

Scoping Subprograms

Lecture 7-8

Data control

- Problem: how to provide data to operations and subprograms?

Or what is the “environment” of the reference by name?

- Two major problems:
 1. one name can denote different objects (e.g. local variables)
 2. one object can be denoted by several names (e.g. passing parameters)
- To solve these problems the **environments** were proposed.
- **Environment**: binding between the names (*Ide*) and values:

$$Env : Ide \rightarrow Loc \cup Val$$

Environments

Operations in programming language that affect the environment:

1. **Creation of binding** $\langle name, object \rangle$

- Example: declarations, parameters... in the beginning of execution and when entering the subprogram

2. **Use of the environment**

- Example: reference to the identifier (variables, names of subprograms)

3. **Deactivation the binding**

- Example: when P calls Q , some bindings of P are deactivated

4. **Reactivation the binding**

- Example: when Q returns control to P

5. **Destruction the binding**

- Example: return from subprogram, the end of execution

Blocks and local variables

- A block consists of local declarations and commands:

```
begin
    D    => local declarations
    C    => commands
end
```

- Example (C):

```
x:=5;
{ int x; x:=7;
  printf("%d", x); => 7
}
printf("%d", x);    =>5
```

- A block is like a procedure without parameters

Scoping

- The “scoping” solves the problem of determining...
 - ... when a particular binding $\langle name, object \rangle$ is active?
 - ... or which bindings are valid in a particular moment of execution?
 - ...or which is the environment?
- Different environments:
 - **local environment (LE)** : all bindings created/activated in a block/
subprogram
 - **non-local environment (NLE)** : all bindings used (active) but not local
 - **global environment (GE)**: all bindings shared by all blocks/
subprograms. GE can be considered:
 - as a subset of NLE
 - separately from NLE

Global Environment (GE)

- Example (C):

```
int a[20];  
float b[5];  
struct { int i; char n[10]; } c, d;  
...  
int main() {...}
```

- Contains also all the identifiers (constants, functions...) predefined in the language
- Common table for all the subprograms (including main)
- **Concrete implementation:**
 - treated as a record
 - the names are compiled as fields of the record
 - in the code, it's sufficient to know the address of the base of GE

Local Environment

- **Notation:**

$P \Downarrow Q$ procedure P calls Q

$P \Uparrow Q$ procedure P terminates and returns the control to the caller Q

- Let's consider the computation

$P \Downarrow Q \Downarrow R \Uparrow Q \Uparrow P$

what happens to the local environment of Q ?

- The simple part:

$Q \Downarrow R$ when control is passed to R , LE becomes deactivated

$R \Uparrow Q$ when control is passed back to Q , its LE become reactivated

Local Environment (cont.)

- The management of environment in Q

$P \Downarrow Q$ and $Q \Uparrow P$

is more delicate.

Two possible solutions:

1. **DLE**: Dynamic Local Environment

$P \Downarrow Q$ LE of Q is **created**

$Q \Uparrow P$ LE of Q is **destroyed**

2. **SLE**: Static Local Environment

$P \Downarrow Q$ LE of Q is **reactivated**

$Q \Uparrow P$ LE of Q is **deactivated**

Local Environment (cont.)

- Example: static option in C creates static local environment

```
void f()  
{  
    static int x = 0;  
    x++;  
    printf("%d ", x);  
    f();  
}  
...
```

```
while(1) { f(); }
```

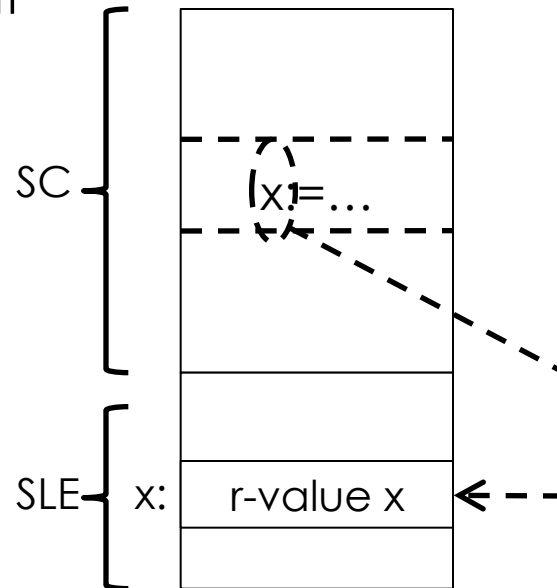
⇒ 1 2 3 4 5 ...

