Compiler Design

Fatemeh Deldar

Isfahan University of Technology

1402-1403

References

1. **Compilers: Principles, Techniques, and Tools** (Second Edition), Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Addison-Wesley, 2007.

2. **Modern Compiler Design** (Second Edition), D. Grune, H. Bal, C. Jacobs, and K. Langendoen, John Wiley, 2012.

Syllabus

- **Introduction**
- **Lexical Analysis**
- **Syntax Analysis**
- **Semantic Analysis**
- **Intermediate-Code Generation**
- **Run-Time Environments**
- **Code Generation**
- **Machine-Independent Optimizations**

Introduction

Introduction

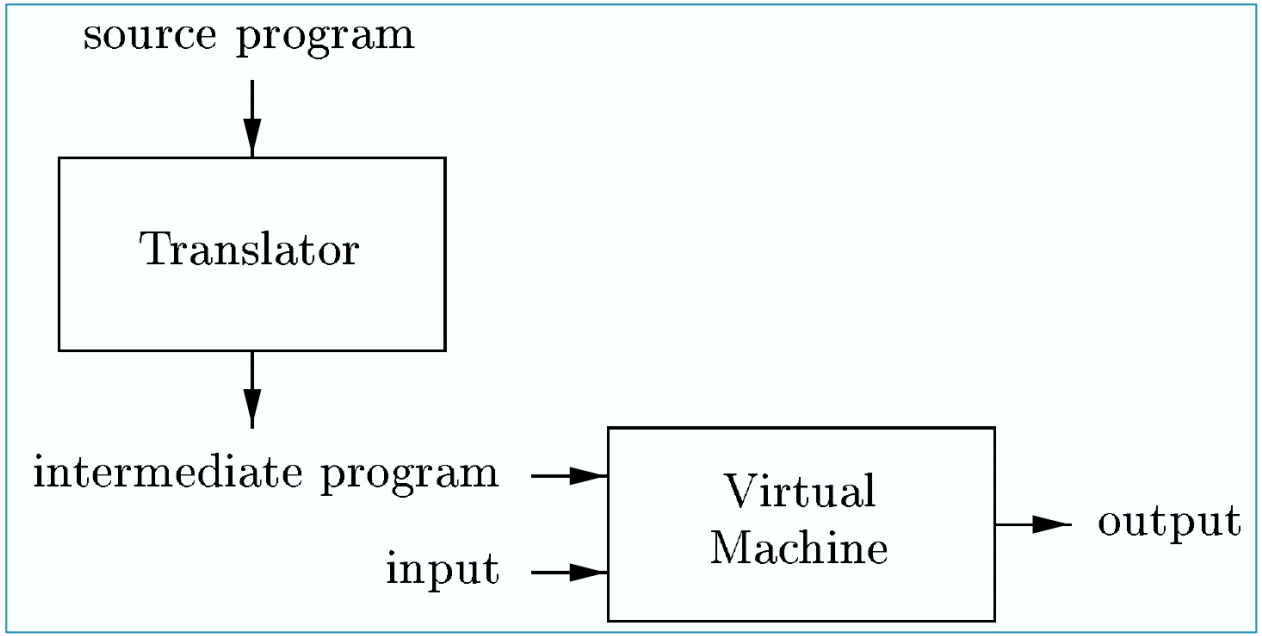
- Programming languages are notations for describing computations to people and to machines
- Before a program can be run, it first must be **translated** into a form in which it can be executed by a computer
  - **The software systems that do this translation are called compilers**
- **This course is about how to design and implement compilers**

