G22.2210

# Programming Languages: PL I

B. Mishra
New York University.

**Lecture # 2**
September 18 1995

—Slide 1—

*What Constitutes a Programming Language?*

- **Desiderata**
  1. Every '*computable function*' can be expressed.

     **Note:** "Application Level" language $\neq$ Full Programming Language. E.g., *Job Control Language*, *Database Language*.
  2. Every program is unambiguous and implementable.

     E.g., English $\neq$ a Programming Language
- **Turing Computable**
  A function can be computed by a Universal Turing Machine.

—Slide 2—

## Church-Turing Thesis

*Any function that can be described finitely and computed in finite time is Turing-computable.*

*Every computable function is Turing-computable.*

- **Examples**

  *1) Turing Machine, 2) Church's $\lambda$-calculus*

  *3) Thue System 4) Post Correspondence Process*

  *5) Markov Systems*

  *6) Fredkin's Billiard Ball Machine*

  *7) Feynmann's Quantum Computers*

  *8) Adleman's DNA Computer . . .*

- **Human Brain** + infinite supply of ink and papers

—Slide 3—

*Unsolvable Problems!*

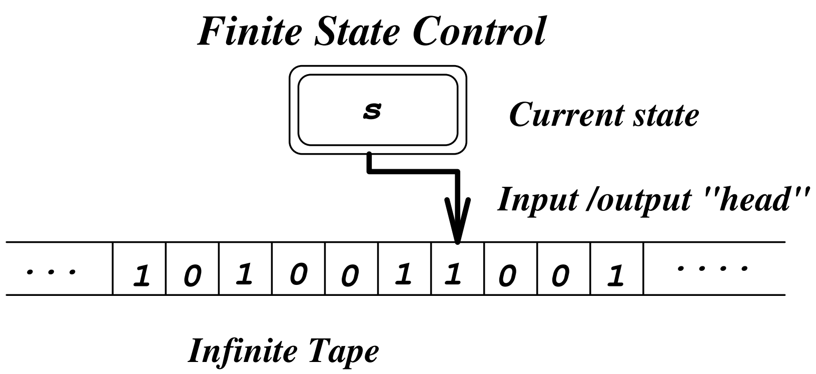
- **Note:**
  There are "countably" many computable functions. But there are "uncountably" many functions,
  $\mathbf{N} \mapsto \{0, 1\}$.
- **Diagonalization argument**:
  Alan Turing showed that

  > There are functions that are not *Turing-computable.*

- **Halting Problem:**
  Is a given program in an "highly expressive language" (e.g., Pascal) *nonterminating*?

## —Slide 4—

## *Turing Machine*




- According to its "program" (i.e., Finite State Control) and "input" (i.e., initial string on the tape)
  - Read the **current symbol** on the cell on the tape under the head.
  - Check the **current state**
  - Write a new symbol on the tape
  - Move the head left or right one cell
  - Go to the next state
- *The next state is a function of the current state and the current symbol.*

—Slide 5—

*Turing Machine*

- A Turing Machine is equivalent to a program.
- **Universal Turing Machine**

  Given any Turing machine $\mathcal{M}$ and some input $\mathcal{W}$, a universal Turing machine $\mathcal{U}$ will mimic (i.e., simulate) the behavior of $\mathcal{M}$ on $\mathcal{W}$.