- What happens without static?

Local environment: Implementation

10

1. Static local environment

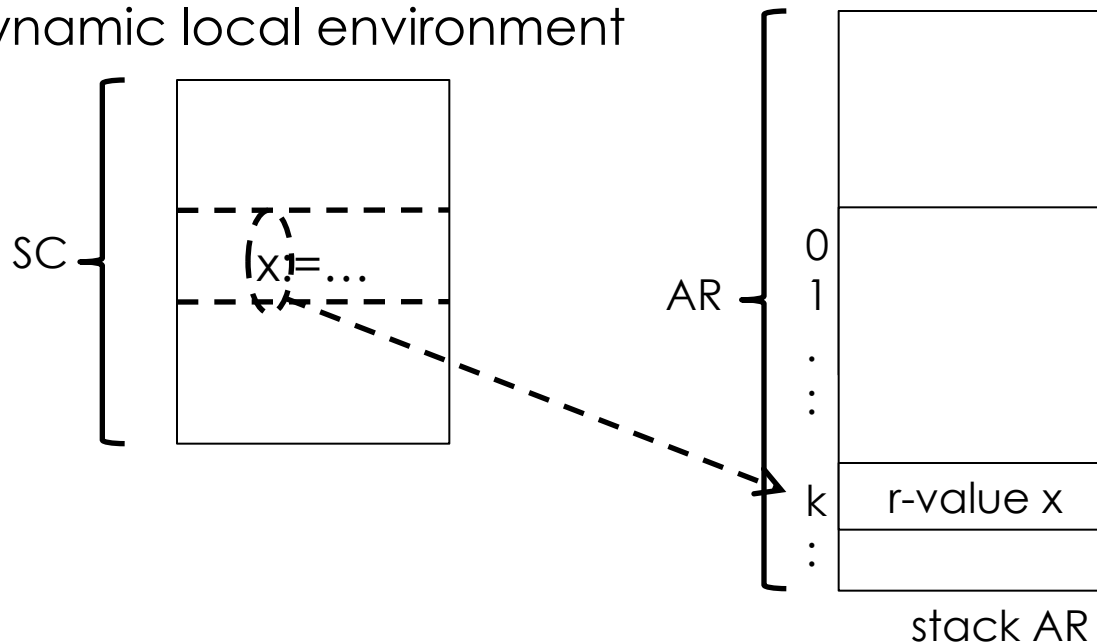


- The table of static local environment: it's memorized only once and divided by all the calls of subprogram
- SLE is simply a sequence of r-value
- The names are offset inside the SLE

Local environment: Implementation

11

2. Dynamic local environment



- The local environment is a part of the activation record (AR); different calls of subprogram correspond to different instances of the local environment
- Also in this case the local name of the subprogram is compiled as offset, but this time inside the AR

Non local references

- Example

```
procedure Q()  
begin  
    ...  
    x  
    ...  
end
```

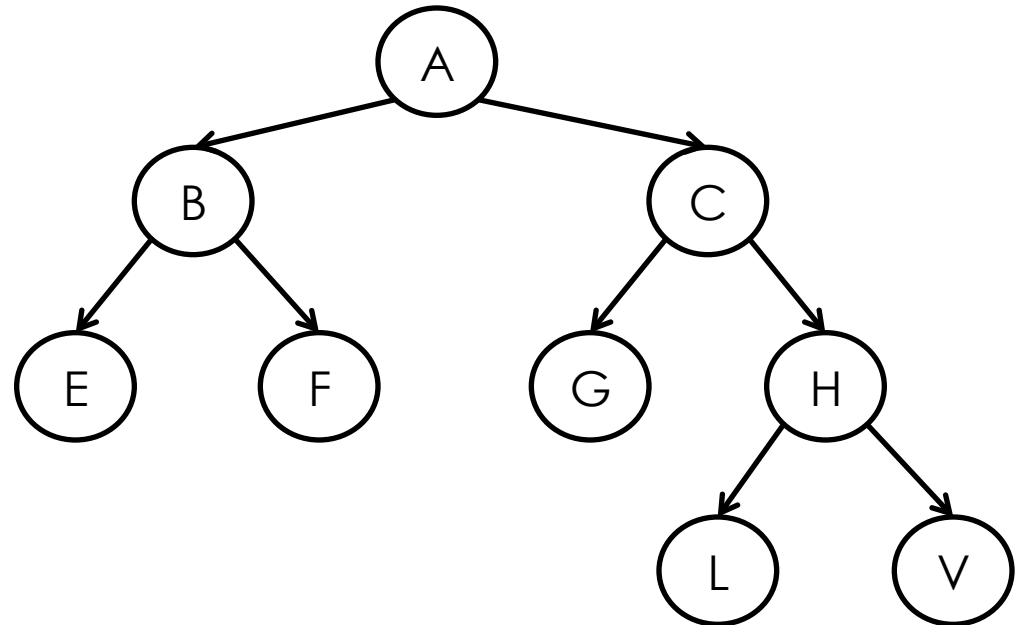
- If x is not local, which binding is used for x?
- Answer: **rules of scoping**
 - Dynamic scoping: rules of visibility are related to the execution (Lisp)
 - Static scoping: rules of visibility are related to the structure (syntax) of the program: it's the most used technique in the modern languages (C, C++, Java, Pascal, ML,...)

