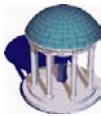


The University of North Carolina at Chapel Hill

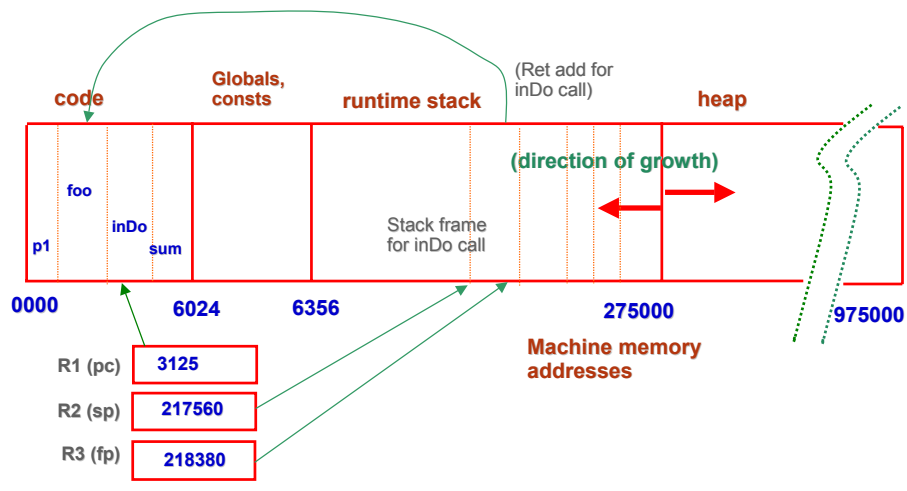
COMP 144 Programming Language Concepts
Spring 2003

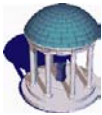
Scope, Symbol Table, Runtime Stack

Hernandez-Campos, Stotts



Sample Memory Layout

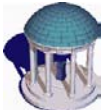




Scope

- **Scope** is the textual region of a program in which a binding is active
- Programming languages implement
 - **Static Scoping**: active bindings are determined using the text of the program at compile time
 - » Most recent scan of the program from top to bottom
 - » Closest nested subroutine rule
 - **Dynamic Scoping**: active bindings are determined by the flow of execution at run time

