Compiler Design

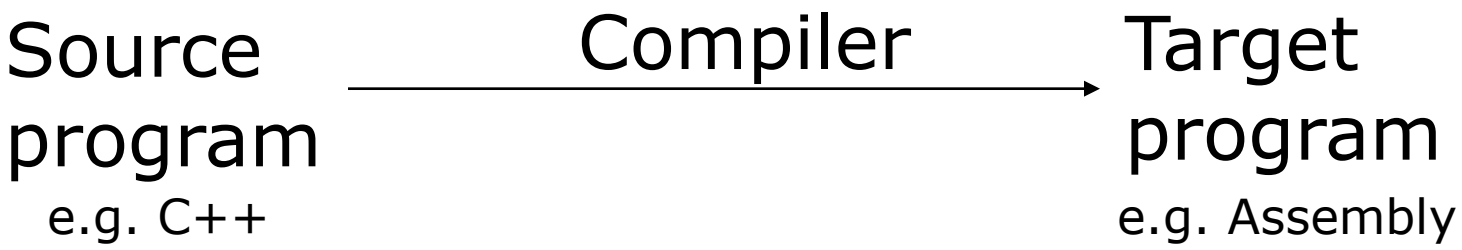
Outline

- Introduction
- Lexical Analysis
- Syntax Analysis
- Syntax Directed Translation
- Type Checking
- Intermediate Code Generation
- Code Generation
- Code Optimization

Introduction

Introduction

- A compiler is a *program* that reads a program written in one language and translates it into an equivalent program in another language.



- A compiler also reports any errors in the source program that it detects during the translation process

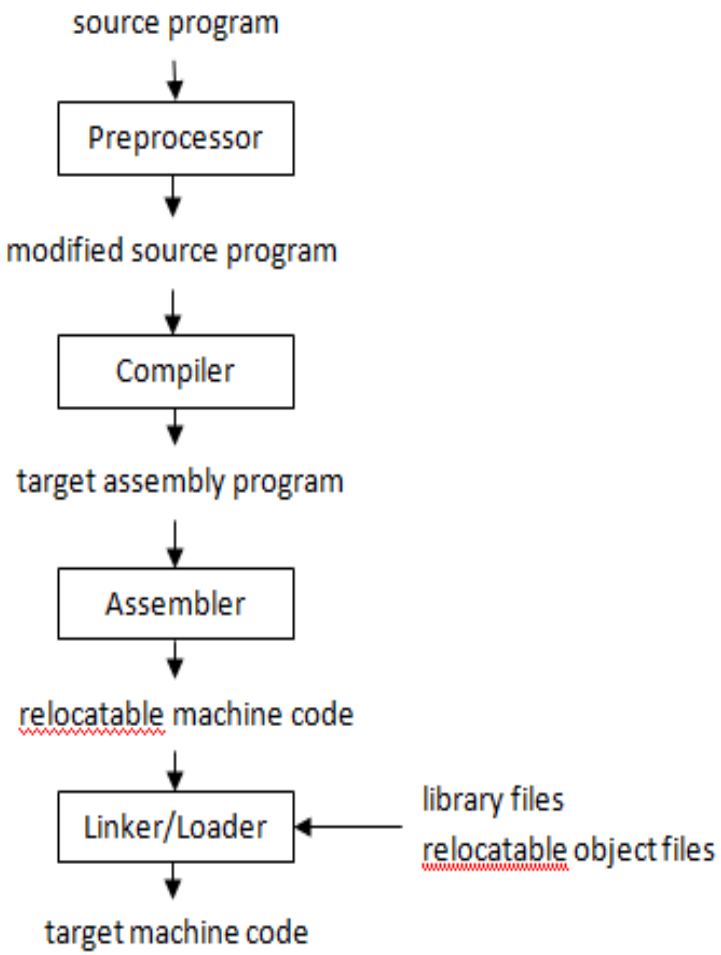


Figure 1.2: A language processing system

- There are other programs used with the compiler, these are:
  - Preprocessor
  - Assembler
  - Loader/linker

- **Preprocessor**
  - A source program may be divided into modules stored in separate files.
  - A preprocessor is a separate program that is called before actual translation begins
  - The task of collecting source program (file inclusion)is performed by a **preprocessor**
  - The preprocessor may also delete comments and perform macro substitutions (a macro is a shorthand description of a repeated sequence of text
  - The modified source program is fed to a compiler

- **Assembler**
  - The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as an output and easier to debug
  - The assembly language program is then processed by a program called assembler that produces a relocatable machine code as its output

- **Linker**
  - Large programs are often compiled in pieces, so that the relocatable machine code may have to be linked with other relocatable object files and library files into the code actually runs on the machine
  - Linker collects code separately compiled or assembled in different object files into a file that is directly executable.
  - A linker also connects an object program to the code for standard library functions.
  - The *linker* resolves external memory addresses, where the code in one file may refer to a location in another file

