G22.2210

Programming Languages: PL

B. Mishra
New York University.

Lecture # 7
October 23 1995

—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

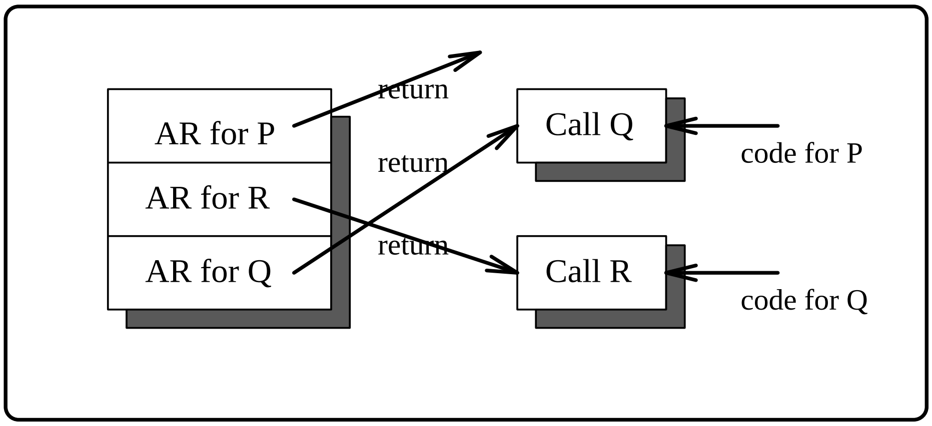
  When a subroutine executes, it always finds its activation record in the same place.

- *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

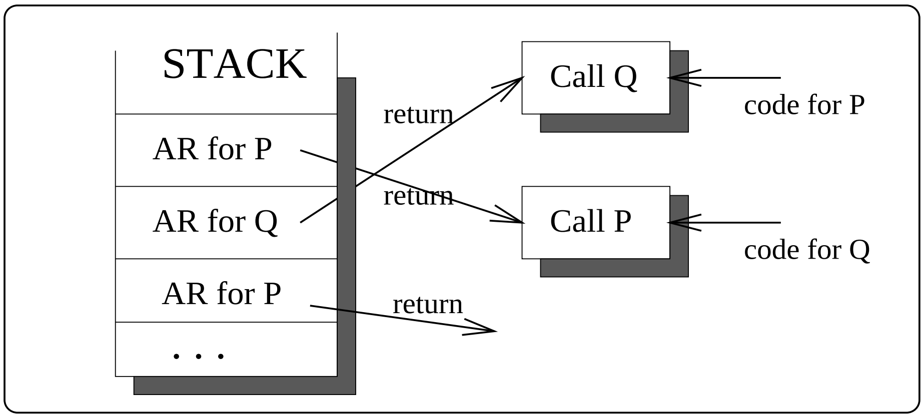
$P \equiv$ Procedure
Call to P

- 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)

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

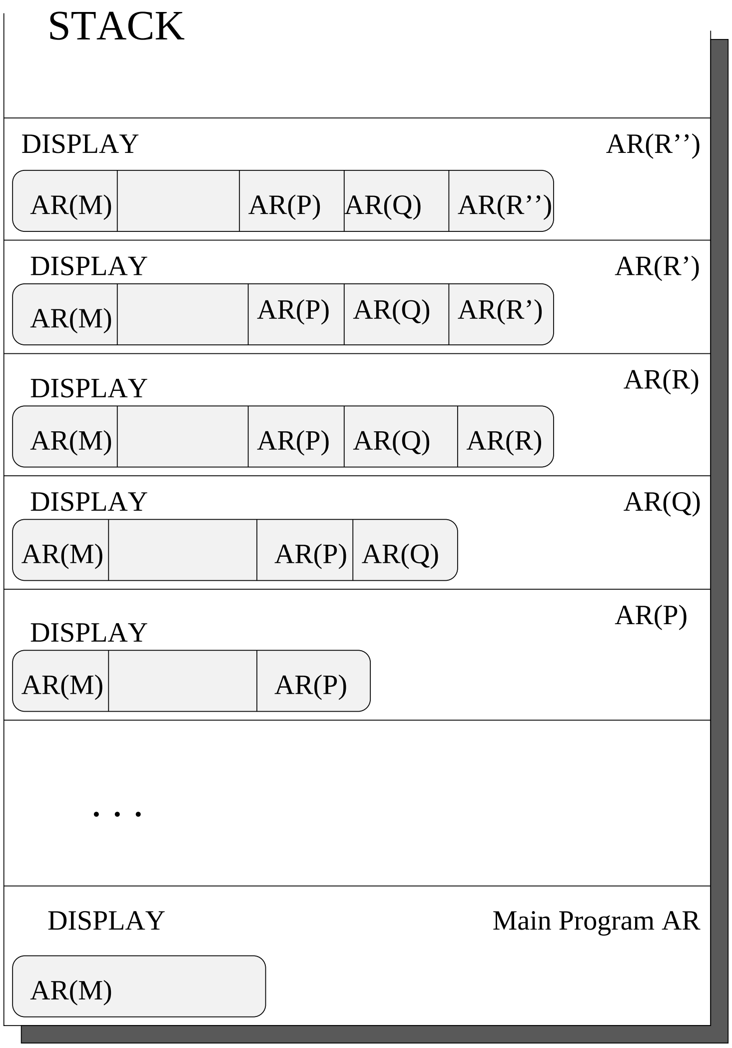
- Case II) $Lexlev(Q) = m < n$ (Q is up-level from P)