Compiler vs. Interpreter

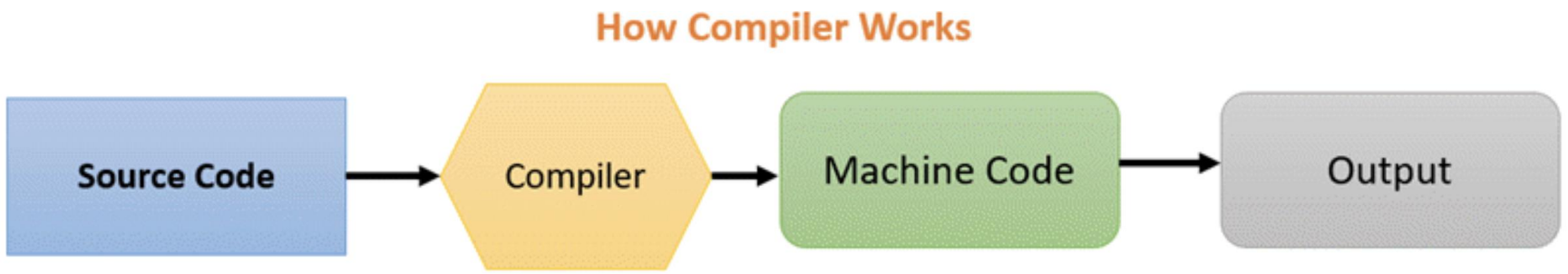
- **A compiler** is a program that can read a program in one language (**the source language**) and translate it into an equivalent program in another language (**the target language**)
- **An interpreter** directly executes the operations specified in the source program on inputs supplied by the user
- ***The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs***
- ***An interpreter can usually give better error diagnostics than a compiler, because it executes the source program statement by statement***

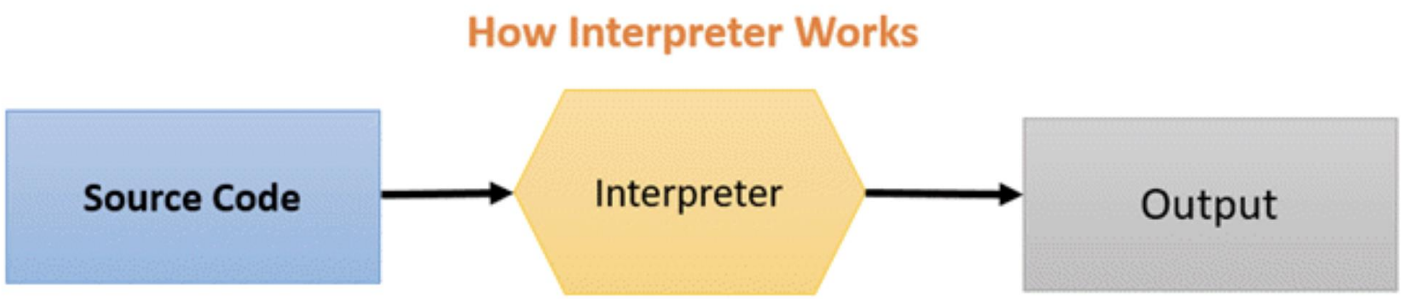
Compiler vs. Interpreter

- **Example**
  - Java language processors combine compilation and interpretation
    - A Java source program **first be compiled** into an intermediate form called bytecodes
    - The bytecodes are **then interpreted** by a virtual machine