Static scoping

- Every identifier has a declaration that statically binds it. This binding is constant at runtime.
 - The type of the identifier is known at compile time
 - The location for the value of identifier can change at runtime (dynamic local environment) or not (static local environment)
- For more rigorous analysis, for every program let's associate a tree called **scoping tree**:
 - [we give different names to blocks (the subprograms already have different names)]
 - nodes of the tree -> names of the blocks and subprograms
 - Q is a child of P if
 - Q is a direct block of P
 - Q is a subprogram declared in P

Static scoping (cont.)

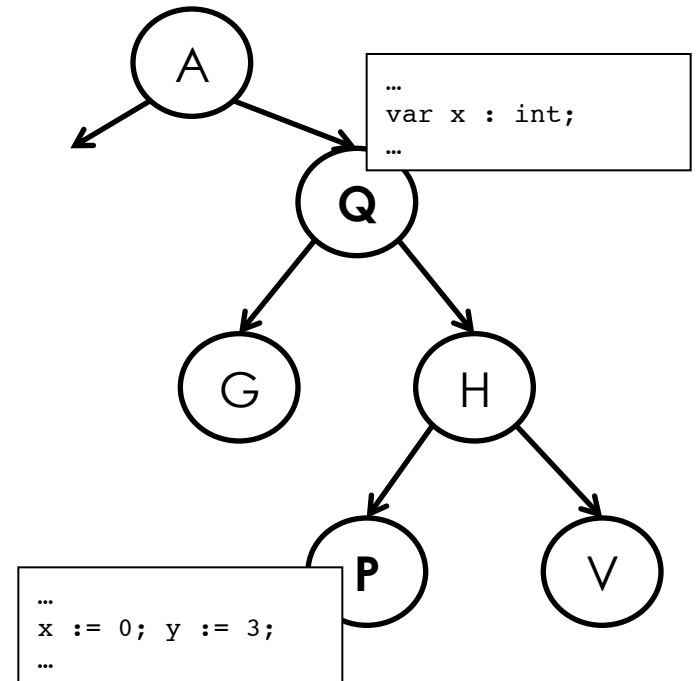
```
A: begin
  proc B;
    begin
      E: begin...end
      F: begin...end
    end {B}
  C: begin
    G: begin...end
    proc H;
      begin L: begin...end
        V: begin...end
      end {H}
    end {C}
  end {A}
```



Rule of static scoping

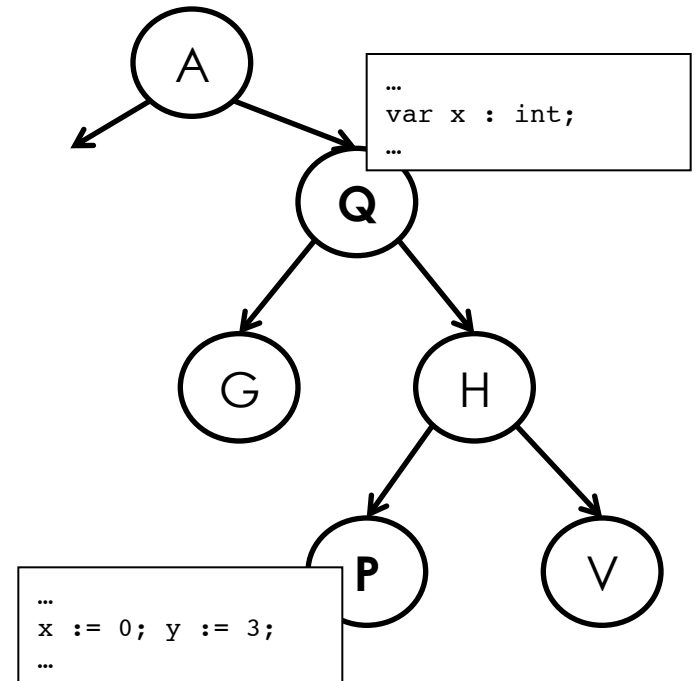
- If x occurs in non local reference in the subprogram/block P
 1. non local environment that provides correct binding for x is the parent Q nearest to P in which x is declared
 2. if there is no parent Q that declares x , the error is generated (this control is made at compile time)

