# Experiment No. 1

**Aim:** Case study on System Programming

**Theory:**

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; see Fig. 1.1. An important role of the compiler is to report any errors in the source program that it detects during the translation process.
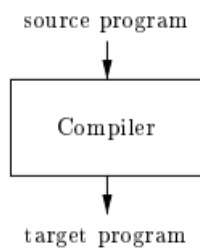


Figure 1.1: A compiler

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs; see Fig. 1.2.
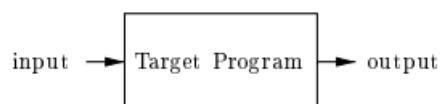


Figure 1.2: Running the target program

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, as shown in Fig. 1.3.
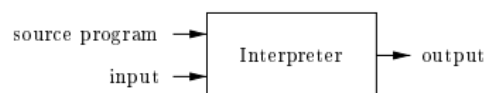


Figure 1.3: An interpreter

Example: Java language processors combine compilation and interpretation, as shown in Fig. 1.4. 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. In order to achieve faster processing of inputs to outputs, some Java compilers, called just-in-time compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input.
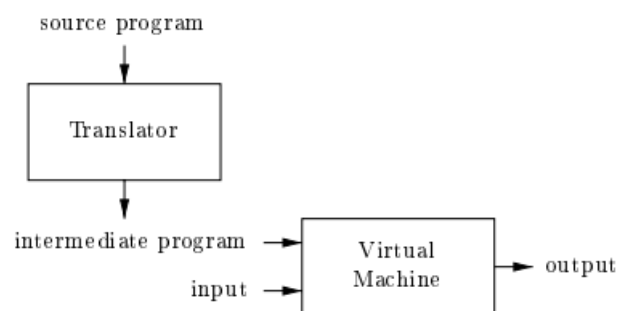
Figure 1.4: A hybrid compiler

In addition to a compiler, several other programs may be required to create an executable target program, as shown in Fig. 1.5. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a preprocessor. The preprocessor may also expand shorthands, called macros, into source language statements.

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

Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually

runs on the machine. The linker resolves external memory addresses, where the code in one file may refer to a location in another file. The loader then puts together all of the executable object files into memory for execution.
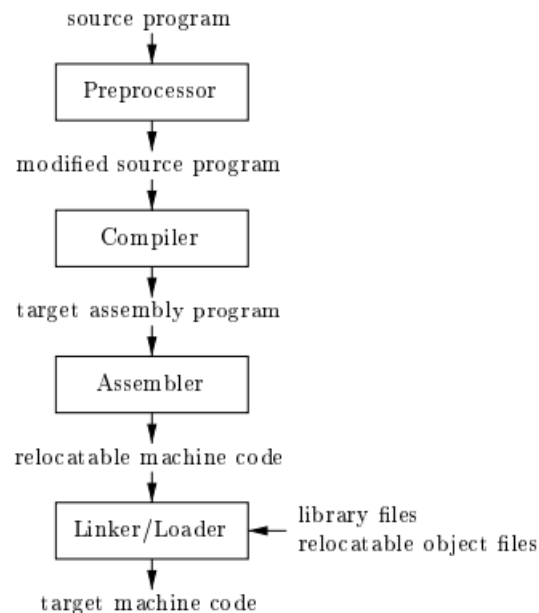


Figure 1.5: A language-processing system

## 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 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. If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in Fig. 1.6. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the entire source program, is used by all phases of the compiler. Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the backend can produce a better target program than it would have otherwise produced from an unoptimized intermediate representation. Since optimization is optional, one or the other of the two optimization phases shown in Fig. 1.6 maybe missing.

