بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین

ایمیل: *h.falsafain@iut.ac.ir* یا *h.falsafain@gmail.com*
آدرس دفتر: اتاق ۳۲۳ دانشکده
شمارهٔ تلفن دفتر: ۳۳۹۱۹۰۶۸ـ۰۳۱
آدرس وبسایت شخصی: *https://falsafain.iut.ac.ir/*

تمامي امور مربوط به درس (اعم از امور مربوط به تکالیف و کوئیزها) از طریق سامانهٔ یکتا صورت می‌پذیرند: *https://yekta.iut.ac.ir/*

لطفاً فقط از طریق ایمیل درخواست‌ها، پرسش‌ها، و نظرات خود را مطرح کنید.
لطفاً پیامی روی اسکایپ، یکتا، تلگرام، و غیره ارسال نکنید.

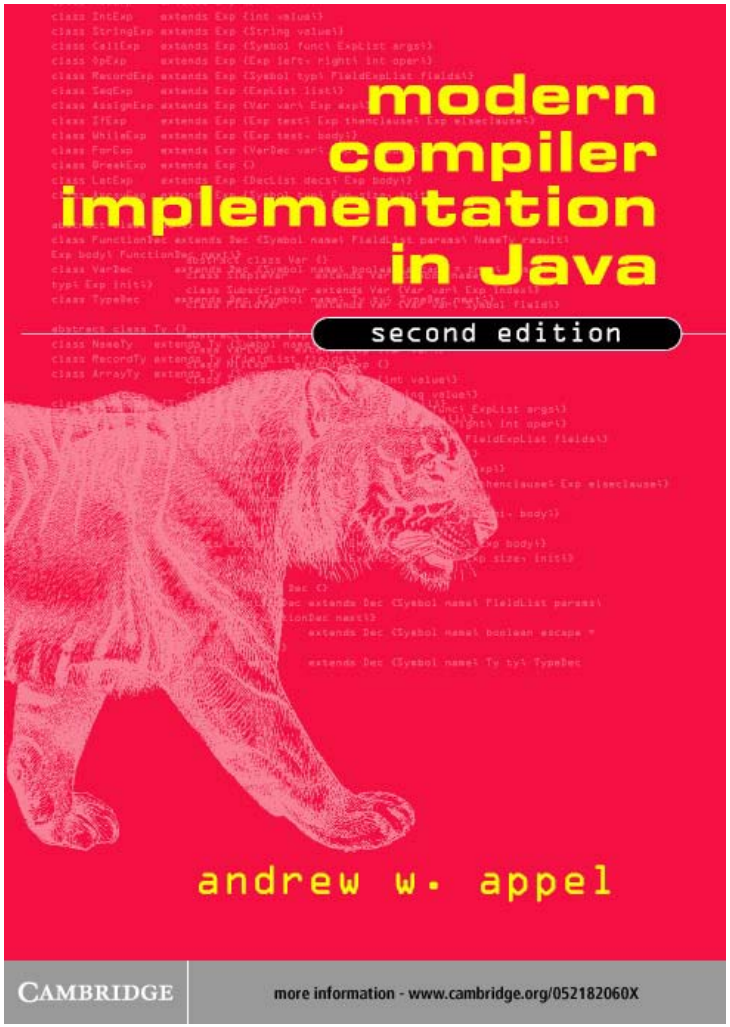
h.falsafain@iut.ac.ir & h.falsafain@gmail.com

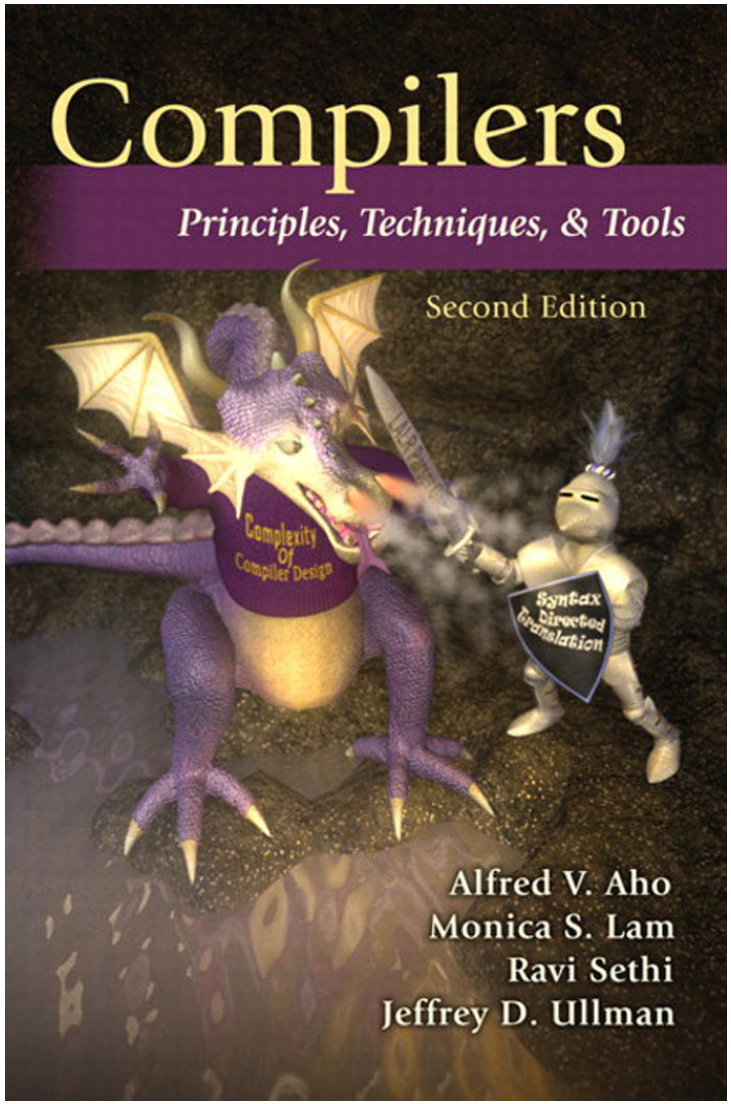
ارزشیابی و کلاس حل تمرین

* تکالیف و کوئیزها: تقریباً ۵ نمره
* آزمون میانترم: تقریباً ۷.۵ نمره
* آزمون پایانترم: تقریباً ۷.۵ نمره
* حضور در کلاس و مشارکت در بحث‌های کلاسی نمره‌ای افزون بر ۲۰ نمره فوق خواهد داشت.

