V22.0490.001
Special Topics: Programming Languages

B. Mishra
New York University.

Lecture # 4

—Slide 1—

Principle of Orthogonality

- **Orthogonal Design**

  Each component of a language should be independent of other components.

- In a truly orthogonal design,
  - There are a small number of separate basic constructs (e.g., data types, control structures, bindings, abstractions, etc.)
  - The constructs are combined according to regular and systematic rules without *arbitrary restrictions*.

- **Corollary**

  There should not be more than one way of expressing any action in the language.

- Expressiveness vs. Complexity

  The complexity of the language may increase without a corresponding gain in facility.

—Slide 2—

Control Structures

- **Concatenation**

```
begin
  S0;
  S1
end;
```

- **Selection**

```
if B0 then S0
else if B1 then S1
...
else Sn;
```

```
case E of
 L0: S0;
 ...
 Ln: Sn
end;
```

- **Iteration**

```
while B do S;
```

```
for I := I0 to In
  do S;
```

- **Termination/Escape**

```
goto L;
return;
exit;
abort;
```

```
break;
continue;
raise Exception
```

—Slide 3—

Data Structures

- **Scalar Types**
  - *Predefined Types*:
    (numerals, characters, Booleans, reals)
  - *Enumerated Types*: Discrete valued.
- **Composite Types**: Domain constructions:
  - **Products**: $A_1 \times A_2 \times \cdots \times A_n$. (Cartesian Product)
    Projection or *Field Selection* operation selects a component. Examples: Pascal and Ada `record`, `C struct`.
  - **Sums**: $A_1 + A_2 + \cdots + A_n$. (Disjoint Union or Co-product)
    Injection operation constructs elements from the sum by means of a "tag". Examples: Pascal and Ada `variant record`, `C union`.
  - **Function**: $A_1 \mapsto A_2$ (Injective Map, Array)
    Application or *Subscription* operation maps a value in the domain (A_1) to a unique value in the range (A_2). Examples: Pascal, Ada & `C array`
- **Anonymous Types**:
  Access types in Ada, Pointers in `C` and Pascal.

—Slide 4—

Examples of Domain Constructions

- **Products of Domains**

```
record
  i: integer;
  c: char
end;
```

All ordered pairs whose first components are integers and the second components are characters.

- **Sums of Domains**

```
record case tag: Boolean of
  true:  (i: integer);
  false: (c: char)
end;
```

Either an integer *or* a character, together with a Boolean component to differentiate the two possibilities.

- **Function Domains**

```
array[char] of integer;
```

Describes a *function* mapping characters into integers. Each array determines a *unique* integer 'component' for *every* character 'subscript.'

—Slide 5—

Principle of Abstraction

- An abstraction facility may be provided for any *semantically meaningful* category of syntactic constructs.
- **Goals:**
  - No new syntactic category
  - Simple compiler–Parameter passing, type checking
  - Safety
- **Abstraction** is a process of extracting general structural properties in order to allow inessential details to be disregarded.
- **Examples**
  - *Functions*: Abstraction of expression.
  - *Procedures*: Abstraction of statements.
  - *Macros & Inline Expansion*: Abstraction of lexical structure.
  - *Classes & Packages*: Abstraction of domains.
  - *Monitors, Tasks*: Abstraction of processes.

—Slide 6—

Examples of Abstractions

- **Abstraction of an Expression**

```
var F, G: real;
G := (F-32.0)*5.0/9.0;
```

```
function Convert(F: real):real;
begin
 Convert := (F-32.0)*5.0/9.0
end;
```

- **Abstraction of a Statement**

```
begin
  var U, V, Temp: real;

  Temp := U;
  U := V;
  V := Temp
end;
```

```
procedure Swap(var U: real;
               var V: real);
  var Temp: real;
begin
  Temp := U;
  U := V;
  V := Temp
end;
```

- **Abstraction of a Type**

```
const n: integer;
type String =
  array[1..n] of char;
```

```
type function
  String(const n: integer);
begin
  String = array[1..n] of char
end;
```

—Slide 7—

Imperative Language: Assignment

- Objects in a program have two attributes:
  - *Location*: L-value
  - *Value*: R-value

```
X := X + 1;        (* Pascal *)
X := .X + 1;       (* BLISS *)
X := !X + 1;       (* ML *)
```

- In Pascal, `X` in LHS is the L-value of `X` and `X` in RHS is the R-value of `X`.
- In BLISS, `X` refers to the L-value and `.X`, the R-value. BLISS allows arithmetic on L-values (pointer arithmetic). `.` is an explicit *dereferencing operator* in BLISS.
- In ML, `X` refers to the L-value and `!X`, the R-value. `!` is an explicit *dereferencing operator* in ML.

—Slide 8—

L-values

- Some languages allow L-valued expression: E.g., in `C++`:

  ```
  int a[10];
  int& f(int i){return (a[i]);}
  f(5) = 17;
  ```

  `f` is an L-valued function

- If two L-valued expressions denote the same location, they are called *aliases* for that location.
- FORTRAN allows explicit aliasing via `EQUIVALENCE` construct.
- Every L-valued expression has an R-value; but *not the converse*.

—Slide 9—

Variations on Assignment

- **Update Operation**

```
L +:= E;                (* ALGOL 68 *)
L += E;                 /* C        */
```

L-value of `L` contains the sum of R-values of `L` and `E`. R-value of `L` is obtained from a single evaluation of its L-value.

