Chapter 1 :: Introduction

Programming Languages

• What programming languages can you name?
• Which do you know?
Introduction

• Why are there 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

Introduction

• What makes a language successful?
  – easy to learn (BASIC, Pascal, LOGO, Scheme, Alice)
  – easy to express things, easy use once fluent, "powerful" (C, Common Lisp, APL, Algol-68, Perl)
  – easy to implement (BASIC, Forth)
  – possible to compile to very good (fast/small) code (Fortran)
  – backing of a powerful sponsor (COBOL, PL/1, Ada, Visual Basic, C#)
  – wide dissemination at minimal cost (Pascal, Turing, Java, Alice)
Introduction

• Why do we have programming languages?
  What is a language for?
  – way of thinking -- way of expressing algorithms
  – languages from the programmer’s point of view
  – abstraction of virtual machine -- way of specifying what you want the hardware to do without getting down into the bits
  – languages from the implementor’s point of view
Why study programming languages?

• Help you choose a language.
  – C vs. Modula-3 vs. C++ for systems programming
  – Fortran vs. APL vs. Ada for numerical computations
  – Ada vs. Modula-2 for embedded systems
  – Common Lisp vs. Scheme vs. ML for symbolic data manipulation
  – Java vs. C/CORBA for networked PC programs

Why study programming languages?

• Make it easier to learn new languages some languages are similar; easy to walk down family tree
  – concepts have even more similarity; if you think in terms of iteration, recursion, abstraction (for example), you will find it easier to assimilate the syntax and semantic details of a new language than if you try to pick it up in a vacuum. Think of an analogy to human languages: good grasp of grammar makes it easier to pick up new languages (at least Indo-European).
Why study programming languages?

• Help you make better use of whatever language you use
  – understand obscure features:
    • In C, help you understand unions, arrays & pointers, separate compilation, varargs, catch and throw
    • In Common Lisp, help you understand first-class functions/closures, streams, catch and throw, symbol internals

Why study programming languages?

• Help you make better use of whatever language you use (2)
  – understand implementation costs: choose between alternative ways of doing things, based on knowledge of what will be done underneath:
    – use simple arithmetic e.g. (use \(x^2\) instead of \(x**2\))
    – use C pointers or Pascal "with" statement to factor address calculations
    – avoid call by value with large data items in Pascal
    – avoid the use of call by name in Algol 60
    – choose between computation and table lookup (e.g. for cardinality operator in C or C++)
Why study programming languages?

• Help you make better use of whatever language you use (3)
  – figure out how to do things in languages that don't support them explicitly:
    • lack of suitable control structures in Fortran
    • use comments and programmer discipline for control structures
    • lack of recursion in Fortran, CSP, etc
    • write a recursive algorithm then use mechanical recursion elimination (even for things that aren't quite tail recursive)

Why study programming languages?

• Help you make better use of whatever language you use (4)
  – figure out how to do things in languages that don't support them explicitly:
    – lack of named constants and enumerations in Fortran
    – use variables that are initialized once, then never changed
    – lack of modules in C and Pascal use comments and programmer discipline
    – lack of iterators in just about everything fake them with (member?) functions
Language Categories

• Two common language groups
  
  — Imperative
    • von Neumann (Fortran, Pascal, Basic, C)
    • object-oriented (Smalltalk, Eiffel, C++, Java)
    • scripting languages (Perl, Python, JavaScript, PHP)
  
  — Declarative
    • functional (Scheme, ML, pure Lisp, FP)
    • logic, constraint-based (Prolog, VisiCalc, RPG)

Imperative languages

• Imperative languages, particularly the von Neumann languages, predominate
  
  — They will occupy the bulk of our attention

• We also plan to spend time on functional, logic languages
Compilation vs. Interpretation

• Compilation vs. interpretation
  – not opposites
  – not a clear-cut distinction

• Pure Compilation
  – The compiler translates the high-level source program into an equivalent target program (typically in machine language), and then goes away:

  ![Diagram of Pure Compilation]

• Pure Interpretation
  – Interpreter stays around for the execution of the program
  – Interpreter is the locus of control during execution

  ![Diagram of Pure Interpretation]
Compilation vs. Interpretation

• Interpretation:
  – Greater flexibility
  – Better diagnostics (error messages)