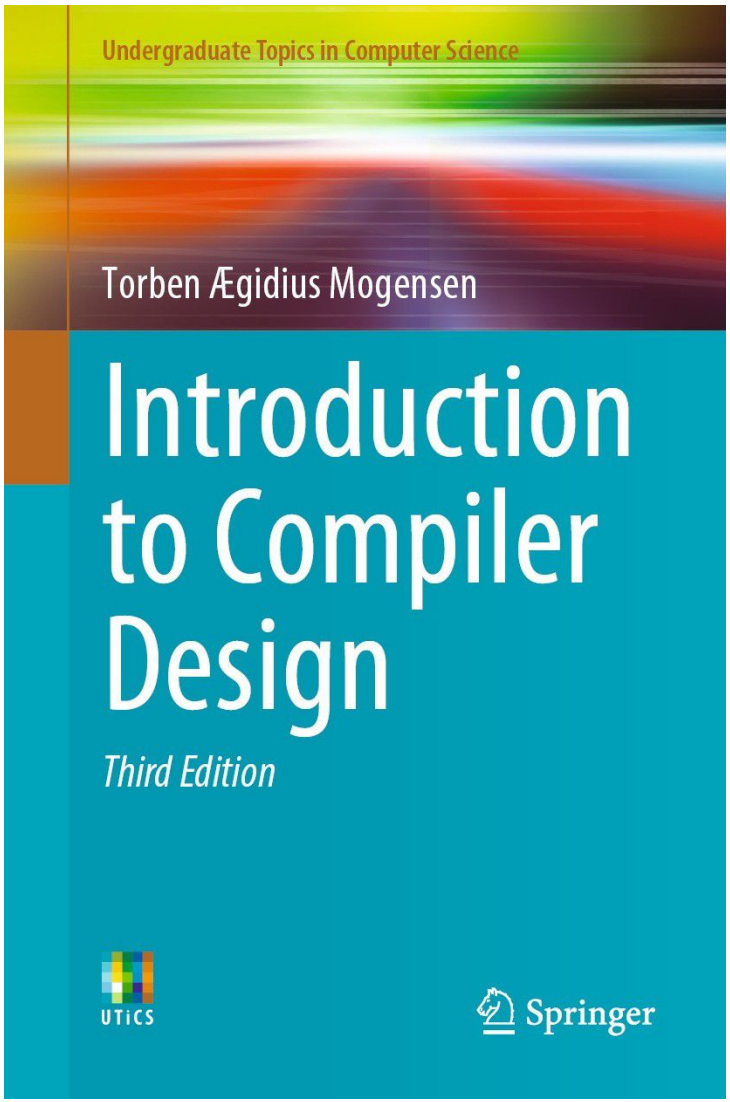
زمان تشکیل کلاس حل تمرین، بر اساس نظرسنجی معیّن خواهد شد.

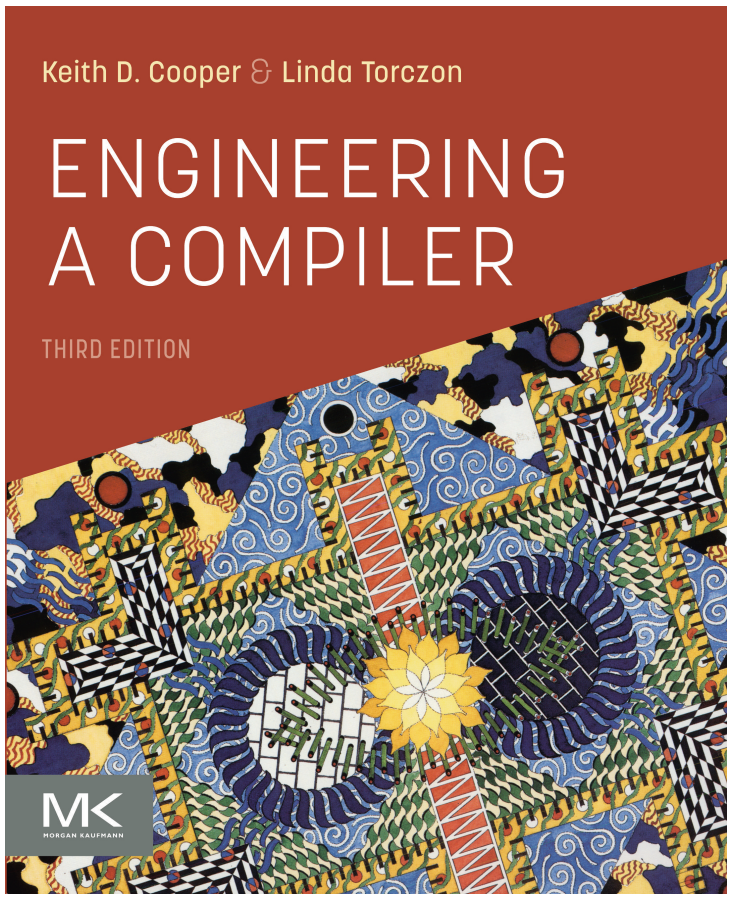
دو مرجع کلاسیک و به‌نسبت قدیمی‌تر (کتاب اژدها و کتاب ببر)





