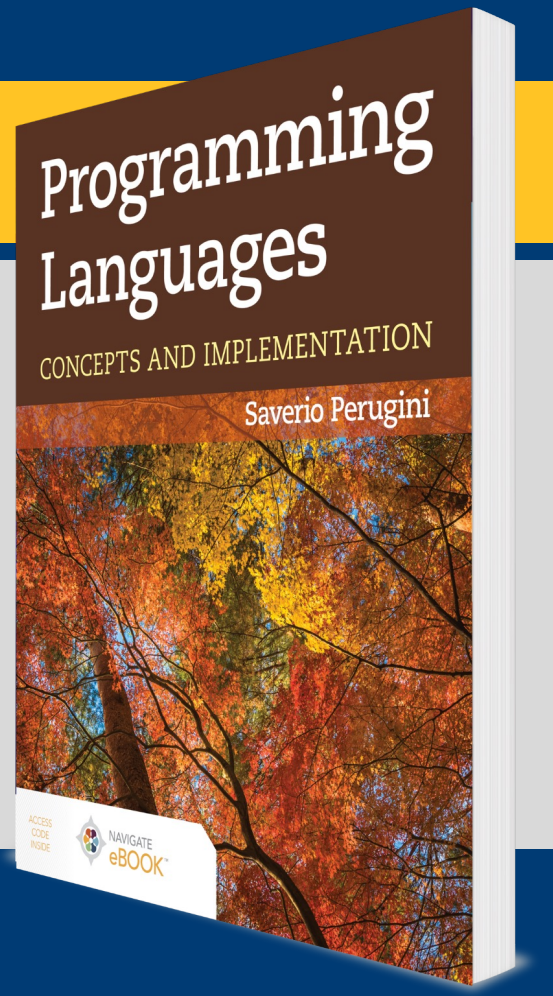
CHAPTER 4

Programming Language Implementation

Additional Material: Brian Hare, UMKC



Chapter 4: Programming Language Implementation

So you are interpreters of interpreters?

—Socrates, *Io*

The front end of a programming language implementation consists of a scanner and a parser. The output of the *front end* is typically an abstract-syntax tree. The actions performed on that abstract-syntax tree determine whether the language implementation is an *interpreter* or a *compiler*, or a combination of both—the topic of this chapter.

Outline

- **4.1 Chapter Objectives**
- 4.2 Interpretation Vis-à-Vis Compilation
- 4.3 Run-Time Systems: Methods of Executions
- 4.4 Comparison of Interpreters and Compilers
- 4.5 Influence of Language Goals on Implementation
- 4.6 Thematic Takeaways

4.1 Chapter Objectives

- Describe the differences between a *compiler* and an *interpreter*.
- Explore a variety of *implementations* for programming languages.

Outline

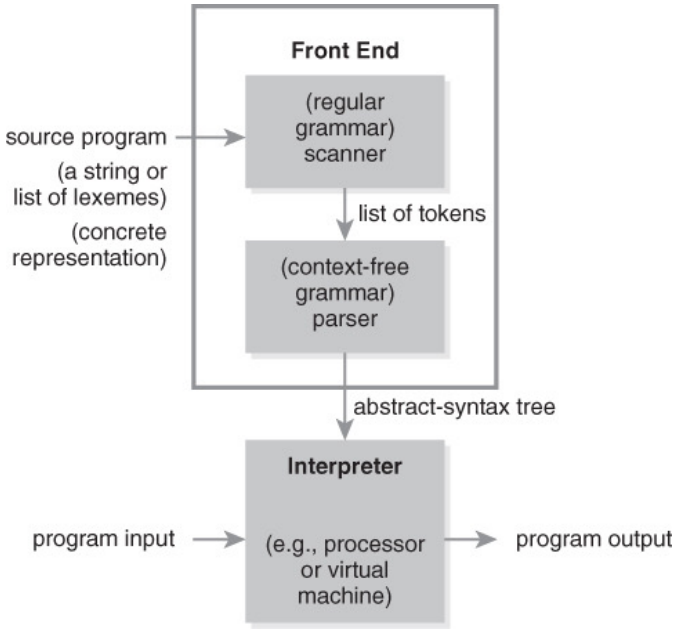
- 4.1 Chapter Objectives
- **4.2 Interpretation Vis-à-Vis Compilation**
- 4.3 Run-Time Systems: Methods of Executions
- 4.4 Comparison of Interpreters and Compilers
- 4.5 Influence of Language Goals on Implementation
- 4.6 Thematic Takeaways