- **Loader**
  - Often a compiler, assembler, or linker will produce code that is not yet completely fixed and ready to execute, but whose principal memory references are all made relative to undetermined starting location that can be anywhere in memory.
  - Such code is said to be relocatable, and a loader will resolve all relocatable addresses relative to a given base, or starting address
  - The *loader* then puts together all executable object files into memory for execution

The Structure of a Compiler

- A compiler consists internally of a number of steps, or phases, that perform distinct logical operations.
- It is helpful to think of theses phases as separate pieces within the compiler, and they may indeed be written as separate coded operations although in practice they often grouped together.
- There are two parts responsible for mapping source program into a semantically equivalent target program: **analysis and synthesis**

- The *analysis* part breaks up the program into constituent pieces and imposes grammatical structure on them and creates an intermediate representation of the source program
- If analysis part detects errors (syntax and semantic), it provides informative messages
- The analysis part also collects information about the source program and stores it in a data structure called *symbol table*, which is passed along with an intermediate representation to the synthesis part

- During analysis, the operations implied by the source program are determined and recorded in hierarchical structure called a tree.
- 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 the ***back end***
- The compilation process operates as a *sequence of phases* each of which transforms one representation of the source program into another

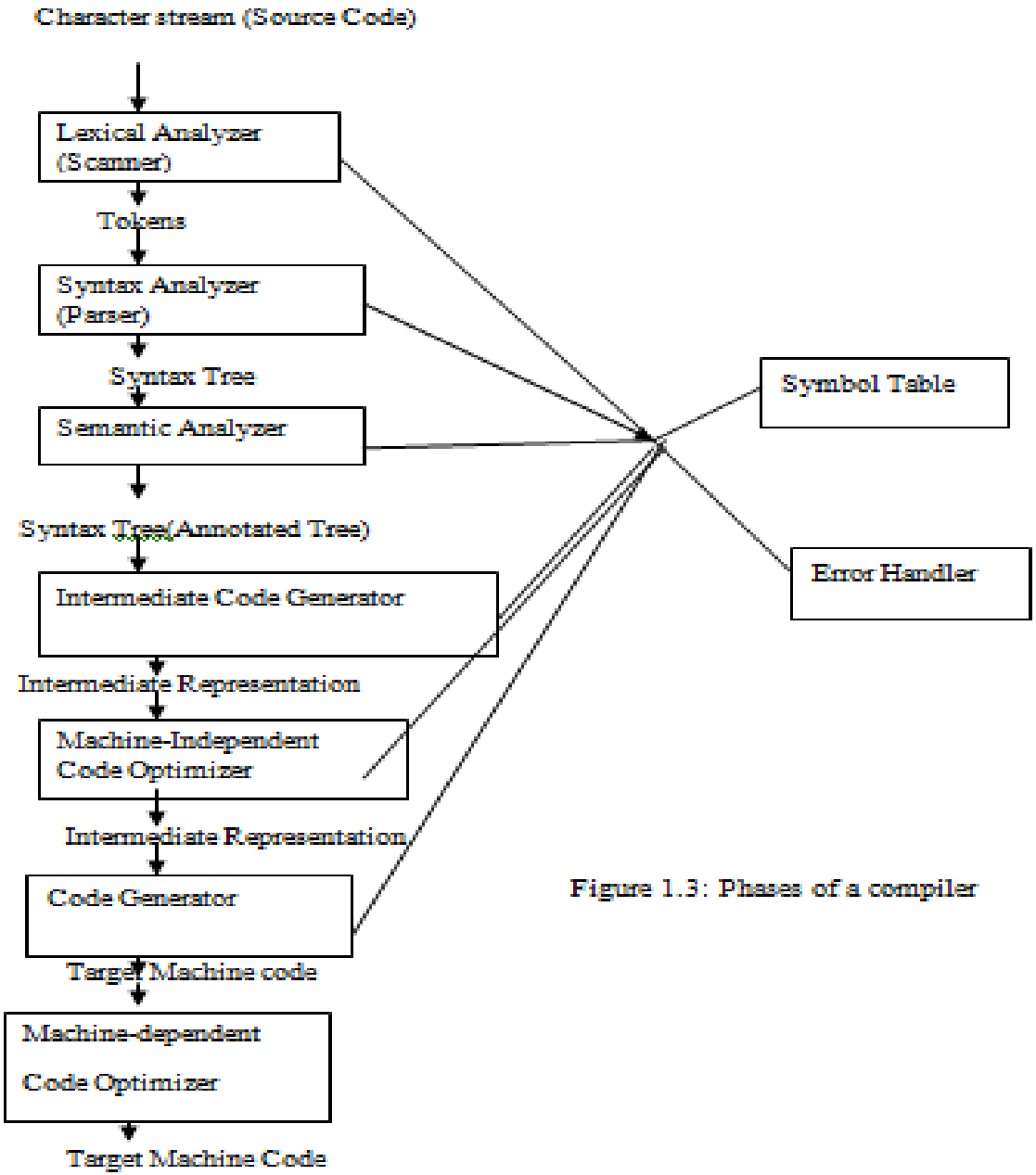


Figure 1.3: Phases of a compiler

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

- **Analysis of the source program**
- Analysis is composed of three phases:
  - Linear/Lexical analysis (Scanning)
  - Hierarchical/Syntax analysis (Parsing)
  - Semantic 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 them into meaningful sequences called **lexemes**
- For each lexeme the lexical analyzer produces a **token** of the form:
- ***<token-name, attribute-value>*** that it passes on to the subsequent phase, syntax analysis

- In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second attribute value points to an entry in the symbol table for this token
- For example, suppose a source program contains the following assignment statement

position = initial + rate * 60

- The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:
- position is a lexeme that would be mapped into a token <id, 1>, where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position. The symbol table entry holds information about the identifier, such as its name and type
- = is a lexeme that is mapped into the token <=>. Since it needs no attribute value, the second component is omitted
- initial is a lexeme that would be mapped into a token <id, 2>, where 2 points to the symbol table entry for position

- + is a lexeme that is mapped into the token <+>
- rate is a lexeme that would be mapped into a token <id, 3>, where 3 points to the symbol table entry for rate
- * is a lexeme that is mapped into the token <*>
- 60 is a lexeme that is mapped into the token <60>
- After lexical analysis, the sequence of tokens in the previous assignment statement is

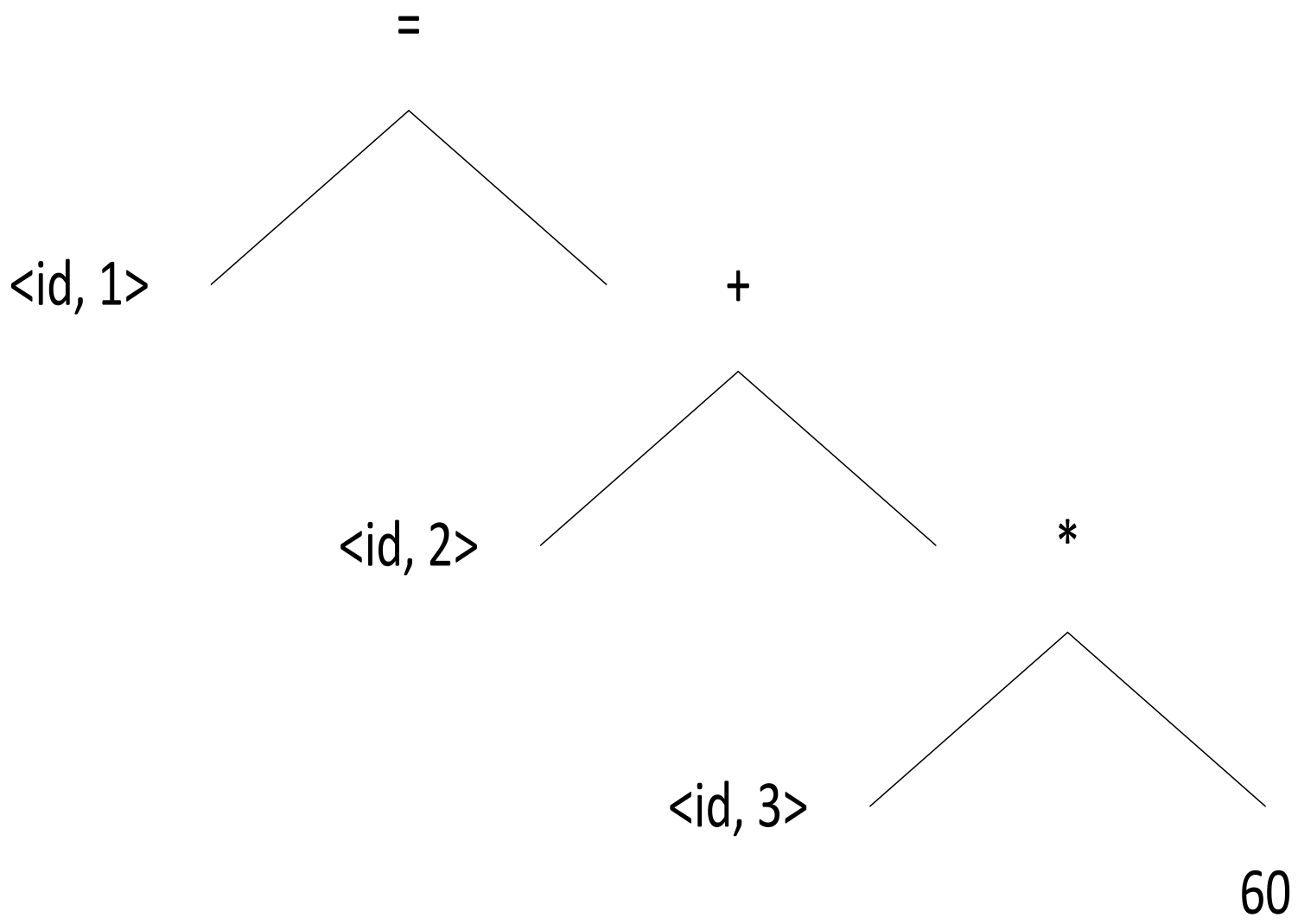
<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>

