G22.2210

Programming Languages: PL

B. Mishra
New York University.

Lecture # 6

—Slide 1—

Scope Issues

- Those features which describe and control the use of named entities such as:
  variables, *procedures* and *types*
- **Scope**, **Extent** & **Range**
  3 Related Concepts
- A *variable* consists of a **name**, an **object** (Location or L-value) & **value** (R-value).
- **Environment**: name $\mapsto$ object
- **Store**: object $\mapsto$ value
- Example: Fortran, Algol 60, Pascal, Alphard, Ada, `C`, ...
- Exception: LISP, CLU, Algol 68, ...

—Slide 2—

Scope

- *The scope of a name is*
  the portion of the program text in which
  all uses of that name has the same meaning:
- *Scope* $\Rightarrow$
  The name denotes the same object.
- *What about a pointer variable*?
- **Note**:

  A pointer variable is a name denoting a single object whose value is a reference to another object.

—Slide 3—

Extent

- **Lifetime** of an object
- *The portion of the execution time of the program during which the value contained in the object persists unless explicitly changed.*
- E.g., Extent of a local variable in Algol 60 = The period between entry and exit of the block in which it is declared.

—Slide 4—

Range

- *A range is a portion of a program, delimited by some construct of the language, such that the scopes of a name defined inside the program portion do not extend outside that portion, unless explicitly* exported.
- *Ranges are building blocks* out of which scopes are constructed.
- **Examples**: Procedures, Blocks (C, Algol 60), Modules (Ada).
- **Note**: Ranges never overlap.
- When one range is nested within another, the outer range leaves off where the inner range begins.

—Slide 5—

Dynamic and Static Ranges

- A context of a range consists of two parts:
  1. **Static Part**: The name environment in which the range is **declared**—*Lexical Environment*.
  2. **Dynamic Part**: The name environment in which the range is **invoked**—*Calling Environment*.
- **Lexical Scope Rule**

  Free or nonlocal variables are given name bindings in the static context of a range.
- **Dynamic Scope Rule**

  Free or nonlocal variables are given name bindings in the dynamic context of a range.

—Slide 6—

Example 1

```
var i, k: integer;

procedure P(var j: integer);
  var i: integer;
  begin i := 1; Q; j := i end;

procedure Q;
  begin i := i+1 end;

begin
  i := 3; P(k); write(k)
end.
```

- What does the program print?
- **Lexical** $\Rightarrow 1$
- **Dynamic** $\Rightarrow 2$

—Slide 7—

Example 2

```
var i: integer;

function GLOP(function Q: integer,
            lower, upper: integer): integer;
  var i,S: integer;
  begin
    S := 0;
    for i := lower to upper do S := S + Q;
    GLOP := S
  end;

function A;
  begin A := i*i end;

begin
  i := 0; write(GLOP(A,1,3))
end.
```

- What does the program print?
- **Lexical** $\Rightarrow 0$
- **Dynamic** $\Rightarrow 1 + 4 + 9 = 14$

—Slide 8—

Procedures and Functions

- In an imperative language, **functions** return values, and **procedures** do not. *Functions* are abstraction of *expressions*—*Procedures* are abstraction of *commands*.
- It is often desirable that functions do not have any side-effect.
- Function has **4 parts**: *name*, *formal parameters*, *result type* and *body*.

```
int succ(int i){
   return (i+1)%size;
}
```

```
function succ(i: in INTEGER)
            return INTEGER is
begin
  return (i+1 mod size);
end succ;
```


—Slide 9—

Parameter Passing Methods

- *Procedure Invocation*: Statements in the body are executed as if they appeared at the point of call.
- **Correspondence between the actual parameters (at call site) & the formal parameters (in the body).**
- Various Calling Mechanisms:
  1. CALL-BY-VALUE
  2. CALL-BY-REFERENCE
  3. CALL-BY-VALUE-RESULT
  4. CALL-BY-NAME
  5. CALL-BY-NEED

