# Scope

- Procedure activation tree
- Run-time stack
- Lexical scope (block structure)
  - Nested procedure declarations
  - Rules
  - Implementation
- Dynamic scope
  - Rules
  - Implementation



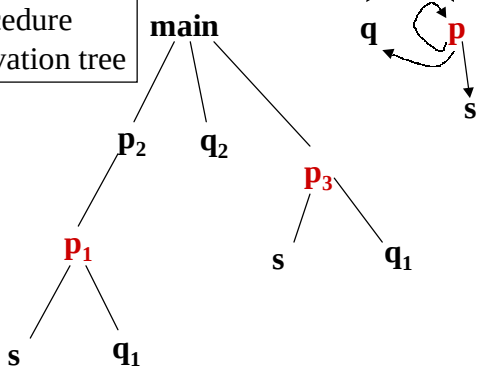
# Procedure Activations

- *Activation* - time between when procedure is entered until it is exited
- *Lifetime* - begins when control enters an activation and ends when control returns from activation
  - Duration of a procedure call
- *Activation tree* - shows flow of control from one activation to another
  - An unfolding of the calling structure of the program

# Example

```
main
{ p
  { s
    { …}
  if ( ) p1 else {s; q1 }
  }
  q
  { …}
  p2 ;
  q2 ;
  p3 ;
}
```

procedure activation tree



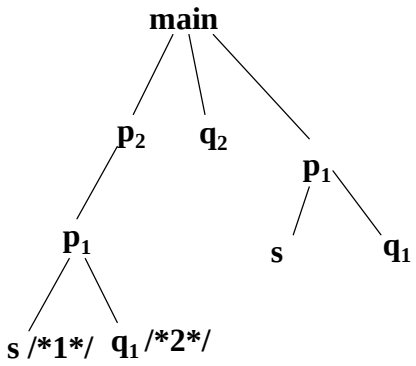

call graph

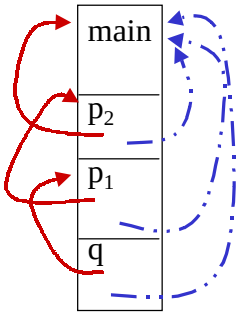



Node calls each of its children from L to R
Leaf : procedure which does not call another procedure
Root: main procedure



# Runtime stack at /*2*/




**What's the runtime stack at /*1*/?**

# Lexical Scoping

- **Block structured PLs**
  - **Allow for local variable declaration**
  - **Inherit global variables from enclosing blocks**
  - **Lookup for non-local variables proceeds from inner to enclosing blocks in inner to outer order.**
  - **Used in Algol, Pascal, Scheme (with *let*), C++**
    - **C is flat language for procedure declarations, disallows nesting**



# Example

```
program
    a, b, c: integer;
    procedure p
        c: integer;
        procedure s
            c, d: integer;
            procedure r
            …
            end r;
            r;
        end s;
        r;
        s;
    end p;
    procedure r
        a: integer;
        = a, b, c;
    end r;
    …; p; …
end program
```

nested block structure

# Runtime Stack

- Mechanism to manage block structured storage
- One frame per block on stack
  - Storage for local variables allocated on block entry, freed on block exit
- Variable lookups conceptually performed on stack along static chain of lexically nested environments
- Stack contains frames of all blocks which have been entered and not yet exited from



# Frame

- Fixed length portion (per procedure)
  - Return pointer into stack frame of caller
  - Return address (to code within caller)
  - Saved state (register values of caller)
  - Address accessing mechanism for nonlocal variables
- Variable length portion
  - Local variable storage (including parameters)
  - Compiler-generated temporary storage for subexpressions

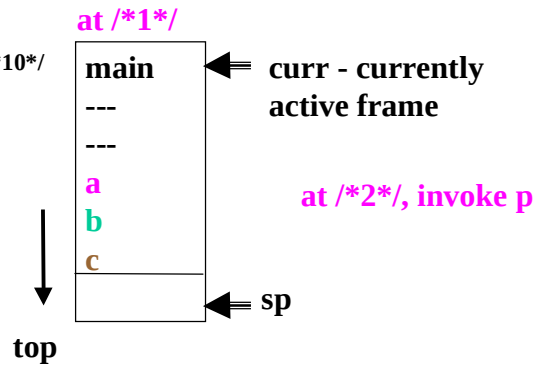
# When a procedure is called….

