V22.0490.001

Special Topics: Programming Languages

B. Mishra

New York University.

Lecture #2.5

—Slide 1—

Recapitulation

- Computers work in machine language, which is difficult for human understanding; Assembly languages were invented to provide mnemonic abbreviations;
- Originally, assembly language was designed with a one-to-one correspondence with machine language; Translating from assembly language to machine language was done with an *assembler*;
- Higher level machine-independent languages (starting with numerical computation): Fortran (1950s)
- Other high-level languages like Lisp and Algol.

—Slide 2—

Compilation and Interpretation

- *Compilers* were devised to translate high-level languages to assembly- or machine-languages
- Source program is translated by a compiler into a target program (at compile time); when the user runs the target program, it takes an input and computes the output.
- The compiler is the locus of control during compilation; the target program is the locus of control during its own execution.
- An interpreter takes a source program and the input, and interprets each program statement one at a time to produce the computed output.
- Interpreter is the locus of control during execution.

—Slide 3—

Virtual machine

- Most language implementations include a mixture of both compilation and interpretation;
- A source program is translated (by a Translator) into an intermediate program; the intermediate program together with an input is executed on a virtual machine (by an Interpreter) to produce an output.

—Slide 4—

•

- **Implementation of Java**: The Java language definition defines a machine independent intermediate form: *byte code*
- Byte code is the standard format for distribution of Java programs; it allows program to be transferred easily over the inter-net and then run on any platform.
- Java implementations *used to be* based on byte-code interpreters; but more recent faster implementations employ *just-in-time* compiler that translates byte-codes into machine language immediately before each execution of the program.

—Slide 5—

Additional Tools

- **Linker**: A compiler (as in Fortran) may translate the source program to a machine language, but cannot execute it without auxiliary library of subroutines; the compiler relies on a separate program, known as a *linker* to merge the appropriate library routines into the final program.
- **Assembler**: Many compilers generate assembly language code instead of machine language. It needs an *assembler* to translate the assembly code to machine language. It isolates the compilers from changes in the format of the machine language (usually mandated by the operating system)

—Slide 6—

• •

- **Preprocessor**: A *preprocessor* may be used to create a "modified source program" that can easily/efficiently compiled into assembly language: It may remove comments, expand macros, aid *conditional compilation*, etc.
- There are other variations to these themes...

—Slide 7—

• • •

- **Compiler compiler**: From the specification of a programming language semantics, and of a computer machine architecture, it generates a complier automatically; it usually works on the front-end and back-end separately. The front end consists of preprocessor, syntax and semantics analyzer, etc. and generates machine-independent intermediate code; the back end consists of code generator, optimizer, linker, assembler, etc. and generates the machine language code from intermediate code.

—Slide 8—

Programming Environments

- **Editors**: Cross-referencing facilities; syntax-directed editing, etc.
- **Pretty Printers**: Enforce formatting conventions;
- **Style Checkers**: Enforce syntactic or semantic conventions (often more stringent than that required by the compiler);
- **Configuration Management Tools**: Track dependence among the (many versions of) separately compiled modules;
- **Perusal Tools**:
- **Debuggers**:
- **Profilers and other Performance Analysis Tools**:

—Slide 9—

The Programming Language Spectrum

- **Declarative**
  - *Functional*: Lisp, Scheme, ML, Haskell
  - *Data Flow*: Id, Val
  - *Logic, Constraint-based*: Prolog, (spreadsheet)
  - *Template-Based*: XSLT
- **Imperative**
  - *von Neumann*: C, Ada, Fortran, etc.
  - *Scripting*: Perl, Python, PHP, Ruby, etc.
  - *Object-Oriented*: Smalltalk, Eiffel, C++, Java, etc.

—Slide 10—

•

- *Functional Languages*: Use a computational model based on the recursive definition of functions. Based on λ-*calculus*, that formalizes operations on function: Developed by Alonzo Church (1930). Examples: Lisp, ML, Haskell.
- *Data Flow Languages*: Use a computational model based n the flow of information (*tokens*) among primitive functional *nodes*. An inherently parallel (distributed, concurrent) model... Examples: Id, Val, Sisal.

