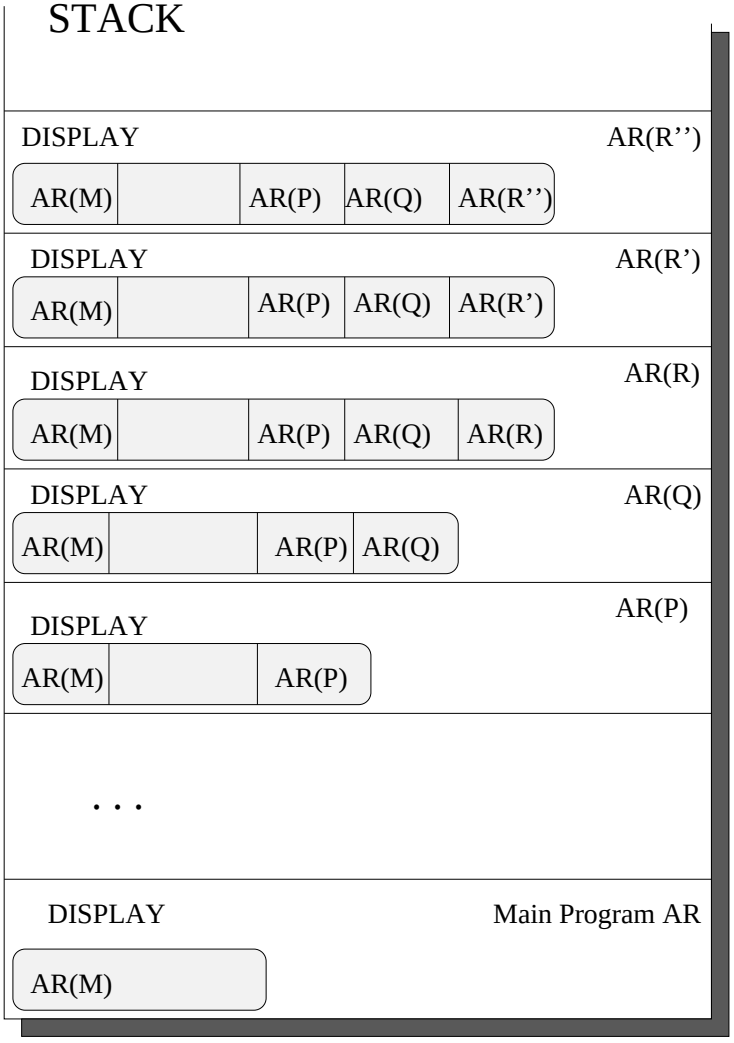
V22.0490.001
Special Topics: Programming Languages

B. Mishra
New York University.

Lecture # 18

—Slide 1—

Sample Display Configuration



—Slide 2—

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 3—

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 4—

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 5—

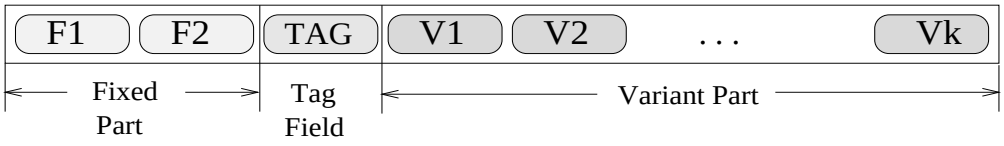
AR for Non-Static Arrays

- **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 6—

Variant Records

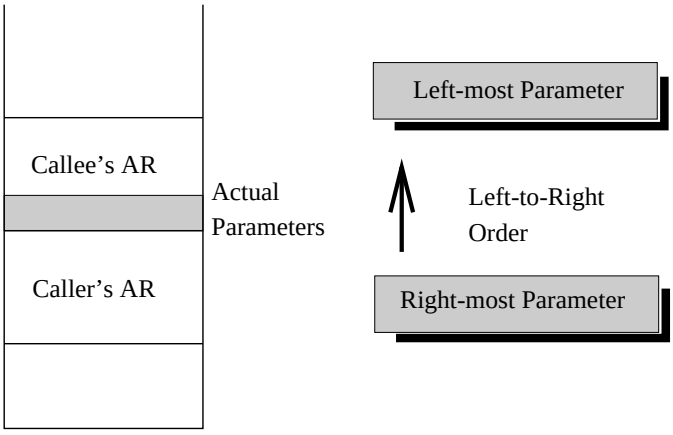
- Two Parts:
  - *Fixed Fields*
  - *Variant Fields*
- The field layouts and hence the size of a variant record depends on — *the value of the tag*
- Allocate enough space on the stack so that the variant part is able to hold the fields in the largest part



—Slide 7—

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 8—

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 9—

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 10—

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 11—

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 12—

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:

$$THUNK: \emptyset \rightarrow \textit{L-values}$$

– Takes no parameter itself

– Delivers the address of its corresponding actual in the environment of the caller

—Slide 13—

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 14—

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

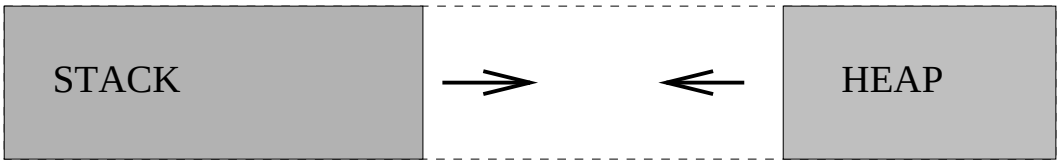
  (If A is allocated on the stack and outlives P, it leaves a hole in the stack).

—Last Slide—

Heap Allocation

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

A Block of Memory



Heap allocation is done on the free space in heap in the direction of the stack.

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

[End of Lecture #18]