دو مرجع به‌نسبت جدیدتر





http://6.035.scripts.mit.edu/fa18/

6.035: Computer Language Engineering Fall 2018

Home

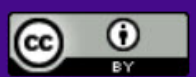
Overview

General Administrivia

Schedule

Reference Materials

CyberPortal 1.1



News

- **November 5:** Lectures will restart on Thursday November 8. There is no lecture this Wednesday.
- **September 26:** Teams repositories have been created! You should have received an invitation to your repo. Links to all the repositories are also on piazza. If you don't have a team yet or didn't receive an invite please email the TAs asap!
- **September 5:** Please fill out this form so that we could start adding github usernames to the course repository. Also, sign up for the piazza page here.
- **September 5:** First Day of Class! You could find most syllabus related information in the General Administrivia page. Note that the Projects Overview is available on the calendar for today, to help you get a sense of deadlines and a high level understanding of the project. Also note that there are more handouts posted on the calendar for next Monday, but you don't need to worry about it until later this week.
- **September 3:** Welcome to 6.035! Lectures are in **3-370 from 11 am to 12 pm MWF**,and **4-149 from 11 am to 12 pm TR**. Note that there are no lectures on some days, please check the schedule.

http://web.stanford.edu/class/cs143/

Welcome to CS143! Assignments and handouts will be available here. Discussion will happen through Ed Discussion on **Canvas**. Written assignments will be handed in through **Gradescope**.

Lectures are held Tuesday and Thursday mornings at 10:30-11:50 in Gates B1.

More Information

- **Schedule/Syllabus**
- **Course Information**
- **Course Policies**
- **Canvas**
- **Ed Discussion**
- **Gradescope (WAs)**
- Stanford myth: ssh to myth.stanford.edu

Handouts

- **Final 2022 (Solutions)**
- **Final 2021 (Solutions)**
- **Midterm 2023 (Solutions)**

Resources

- **Cool Reference Manual**
- **Tour of the Cool Support Code**
- **Flex Manual**
- **Bison Manual**
- **Cool Runtime**
- **SPIM Manual**

Lectures

1. **Course Overview**
2. **Cool: The Course Project**
3. **Lexical Analysis**
4. **Implementation of Lexical Analysis**
5. **Introduction to Parsing**
6. **Syntax-Directed Translation**
7. **Top-Down Parsing & Bottom-Up Parsing I**
8. **Bottom-Up Parsing II**
9. **Semantic Analysis & Type Checking I**
10. **Type Checking II**
11. **Run-time Environments**

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 important role of the compiler is to report any errors in the source program that it detects during the translation process.

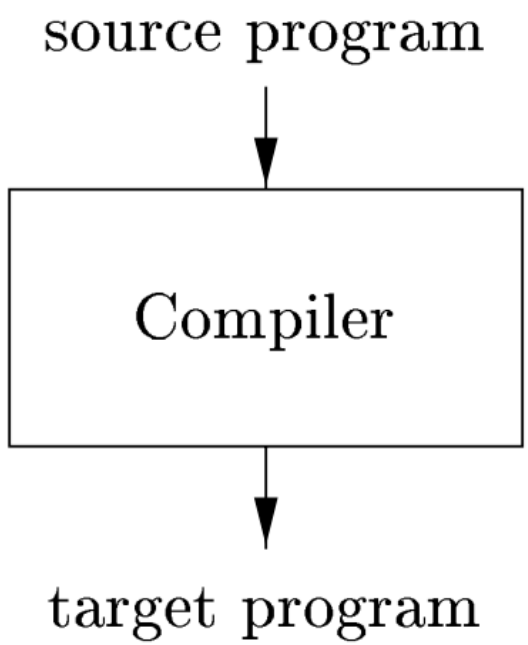


Figure 1.1: A compiler

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

Figure 1.2: Running the target program

https://en.wikipedia.org/wiki/Executable
https://en.wikipedia.org/wiki/.exe

Compilation can be slow because it is not simple to translate from a high-level source language to low-level languages. But once this translation is done, it can result in fast code. Traditionally, the target language of compilers is machine code, which the computer's processor knows how to execute.

Interpretation

An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

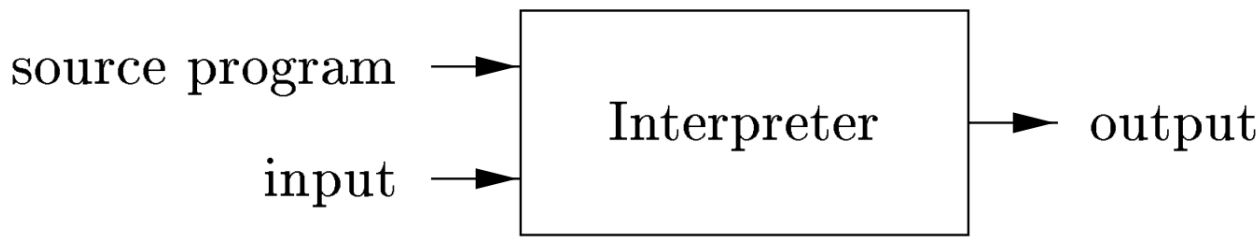


Figure 1.3: An interpreter

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs . An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

مزایا و معایب تفسیر

☞ *One advantage that a software interpreter offers over a compiler is that, given a program, it can quickly start running it without spending time to compile it. A second advantage is that the code is more portable to different hardware architectures; it can run on any hardware architecture that the interpreter itself can run on.*

☞ *The disadvantage of software interpretation is that it is orders of magnitude slower than hardware execution of the same computation. This is because for each machine operation (say, adding two numbers), a software interpreter has to do many operations to figure out what it is supposed to be doing. Adding two numbers can be done in a single machine-code instruction requiring just one machine cycle.*

Hybrids

Java language processors combine compilation and interpretation. A Java source program may first be compiled into an intermediate form called bytecodes. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.

