CHAPTER ONE
1. Introduction

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

A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes.

1.1. Language Processors

A compiler is a program that can read a program written in one language, called the source language, and translate it into an equivalent program in another language – called the target language (see Figure 1.1). The target program is then provided the input to produce output.

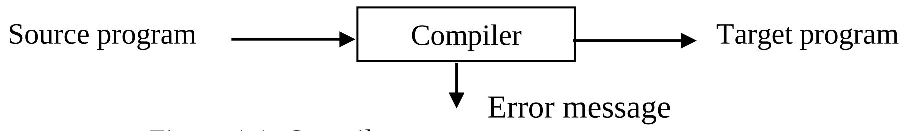


Figure 1.1: Compiler

A compiler also reports any errors in the source program that it detects during the translation process. If the target program is an executable machine-language program, it can then be called the user to process input and produce output.

A compiler acts as a translator, transforming human-oriented programming languages into computer-oriented machine languages. It is a program that translates an executable program in one language into an executable program in another language.

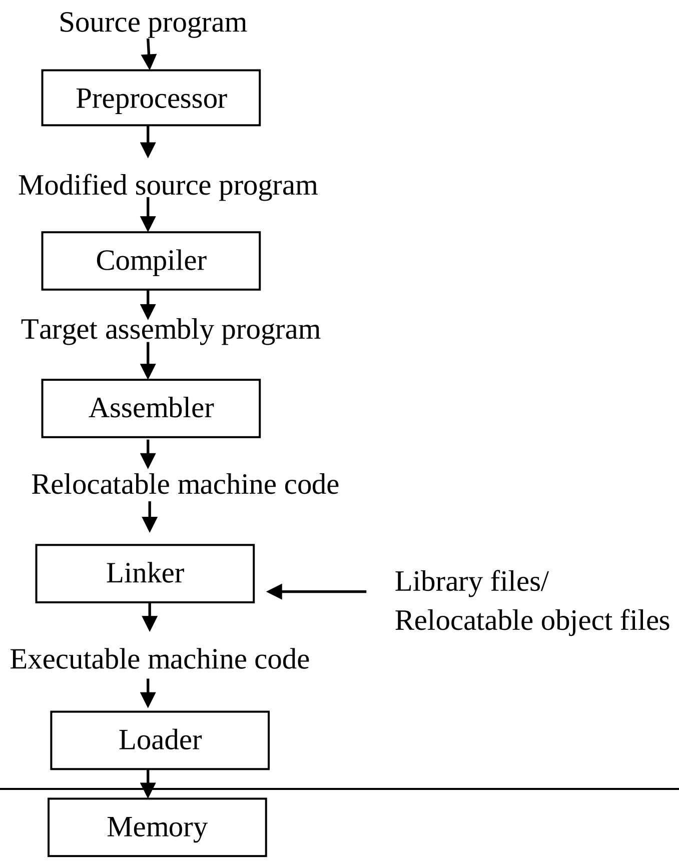


Figure 1.2: A language processing system

The high-level language is converted into binary language in various phases. A compiler is a program that converts high-level language to assembly language. Similarly, an assembler is a program that converts the assembly language to machine-level language.

Preprocessor: A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

The task of a *preprocessor* (a separate program) is collecting modules of a program stored in separate files. It may also expand short hands, called macros, into source language statements. The modified source program is fed to a compiler.

Interpreter: An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it; whereas a compiler reads the whole program even if it encounters several errors.

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

The machine-language target 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. Several other programs may be needed in addition to a compiler to create an executable program as shown in Figure 1.2.

Assembler: An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

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: Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program.

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. The *linker* resolves external memory addresses, where the code in one file may refer to a location in another file.

Loader: Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution. It puts together all executable object files into memory for execution.

