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** <*name, object*>
   - 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 <*name, object*> 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

$$P \Downarrow Q \texttt{ and } Q \Uparrow P$$

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.)

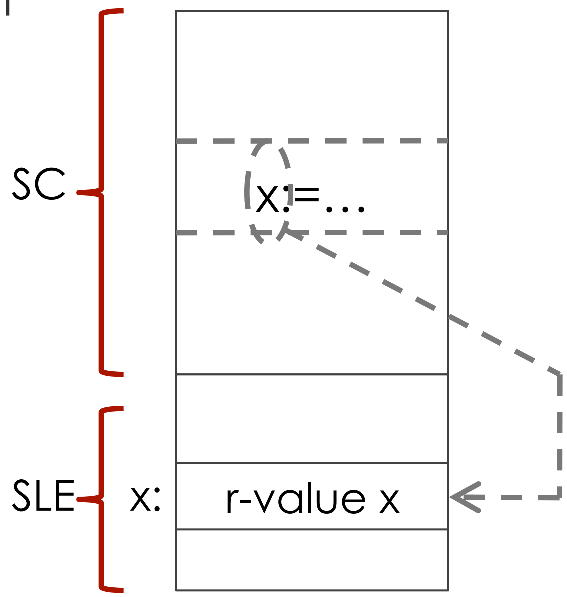
- Example: static option in C creates static local environment

```
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

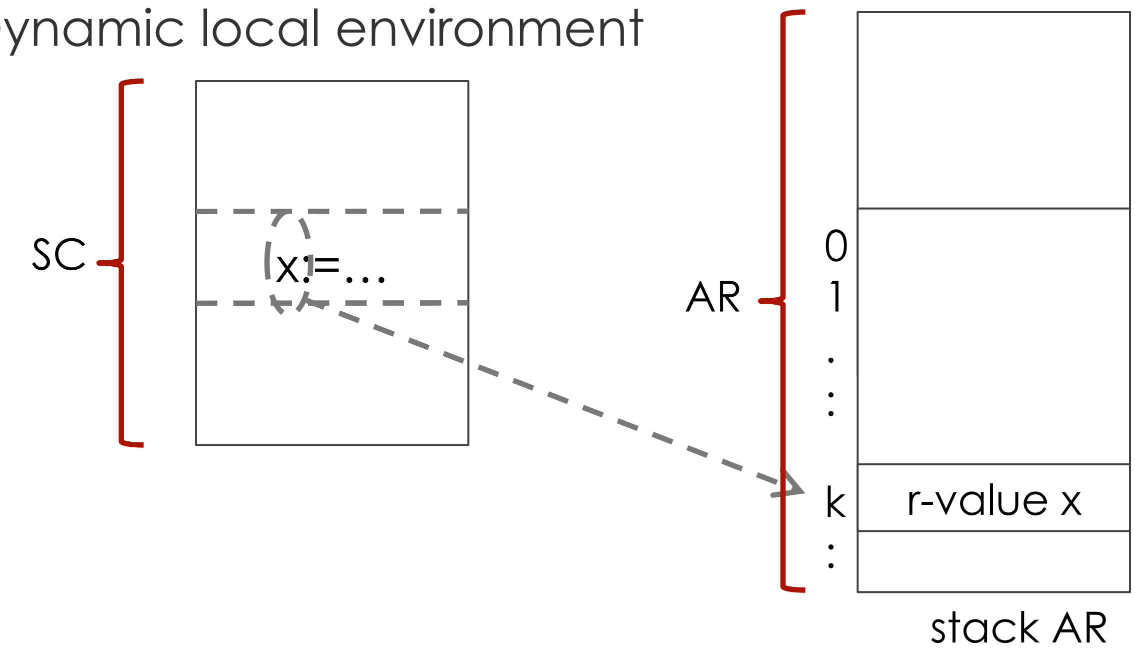
1. Static local environment



- 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

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
```