4.2 Interpretation Vis-à-Vis Compilation

- Both compilers and interpreters have a front end, which consists of a scanner (lexical analyzer) and parser (syntactic analyzer).
- Interpreter
  - An *interpreter* is a software simulation of machine which natively (i.e., no translation involved) understands instructions in the source language.
  - An interpreter provides a virtual machine for a programming language.
- Compiler
  - A *compiler* is a program that translates a program in one language (the source language) to an equivalent program in another language (the target language).
  - A compiler is just a translator, nothing more.

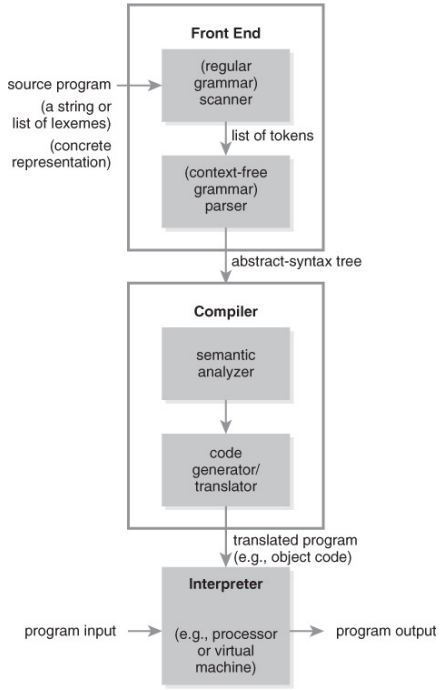
Figure 4.1 Execution by Interpretation

- Preprocessing (purges comments)
- Lexical analysis (scanning)
- Syntax analysis (parsing)
- Semantic analysis



Data from Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. 2001. *Essentials of Programming Languages*. 2nd ed. Cambridge, MA:MIT Press.

Figure 4.2 Execution by Compilation



Data from Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. 2001. *Essentials of Programming Languages*. 2nd ed. Cambridge, MA:MIT Press.

Levels of Languages

4. fourth-generation language (e.g., `lex` and `yacc`)

↓

3. high-level language (e.g., Python, Java, and Scheme)

↓

2. assembly language (e.g., MIPS)

↓

1. machine language (e.g., x86)