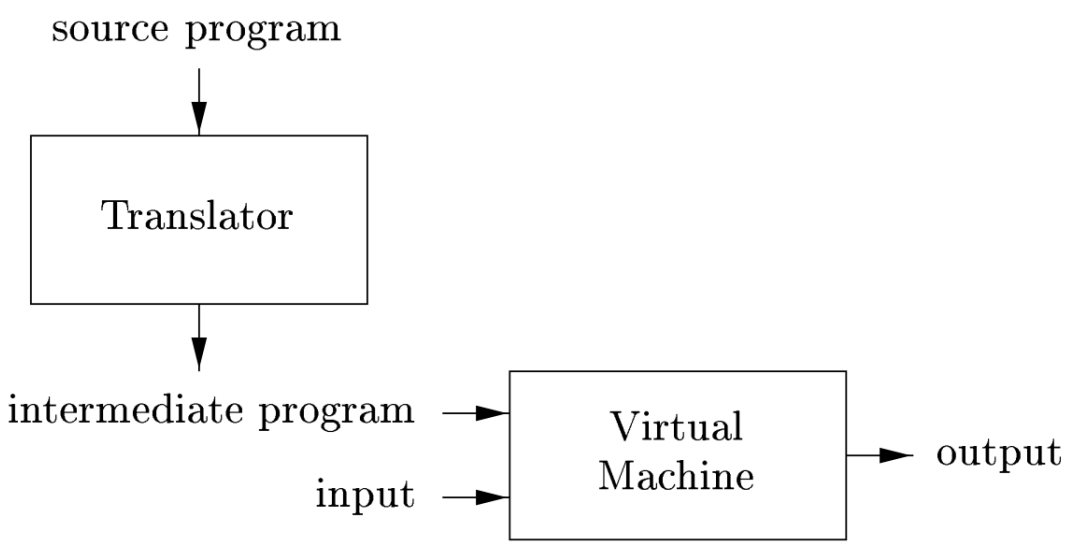
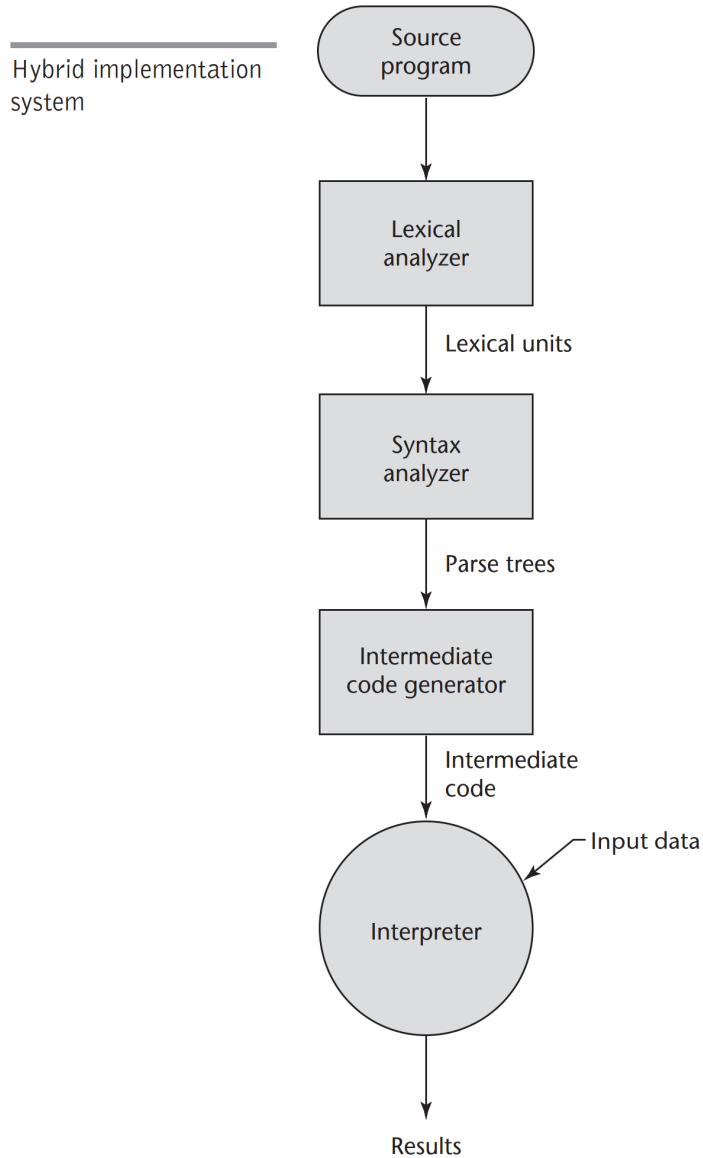


Figure 1.4: A hybrid compiler

دیاگرام مفصل‌تری برای یک هیبرید



Virtual Machines

An abstract machine is a device, which may be implemented in software or in hardware, for executing programs in an intermediate instruction-oriented language. The intermediate language is often called bytecode, because the instruction codes are short and simple compared to the instruction set of "real" machines such as the x86, PowerPC or ARM architectures. Abstract machines are also known as virtual machines.

Prime examples: the Java Virtual Machine, and Microsoft's Common Language Infrastructure.

