Compiler Theory

Chapter 1
Introduction

Outlines

1. Compilers and Interpreters

- Compilation
  - Translation of a program written in a source language into a semantically equivalent program written in a target language
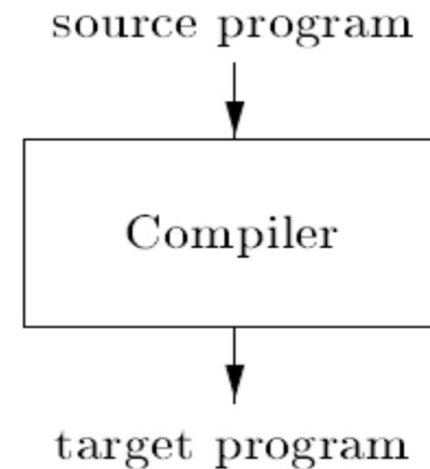
source program

↓

Compiler

↓

target program

Figure 1.1: A compiler

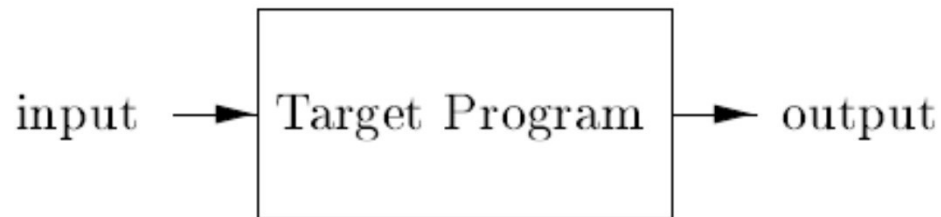input → Target Program → output

Figure 1.2: Running the target program

Compilers and Interpreters (cont)