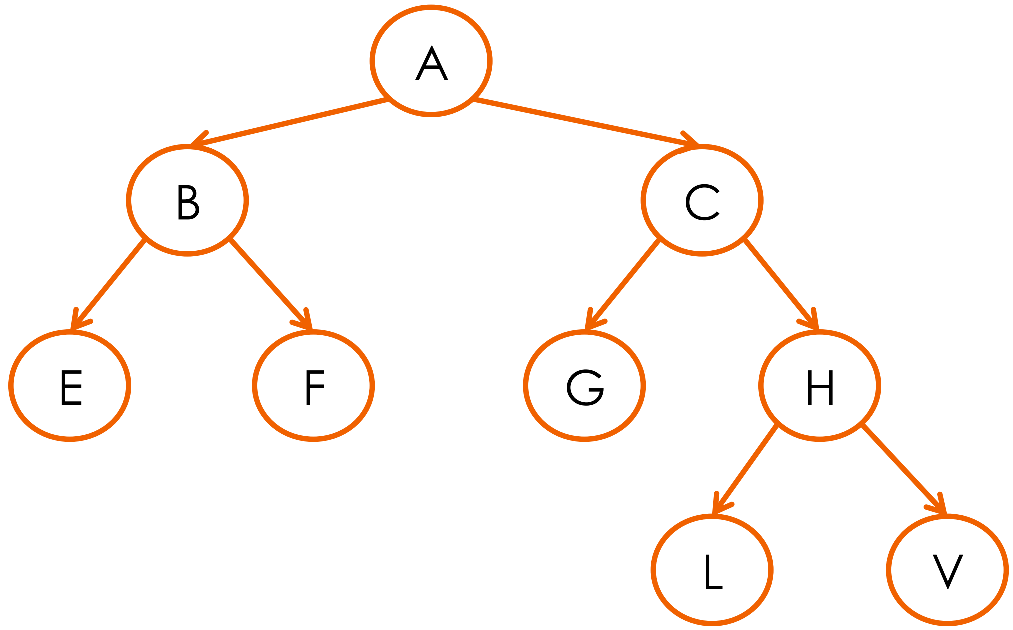
- If x is not local, which binding is used for x?
- Answer: **rules of scoping**
  - Dynamic scoping: rules of visibility are related to the execution (Lisp)
  - Static scoping: rules of visibility are related to the structure (syntax) of the program: it's the most used technique in the modern languages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

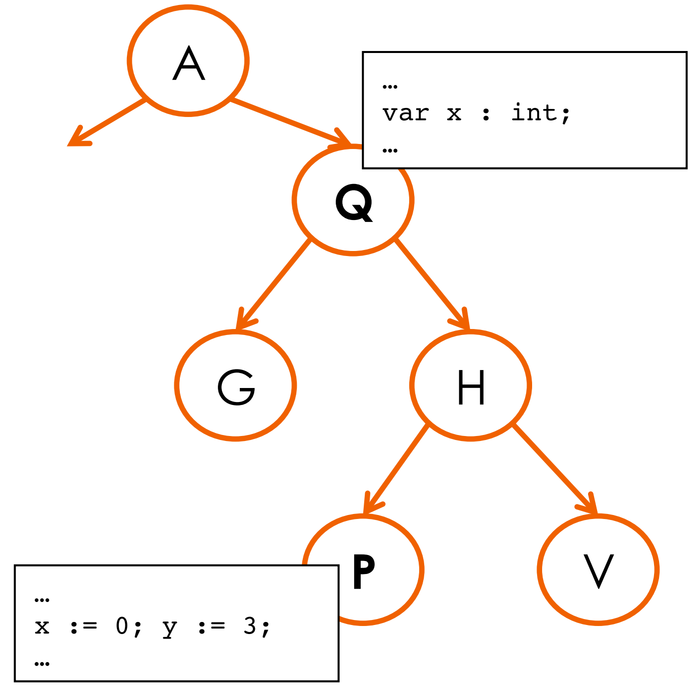
```
A: begin
    proc B;
      begin
        E: begin...end
        F: begin...end
      end {B}
    C: begin
          G: begin...end
          proc H;
            begin L: begin...end
                    V: begin...end
            end {H}
        end {C}
end {A}
```