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

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

—Slide 16—

Sample Display Configuration



—Slide 17—

Displays with Procedure as Parameters

```
procedure Q(procedure F);           {LexLev(Q) = n}
  begin ... F(x) ... end;

procedure P;                        {LexLev(P) = n}
  var a, b: T;
  procedure R(y: T); begin ... a ... end;
  begin
  ... P; ... Q(R); ...              {LexLev(R) = n+1}
  end;
{ Q calls R --- LexLev(R) > LexLev(Q) }
```

- `P` calls itself recursively
- Eventually, `P` calls `Q` and passes `R` as a parameter.
- `P` and `R` have access to `a` and `b`; but not `Q`.
- When `Q` calls `R`, `R` cannot establish access to `a` and `b` from the display of `Q`.

—Slide 18—

Procedure Parameters (Contd.)

- In order to create the display of `R`, `P` must pass **two things**
  - Address of `R`, and
  - Current environment in the form of $AR(P)$.
- Initial display elements of `R` are established from `P`'s display (not `Q`'s).

—Slide 19—

Allocation of Assignable Data Types

- **Static Arrays:** (FORTRAN, Pascal)
  – Size of the arrays is fixed at the compile time

  – Procedure with local arrays
  – Has a fixed size AR. Local arrays are allocated on the stack.
- **Non-Static Arrays:** (Algol, PL/I)
  – Bounds of an array can be determined at run time

  – Procedure with local non-static arrays
  – Has a variable size AR, whose size can be determined at run-time
- But in order for the array bounds to be computed, the AR must have already been established. The expression computing the bounds may involve function calls:

  (e.g., `var A: array[0..Fib(N)] of integer;`)

—Slide 20—

AR for Non-Static Arrays

- **Descriptor for the array in AR**
  – Has place holders for array bound values
  – Descriptor size can be computed at compile time

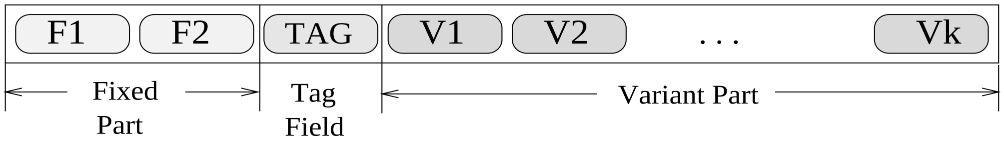
- **Steps in Creating AR**

  1. Establish a "partial" activation record with descriptor allocated.
  2. Compute the bounds for the descriptors filling in the place-holders.
     — Each bound computation may trigger other procedure calls, but as each bound is evaluated, the stack returns to environment of the AR under consideration —
  3. When all bounds are computed the sizes of the arrays are known. The activation record is "extended" by allocating the arrays on top of the stack.
  4. The procedure execution begins.
  5. When procedure returns the allocated local array disappears along with the AR.

—Slide 21—

Variant Records

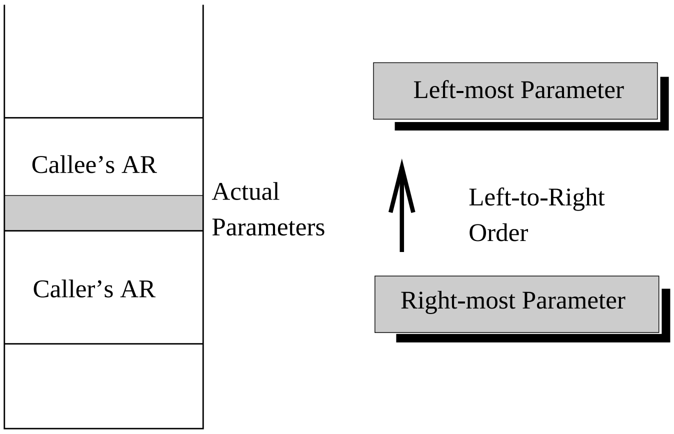
- Two Parts:
  - *Fixed Fields*
  - *Variant Fields*
- The field layouts and hence the size of a variant record depends on — *the value of the tag*
- Allocate enough space on the stack so that the variant part is able to hold the fields in the largest part



—Slide 22—

Parameter Passing

- Parameter Passing:
  - Call-by-Value
  - Call-by-Reference
  - Call-by-Value/Result
  - Call-by-Name
- Runtime Representation:
  - Caller: Stores the actual parameters
  - Callee: Uses the parameters in the body
- Actual Parameter List = Contiguous area on top of the stack at the very front of the callee's AR.