• Compilation
  – Better performance

Compilation vs. Interpretation

• Common case is compilation or simple pre-processing, followed by interpretation
• Most language implementations include a mixture of both compilation and interpretation

Source program → Translator → Intermediate program

Intermediate program → Virtual machine → Output
Compilation vs. Interpretation

• Note that compilation does NOT have to produce machine language for some sort of 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; pre-processing does not
• A pre-processor will often let errors through. A compiler hides further steps; a pre-processor does not

Compilation vs. Interpretation

• Many compiled languages have interpreted pieces, e.g., formats in Fortran or C
• Most use “virtual instructions”
  – set operations in Pascal
  – string manipulation in Basic
• Some compilers produce nothing but virtual instructions, e.g., Pascal P-code, Java byte code, Microsoft COM+
Compilation vs. Interpretation

• Implementation strategies:
  – Preprocessor
    • Removes comments and white space
    • Groups characters into *tokens* (keywords, identifiers, numbers, symbols)
    • Expands abbreviations in the style of a macro assembler
    • Identifies higher-level syntactic structures (loops, subroutines)

Compilation vs. Interpretation

• Implementation strategies:
  – Library of Routines and Linking
    • Compiler uses a *linker* program to merge the appropriate *library* of subroutines (e.g., math functions such as sin, cos, log, etc.) into the final program:
Compilation vs. Interpretation

• Implementation strategies:
  – Post-compilation Assembly
    • Facilitates debugging (assembly language easier for people to read)
    • Isolates the compiler from changes in the format of machine language files (only assembler must be changed, is shared by many compilers)

![Diagram](source_program -> Compiler -> Assembly language)

Compilation vs. Interpretation

• Implementation strategies:
  – The C Preprocessor (conditional compilation)
    • Preprocessor deletes portions of code, which allows several versions of a program to be built from the same source

![Diagram](source_program -> Preprocessor -> Modified source program)

Modified source program -> Compiler -> Assembly language
Compilation vs. Interpretation

• Implementation strategies:
  – 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:

```
Source program → Preprocessor → Modified source program
Modified source program → C++ compiler → C code
C code → C compiler → Assembly language
```

Compilation vs. Interpretation

• Implementation strategies:
  – Bootstrapping
• Early Pascal compilers built around a set of tools that included:
  – A Pascal compiler, written in Pascal, that would generate output in P-code, a simple stack-based language
  – A Pascal compiler already translated into P-code
  – A P-code interpreter, written in Pascal

```
Compiler.p
Compiler.pcode
Interpreter.p
P-code interpreter translated to C
We have to write this
Interpreter.exe
```
Pascal Interpreter

Compiler.pcode → Interpreter.exe → Program.p

Program.pcode → Interpreter.exe → Output of Program.p

Bootstrap compiler

Modify Compiler.p to compile to native code instead of P-code, then use the compiler to compile itself

Compiler.p → Compiler.p to x86 run via Interpreter → X86 Compiler.exe

Program.p → Program.exe
Compilation vs. Interpretation

• Implementation strategies:
  – Compilation of Interpreted Languages
    • The compiler generates code that makes assumptions about decisions that won’t be finalized until runtime. If these assumptions are valid, the code runs very fast. If not, a dynamic check will revert to the interpreter.

• Implementation strategies:
  – Dynamic and Just-in-Time Compilation
    • In some cases a programming system may deliberately delay compilation until the last possible moment.
      – Lisp or Prolog invoke the compiler on the fly, to translate newly created source into machine language, or to optimize the code for a particular input set.
      – The Java language definition defines a machine-independent intermediate form known as byte code. Byte code is the standard format for distribution of Java programs.
      – The main C# compiler produces .NET Common Language Runtime (CLR), which is then translated into machine code immediately prior to execution.
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
  - silicon compilers
  - query language processors

Programming Environment Tools

- Tools; Integrated in an Integrated Development Environment (IDE)

<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>ccpp, 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</td>
</tr>
<tr>
<td>Assemblers</td>
<td>as</td>
</tr>
<tr>
<td>Link editors, loaders</td>
<td>ld, ld-So</td>
</tr>
<tr>
<td>Perlual tools</td>
<td>MORE, LESS, CD, NM</td>
</tr>
<tr>
<td>Program cross-reference</td>
<td>ctags</td>
</tr>
</tbody>
</table>
An Overview of Compilation

