Lecture Notes on Programming Languages

Elvis C. Foster

Lecture 01: Introduction to Programming Languages

This lecture contains:

- Rationale
- Brief History of Programming Languages
- Programming Environment and the Compiling Process
- Programming Domains
- Programming Paradigms
- Evaluation criteria for Programming Languages

1.1 Rationale for Studying Programming Languages

Since the 1960s, hundreds of programming languages have been proposed and introduced to the software industry. Most of these languages have faded into oblivion, becoming victims of the competitive nature of the industry. This course probes beyond the superficial features of programming languages to delve into the underlying design concepts that drive their implementation.

A study of programming languages is useful and necessary because the knowledge and expertise gained provide the following benefits:

- Gain useful insights into the intricate design principles that govern programming languages
- Enhance one's ability to develop effective and efficient algorithms
- Improve one's use of existing programming languages
- Increase one's vocabulary of useful programming constructs
- Allow for a better choice of programming languages
- Make it easier to learn a new programming language
- Make it easier to design and construct a programming language
- Improve one's capacity to communicate ideas in computer science
- Better understanding of the significance of language implementation issues

1.2 Brief History of Programming Languages

Chapter 2 of the text provides a detailed history of programming languages. Figure 1.1 provides a brief summary.

Figure 1.1: Summary of the History of Programming Languages

Period	Languages Developed
1950s	FORTRAN, LISP
1960s	Simula, COBOL, RPG, ALGOL, PL1
1970s	Ada, C, Pascal, Prolog, Small Talk
1980s	C++, ML, Eiffel, Visual languages
1990s	Java, Hypermedia languages, Visual languages, Ada 95

In studying and/or specifying programming languages, it is often useful to express syntactic components via the Baccus-Naur Form (BNF). The BNF notation was developed by John Baccus and Peter Naur in 1958, and has become widespread since its introduction. Figure 1.2 shows the symbols used in the notation.

Figure 1.2: BNF Notation Symbols

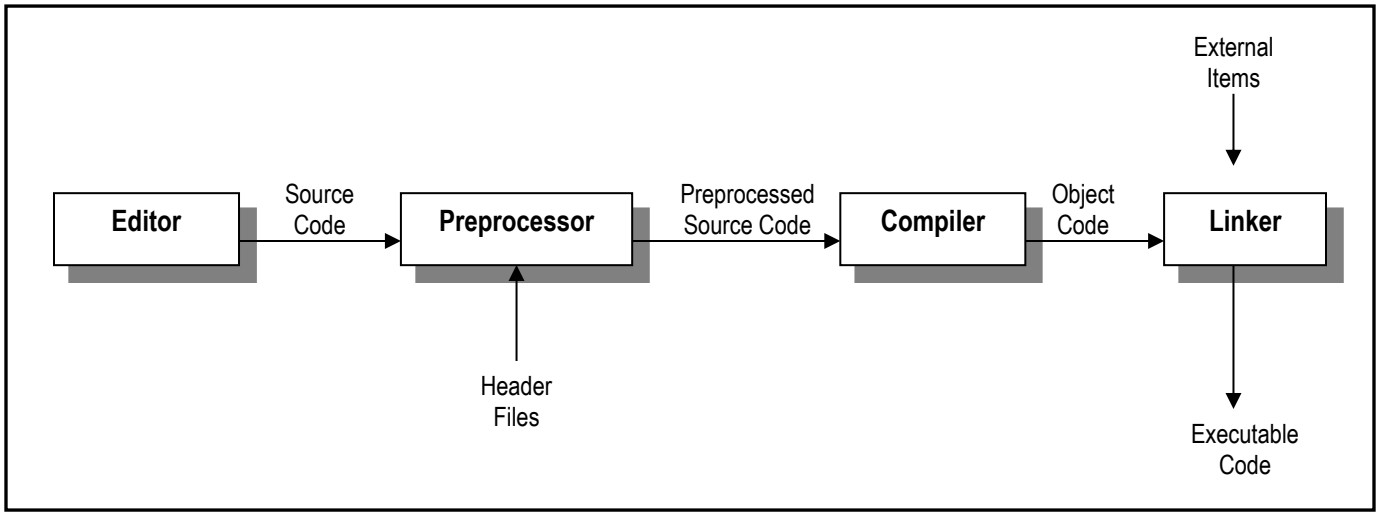
Symbol	Meaning
::=	"is defined as"
[. . .]	Denotes optional content (except when used for array subscripting)
<Element>	Denotes that the content is supplied by the programmer and/or is non-terminal
|	Indicates choice (either or)
{<Element>}	Denotes zero or more repetitions
<Element>*	Alternate notation to denote zero or more repetitions
<l>*<m><Element>	Denotes l to m repetitions of the specified element
[* <Element> *]	Alternate and recommended notation to denote zero or more repetitions for this course

Note: The construct {<Element>} is the original construct for repetition. However, C-based languages use the left curly brace ({) and right curly brace (}) as part of their syntax. To avoid confusion, it has been recommended that for these languages, the construct <l>*<m> <Element> or <Element>* be used. But that too is potentially confusing. Therefore, for this course, we will sometimes use the construct [* <Element> *] to denote zero or more repetitions.

