Variables and Definitions

February 19, 2001

CS 131: Programming Languages

Some SML Code

What Gets Printed?

```
val x = 0
fun f(y:int) = x * y
fun g(z:int) = let
                  val x = 1
               in
                  f(x + z)
               end
val _ = print (Int.toString (g 1))
```

Same Code in Emacs Lisp

What Gets Printed?

```
(defvar x 0)
(defun f (y) (* x y))
(defun g (z) (let ((x 1))
                (f (+ x z))))
(print (g 1))
```

Same Code in Perl (twice)

```
$x = 0;
sub f {
  local ($y) = @_;
  return ($x * $y);
}
sub g {
  local ($z) = @_;
  local $x = 4;
  return (f($x + $z));
}
print (g(1));
```

```
$x = 0;
sub f {
  local ($y) = @_;
  return ($x * $y);
}
sub g {
  local ($z) = @_;
  my $x = 4;
  return (f($x + $z));
}
print (g(1));
```

What's going on?

```
val x = 0
fun f(y) = x * y
```

Defines `f` to be the function which multiplies its argument by `0`

```
fun g(z) =
   let val x = 1
   in (f (x + z))
   end
```

```
(defvar x 0)
(defun f (y) (* x y))
```

Defines `f` to be the function which multiplies its argument by `x`

```
(defun g (z)
   (let ((x 1))
      (f (+ x z))))
```

More Precisely...

```
val x = 0
fun f(y) = x * y
```

`f` refers to the `x` in scope when `f` was *defined*. (Static Scope)

```
fun g(z) =
   let val x = 4
   in (f z)
   end
```

```
(defvar x 0)
(defun f (y) (* x y))
```

`f` refers to the most-recently defined `x` when `f` is *called*. (Dynamic Scope)

```
(defun g (z)
   (let ((x 4))
      (f z)))
```

Scoping in Languages

- Lexical
  - Fortran, Pascal, C, C++, Java, SML, Scheme, ...
- Dynamic
  - APL, Snobol, Original LISP, Emacs LISP, Perl 4, ...
- Both
  - Perl 5, Common LISP

Arguments for Lexical Scope

- Names of local variables and function arguments shouldn't matter
  - Avoids accidental clashes between separate pieces of code without having to choose obscure variable names
    - e.g., `verylongatomunlikelytobeusedbyprogrammer1`
- Easier to typecheck
  - Otherwise, what is the type of `fn(y:int)=>x*y` ?
- Easier to implement efficiently in compilers

Arguments for Dynamic Scope

- Easier to implement in an interpreter
- Customization of subroutines

```
(defvar base 10)
(defun print_int (n)
     (… print the number n in base base …))
(defun foo (y)
     (… do computation then call print_int …))

(let ((base 8)) (print_int 42))
(print_int 100)
(let ((base 2)) (foo 7))
(print_int 100)
```

Arguments for Dynamic Scoping

"Dynamic binding is especially useful for elements of the command dispatch table. For example, the RMAIL command for composing a reply to a message temporarily defines the character Control--Meta--Y to insert the text of the original message into the reply. The function which implements this command is always defined, but Control--Meta--Y does not call that function except while a reply is being edited. The reply command does this by dynamically binding the dispatch table entry for Control--Meta--Y and then calling the editor as a subroutine. When the recursive invocation of the editor returns, the text as edited by the user is sent as a reply"

Richard Stallman
EMACS: The Extensible, Customizable Display Editor

Implementing Dynamic Scope

- Easy to implement in an interpreter
  - As program is executing, maintain mapping from variable names to values
    - Update mapping whenever new variable is declared
    - Restore mapping when leaving scope of this variable
  - Whenever variable is encountered, look up its current definition

Interpreting Dynamic Scope

```
(defvar base 10)
(defun print_int (n)
     (… print the number n in base base …))


(let ((base 8)) (print_int 42))
(print_int 100)
```

When execution has reached this point, `base` is bound to `10` while `print_int` is bound to a function value.

Interpreting Dynamic Scope

```
(defvar base 10)
(defun print_int (n)
     (… print the number n in base base …))


(let ((base 8)) (print_int 42))
(print_int 100)
```

Here the environment has been updated to give `base` the value 8. Next the program calls `print_int`

Interpreting Dynamic Scope

```
(defvar base 10)
(defun print_int (n)
     (… print the number n in base base …))


(let ((base 8)) (print_int 42))
(print_int 100)
```

The function `print_int` looks up `base` in the environment and finds the value 8.

Interpreting Dynamic Scope

```
(defvar base 10)
(defun print_int (n)
     (… print the number n in base base …))


(let ((base 8)) (print_int 42))
(print_int 100)
```

After exiting the scope of the local variable `base`, the environment is restored to make `base` refer to the global variable, which has value **`10`**.

Interpreting Dynamic Scope

```
(defvar base 10)
(defun print_int (n)
     (… print the number n in base base …))


(let ((base 8)) (print_int 42))
(print_int 100)
```

Thus this call to `print_int` will look up the variable `base` and find the value `10`.