$$\mathcal{U}(\mathcal{M}, \mathcal{W}) \equiv \mathcal{M}(\mathcal{W})$$

- **Virtual Machine**

$$\text{PROGRAM} \equiv \text{TURING MACHINE}$$

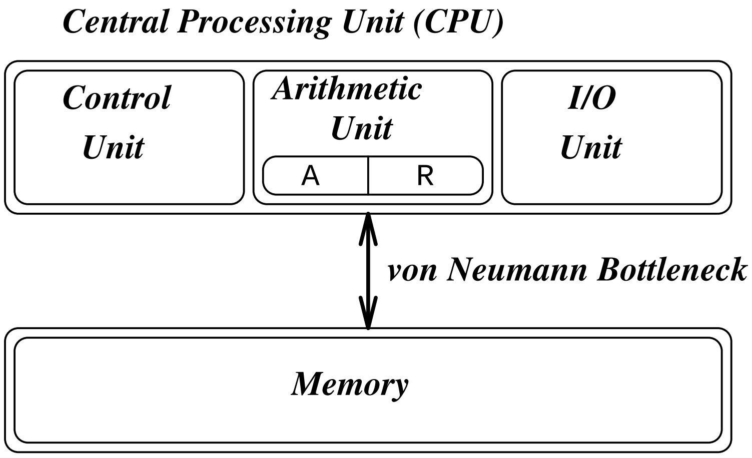
$$\left.\begin{array}{c}\text{PROGRAMMING}\\ \text{LANGUAGE}\end{array}\right\} \equiv \text{UNIVERSAL T.M.}$$

## —Slide 6—

## *von Neumann Architecture*

- John von Neumann

(1940's, Burks, Goldstein & von Neumann)




- "**Dance-Hall Architecture**"
- Original Design
  1. CPU: 2 registers:
     $A$ = Accumulator, $R$ = Register
  2. MEMORY: 4096 Words (40 bits)
     Data or Instructions

# —Slide 7—

## *Instruction Set Architecture*

- **Data: Only integers**
- **Arithmetic Operations:**

  *Add, Subtract, Multiply, Divide, Absolute Value*
- **Add & Subtract:**
  result was held in an accumulator.

```
A := A + M[i];      A := A - M[i];
A := A + |M[i]|;    A := A - |M[i]|;
A := - M[i];        A := |M[i]|;      A := -|M[i]|;
A := A * 2;         A := A div 2;
{A, R} := { (M[i]*R) div 2^39, (M[i]*R) mod 2^39 };
{A, R} := { A mod M[i], A div M[i] };
```

- **Assignment to Memory Location**

```
A := M[i];     M[i] := A;    R := M[i];    A := R;
```

- **Control Flow**

```
goto M[i].left;             goto M[i].right;
if A >= 0                   if A >= 0
  goto M[i].left;             goto M[i].right;
```

## —Slide 8—

### *Modifiable Statements*

- Since data & instructions are treated the same way, the instructions can be manipulated just as data.
- **Modifiable statements**
  - Modify the address in `M[i].left` from `A`
  - Modify the address in `M[i].right` from `A`
- Usage: Array indexing in von Neumann's machine. In the modern architectures, index registers solve this problem.
- Amenable to misuse, as control structure of a program can be modified dynamically.

```
9.left)     A := <address>;
9.right)    Modify M[10].left from A;
10.left)    goto M[3].left
```

**Question**: Where does the control transfer?

## —Slide 9—

## *Machine Language*

- Binary Code: Each Instruction is coded in binary.
  *Machine operations, Values & Storage Locations*
- RISC **(Reduced Instruction Set Computer)**
  CISC **(Complex Instruction Set Computer)**
- Depends upon
  1. Register Structure
  2. Data & Control Paths
  3. Pipelining, Prefetching
  4. Microprogramming

## —Slide 10—

### *Assembly Language*

- **Symbolic Names** are assigned to operations, values and locations.
- Assembler: 2-pass

  **Pass I:** Locations are assigned addresses.

  **Pass II:** Symbolic Names $\mapsto$ Codes

## —Slide 11—

## *Translators*

- **Compiler**:
  Translates *source code* into *target code* (in machine language) at **compile time**.

