Chapter 1 Introduction

The compilers area field is “language processing”, they act as translator of the programming language being compiled (the source) to some machine language (the *target*).

Compilers may be distinguished in two ways:

Machine code

*Pure Machine Code***:** Mainly used in compilers for **system implementation languages** This form of target code can execute on bare hardware without dependence on any other software.

Augmented Machine Code: A compiler requires that a particular operating system be present on the target machine and a collection of language-specific runtime support routines (I/O, storage allocation, mathematical functions, etc.)

Virtual Machine Code: Code generated by the compiler can then be run on any architecture for which a VM interpreter is available. The process is called **bootstrapping**

Target Code

Assembly or other source formats: Is useful for **cross-compilation**, The symbolic assembly code is easily transferred between different computers.

Relocatable binary: Is essentially the form of code that most assemblers generate. External references, local instruction addresses, and data addresses are not yet bound. Instead, addresses are assigned relative either to the beginning of the module or to some symbolically named locations. A linkage step is required The result is an **absolute binary format** that is executable.

Absolute binary: **Absolute binary format** can be directly executed when the compiler is finished. This process is usually faster than the other approaches. However, the ability to interface with other code may be limited.

Java compilers emit **bytecodes** that are subsequently subjected to interpretation or dynamic compilation to native machine code. Java has a **native interface** that is designed to allow Java code to interoperate with code written in other languages. Java also requires **dynamic linking** of classes used by an application.

Interpreters share some of the functionality found in compilers, such as syntactic and semantic analyses. However, interpreters differ from compilers in that they execute programs without explicitly performing much translation.

- Programs can be easily modified as execution proceeds. This provides a straightforward **interactive debugging** capability
- Languages in which the type of an object is developed dynamically (e.g., Lisp and Scheme) are easily supported in an interpreter. Some languages (such as Smalltalk and Ruby) allow the type system itself to change dynamically.
- Interpreters provide as ignificant degree of machine independence, since no machine code is generated. Porting an interpreter can be as simple as recompiling the interpreter on a new machine.

A complete definition of a programming language must include the specification of its **syntax** (structure) and its **semantics** (meaning). Syntax typically means context-free syntax because of the almost universal use of **context-free grammars** (CFGs) as a syntactic specification mechanism. Syntax defines the sequences of symbols that are legal.

Semantics are commonly divided into two classes:

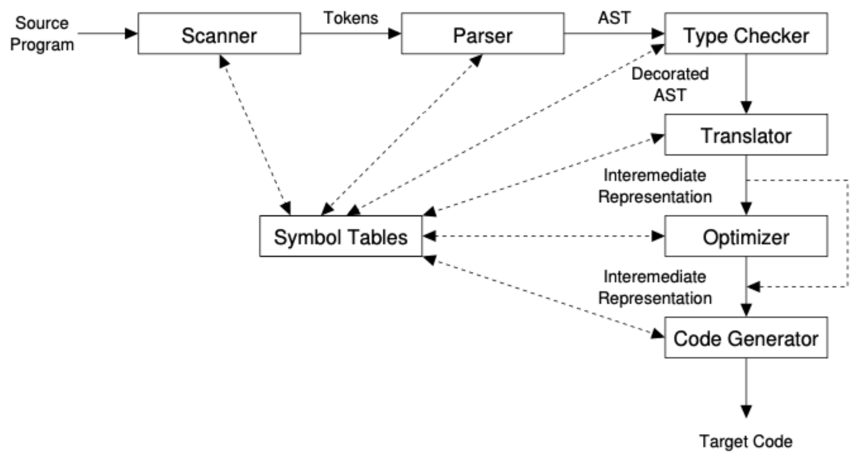
Static semantics: Provides a set of rules that specify which syntactically legal programs are actually valid. Most compiler-writing systems propagate semantic information through a program’s **abstract syntax tree** (AST) in a manner similar to the evaluation of attribute grammar systems.

Runtime semantics: Are often specified as an operational or interpreter model. a program “state” is defined and program execution is described in terms of changes to that state.

<u>Natural semantics</u> Given assertions known to be true before the evaluations of a construct, we can infer assertions that will hold after the construct’s evaluation.

Axiomatic semantics They are based on formally specified relations, or predicates, that relate program variables. Statements are defined by how they modify these relations.

Denotational models may be viewed as a syntax-directed definition that specifies the meaning of a construct in terms of the meaning of its immediate constituents.



The **scanner** begins the analysis of the source program by reading character by character and grouping them into **tokens** such as identifiers, integers, reserved words, and delimiters. The tokens are encoded (often as integers) and fed to the parser for syntactic analysis. **Regular expressions** are a formal notation sufficiently powerful to describe the variety of tokens required by modern programming languages. In addition, they can be used as a specification for the automatic generation of finite that recognize **regular sets,** that is the basis of the scanner generator

The **parser** is based on a formal syntax specification such as a CFGs The parser verifies correct syntax. If a syntax error is found, it issues a suitable error message. In many cases, **syntactic error recovery** or **repair** can be done automatically by consulting structures created by a suitable parser generator.

The type checker checks the **static semantics** of each AST node. That is, it verifies that the construct the node represents is legal and meaningful.

Translator (Program Synthesis) is largely dictated by the semantics of the source language. some general aspects of the target machine may be exploited. More elaborate compilers such as the **GNU Compiler Collection** (GCC) may first generate a high-level IR (that is source-language oriented) and then subsequently translate it into a low-level IR (that is target-machine oriented).

A **symbol table** is a mechanism that allows information to be associated with identifiers and shared among compiler phases.

The optimizer phase helps in a way that can significantly speed program execution by simplifying, moving, or eliminating unneeded computations. An example is **peep- hole optimization**.

The code generator phase requires detailed information about the target machine and includes machine-specific optimization such as register allocation and code scheduling.

Optimizing compilers actually use a wide variety of transformations that improve a program’s performance. The complexity of an optimizing compiler arises from the need to employ a variety of transforms, some of which interfere with each other. For example, keeping frequently used variables in registers reduces their access time but makes procedure and function calls more expensive. The choice of which improvements are most effective (and least expensive) is a matter of judgment and experience.

A retargetable compiler is one whose target architecture can be changed without its machine-independent components having to be rewritten, is more difficult to write than an ordinary compiler because target-machine dependencies must be carefully localized.