1.2. The phases of a Compiler

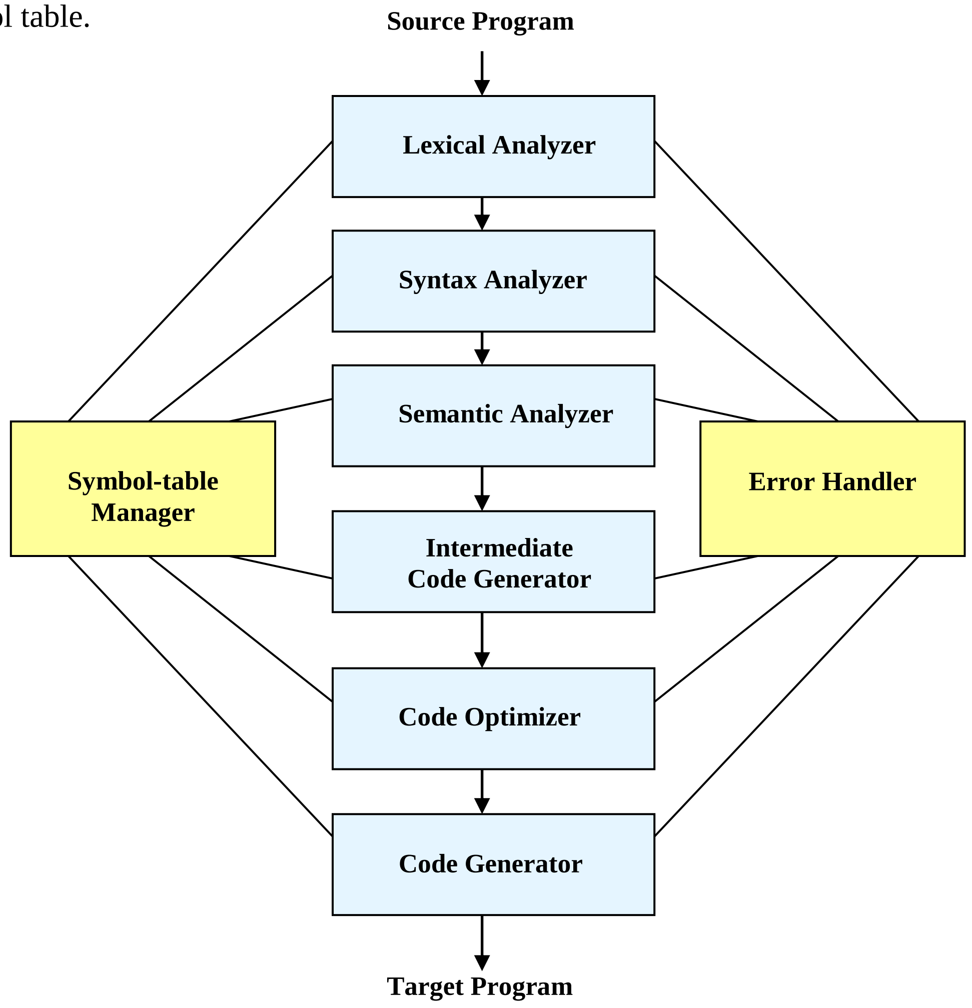
The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.

There are two parts responsible for mapping source program into a semantically equivalent target program: analysis and synthesis phases.

In **analysis phase**, an intermediate representation is created from the given source program. The *analysis* part breaks up the program into constituent pieces and imposes grammatical structure on them. It then uses this structure to create 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.

The analysis phase of the compiler is known as the front-end of the compiler, it reads the source program, divides it into core parts, and then checks for lexical, grammar, and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

The **synthesis part** constructs the desired target program from the intermediate representation and the information in the symbol table. The synthesis phase is known as the back-end of the compiler, it generates the target program with the help of intermediate source code representation and symbol table.