- **Prologue - setup stack frame**
  - **Initialize fixed length fields (including parameters)**
- **Execution - of code in the called procedure**
- **Epilogue - release stack frame after restoring caller's registers and processing the parameters (if necessary)**



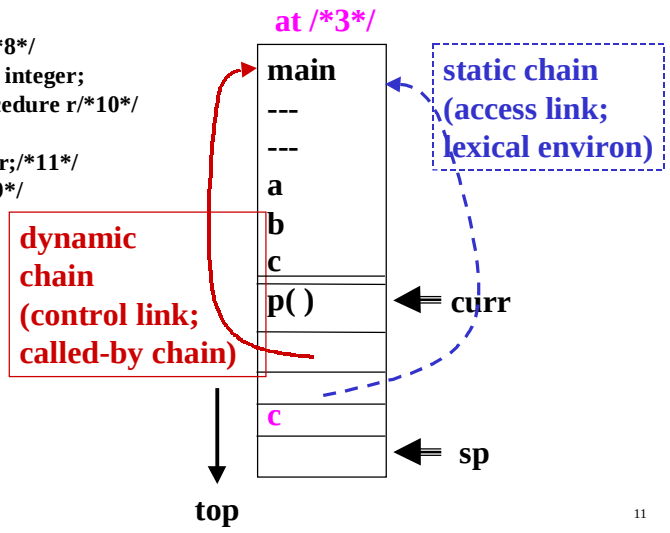
# Example

```
program
    a, b, c: integer;/*1*/
    procedure p /*3*/
        c: integer;
        procedure s /*8*/
            c, d: integer;
            procedure r/*10*/
            …
            end r;/*11*/
            r;/*9*/
        end s;/*12*/
        r;/*4*/
        s;/*7*/
    end p;/*13*/
    procedure r /*5*/
        a: integer;
        = a, b, c;
    end r; /*6*/
    …; p;/*2*/ …
end program /*14*/
```



# Example

```
program
    a, b, c: integer;/*1*/
    procedure p /*3*/
        c: integer;
        procedure s /*8*/
            c, d: integer;
            procedure r/*10*/
            …
            end r;/*11*/
            r;/*9*/
        end s;/*12*/
        r;/*4*/
        s;/*7*/
    end p;/*13*/
    procedure r /*5*/
        a: integer;
        = a, b, c;
    end r; /*6*/
    …; p;/*2*/ …
end program /*14*/
```

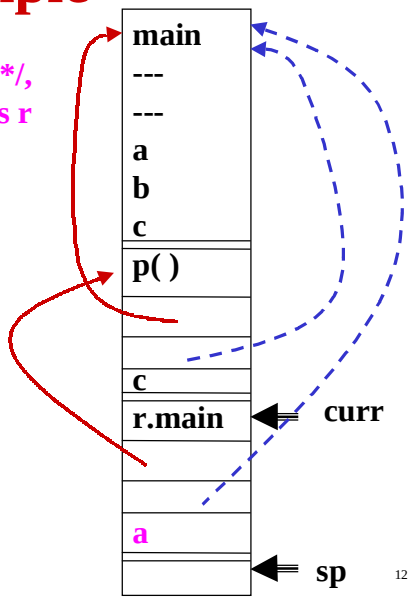





# Example

```
program
    a, b, c: integer;/*1*/
    procedure p /*3*/
        c: integer;
        procedure s /*8*/
            c, d: integer;
            procedure r/*10*/
            …
            end r;/*11*/
            r;/*9*/
        end s;/*12*/
        r;/*4*/
        s;/*7*/
    end p;/*13*/
    procedure r /*5*/
        a: integer;
        = a, b, c;
    end r; /*6*/
    …; p;/*2*/ …
end program /*14*/
```