Note: Here the global environment is the environment of the outermost subprogram/block



Rule of static scoping (cont)

- If the language defines a global environment outside of subprograms/blocks, then scoping rule is rewritten:
 1. non local environment that provides correct binding for x is the parent Q nearest to P in which x is declared [as above]
 2. if there is no parent Q of P that declares x , then x is searched in the global environment
 3. if not found an error is generated (at compile time)



Static scoping: semantics

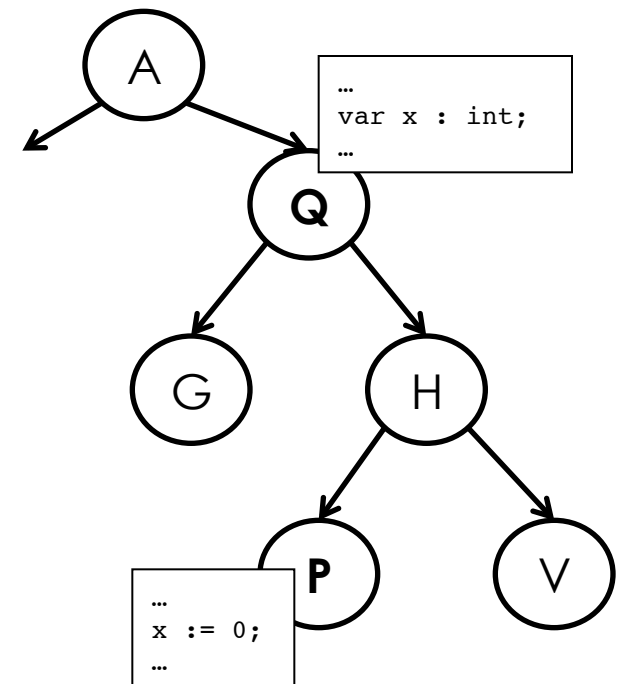
- Let's change the definition of the environment: an environment (global) becomes a sequence of local environments:

$$Env = List(IDE \rightarrow DVal) \quad r = [r_0, r_1, \dots, r_k]$$

$$DVal = (Val \cup Loc)$$

- Rule of scoping: $r(x)$ is defined as follows:
 - if $r_k(x)$ is defined, then $r_k(x)$, otherwise:
 - if $r_{k-1}(x)$ is defined, then $r_{k-1}(x)$, otherwise:
 - ...
 - if $r_0(x)$ is defined, then $r_0(x)$, otherwise:
 - ERROR
- Also $DVal$ is changed, in order to keep track of the declarations of subprograms:

$$DVal = (Val \cup Loc \cup Com)$$



Static scoping: semantics (cont.)

$D \parallel \text{const } v = n \parallel_{[r_0, r_1, \dots, r_k]} s = [r_0, r_1, \dots, r_k] s$ where :

$$r_k'(y) = \begin{cases} r_k'(y) & \text{if } y \neq v \\ n & \text{if } y = v \end{cases}$$

$D \parallel \text{var } v := n \parallel_{[r_0, r_1, \dots, r_k]} s = [r_0, r_1, \dots, r_k] s'$ where :

$$r_k'(y) = \begin{cases} r_k'(y) & \text{if } y \neq v \\ l & \text{if } y = v \end{cases} \quad s'(x) = \begin{cases} s(x) & \text{if } x \neq l \\ n & \text{if } x = l \end{cases}$$

where $l = (\text{newmem } s)$ is a new location in s

$D \parallel \text{proc } P = C \parallel_{[r_0, r_1, \dots, r_k]} s = [r_0, r_1, \dots, r_k] s$ where :

$$r_k'(y) = \begin{cases} r_k'(y) & \text{if } y \neq P \\ C & \text{if } y = P \end{cases}$$

