

# Scope

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

# 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;
    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;**

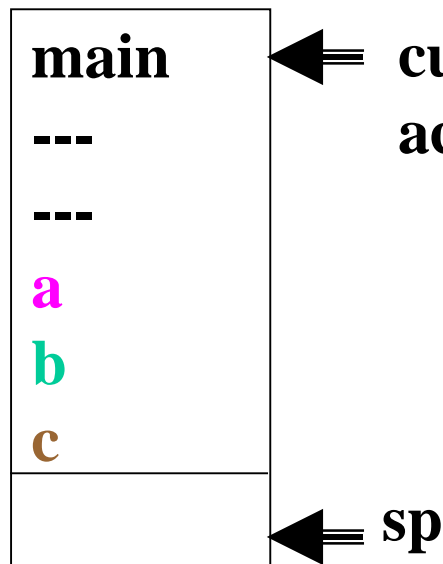
**= a, b, c;**

**end r; /\*6\*/**

**...; p; /\*2\*/ ...**

**end program /\*14\*/**

**at /\*1\*/**

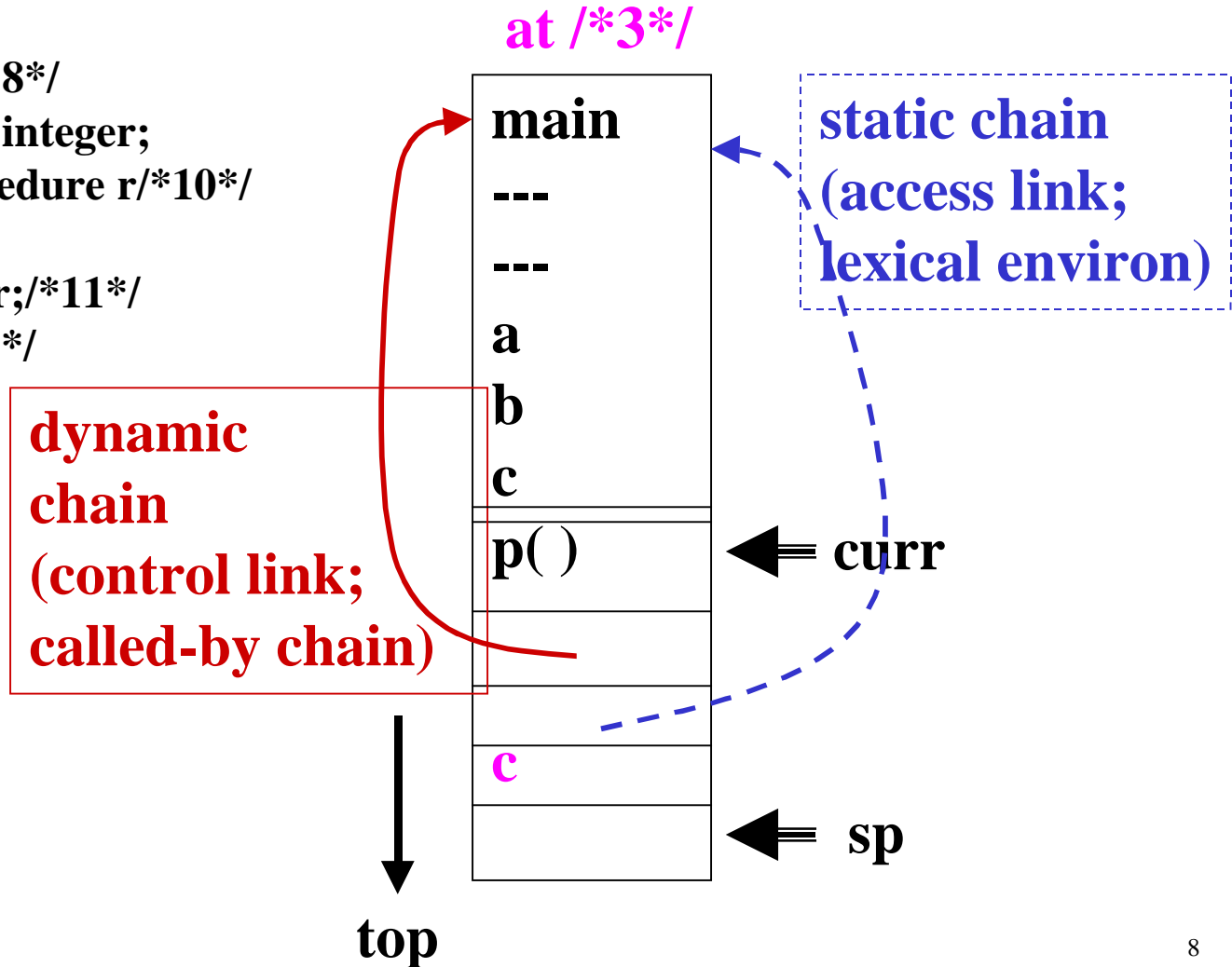


**at /\*2\*/, invoke p**

# 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;

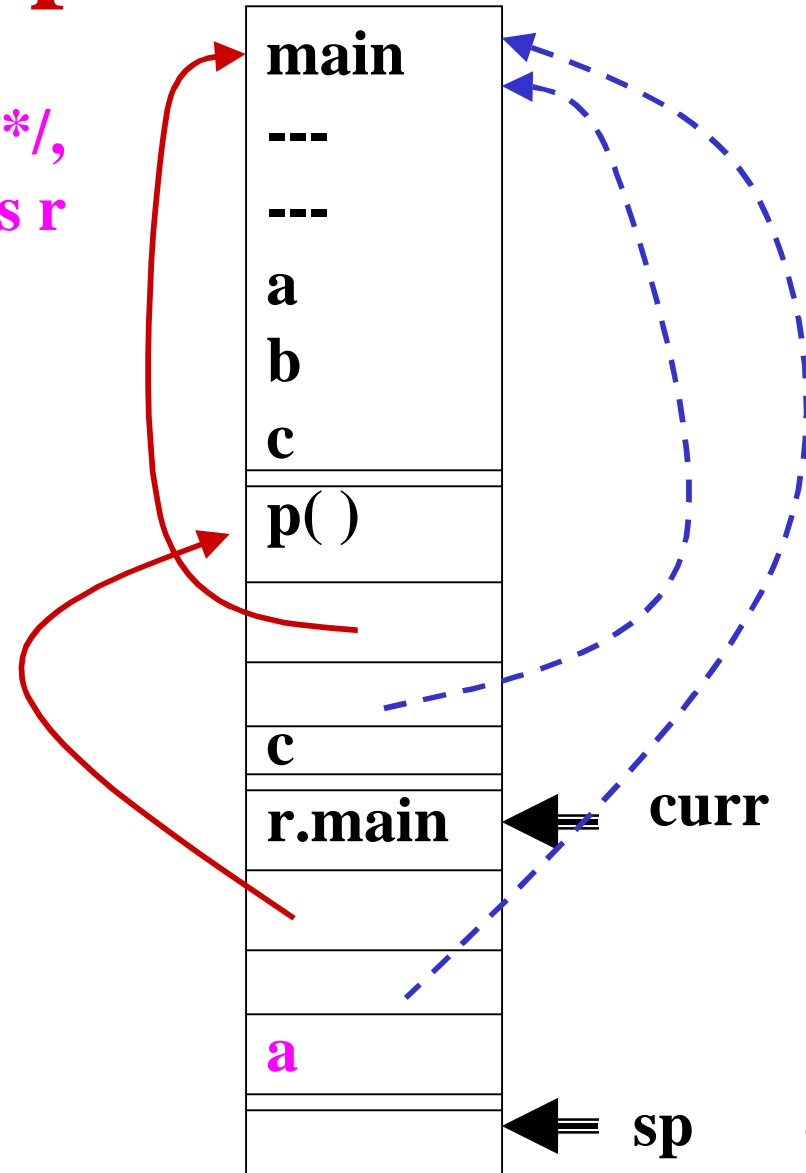
end r; /\*6\*/

...; p; /\*2\*/ ...

