V22.0490.001
Special Topics: Programming Languages

B. Mishra
New York University.

Lecture # 17

—Slide 1—

Runtime Representations

- **Variable Names** $\Rightarrow$ *Environment* $\Rightarrow$ **L-values**
- **Scope, Extent & Binding Time**
  - **Scope:** *Portion of the program text* in which the identifier has a specific meaning.
  - **Extent:** *Duration* for which the identifier is allocated the location during execution (runtime).
  - **Binding Time:** *Time* at which the association is made between an identifier and its allocated location.
- Usually, Extent $\leq$ Scope.

—Slide 2—

Dangling Reference Problem

- Recall storage insecurity, when

$$\text{Extent} > \text{Scope}.$$

- Example

```
type r = record ... end;
     t = ^r;

procedure P;
  var q: t;
  procedure A;
    var s: t;
    begin
      new(s); q:= s; dispose(s);
    end;

  begin
    ... A ...;
    q := ...; (*L-value whose lifetime has passed*)
  end.
```

—Slide 3—

Static Storage Management

- FORTRAN (as implemented commonly)
  - The main program and each subroutine may declare local data.

    But all data are preserved across successive calls on subroutines.
  - A given subroutine cannot be called if there is an as yet unfinished call of that same subroutine.

    Forbids both direct and indirect (mutual) recursion.
- **All storage for FORTRAN data can be allocated statically before execution begins.**

—Slide 4—

Activation Record

- **Activation Record:**

  Set of informations necessary for the execution of a subprogram.

- FORTRAN

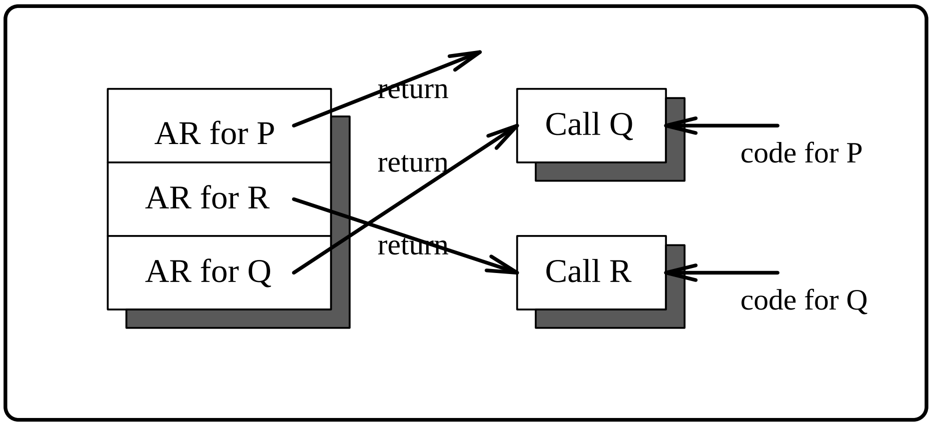
  When a subroutine executes, it always finds its activation record in the same place.

- *Runtime structure of FORTRAN*:

  Needs only to store the “return address” in the activation record of the called subroutine.

—Slide 5—

Static Storage Management in FORTRAN



- **Activation Records:**

```
SUBROUTINE P(...)      SUBROUTINE Q(...)      SUBROUTINE R(...)
  ...                    ...                    ...
  CALL Q(...)            CALL R(...)            ...
  ...                    ...                    RETURN
  RETURN                 RETURN                 END
  END                    END
```

- In FORTRAN, each of the following has an *Activation Record*
  - Each subroutine
  - The mainprogram
  - Each `COMMON` block

—Slide 6—

Stack Based Modern Languages (ALGOL-like)

- *Subprograms are allowed to be recursive.*
  - More than one activation of the same subprogram can exist simultaneously.
  - Each invocation of the subprogram may have
    1) A different return point
    2) Different values for local variables.
  - Number of activations of a subprogram (that can exist simultaneously) is **unpredictable**.