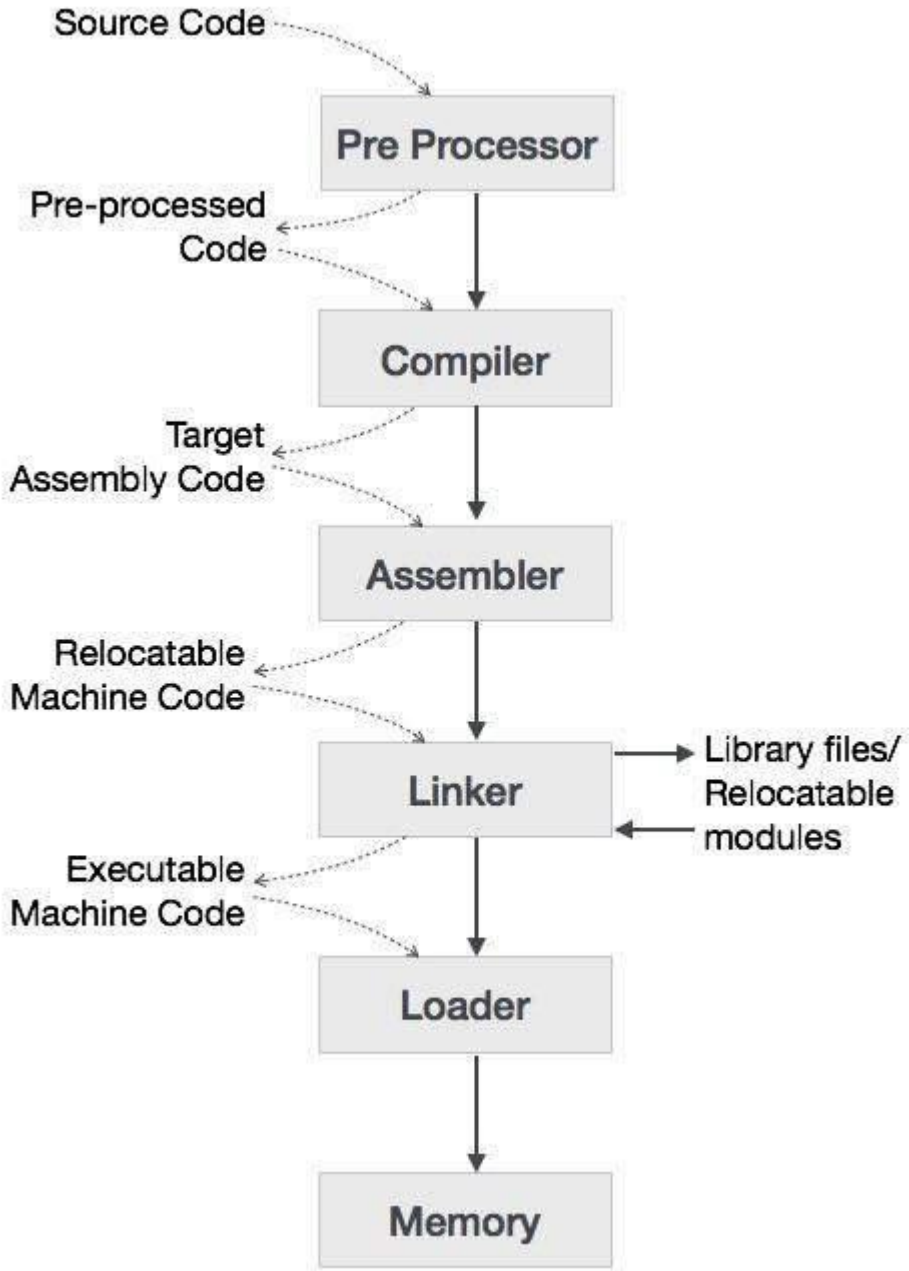
Compiler vs. Interpreter





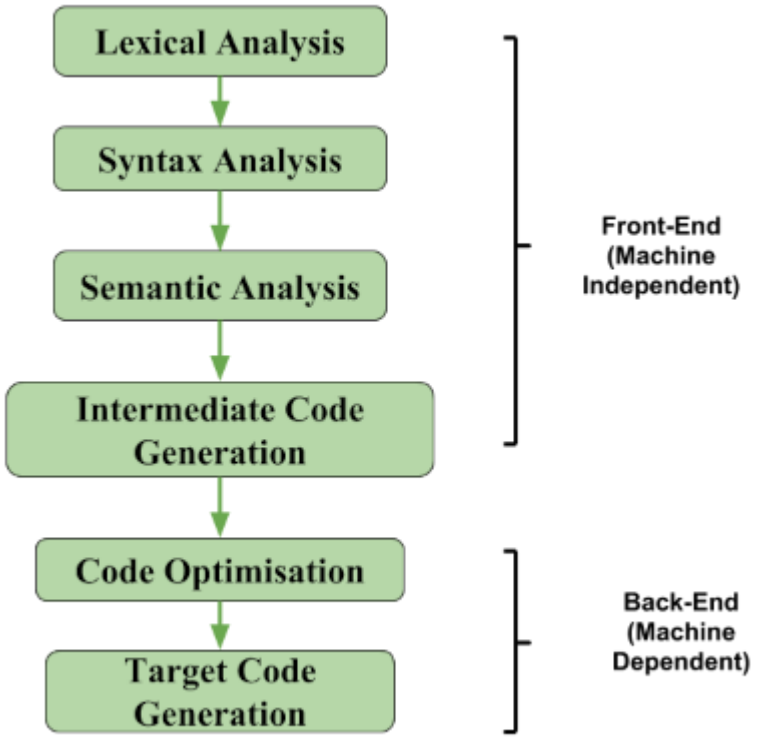
A Language-Processing System

- **Preprocessor**
  - Collecting the source program
- **Compiler**
  - Producing an assembly-language program
- **Assembler**
  - Producing relocatable machine code
- **Linker**
  - Resolving external memory addresses, where the code in one file may refer to a location in another file
- **Loader**
  - Putting together all of the executable object files into memory for execution



The Structure of a Compiler

- Two general parts of a compiler
  - **The analysis part (Front-end)**
  - **The synthesis part (Back-end)**



The Structure of a Compiler

- Two general parts of a compiler
  - **The analysis part**
    - This 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**
    - The analysis part also collects information about the source program and stores it in a data structure called a **symbol table**
  - **The synthesis part**
    - The synthesis part **constructs the desired target program** from the intermediate representation and the information in the symbol table

The Structure of a Compiler

- **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:

<token-name, attribute-value>