end program /\*14\*/

at /\*4\*/,  
p calls r

at /\*5\*/



# 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*/
```

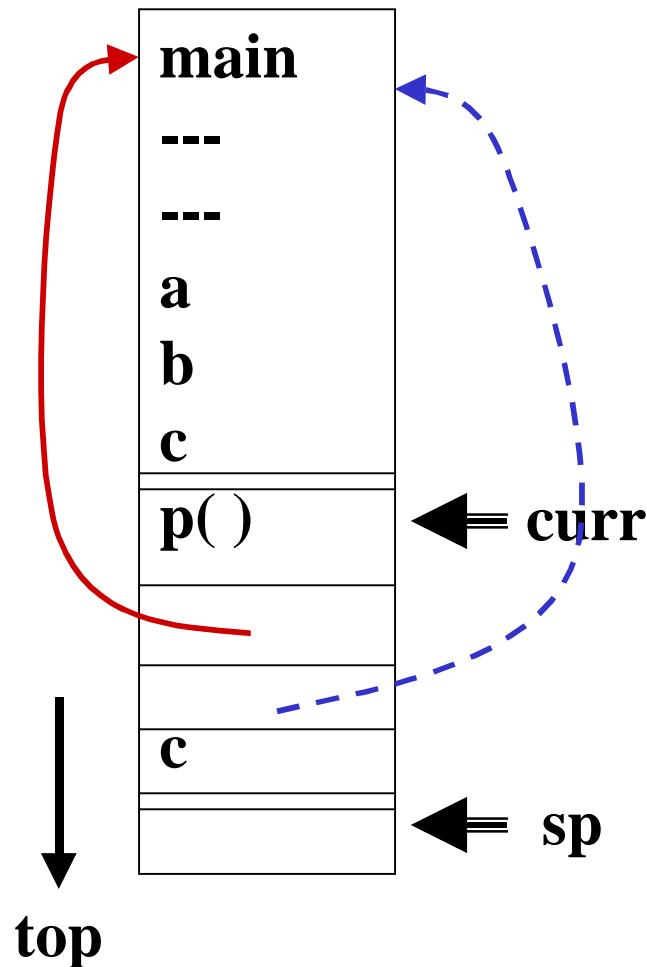
```
...; p; /*2*/ ...
```

```
end program /*14*/
```

at /\*6\*/ r.main exits

sp ← curr

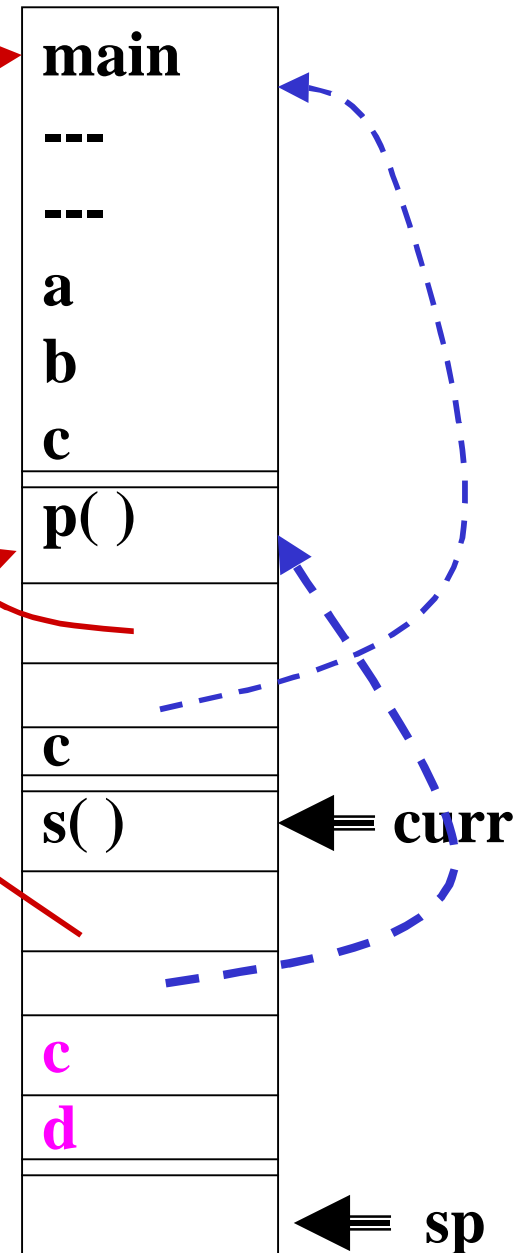
curr ← called-by link  
in r.main's frame



# 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*/
```