  $\Rightarrow$ *The activation record for a subprogram can only be created, when the subprogram is actually called.*

- *Support for scope-entry declaration of local variables.*

—Slide 7—

Implications of Call-Time Allocation of AR's

- Allocation lasts for precisely the duration of a particular subprograms execution.
  - ⇒ Allocate AR, when the execution begins.
  - ⇒ Release that space, when execution finishes.
- If a subprogram `P` calls a subprogram `Q` then `P` *cannot complete before* `Q`.

  $$\text{Extent}(\texttt{Q}\text{'s AR}) \subset \text{Extent}(\texttt{P}\text{'s AR})$$

  `Q`'s extent is wholly contained in `P`'s.
- *Storage requirements of AR's are Last-In-First-Out.*
- Stack-like data structure for AR's suffices.

—Slide 8—

Procedure Call

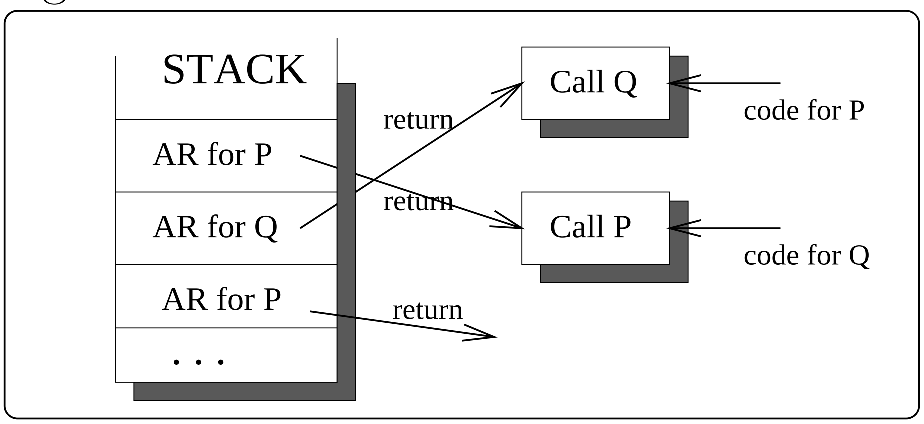
$P \equiv$ Procedure
Call to P

- Push a new AR for P on stack (containing "return address" as its return field)
- Execute in the "new" environment
- Pop the current AR for P (saving the "return address" in T)
- Go to T.

—Slide 9—

Activation Records (AR)
Stack Based Storage Management

Algol and its relatives:



- **Mutually Recursive Procedures:**

```
procedure P(...)<--      procedure Q(...)
  ...              \        ...
  begin             \       begin
    ...              \        ...
    Q(...)            -----   P(...)
    ...                  ->   ...
  end ------------------/    end
```

- Each Activation of a procedure has an AR (allocated dynamically).

—Slide 10—

Up-Level Addressing and the Display

- In the absence of reference to "global variables" (procedures reference only *formal* and *locally declared variables*)

$$\text{L-VALUE OF A VARIABLE} \equiv \begin{cases} \text{ADDRESS OF THE CURRENT AR} \\ + \\ \text{"OFFSET" ADDRESS WITHIN THE AR} \end{cases}$$

- **Procedure Call**:
  - Allocate AR on top of the stack
  - Save caller's AR address in its own AR
- **Return from Call**:
  - Restore the old AR address
  - Branch to the return point

—Slide 11—

Reference to the Global Variable

- **Reference to the Global Variables**:
  Simple Case Global variables are all declared in the main program—OUTER-MOST LEVEL.
- Each variable reference is either:
  - Relative to the current AR (for locals), or
  - Relative to the stack base (for globals).
- **What about the intermediate non-local variables?**:
  Up-Level Addressing Problem

—Slide 12—

Up-level Addressing Problem
Intermediate Non-Local Variables

Algol, Pascal, Ada, — Scope Rules

```
procedure P;
  begin
  var x, y: T1;

  procedure Q;
  begin
    var z: T2;

    procedure R;
      begin
      var: a, b: T3;
        ...       <----------z is accessible in Q & R
      end;
      ...         <----------x, y are accessible in P, Q & R
  end;
  ...
  end;
```

- Can `R` tell where `x`, `y` and `z` are located?
- *Not easily*
  (specially, if `R` calls itself recursively.)

—Slide 13—

Displays

- **Lexical Level of a Procedure**:

  An integer value one greater than the lexical level of the procedure in which it is declared.

- Lexical level of the main Program = 0.
  $LexLev(P) = n \Rightarrow$

  $$LexLev(Q) = n + 1 \;\&\; LexLev(R) = n + 2.$$

- **Up-level Addressing**
  Accessing an "intermediate non-local variable" at level L, where $0 < L <$ Current-level.