An abstract symbol that is used during syntax analysis

Points to an entry in the symbol table for this token

The Structure of a Compiler

- **Lexical Analysis**
  - **Example**

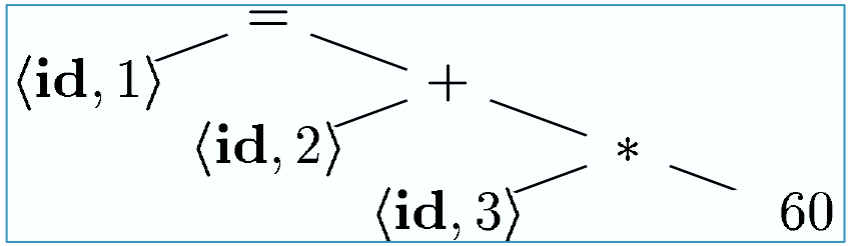
position = initial + rate * 60

$\langle \mathbf{id}, 1 \rangle \; \langle = \rangle \; \langle \mathbf{id}, 2 \rangle \; \langle + \rangle \; \langle \mathbf{id}, 3 \rangle \; \langle * \rangle \; \langle 60 \rangle$

  - **Example: Lexical errors**
    - int 5temp;
    - a = 123.23.45;
    - char s[10] = “ali;

The Structure of a Compiler

- **Syntax Analysis**
  - The second phase of the compiler is syntax analysis or parsing
  - The parser uses the first components of the tokens produced by the lexical analyzer **to create a tree-like intermediate representation that depicts the grammatical structure of the token stream**
  - 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

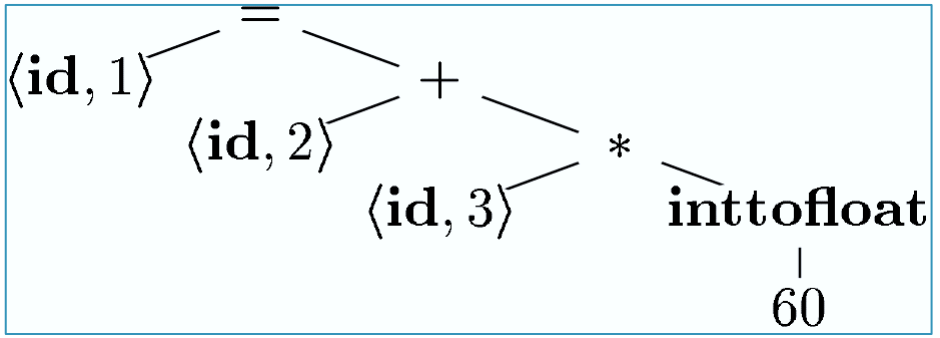


The Structure of a Compiler

- **Syntax Analysis**
  - **Example: Syntax errors**
    - a b =;
    - int q = 6
    - if (a > 1

The Structure of a Compiler

- **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**
  - It also **gathers type information** and saves it in either the syntax tree or the symbol table