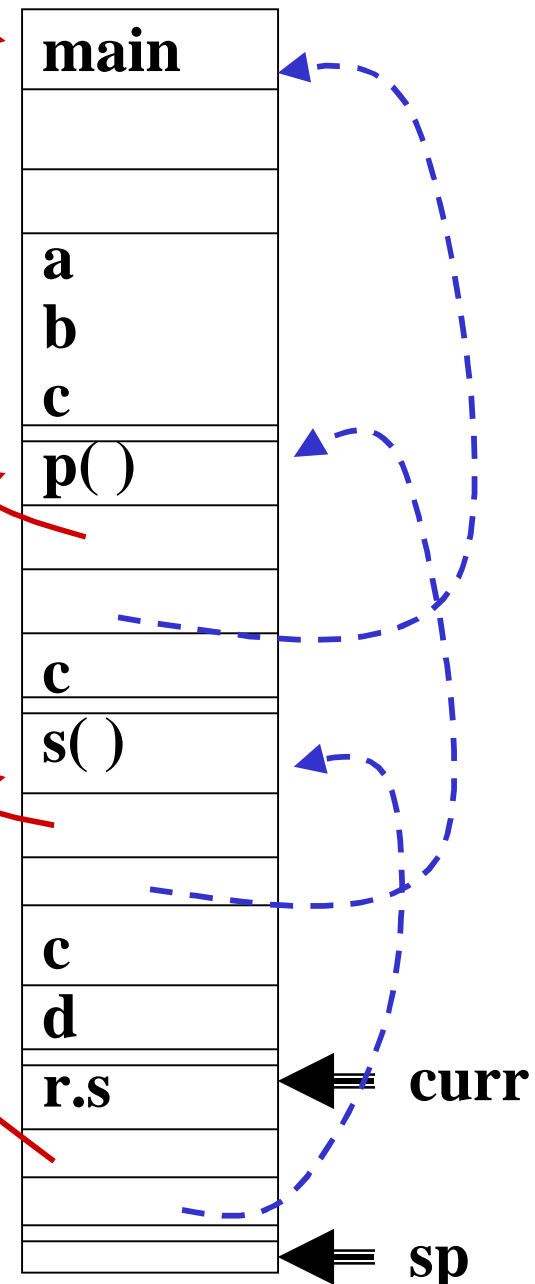
at /\*7\*/,  
call of s in p;  
at /\*8\*/:



# 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\*/



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;