- Interpretation
  - Performing the operations implied by the source program

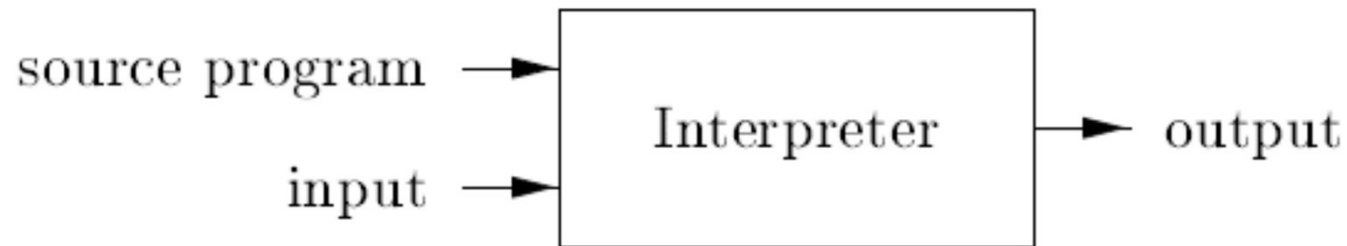
Figure 1.3: An interpreter

Compilers and Interpreters (cont)
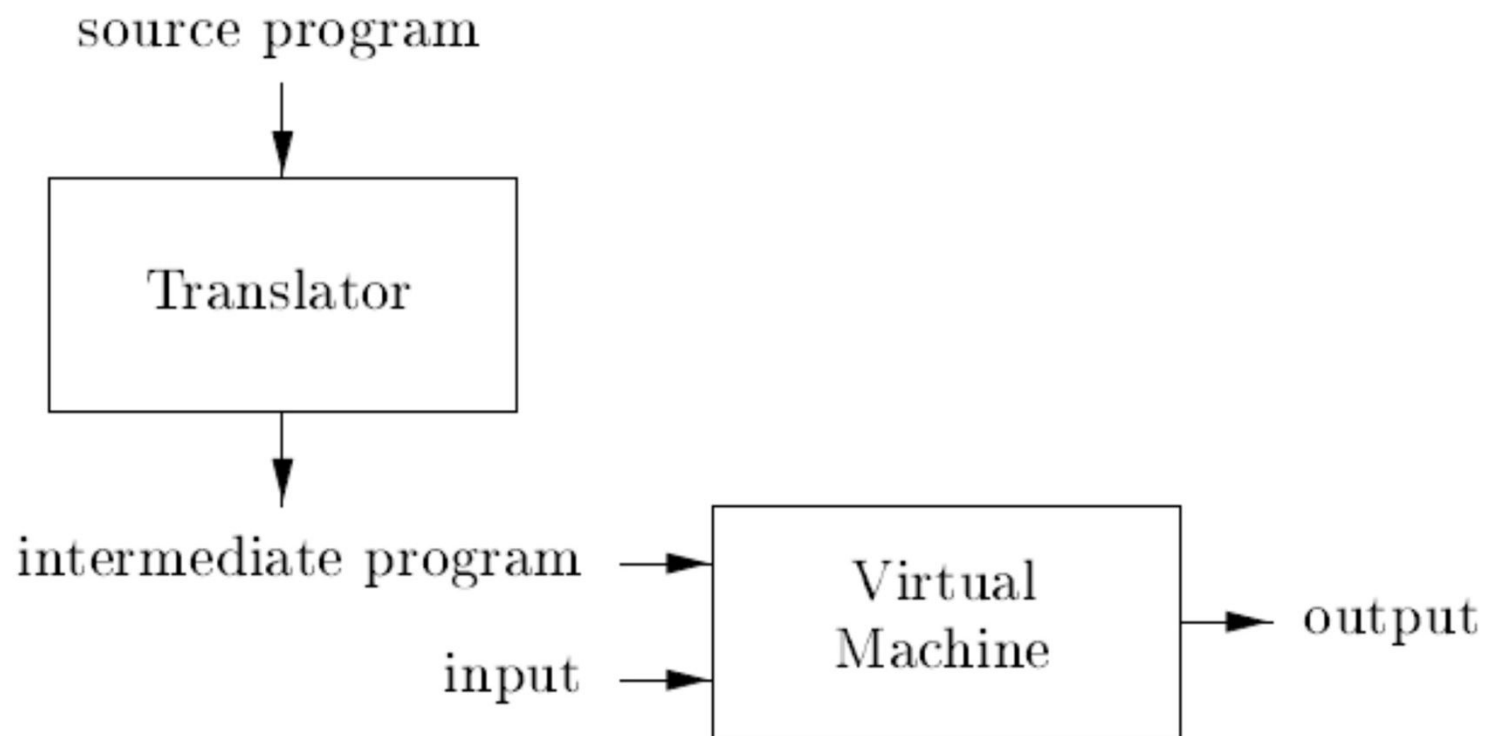
Figure 1.4: A hybrid compiler

Language Preprocessing System

