Textbook:

Crafting a Compiler
by Fischer, Cytron, and LeBlanc

Pearson 2010

ISBN: 978-0-13-801785-9

Chapter 1

Introduction

Introduction

A **compiler** is a program that

accepts, as input, a program text in a certain **programming language** (source language),

and produces, as output, a program text in an **assembly language** (target language), which will later be assembled by the assembler into machine language.

This process is called **compilation**. (See Fig. 1.1).

Introduction (Cont.)

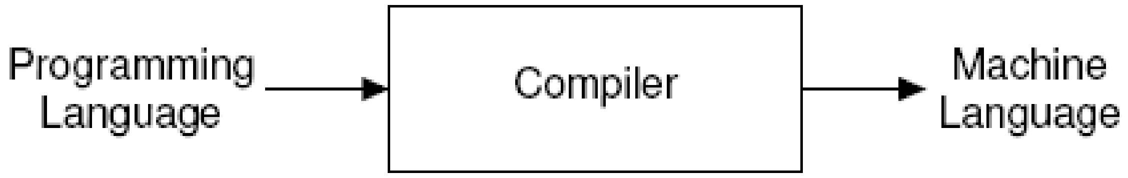


Figure 1.1: A user's view of a compiler.

Front-end and Back-end

- **Front-end** performs the analysis of the **source** language code to build the internal representation, called

  intermediate representation (IR).

- **Back-end** performs the target language synthesis to generate the **target code** (assembly code).

Machine Code Generated by Compilers

Pure machine code

- Compiler may generate code for **a particular machine's instruction set** without assuming the existence of any operating system or library routines.

- Pure machine code is used in compilers for **system implementation languages**, which are for implementing operating systems or embedded applications.

Machine Code Generated by Compilers (Cont.)

Augmented machine code

- Compilers generate code for a machine architecture that is **augmented** with:

  1) operating system routines (aka system calls, window APIs) and

  2) runtime language support routines.

Machine Code Generated by Compilers (Cont.)

Virtual machine code

- Compilers generate virtual machine code that is composed entirely of **virtual machine instructions**.
- That code can run on any architecture for which a VM interpreter is available. For example, the VM for Java, Java virtual machine (JVM), has a JVM interpreter.
  - Virtual machine and a universal intermediate language is not a new idea. But it becomes acceptable now when CPUs are fast enough.
- **Portability** is achieved by writing just one **virtual machine (VM)** interpreter for all the target architectures.
  - In different platform you will need a different VM interpreter

Bootstrapping

When the source language (e.g., L) is also the implementation language (e.g., L) and the source text to be compiled is actually a new compiler of the compiler itself,

Ex: 你有一個 C compiler X 在 x86 平台，此Compiler X 是以 C寫成的source code *P*，你現在在 ARM 沒有C compiler，所以你就將此C compiler 的 code generation 改成 ARM assembly code (or machine code)，然後將 source code P 當作輸入餵給自己 (X)

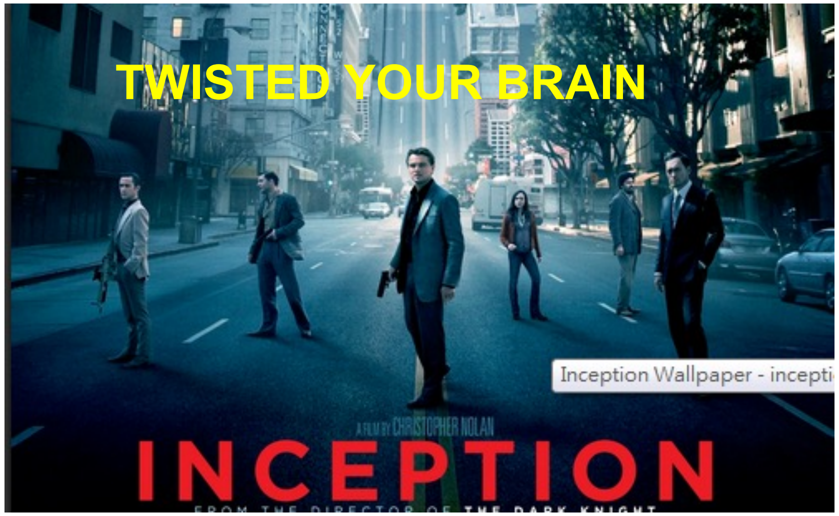
the process is called **bootstrapping**

(see Fig. 1.2)

Chicken & Egg problem

"No, *you* back off! I was here before you!"