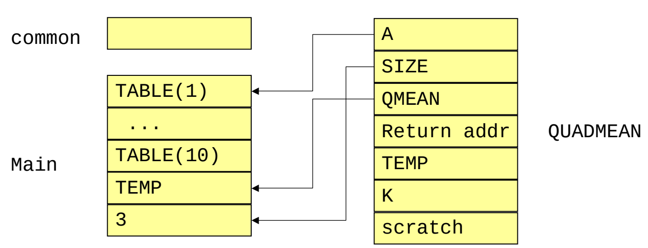
Compiling Static Scope

- Outline
  - Static storage (Fortran)
  - Unnested procedures (C)
  - Nested procedures (Pascal, Modula-2)
  - Procedures as arguments (Pascal, Modula-2)
  - Procedures as results (SML, Scheme)

Fortran Sample

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

      SUBROUTINE QUADMEAN(A,SIZE,QMEAN)
      INTEGER SIZE
      REAL A(SIZE), QMEAN, TEMP
      INTEGER K
      TEMP = 0.0
      IF ((SIZE.GT.10).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?

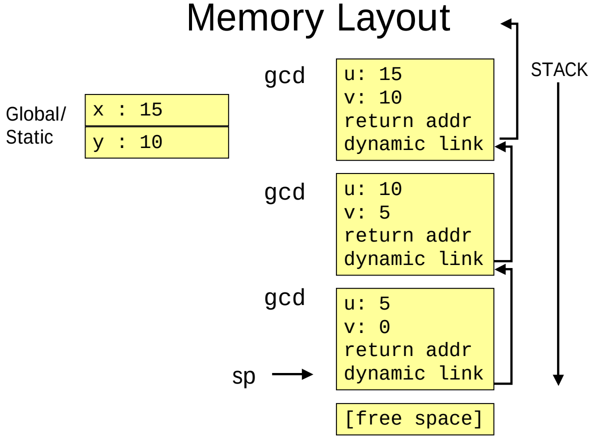
Stack Frames

- 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

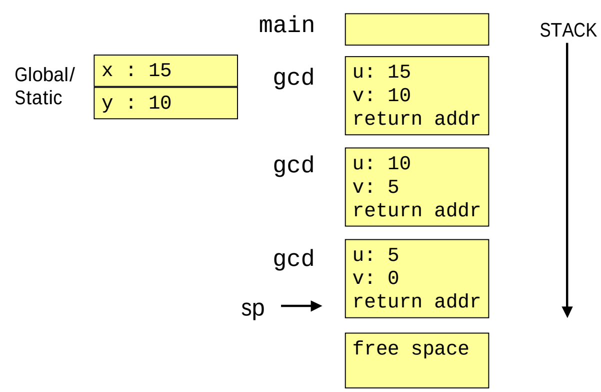
C Sample

```
#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;
}
```

Memory Layout



Alternate Memory Layout

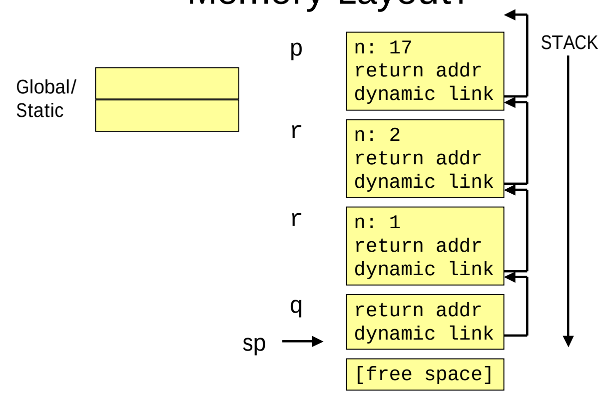


Nested Procedures

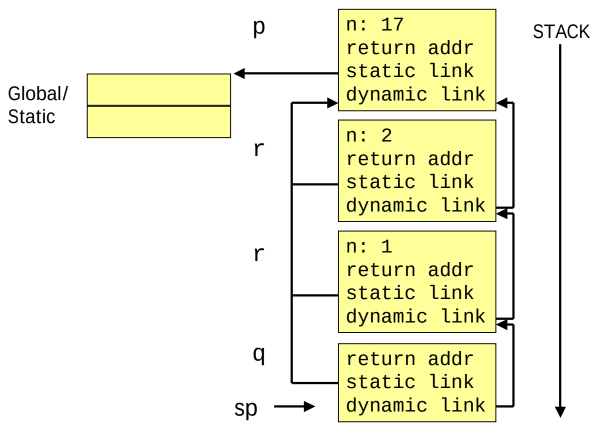
```
procedure p;
    var n: integer;

    procedure q;
    begin
       writeln(n)
    end;

    procedure r(n:integer);
    begin
       if n>1 then r(n-1)
       else q
    end;

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

```
fun p() =
   let
      val n = 17
      fun q() = print_int n
      fun r(n) =
         if n>1 then r(n-1)
         else q()
   in
      r(2)
   end
```

Memory Layout?



Idea: Static Links



Procedure Calls with Static Links

- Subroutine code:
  - Expects to be passed both its arguments and a pointer to its static link
  - Allocates its own stack frame and sets up its own dynamic link

Code Pointers

- C does not allow nested function definitions
- In C can pass/return function pointers (code addresses)