The Structure of a Compiler

- **Semantic Analysis**
  1. **Type checking**
     - **Example**
       - The compiler must report an error if a floating-point number is used to index an array
  2. **Type casting**
     - **Example**
       - The compiler may convert the integer into a floating-point number
  3. **Redefine variable**
  4. **Check function parameters**

The Structure of a Compiler

- **Intermediate Code Generation**
  - Many compilers **generate an explicit low-level or machine-like intermediate representation**
  - This intermediate representation should have two important properties
    - It should be easy to produce
    - It should be easy to translate into the target machine
  - Three-address code consists of a sequence of assembly-like instructions with three operands per instruction

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

The Structure of a Compiler

- **Code Optimization**
  - The machine-independent code-optimization phase attempts to **improve the intermediate code** so that better target code will result
  - There is a great variation in the amount of code optimization different compilers perform

```
t1 = id3 * 60.0
id1 = id2 + t1
```

The Structure of a Compiler

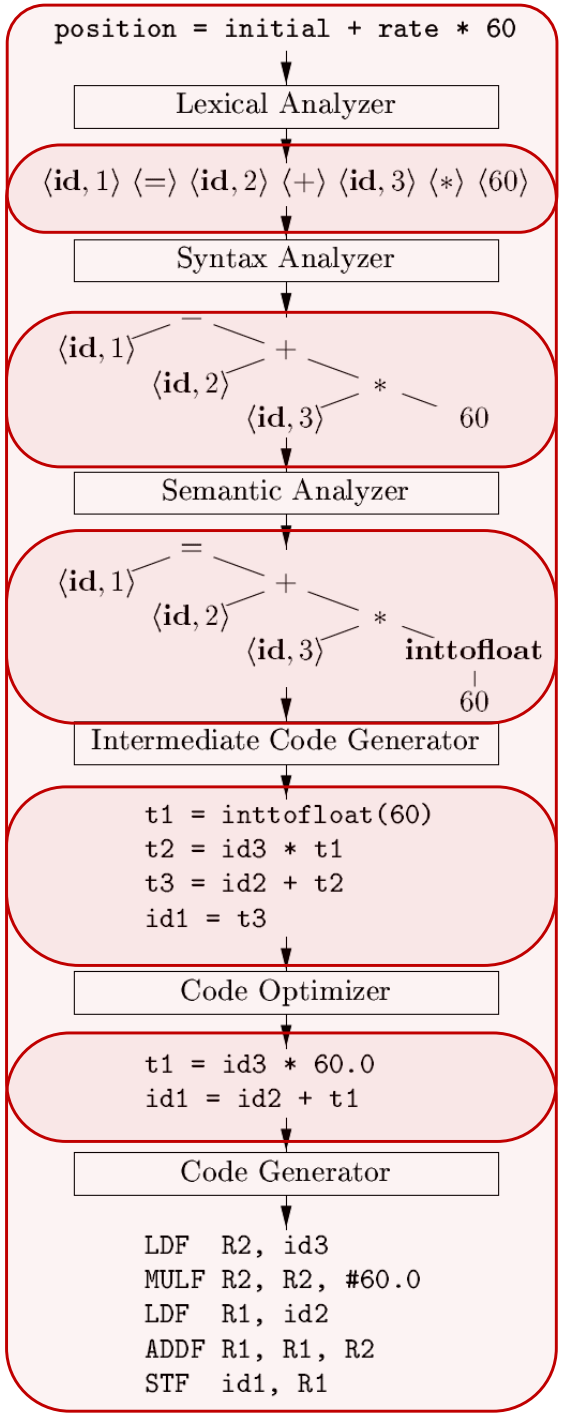
- **Code Generation**
  - The code generator takes as input an intermediate representation of the source program and maps it into the target language

```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2
ADDF R1,  R1, R2
STF  id1, R1
```

The Structure of a Compiler

1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE



Lexical Analysis

Lexical Analysis

- The main task of the lexical analyzer:
  1. **Read the input characters of the source program**
  2. **Group them into lexemes**
  3. **Produce as output a sequence of tokens**

- **Lexical errors**
  - It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error
  - **Example**

    fi (a == f(x)) ...