Chapter 1 :: Introduction

Programming Language Pragmatics

Michael L. Scott

1

Introduction: History

1804 Joseph Marie Jacquard patents card-controlled loom

1842 Ada Lovelace proposes implementation of algorithm for the Analytical Engine

1889 Herman Hollerith tabulation machine patented; revolutionizes the US Census of 1890

1908 Player piano rolls standardized

1945 ENIAC (Electronic Numerical Integrator and Computer), first modern digital computer, completed

1947 Kathleen and Andrew Booth introduce the idea of assembly language

1951 The first commercial digital computers, the Ferranti Mark 1 (UK) and the Remington Rand UNIVAC I (USA) are introduced

1952 The IBM 704 computer is introduced

2

Introduction: History

Programming ENIAC involved rewiring it!

3

Introduction

- early computers (1940s)
  - cost millions of dollars
  - programmed in machine language
    - bit sequences to perform low-level tasks
    - close to hardware
    - tedious
  - machine's time more valuable than programmer's

4

Introduction

- example: Euclid's algorithm for GCD

```
55 89 e5 53  83 ec 04 83  e4 f0 e8 31  00 00 00 89  c3 e8 2a 00
00 00 39 c3  74 10 8d b6  00 00 00 00  39 c3 7e 13  29 c3 39 c3
75 f6 89 1c  24 e8 6e 00  00 00 8b 5d  fc c9 c3 29  d8 eb eb 90
```


5

Introduction

- less error-prone method needed
  - assembly language: binary operations expressed with mnemonic abbreviations

```
    pushl   %ebp                    jle     D
    movl    %esp, %ebp              subl    %eax, %ebx
    pushl   %ebx                B:  cmpl    %eax, %ebx
    subl    $4, %esp                jne     A
    andl    $-16, %esp          C:  movl    %ebx, (%esp)
    call    getint                  call    putint
    movl    %eax, %ebx              movl    -4(%ebp), %ebx
    call    getint                  leave
    cmpl    %eax, %ebx              ret
    je      C                   D:  subl    %ebx, %eax
A:  cmpl    %eax, %ebx              jmp     B
```


6

Introduction

- assembly language is specific to a certain machine, however
  - tedious to re-write code for each computer type
  - machine-independent language desired
  - Fortran (mid-1950s) used a compiler to bridge the gap between high-level language and machine-dependent code
  - many other languages followed:

1957 Fortran	1966 Apl	1980 Ada
1959 Cobol	1967 Snobol 4	1983 Standard ML
1960 Algol 60	1970 Pascal	1987 Haskell
1960 Lisp	1972 C	…
1964 PL/I	1972 Smalltalk	
1964 Basic	1975 Scheme	

7

Introduction

- Why are there so many programming languages?
  - 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

8

Introduction

- What makes a language successful?
  - easy to learn (BASIC, Pascal, Scheme, Python)
  - easy to express things, easy to use once fluent, "powerful" (C, Algol-68, Perl)
  - easy to implement (BASIC)
  - possible to compile to very good (fast/small) code (Fortran)
  - backing of a powerful sponsor (COBOL, PL/1, Ada, C#)
  - wide dissemination at minimal cost (Pascal, Java)

9

Introduction

- Why do we have programming languages? What is a language for?
  - way of thinking -- way of expressing algorithms
  - languages from the user'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

10

Why study programming languages?

- studying programming languages can help you choose the right language for an application
- makes it easier to learn new languages
  - some languages are similar
  - 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)

11

Why study programming languages?

- helps you make better use of whatever language you use
  - understanding implementation costs: choosing between alternative ways of doing things
    - using simple arithmetic (use x*x instead of x**2)
    - avoiding call by value with large data items in C
  - figuring out how to do things in languages that don't support them explicitly
    - lack of recursion in Fortran
      - write a recursive algorithm then use mechanical recursion elimination
    - lack of modules in C and Pascal
      - use comments and programmer discipline
    - lack of iterators in just about everything
      - fake them with (member) functions

12

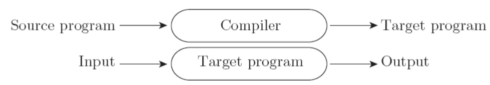
Programming Language Paradigms

- four categories
  - imperative — Fortran, Pascal, Basic, C
  - object-oriented — C++, Java, Smalltalk
  - functional — Haskell, Scheme, Lisp
  - logic — Prolog