—Slide 10—

Calling Mechanisms

- CALL-BY-VALUE: *Pass the R-value.*
  value(Formal) = Store(Environment(Actual))
  ... ⟨ Procedure Body ⟩
- CALL-BY-REFERENCE: *Pass the L-value.*
  Location(Formal) = Environment(Actual)
  ... ⟨ Procedure Body ⟩
  Since actuals and formals share the L-values, the values of actual can be modified after the procedure call.
- CALL-BY-VALUE-RESULT: *Pass the R-value. Save the L-value. After the call, update.*
  value(Formal) = Store(Environment(Actual))
  ... ⟨ Procedure Body ⟩
  value(Actual) = Store(Environment(Formal))
- CALL-BY-NAME: *Pass the Environment.*
  Environment(Formal) = Environment(Actual)
  ... ⟨ Procedure Body ⟩
  The expression in the actual parameter position is reevaluated each time the formal parameter is used.

—Slide 11—

Variations

- COPY-OUT : Also, called call-by-result...Converse of call-by-value. A variable local to the procedure is created and at the procedure termination, its value is copied back to the corresponding actual. The actual argument must be an L-valued expression.
- CALL-BY-NEED: Evaluation of the actual is deferred until the value of the formal parameter is first needed. This value is saved for any subsequent uses of the formal parameter. In the absence of side-effects, it is same as call-by-name.
- Ada: It has three modes: `in`, `out` and `in out`. The compiler can implement these with *call-by-value-result* (copy-in/copy-out) or *call-by-reference*, the choice based on efficiency considerations.
  - `in`: Used for passing information to the **callee**.
  - `out`: Used for passing information to the **caller**.
  - `in out`: Information is passed in and back out through the same parameter.

```
procedure USE(X: in INTEGER) is ...
procedure GENERATE(X: out INTEGER) is ...
procedure MODIFY(X: in out INTEGER) is ...
```

—Slide 12—

Example

```
var i: integer;
A: array[1..3] of integer;

procedure testbind( <binding> f, g: integer);
begin
   g := g + 1;
   f := 5 * i;       (* i= nonlocal)
end;

begin
  for i := 1 to 3 do A[i] := i;
  i := 2;
  testbind(A[i], i);
  print(i, A[1], A[2], A[3]);
end;
```


—Slide 13—

Example (contd)

- **CALL-BY-VALUE**: Actuals are unaffected.

  `i = 2;          A = (1, 2, 3);`

- **CALL-BY-REFERENCE**:
  Actuals and nonlocals are affected

  `i = 3;          A = (1, 15, 3);`

- **CALL-BY-VALUE-RESULT**: Actuals are affected

  `i = 3;          A = (1, 10, 3);`

- **CALL-BY-NAME**:
  Actuals and their environments are affected

  `i = 3;          A = (1, 2, 15);`

- **Note**: *If the actual parameter is a simple identifier, then the effect of call-by-name is same as call-by-reference.*
- However, if the actual is an expression or a selector expression, and the parameters depend on each other, then the procedure may get different L-values each time it accesses the parameters.

—Slide 14—

Macros: Text Substitution

- Inline Expansion in Ada, Macro in `C`, ...
- Macro Processor:
  1. Actual parameters are textually substituted for the formals.
  2. Resulting procedure is textually substituted for the call.
- `#define` compiler directive in `C`

```
#define SQR(x)  ((x)*(x))

main(){
  int a, b=2;

  a = SQR(b);          /* ((b)*(b))     */
  a = SQR(b++);        /* ((b++)*(b++)) */
}
```


—Slide 15—

Macros (contd)

- Scope rules and the syntax of the language are ignored. Macro processors ignore the “naming conflicts” that arise during substitution.
- Treating `testbind` as a macro, we get:

```
for i := 1 to 3 do A[i] := i;
i := 2;
i := i + 1;
A[i] := 5 * i;       (* i is captured *)
print(i, A[1], A[2], A[3]);
```

—Slide 16—

Subprograms in Ada

- *Functions & Procedures*
  Side-effects (assignments to nonlocal variables) are allowed.

```
function <name>(<parameters>) return <result-type> is
   --Local declarations
begin
   --Local statements
end;

procedure <name>(<parameters>) is
   --Local declarations
begin
   --Local statements
end;
```

- A function can only have `in` (copy-in or call-by-value) parameters.
- A procedure can have `in` (copy-in or call-by-value), `out` (copy-out or call-by-result) or `in out` parameters.
- Default mode is `in`.