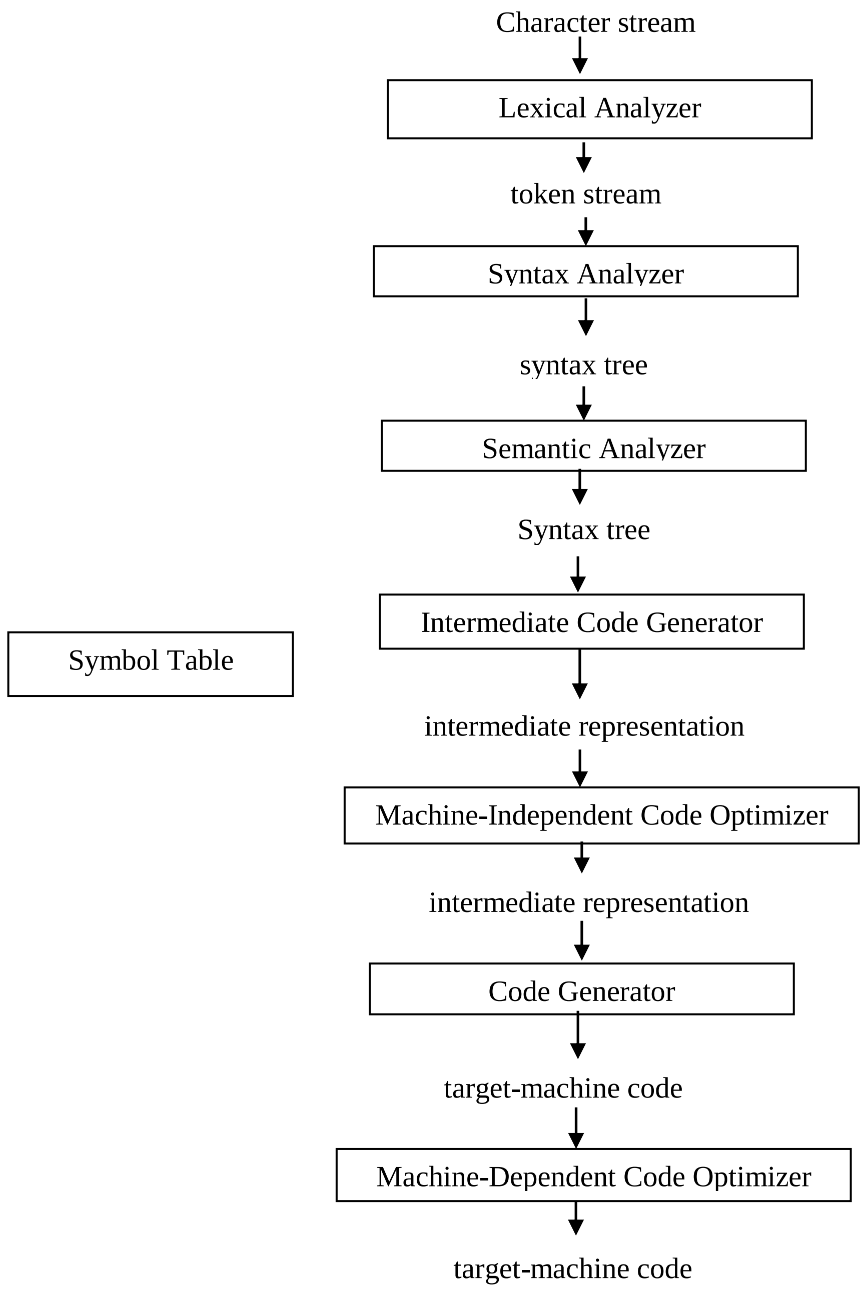


Figure 1.3: Phases of a compiler

A typical decomposition of a compiler into phases is shown in in the above figures. In practice, several phases may be grouped together and the intermediate representations between the grouped phases need not be constructed explicitly

The analysis phase consists of Lexical Analyzer, Syntax Analyzer and Semantic Analyzer. The synthesis phase comprises of Intermediate Code Generator, Code Generator, and Code Optimizer.
The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

1.2.1. 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 them into meaningful sequences called lexemes. The text is read and divided into tokens, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number.

