Computer Science 131
Programming Languages

September 21, 2000
Static and Dynamic Scope
Consider This Code

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))
Consider This Code

```plaintext
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))
```

Prints 0
Same Code in Elisp

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

Prints 2
Same Code in Perl (twice)

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

```perl
$x = 0;
sub f {
    local ($y) = @_;  # Original: use local
    return ($x * $y);
}
sub g {
    local ($z) = @_;  # Original: use local
    my $x = 4;        # Use my instead of local
    return (f($x + $z));
}
print (g(1));
```
Same Code in Perl (twice)

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

Prints 0

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

Prints 2
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.

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

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

f refers to the x in scope whenever f is called.

(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

• Permits static typechecking
• Easier to implement efficiently in compilers
Arguments for Dynamic Scope

- 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)
"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

(let ((base 8)) (print_int 42))
(print_int 100)
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 Lexical Scope

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

• At compile-time:
  - Choose fixed address for every variable declaration in the program
  - Code refers directly to these addresses

• Used in early versions of Fortran

• Consequences:
Compiling Static Scope

• At compile-time:
  - Choose fixed address for every variable declaration in the program
  - Code refers directly to these addresses

• Used in early versions of Fortran

• Consequences:
  - No re-entrancy
  - No recursion, limits on concurrency
Compiling Unnested Procedures

- All functions defined at top-level
  - Example: C language
- Global variables have addresses fixed
- Space for local variables stack-allocated at function call
- Code refers directly to globals, and locals are accessed via constant offset from bottom of stack.
Compiling Nested Procedures

• Functions may be defined inside other functions but are not full first-class values
  - E.g., Pascal or Modula-2
  - Must be able to find local variables of lexically enclosing procedures, which do not have fixed positions.
Compiling Functions as Arguments (lexical scope)

• Functions can be arguments but not results
  - "Downward FUNARG"
  - E.g., Algol, Pascal, Modula-2
  - Need to pass code address and information on location of free (non-global) variables
    • E.g., pointer to the right part of the stack
Returning Functions as Results

• Functions as ordinary values
  - "Upward FUNARG"
  - E.g., SML, Scheme, Haskell, etc.
  - Function variables may outlive stack frames!
    • Function values implemented as closures
Closures

• **Two parts**
  - Code to execute when function is called
  - Values of the function's free variables.

• **Interpreter**
  - Values part can be simply the current environment
  - Use this environment when function is called

• **Compiler**
  - Data structure to contain the values of free vars
  - Passed as an extra argument to function when called
Closures vs. Code Pointers

• In C can pass & return function pointers
  ```c
  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

  ```ocaml
  val mergesort:
  ('a * 'a -> 'a) -> ('a list -> 'a list)
  val send:
  channel -> (msg -> 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.
void PressMe(Widget w, XtPointer client_data, 
    XtPointer call_data) {
    printf("%s\n", client_data);
}

int main(int argc, char** argv) {
    Widget button1, button2;
    /* …initialization code for buttons here… */
    XtAddCallback(button1, XmNactivateCallback, 
        PressMe, "aha");
    XtAddCallback(button2, XmNactivateCallback, 
        PressMe, "oho");
}