3



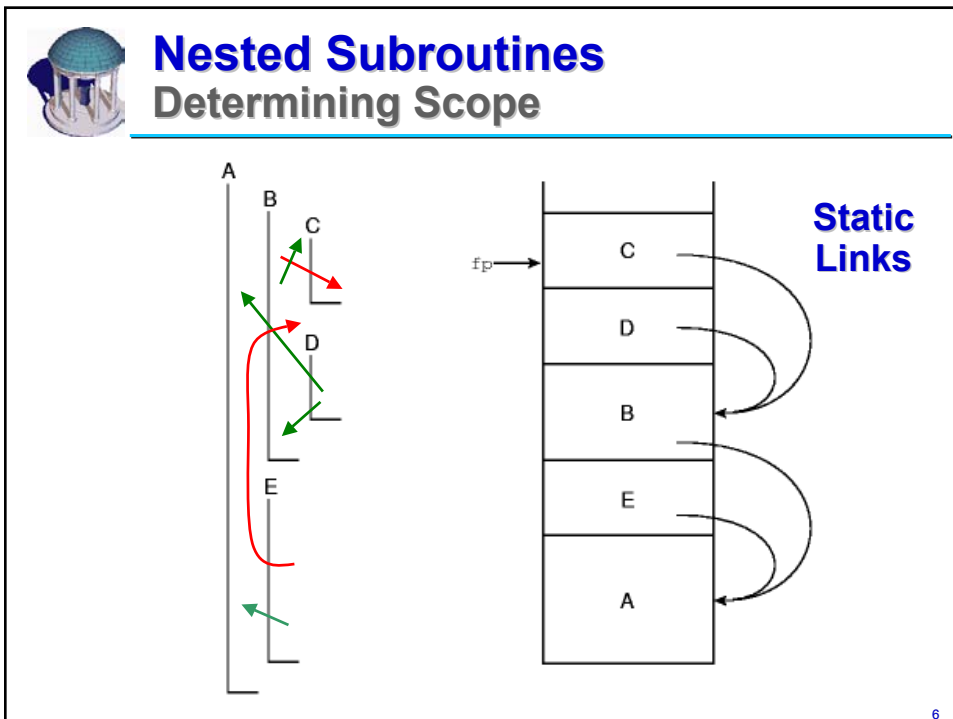
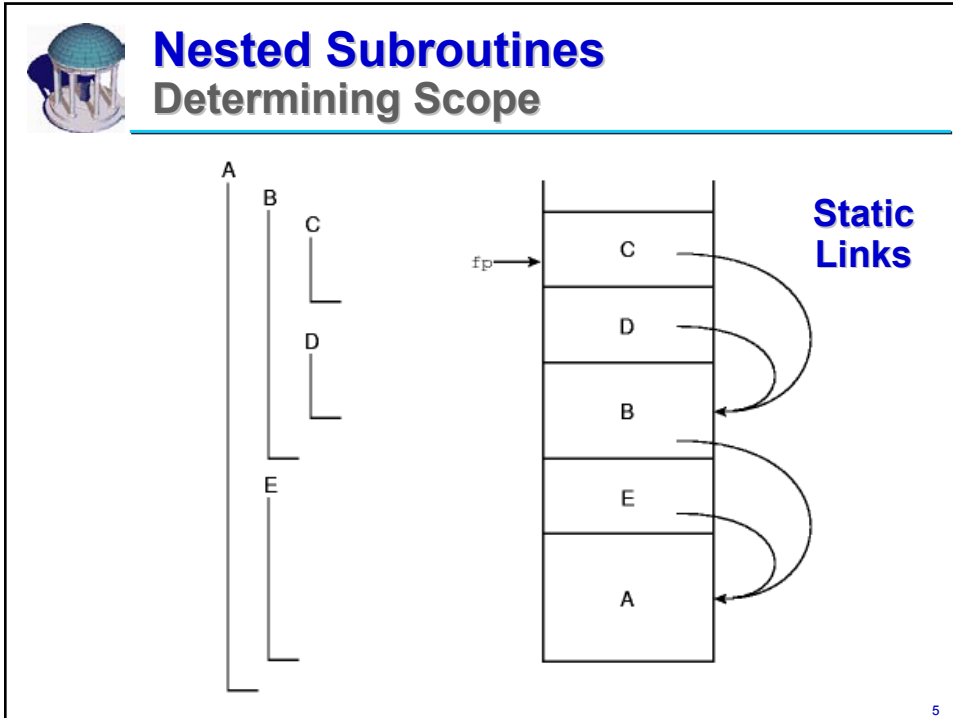
Nested Subroutines Closest Nested Subroutine Rule

- *Nested subroutines* are able to access parameters and local variables of the surrounding scope(s)

```
procedure P1 (A1 : T1);
var X : real;
...
  procedure P2 (A2 : T2);
  ...
    procedure P3 (A3 : T3);
    ...
    begin
      ...      (* body of P3 *)
    end;
  ...
begin
  ...      (* body of P2 *)
end;
...
```

```
...
procedure P4 (A4 : T4);
...
  function F1 (A5 : T5) : T6;
  var X : integer;
  ...
  begin
    ...      (* body of F1 *)
  end;
  ...
begin
  ...      (* body of P4 *)
end;
...
begin
  ...      (* body of P1 *)
end
```

4



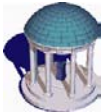


Dynamic Scope

- Bindings between names and objects depend on the flow of control at run time
 - The *current* binding is the one found most recently *during execution*
- Example
 - If static scoping, output is 1
 - If dynamic scoping, output is 2

```
1:  a : integer    -- global declaration
2:  procedure first
3:      a := 1
4:  procedure second
5:      a : integer    -- local declaration
6:      first ()
7:  a := 2
8:  if read_integer () > 0
9:      second ()
10: else
11:     first ()
12: write_integer (a)
```

7



Perl and Dynamic Scope

- Perl allows dynamic scope
- If not declared, variables are *dynamically created, global, and persistent*.
 - Dynamic creation: variables appear when referenced
 - Global: variables thus created can be referenced in any and all code you write.
 - Persistent: variables stay around until the end of execution.
- However... you may *declare* variables in order to override this default...
 - Static Scope (lexical scope): `my $abc`
 - Dynamic Scope: `local $xyz`

8



Perl and Dynamic Scope

my \$abc ;

- makes a variable statically (lexically/locally) scoped.
- only available to this subroutine (scope)
- not available to subroutines you call.
- not available to a subroutine that called you.
- destroyed when execution exits the block it's in.