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 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 following assignment statement

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

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 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 it needs no attribute value, the second component is omitted. 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 would be mapped into a 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 would be mapped into a 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 <**60**>

The blanks separating the characters of these tokens would normally be eliminated during lexical analysis.

After lexical analysis, the sequence of tokens in equation 1.1 are

<id, 1> < = > <id, 2> <+> <id, 3> <*> <60> (1.2)

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

1.2.2. Syntax Analysis

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. In a syntax tree, each interior node represents an operation and the children of the node represent the arguments of the operation.

The syntax tree for the previous token stream in Equation 1.2 is shown in Figure 1.4

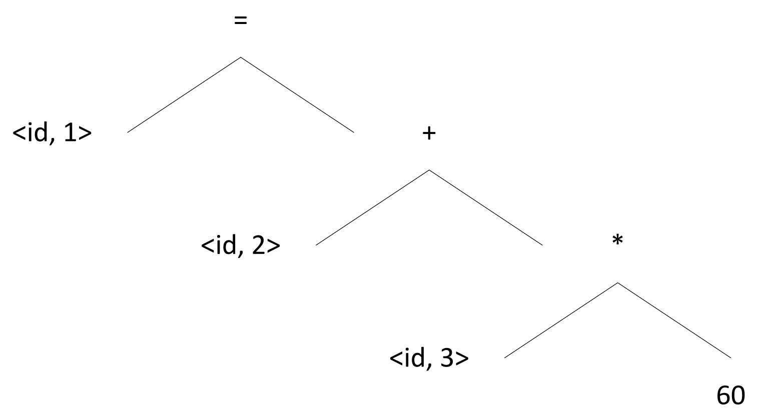


Figure 1.4: Syntax tree

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 labeled ***** with **<id, 3>** as its left child and the integer 60 as its right child. The node **<id, 3>** represents the identifier *rate*. The node labeled * makes it explicit that we must first multiply the value of *rate* by 60. The node labeled + indicates that we must add the result of this multiplication to the value of *initial*. The root of the tree, labeled =, 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.

1.2.3. 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.

This phase analyses the syntax tree to determine if the program violates certain consistency requirements, e.g., if a variable is used but not declared or if it is used in a context that does not make sense given the type of the variable.

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.

Suppose that *position*, *initial* and *rate* have been declared to be floating-point numbers, and lexeme 60 by itself forms an integer. The type checker in the semantic analyzer in Fig. 1.5 discovers that the operator * is applied to a floating-point number rate and an 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 ***inttofloat***, which explicitly converts its integer argument into a floating-point number. Semantic analyzer first converts integer 60 to a floating point number before applying *

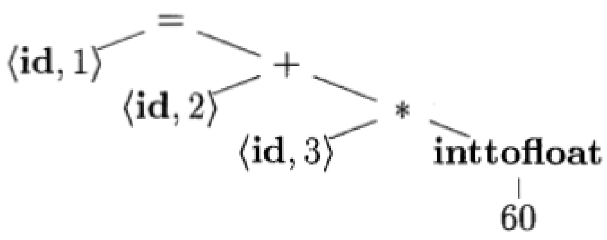


Figure 1.5: Syntax tree

1.2.4. Intermediate Code Generation

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.

One of the intermediate code representations is called 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). Each operand can act like a register. The output of the intermediate code generator can consist of the following three-address code sequence.

```
t1 = inttofloat(60)
t2 = id3 * t1                                    (1.3)
t3 = id2 + t2
id1 = t3
```

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. 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.

1.2.5. 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 inttofloat 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)
```

1.2.6. 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.