Source Program

Preprocessor

Modified Source Program

Compiler

Target Assembly
Program

Assembler

Relocatable Object Code

Linker ← Libraries and
Relocatable Object Files

Absolute Machine Code
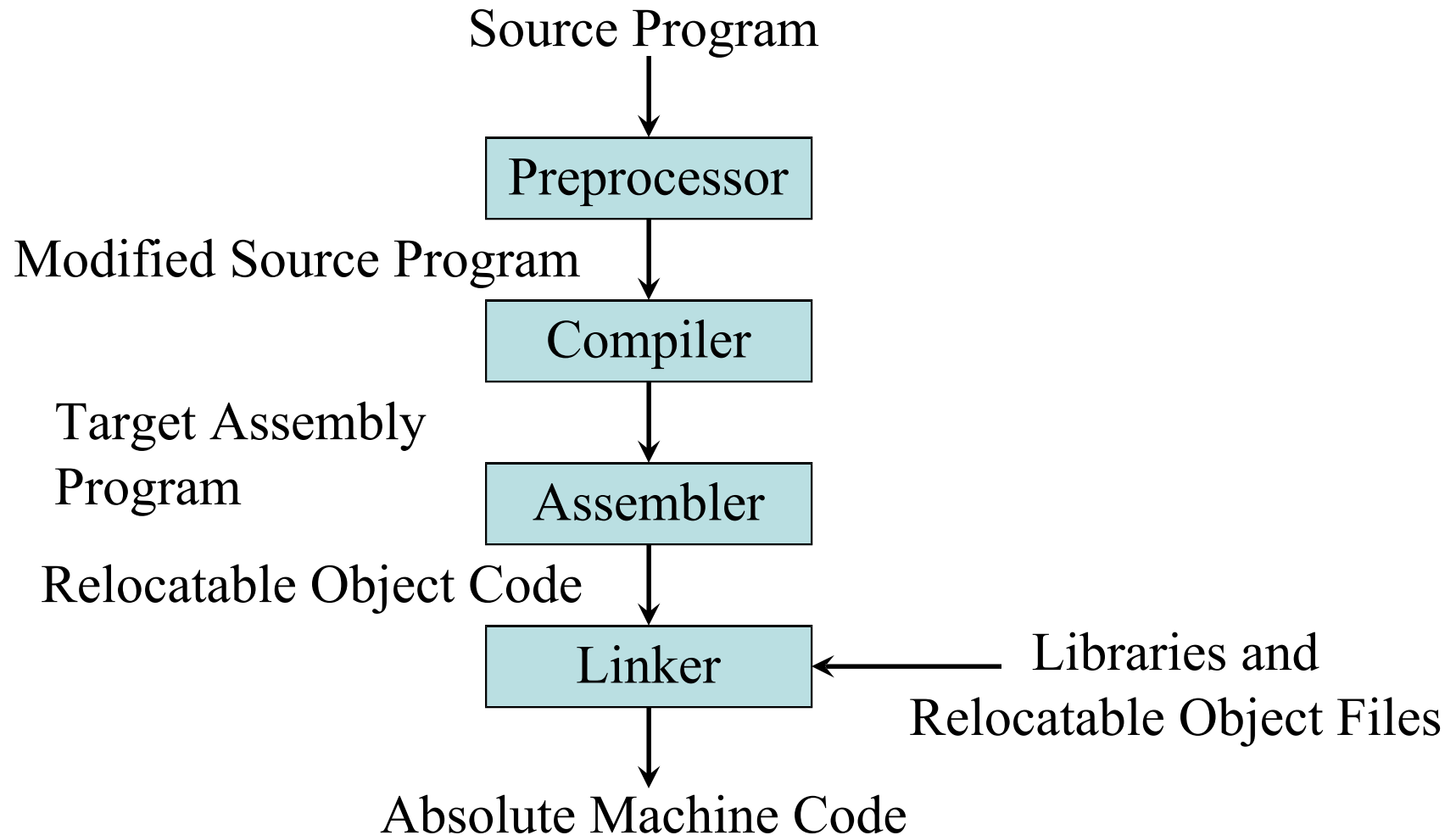
2. The Structure of a Compiler