- **Note**:
  The number of AR's accessible to procedure `R` = $Lexlev(R) + 1$. (Dictated by the static nature of the lexical scope rule.)

—Slide 14—

Displays and Setting Them Up

- **Solving the Up-level Addressing Problem**:
  Add $LexLev(R) + 1$ *locations to the AR of* `R`.
  These locations are called a **DISPLAY**—Vector of pointers to accessible AR's
- **Setting the displays**:
  Assumption: No procedure parameters (False in Ada)
- Procedure P calls procedure Q:
  Assumption $\Rightarrow LexLev(Q) \leq Lexlev(P)$
  1. $LexLev(P) = n$ then $Display(P) = D_P[0..n]$
  2. Two cases to consider:
     1) $Lexlev(Q) = n$ (P and Q are at the same level) and
     2) $Lexlev(Q) = m < n$ (Q is up-level from P)

—Slide 15—

Setting the Displays

- Case I) $Lexlev(Q) = n$ (P and Q are at the same level)

$$\begin{aligned} Display(Q) &= D_Q[0..n] \\ D_Q[0..n-1] &:= D_p[0..n-1] \\ &\quad\quad \text{First } n \text{ displays are the same} \\ D_Q[n] &:= AR_Q \\ &\quad\quad \text{The } n\text{th display of} \\ &\quad\quad Q \text{ is the base of the current AR} \end{aligned}$$

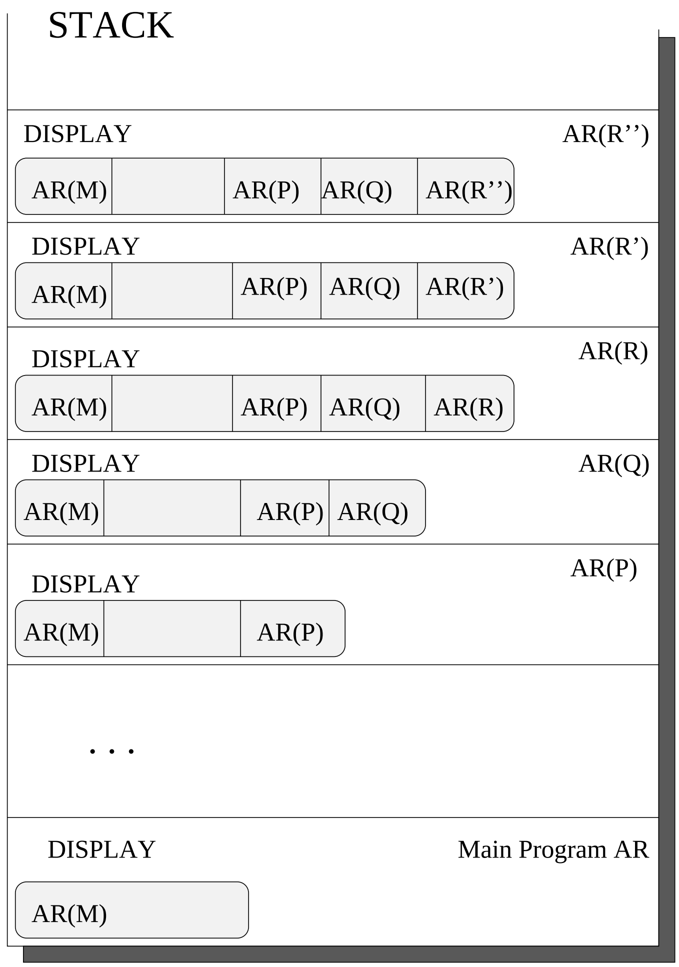
- Case II) $Lexlev(Q) = m < n$ (Q is up-level from P)

$$\begin{aligned} Display(Q) &= D_Q[0..m] \\ D_Q[0..m-1] &:= D_p[0..m-1] \\ &\quad\quad \text{First } m \text{ displays are the same} \\ D_Q[m] &:= AR_Q \\ &\quad\quad \text{The } m\text{th display of} \\ &\quad\quad Q \text{ is the base of the current AR} \end{aligned}$$

- *More Efficient Implementation*
  1) Maintain one vector common to all AR's + 2) One additional word for each AR.

—Slide 16—

Sample Display Configuration



[End of Lecture #17]