Static scoping: implementation

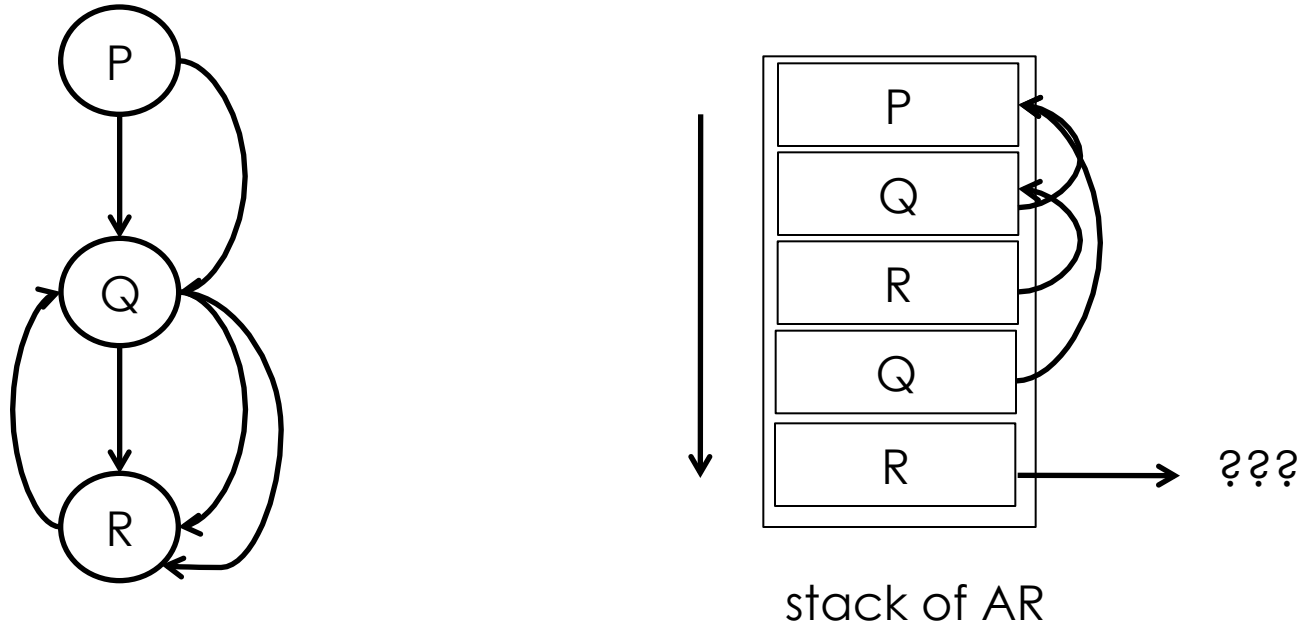
- **Problem:** the stack of AR provides a temporal order between local environments (useless for static scoping), but gives no indication on the structure of the program.
- **Solution:**
 - To each AR the **static chain pointer** (SCP) is added.
 - The "static" information on the syntactic structure (scoping tree) is implemented through the SCP.
 - Let's assume that a subprogram/block Q is a parent of subprogram/block P in the scoping tree. Then, the SCP of an AR of P points to AR of Q according to the rule of static scoping.
- Note: we consider the case of dynamic local environment.

Static scoping: implementation (cont.)

20

Suppose that $Q \Downarrow R$

then, the AR of P is pushed in the stack of AR



R is a child of Q but in the stack there are several occurrences of Q.

Algorithm to determine SCP

- Suppose α and β are nodes of the scoping tree
- Suppose that $\alpha \Downarrow \beta$

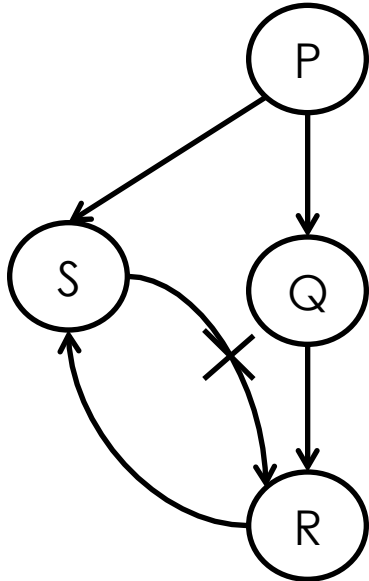
then, the parent of β should be an ancestor of α