مزیت بهره‌گیری از یک ماشین مجازی

The purpose of an abstract machine typically is to increase the portability and safety of programs in the source language, such as Java. By compiling Java to a single bytecode language (the JVM), one needs only a single Java compiler, yet the Java programs can be run with no changes on different "real" machine architectures and operating systems. Traditionally it is cumbersome to develop portable software in C, say, because an `int` *value in C may have 16, 32, 36 or 64 bits depending on which machine the program was compiled for.*

یک هیبرید بسیار معروف و حائز اهمیت: جاوا

The Java Virtual Machine (JVM) is an abstract machine and a set of standard libraries developed by Sun Microsystems since 1994. Java programs are compiled to JVM bytecode to make Java programs portable across platforms. There are Java Virtual Machine implementations for a wide range of platforms, from large high-speed servers and desktop computers to very compact embedded systems.

The JVM knows nothing of the Java programming language, only of a particular binary format, the `class` *file format. A* `class` *file contains JVM instructions (or bytecodes) and a symbol table, as well as other ancillary information.*

The JVM was planned as an intermediate target language only for Java, but several other languages now target the JVM, for instance the dynamically typed Groovy, JRuby (a variant of Ruby), Jython (a variant of Python), Clojure, and the statically typed object/functional language Scala.

Just-in-Time (JIT) implementation system

Initial implementations of Java were all hybrid. Its intermediate form, called bytecode, provides portability to any machine that has a bytecode interpreter and an associated run-time system. Together, these are called the Java Virtual Machine. There are now systems that translate Java bytecode into machine code for faster execution. A Just-in-Time (JIT) implementation system initially translates programs to an intermediate language. Then, during execution, it compiles intermediate language methods into machine code when they are called. The machine code version is kept for subsequent calls. JIT systems now are widely used for Java programs.

Also, the .NET languages are all implemented with a JIT system.

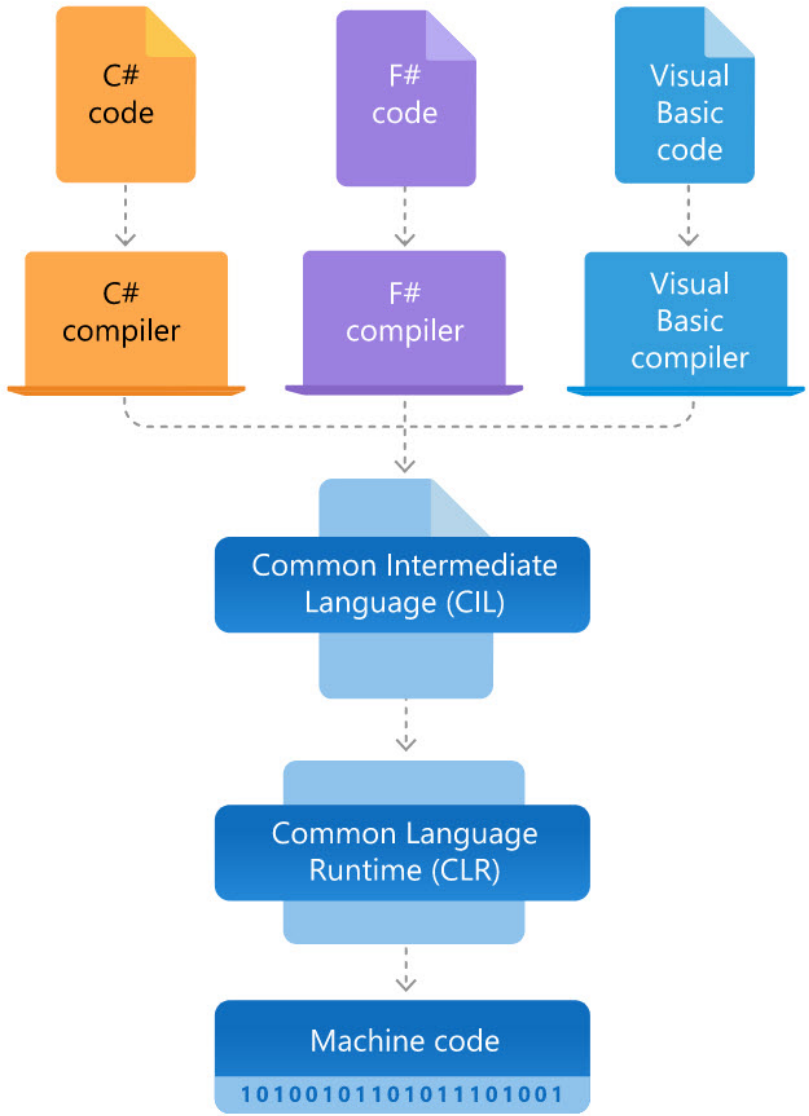
☞ *.NET applications are written in the C#, F#, or Visual Basic programming language. Code is compiled into a language-agnostic Common Intermediate Language (CIL). Compiled code is stored in assemblies—files with a .dll or .exe file extension.*

☞ *When an app runs, the CLR takes the assembly and uses a just-in-time compiler (JIT) to turn it into machine code that can execute on the specific architecture of the computer it is running on.*

☞ *The Common Language Runtime (CLR) runs .NET applications on a given machine, converting the CIL to machine code.*

The Common Language Infrastructure is an abstract machine and a set of standard libraries developed by Microsoft since 1999, with very much the same goals as Sun's JVM. The platform has been standardized by Ecma International and ISO. Microsoft's implementation of CLI is known as the Common Language Runtime (CLR) and is part of .NET, a large set of languages, tools, libraries and technologies.

While the JVM has been implemented on a large number of platforms (Solaris, Linux, MS Windows, web browsers, mobile phones, personal digital assistants) from the beginning, CLI was primarily intended for modern versions of the Microsoft Windows operating system.


C#
code
F#
code
Visual
Basic
code
C#
compiler
F#
compiler
Visual
Basic
compiler
Common Intermediate
Language (CIL)
Common Language
Runtime (CLR)
Machine code
10100101101011101001

کامپایلر حلقه‌ای از یک زنجیره است

In addition to a compiler, several other programs may be required to create an executable target program, as shown in Fig. 1.5.