9

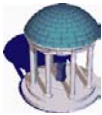


Perl and Dynamic Scope

local \$var;

- makes a variable dynamically scoped.
- sort of a 'temporary global'.
- available to subroutines you call.
- not available to a subroutine that called you.
- can shadow and protect an existing global.
- destroyed when execution exits the block it's in.

10



Perl Example...

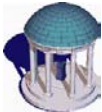
```
$name = "Bill Green";
$you = "Bob Smith";
print "In main script block:\n  Name is $name and you is $you.\n\n";
&printstuff; # call subroutine
print "Back in main script block:\n  Name is $name and you is $you.\n\n";

sub printstuff{
  my $name = "Joe Brown";
  local $you = "Jim White";      # assign in dynamic scope
  print "In 1st level1 sub:\n"   Name is $name and you is $you.\n\n";
  &printmorestuff;              # call subroutine
  $you = "Mary Jones";          # set $you to local/global value
  &printevenmorestuff($name);   # call subroutine, pass value      }

sub printmorestuff{
  print "In 1st level2 sub:\n  Name is $name and you is $you.\n\n";      }

sub printevenmorestuff{
  my $name = @_; # locally assign passed value to $name
  print "In 2nd level2 sub:\n  Name is $name and you is $you.\n\n";      }
```

11



Perl Example...

The program is [here](#)

Output is

```
In main script block:
  Name is Bill Green and you is Bob Smith.

In 1st level1 sub:
  Name is Joe Brown and you is Jim White.

In 1st level2 sub:
  Name is Bill Green and you is Jim White.

In 2nd level2 sub:
  Name is Joe Brown and you is Mary Jones.

Back in main script block:
  Name is Bill Green and you is Bob Smith.
```

12



Lifetime vs. Scope

- Many object exist only when scope is active
- BUT, they are two different things

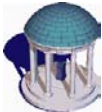
```
class goober {  
    public static int sum = 0;  
    void doFoo (int a, int b) {  
        sum += 1;  
    }  
}
```

does sum exist? ... it's not visible

we can't say sum = sum +5 here

```
g1 = new goober();  
g1.doFoo(3,4);
```


13



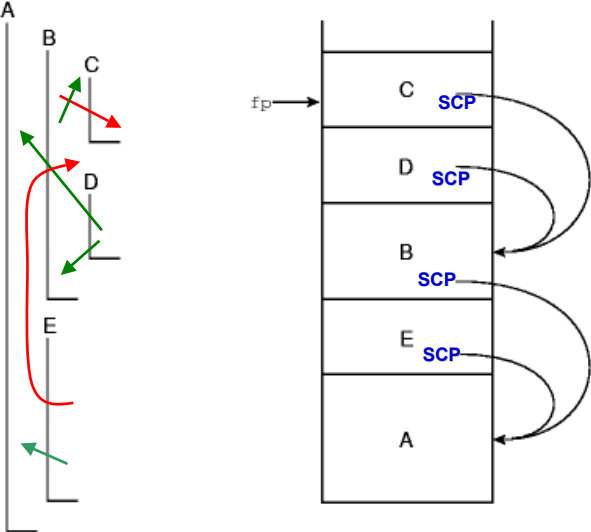
Static Chain

- For finding non-local bindings at run-time
- Each frame contains an **SCP** (*static chain pointer*), a pointer to the most recent frame on the next lexical level out

14




SCP referencing



Static Links

15



Symbol Table

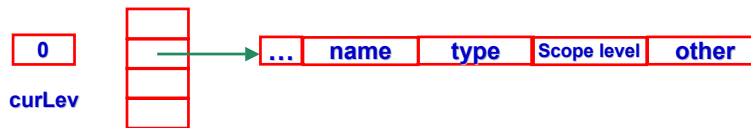
- In statically scoped languages, compilers keep track of names using a data structure called a *symbol table*
- The symbol table *might be* retained after compiling and made available at runtime (for debugging, *e.g.*)
- *Consider first a simple static nested scope language*
 - No modules
 - No objects
 - Like C, Pascal

16

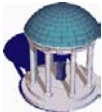


Symbol Table

- In statically scoped languages, compilers keep track of names using a data structure called a *symbol table*
- The symbol table maps names to information about objects
 - It works like a *hash* in Perl



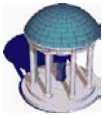
17



Symbol Table: Simplified

- Seeing a new name during parsing makes several things happen
 - *addName* to the ST
 - Is the name a new scope? *addScope*
 - » *New scopes: procedure/method names, nested blocks...*
 - Nesting levels (lexical level) is counted as parsing goes
 - Each name is stored with its scope (lexical level) number
- Compiler keeps track of the lexical level in force when a name is declared
- Multiple entries are made for a name in the hash table... a new *inner* declaration “hides” an outer declaration

18

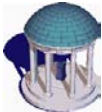


Sample program

Consider this code

```
proc sum (int x17) {  
    int kk = 0;  
  
    proc doFoo {  
        real sum = 0.0;  
  
        proc inDo (int sum) {  
            return sum * x17;  
        }  
    }  
}
```

19

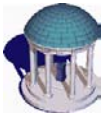


Sample program

Ok... now we parse a little...