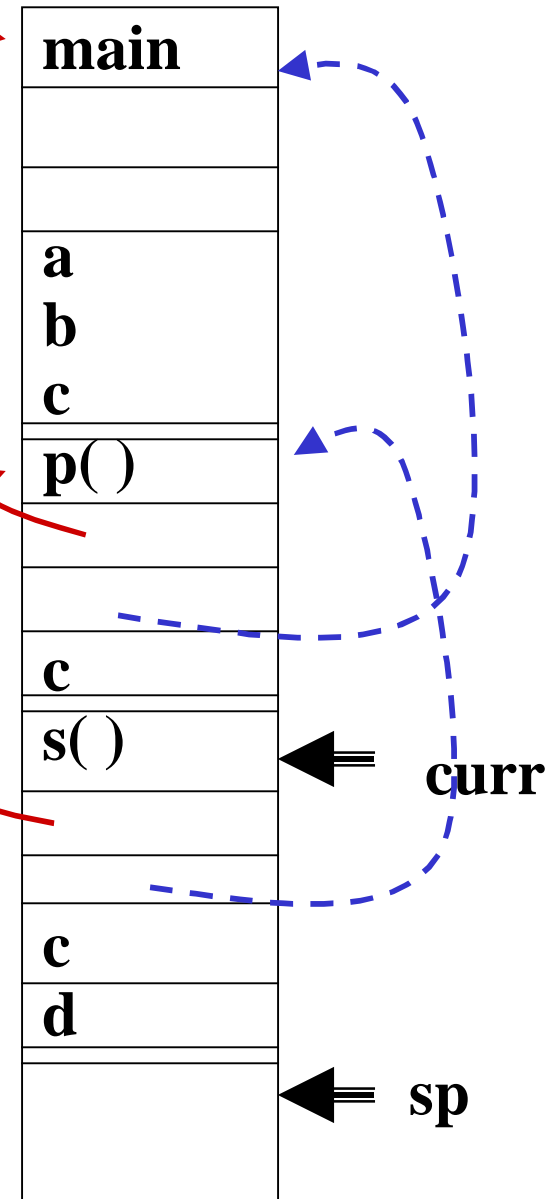
end r; /\*6\*/

...; p; /\*2\*/ ...

end program /\*14\*/

# Example

/\*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;

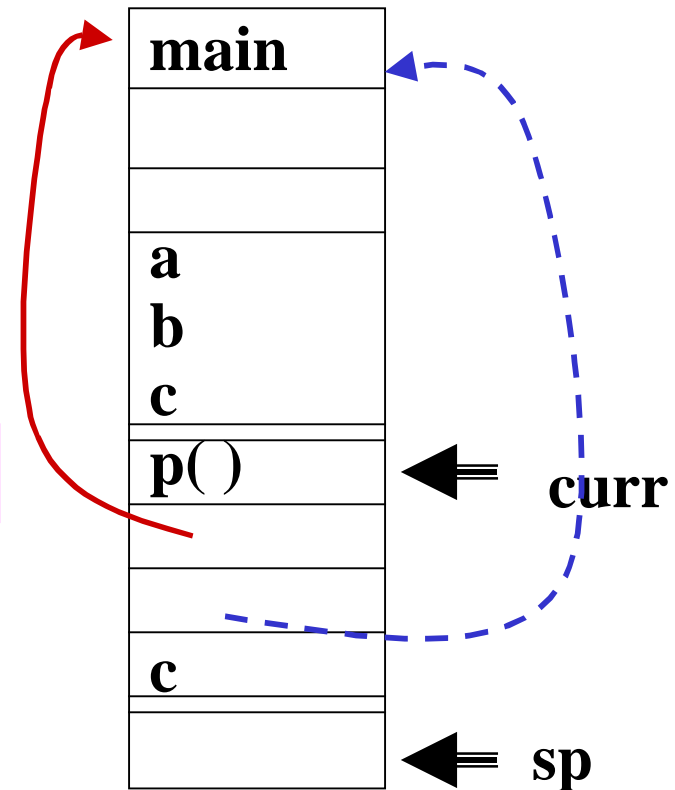
= a, b, c;

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;