Rule of static scoping

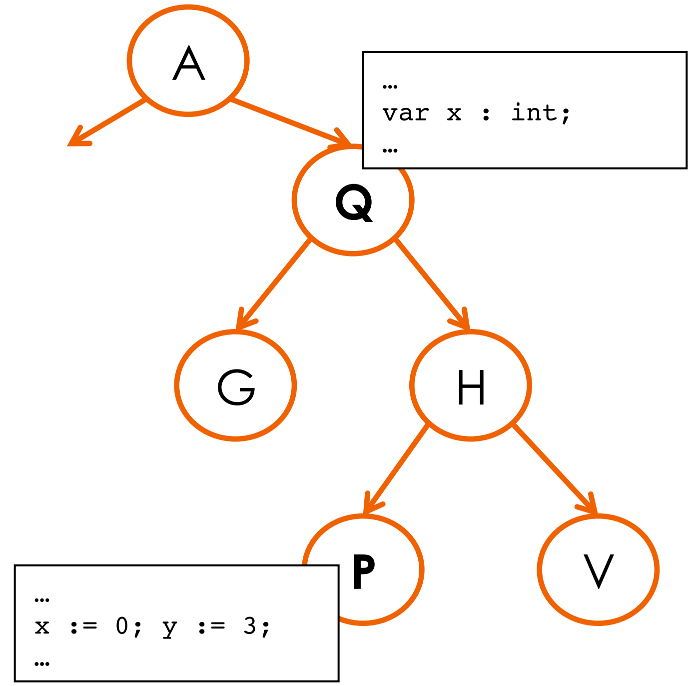
- If **x** occurs in non local reference in the subprogram/block *P*
  1. non local environment that provides correct binding for **x** is the parent Q nearest to P in which **x** is declared
  2. if there is no parent Q that declares **x**, the error is generated (this control is made at compile time)

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)



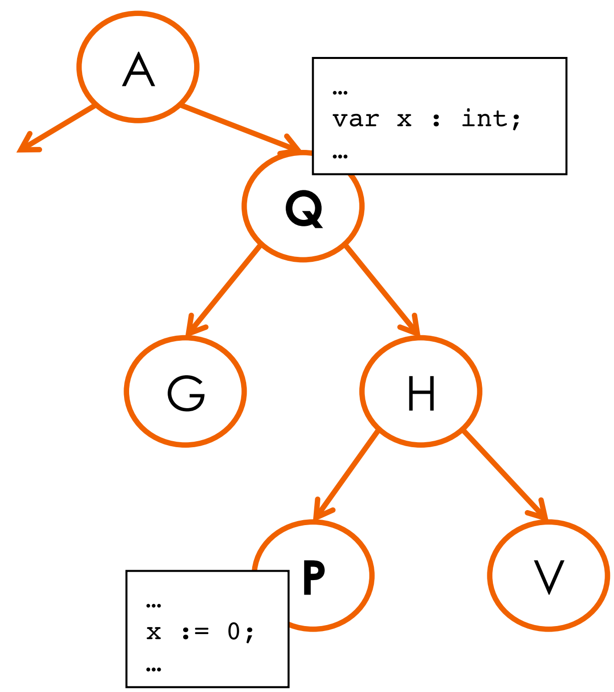
Static scoping: semantics

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

$$Env = List(Ide \rightarrow DVal) \qquad r = [r_0, r_1, \ldots, r_k]$$

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

- 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.)

