CONCEPTS OF

PROGRAMMING LANGUAGES

Fall 1995
Dr. James D. Arthur
(Revised August 1995)

Fall 1996
Dr. J.A.N. Lee
(Revised January 1997)

Spring 1998
Dr. Lenwood S. Heath
(Revised January 1998)

Introduction, or Why Bother With This Stuff, Anyway?

- **Increased capacity to express ideas.**
- **Improved background for choosing languages.**
- **Increased ability to learn languages.**
- **Understanding of significance of implementation.**
- **Ability to design new languages.**
- **Overall advancement of computing.**

Language Evaluation Criteria

- **Readability & Writability**
  - → **Simplicity**
    - small # of basic components
      (subsets a poor solution!)
    - one syntax: one meaning
      Counter-example in C: four ways to increment x:
      x++;
      x=x+1;
      x+=1;
      ++x;
      In FORTRAN, two meanings for:
      Y=SUM(I,J)
      -- array reference
      -- function call

Language Evaluation Criteria

→ Orthogonality

=> Any composition of basic primitives is allowed

- need small set of primitives, ways to combine them
- Pascal not very orthogonal.

  Functions can't return structured types.

  Type of formal parameter must be stated in function/ procedure heading unless parameter is a function or procedure.

  Enumerated types can't be read or written.

  etc...
- Non-orthogonality is often to simplify implementation.
- LISP is much more orthogonal than Pascal.

Language Evaluation Criteria

→ **Control Statements/Constructs**

Importance and desirability of various control mechanisms varies with the language.

→ **Data Types**

Rich set of data types makes programs much easier to write and understand. Provides abstraction.

→ **Syntax**

Matters more than you think!

Identifier length, reserved words, layout, etc.

→ **Abstraction**

Must be able to hide details, or complexity is too great.

process abstraction

data abstraction

Language Evaluation Criteria

- **Reliability**

  → **Definition: performs to specifications under all conditions**

  → **Impact from:**

  type checking (or lack thereof)

  compile-time *best*

  runtime *good*

  exception handling

  Special language features to help intercept and handle unusual situations. No magic.

  Somewhat controversial.

  aliasing (Y)

  Two or more names for same memory cell.

```
PASCAL:     var p,q: ^int;
            begin
                new(p);
                q:=p;
FORTRAN:    Character *20 Last, First
            Character *40 Name
            Equivalence (Name (1:20), Last),
                        (Name (21:40), First)
```

Language Evaluation Criteria

- **Cost**

  → **More than just runtime!**

  time to train programers

  *program development time

  compile time

  runtime

  *maintenance time (>50%!)

  → *** functions of writability and readability**

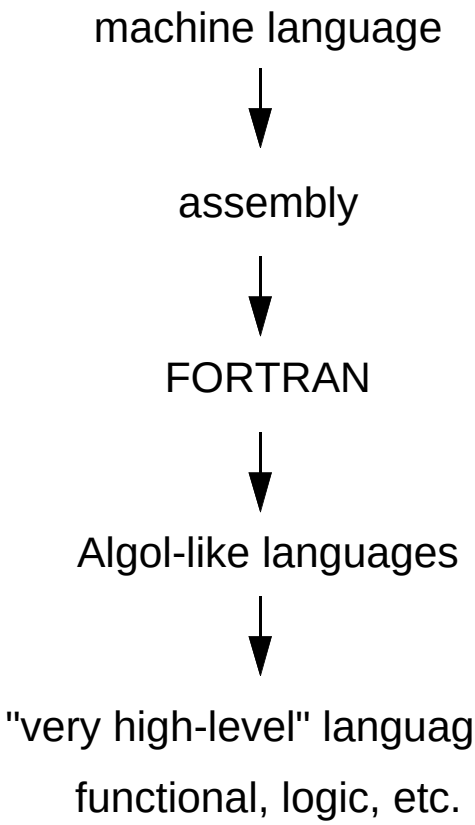
  => most important

Influences on Language Design

(1) Computer architecture

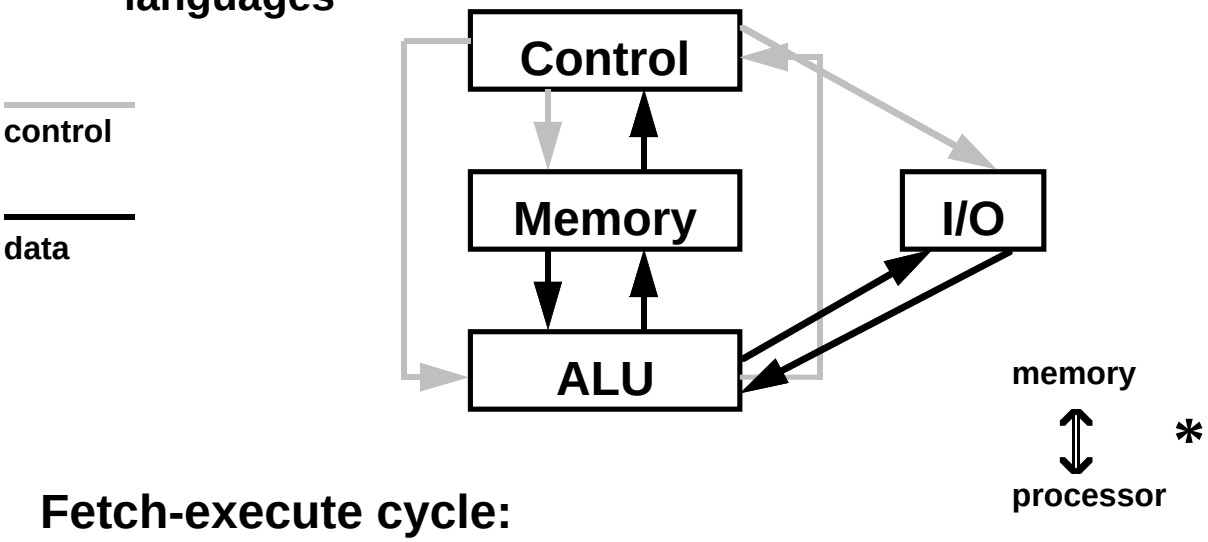
(2) Programming methodologies

Historically more of (1), moving toward (2). Getting higher and higher-level:



The von Neumann Architecture

- **John von Neumann at Princeton**
  - → **late 1940's**
  - → **has influenced the design of most programming languages**



- **Fetch-execute cycle:**
  - → **Control fetches next instr from memory**
  - → **Control decodes instr**
  - → **Execution**

    data from memory to ALU, or

    data from ALU to memory, or

    I/O

The von Neumann Architecture

- **The von Neumann architecture is reflected in traditional programming languages in two ways:**
  - → **sequential, step-by-step execution of instructions**
  - → **modifiable variables -- "cubbyholes" in memory**
- **These languages became popular and drove further architectural designs. Vicious circle . . . other language designs didn't have much chance until recently.**

Programming Methods and Methodologies

- **Noted Language Deficienciences**
  - → **Type Checking**
  - → **Inadequate Control Structures**
  - → **Lack of Exception Handling**
- **Novel Approaches to Problem Solving**
  - → **Data and Process Abstraction**
    - \+ Object-Oriented
      - Classes (Simula 67)
        - Encapsulation
        - Data Abstraction
    - \+ Data Flow
  - → **Concurrency**

Translation and Interpretation

- **We could build a special machine to execute each language directly, but this is impractical. So how to get a program in a high-level language down to machine code?**

- **Interpretation**

  An interpreter takes statements of a program one at a time and executes them directly as follows:

  Get next statement

  Determine actions

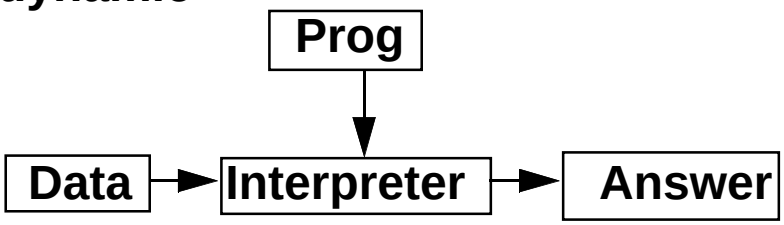
  Perform actions

  Repeat

  (*look familiar?*)

Notes on Interpretation:

- **data is provided to the interpreter as required**
- **one high-level instruction (HLI) => one sequence of machine-level instructions**
- **redetermine actions each time HLI is encountered**
- **highly dynamic**



Translation

- **Translation**

  → **A translator takes a program in language A and produces an equivalent program in language B. If B is "closer" to machine code than A, it's called a compiler.**

- **Notes on translation:**

  → **high-level program --> machine-level prog**

  (*not* HLI-->MLI)

  → **parsing + code generation**

  → **decode each statement once**

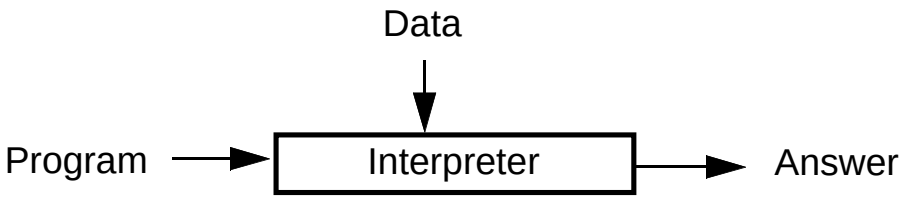
  => saves time

  → **store expanded version of program**

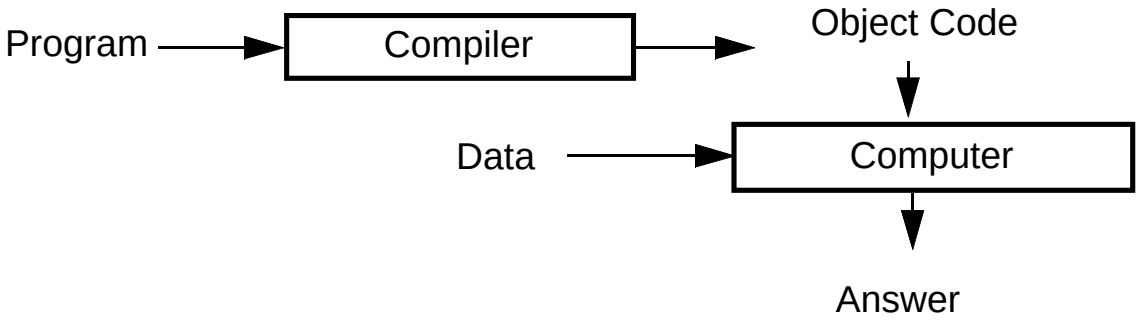
  => costs space

Interpretation vs. Translation

- **Interpretation**



- **Translation**



→ **Combine translation and interpretation by substituting Interpreter for Computer above.**