(otherwise β would not be visible from α)

Algorithm to determine SCP

- Suppose α and β are nodes of the scoping tree
- Suppose that $\alpha \Downarrow \beta$

then, the parent of β should be an ancestor of α
 (otherwise β would not be visible from α)



```

P: begin
  proc S; begin...end {S}
  proc Q;
    begin
      proc R; begin...end {R}
    end {Q}
  end {P}
  
```

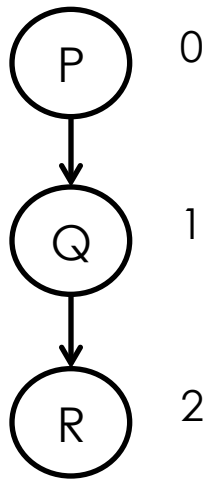
Algorithm to determine SCP

- Suppose α and β are nodes of the scoping tree
- Suppose that $\alpha \Downarrow \beta$

then, the parent of β should be an ancestor of α

(otherwise β would not be visible from α)

- Let's define $\#(\alpha, \beta) = \text{depth}(\alpha) - \text{depth}(\text{parent}(\beta))$
- Example:



$$Q \Downarrow R \text{ then } \#(Q, R) = 1 - 1 = 0$$

$$R \Downarrow Q \text{ then } \#(R, Q) = 2 - 0 = 2$$

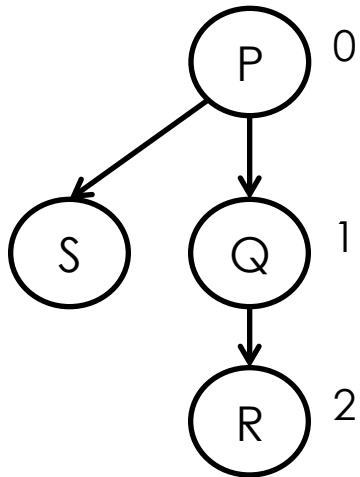
Algorithm to determine SCP

- If $P \Downarrow Q$ then
 1. The AR of Q (AR_Q) is put in the stack
 2. The distance $\#(P, Q)$ is calculated
 3. The address a is reached by making $\#(P, Q)$ steps starting from SCP of AR of the caller P.

This is the address of an AR corresponding to a subprogram/block T that declares Q.
 4. SCP of AR_Q has a value a