at /*4*/,
p calls r



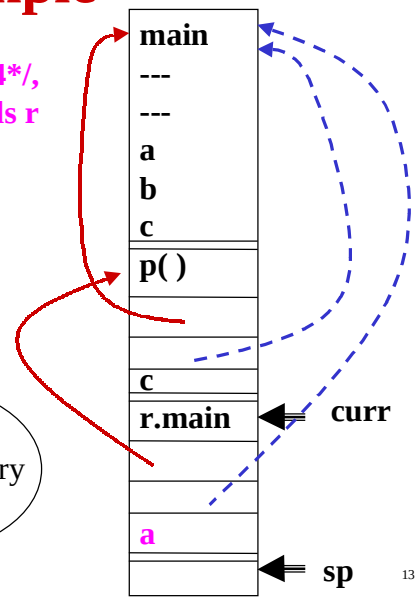
# Example

at /*5*/

```
program
    a, b, c: integer;/*1*/
    procedure p /*3*/
        c: integer;
        procedure s /*8*/
            c, d: integer;
            procedure r/*10*/
            …
            end r;/*11*/
            r;/*9*/
        end s;/*12*/
        r;/*4*/
        s;/*7*/
    end p;/*13*/
    procedure r /*5*/
        a: integer;
        = a, b, c;
    end r; /*6*/
    …; p;/*2*/ …
end program /*14*/
```

at /*4*/, p calls r

How to map c to its memory location?

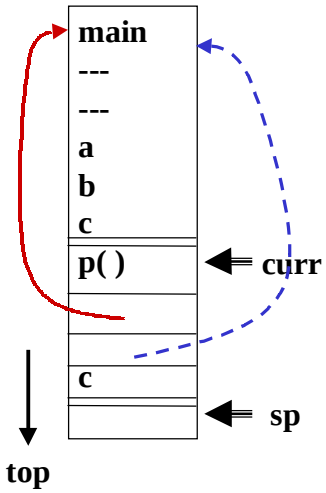





# Example

at /*6*/ r.main exits
sp ← curr
curr ← called-by link in r.main's frame

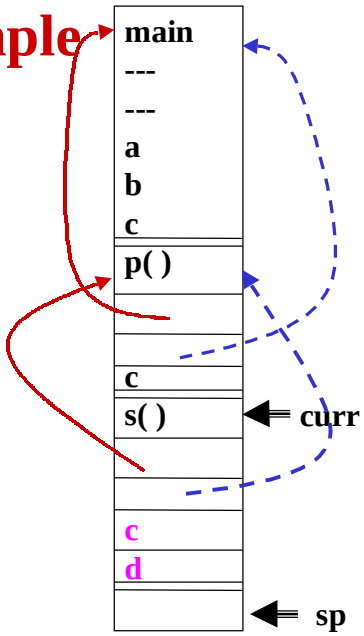
```
program
    a, b, c: integer;/*1*/
    procedure p /*3*/
        c: integer;
        procedure s /*8*/
            c, d: integer;
            procedure r/*10*/
            …
            end r;/*11*/
            r;/*9*/
        end s;/*12*/
        r;/*4*/
        s;/*7*/
    end p;/*13*/
    procedure r /*5*/
        a: integer;
        = a, b, c;
    end r; /*6*/
    …; p;/*2*/ …
end program/*14*/
```



# Example

```
program
    a, b, c: integer;/*1*/
    procedure p /*3*/
        c: integer;
        procedure s /*8*/
            c, d: integer;
            procedure r/*10*/
            …
            end r;/*11*/
            r;/*9*/
        end s;/*12*/
        r;/*4*/
        s;/*7*/
    end p;/*13*/
    procedure r /*5*/
        a: integer;
        = a, b, c;
    end r; /*6*/
    …; p;/*2*/ …
end program/*14*/
```

at /*7*/, call of s in p; at /*8*/:

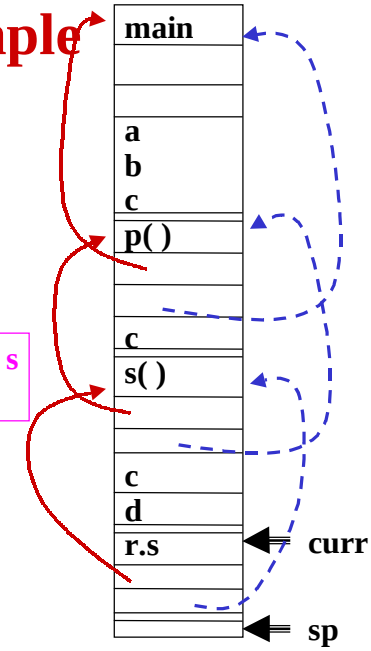





# Example

```
program
    a, b, c: integer;/*1*/
    procedure p /*3*/
        c: integer;
        procedure s /*8*/
            c, d: integer;
            procedure r/*10*/
            …
            end r;/*11*/
            r;/*9*/
        end s;/*12*/
        r;/*4*/
        s;/*7*/
    end p;/*13*/
    procedure r /*5*/
        a: integer;
        = a, b, c;
    end r; /*6*/
    …; p;/*2*/ …
end program/*14*/
```

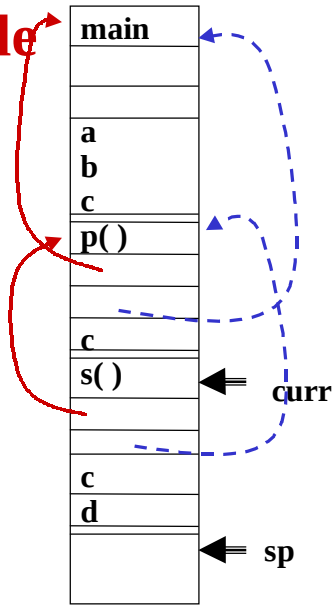
/*9*/call of r.s in s at /*10*/



# Example

```
program
    a, b, c: integer;/*1*/
    procedure p /*3*/
        c: integer;
        procedure s /*8*/
            c, d: integer;
            procedure r/*10*/
            …
            end r;/*11*/
            r;/*9*/
        end s;/*12*/
        r;/*4*/
        s;/*7*/
    end p;/*13*/
    procedure r /*5*/
        a: integer;
        = a, b, c;
    end r; /*6*/
    …; p;/*2*/ …
 end program/*14*/
```

/*11*/ pop r's frame

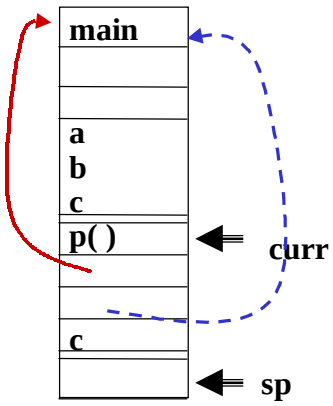





# Example

```
program
    a, b, c: integer;/*1*/
    procedure p /*3*/
        c: integer;
        procedure s /*8*/
            c, d: integer;
            procedure r/*10*/
            …
            end r;/*11*/
            r;/*9*/
        end s;/*12*/
        r;/*4*/
        s;/*7*/
    end p;/*13*/
    procedure r /*5*/
        a: integer;
        = a, b, c;
    end r; /*6*/
    …; p;/*2*/ …
 end program/*14*/
```