13

Compilation vs. Interpretation

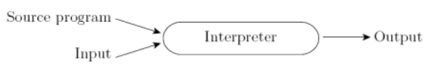
- compilation vs. interpretation
  - not opposites
  - not a clear-cut distinction
- pure compilation
  - compiler translates a high-level source program into an equivalent target program (typically in machine language), and then goes away



14

Compilation vs. Interpretation

- pure interpretation
  - interpreter stays around for the execution of the program
  - interpreter is the locus of control during execution



15

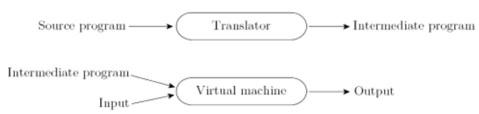
Compilation vs. Interpretation

- interpretation
  - greater flexibility
  - better diagnostics (error messages)
- compilation
  - better performance

16

Compilation vs. Interpretation

- some language implementations include a mixture of both compilation and interpretation



17

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
- unconventional compilers
  - text formatters (LaTex)
  - silicon compilers
  - query language processors

18

Compilation vs. Interpretation

- many compiled languages have interpreted pieces
  - print formats in C
- some compilers produce nothing but virtual instructions
  - Java byte code

19

Compilation vs. Interpretation

- many compilers are self-hosting
  - they are written in the language they compile
  - e.g., C compiler written in C
- how?
  - bootstrapping
  - write small interpreter
  - hand-translate small number of statements into assembly
  - extend through incremental runs of the compiler through itself

20

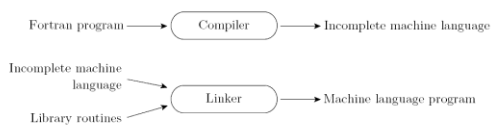
Compilation vs. Interpretation

- 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)
    - a pre-processor will often let errors through

21

Compilation vs. Interpretation

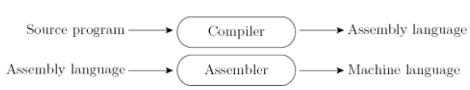
- implementation strategies (cont.)
  - library of routines and linking
    - compiler uses a *linker* program to merge the appropriate *library* of subroutines (e.g., math functions such as sin, cos, log, etc.) into the final program



22

Compilation vs. Interpretation

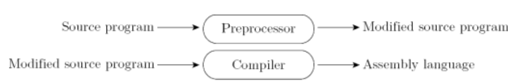
- implementation strategies (cont.)
  - post-compilation assembly
    - facilitates debugging (assembly language easier for people to read)
    - isolates the compiler from changes in the format of machine language files (only assembler must be changed, is shared by many compilers)



23

Compilation vs. Interpretation

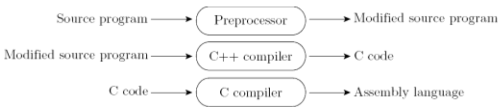
- implementation strategies (cont.)
  - the C preprocessor (conditional compilation)
    - preprocessor deletes portions of code, which allows several versions of a program to be built from the same source



24

Compilation vs. Interpretation

- implementation strategies (cont.)
  - 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



25

Compilation vs. Interpretation

- implementation strategies (cont.)
  - 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
  - compilers exist for some interpreted languages, but not 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

26

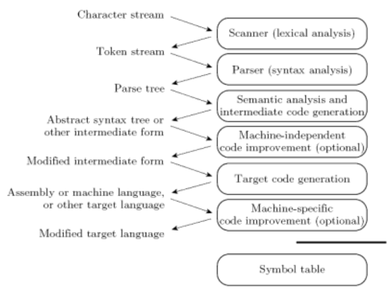
Compilation vs. Interpretation

- implementation strategies (cont.)
  - 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 Intermediate Language (CIL), which is then translated into machine code immediately prior to execution

27

An Overview of Compilation

- Phases of Compilation



28

Course Overview

- Introduction
- Scanning and Parsing
- Imperative Languages: C
- Object-oriented Languages: C++
- Functional Languages: Haskell
- Logic Languages: Prolog

29