V22.0490.001
Special Topics: Programming Languages

B. Mishra
New York University.

Lecture # 15

—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* ⇒
  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.

[End of Lecture #15]