Simple Interpreter

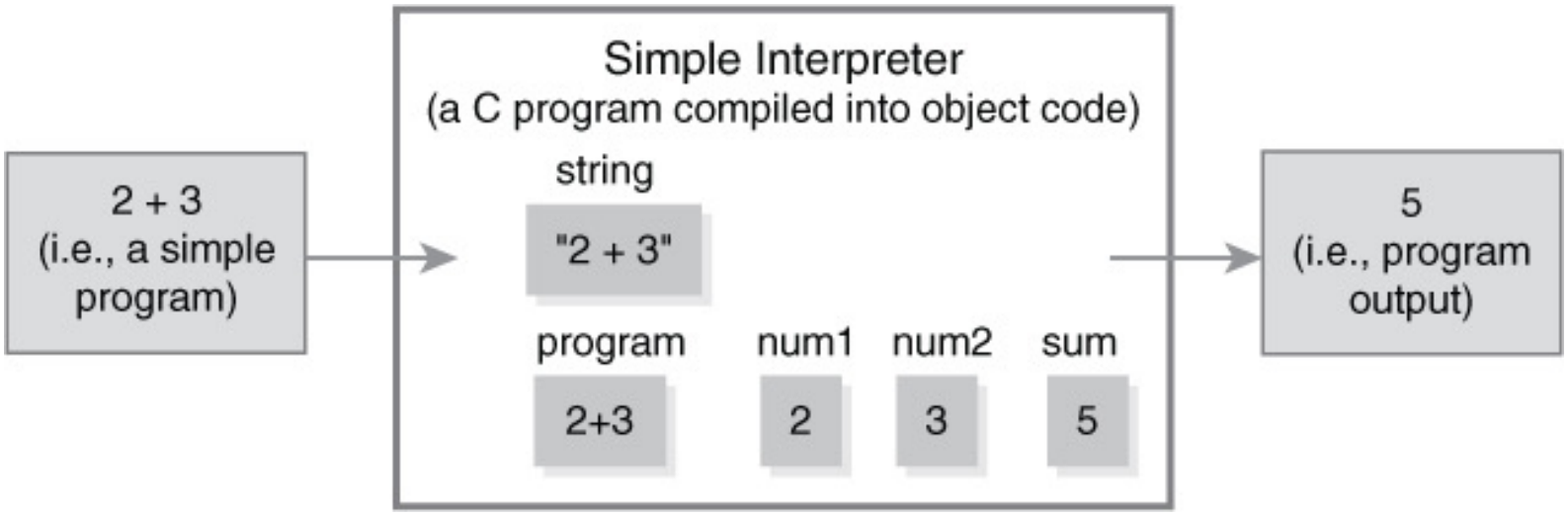


Figure 4.3 Interpreter for the language *simple*, illustrating that the simple program becomes part of the running interpreter process.

Outline

- 4.1 Chapter Objectives
- 4.2 Interpretation Vis-à-Vis Compilation
- **4.3 Run-Time Systems: Methods of Executions**
- 4.4 Comparison of Interpreters and Compilers
- 4.5 Influence of Language Goals on Implementation
- 4.6 Thematic Takeaways

4.3 Run-Time Systems: Methods of Executions

1. Traditional compilation directly to object code (e.g., Fortran, C)
2. Hybrid systems: interpretation of a compiled, final representation through a compiled interpreter (e.g., Java, Python, Perl)
3. Pure interpretation of a source program through a compiled interpreter (e.g., Scheme, ML)
4. Interpretation of either a source program or a compiled final representation through a stack of interpreted software interpreters