• Phases of Compilation

An Overview of Compilation

• Scanning:
  – divides the program into "tokens", which are the smallest meaningful units; this saves time, since character-by-character processing is slow
  – we can tune the scanner better if its job is simple; it also saves complexity (lots of it) for later stages
  – you can design a parser to take characters instead of tokens as input, but it isn't pretty
  – scanning is recognition of a regular language, e.g., via DFA (deterministic finite automaton)
An Overview of Compilation

- **Parsing** is recognition of a context-free language, e.g., via Pushdown Automaton (PDA)
  - Parsing discovers the "context free" structure of the program
  - Informally, it finds the structure you can describe with syntax diagrams (the "circles and arrows" in a Pascal manual)

Pascal “Railroad” diagram
An Overview of Compilation

• **Semantic analysis** is the discovery of *meaning* in the program
  
  – The compiler actually does what is called STATIC semantic analysis. That's the meaning that can be figured out at compile time
  
  – Some things (e.g., array subscript out of bounds) can't be figured out until run time. Things like that are part of the program's DYNAMIC semantics

An Overview of Compilation

• **Intermediate form** (IF) done after semantic analysis (*if* the program passes all checks)
  
  – IFs are often chosen for machine independence, ease of optimization, or compactness (these are somewhat contradictory)
  
  – They often resemble machine code for some imaginary idealized machine; e.g. a stack machine, or a machine with arbitrarily many registers
  
  – Many compilers actually move the code through more than one IF
An Overview of Compilation

• **Optimization** takes an intermediate-code program and produces another one that does the same thing faster, or in less space
  – The term is a misnomer; we just improve code
  – The optimization phase is optional

• **Code generation phase** produces assembly language or (sometime) relocatable machine language

An Overview of Compilation

• Certain *machine-specific optimizations* (use of special instructions or addressing modes, etc.) may be performed during or after **target code generation**

• **Symbol table**: all phases rely on a symbol table that keeps track of all the identifiers in the program and what the compiler knows about them
  – This symbol table may be retained (in some form) for use by a debugger, even after compilation has completed
An Overview of Compilation

• Lexical and Syntax Analysis
  – GCD Program (Pascal)

    program gcd(input, output);
    var i, j : integer;
    begin
      read(i, j);
      while i <> j do
        if i > j then i := i - j
        else j := j - i;
      writeln(i)
    end.

An Overview of Compilation

• Lexical and Syntax Analysis
  – GCD Program Tokens

    • Scanning (lexical analysis) and parsing recognize the structure of the program, groups characters into tokens, the smallest meaningful units of the program
An Overview of Compilation

• Lexical and Syntax Analysis
  – Context-Free Grammar and Parsing
    • Parsing organizes tokens into a parse tree that represents higher-level constructs in terms of their constituents
    • Potentially recursive rules known as context-free grammar define the ways in which these constituents combine

An Overview of Compilation

• Context-Free Grammar and Parsing
  – Example (Pascal program)

\[
\text{program} \quad \rightarrow \quad \text{PROGRAM id (id more_ids); block.}
\]

where

\[
\text{block} \quad \rightarrow \quad \text{labels constants types variables subroutines BEGIN stmt more_stmts END}
\]

and

\[
\text{more_ids} \quad \rightarrow \quad , \text{id more_ids}
\]

or

\[
\text{more_ids} \quad \rightarrow \quad \emptyset
\]
An Overview of Compilation

• Context-Free Grammar and Parsing
  – GCD Program Concrete Parse Tree

• Context-Free Grammar and Parsing
  – GCD Program Parse Tree (continued)
An Overview of Compilation

• Syntax Tree
  – GCD Program Abstract Parse Tree

Code Generation

• Naïve MIPS assembly code fragment

```
addiu sp, sp, -32  # Reserve room for local vars
sw ra, 20(sp)      # save return address
jal getint         # read
nop
sw v0, 28(sp)      # store i
jal getint         # read
nop
sw v0, 24(sp)      # store j
lw t6, 28(sp)      # load i to t6
lw t7, 24(sp)      # load j to t7
nop
beq t6, t7, D     # branch if I = J
nop
A: lw t8, 28(sp)   # load I
...```

Figure 1.4: Syntax tree and symbol table for the GCD program.