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
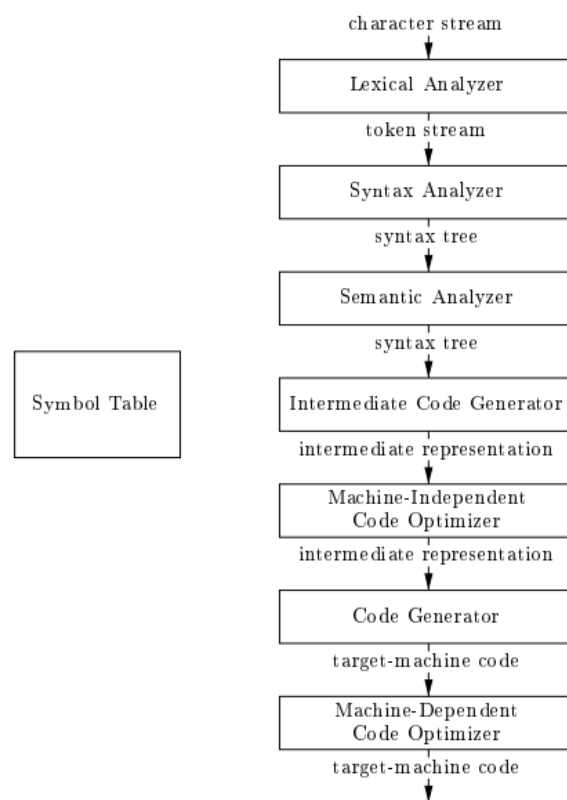Code Optimizer

target-machine code

↓

Figure 1.6: Phases 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> 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 component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation. For example, suppose a source program contains the assignment statement

$$position\ =\ initial\ +\ rate\ *\ 60\ (1.1)$$

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. 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 for an identifier holds information about the identifier, such as its name and type.

2. The assignment symbol = is a lexeme that is mapped into the token <=>. Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as assign for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

3. Initial is a lexeme that is mapped into the token <id, 2>, where 2 points to the symbol-table entry for initial.

 4. + is a lexeme that is mapped into the token <+>.

5. Rate is a lexeme that is mapped into the token <id, 3>, where 3 points to the symbol-table entry for rate.

6. * is a lexeme that is mapped into the token <*>.

7. 60 is a lexeme that is mapped into the token h60i.1 Blanks separating the lexemes would be discarded by the lexical analyzer. Figure 1.7 shows the representation of the assignment statement (1.1) after lexical analysis as the sequence of tokens

$$< id, 1 > <=> < id, 2 > < +> < id, 3 > < 60 > \quad (1.2).$$

In this representation, the token names =, +, and are abstract symbols for the assignment, addition, and multiplication operators, respectively.
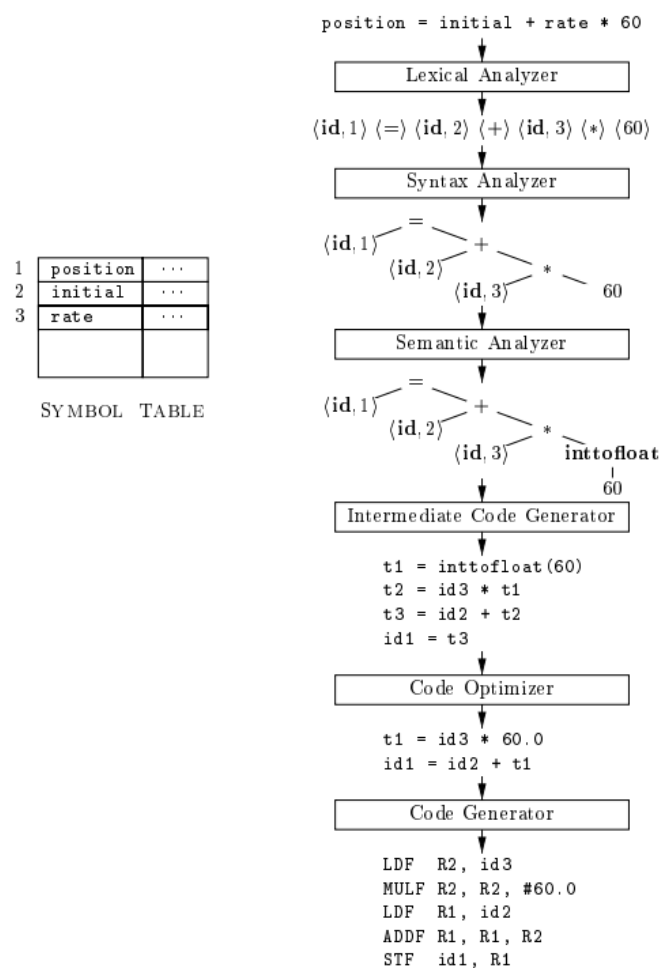


