Compilers: Principles, Techniques, & Tools

By Alfred V. Aho et all

First Edition

CONTENTS

Contents **0**

1 Introduction **1**

1 Language Processors . . . 1

2 The Structure of a Compiler . . . 1

2.1 Lexical Analysis . . . 1

2.2 Syntax Analysis . . . 2

2.3 Semantic Analysis . . . 2

2.4 Intermediate Code Generation . . . 2

2.5 The Grouping of Phases into Passes . . . 2

2.6 Compiler-Construction Tools . . . 2

3 The Evolution of Programming Language . . . 2

3.1 The Move to Higher-Level Languages . . . 2

4 Applications of Compiler Technology . . . 3

4.1 implementation of High-Level Programming Languages . . . 3

4.2 Optimizations for Computer Architectures . . . 3

5 Programming Language Basics . . . 3

5.1 The Static/Dynamic Distinction . . . 3

5.2 Environments and States . . . 3

5.3 Static Scope and Block Structure . . . 4

5.4 Explicit Access Control . . . 4

5.5 Dynamic Scope . . . 4

5.6 Parameter Passing Mechanisms . . . 4

5.7 Aliasing . . . 5

2 The Continuous-Time Fourier Transform **7**

CHAPTER 1

INTRODUCTION

The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.

The software systems that do this translation are called *compilers*.

1 Language Processors

Simply stated, a compiler is a program that can read a program in one language–the *source* language–and translate it into an equivalent program in another language–the *target* language.

An *interpreter* is another common kind of language processor.

The task of collecting the source program is sometimes entrusted to a separate program, called a *preprocessor*.

The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an *assembler* that produces relocatable machine code as its output.

The *linker* resolves external memory addresses, where the code in one file may refer to a location in another file. The *loader* then puts together all of the executable object files into memory for execution.

2 The Structure of a Compiler

Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis.

The *analysis* part breaks up the source program into constituent pieces and imposes a grammatical structure on them. The analysis part also collects information about the source program and stores it in a data structure called a *symbol table*, which is passed along with the intermediate representation to the synthesis part.

The *synthesis* part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the *front end* of the compiler; the synthesis part is the *back end*.

If we examine the compilation process in more detail, we see that it operates as a sequence of *phases*, each of which transforms one representation of the source program to another.

2.1 Lexical Analysis

The first phase of a compiler is called *lexical analysis* or *scanning*. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called *lexemes*. For each lexeme, the lexical analyzer produces as output a *token* of the form

$$\langle \textit{token-name, attribute-value} \rangle$$

that is passes on to the subsequent phase, syntax analysis. In the token, the first component *token-name* is an abstract symbol that is used during syntax analysis, and the second component *attribute-value* points to an entry in the symbol table for this token.

2.2 Syntax Analysis

The second phase of the compiler is *syntax analysis* or *parsing*. A typical representation is a *syntax tree* in which each interior node represents an operation and the children of the node represent the arguments of the operation.

2.3 Semantic Analysis

The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.

An important part of semantic analysis is *type checking*, where the compiler checks that each operator has matching operands.

The language specification may permit some type conversions called *coercions*.

2.4 Intermediate Code Generation

We consider an intermediate form called *three-address code*, which consists of a sequence of assembly-like instructions with three operands per instruction.

2.5 The Grouping of Phases into Passes

In an implementation, activities from several phases may be grouped together into a *pass* that reads an input file and writes an output file.

2.6 Compiler-Construction Tools

Some commonly used compiler-construction tools include

1. *Parser generators* that automatically produce syntax analyzers from a grammatical description of a programming language.
2. *Scanner generators* that produce lexical analyzers from a regular-expression description of the tokens of a language.
3. *Syntax-directed translation engines* that produce collections of routines for walking a parse tree and generating intermediate code.
4. *Code-generator generators* that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
5. *Data-flow analysis engines* that facilitate the gathering of information about how values are transmitted from one part of a program to each other part.
6. *Compiler-construction toolkits* that provide an integrated set of routines for construction various phases of a compiler.

3 The Evolution of Programming Language

3.1 The Move to Higher-Level Languages

One classification is by generation. *First-generation languages* are the machine languages, *second-generation* the assembly languages, and *third-generation* the higher-level languages. *Fourth-generation languages* are languages designed for specific applications. The term *fifth-generation language* has been applied to logic- and constraint-based languages.