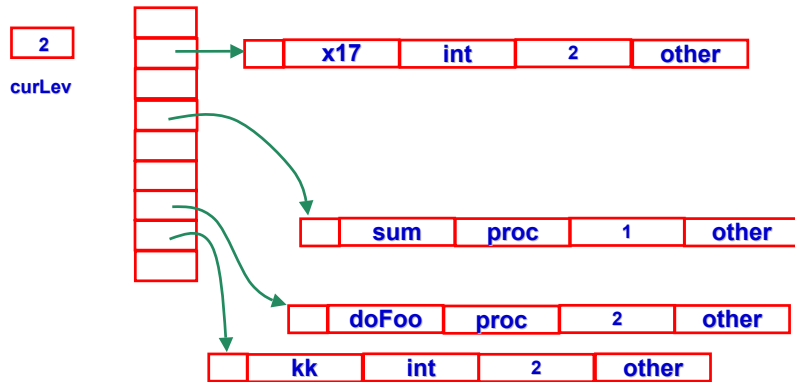
```
proc sum (int x17) {  
    int kk = 0;  
  
    proc doFoo {  
        real sum = 0.0;  
  
        proc inDo (int sum) {  
            return sum * x17;  
        }  
    }  
}
```

20

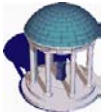


After some parsing...

Symbol table looks like this:



21



Sample program

We parse a little more...

```
proc sum (int x17) {  
  int kk = 0;  
  
  proc doFoo {  
    real sum = 0.0;  
  
    proc inDo (int sum) {  
      return sum * x17;  
    }  
  }  
}
```

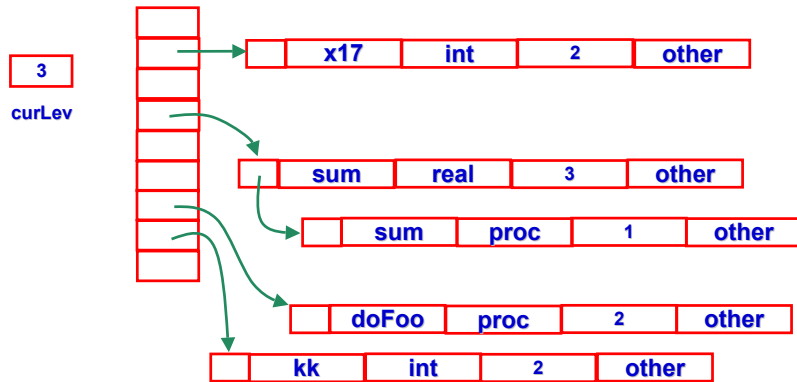
22



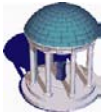
After more parsing...

Symbol table looks like this:

- A second binding for "sum"
- Goes at head of the hash hit list, *hides other bindings*



23

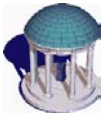


Keep parsing...

a little more...

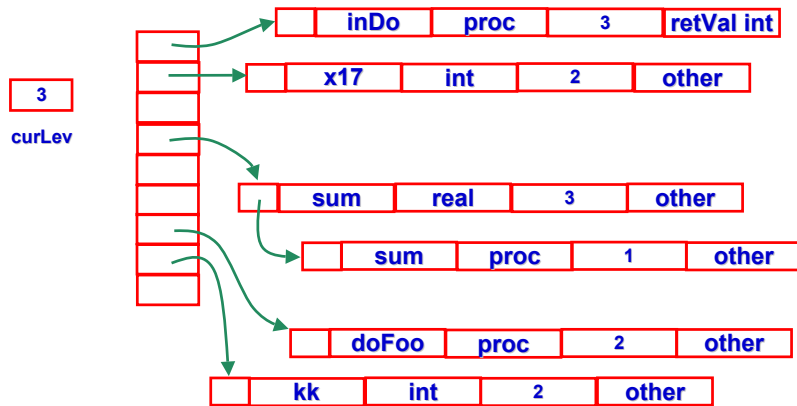
```
proc sum (int x17) {  
    int kk = 0;  
  
    proc doFoo {  
        real sum = 0.0;  
  
        proc inDo (int sum) {  
            return sum * x17;  
        }  
    }  
}
```

24

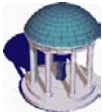


And after a little more...