- Analysis Part

- Synthesis Part

Analysis and Synthesis

- There are two parts to compilation:
  - **Analysis** 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
  - **Synthesis** constructs the desired target program from the intermediate representation and the information in the symbol table.

The Phases of a Compiler

- A compiler operates in phases, each of which transforms the source program from one representation to another.

The Phases of a Compiler

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

↓
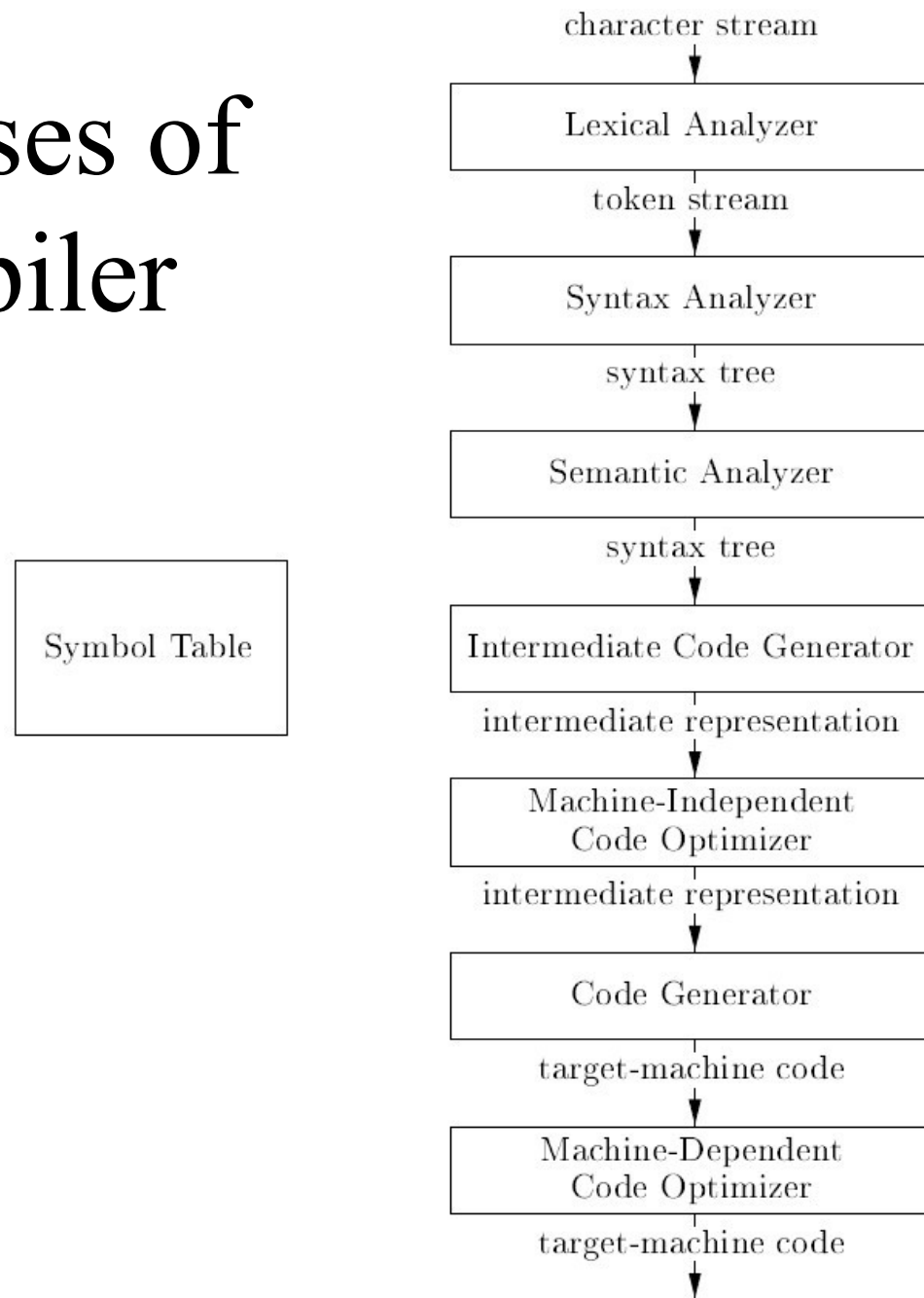
Figure 1.6: Phases of a compiler

Lexical Analysis

$$a = b + c * 60$$
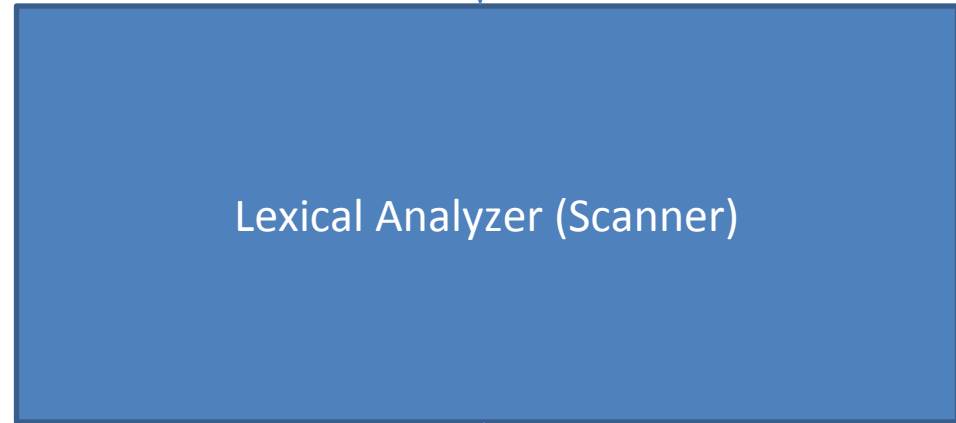
- The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences / tokens called lexemes.
- Tokens are classified as key words, identifiers, operators, etc.
- For each lexeme, the lexical analyzer produces as output a token of the form
  <token-name, attribute-value>
- Consider a = b + c * 60 (HLL Statement)
- a is a lexeme that would be mapped into a token <id, I>, where id is an abstract symbol standing for identifier and 1 points to the symbol table.
- = is a lexeme that is mapped into the token < = >
- b is a lexeme that is mapped into the token <id, 2>
- + is a lexeme that is mapped into the token <+>
- c is a lexeme that is mapped into the token <id, 3>
- * is a lexeme that is mapped into the token < * >
- 60 is a lexeme that is mapped into the token <60>

Lexical Analyzer (Scanner)

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

Symbol Table

1	a	Float
2	b	Float
3	c	Float

Syntax Analysis

- 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 = b + c * 60

Lexical Analyzer (Scanner)