1.3 Programming Environment and the Compilation Process

Your program passes through a number of important stages before it is executed by the computer.
Figure 1.3 illustrates the interrelated components of a programming language environment and the various processes that the program passes through. As can be seen from the figure, a typical programming environment consists of an *editor*, a *preprocessor*, a *compiler* or *interpreter*, a *linkage editor* (also called a linker), and a library of enhancement resources (not shown in the figure). When you install a programming language, all these items are automatically included in a seamless manner. When you run the programming language, you are typically communicating to the editor.

Figure 1.3: Typical Programming Environment



1.3 Programming Environment and Compilation Process (continued)

Editor: The editor is a program that allows the user (programmer) to key in his/her program (source code). The editor may be a traditional line editor, or a graphical editor; this affects to a large extent, your programming environment. Typically, it provides facilities for the following:

- Entering and editing the program
- Loading a program (from disk) into memory
- Compiling the program
- Debugging the program
- Running the program

Preprocessor: The preprocessor is a program that removes all comments from the source code and modifies it according to directives supplied to the program. In a C^{++} environment, a directive begins with the pound symbol (#).

Example 1:

```
#include <iostream>
```

Compiler: The compiler is a program that accepts as input, the preprocessed source code, analyzes it for syntax errors, and produces one of two possible outputs:

- If syntax error(s) is (are) found, an error listing is provided.
- If the program is free of syntax errors, it is converted to object code (assembler language code or machine code).

Note:

1. If the preprocessed code is converted to assembler code, an assembler then converts it to machine code.
2. Machine code varies from one (brand of) machine to the other. Each machine (brand) has an assembler. Assembler language programming is particularly useful in system programming and writing communication protocols. An assembler language is an example of a low level language.

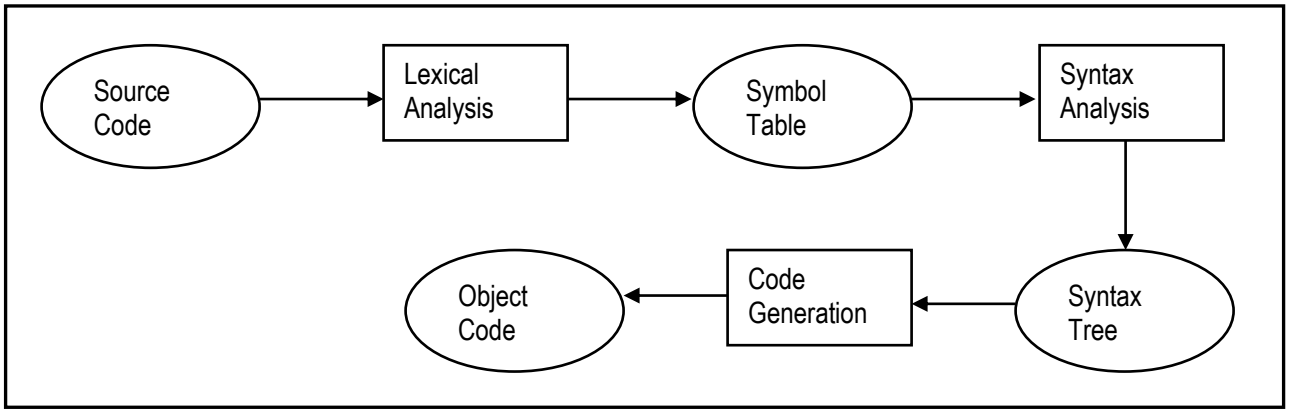
In interpretive languages, an *interpreter* replaces the compiler. The main differences between a compiler and an interpreter are as follows: Firstly, the compilation process is batch-oriented while the interpretation process is more instantaneous. What this means is that the compiler looks at the entire code before attempting to translate the program. If errors are found, a list of error messages is provided; otherwise, the source code is converted to object code. The interpreter on the other hand, examines the code on a command-by-command basis while the code is being written. Feedback to the programmer is more immediate. Secondly, a compiling language tends to be more efficient than an interpretive language. This is so because the object code generated is typically stored for subsequent usage; an interpretive language environment may include a feature to store the previously translated object code but this is not a requirement. Finally, the interpretive language environment tends to provide the programmer with more conveniences than the compiling language environment.

Linker: A linker (linkage editor) is a program that combines all object code of a program with other necessary external items to form an executable program.