```
void qsort
     (void  *base,  size_t  nel, size_t  width,
      int  (*compar)(const void *, const void *));
```

- This is not a coincidence.

Procedures as Parameters

```
fun p() =
   let
      val n = 17
      fun q() = print_int n
      fun r(n) =
         if n>1 then r(n-1)
         else q()
   in
      r(2)
   end
```

```
fun call_arg(f) = f()
fun p() =
   let
      val n = 17
      fun q() = print_int n
      fun r(n) =
         if n>1 then r(n-1)
         else call_arg(q)
   in
      r(2)
   end
```

Procedures as Parameters

- Problem:
  - The `call_arg` function doesn't know what static link to give its argument `f`.
- Solution:
  - Cannot simply pass around a function pointer to `q`.
  - Pass around a *closure.*
    - Function pointer + information about its free variables.
    - e.g., <function pointer, static link>

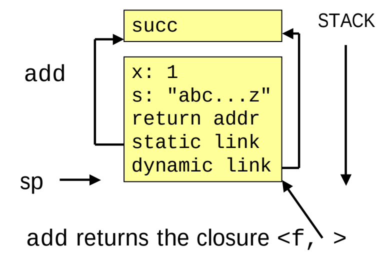
Returning Functions as Results

- Languages like Pascal and Algol permit nested function definitions and functions as arguments, but prohibit returning functions as results!
- Problem:
  - Closures implemented with static links
  - Stack frame pointed to by the static link may be popped before the closure is used!

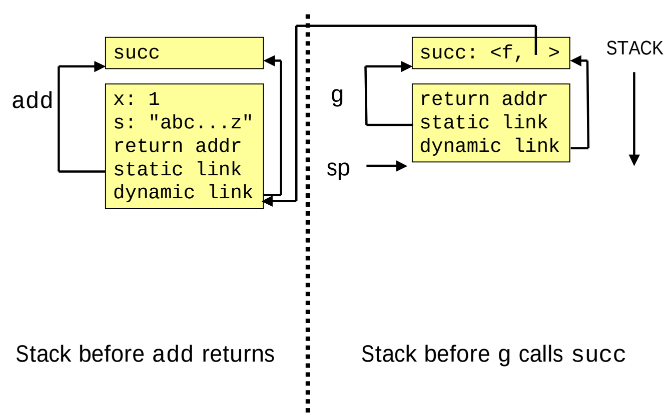
Procedures as *Results*

```
let
   fun add x =
     let
        fun f y = x+y
     in
        f
     end
   val succ = add 1
   fun g() = succ 2
in
   g()
end
```

Just Before add Returns



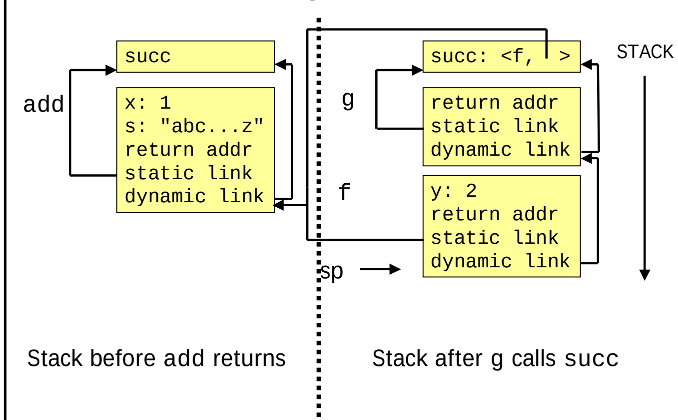
Before g calls succ



Stack before add returns

Stack before g calls succ

After g calls succ



Stack before add returns

Stack after g calls succ

Summary

- 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 on return
    - deallocate (GC) only when no longer referenced.
  - Probably should be a bit cleverer; this leaks space

Closures vs. Code Pointers

- In C can pass/return function pointers

```
void qsort
     (void  *base,  size_t  nel, size_t  width,
      int  (*compar)(const void *, const void *));
```

- But, cannot create functions at run-time
- No equivalent to

```
mergesort : ('a * 'a -> 'a) -> ('a list -> 'a list)
send      : channel -> (message -> unit)
```

Example: X-Windows

- The X Toolkit defines an event-driven model for X-windows
  - Can specify functions to be called when a particular event occurs (e.g, button pressed)
  - These functions called "upcalls" or "callbacks" because rather than making calls *to* the system, these functions are *called by* the system.
  - Implements closures *manually*
    - When specifying the callback, give both a code pointer and a piece of data to be passed to this function when it is called.

Sample Code

```
void PressMe(Widget w, XtPointer client_data,
             XtPointer call_data) {
   printf("%s\n", client_data);
}
int main() {
   Widget button1, button2;
   /* …initialization code for buttons here… */
   XtAddCallback(button1, XmNactivateCallback,
                 PressMe, "aha");
   XtAddCallback(button2, XmNactivateCallback,
                 PressMe, "oho");
}
```