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

Syntax Analyzer (Parser)

```
        =
     /     \
   id1      +
         /     \
       id2      *
             /     \
           id3      60
```
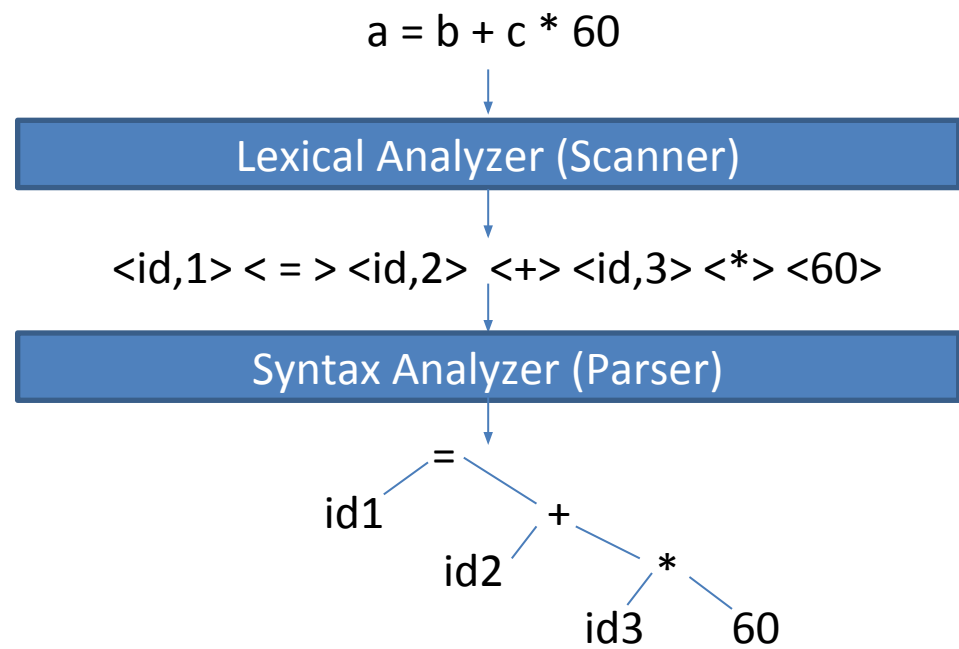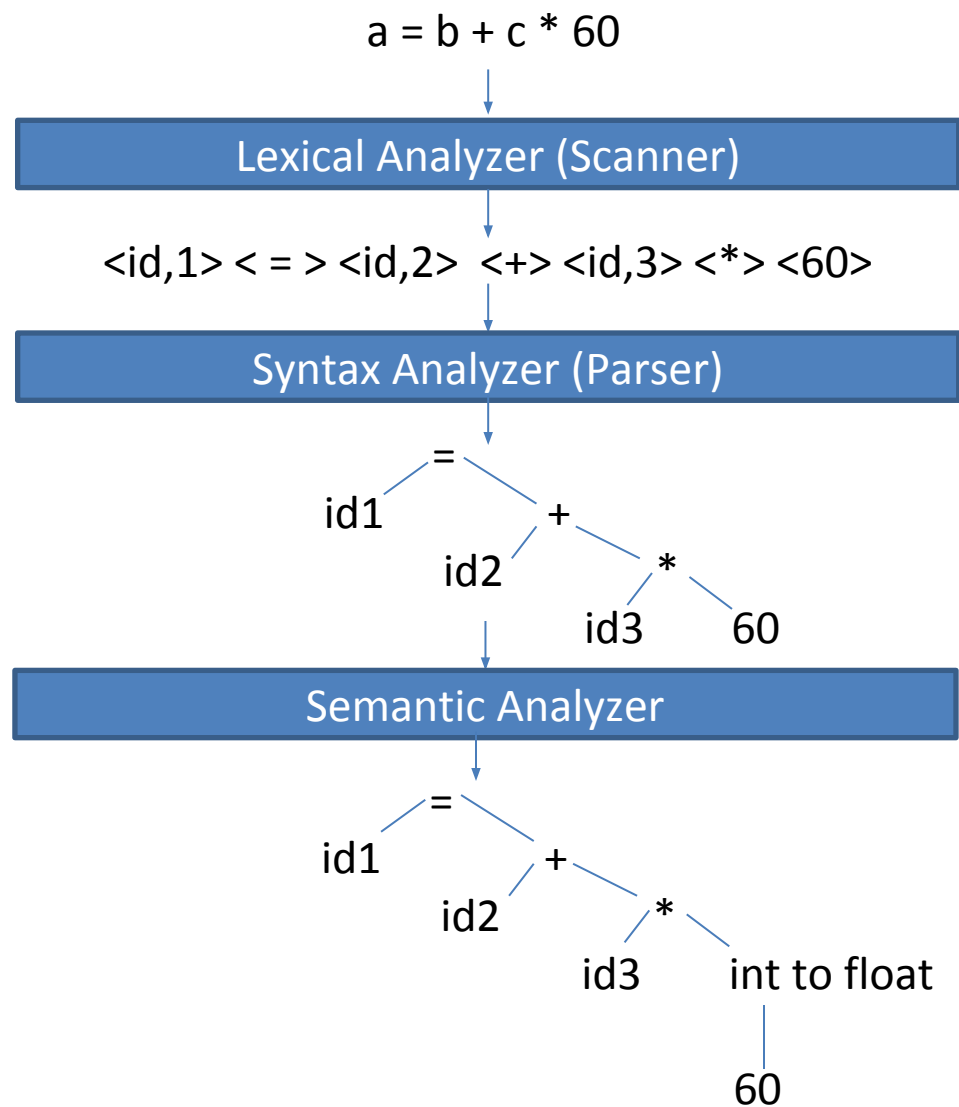
Symbol Table

1	a	Float
2	b	Float
3	c	Float

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 the symbol table.
- 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.