Another classification of languages uses the term *imperative* for languages in which a program specifies *how* a computation is to be done and *declarative* for languages in which a program specifies *what* computation is to be done.

The term *von Neumann language* is applied to programming languages whose computational model is based on the von Neumann computer architecture.

An *object-oriented language* is one that supports object-oriented programming, a programming style in which a program consists of a collection of objects that interact with one another.

Scripting languages are interpreted languages with high-level operators designed for "gluing together" computations.

4 Applications of Compiler Technology

4.1 implementation of High-Level Programming Languages

A body of compiler optimizations, known as *data-flow optimizations*, has been developed to analyze the flow of data through the program and removes redundancies across these constructs.

Object-oriented programs are different from those written in many other languages, in that they consist of many more, but smaller, procedures (called *methods* in object-oriented terms).

4.2 Optimizations for Computer Architectures

Almost all high-performance systems take advantage of the same two basic techniques: *parallelism* and *memory hierarchies*. Parallelism can be found at several levels: at the *instruction level*, where multiple operations are executed simultaneously and at the *processor level*, where different threads of the same application are run on different processors.

5 Programming Language Basics

5.1 The Static/Dynamic Distinction

If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a *static* policy or that the issue can be decided at *compile time*. On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a *dynamic policy* or to require a decision at *run time*.

The *scope* of a declaration of x is the region of the program in which uses of x refer to this declaration. A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses *dynamic scope*.

5.2 Environments and States

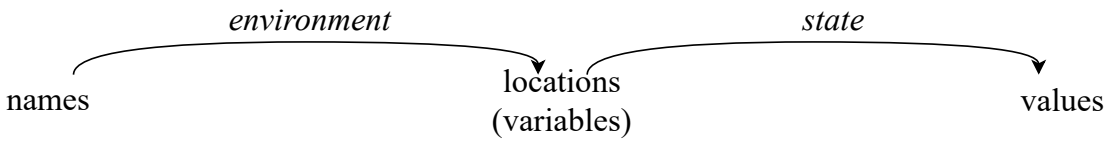

Figure 1.1: Two-stage mapping from names to values

The association of names with locations in memory (the *store*) and then with values can be described by two mappings that change as the program runs:

1. The *environment* is a mapping from names to locations in the store.
2. The *state* is a mapping from locations in store to their values.

The environment and state mappings in Fig. 1.1 are dynamic, but there are a few exceptions:

1. *Static versus dynamic binding* of names to locations.
2. *Static versus dynamic binding* of locations to values.

Names, Identifiers, and Variables

An *identifier* is a string of characters, typically letters or digits, that refers to (identifies) an entity. Composite names are called *qualified* names.

A *variable* refers to a particular location of the store.

5.3 Static Scope and Block Structure

The scope rules for C are based on program structrue; the scope of a declaration is determined implicitly by where the declaration appears in the program. Later languages also provide explicit control over scopes through the use of keywords like **public**, **private** and **protected**.

A *block* is a grouping of declarations and statements. C uses braces `{` and `}` to delimit a block; the alternative use of **begin** and **end** for the same purpose dates back to Algol.

In C, the syntax of blocks is given by

1. One type of statement is a block. Blocks can appear anywhere that other types of statement can appear.

2. A block is a sequence of declarations followed by a sequence of statements, all surrounded by braces.

Note that this syntax allows blocks to be nested inside each other. This nesting property is referred to as *block structure*.

5.4 Explicit Access Control

Through the use of keywords like **public**, **private**, and **protected**, object-oriented languages provide explicit control over access to member names in a superclass. These keywords support *encapsulation* by restricting access.

5.5 Dynamic Scope

Technically, any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes. The term *dynamic scope*, however, usually refers to the following policy: a use of a name x refers to the declaration of x in the most recently called procedure with such a declaration.

Declarations and Definitions

In C++, a method is declared in a class definition, by giving the types of the arguments and result of the method (often called the *signature* for the method).

5.6 Parameter Passing Mechanisms

Actual parameters (the parameters used in the call of a procedure) are associated with the *formal parameters* (those used in the procedure definition).

Call-by-Value

In *call-by-value*, the actual parameter is evaluated (if it is an expression) or copied (if it is a variable).

Call-by-Reference

In *call-by-reference*, the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter.

5.7 Aliasing

It is possible that two formal parameters can refer to the same location; such variables are said to be *aliases* of one another.

CHAPTER 2

THE CONTINUOUS-TIME FOURIER TRANSFORM