1.3 Programming Environment and Compilation Process (continued)

The *compilation process* is the series of stages that the program passes through in order to be converted from *source code* to *object code* (also called *executable code*). These steps are depicted in figure 1.4. As can be seen from the figure, the compilation process involves three stages: *lexical analysis*, *syntax analysis*, and *code generation*. However, do note that the term is often used loosely to refer to the combination of figures 1.3 and 1.4. We will revisit the compilation process later in the course.

Figure 1.4: Compilation Process



1.4 Programming Domains

Computer science affects all aspects of business and life. Because of this, there are different programming languages for different purposes. Among the programming domains identified are scientific application, business application, artificial intelligence (AI) applications, systems programming, scripting languages, and hybrid languages. This section summarizes each.

1.4.1 Scientific Applications

Languages that fall in this category typically have simple data structures, but with the facility to represent very small and very large numbers. ALGOL 60 was the first language for this purpose. Others include Fortran, C, and C++.

1.4.2 Business Applications

Languages that fall in this category tend to support a wider range of data structures, but not as a wide a range for numeric data. COBOL was the first such language. Others include RPG, C++, Java, etc.

1.4.3 Artificial Intelligence Languages

This category of languages includes languages that can be used for cognitive processing. The first widely used AI language was LISP. Others include C++, CLOS, and ML.

1.4.4 Systems Programming

Languages that fall into the category of systems programming languages are typically used for operating system support. Various command languages have been proposed by different operating systems. Other systems use standard C or C++. Below are some examples:

- Unix was written in C, but there are several shell languages
- Linux was written in C++, but there are several shell languages
- OS-400 has its own control language
- Windows was written in C and C++, but there are various script languages

1.4.5 Scripting Languages

Scripting languages are widely used for Web programming. They have non-procedural features and easy to learn. Examples include HTML, Java Script, PHP, Perl, XML, etc.

1.4.6 Hybrid Languages

Hybrid languages are languages that are used across multiple programming domains. Examples include Java, C++, Ada, and C#.

1.5 Programming Paradigms

Programming languages may also be classified based on their design characteristics. Following are five classifications:

Procedural Languages: Also called imperative languages, these languages represent the procedural programming paradigm. Examples include C, Pascal, COBOL, Fortran, RPG-400, ALGOL, PL1, FoxPro, etc. — most of the traditional languages.

Object-Oriented Languages: Object-oriented programming languages (OOPLs) represent the object-oriented paradigm. Examples include Eiffel, Small Talk, Simula, Scalable Application Language (SAL), Java, C#, Ruby, etc.

Rule-based Languages: These languages are widely used in AI applications. Examples include LISP, CLOS, ML, Haskell, Scheme, etc.

Hybrid Languages: These languages represent procedural as well as OO programming. Examples include C++, Object Pascal, Python, Object COBOL, Ada, etc.

Declarative Languages: Declarative languages are more sophisticated (and powerful) than other types of programming languages. However, they are also more limited in their scope. Declarative languages are typically used to manage databases (DB) and knowledge bases (KB). They are higher level languages than third generation languages; in fact, most 4GLs are declarative languages. Examples of declarative languages are SQL (Structured Query Language), Ideal, KQL (Knowledge Query Language). .Arguably, we may classify hypermedia markup languages as declarative; or we may classify them as being part of the fifth generation languages. Examples of these include HTML (Hypertext Markup Language), VRML (Virtual Reality Markup Language), and XML (Extensible Markup Language).

1.6 Criteria for Evaluating Programming Languages

The following criteria are useful in evaluating programming languages: readability, simplicity, orthogonality, support for abstraction, program verification, programming environment, portability, use of reserve words, type checking, and usage cost.

1.6.1 Readability

Readability (also called understandability) includes clarity, and consistency. Clarity relates to how readable the code is. Are there cryptic keywords and constructs, or are they easy to remember? For example, C++ tends to be a cryptic language.

Consistency: Are the rules consistent, or are there many exceptions to rules? For example, C++ has few rules but several exceptions to these rules. For instance, ***** could be used as a multiplication symbol, a pointer declaration, or as an indirection.

1.6.2 Simplicity

Simplicity also affects the understandability of the language, and how easy it is to learn.

- A language that has a large number of basic components is more difficult to learn than one with fewer components.
- Multiplicity of features provide flexibility, but can lead to increased difficulty in learning the language. For example, in C++ and Java, you can increment a variable in four ways:
  x = x+1; x += 1; x++; ++x;
- Operator overloading, though powerful, is potentially confusing. This feature is supported in C++, but elegantly circumvented via the **Object** class in Java.

1.6.3 Orthogonality

By orthogonality, we mean that the primitive features (particularly data types) are independent of the context in which they can be used. To illustrate, suppose that a language has x primitive data types and y type operators. If the language is orthogonal, each operator can be applied to each data type. In other words, the language is symmetrical.