a = b + c * 60

| Lexical Analyzer (Scanner) |

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

| Syntax Analyzer (Parser) |

```
        =
    id1    +
        id2    *
            id3   60
```

| Semantic Analyzer |

```
        =
    id1    +
        id2    *
            id3   int to float
                        60
```
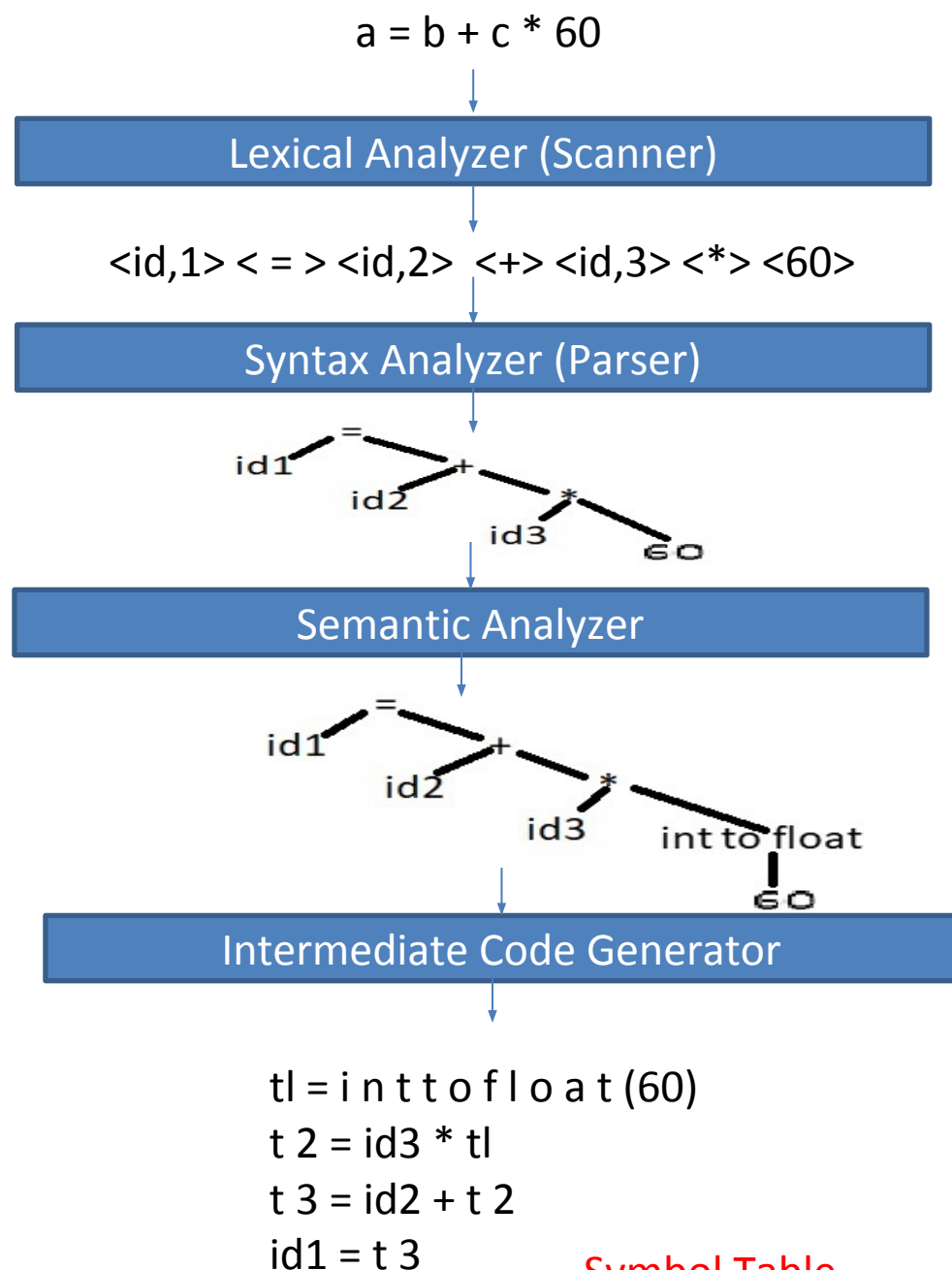
Symbol Table

1	a	Float
2	b	Float
3	c	Float

Intermediate Code Generator

- 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
- 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.
- we consider an intermediate form called three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction.

a = b + c * 60

Lexical Analyzer (Scanner)

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

Syntax Analyzer (Parser)