—Slide 17—

Example

- An argument corresponding to `in` parameter can be an R-value (e.g., expression).
- An argument corresponding to `out` or `in out` parameter must have an L-value (e.g., variable, selector).
- **Note**:

  `in` Parameter acts as a **local constant**.

  `out` Parameter acts as a **local variable**—whose value is assigned to the corresponding actual parameter.

  `in out` Parameter acts as a **local variable**—permits access and assignment to the corresponding actual parameter.
- **Example**:

```
procedure COMPUTE_RISE(V_X, V_Y, DIST: in FLOAT;
                       RISE: out FLOAT) is
   TIME: FLOAT;
begin
   TIME := DIST/V_X;
   RISE := V_Y * TIME - (G/2.0) * (TIME**2);
end;
```


—Slide 18—

Procedure Calls in Ada

```
COMPUTE_RISE(X_VEL, Y_VEL, TARGET_DIST, ELEVATION);
COMPUTE_RISE(50.0, 60.0, (POSN + 10.0), R(1));
  --R = array of FLOAT
```

- The order of the arguments can be changed.

```
COMPUTE_RISE(RISE => Z, DIST => D,
             V_X => 50.0; V_Y => 60.0);
```

- *Default values can be specified for* `in` *parameters.*

```
procedure GENERATE_REPORT(
          DATA_FILE: in FILE := SUPPLY_DATA;
          NUM_COPIES: in INTEGER := 1;
          HEADER: in LINE := STD_HEADER;
          CENTERING: in BOOLEAN := TRUE;
          TYPE_FONT: in PRINTER := TIMES_ROMAN) is ...
```

- Valid Calls:

```
GENERATE_REPORT;
GENERATE_REPORT(NUM_COPIES => 5);
GENERATE_REPORT(HEADER => "FINAL REPORT",
                TYPE_FONT => HELVETICA,
                NUM_COPIES => 100);
```


—Slide 19—

Usage of Default Values

- In a procedure definition, certain arguments must be always present in a standard order.
- The remaining arguments are optional:
- Example:

```
procedure PLOT(X, Y: in FLOAT;
               PEN: in PEN_POS := DOWN;
               GRID: in BOOLEAN := FALSE;
               ROUND: in BOOLEAN := FALSE) is ...


PLOT(0.0, 0.0);
PLOT(0.0, 0.0, GRID => TRUE);
PLOT(0.0, 0.0, UP, ROUND => TRUE);
PLOT(0.0, 0.0, ROUND => TRUE, PEN => UP);
```

—Slide 20—

Separation of Subprogram Bodies

- `Declaration` ⇒ specifies its interface.
- `Body` ⇒ specifies its implementation.
- *Example*

```
Declaration:
   procedure ADD(I: in ITEM; Q: in out QUEUE);
   procedure REMOVE(I: out ITEM; Q: in out QUEUE);
   function FRONT(Q: QUEUE) return ITEM;
   function IS_EMPTY(Q: QUEUE) return BOOLEAN;
Body:
   procedure ADD(I: in ITEM; Q: in out QUEUE) is
     -- ...
   end
   ...
   function IS_EMPTY(Q: QUEUE) return BOOLEAN is
     -- ...
   end
```


—Last Slide—

Overloading

- *Use of two or more subprograms with the same name but different types of parameters—hence different bodies.*

```
procedure PUT(X: INTEGER);
procedure PUT(X: FLOAT);
procedure PUT(X: STRING);
function "+"(X,Y: VECTOR) return VECTOR;
```

- They define the same conceptual operation on arguments of different types.

```
PUT(I+1);  PUT(SQRT(Y));  PUT("HELLO WORLD!");
A := A + B;
```

- **Overload Resolution**: The compiler chooses, from different alternatives, the code to run for some operation.
- In the presence of coercion and operator overloading, overload resolution can be *tricky*!

```
I, J: INTEGER; R: FLOAT;
R := I + J;       --Which + does the compiler use?
```

[End of Lecture #6]