Figure 4.4 Low-Level View of Execution by Compilation

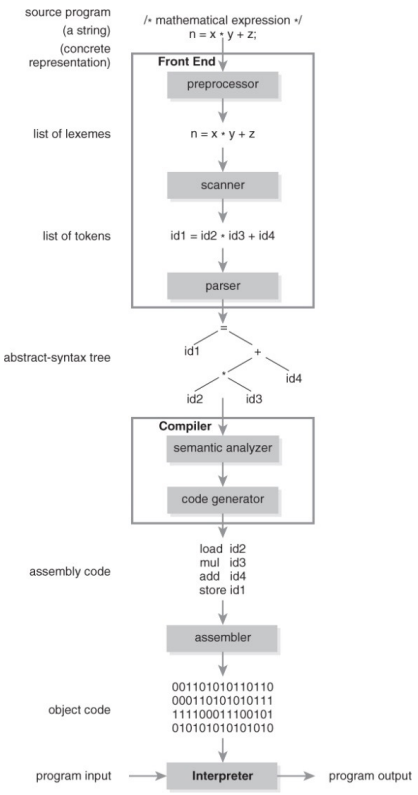


Figure 4.5 Alternative View of Execution by Interpretation

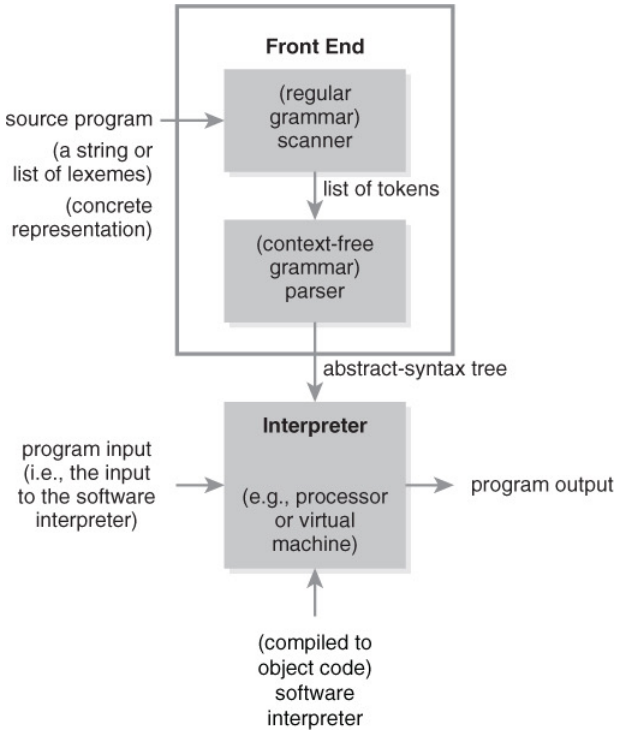


Figure 4.6 Four Different Approaches to Language Implementation

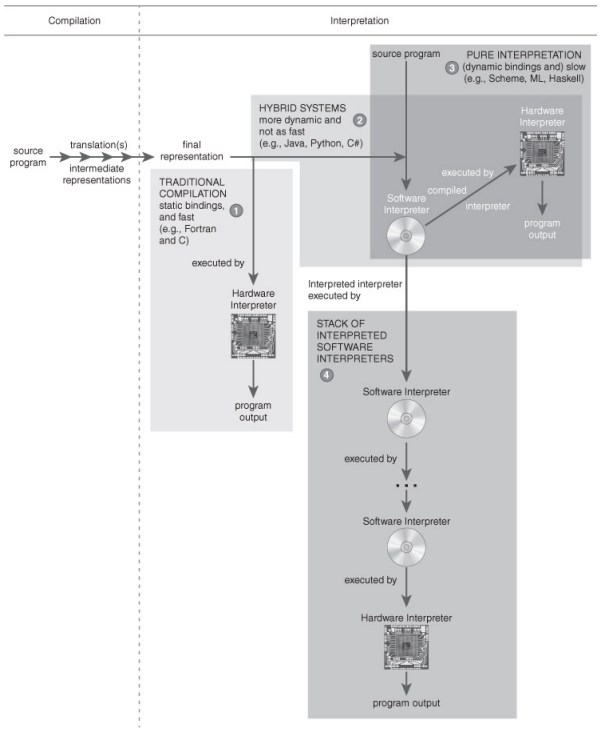
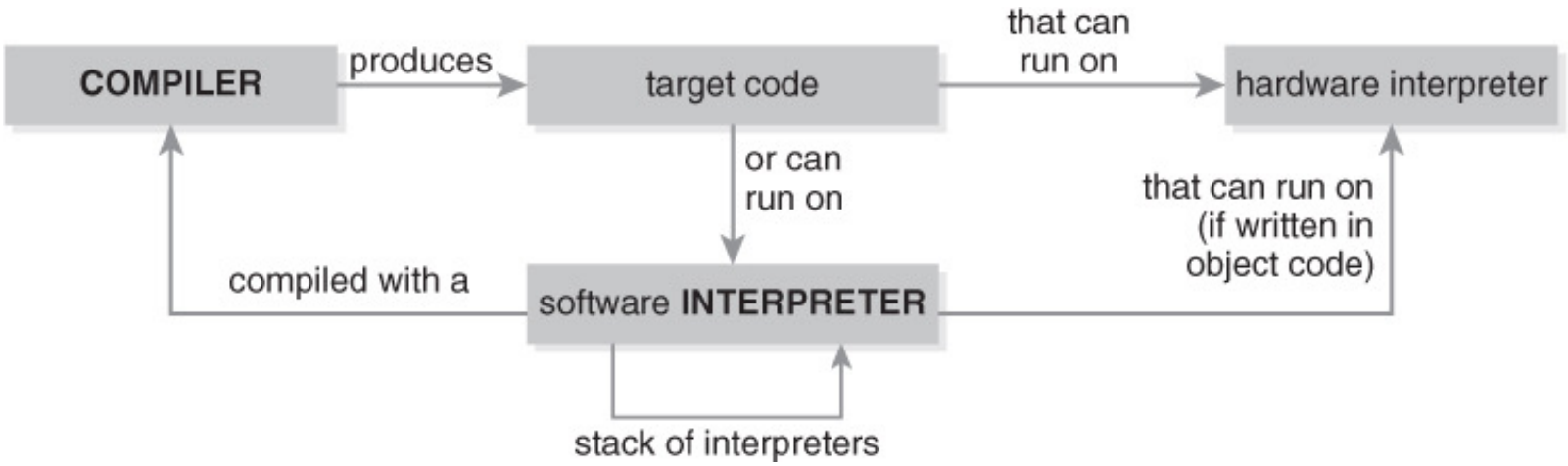