Semantic Analyzer

Intermediate Code Generator

tl = i n t t o f l o a t (60)
t 2 = id3 * tl
t 3 = id2 + t 2
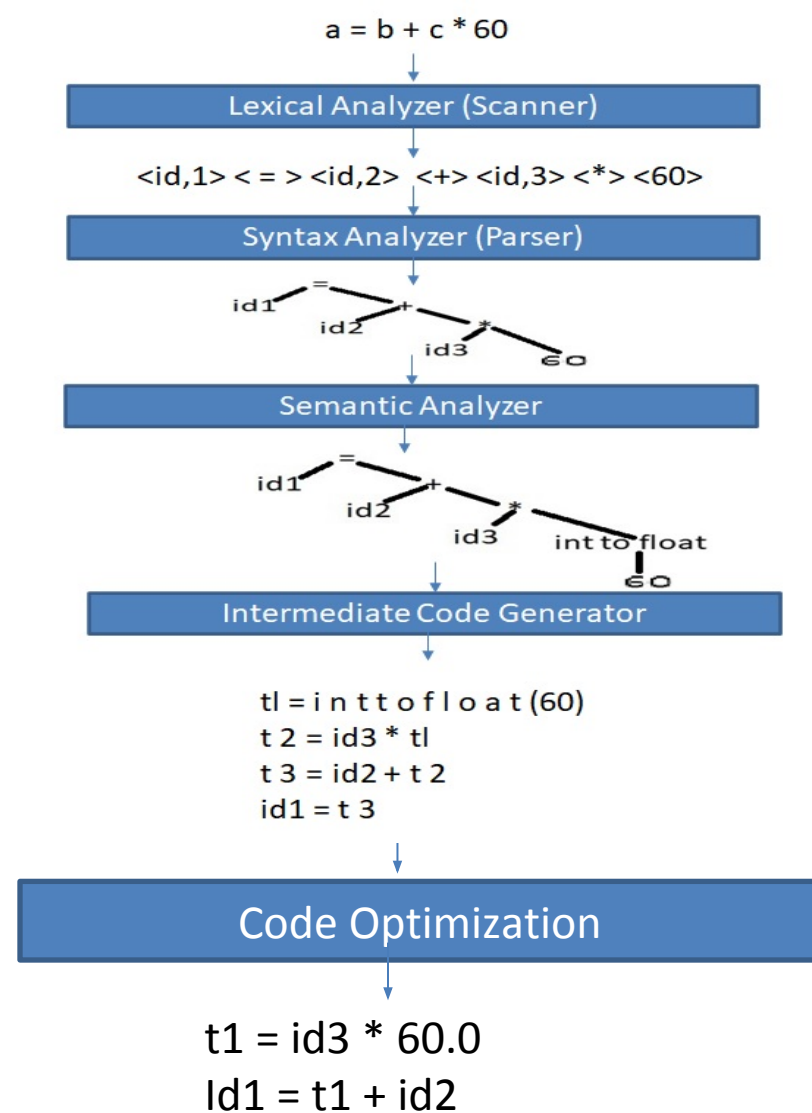id1 = t 3

Symbol Table

1	a	Float
2	b	Float
3	c	Float

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..
- Automatically modify programs so that they
  – Run faster
  – Use less memory
  – In general, conserve some resource
  – Preserve correctness

a = b + c * 60

Lexical Analyzer (Scanner)

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

Syntax Analyzer (Parser)

```
      =
id1
      id2   +
            id3   *
                  60
```

Semantic Analyzer

```
      =
id1
      id2   +
            id3   *
                  int to float
                        60
```

Intermediate Code Generator

```
tl = i n t t o f l o a t (60)
t 2 = id3 * tl
t 3 = id2 + t 2
id1 = t 3
```