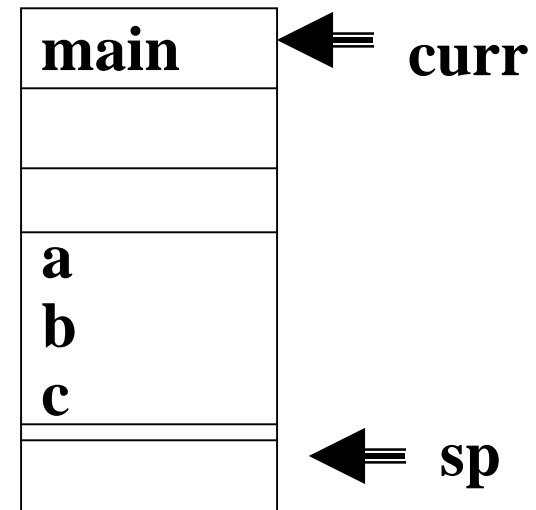
= a, b, c;

end r; /\*6\*/

...; p; /\*2\*/ ...

end program /\*14\*/

at /\*13\*/



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

# Observations

- **Where a procedure appears on the static chain, is the same place no matter where it is called from**
  - Used to implement static scoping using a *display*
- **Where a procedure appears on the dynamic chain varies according to calling context**

# Display

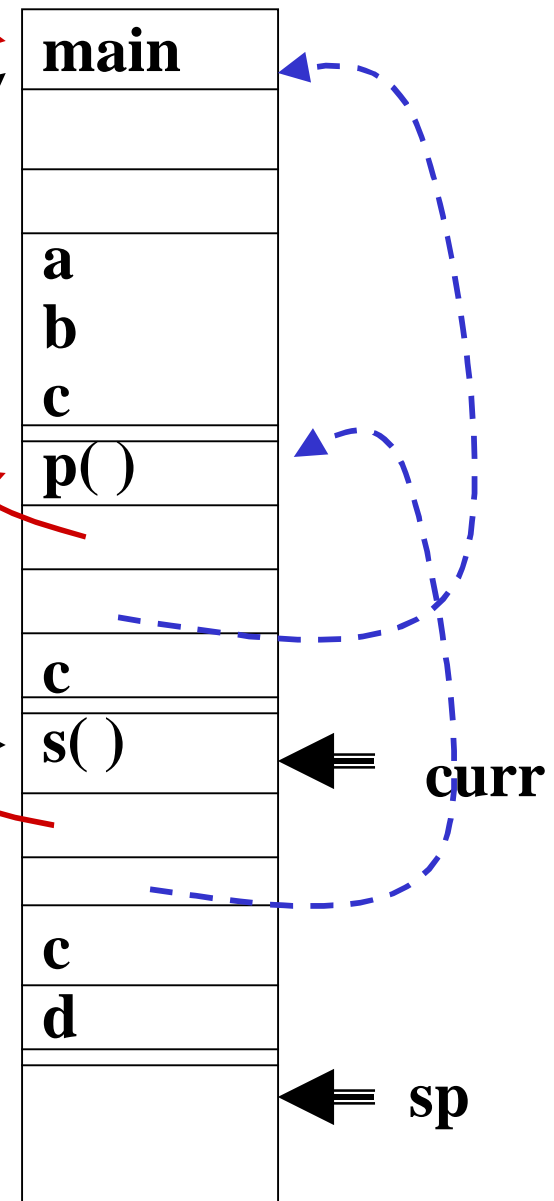
- **First used in Algol60 implementation**
- ***Display* - a vector of pointers to currently active static chain frames on the runtime stack**
  - **A stack of nested environments**
- **Variable address encoded**
  - **Contents of  $\text{display}[j] + \text{offset}$** 
    - **where  $j$  is nesting level of an open block**
- **Manage display in procedure prologue and epilogue**

# Example

Display at /\*8\*/:

[1] main  
[2] proc p  
[3] proc s  
top=3

c.s is  $\langle \text{display}[\text{top}] + 3 \rangle$   
b.main is  $\langle \text{display}[\text{top} - 2] + 4 \rangle$



# Display

- **Allows direct access to data in runtime stack without list operations**
- **Procedure prologue**
  - Put frame on stack
  - Update display and adjust Top, if needed
- **Procedure epilogue**
  - Pop frame from stack
  - Update display and adjust Top, if needed (note: sometimes update means replace some entries by others)
- **Use static chain to setup display**