  The target code takes input data and produces output data at **run time**

- **Interpreter**:
  Interprets an instruction in the language in terms of the equivalent sets of operations in the machine language.
  Takes instructions and input data and produces output data.

```
K := I + J;         LOAD I;        1001 0000;
                    ADD  J;        0001 0001;
                    STORE K;       1010 0010;
(High-Level)        (Assembly)     (Machine)
```

## —Slide 12—

## *Description of a Programming Language*

- **Syntax:** *The grammatical structure.*
- **Semantics:** *The meaning of the constructs.*
- **Pragmatics:** *Practicality*
  - Implementation Issues
  - Efficiency
  - Portability
  - Interactive/ Static
- **Aesthetics:** *Appeal or usability based on the design principles.*
  - Reasoning about programs
  - Program Synthesis
  - Verification
  - Analysis

## —Slide 13—

## *Syntax*

- A set of rules governing the organization of "symbols" in a program.
- **Formalisms**: PBF/ BNF (Panini-Backus Form, Backus-Naur Form, Backus Normal Form), EBNF (Extended BNF), Syntax Chart

```
<sentence> ::= <noun> <verb>
<noun> ::= bud | sam | tom
<verb> ::= hacks | builds | proves
```

- Syntax restricts
  - **Names**: variables, procedures
  - **Expressions**: Identifiers, their order & operators
  - **Statements**
  - **Definitions**: Procedures, declarations
  - **Programs**: Groups of all of above

## —Slide 14—

### *Syntax (contd)*

- Examples of Syntactic Errors

```
"Illegal variable name"
"Missing semicolon"
```

- Compiler uses **lexer** & **parser** to determine the structure (parse tree). Relatively easy, for most programming languages.

## —Slide 15—

## *Semantics*

- *Semantics* defines the behavior of the constructs in a programming language.
- Gives meanings to a program.

  Allows precise interpretation of a program.

  1. Compilers use it for "*syntax-driven semantics analysis*"
  2. Semantics definition of language. Eliminating semantics ambiguities.
  3. Helps users in reasoning/pondering about programs.

## —Slide 16—

### *Semantics Analysis*

- What is the meaning of the following program?

```
function F(x, y: integer) returns integer is
  if x = y
    then y + 1
    else F(x, F(x - 1, y + 1));
```

- **Exercise**:
  Check that any of the following two interpretations could be valid (i.e., *fixpoints*):

```
if x = y then y + 1 else x + 1
if x >= y then x + 1 else y - 1
```

- Are there other valid interpretations?

- What is the simplest valid interpretation (i.e., *least fixpoint*)?

## —Slide 17—

## *Types of Semantics*

- **Operational Semantics**
  Language is defined by its implementation on an "*abstract*" machine.
  - VAX compiler for `C` on `UNIX`
  - VDL (Vienna Definition Language) for PL/1
- **Axiomatic Semantics**
  Axioms are defined for statements specifying *post-conditions* given their *pre-conditions*
  - **Post-condition**: *what must be true after executing a statement*,
  - **Pre-condition**: *what was true before the statement was executed*.
- **Denotational Semantics**
  - **Semantics Valuation Functions**: Map syntactic constructs to abstract values they denote—e.g., numbers, truth values, functions, etc.
  - Value denoted by a construct is specified in terms of the values denoted by its *syntactic subcomponent*.

## —Slide 18—

### *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 19—

## *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 20—

*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 Coproduct)
    *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 21—

## *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 22—

## *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 23—

## *Examples of Abstractions*

- **Abstraction of an Expression**

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

- **Abstraction of a Statement**

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

- **Abstraction of a Type**

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

# —Slide 24—

## *Principle of Correspondence*

- **Correspondence between declarative and parametric constructs**
- For any formal parameter, $\phi$ and a compatible actual parameter, $A$

  > *the effect on a block body of the qualifying declaration $\phi = A$*

  should be identical to

  > *the effect on an abstraction body with $\phi$ as its formal parameter and $A$ the corresponding actual argument.*

- Example

```
var i: integer;
begin
  i := -j;
  write(i)
end;
```

```
procedure p(i: integer);
  begin
    write(i)
  end;
begin
  p(-j)
end;
```

# —Last Slide—

## *Principle of Qualification*

- **Relationship between the bodies of abstracts and the bodies of blocks**
- **Example**: Command `C` in procedure

```
procedure p(I: I´);
begin
  C
end;
```

corresponds to command `C` in block

```
var I: I´;
begin
  C
end;
```

- As a consequence of the principle of abstraction, we conclude that the body of a block can be any meaningful syntactic category.

[End of Lecture #2]