—Slide 11—

•

- *Logic or Constraint-Based Languages*: Use a computational model based on predicate logic. A specified relationship is described; an inference engine attempts to find values that satisfy the specification. Example: Prolog, Excel, VisiCalc or Lotus 1-2-3.
- von Neumann Languages: Use a computational model (Turing Machine) based on configurations and commands to transform configuration. ("Stuff doing stuff to other stuff.") Primarily, based on statements (assignments, etc.) that influence subsequent computations via the *side effects* of changing the value of memory. Examples: Fortran, Ada 83, C, etc.

—Slide 12—

•

- *Object Oriented Languages*: Closely related to vN languages, but have a much more structured and distributed model of both memory and computation. They use a model involving semi-independent *objects*, each of which has both its internal states and methods/subroutines to manage those states. Example: Smalltalk, C++, Java, CLOS (Common Lisp Object Systems), etc.
- *Scripting Languages*: A subset of vN languages; work by "gluing together" components originally developed as independent programs. Examples: csh, bash, Awk, Java Script, PHP, Perl, Python, Ruby, Tcl, etc.

—Slide 13—

Criteria for Selecting a Language

- *Expressive Power*: Church-Turing Thesis: All languages are equally powerful; however, each one may give an advantage in describing a language over another; Ability to write clear, concise, readable, maintainable code for large systems... Examples of features in Common Lisp or Ada.
- *Ease of Use (by a novice)*: Learnability. Created out of few orthogonal set of facilities that can be combined in a powerful way. Example: Java is much simpler than C++, and is almost as easy to learn as Pascal.

—Slide 14—

•

- *Ease of Implementation*: Certain languages are designed to be very close to machine level and can be implemented very easily; Examples: C, Bliss, Pascal (through P-code), Scheme, etc.
- *Efficiency/ Excellent Compilers*: Ability to design good, portable, optimizing compilers; Examples: Java, Common Lisp, etc.

—Slide 15—

•

- *Open Source*: Existence of a user community that contribute to evolve the language to suite the changing needs of the community. Examples: C/Unix,
- *Inertia, Economics, Patronage, etc.*: Commercially backed to achieve market monopoly. Examples: Ada(DoD), PL1 (IBM), Bliss (DEC), C# (Microsoft), etc.

—Slide 16—

Why study languages?

- *See a common theme running through the language design landscape*: Despite apparent differences, all languages are put together out of few common features!
- *Understand obscure features*: Languages have obscure features (rendezvous in Ada with certain communication features), but they make sense in certain contexts!
- *Choose among alternative ways to express things*: Languages allow you to develop patterns, idioms and styles depending on how you wish your code to be understood by other users (readability) or by the compiler (efficiency)!

—Slide 17—

•

- *Make good use of debuggers, assemblers, linkers and other tools*: Understanding the implementation details may help you to avoid or correct a bug!
- *Simulate useful features in languages that lack them*: An algorithm may have a natural structure that has to be tailored to fit the features of a language in as natural a way as possible! Writing Quicksort in an early Fortran!
- *Make better of language technology wherever they appears*: You may need to use a language technology to describe a specific set of ideas! Designing a XML specification for bioinformatics microarray data.

—Slide 18—

Rationale

Do not forget that languages are not designed rationally! Most of the languages one studies are flawed, ad hoc, put together by committees, and left neglected! So, in them, do not look for any sign of "intelligent design."

Programming languages may have hindered the intellectual growth of computer science as a discipline more than anything else.

—Slide 19—

•

Historical forces that have affected programming languages...

- **Evolution**: Computer science is still a young field, and is experimenting with new ideas: structured programming, object oriented programming, etc.
- **Special Purposes**: Some languages were designed to solve a specific problem, but then influenced unrelated sub-disciplines. Simula was designed to simulate real objects and their interactions, but gave rise to OOP!
- **Personal Preference**: Language as theology. Different people like different things, but often most vocal people win!

[End of Lecture #2.5]