V22.0490.001
Special Topics: Programming Languages

B. Mishra
New York University.

Lecture #2

—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 λ-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!

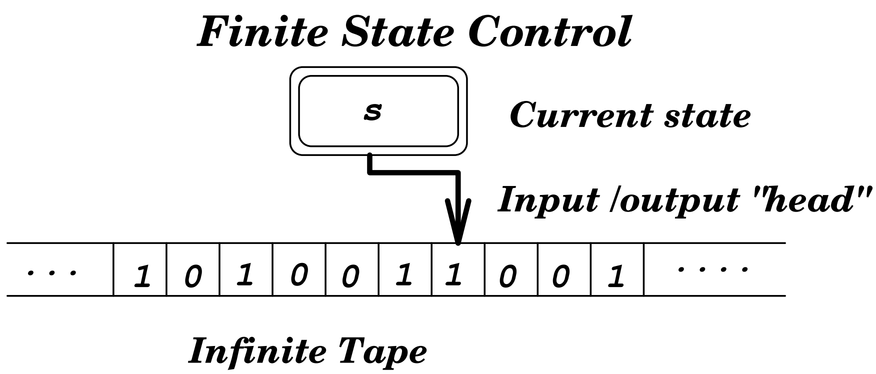
- **Note:**
  There are "countably" many computable functions. But there are "uncountably" many functions,
  $\mathbf{N} \mapsto \{0, 1\}$.
- **Diagonalization argument**:
  Alan Turing showed that

  > 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**

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

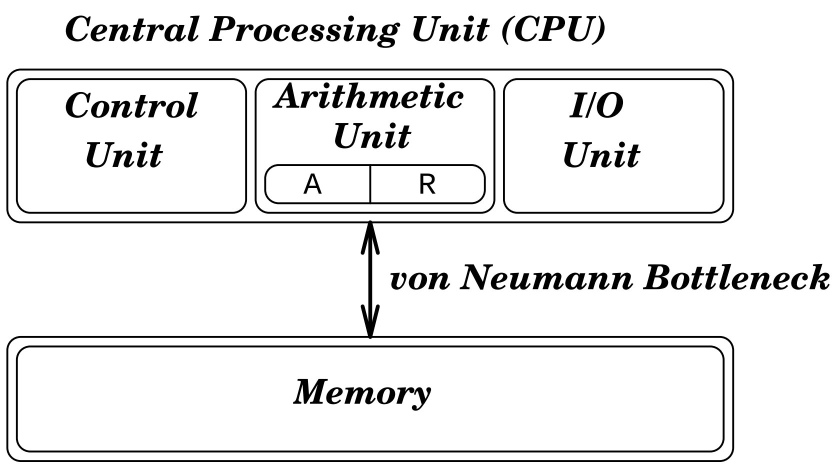
$$\left.\begin{array}{c}\text{PROGRAMMING}\\ \text{LANGUAGE}\end{array}\right\} \equiv \text{UNIVERSAL T.M.}$$

—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—

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*.

[End of Lecture #2]