- In this representation, the token names =, +, and * are abstract symbols for the assignment, addition, and multiplication operators, respectively
- A scanner may perform other operations along with the recognition of tokens. For example,
  - It may enter identifiers into the symbol table
  - It may enter literals into literals table
  - It may remove blanks, new lines and comments.

Hierarchical/Syntax analysis (Parsing)

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

Figure 1.4: Syntax tree



- This tree shows the order in which the operations in the assignment statement are to be performed
- Multiplication is done first, followed by addition, and finally assignment
- Syntactic rules of the source language are given via a Grammar.
- Most syntactic errors in the source program are caught in this phase

Semantic Analysis

- The semantics of a program is its “meaning” as opposed to its syntax, or structure.
- The semantics of a program determines its runtime behavior.
- Most programming languages have features that can be determined prior to execution. Such features are referred to as static semantics, and the analysis of such semantics is the task of the **semantic analyzer**.

- The dynamic semantics of a program-those properties of a program that can only be determined by executing it- cannot be determined by a compiler.
- Typical static features of common programming languages include **declarations** and **type checking.**
- 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, for subsequent use during intermediate code generation

Intermediate Code Generation

- After syntax and semantic analysis, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine

- This intermediate representation should have two properties: it should be **easy to produce** and it should **be easy to translate** it into the target machine

- One way of representing the intermediate code is using the three-address code which consists of an assembly like instructions with a **maximum of three operands** per instruction (or at most one operator at the right side of an assignment operator)

- Example: For the previous example, the intermediate code representation using three address code is given below:

  t1 = intToFloat(60)

  t2 = id3 * t1

  t3 = id2 + t2

  id1 = t3

- The compiler must also generate temporary name to hold the value computed by a three-address instruction

- **Synthesis of the target program**
- Two phases:
  - **Code optimization**
  - **Code generation**

1. Code optimization

- This phase changes the IC so that the code generator produces a faster and less memory consuming program.
- The optimized code does the same thing as the original (non-optimized) code but with less cost in terms of CPU time and memory space.
- There are several techniques of optimizing code and they will be discussed in the last chapter.

Code optimization (cont'd)

- Example

  Unnecessary lines of code in loops (i.e. code that could be executed outside of the loop) are moved out of the loop.

```
for(i=1; i<10; i++){
        x = y+1;
        z = x+i;
}

x = y+1;
for(i=1; i<10; i++)
        z = x+i;
```

Example:

- For the previous example
  - t1 = id3 * 60.0
  - id1 = id2 + t1

2. Code generation

- The final phase of the compiler.
- Generates the target code in the target language (e.g. Assembly)
- The instructions in the IC are translated into a sequence of machine instructions that perform the same task.

Example:

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

Grouping of phases

- The phases seen above are not always found in a compiler.

- In practice most compilers are divided into a **front end** that depends on the source language and a **back end** that depends on the target language.

Grouping of phases (cont'd)

- A **pass** consists of reading an input file and writing an output file.
- Several phases may be grouped in one pass.
- It is desirable to have few passes to avoid I/O overhead.
- However, this may require too much memory to store intermediate information.

Compiler construction tools

- Various tools are used in the construction of the various parts of a compiler.

- Using these tools, it is now relatively easier to construct a compiler.

Tools (cont'd)

1. **Scanner generators**

   Ex. Lex, flex, JLex

   These tools generate a scanner /lexical analyzer/ given a regular expression.

2. **Parser Generators**

   Ex. Yacc, Bison, CUP

   These tools produce a parser /syntax analyzer/ given a Context Free Grammar (CFG) that describes the syntax of the source language.

Tools (cont'd)

3. **Syntax directed translation engines**

   Ex. Cornell Synthesizer Generator

   It produces a collection of routines that walk the parse tree and execute some tasks.

4. **Automatic code generators**

   Take a collection of rules that define the translation of the IC to target code and produce a code generator