Determining SCP: Examples

$P \Downarrow Q \Downarrow R \Downarrow Q \Downarrow R \Downarrow S$

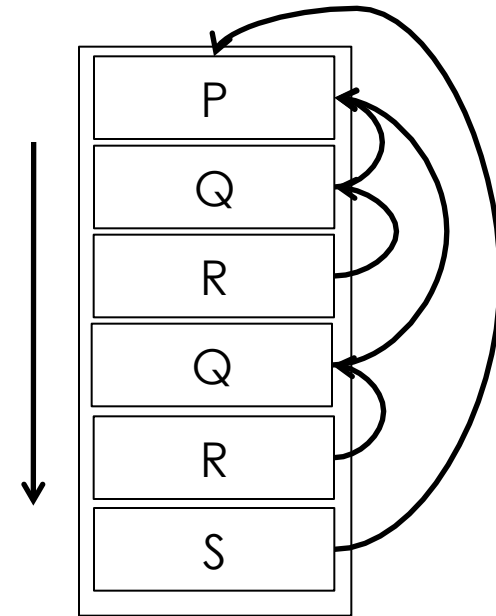


$$\#(P, Q) = 0$$

$$\#(Q, R) = 0$$

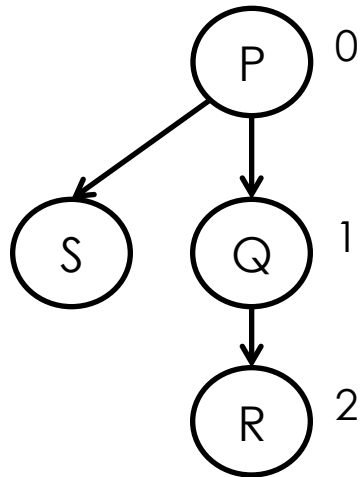
$$\#(R, Q) = 2$$

$$\#(R, S) = 2$$



Determining SCP: Examples

$P \Downarrow Q \Downarrow R \Downarrow Q \Downarrow S$

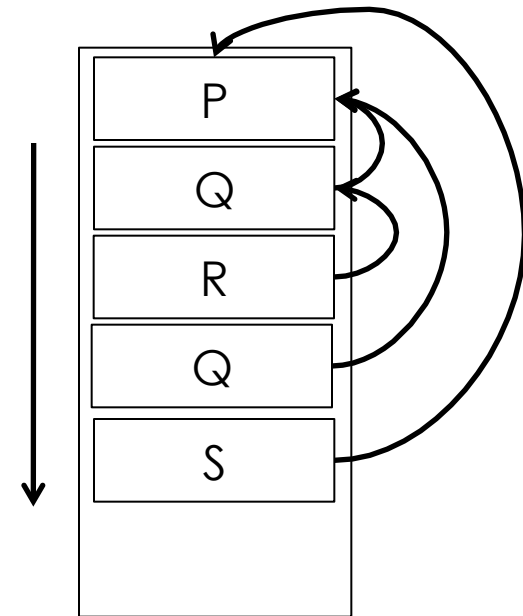


$$\#(P, Q) = 0$$

$$\#(Q, R) = 0$$

$$\#(R, Q) = 2$$

$$\#(Q, S) = 1$$



Calling a subprogram: semantics

- If $r = [r_0, r_1, \dots, r_k]$, then

$$C \parallel \text{call } P \parallel_{r_0} = C \parallel \text{Cmd} \parallel_{r'_0}$$

where

- program P is declared as $\text{proc } P = \text{Cmd}$
- $\text{Cmd} = \tau(P) \in \text{Com}$
- $r' = [r_0, r_1, \dots, r_h, r_\varepsilon]$ where:
 - $h = \text{depth}(r, P)$, or r_h is “the deepest” environment where P is defined:
 - $r_h(P)$ is defined,
 - $r_{h+1}(P), r_{h+2}(P), \dots, r_k(P)$ are not defined
 - r_ε is a new local (empty) environment for Cmd

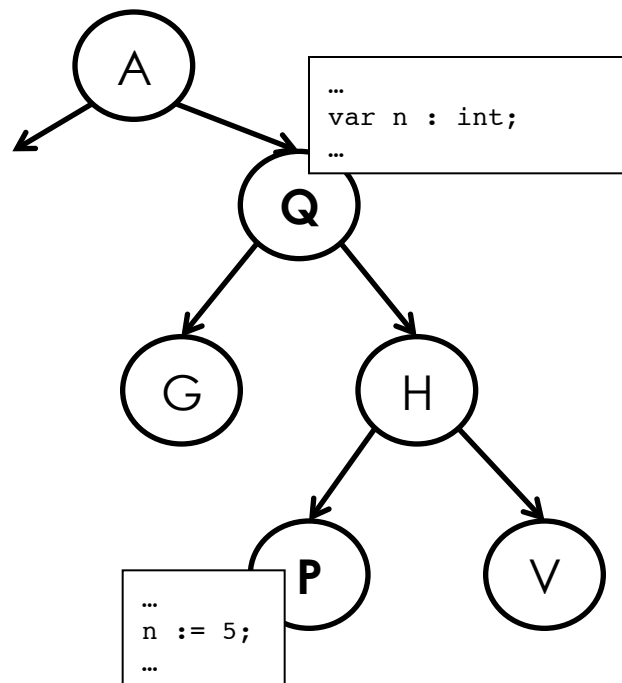
Non local references

- Suppose that a subprogram/block P is using a name n

- Define:

$$\#(P, n) = \text{depth}(P) - \text{depth}(\text{subprg./blk that declares } n)$$

- Example:



$$\#(P, n) = \text{depth}(P) - \text{depth}(Q) = 2$$

Non local references (cont.)

- Every non local reference n in the subprogram/block P is represented as

$$\langle x, y \rangle$$

- where
 - $x = \#(P, n)$
 - $y =$ position (offset) of n in the template of AR of the subprogram/block that declares n
- If $x=0$ then n is local and is compiled simply as y .

