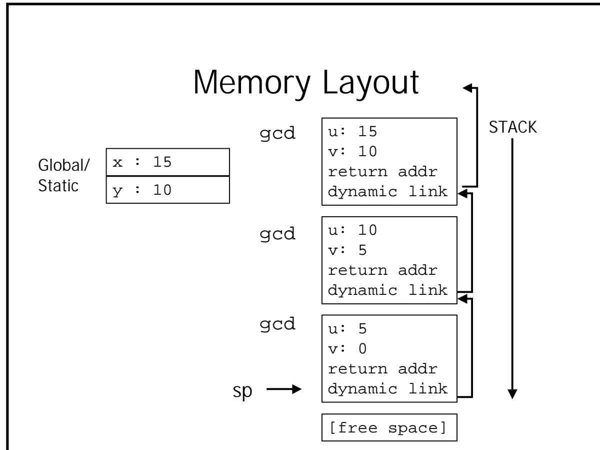
C Sample

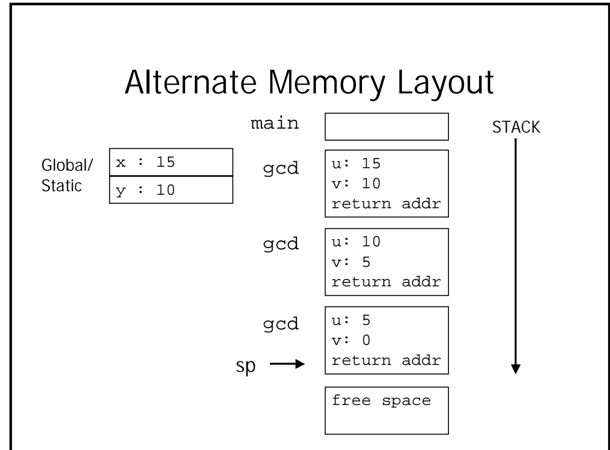
```
#include <stdio.h>

int x,y;

int gcd(int u, int v) {
  if (v == 0) return u;
  else return gcd(v, u%v);
}

int main() {
  scanf("%d%d",&x,&y);
  printf("%d\n", gcd(x,y));
  return 0;
}
```

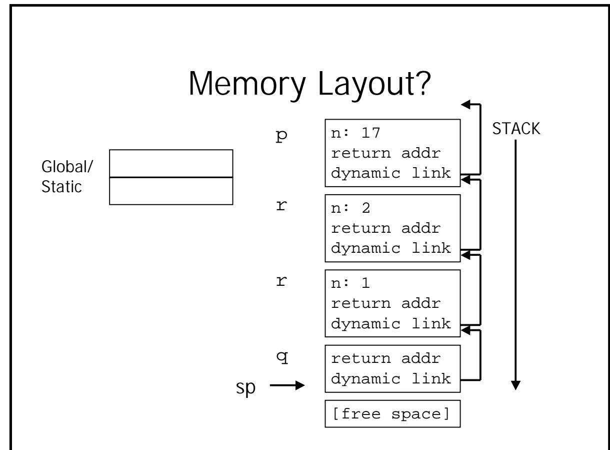


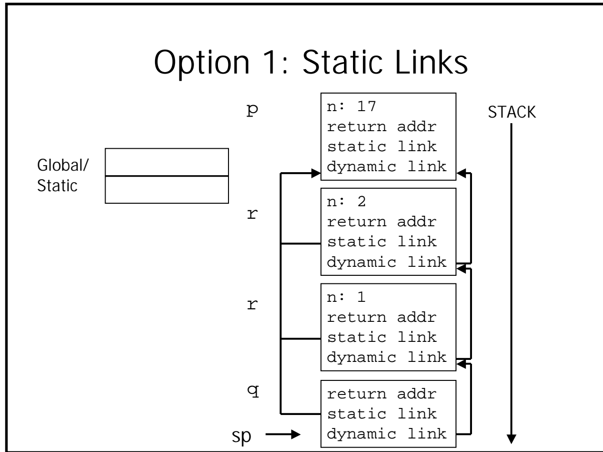


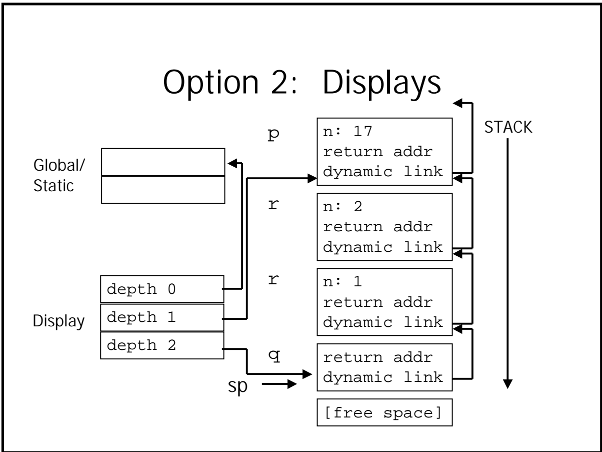
Pascal Sample

```
procedure p;
    var n: integer;
    procedure q;
    begin
       n := n+1;
    end;
    procedure r(n:integer);
    begin
       if n>1 then r(n-1);
       else q;
    end;

begin
  n := 17;
  r(2);
end;
```