Figure 4.7 Mutually Dependent Relationship Between Compilers and Interpreters



Outline

- 4.1 Chapter Objectives
- 4.2 Interpretation Vis-à-Vis Compilation
- 4.3 Run-Time Systems: Methods of Executions
- **4.4 Comparison of Interpreters and Compilers**
- 4.5 Influence of Language Goals on Implementation
- 4.6 Thematic Takeaways

Table 4.1 Advantages and Disadvantages of Compilers and Interpreters

Implementation	Advantages	Disadvantages
Traditional Compiler	fast execution; compile once, run repeatedly	inconvenient program development; no REPL; less source-level debugging; less run-time flexibility
Pure Interpreter	convenient program development; REPL; direct source-level debugging; run-time flexibility	slow execution (decoding); often requires more run-time space

Outline

- 4.1 Chapter Objectives
- 4.2 Interpretation Vis-à-Vis Compilation
- 4.3 Run-Time Systems: Methods of Executions
- 4.4 Comparison of Interpreters and Compilers
- **4.5 Influence of Language Goals on Implementation**
- 4.6 Thematic Takeaways

4.5 Influence of Language Goals on Implementation (1 of 2)

- The goals of a language (e.g., speed of execution, ease of development, safety) influence its design choices (e.g., static or dynamic bindings).
- For instance, Fortran and C programs are intended to execute fast and, therefore, are compiled.
- The speed of the executable produced by a compiler is a direct result of the efficient decoding of machine instructions (vis-à-vis high-level statements) at run-time coupled with few semantic checks at run-time.
- It is natural to implement a language designed to support static bindings through compilation because establishing those bindings and performing semantic checks for them can occur at compile time so they do not occupy CPU cycles at run-time—yielding a fast executable.
- A compiler for a language supporting static bindings need not generate code for performing semantic checks at run-time in the target executable.

4.5 Influence of Language Goals on Implementation (2 of 2)

- UNIX shell scripts, by contrast, are intended to be quick and easy to develop and debug; thus, they are interpreted.
- It is natural and easier to interpret programs in a language with dynamic bindings (e.g., identifiers that can be bound to values of any type at run-time), including Scheme, since the necessary semantic checks cannot be performed before run-time.
- Compiling programs written in languages with dynamic bindings requires generating code in the target executable for performing semantic checks at run-time.

Outline

- 4.1 Chapter Objectives
- 4.2 Interpretation Vis-à-Vis Compilation
- 4.3 Run-Time Systems: Methods of Executions
- 4.4 Comparison of Interpreters and Compilers
- 4.5 Influence of Language Goals on Implementation
- **4.6 Thematic Takeaways**

4.6 Thematic Takeaways (1 of 2)

- Languages lend themselves to implementation through either interpretation or compilation, but usually not through both.
- An interpreter or compiler for a computer language creates a virtual machine for the language of the source program (i.e., a computer that virtually understands the language).
- Compilers and interpreters are often complementary in terms of their advantages and disadvantages. This leads to the conception of hybrid implementation systems.
- Compilation results in a fast executable; interpretation results in slow execution because it takes longer to decode high-level program statements than machine instructions.

4.6 Thematic Takeaways (2 of 2)

- Interpreters support run-time flexibility in the source language, which is often less practical in compiled languages.
- Trade-offs between speed of execution and speed of development have been factors in the evolution and implementation of programming languages.
- The goals of a language (e.g., speed of execution, speed of development) and its design choices (e.g., static or dynamic bindings) have historically influenced the implementation approach of the language (e.g., interpretation or compilation).