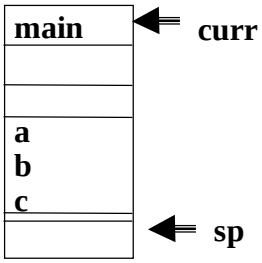
/*12*/pop s's frame



# Example

```
program
    a, b, c: integer;/*1*/
    procedure p /*3*/
        c: integer;
        procedure s /*8*/
            c, d: integer;
            procedure r/*10*/
            …
            end r;/*11*/
            r;/*9*/
        end s;/*12*/
        r;/*4*/
        s;/*7*/
    end p;/*13*/
    procedure r /*5*/
        a: integer;
        = a, b, c;
    end r; /*6*/
    …; p;/*2*/ …
end program/*14*/
```

at /*13*/




/*13*/ pop p's frame
/*14*/ pop main's frame
so that curr == sp



# Scoping and the Run-time Stack

- **Static Scope: Access link points to stack frame of the lexically enclosing procedure; nesting depth of a procedure is determined at compile time, and does not change.**
  - **Non-local name binding done at compile time**
- **Dynamic Scope: Control link points to stack frame of caller. Location of procedure in dynamic calling chain can change.**
  - **Non-local name binding done at run-time**

# Lexical Scoping

```
1  program
       a, b, c: integer;
   2   procedure p
           c: integer;
       3   procedure s
               c, d: integer;
           4   procedure r
               …
               end r;
               r;
           end s;
           r;
           s;
       end p;
   2   procedure r
           a: integer;
           = a, b, c;
       end r;
       …; p; …
   end program
```

Each name: (nesting_level, local_index)

```
program
    (1.1), (1,2), (1,3): integer;
    procedure (1,4)
        (2,1): integer;
        procedure (2,2)
            (3,1), (3,2): integer;
            procedure (3,3)
            …
            end r;
            (3,3);
        end s;
        (1,5);
        (2,2);
end p;
procedure (1,5)
        (2,1): integer;
        = (2,1), (1,2), (1,3);
end r;
…; p; …
end program
```



# Find run-time stack location for (level, index) pair?

- need current procedure level k
- if k = level, look in current frame (local var)
- if k > level, must find level's activation record

  ⇒ follow (k - level) access links
- k < level cannot occur

Note: access links need to be maintained on procedure entry and exit

# Find run-time stack location for (level, index) pair?

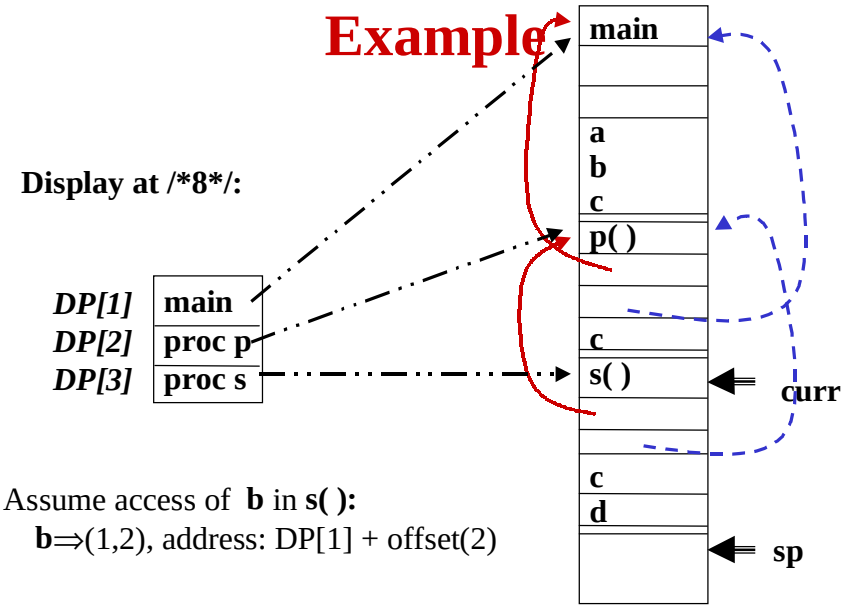
- To reduce run-time access cost, use a display: A table of access links, one entry for each nesting level
- uses data structure at known location, DP
- find (level, index) activation record:

  just look up DP(level)