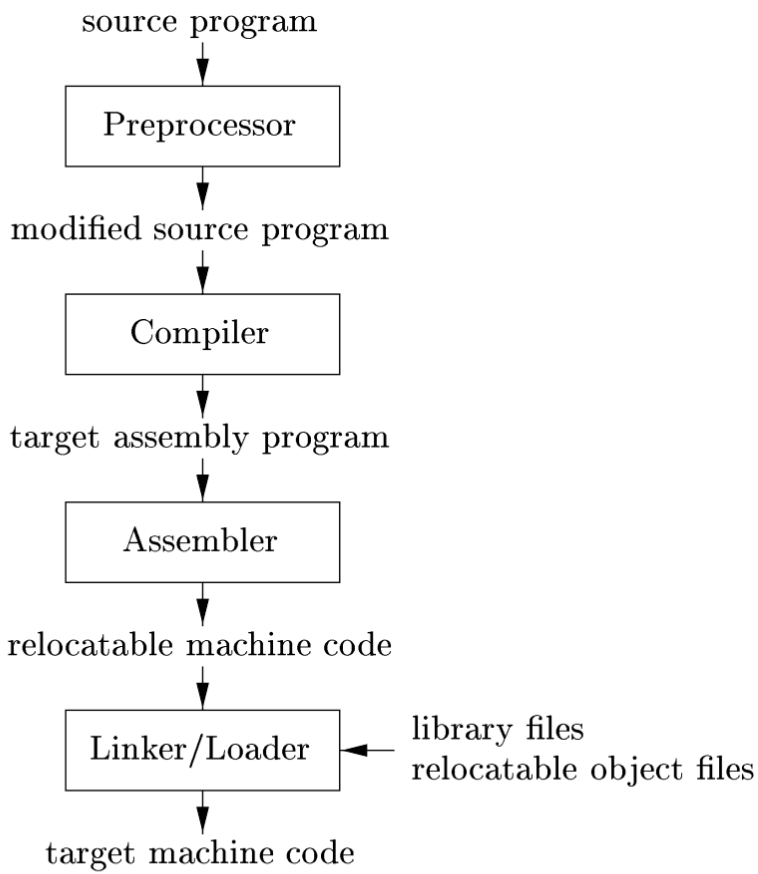


Figure 1.5: A language-processing system

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. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.*

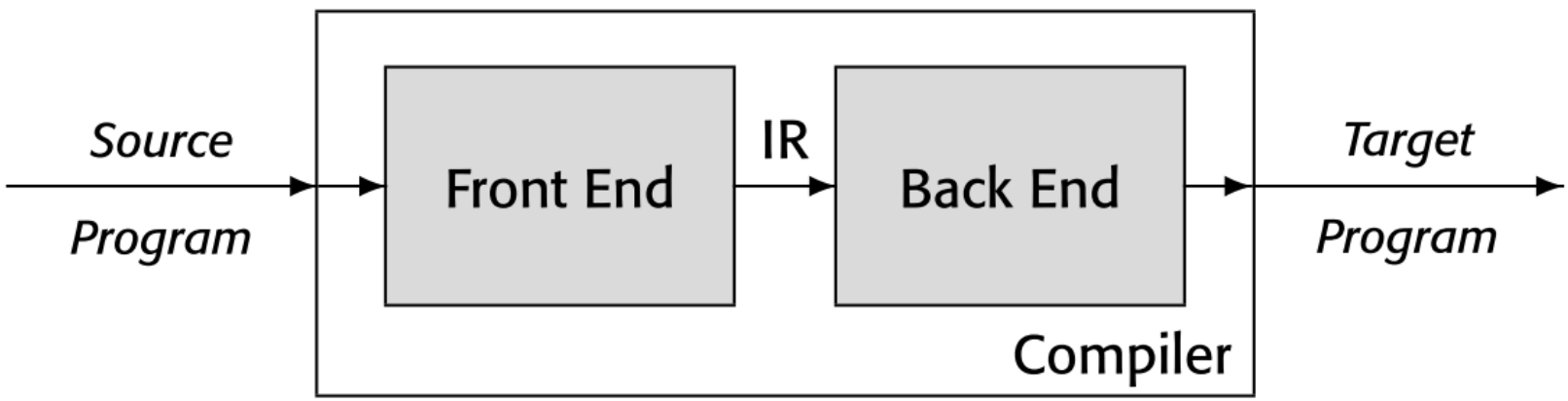
☞ *The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.*

☞ *The symbol table, which stores information about the entire source program, is used by all phases of the compiler.*

☞ *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.*

The front end focuses on understanding the source-language program. The back end focuses on mapping programs to the target machine. This separation of concerns has several important implications for the design and implementation of compilers.



Intermediate Representation (IR)

The front end must encode its knowledge of the source program in some structure, an intermediate representation (IR), for later use. The IR becomes the compiler's definitive representation for the code it is compiling. At each point in the process, the compiler will have a definitive representation. It may, in fact, use several different IRs for different purposes, but, at each point, one representation will be the definitive IR. We think of the definitive IR as the version of the program passed between independent phases of the compiler, like the IR passed from the front end to the back end in the preceding drawing.

