Programming Languages: Introduction

I. Definitions:
   A. A *program* is a specification of a computation.
   B. A *Programming Language (PL)* is a notation for writing programs

II. The need to understand programming languages, or "Why study this stuff?"
   A. Practical Reasons
      1. [Sethi, p. 3 -4 on Mariner I being destroyed, ~$18 - $20 million]
      2. Techniques for small programs don’t necessarily scale up
         a. Miller: "The magic number 7 +/- 2"
         "The process of organization enables us to package the same amount of information into far fewer symbols, and so eases the task of remembering."
         b. Experts: 50K-100K "chunks" of heuristic info., takes 7 sec. to store a chunk, & so takes 10 years to become an "expert", 70 ms. to retrieve. [Harmon & King]
   B. Sapir-Whorf hypothesis:
      - [MacLennan, p. xxi] controversial linguistic theory that states that the structure of language defines the boundaries of thought. (i.e. thought follows language)
      - No evidence that a certain lang. will *prevent* certain thoughts, however a given lang. can *facilitate* or *impede* certain modes of thought. E.g. Inuit (Eskimo): dozens of words for snow [Drake]
      - In PL this means that though it may not be impossible to do something in a given PL, it may not *lend* itself to it
   C. Improved background
      1. You can choose the right language for the right job
      2. Historically the *best* has not always won out. E.g. in early 60’s, ALGOL had better control statements, block structure, & recursion than did FORTRAN, yet FORTRAN won out.
      3. We won’t all design new languages, but the concepts discussed will help us design *user interfaces*

III. Why Have Different Languages?
   A. It has been shown that any program on a VonNeuman machine can be written using only the constructs:1.Sequence, 2. Repetition, and 3. Decision.
   B. Different languages lend themselves to different application areas.
IV. Factors Influencing Language Design

A. Computer Architecture

1. Von-Neumann architecture [draw diagram of Mem, CPU, I/O]
   a. Data & programs in same memory
2. Central feature of imperative languages are variables because of Von N. arch.
   a. variables model the memory cells, assignment stmt based on "piping" memory info. to CPU, iteration efficient
3. In contrast: functional (applicative) lang. simply apply functions to parameters & don't need variables, assignment statements, or iteration
4. Problem with these: don't naturally lend themselves to the Von N. arch.

B. Program Design Methodologies

1. Late 60's & 70's: shift in major cost from hardware to software
2. Larger progs. meant new methods needed: top-down design & stepwise refinement [Parnas ’71]
3. Needed to solve problems of: incomplete type checking, inadequate control statements (needed goto's)
4. Shift from process-oriented to data-oriented led to ADT's
5. OOP takes ADT's a step further by making them reusable (inheritance). Need run-time binding to take advantage of inheritance (i.e. operator overloading)
   a. Examples of OOP: C++, smalltalk, CLOS
6. Process-orienting still being explored for concurrent processing (Ada, parallel Fortran)

V. Implementation Methods

A. Virtual Machine

1. Layers: Bare machine, Assembly lang., OS, compilers for "virtual" machines (for COBOL, PASCAL, LISP, C, etc.) [See Fig. 1.2 p. 25]

<table>
<thead>
<tr>
<th>Lisp</th>
<th>Cobol</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compilers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Assembly Lang.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bare Hardware</td>
<td></td>
</tr>
</tbody>
</table>

B. Compilation to Machine Language

1. Fetch-decode-execute cycle [see overhead]
2. Translation from top-most to bottom-most layer [diagram above]
3. Compile, link & load [e.g. Hello world w/#include <stdio.h>]
4. [Fig. 1.3, p. 26] Compilation steps in more detail, pseudo-code (p-code)

C. Intermediate code: pseudo-code & Interpreters

1. History of pseudo-code
2. Interpretation is slow & requires many more resources

VI. Programming Environments: Set of tools

A. debugger
B. browser (for class libraries, headers)
C. formatter (pretty-printer)
D. pre-defined library (objects in OOP)
E. GUI