Note: work needed to maintain display on procedure entry and exit



# Example




Assume access of **b** in **s( ):**

**b**⇒(1,2), address: DP[1] + offset(2)

# Dynamic Scoping

- **Allows for local variable declaration**
- **Inherit global variables from procedures which are *live* when current procedure is invoked**
  - ***Reference to identifier is resolved to the declaration of that identifier in the most recently invoked and not yet terminated block that contains a declaration of that identifier***



# Dynamic Scoping

- **Lookup for non-local variables proceeds from closest dynamic predecessor to farthest**
- **Incurs a runtime cost of the lookup**
  - **Why can't this information of "who called this procedure" be precomputed?**
- **Used in APL, (old)Lisp, Snobol**

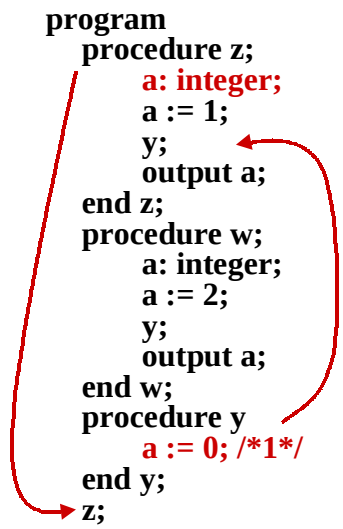
# Example

```
program
   procedure z;
         a: integer;
         a := 1;
         y;
         output a;
   end z;
   procedure w;
         a: integer;
         a := 2;
         y;
         output a;
   end w;
   procedure y
         a := 0; /*1*/
   end y;
   z;
   w;
end program;
```

Which a is modified by /*1*/ under dynamic scoping? a.z or a.w or both?



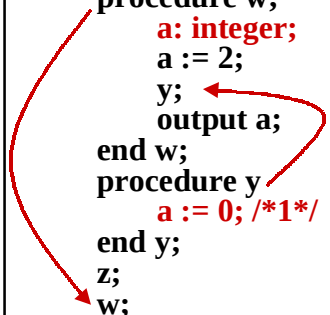
# Example




main calls z,
z calls y,
y sets a.z to 0.

# Example

```
program
   procedure z;
        a: integer;
        a := 1;
        y;
        output a;
   end z;
   procedure w;
        a: integer;
        a := 2;
        y;
        output a;
   end w;
   procedure y
        a := 0; /*1*/
   end y;
   z;
   w;
end program;
```



main calls w,
w calls y,
y sets a.w to 0.

Is this program legal under static scoping? If so, which a is modified? If not, why not?



# Central Reference Table

- Can't use <level,index> addressing of display because dynamic chain is NOT FIXED LENGTH
- Try to minimize cost of runtime variable lookup
- Runtime access to variables is indirect through this hash table, 1 entry per active identifier name

# Central Reference Table

- **1 entry per distinct identifier name plus active/inactive flag**
  - **If active flag on, entry contains variable's address**
- **Procedure prologue initializes the table entries for local variables of this procedure (each entry is really a stack)**
- **Procedure epilogue pops entries for local variables from the table**



# New Example

```
program
   procedure z;/*4*/
         a: integer;
         a := 1;
         w;
         /*9*/ y;
         output a;
   end z;/*10*/
   procedure w;/*5*/
         a: integer;
         a := 2;
         y;
         output a;
   end w;/*8*/
   procedure y/*6*/
         a := 0;
   end y;/*7*/
   /*3*/ z;
end program;
```

**table entry for a at:**

| | | |
|---|---|---|
| /*3*/ | empty | top |
| /*4*/ | &(a.z) | ← |
| /*5*/ | &(a.w), &(a.z) | |
| /*6*/ | *&(a.w)*, &(a.z) | |
| /*7*/ | &(a.w), &(a.z) | |
| /*8*/ | &(a.z) | |
| /*9*/ | &(a.z) | |
| /*6*/ | *&(a.z)* | |
| /*7*/ | &(a.z) | |
| /*10*/ | empty | |