—Slide 23—

Evaluation of Actual Parameter List

- **Assume** Number of parameters to a given procedure is fixed at compile time.
- *Steps*
  -1- As each actual parameter is evaluated, it is pushed on the stack. (Evaluation proceeding from left-to-right order — possibly triggering other calls)
  -2- Computation proceeds at top of the stack. Parameters already computed are not disturbed
  -3- When callee receives control, its actuals are on top of the stack & are treated as part of the callee's AR
- *Variable number of parameters*: The callee must know how many parameters is received. Usually kept in an extra word on top of the stack.

—Slide 24—

Call-By-Value

- Actuals are evaluated in the current context (using caller's AR).
- Computed values are pushed onto the stack in left-to-right order. They become part of the callee's AR
- These values may be modified by the callee.

—Slide 25—

Call-By-Reference

- The addresses of the actuals are computed in the caller's AR.
- The addresses are placed on the stack in left-to-right order in the callee's AR
- The callee references these parameters *indirectly* through the appropriate parameter list location
- **Note:** The address is computed by the caller in the current context prior to the call and remains fixed during the call

  (e.g. `Add(A[i,j], A[i+1, j+1])`)

  A change to `i` and/or `j` by the callee does not change the addresses of the actuals, `A[i,j]` or `A[i+1,j+1]`.

—Slide 26—

Call-By-Value/Result

- Actual parameters are passed as in call-by-reference.
- The callee copies the values of formals into local variables in its AR.
- The callee executes its body, while accessing only the local variables.
- Before returning the control, the callee copies the values of local variables back into actual parameter locations.
- *Alternatively*, the caller passes the parameters as for call-by-value and copies the final values back after the callee finishes.

—Slide 27—

Call-By-Name [Algol60]

- **Two Problems**
  - Evaluation occurs in the environment of the caller (not callee)
  - Necessary to change the environments to that of the caller every time a name parameter is accessed.
  - High Overhead — in representation and procedure invocation.
- **Main Idea**
  Pass a function (not a value or location) as the actual parameter:

$$THUNK\colon \emptyset \rightarrow L\text{-}values$$

– Takes no parameter itself
– Delivers the address of its corresponding actual in the environment of the caller

—Slide 28—

Thunks

Note

- Thunks don't exist in the source. Generated at the call site by the compiler
- Thunks inherit the environment of the caller as is they were declared local to the caller
- Each time callee needs an actual, it invokes the thunk and uses the L-value returned to access the actual name parameter.

—Slide 29—

Dynamic Storage

```
type ref = ^T;
var  x   : ref;

procedure P; begin ... new(x) ... end;
```

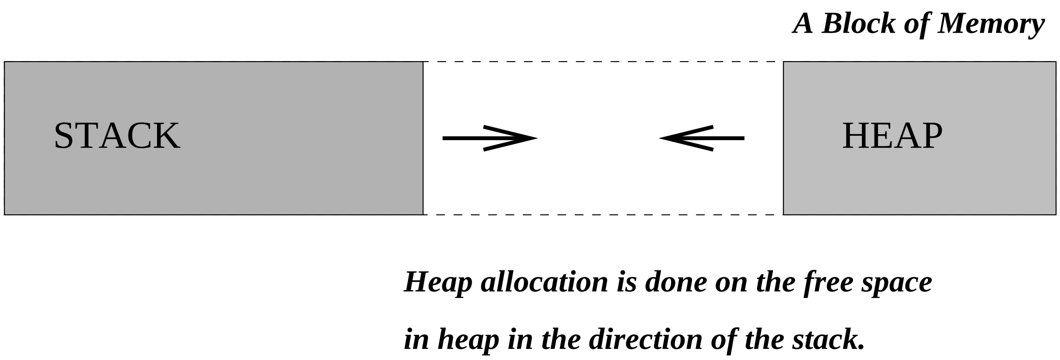
- P allocates an object A (anonymous)
- Assigns the L-value of A to the non-local variable x.
- Object A must outlive the activation of P & hence, must not reside P's activation record

  (If A is allocated on the stack and outlives P, it leaves a hole in the stack).

—Last Slide—

Heap Allocation

Storage whose extent is not tied to a particular scope cannot be allocated on the stack.



- Collision: Error "`Stack overruns heap.`"
- Garbage Collection

[End of Lecture #7]