Compilation I

Hwansoo Han
Language Groups

- **Imperative**
  - von Neumann (Fortran, Pascal, Basic, C)
  - Object-oriented (Smalltalk, Eiffel, C++)
  - Scripting languages (Perl, Python, JavaScript, PHP)

- **Declarative**
  - Functional (Scheme, ML, pure Lisp, FP)
  - Logic, constraint-based (Prolog, VisiCalc, RPG)
Why so many programming languages?

- Evolution
  - We've learned better ways of doing things over time

- Socio-economic factors
  - Proprietary interests, commercial advantage

- Orientation toward special purposes
- Orientation toward special hardware

- Diverse ideas about what is pleasant to use
Pure Compilation

- Compiler
  - translates the high-level source program into an equivalent target program (typically in a machine language)
  - not used while running program
Pure Interpretation

- Interpreter
  - stays around for the execution of the program
  - the center of control during execution
Compilation vs. Interpretation (1)

- Compilation vs. interpretation
  - Not opposites
  - Not a clear-cut distinction

- Interpretation:
  - Greater flexibility
  - Better diagnostics (error messages)

- Compilation
  - Better performance
Compilation vs. Interpretation (2)

- **Common case**
  - Compilation or simple pre-processing, followed by interpretation

- **Most language implementations include a mixture of both compilation and interpretation**
Compilation vs. Interpretation (3)

- Note that compilation does NOT have to produce machine language for a hardware

- Compilation
  - is *translation* from one language into another,
  - with full analysis of the meaning of the input

- Compilation
  - entails semantic *understanding* of what is being processed,
  - but a pre-processing does not
Compilation vs. Interpretation (4)

- Many compiled languages have interpreted pieces,
  - e.g., formats in C or Fortran

- Most use “virtual instructions”
  - set operations in Pascal
  - string manipulation in Basic

- Some compilers produce only virtual instructions
  - e.g., Java bytecode, Microsoft CIL, Pascal P-code
Implementation Strategies (1)

- Preprocessor
  - Removes comments and white spaces
  - Groups characters into tokens
    - e.g., keywords, identifiers, numbers, symbols
  - Expands abbreviations in the style of a macro assembler
  - Identifies higher-level syntactic structures
    - e.g., loops, subroutines
Implementation Strategies (2)

- Library of Routines and Linking
  - Compiler uses a *linker* program to merge the appropriate *library* of subroutines into the final program
  - e.g., math functions such as $\sin$, $\cos$, $\log$, etc

```
Source program

Compiler

Incomplete machine language  Library routines

Linker

Machine language program
```
Implementation Strategies (3)

- **Post-compilation Assembly**
  - Facilitates debugging
    - assembly languages easier for human than machine binaries
  - Isolates the compiler from changes in the format of machine language files
    - only assembler must be changed, and shared by many compilers
The C Preprocessor (conditional compilation)
- deletes portions of code,
- which allows several versions of a program to be built from the same source

```
#if _X86_
A=...
#elif _ARM_
B=...
#else
C=...
#endif
-D _X86_
A=...
```
Implementation Strategies (5)

- Source-to-Source Translation (C++)
  - C++ implementations based on the early AT&T compiler generated an intermediate program in C, instead of an assembly language:
Implementation Strategies (6)

- **Bootstrapping**

Pascal to machine language compiler (in Pascal)

Pascal to P-code compiler (in Pascal)

P-code interpreter

Pascal to machine language compiler (in P-code)

P-code interpreter

Pascal to machine language compiler (in machine language)
Implementation Strategies (7)

- Compilation of Interpreted Languages
  - Some features of IL are not finalized until runtime
    - Late binding
  - Generates code with assumptions on runtime decision
    - If these assumptions are valid, the code runs very fast
    - If not, a dynamic check will revert to the interpreter.
Implementation Strategies (8)

- Dynamic and Just-in-Time Compilation
  - Deliberately delay compilation until the last possible moment

- Dynamic compilation
  - Lisp or Prolog invoke the compiler on the fly,
    - Translate newly created source into machine language, or
    - Optimize the code for a particular input set

- Just-in-time (JIT) compilation
  - Java defines a machine-independent intermediate form (bytecode)
  - C# compiler produces Common Intermediate Language (CIL)
  - Bytecode, CIL are translated into machine code immediately prior to execution
Implementation Strategies (9)

- **Microcode (firmware)**
  - Assembly-level instruction set is not implemented in hardware, but runs on an interpreter
  - Interpreter is written in low-level instructions (microcode or firmware), which are stored in read-only memory (ROM) and executed by the hardware
  - Popular in machines before the mid 1980s
Compilation vs. Interpretation

- Compilers exist for some interpreted languages, but they aren't pure:
  - Selective compilation of compilable pieces and extra-sophisticated pre-processing of remaining source
  - Interpretation of parts of code, at least, is still necessary for reasons above

- Unconventional compilers
  - Text formatters (TEX, troff)
  - Query language processors
  - Silicon compilers
# Programming Environment Tools

## Tools

<table>
<thead>
<tr>
<th>Type</th>
<th>Unix examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editors</td>
<td>vi, emacs</td>
</tr>
<tr>
<td>Pretty printers</td>
<td>cb, indent</td>
</tr>
<tr>
<td>Pre-processors (esp. macros)</td>
<td>cpp, m4, watfor</td>
</tr>
<tr>
<td>Debuggers</td>
<td>adb, sdb, dbx, gdb</td>
</tr>
<tr>
<td>Style checkers</td>
<td>lint, purify</td>
</tr>
<tr>
<td>Module management</td>
<td>make</td>
</tr>
<tr>
<td>Version management</td>
<td>sccs, rcs, cvs, subversion</td>
</tr>
<tr>
<td>Assemblers</td>
<td>as</td>
</tr>
<tr>
<td>Link editors, loaders</td>
<td>ld, ld-so</td>
</tr>
<tr>
<td>Perusal tools</td>
<td>more, less, od, nm</td>
</tr>
<tr>
<td>Program cross-reference</td>
<td>ctags</td>
</tr>
</tbody>
</table>