Many compilers for many programming languages are bootstrapped


TWISTED YOUR BRAIN
Inception Wallpaper - incepti
A FILM BY CHRISTOPHER NOLAN
INCEPTION
FROM THE DIRECTOR OF THE DARK KNIGHT

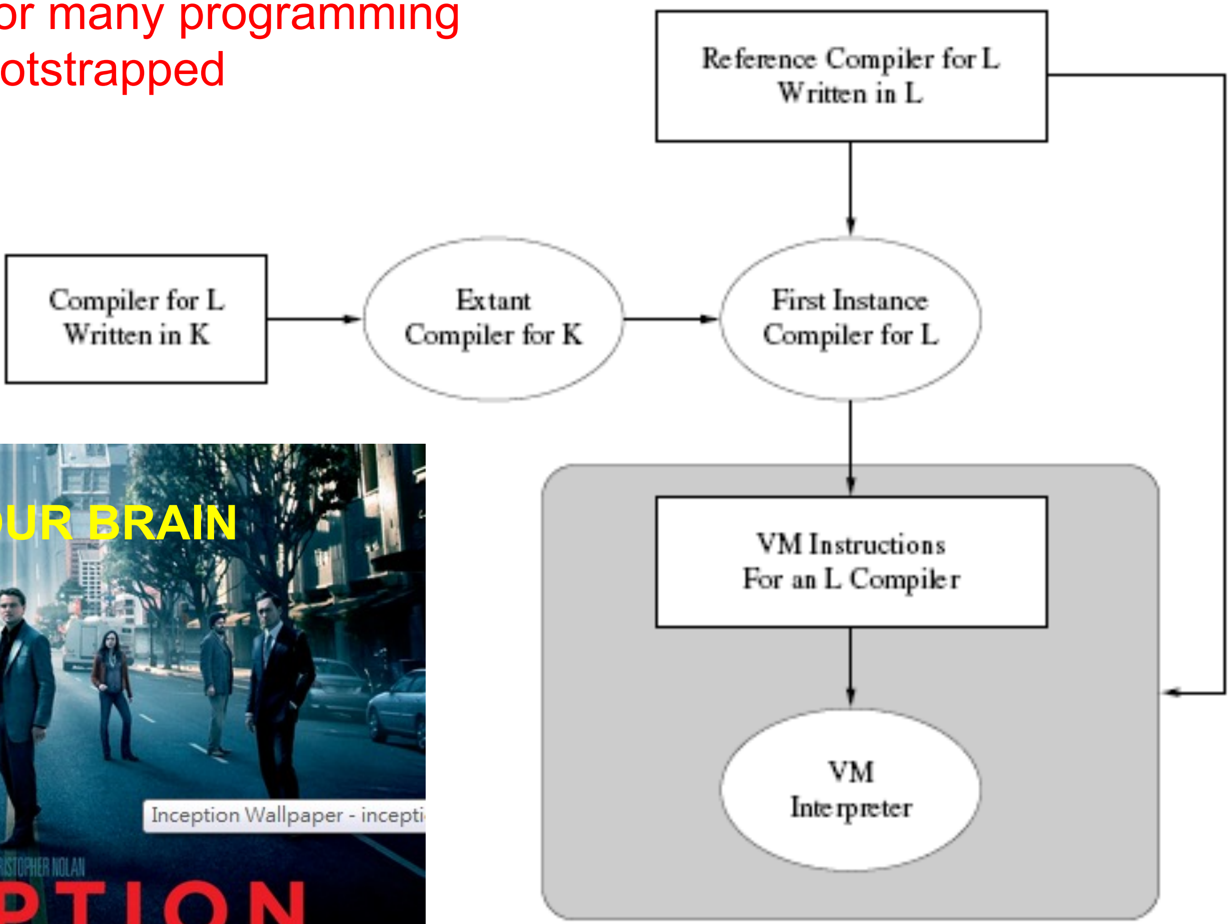

Reference Compiler for L Written in L
Compiler for L Written in K
Extant Compiler for K
First Instance Compiler for L
VM Instructions For an L Compiler
VM Interpreter

Figure 1.2: Bootstrapping a compiler that generates VM instructions. The shaded portion is a portable compiler for L that can run on any architecture supporting the VM.