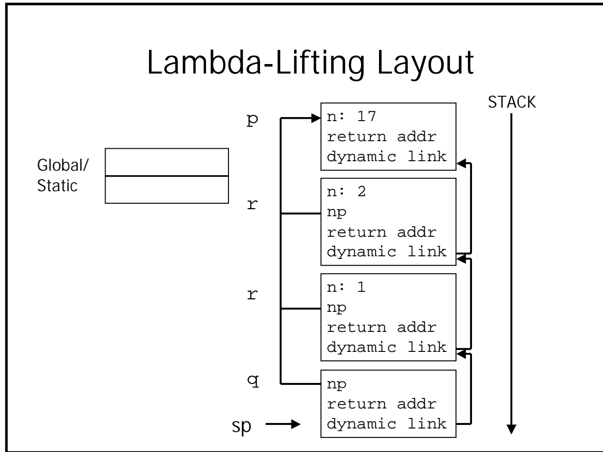
Option 3: Lambda Lifting

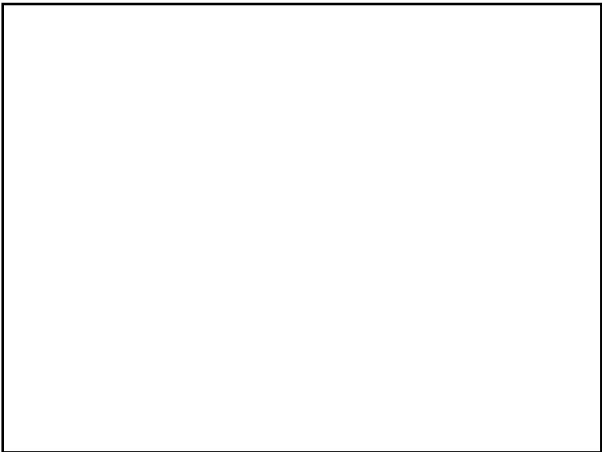
```
procedure p;
    var n: integer;
    procedure q(var np:integer);
    begin
       np := np+1;
    end;
    procedure r(n:integer, var np:integer);
    begin
       if n>1 then r(n-1,np)
       else q(np)
    end;

begin
  n := 17;
  r(2,n)
end;
```

Option 3: Lambda Lifting

```
procedure q(var np:integer);
begin
   np := np+1;
end;
procedure r(n:integer, var np:integer);
begin
   if n>1 then r(n-1,np) else q(np)
end;
procedure p;
  var n: integer;
begin
  n := 17;
  r(2,n)
end;
```





Procedures as *Parameters*

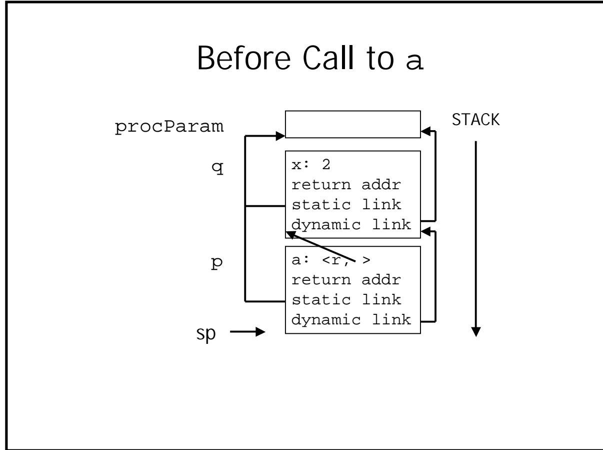
```
program procParam(output);
   procedure p(procedure a);
   begin
      a                          How does r find x?
   end;
   procedure q;
      var x :integer;

      procedure r;
         var z : int
      begin
         z := 1;
         writeln(x+z)
      end;
   begin
     x := 2;  p(r)
   end;
begin
   q;
end.
```