`https://gcc.gnu.org/wiki/GIMPLE`

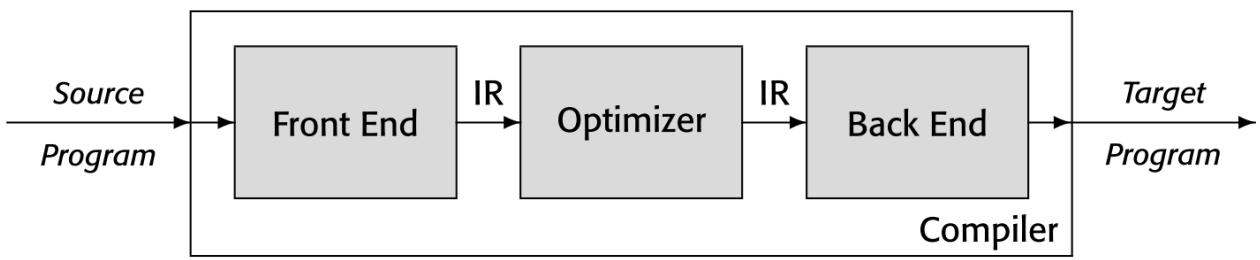
In a two-phase compiler, the front end must ensure that the source program is well formed, and it must map that code into the IR. The back end must map the IR program into the instruction set and the finite resources of the target machine. Because the back end only processes an IR created by the front end, it can assume that the IR contains no syntactic or semantic errors.

Optimizer

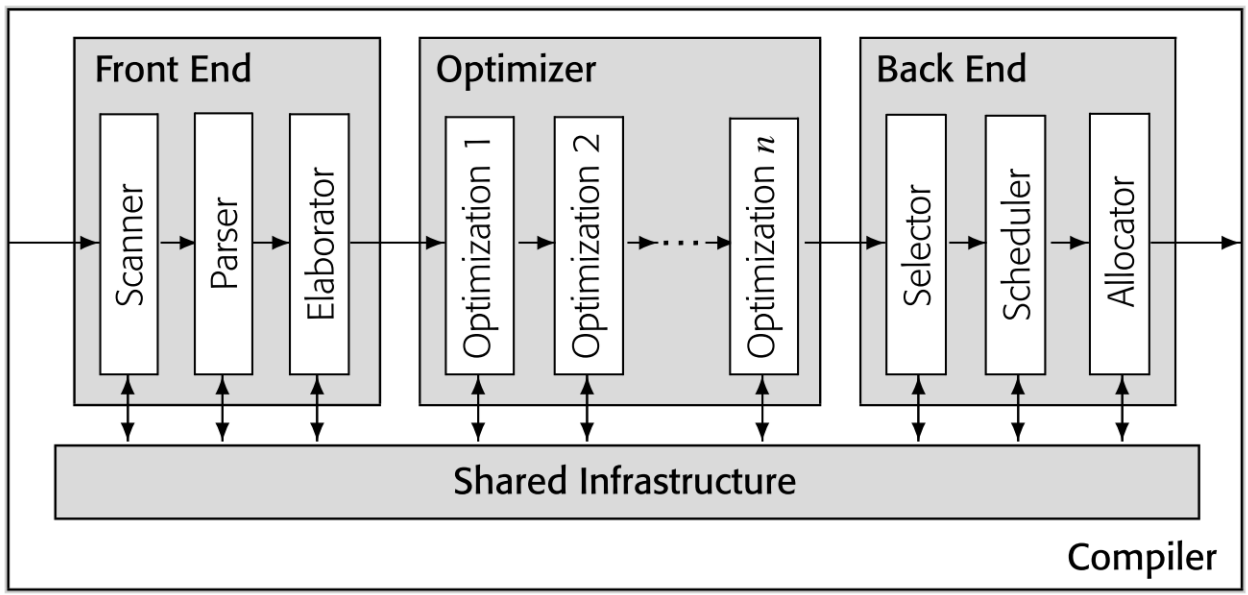
Introducing an IR lets the compiler writer add a phase between the front end and the back end. This middle phase, or optimizer, tries to improve the IR program. With the IR as an interface, the compiler writer can insert optimizations with minimal disruption to the front end and back end. The result is a three-phase compiler.

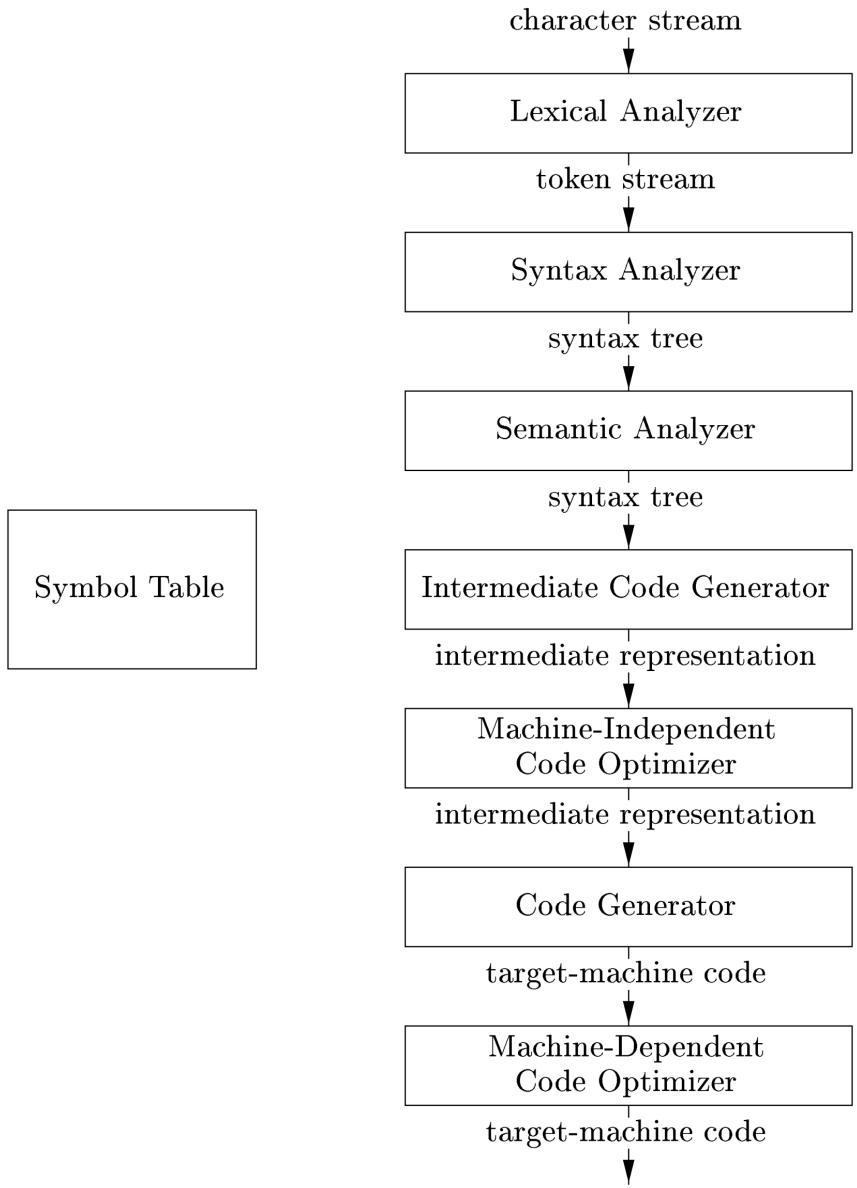
Optimizer: The middle section of a compiler, called an optimizer, analyzes and transforms the IR in an attempt to improve it.