Compiler vs. Interpreter

Compiler has compilation and execution phases:

1) The compilation phase generates target program from source program.
2) The execution phase executes the target program.

Interpreter directly interprets (executes) the source program that reads inputs and writes outputs (Fig. 1.3).

Compiler versus Interpreter (Cont.)

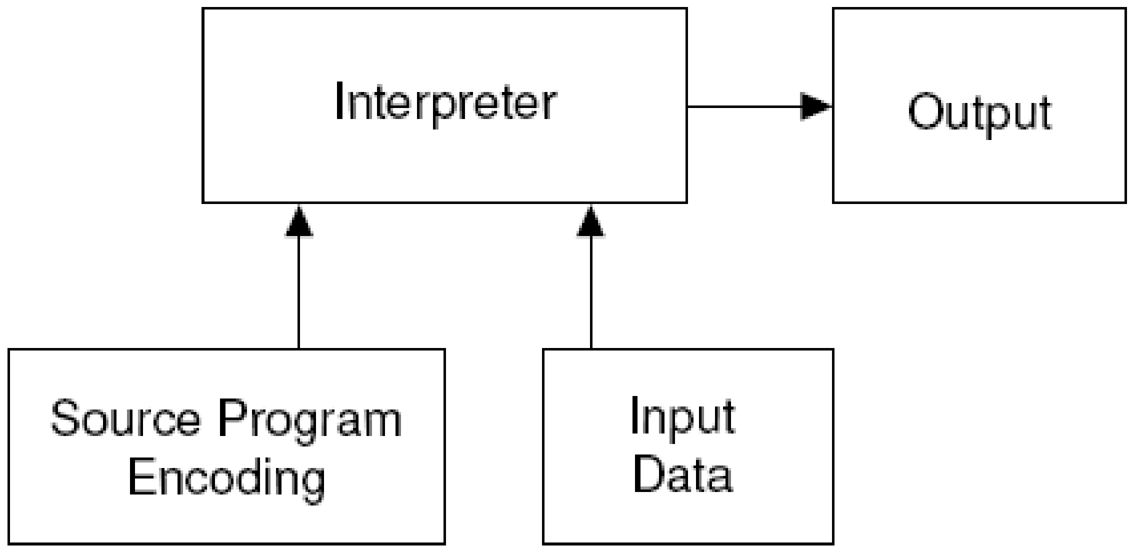


Figure 1.3: An interpreter.

Interpreter

- An interpreter generally uses one of the following strategies for program execution:
  1. execute the source code directly (LISP, BASIC)
  2. translate source code into some efficient intermediate representation and immediately execute this (Perl, Python, MATLAB, and Ruby)
  3. explicitly execute stored precompiled code[1] made by a compiler which is part of the interpreter system (UCSD Pascal)
- Small talk, contemporary basic, Java combines 2 and 3