```
LDF     R2,   id3
MULF    R2,   R2,   #60.0
LDF     R1,   id2                               (1.5)
ADDF    R1,   R1,   R2
STF     id1,  R1
```

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).

1.2.7. 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 an identifier, its 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.

When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table. However, the attributes of an identifier cannot normally determined during lexical analysis.

The remaining phases enter information about identifiers into the symbol table and then use this information in various ways. For example, when doing semantic analysis and intermediate code generation, we need to know what the types of identifiers are, so we can check that the source program uses them in valid ways, and so that we can generate the proper operations on them. The code generator typically enters and uses detailed information about the storage assigned to identifiers.

1.2.8. Error Detection and Reporting

Each phase can encounter errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected. A compiler that stops when it finds the first error is not as helpful as it could be.

The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler. The lexical phase ern detect errors where the characters remaining in the input do not form any token of the language. Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase. During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved, e.g, if we try to add two identifiers, one of which is the name of an array, and the other the name of a procedure.

1.3. The Cousins of Compiler

As we saw in Fig. 1.2, the input to a compiler may be produced by one or more preprocessors, and further processing of the compiler's output may be needed before running machine code is obtained. In this section, we discuss the context in which a compiler typically operates.

Preprocessors

Preprocessors produce input to compilers. They may perform the following functions:

i. Macro processing. A preprocessor may allow a user to define macros that are short hands for longer constructs.
ii. File inclusion. A preprocessor may include header files into the program text.
iii. "Rational" preprocessor. These processors augment older languages with more modern flow-of-control and data-structuring facilities. For example, such a preprocessor might provide the user with built-in macros for constructs like while-statements or if-statements, where none exist in the programming language itself.
iv. Language extensions. These processors attempt to add capabilities to the language by what amounts to built-in macros.

Assemblers

Some compilers produce assembly code, as in (1.5), that is passed to an assembler for further processing. Other compilers perform the job of the assembler, producing relocatable machine code that can be passed directly to the loader/link-editor.

Assembly code is a mnemonic version of machine code, and in which names are used instead of binary codes for operations, names are also given to memory addresses. A typical sequence of assembly instruction might be

```
MOV a, R1
ADD #2, R1                                    (1.6)
MOV R1, b
```

This code moves the contents of the address *a* into register 1, then adds the constant 2 to it, treating the contents of register 1 as a fixed-point number, and finally stores the result in the location named by *b*. Thus, it computes *b:=a+2*.

Loaders and Link-Editors

Usually, a program called a *loader* performs the two functions of loading and link-editing. The process of loading consists of taking relocatable machine code, altering the relocatable addresses, and placing the altered instructions and data in memory at the proper locations.

The link-editor allows us to make a single program from several files of relocatable machine code. These files may have been the result of several different compilations, and one or more may be library files of routines provided by the system and available to any program that needs them.

1.4. The Grouping of Phases into Passes

The discussion of phases in Section 1.2 deals with the logical organization of a compiler. In an implementation, activities from more than one phases are grouped together.

Front and Back ends

Often, the phases are collected into a front end and a back end. The front end consists of those phases, or parts of phases, that depend primarily on the source language and are largely independent of the target machine. These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis, and the generation of intermediate code. A certain amount of code optimization can be done by the front end as well. The front end also includes the error handling that goes along with each of these phases.

The back end includes those portions of the compiler that depend on the target machine, and generally, these portions do not depend on the source language, just the intermediate language. In the back end, we find aspects of the code optimization phase, and we find code generation, along with the necessary error handling and symbol-table operations.

Passes

Several phases of compilation are usually implemented in a single pass consisting of reading an input file and writing an output file. In practice, there is great variation in the way the phases of a compiler are grouped into passes, so we prefer to organize our discussion of compiling around phases rather than passes.

As we have mentioned, it is common for several phases to be grouped into one pass. For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass. Code optimization may be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine

1.5. 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. *Scanner generators* that produce lexical analyzers from a regular expression description of the tokens of a language.
2. *Parser generators* that automatically produce syntax analyzers from a grammatical description of a programming 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 into the machine language for a target machine.
5. *Data-flow analysis engines* that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow 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**.**