Symbol table looks like this:



25




Keep parsing...

a little more...

```
proc sum (int x17) {  
    int kk = 0;  
  
    proc doFoo {  
        real sum = 0.0;  
  
        proc inDo (int sum) {  
            return sum * x17;  
        }  
    }  
}
```

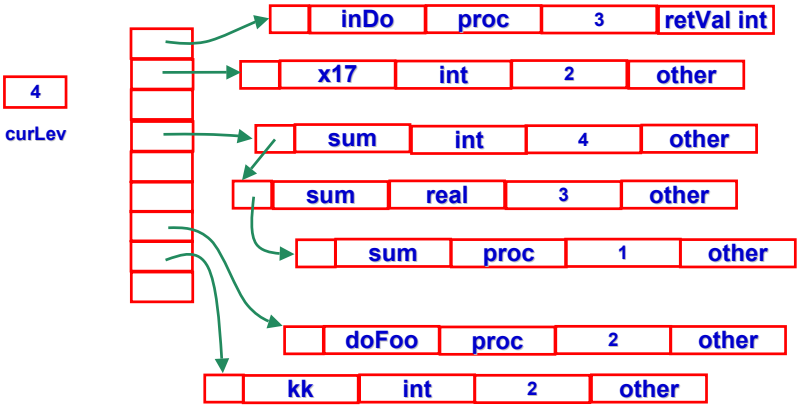
26



Finally...


Symbol table looks like this:

- A third binding for "sum"



inDo	proc	3	retVal int
x17	int	2	other
sum	int	4	other
sum	real	3	other
sum	proc	1	other
doFoo	proc	2	other
kk	int	2	other

27



(Slightly aside...)

If we keep parsing...

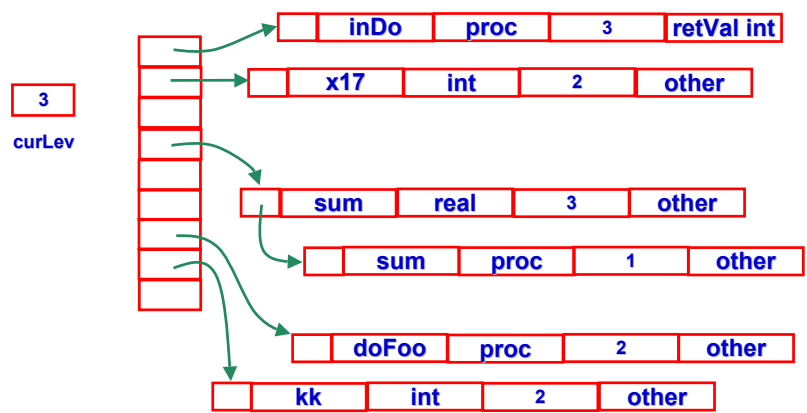
```
proc sum (int x17) {  
    int kk = 0;  
  
    proc doFoo {  
        real sum = 0.0;  
  
        proc inDo (int sum) {  
            return sum * x17;  
        } # here we are leaving a scope, level 4  
    }  
}
```

28



“Pop” off scope information

We remove the head item for all level 4 bindings



29



OK... end aside..

Let's go back to inside
proc inDo

lexical level 4

30

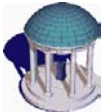


Runtime: Finding a binding

- Now... the questions are ...
 - what storage is bound to "sum"?
 - What storage is bound to "x17"?
 - How do we locate this storage at runtime?

```
proc sum (int x17) {  
  int kk = 0;  
  
  proc doFoo {  
    real sum = 0.0;  
  
    int proc inDo (int sum) {  
      return sum * x17;  
    }  
  }  
}
```