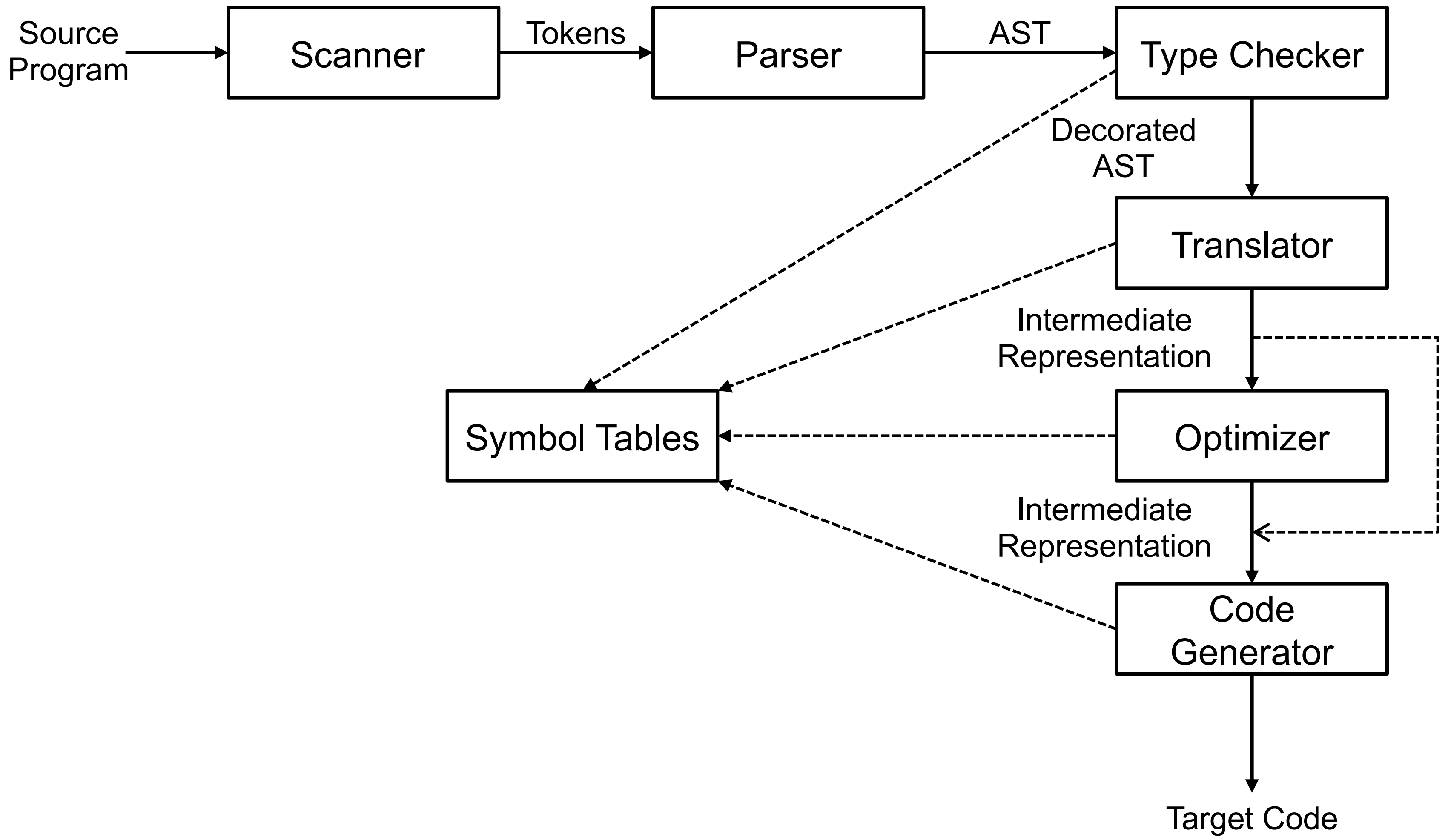
Organization of a Compiler

Compilers generally perform the following tasks:

1) **Analysis** of source program such as scanning and parsing.

2) **Synthesis** of target program such as code generation.

Organization of a Compiler (Cont.)



Revised Figure 1.4: A syntax-directed compiler. AST denotes the Abstract Syntax Tree.

Organization of a Compiler (Cont.)

- Scanner
  - The **scanner** begins the analysis of the source program by reading the input text (character by character) and grouping individual characters into **tokens** such as identifiers and integers.

- Parser
  - The parser is based on a formal syntax specification such as **context free grammar (CFG)**.
  - The parser usually builds an **abstract syntax tree (AST)** as a concise representation of program structure.

Organization of a Compiler (Cont.)

- Type checker (semantic analyzer)
  - The type checker checks and decorates the **semantics** of each AST code.
  - It checks type correctness, reachability and termination, and exception handling.

Organization of a Compiler (Cont.)

- Translator
  - If an AST node is semantically correct, it can be **translated** into IR code (such as byte code in Java) that correctly implements the meaning of the program.

Organization of a Compiler (Cont.)

Optimizer improves IR code’s performance.

