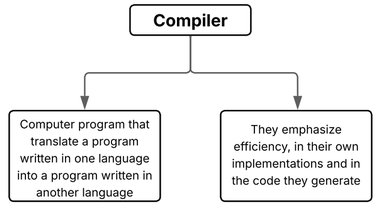


This book presents the fundamental techniques of automatic translation that are used to build compilers. It describes many of the challenges that arise in compiler construction and the algorithms that compiler writers use to address them.

Interpreter vs. Compiler
An interpreter takes as input an executable specification and produces as output the result of executing the specification whereas a compiler takes an executable
specification and produces another executable specification

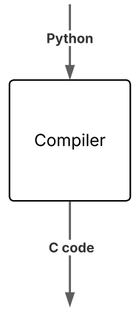
A good compiler contains a microcosm of computer science. It makes practical use of greedy algorithms (register allocation), heuristic search techniques (list scheduling), graph algorithms (dead-code elimination), dynamic programming (instruction selection), finite automata and push-down automata (scanning and parsing), and fixed-point algorithms (data-flow analysis). It deals with problems such as dynamic allocation, synchronization, naming, locality, memory hierarchy management, and pipeline scheduling.

The Fundamental Principles of Compilation

1. The compiler must preserve the meaning of the program being compiled
2. The compiler must improve the input in some discernible way

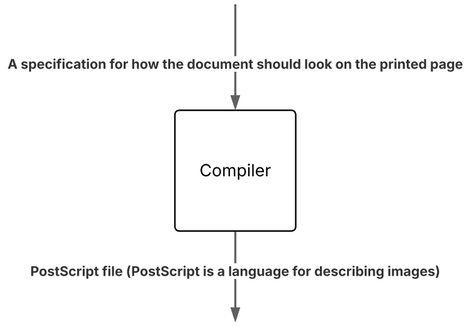
Example 1 of compiler

There exist compilers from source code to source code

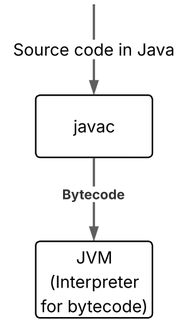


Example 2 of compiler

A typesetting program that produces PostScript



Example 3: The Java compiler method (an interesting case)



Virtual machine or VM
Simulator of some processor. It is an interpreter for the set of instructions of that machine

Many implementations of the JMV include a compiler that executes at runtime, sometimes called a *just-in-time compiler*, or *jit*, that translates heavily used bytecode sequences into native code for the underlying computer

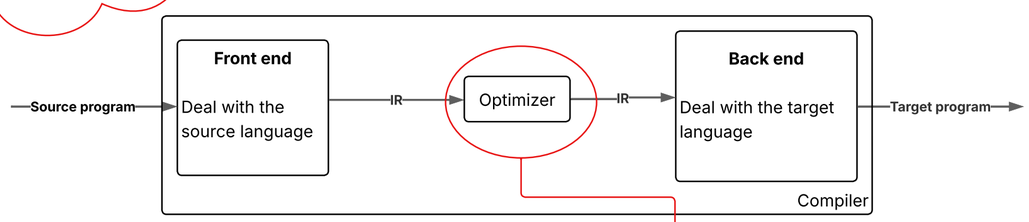
Retargeting the compiler
The task of changing the compiler to generate code for a new processor

We can easily envision constructing multiple back ends for a single front end to produce compilers that accept the same language but target different machines

It has a formal structure representing the program in an intermediate form (intermediate representation or IR)

It also includes an optimizer that analyzes and rewrites that intermediate form

General Structure of a compiler



A three-phase compiler

A compiler may, in fact, use several different IRs as compilation progresses, but at each point, one representation will be the definitive IR. This is important because the IR could remember information of the translation at one phase to improve that translation in later phases

The optimizer can make one or more passes over the ir, analyze the ir, and rewrite the ir

A good optimizing compiler can improve the quality of the code, relative to an unoptimized version. However, an optimizing compiler will almost always fail to produce optimal code

- The **front end** consists of two or three passes that handle the details of recognizing valid source-language programs and producing the initial IR form of the program.
- The middle section contains passes that perform different **optimizations.** The number and purpose of these passes vary from compiler to compiler.
- The **back end** consists of a series of passes, each of which takes the ir program one step closer to the target machine's instruction set

There are two possible structures for the phase of optimization

Monolothic structure

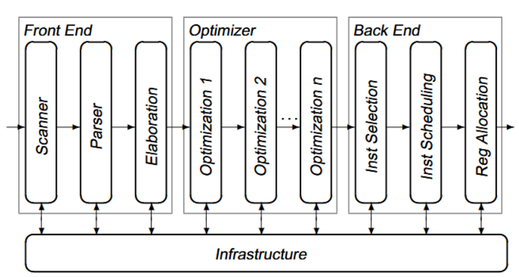
- All optimizations are applied at only one step (or module)
- It may be more efficient, but more difficult of implementing and debugging

Multipass structure

- It may lend itself to a less complex implementation and a simpler approach to debugging the compiler
- It also creates the flexibility to employ different sets of optimization in different situations

Remember
- Syntax = Form
- Semantics = Meaning

More specific structure of a typical compiler



■ **FIGURE 1.1** Structure of a Typical Compiler.