# Example

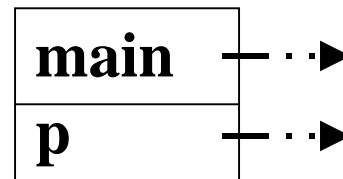
```
main  
{ a: integer;  
  procedure p( )/*2*/  
  {a: integer;  
    ... q( );....  
  }  
  procedure q( )/*3*/  
  {...}  
  p( )/*1*/  
}
```

at /\*1\*/, display:

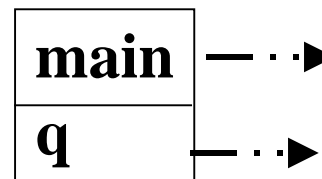


Draw the corresponding runtime stacks.

at /\*2\*/, display:



at /\*3\*/, display:



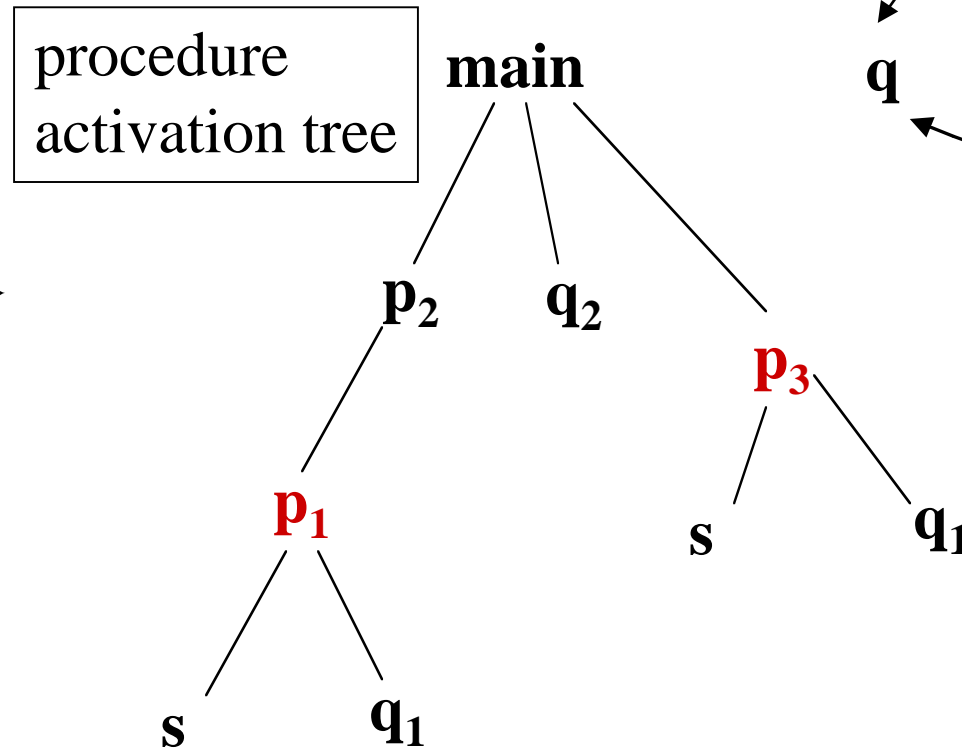
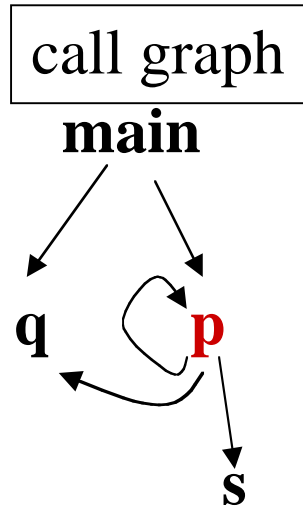
# Nesting Depth