The next example optimizes a matrix multiplication program A=B*C (Fig. 14.1) in 2 steps:

1) Fig.14.2 **in-lines 2 methods (* and =)** to avoid the overhead of method call.
2) Fig.14-3 **fuses the outer two loops (for i , for j)** and **eliminates** the “Result” matrix.

```
procedure        ( )
    /* $A$, $B$, and $C$ are declared as $N \times N$ matrices        */
    $A = B \times C$
end

function $\times(Y, Z)$ returns $Matrix$
    if $Y.cols \neq Z.rows$                                          ①
    then   /* Throw an exception */
    else
        for $i = 1$ to $Y.rows$ do
            for $j = 1$ to $Z.cols$ do
                $Result[i, j] \leftarrow 0$
                for $k = 1$ to $Y.cols$ do
                    $Result[i, j] \leftarrow Result[i, j] + Y[i, k] \times Z[k, j]$
    return ($Result$)
end

procedure $=(To, From)$
    if $To.cols \neq From.cols$ or $To.rows \neq From.rows$          ②
    then   /* Throw an exception */
    else
        for $i = 1$ to $To.rows$ do
            for $j = 1$ to $To.cols$ do
                $To[i, j] \leftarrow From[i, j]$
end
```

Figure 14.1: Matrix multiplication using overloaded operators.

```
for $i = 1$ to $N$ do                                                    ③
    for $j = 1$ to $N$ do                                                ④
        $Result[i, j] \leftarrow 0$
        for $k = 1$ to $N$ do
            $Result[i, j] \leftarrow Result[i, j] + B[i, k] \times C[k, j]$    ⑤
for $i = 1$ to $N$ do                                                    ⑥
    for $j = 1$ to $N$ do
        $A[i, j] \leftarrow Result[i, j]$
```

Figure 14.2: Inlining the overloaded operators.

```
for $i = 1$ to $N$ do
    for $j = 1$ to $N$ do
        $A[i, j] \leftarrow 0$
        for $k = 1$ to $N$ do                                          ⑦
            $A[i, j] \leftarrow A[i, j] + B[i, k] \times C[k, j]$      ⑧
```

Figure 14.3: Fusing the loop nests.

Organization of a Compiler (Cont.)

Code generator

- Mapping the IR code (such as Java byte code) or abstract syntax tree into target code (assembly code).

Homework

4. One of the most important tasks of a compiler is detecting incorrect or illegal program components. Some errors are detected at compiler-time. Thus the statement a=b/0; would be marked as containing an illegal divide operation.

   Other errors are detected at run-time. Thus give:

   ```
   c=0;
   a=b/c;
   ```

   an illegal division would be detected as the program executes.
   One difference between compiler-time and run-time errors is that run-time errors are only detected if the illegal code is actually executed.
   Compiler-time errors make an entire program illegal, even if the erroneous code is rarely or never reached.

   Which approach do you recommend? Should a program with compiler-time errors be allowed to execute, with termination only when illegal code is executed? Are there any advantages to making a whole program illegal if some illegal component of it *might* be executed?

No. 4 Solution

4. 1) Compile-time error is a better approach.

2) A program with compiler-time errors SHOULD NOT be allowed to execute.

3) The advantage is that it ensures all the source codes are syntactically correct.

Homework (Cont.)

8. Most programming languages, such as C and C++, are compiled directly into the machine language of a “real” microprocessor (for example, an Inter x86 or Sparc). Java takes a different approach.

   It is commonly compiled into the machine language of the JVM. The JVM is not implemented in its own microprocessor, but is instead interpreted on some existing processor. This allows Java to be run on a wide variety of machines, thereby making it highly platform independent.

   Explain why building an interpreter for a virtual machine like the JVM is easier and faster than building a complete Java compiler. What are the disadvantages of this virtual machine approach?

No. 8 Solution

8. Building an interpreter for a virtual machine is easier than building a complete Java compiler is that there are fewer phases in developing interpreters.

The disadvantages of this virtual machine approach is that the source code must be compiled to byte code, and then a java interpreter can execute it. This takes more time.

Homework (Cont.)

11. C is sometimes called the **universal assembly language** in light of its ability to be very efficiently implemented on a wide variety of computer architecture.

In light of this characterization, some compiler writers have chosen to generate C code as their output instead of a particular machine language.

What are the advantages to this approach to compilation? Are there any disadvantages?

No. 11 Solution

11. Generating C code is more portable than generating an assembly code for a particular machine language.

However, the source code will be translated into C code, and then to assembly code. This is time-consuming.