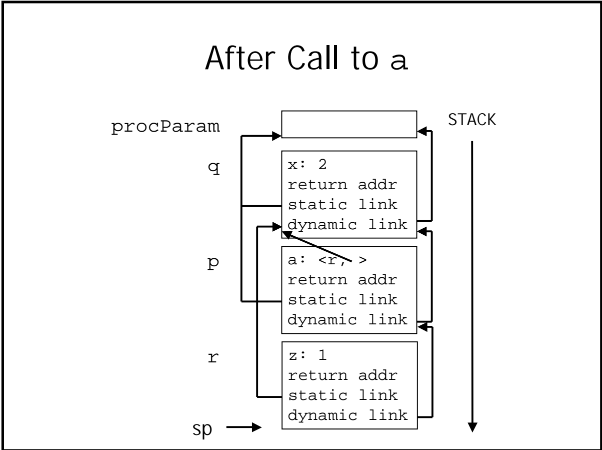
Idea: Closures

- When we pass around `r`, it's not enough to just pass the address of the code for `r`.
  - Needs to maintain enough information for this code to be able to find its *free* variables
- Solution: instead of passing pointers to code, pass around *closures*
  - Data structure containing code pointer and free variable information.
  - Simplest version: pair the code pointer and its *static* link
- Digression: Why isn't this necessary in C?

Before Call to a



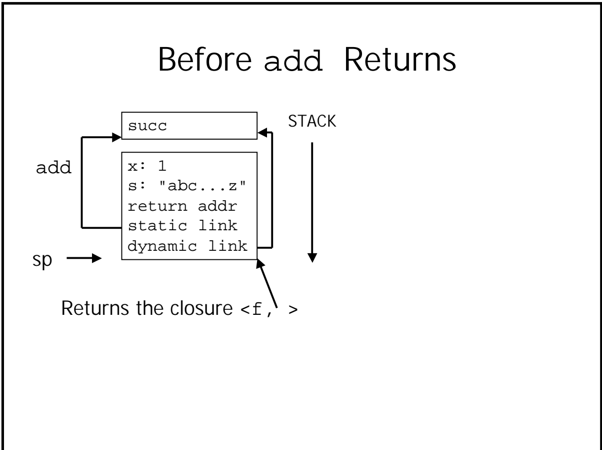
After Call to a

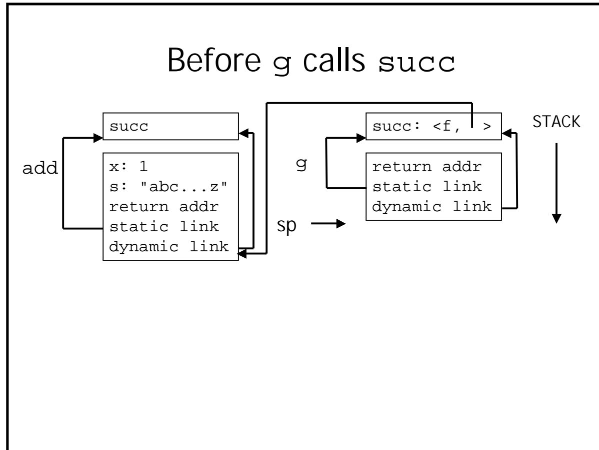


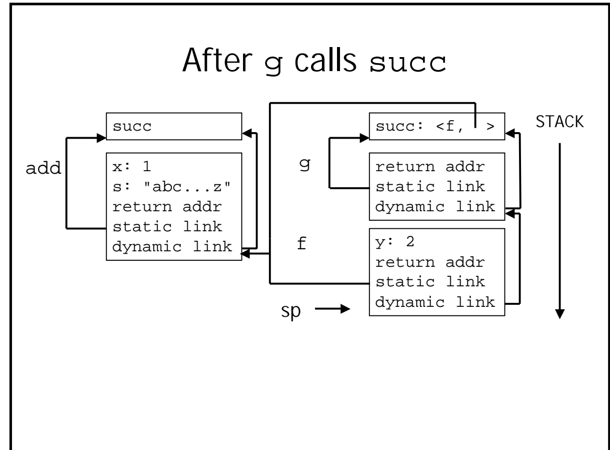
Procedures as *Results*

```
let
   fun add x =
     let
        val s = "abc...z"
        fun f y = x+y
     in
        f
     end
   val succ = add 1
   fun g() = succ 2
in
   g()
end
```

Before add Returns







Indefinite Extent

- The problem is that the value of `x` has *indefinite extent*
  - It needs to stay around even after `add` returns
  - This is probably the biggest difficulty in compiling "functional" languages.
- Simplest solution: don't use a stack
  - Allocate activation records on the heap (linked list)
  - Never pop activation records; can be garbage collected when no longer referenced.
  - Probably should be a bit cleverer; this leaks space