A good example of this symmetry is in SQL, where many or the commands (for instance CREATE, ALTER, DROP) are orthogonal to database objects (such as tables, views, user accounts, indexes, etc.). This feature facilitates easier learning of the language.

1.6.4 Support for Abstraction

Does the language allow programmers to create advanced (programmer-defined) data structures easily? This is desirable. C is not particularly outstanding in this area. C++ and Java are much better. RPG-400 is quite useful in this area also.

1.6.5: Program Verification

Does the language provide proof of program correctness via formal or informal methods? This affects its usefulness.

1.6.6 Programming Environment

Usefulness of the programming environment is also important. Does the programming environment provide useful, user-friendly features (such as debugging, tracing, etc.) for the programmer? Is the language an interpretive or compiler-based? The latter is more efficient; the former is more user friendly.

1.6.7 Portability

How portable is the language? C++ and Java are very portable languages for different reasons —C++ due to its availability on all major operating systems, and Java due to its Java virtual machine (JVM).

1.6.8 Control Statements

Does the language support the standard control structures in a non-ambiguous, easy-to-understand way? We will discuss ambiguity later in the course.

1.6.9 Use of Reserve Words

Does the language provide a large list of reserve words? The more reserve words there are, the harder it is to remember them, and thereby gain mastery in the language.

1.6.10 Type Checking

Does the language have a stringent type checking mechanism at compile time rather than execution time? The former is less costly and therefore more desirable than the latter. For example, Java and RPG-400 are excellent examples of languages that support early type checking; C and C++ are notorious culprits for violating this principle.

1.6.11 Exception Handling

Does the language provide adequate exception handling features? C++, Java, and Ada are excellent examples; C is notorious.

1.6.12 Input/output Processing

How does the language support input and output of data? How easy is it to do this? How are files processed?

1.6.13 Usage Cost

The usage cost of a programming language is comprised of several constituents:

a. **Training Cost:** This is significantly increased if the language is difficult to learn.
b. **Development Cost:** If it is difficult to use the language for software development, this is not desirable. This factor is also influenced by the complexity of the language.
c. **Compilation Cost:** If the compiler is not efficient in its operation, this creates a drag on the system, particularly for large programs.
d. **Program Execution Cost:** If the language requires many run-time checks, this could significantly inhibit efficient execution.
e. **Marketing Cost:** Languages that are free or relatively inexpensive will attract more takers, especially if the product is good. Two excellent cases in point is Java and C++.
f. **Unreliability Cost:** Failure of a system due to language unreliability can be quite costly.
g. **Maintenance Cost:** If the language is difficult to learn and use,, this will affect the maintainability of systems developed with the language.

1.7 Summary and Concluding Remarks

Let us summarize what has been covered in this lecture:

- Studying of programming languages in order to gain useful insights about language design principles, as well as to improve one's ability to learn, critique, and select appropriate programming languages for different scenarios.
- Programming languages have been through several eras of development — from the days of Fortran to the current era with languages such as Java, C#, Ruby, etc.
- The BNF notation is a widely used notation for specifying the syntax of programming languages.
- A programming environment typically consists of a language editor, a preprocessor, a compiler or interpreter, and a linker.
- The compilation process consists of three main areas, namely, lexical analysis, syntax analysis, and code generation.
- Among the programming domains identified are scientific application, business application, artificial intelligence (AI) applications, systems programming, scripting languages, and hybrid languages.
- The main programming paradigms than languages fall in are procedural programming languages, object-oriented programming languages, rule-based programming languages, and hybrid programming languages.
- The main programming domains are scientific applications, business applications, artificial intelligence (AI) applications, systems programming, scripting programming, hybrid programming.
- The main criteria to look for when studying or critiquing a language are readability, simplicity, orthogonality, support for abstraction, problem verification, programming environment, portability, control structures, reserve words, exception handling, input/out processing, and usage cost.

Mastery in studying languages based on these criteria is a very useful skip for the computer science professional. Subsequent lectures will delve more deeply into these concepts.

1.8 Recommended Readings

[Parsons 1992] Parsons, Thomas W. 1992. *Introduction to Compiler Construction*. New York: W.H. Freeman and Company. See chapter 1.

[Pratt & Zelkowitz 2001] Pratt, Terrence W. and Marvin V. Zelkowits. 2001. *Programming Languages: Design and Implementation* 4th Edition. Upper Saddle River, NJ: Prentice Hall. See chapter 1.

[Sebesta 2012] Sebesta, Robert W. 2012. *Concepts of Programming Languages* 10th Edition. Colorado Springs, Colorado: Pearson. See chapters 1 and 2.

[Webber 2003] Webber, Adam B. 2003. *Modern Programming Languages: A Practical Introduction*. Wilsonville, Oregon: Franklin, Beedle & Associates. See chapters 1, 2, & 24.