- **Multiple Targets**:

```
L1 := L2 := ... := Ln := E;
            (* ALOGOL 60 *)
```

All the L-values of L-expressions `L1`, `L2`, . . ., `Ln`, and the R-value of `E` are evaluated. Then all the L-values are updated.

—Slide 10—

Variations on Assignment (contd)

- **Multiple Assignments**:

```
L1, L2, ..., Ln := E1, E2, ..., En;
         (* ALOGOL 60 *)
```

All the L-values of L-expressions `L1`, `L2`, . . ., `Ln`, and then all the R-values of `E1`, `E2`, . . ., `En` are evaluated. Then all the L-values are updated, while maintaining positional correspondence.

- **Assignment Expression**:

```
L := E                    (* ALOGOL 68 *)
L = E                     /* C         */
```

The expression's value is the R-value of `L`. The expression updates `L` as a side effect.

```
if((n += a) > 0) n--;   /* C */
a = b = c = d = e;
/* C, = is right-associative */
```

—Slide 11—

Pointer

- **Pointer** (Access, Anonymous Variable)
  A variable `E` whose R-value is an L-valued expression (or a special value `null`). Its L-value is a location giving access to a storage indirectly. The L-value of `E^` is an anonymous variable which can be updated as any other L-valued expression

```
E^ := E^ + 1;
```

- **Allocation & Disposal**
  As pointers allow storage to be addressed indirectly, it may be allocated and disposed of at arbitrary execution points.

  In some languages, allocated locations are subsequently disposed of explicitly by the user. In others, inaccessible locations are automatically searched for and disposed—*garbage collection*.

- **Inaccessible Locations**

```
new(p); p := nil;
```

—Slide 12—

Storage Insecurities

- **Storage Insecurities**
  Dangling reference *is a pointer to a location that has potentially been used for another purpose—*

  Extent (lifetime) of the location ended before all ways of accessing the location have ended.

- Dangling reference can be created by *aliasing* and by *implicit release of storage in an activation record*, with a pointer pointing to that storage.

```
var p, q: ^integer;
begin
  new(p);
  q := p;
  dispose(p)
end;
```

```
var p: ^integer;
procedure q;
  var i: integer;
begin
  p := ADDR(i)
end;
```

—Slide 13—

Binding

- **Binding**

  Association of a *name* to an *attribute*. Following are examples of some of the attributes:

  - *L-value*—Location.
  - *Type*—The set of possible R-values allowing a set of allowable operations on them.
  - *Miscellaneous Constraints*—Assertions, array bounds, discriminant or tag values.

- **Binding Points**:

  Binding is done by *declarations* & happens at different and invisible points after the program is submitted for execution.

  The later the binding, the more flexible is the language.

—Slide 14—

Referential Transparency

- **Scope**

  Scope of a name or a declaration =

  The section of a program text in which the name has the attributes established by declaration

- *Referential transparency*.

  Thus scope corresponds to a *local name space*. Bound occurrences of a variable can be renamed without changing the meaning.

```
function succ(x:integer):integer;
begin
  succ := x + 1;
end;
```

```
function succ(y:integer):integer;
begin
  succ := y + 1;
end;
```

—Slide 15—

Types

- **Static Type-Checking**
  The type of an expression is known at the *compile time*.
  A language is **strongly-typed**, if all type checking can be done at compile time.
  —Fewer programming errors
  —Better compiled code.
- A language is **type complete** if all the objects in the language have equal status (first class citizens).
- *Type Insecurities*
  A domain incompatibility cannot be determined at compile time.
- Domain incompatibility is handled either by invoking an *exception* that aborts the program with an error message or by *type coercion*.

—Slide 16—

Type Insecurities & Coercion

- Example

```
var                           (* PASCAL *)
  wide:1..100; narrow:10..20; farout:150..300;
begin
  narrow:=farout; wide:=narrow; narrow:=wide
end;
```

–Compiler cannot determine whether the last assignment is illegal.
–Ada solves this by assigning a new compile-time *type* for every subrange constraint.

- *Type Coercion*
If the operation and its arguments are incompatible then convert the *argument* or the *operation*, so that the types are compatible.

```
var                           (* PASCAL *)
  x: real; i: integer;
x := i;
```

—Slide 17—

Type Equivalence

- In the presence of structured types and user-defined types, it is necessary to determine if two types are equivalent.
- Two categories:
  - *Name Equivalence*: Types with same name.
  - *Structural Equivalence*:
    Types with same structure.
- Example

```
declare
  type BLACK is INTEGER;
  type WHITE is INTEGER;
  B:BLACK; W:WHITE; I:INTEGER;
begin W := 5; B := W; I := B; end;
```

–All assignments are legal under structural equivalence;
–All assignments are illegal under name equivalence.

—Last Slide—

Type Equivalence (Contd)

- Structural equivalence is hard to determine:

```
--Ada
type T1 is record            type T2 is record
  X:INTEGER;                   X:INTEGER;
  N:access T1                  N:access T2
end record;                  end record;

type T3 is record            type T4 is record
  X:INTEGER;                   X:INTEGER;
  N:access T2                  N:access record
end record;                       X: INTEGER;
                                  N: access T4
                                end record;
                             end record;
```

- **Examples**

`C`	*Structural Equivalence*
`C++`	*Name Equivalence*
PASCAL	*Declaration/Name Equivalence*
Ada	*Name Equivalence*

[End of Lecture #4]