Chapter 1. Introduction

1.1 The Art of Language Design

- There are thousands of high-level programming languages, and ***new ones continue to emerge***.
  Why? ➔ ***Evolution***, ***Special purposes***, ***Personal preference***

- What makes a ***language successful***?
  (1) Expressive power
  (2) Ease of use for the novice
  (3) Easy of implementation
  (4) Standardization
  (5) Open source
  (6) Excellent compilers
  (7) Economic, patronage, and inertia

- No single factor determines whether a language is "good".

-We shall need to consider the ***viewpoints of*** both the ***programmer*** and the ***language implementor***.

1.2 The Programming Language Spectrum

- The many existing languages can be classified into families based on their ***model of computation***.

Declarative	
functional	Lisp/Scheme, ML, Haskell
dataflow	Id, Val
logic, constraint-based	Prolog, spreadsheets, SQL
Imperative	
von Neumann	C, Ada, Fortran, …
object-oriented	Smalltalk, Eiffel, Java, …
scripting	Perl, Python, PHP, …

(Note) Categories are fuzzy, and open to debate

- ***Declarative*** languages focus on ***what the computer is to do***. More in tuned with ***programmer's point of view***. ***Imperative*** languages focus on ***how the computers should do it***. Mainly for ***performance reason***.

1.3 Why study Programming Languages?

- A good understanding of language design and implementation can help one ***choose the most appropriate language for any given task***. (Note) Most languages are better for some things than others.

- Make it ***easier to learn new languages***. There are ***basic concepts that underlie all programming languages***: types, control (iteration, selection, recursion, nondeterminancy, concurrency), abstraction, and naming.

- Programmers with a strong grasp of language theory will be in a ***better position to design elegant, well-structured notation*** that meets the needs of current users and facilitates future development.

1.4 Compilation and Interpretation

- The compiler ***translates*** the high-level ***source program*** ***into*** an equivalent ***target program*** (typically in ***machine language***), and then goes away.

- ***The compiler is itself a machine language program***, presumably created by compiling some other high-level program. When written to a file in a format understood by the operating system, machine language is commonly known as ***object code***.

- An ***alternative style*** of implementation for high-level language is known as ***interpretation***. ***The interpreter implements a virtual machine*** whose machine language is the high-level programming language. The interpreter reads statements in that language more or less one at a time, executing them as it goes along.

- ***Compilation generally leads to better performance***. In general, a decision made at compile time is a decision that does not need to be made at runtime.

- Since the (final version of a) program is ***compiled only once***, but generally ***executed many times***, the savings can be substantial.

(note) Compiler—better performance and more efficient
Interpreter—more flexible and better debugging

- ***Most language*** implementations include ***a mixture of both***.

- We generally say that a language is ***interpreted*** when ***the initial translator is simple***. If the ***translator is complicated***, we say that the language is ***compiled*** (: ***subjective and confusing***). ***Thorough analysis*** and ***nontrivial transformation*** are the ***hallmarks of compilation***.
- ***Most interpreted languages employ an initial translator*** (a ***preprocessor***) that removes comments and white space, and groups characters together into tokens (i.e. keywords, identifiers, numbers, and symbols). The translator may identify high-level syntactic structures, such as loops and subroutines.

- The typical ***Fortran*** implementation comes close to ***pure compilation***. The compiler relies on a separate program, known as a "linker", to merge the appropriate routines into the final program:

Library routines

Fortran program Compiler Incomplete machine language Linker

Machine language program

- Many compilers generate ***assembly language instead of machine language***. ***This convention facilitates debugging***, since assembly language is easier for people read, and isolates the compiler from changes in the format of machine language files.

Source program — Compiler — Assembly language — Assembler — Machine language

- Compilers for C (and many other languages running under Unix) begin with a preprocessor that removes comments and expands macros.

Source program — Preprocessor — Modified source program — Compiler — Assembly language

- C++ implementations based on the early AT&T compiler actually generated an intermediate program in C, instead of in assembly language. This C++ compiler was indeed a true compiler, though it generated C instead of assembler.

Source program — Compiler 1 — Alternative source program (e.g., in C) — Compiler 2 — Assembly language

- Many compilers are "***self-hosting***": they are written in the language they compile – ***Ada compilers in Ada***, ***C compilers in C***. This uses a technique known as "***bootstrapping***": one starts with a simple implementation and use it to build progressively more sophisticated versions. For example, if we had a C compiler already, we might start by writing, in a simple subset of C, a compiler for an equally simple subset of Java. Once this compiler was working, we could hand-translate the C code into our subset of Java and run the compiler through itself. We could then repeatedly extend the compiler to accept a larger subset of Java, bootstrap it again, and use the extended language to implement an even larger subset.

- One will sometimes find compilers for languages (e.g. Lisp, Prolog, Smalltalk, etc.) that permit a lot of ***late binding***, and are traditionally ***interpreted***.

- In some cases a programming system may ***deliberately delay compilation until the last possible moment***. More recent implementations of ***Java*** employ a ***just-in-time compiler*** that ***translates byte code into machine language immediately before each execution*** of the program. ***C#***, similarly, is intended for ***just-in-time translation***.

- A compiler does not necessarily translate from a high-level language into machine language. The term "***compilation***" applies whenever we ***translate automatically from one nontrivial language to another, with fully analysis of the meaning of the input***.

1.5 Programming Environments

- Programmers are assisted in their work by a host of other tools.
  (ex) assemblers, preprocessors, linkers, style checkers, configuration management, …

- In ***older programming environments***, ***tools may be executed individually***, at the explicit request of the user.

- ***More recent environments*** provide much more ***integrated tools, IDE (Integrated Development Environment)***. The editor for an IDE may incorporate knowledge of language syntax, providing templates for all the standard control structures, and checking syntax as it is typed in.

1.6 An Overview of Compilation

- In a typical compiler, compilation proceeds through ***a series of well-defined phases***.

  (Note) See ***Figure 1.3*** Phases of compilation and ***Figure 1.4*** Phases of interpretation (***p27*** of the textbook)