Figure 1.7: Translation of an assignment statement

## 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. A syntax tree for the token stream (1.2) is shown as the output of the syntactic analyzer in Fig. 1.7. This tree shows the order in which the operations in the assignment

$$position\ =\ initial\ +\ rate\ *\ 60$$

are to be performed. The tree has an interior node labelled with hid; 3i as its left child and the integer 60 as its right child. The node hid; 3i represents the identifier rate. The node labelled makes it explicit that we must first multiply the value of rate by 60. The node labelled + indicates that we must add the result of this multiplication to the value of initial. The root of the tree, labelled =, indicates that we must store the result of this addition into the location for the identifier position. This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition. The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program.

## 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, for subsequent use during

intermediate-code generation. An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array. The language specification may permit some type conversions called coercions. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Such a coercion appears in Fig. 1.7. Suppose that position, initial, and rate have been declared to be floating-point numbers, and that the lexeme 60by itself forms an integer. The type checker in the semantic analyzer in Fig. 1.7discovers that the operator is applied to a floating-point number rate and n integer 60. In this case, the integer may be converted into a floating-point number. In Fig. 1.7, notice that the output of the semantic analyzer has an extra node for the operator int to float, which explicitly converts its integer argument into a floating-point number.

## Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis. After syntax and semantic analysis of the source program, 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 important properties: it should be easy to produce and it should be easy to translate into the target machine. The output of the intermediate code generator in Fig. 1.7 consists of the three-address code sequence

$$t1 = inttofloat(60)$$

$$t2 = id3 * t1$$

$$t3 = id2 + t2$$

$$id1 = t3$$

$$(1.3)$$

There are several points worth noting about three-address instructions. First, each three-address assignment instruction has at most one operator on the right side. Thus, these instructions fix the order in which operations are to be done; the multiplication precedes the addition in the source program (1.1). Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction. Third, some "three-address instructions" like the first and last in the sequence (1.3), above, have fewer than three operands.

## Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power. For example, a straightforward algorithm generates the intermediate code (1.3), using an instruction for each operator in the tree representation that comes from the semantic analyzer. A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the int to float operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, t3 is used only once to transmit its value to id1 so the optimizer can transform (1.3) into the shorter sequence

$$t1 = id3 * 60.0$$

$$id1 = id2 + t1$$

$$(1.4)$$

There is a great variation in the amount of code optimization different compilers perform. In those that do the most, the so-called "optimizing compilers," a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.

## Code Generation

The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables. For example, using registers R1 and R2, the intermediate code in (1.4) might get translated into the machine code

$$\text{LDF R2, id3}$$

$$\text{MULF R2, R2, 60.0}$$

$$\text{LDF R1, id2}$$

$$\text{ADDF R1, R1, R2}$$

$$\text{STF id1, R1}$$

$$(1.5)$$

The first operand of each instruction specifies a destination. The F in each instruction tells us that it deals with floating-point numbers. The code in (1.5) loads the contents of address id3 into register R2, then multiplies it with floating-point constant 60.0. The # signifies that 60.0 is to be treated as an immediate constant. The third instruction moves id2 into register R1 and the fourth adds to it the value previously computed in register R2. Finally, the value in register R1 is stored into the address of id1, so the code correctly implements the assignment statement (1.1).

## Symbol Table Management

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned. The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly

## Compiler -Construction Tools

The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on. In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler. These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms. The most successful tools

are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler. Some commonly used compiler-construction tools include

1. Parser generators that automatically produce syntax analyzers from a grammatical description of a programming language.

2. Scanner generators that produce lexical analyzers from a regular-expression description of the tokens of a language.

3. Syntax-directed translation engines that produce collections of routines for walking a parse tree and generating intermediate code.

4. Code-generator generators that produce a code generator from a collection of rules for translating each operation of the intermediate language in to the machine language for a target machine.

5. Data- ow analysis engines that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data- ow analysis is a key part of code optimization.

6. Compiler-construction toolkits that provide an integrated set of routines for constructing various phases of a compiler.