$$D\|\texttt{const}\ \ v = n\ \|_{[\mathrm{r}_0,\mathrm{r}_1,\ldots,\mathrm{r}_k]\mathrm{s}} = [\mathrm{r}_0,\mathrm{r}_1,\ldots,\mathrm{r}_k{}']\,\mathrm{s}\quad \texttt{where :}$$

$$\mathrm{r}_k{}'(y) = \begin{cases} \mathrm{r}_k{}'(y) & \texttt{if}\quad y \neq v \\ n & \texttt{if}\quad y = v \end{cases}$$

$$D\|\texttt{var}\ \ v := n\ \|_{[\mathrm{r}_0,\mathrm{r}_1,\ldots,\mathrm{r}_k]\mathrm{s}} = [\mathrm{r}_0,\mathrm{r}_1,\ldots,\mathrm{r}_k{}']\,\mathrm{s}'\quad \texttt{where :}$$

$$\mathrm{r}_k{}'(y) = \begin{cases} \mathrm{r}_k{}'(y) & \texttt{if}\quad y \neq v \\ l & \texttt{if}\quad y = v \end{cases} \qquad \mathrm{s}'(x) = \begin{cases} \mathrm{s}(x) & \texttt{if}\quad x \neq l \\ n & \texttt{if}\quad x = l \end{cases}$$

where l=`(newmem s)` is a new location in s

$$D\|\texttt{proc}\ \ P = C\|_{[\mathrm{r}_0,\mathrm{r}_1,\ldots,\mathrm{r}_k]\mathrm{s}} = [\mathrm{r}_0,\mathrm{r}_1,\ldots,\mathrm{r}_k{}']\,\mathrm{s}\quad \texttt{where :}$$

$$\mathrm{r}_k{}'(y) = \begin{cases} \mathrm{r}_k{}'(y) & \texttt{if}\quad y \neq P \\ C & \texttt{if}\quad y = P \end{cases}$$

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.

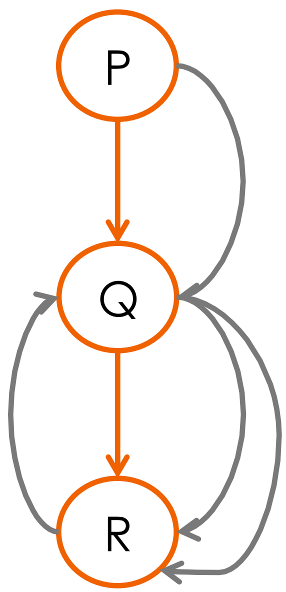
- **Solution**:
  - To each AR the **static chain pointer** (SCP) is added.
  - The "static" information on the syntactic structure (scoping tree) is implemented through the SCP.
  - Let's assume that a subprogram/block Q is a parent of subprogram/block P in the scoping tree. Then, the SCP of an AR of P points to AR of Q according to the rule of static scoping.

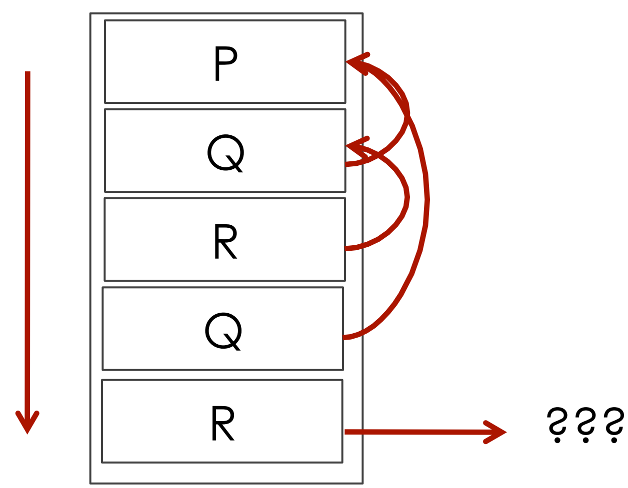
- Note: we consider the case of dynamic local environment.

Static scoping: implementation (cont.)