- Nesting depth of a procedure - number of static links which may have to be followed to resolve references + 1
- In previous example (slide 18), *main* is at depth 1, *p* at depth 2 and *s* at depth 3
  - *a.main* in *s* is at  $\text{Display}[1] + \text{offset}$  where 1 is nesting depth of *main*
  - Previous encoding conversion: *a.main* is at  $\text{Display}[\text{top}-k] + \text{offset}$ 
    - where  $k = \text{nesting depth}(s) - \text{nesting depth}(\text{main})$
    - e.g.,  $\text{top}=3$ ,  $k = 3-1 = 2$ ,  $\text{Display}[1]$  with pointer to *main*

# 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

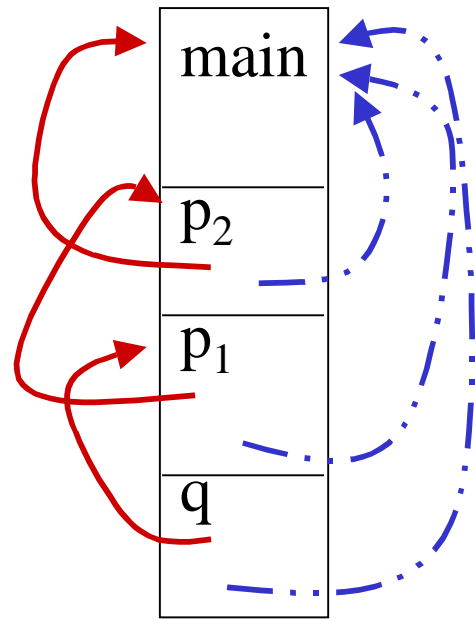
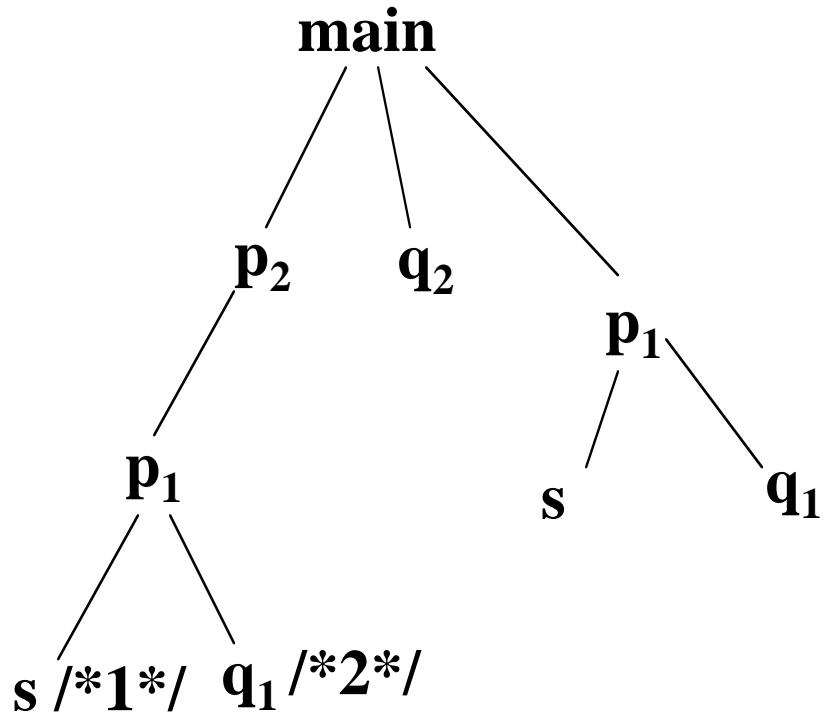


main

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

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\*/?

# 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

```
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 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 <base,offset> 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*/	<span style="border: 1px solid black;">&amp;(a.w)</span> , &(a.z)	
/*7*/	&(a.w), &(a.z)	
/*8*/	&(a.z)	
/*9*/	&(a.z)	
/*6*/	<span style="border: 1px solid black;">&amp;(a.z)</span>	
/*7*/	&(a.z)	
/*10*/	empty	