The optimizer is an IR-to-IR transformer, or a series of them, that tries to improve the IR program in some way. The optimizer can make one or more passes over the IR, analyze the IR, and rewrite the IR. The optimizer may rewrite the IR in a way that is likely to produce a faster target program from the back end or a smaller target program from the back end. It may have other objectives, such as a program that produces fewer page faults or uses less energy. Conceptually, the three-phase structure represents the classic optimizing compiler. In practice, each phase is divided internally into a series of passes.

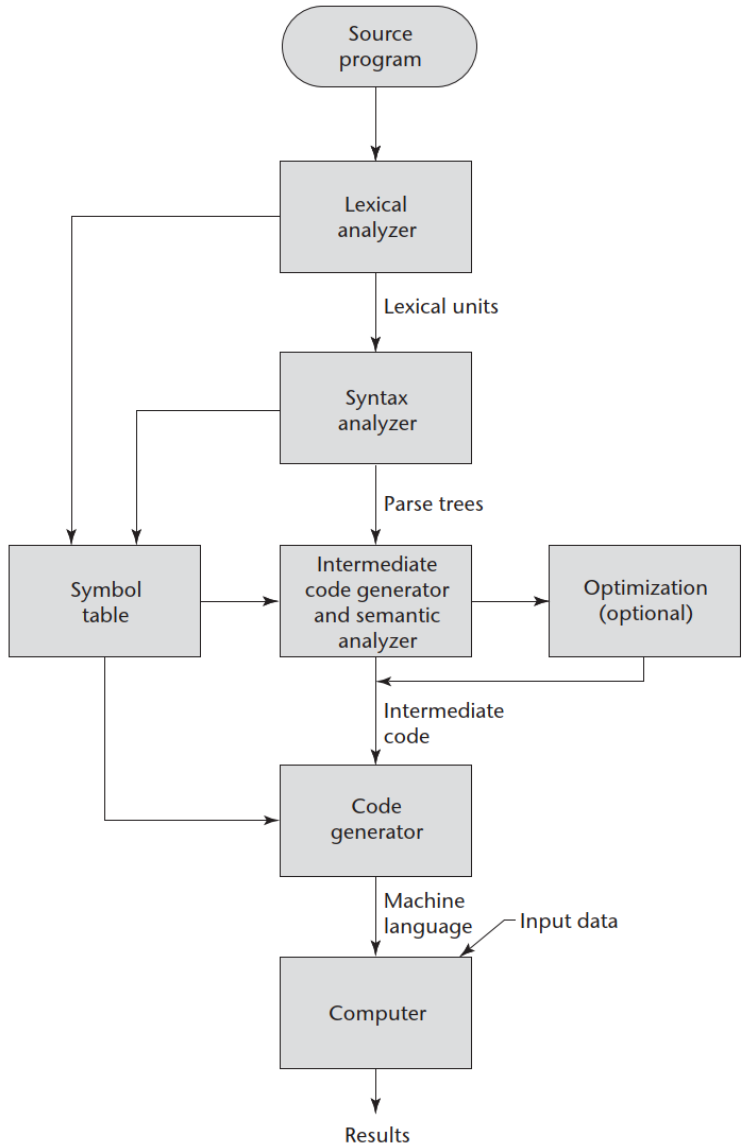


The front end has two or three distinct passes that combine to recognize valid source-language programs and produce the initial IR form of the program. The optimizer contains passes that use distinct analyses and transformations to improve the code. The number and purpose of these passes vary from compiler 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.




character stream
Lexical Analyzer
token stream
Syntax Analyzer
syntax tree
Semantic Analyzer
syntax tree
Symbol Table
Intermediate Code Generator
intermediate representation
Machine-Independent
Code Optimizer
intermediate representation
Code Generator
target-machine code
Machine-Dependent
Code Optimizer
target-machine code

یک دیاگرام دیگر



The Front End

Before the compiler can translate an expression into executable target-machine code, it must understand both its form, or syntax, and its meaning, or semantics. The front end determines if the input code is well formed, in terms of both syntax and semantics. If it finds that the code is valid, it creates a representation of the code in the compiler's IR; if not, it reports back to the user with diagnostic error messages to identify the problems with the code.

To translate a program from one language into another, a compiler must first pull it apart and understand its structure and meaning, then put it together in a different way. The front end of the compiler performs analysis; the back end does synthesis.

The analysis is usually broken up into

☞ *Lexical analysis: breaking the input into individual words or "tokens";*
☞ *Syntax analysis: parsing the phrase structure of the program; and*
☞ *Semantic analysis: calculating the program's meaning.*