Suppose that $Q \Downarrow R$

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





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

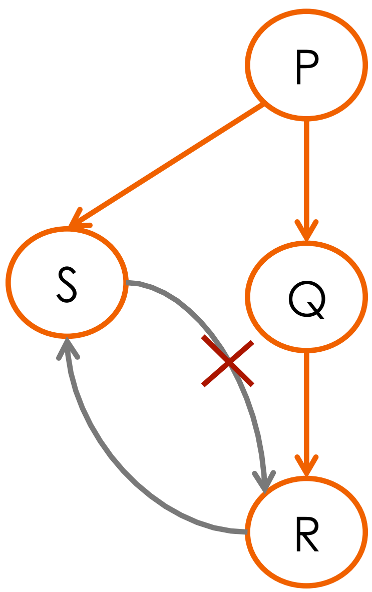
  (otherwise β would not be visible from α)

Algorithm to determine SCP

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

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

  (otherwise β would not be visible from α)



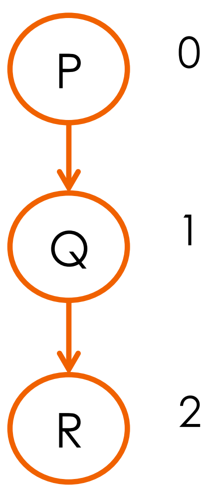

```
P: begin
    proc S; begin...end {S}
    proc Q;
      begin
        proc R; begin...end {R}
      end {Q}
end {P}
```

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 $\#(\alpha, \beta) =$ depth(α) – depth(parent(β))
- Example:



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

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

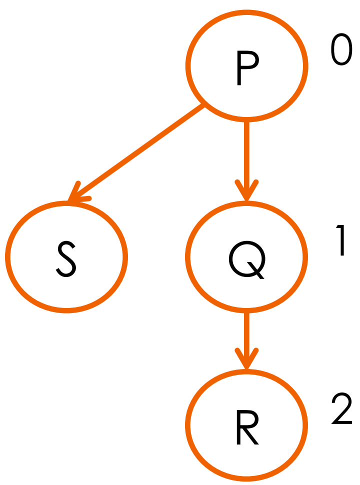
Algorithm to determine SCP

- 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

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

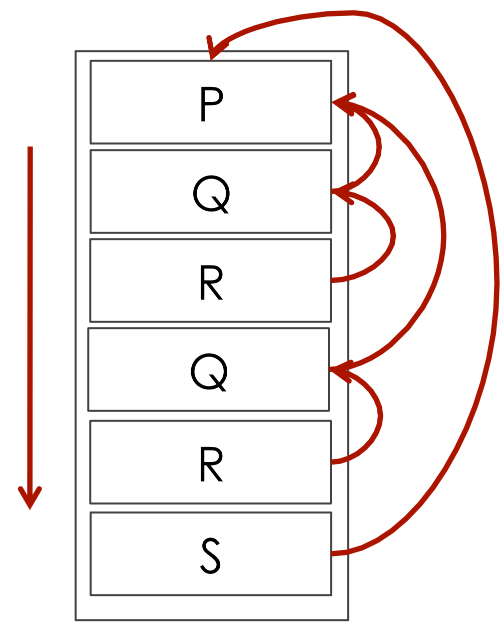


$\#(P, Q) = 0$

$\#(Q, R) = 0$

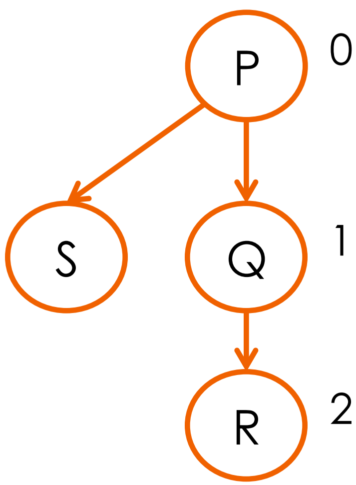
$\#(R, Q) = 2$

$\#(R, S) = 2$



Determining SCP: Examples

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

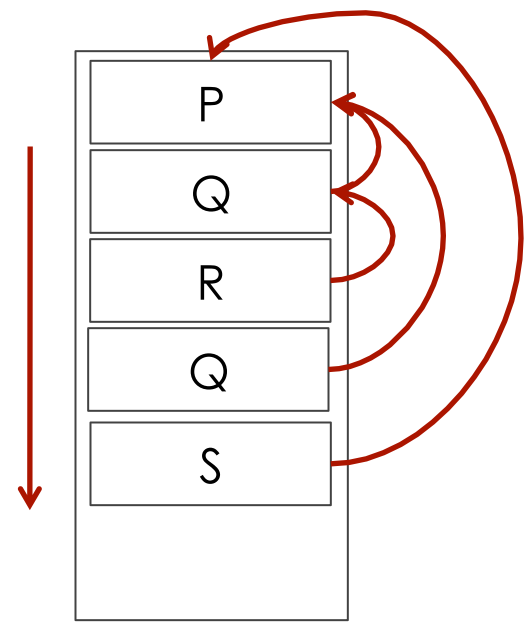


$\#(P, Q) = 0$

$\#(Q, R) = 0$

$\#(R, Q) = 2$

$\#(Q, S) = 1$



Calling a subprogram: semantics

- If $\mathtt{r} = [\mathtt{r}_0, \mathtt{r}_1, \ldots, \mathtt{r}_k]$, then

$$C\|\mathtt{call}\ P\ \|_{\mathtt{rs}} = C\|Cmd\|_{\mathtt{r's}}$$

where

- program P is declared as $\mathtt{proc}\ P = Cmd$
- $Cmd = \mathtt{r}(P) \in Com$
- $\mathtt{r}' = [\mathtt{r}_0, \mathtt{r}_1, \ldots, \mathtt{r}_h, \mathtt{r}_\varepsilon]$ where:

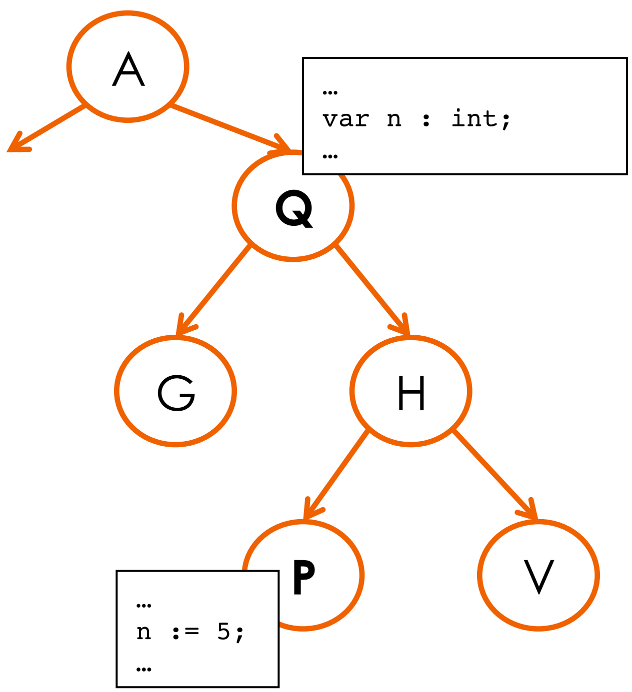
  $\mathtt{h}$ =depth$(\mathtt{r}, P)$, or $\mathtt{r}_h$ is "the deepest" environment where P is defined:
  - $\mathtt{r}_h(P)$ is defined,
  - $\mathtt{r}_{h+1}(P), \mathtt{r}_{h+2}(P), \ldots, \mathtt{r}_k(P)$ are not defined
- $\mathtt{r}_\varepsilon$ is a new local (empty) environment for Cmd

Non local references

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

$\#(P,\ n)$ = depth(P) – depth(subprg./blk that declares n)

- Example:



$\#(P,\ n)$ = depth(P) – depth(Q) = 2

Non local references (cont.)