Non local references: Implementation

30

- Observation: given a subprogram P ,
 - the length of the static chain when P is executing is statically fixed
 - the non-local reference to a variable n is resolved always at the same point in the chain
- For the reason of efficiency, the static chain is often implemented as a vector (we call it **display**)
- The access to the identifier with the “coordinates” $\langle x, y \rangle$ is calculated as:

$$\text{display}[x] + y$$

- Cost: it is necessary to create the whole display all the times when the execution of subprogram starts (but often the HW machine gives the corresponding instructions)

Passing the parameters

- Let's assume:
 - dynamic local environment
 - static scoping
- Notation:
 - `proc P(x)` – `x` is a formal parameter
 - `call P(e)` – `e` is an actual parameter or an argument
- The formal parameters are treated as local variables (they are then allocated to the activation record).

- Example:

```
proc P(x)
    begin
        int y;
        ...
    end
```

The local variables are `x` and `y`.

Passing the parameters (cont.)

Notation `call P(x \leftarrow_{α} e)` means that

- P is declared as `proc P(x) ...`
- P is invoked as `call P(e)`
- α is type of passing the parameters

Value	<code>call P(x \leftarrow_{Val} e)</code>
Value-result	<code>call P(x $\leftarrow_{\text{Val-res}}$ y)</code>
Result	<code>call P(x \leftarrow_{Res} y)</code>
Reference	<code>call P(x \leftarrow_{Ref} y)</code>
Constant	<code>call P(x $\leftarrow_{\text{Const}}$ e)</code>
Name	<code>call P(x \leftarrow_{Name} e)</code>

Note: x, y are variables, e is an arithmetical expression

Passing by value

`call P(x \leftarrow_{val} e)`

- The expression e is evaluated in the environment of the caller
- In the AR of P the value e is assigned to the variable x

$$C \parallel \text{call } P(x \leftarrow_{\text{val}} e) \parallel_{rx} = C \parallel \text{Cmd} \parallel_{r's'}$$

where

$$l = \text{newmem } s$$

$$v = E \parallel e \parallel_{rs}$$

$$r' = [r_0, \dots, r_{\text{depth}(r, P)}, r_p] \text{ with } r_p(x) = l$$

$$s' = \text{updatemem}(s, l, v)$$

$$\text{Cmd} = r(P)$$

Note: x is local in P !

It is already implemented in crème CAraMeL.

Passing by value-result

`call P(x $\leftarrow_{\text{val-res}}$ y)`

- The value of y is evaluated in the environment of the caller
- this value is assigned to the local variable x in P
- when P terminates, the value of x is copied to the variable y of the caller

$C \parallel \text{call } P(x \leftarrow_{\text{val-res}} y) \parallel_{rs} = s''$ where

$v = E \parallel y \parallel_{rs}$

$l = \text{newmem } s$

$r' = [r_0, \dots, r_{\text{depth}(r,P)}, r_P]$ with $r_P(x) = l$

$s' = \text{updatemem}(s, l, v)$

$Cmd = r(P)$

$C \parallel Cmd \parallel_{r's'} = s'$

$s'' = \text{update}(s', \Lambda \parallel y \parallel_{rs}, s'(l))$

Passing by result

`call P(x \leftarrow_{Res} y)`

- when P terminates, x is copied to the variable y
- initial value of x is not specified
- the semantics is like in passing by value-result without the evaluation of y

Passing by reference

`call P(x \leftarrow_{Ref} y)`

- The location l of y is evaluated in the environment of the caller
- The location of x in P is set to l

$C \llbracket \text{call } P(x \leftarrow_{\text{Ref}} y) \rrbracket_{r_2} = C \llbracket \text{Cmd} \rrbracket_{r'_2}$ where

$$l = \Lambda \llbracket y \rrbracket_{r_2}$$

$$r' = [r_0, \dots, r_{\text{depth}(r, P)}, r_P] \text{ with } r_P(x) = l$$

$$\text{Cmd} = \tau(P)$$

Passing by constant

`call P(x $\leftarrow_{\text{const}}$ e)`

- The value of e is evaluated in the environment of the caller
- this value is assigned to the local variable x in P
- x cannot be assigned values in P
- It can be implemented in a similar way to the passing by reference

Passing by name

`call P(x \leftarrow_{Name} e)`

- create a new couple $\langle e, \mathfrak{r} \rangle$, where \mathfrak{r} is an environment of the caller
- every time when x should be evaluated, e is getting evaluated instead in the environment \mathfrak{r} and put instead of x .
- x cannot be assigned values in P