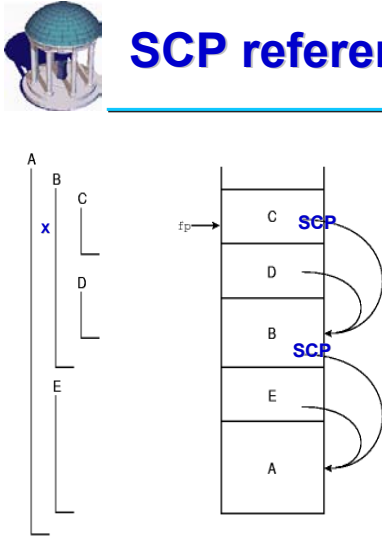
31



Runtime: Finding a binding

- Scope level tells you how many hops to make on the *static chain*
 - Compiler subtracts level of the name from curLev
 - Example: a reference to X
you look up X, find the active one is at level 2
curLev is 4
so at runtime, you must make $(4-1)=3$ hops "up" the static chain to get to the stack frame that contains the proper X
- *Compiler generates code to make these hops happen at runtime*
 - for each reference to a non-local entity
- Alternately, this tells you the index into the *display*

32



The diagram illustrates the process of finding a symbol's location in memory. On the left, a tree structure shows lexical levels A, B, C, D, and E. Level A contains variable X. Level C contains variable X. On the right, a vertical stack of memory frames is shown, with frames for C, D, B, E, and A from top to bottom. A frame pointer (fp) points to the top of frame C. The Symbol Chain Pointer (SCP) is shown in frame C pointing to frame B, and in frame B pointing to frame A. Arrows indicate the path from the SCP in C to B, and then to A, where the variable X is found.

SCP referencing

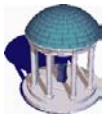
Lets say in **C** we use a var **X** declared in **A**

Compiler will gen code to "chase" the SCP twice

```
Load R1 fp+2 # SCP in C
Load R1 (R1)+2 # fp for B, offset to SCP
Load R1 (R1)-7 # gives fp for A, then
                # offset to X in A's frame
```

Compiler know all these offsets from symbol table

33



Display

- Faster than static chain at run-time
- Requires some overhead to maintain on proc call and return but not really more than maintaining SCP
- Array that contains the active SCP for each lexical level
 - DIS[0] globals*
 - DIS[1] top level*
 - DIS[2] next... etc.*

34



Static Scope: Modules

- Many modern languages are more complicated in their scope rules than Pascal and C
- Modules, classes explicitly manipulate scopes and name visibility
- They are *not nested* in general
- Objects inside a module can see each other (subject to normal lexical scoping)
- Objects outside... able to see in?

37



Import / Export

- Objects in a module are not visible outside unless *exported*
- Objects outside are not visible inside the module unless *imported*
- Bindings made in a module are *inactive* outside, but not *gone*

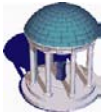
38



Open Scope vs. Closed

- **Open** scope: names do not have to be imported *explicitly* to be visible
 - Nested subroutines in Pascal, *e.g.*
 - We can see the names in outer lexical scopes without having to ask for the ability
- **Closed** scope: names DO have to be imported *explicitly* to be visible
 - Modules in Modula-2, C++, Perl, etc.

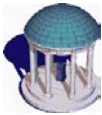
39



Referencing in Modules

- We need a *more complicated symbol table* to generate code for non-local referencing at run-time
- Seeing a new name during parsing makes several things happen
 - Scopes are counted and numbered serially
 - Nesting level is also counted implicitly: *scope stack*

40



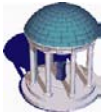
Symbol Table Scope Stack Example

```

type T = record F1: integer;
                F2: real;  end;

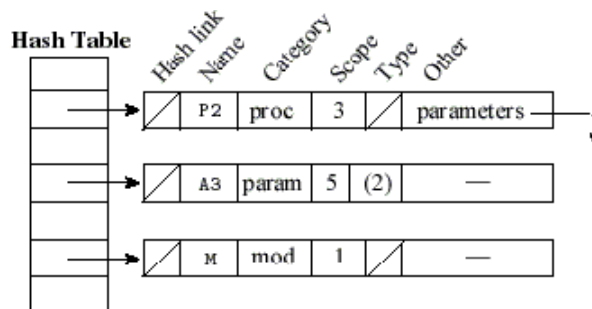
var V : T;
module M;
  export I; import V;
  var I : integer;
  procedure P1 (A1 : real; A2 : integer): real;
    begin . . . end P1;
  procedure P2 (A3 : real);
    var I : integer;
    begin
      with V do . . . end
    end P2;
end M;
    
```

41



Symbol Table Static Scope Example

- Symbol table in Modula-2



42



Reading Assignment

- Scott's chapter 3
 - Section 3.3.2
 - Section 3.3.3