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

$$<x, y>$$

- 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

- 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" $<x, y>$ is calculated as:

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

- 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` $\Leftarrow_\alpha$ `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	`call P(x` $\Leftarrow_{\texttt{Val}}$ `e)`
Value-result	`call P(x` $\Leftarrow_{\texttt{Val-res}}$ `y)`
Result	`call P(x` $\Leftarrow_{\texttt{Res}}$ `y)`
Reference	`call P(x` $\Leftarrow_{\texttt{Ref}}$ `y)`
Constant	`call P(x` $\Leftarrow_{\texttt{Const}}$ `e)`
Name	`call P(x` $\Leftarrow_{\texttt{Name}}$ `e)`

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

Passing by value

$$\texttt{call P(x} \Leftarrow_{\texttt{Val}} \texttt{e)}$$

- 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**

$C\|\texttt{call P(x} \Leftarrow_{\texttt{Val}} \texttt{e)}\|_{\texttt{rs}} = C\|Cmd\|_{\texttt{r's'}}$ where

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

$$\texttt{v} = E\ \|\texttt{e}\|_{\texttt{rs}}$$

$$\texttt{r}' = [\texttt{r}_0, \ldots, \texttt{r}_{\texttt{depth(r,P)}}, \texttt{r}_{\texttt{P}}] \texttt{ with } \texttt{r}_{\texttt{P}}(\texttt{x}) = l$$

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

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

Note: x is local in P!

It is already implemented in crème CAraMeL.

Passing by value-result

$$\mathtt{call\ P(x} \Leftarrow_{\mathtt{Val\text{-}res}} \mathtt{y)}$$

- 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

$C\|\mathtt{call\ P(x} \Leftarrow_{\mathtt{Val\text{-}res}} \mathtt{y)}\|_{\mathtt{rs}} = \mathtt{s}'''$ where

$$\mathtt{v} = E\,\|y\|_{\mathtt{rs}}$$

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

$$\mathtt{r}' = [\mathtt{r}_0, \ldots, \mathtt{r}_{\mathtt{depth(r,P)}}, \mathtt{r}_{\mathtt{P}}] \ \mathtt{with}\ \mathtt{r}_{\mathtt{P}}(\mathtt{x}) = l$$

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

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

$$C\|Cmd\|_{\mathtt{r's'}} = \mathtt{s}''$$

$$\mathtt{s}''' = \mathtt{update(s}'', \Lambda\|y\|_{\mathtt{rs}}, \mathtt{s}''(l))$$

Passing by result

$$\texttt{call P(x} \Leftarrow_{\texttt{Res}} \texttt{y)}$$

- 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

$$\texttt{call P(x} \Leftarrow_{\texttt{Ref}} \texttt{y)}$$

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

$$\mathcal{C}\|\texttt{call P(x} \Leftarrow_{\texttt{Ref}} \texttt{y)}\|_{\texttt{rs}} = \mathcal{C}\|\mathit{Cmd}\|_{\texttt{r's}} \text{ where}$$

$$l = \Lambda\|\texttt{y}\|_{\texttt{rs}}$$

$$\texttt{r}' = [\texttt{r}_0, \ldots, \texttt{r}_{\texttt{depth(r,P)}}, \texttt{r}_{\texttt{P}}] \texttt{ with } \texttt{r}_{\texttt{P}}(\texttt{x}) = l$$

$$\mathit{Cmd} = \texttt{r(P)}$$

Passing by constant

$$\texttt{call P(x} \Leftarrow_{\texttt{Const}} \texttt{e)}$$

- 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` $\Leftarrow_{\text{Name}}$ `e)`

- create a new couple `<e, r>`, where `r` is an environment of the caller
- every time when `x` should be evaluated, `e` is getting evaluated instead in the environment `r` and put instead of `x`.
- `x` cannot be assigned values in `P`