Code Optimization

t1 = id3 * 60.0
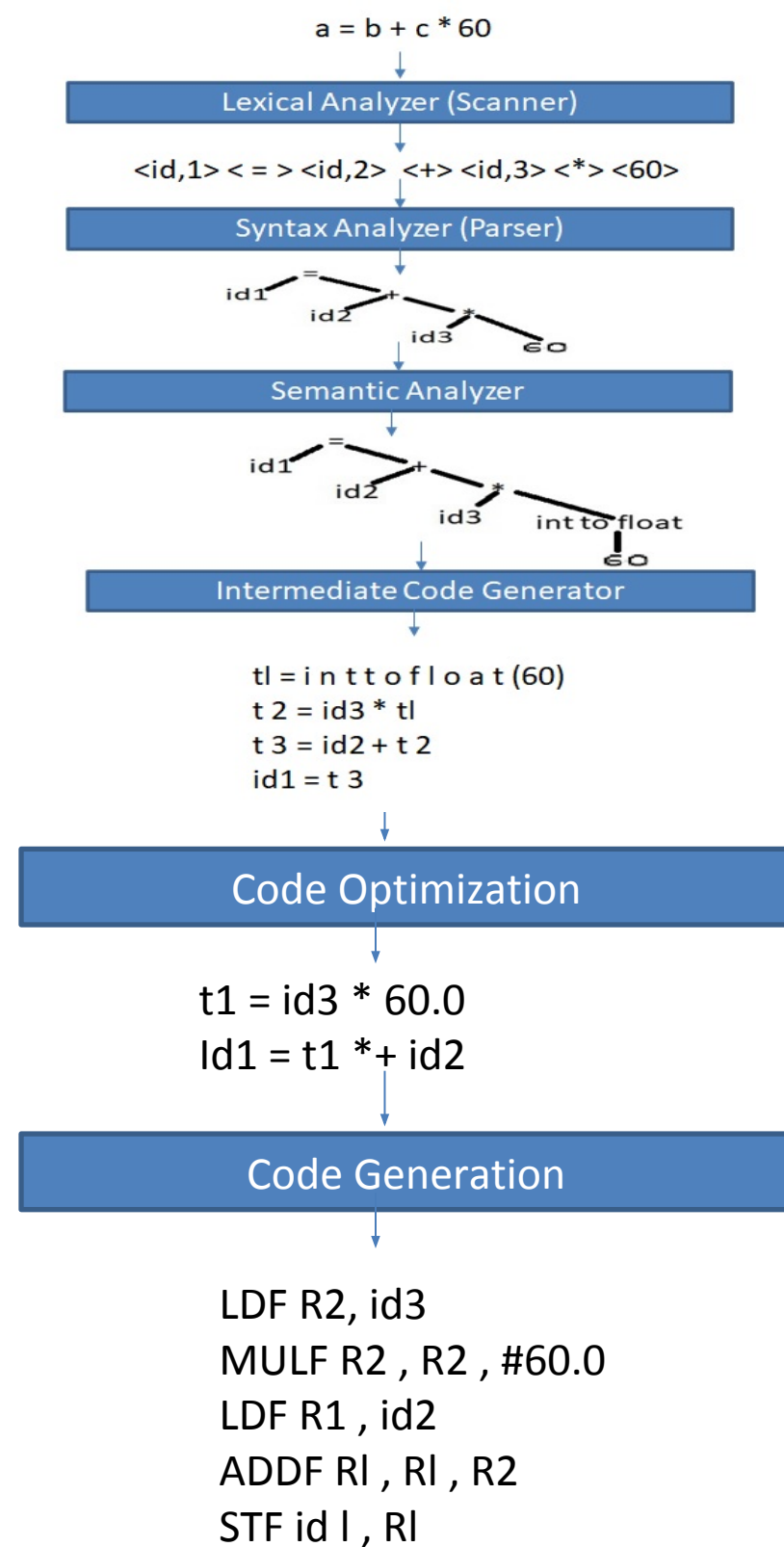Id1 = t1 + id2

Symbol Table

1	a	Float
2	b	Float
3	c	Float

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.

$$a = b + c * 60$$

Lexical Analyzer (Scanner)

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

Syntax Analyzer (Parser)

```
        =
id1
    id2    +
        id3    *
                60
```

Semantic Analyzer

```
        =
id1
    id2    +
        id3    *
            int to float
                60
```

Intermediate Code Generator

```
tl = i n t t o f l o a t (60)
t 2 = id3 * tl
t 3 = id2 + t 2
id1 = t 3
```

Code Optimization

$$t1 = id3 * 60.0$$
$$Id1 = t1 *+ id2$$

Code Generation

```
LDF R2, id3
MULF R2 , R2 , #60.0
LDF R1 , id2
ADDF Rl , Rl , R2
STF id l , Rl
```

1	a	Float
2	b	Float
3	c	Float

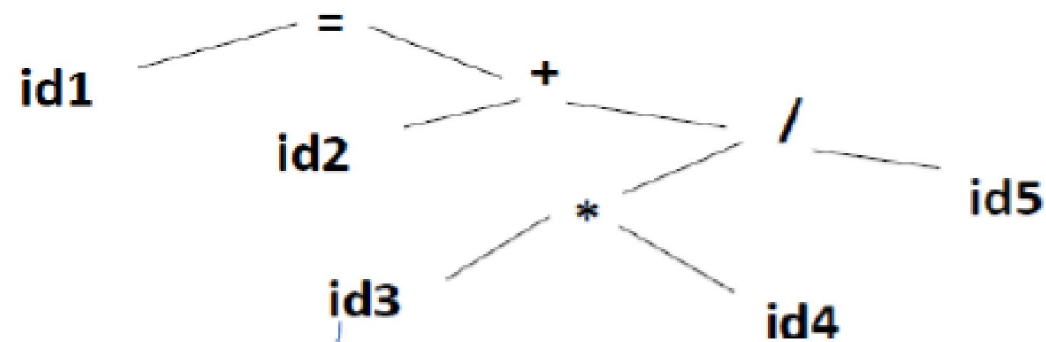Show the output of different phases of compiler for the following statement.

$$x = a + b * c /d.$$

Write the symbol table entries. Assume that a,b,c and d are floating point variables.
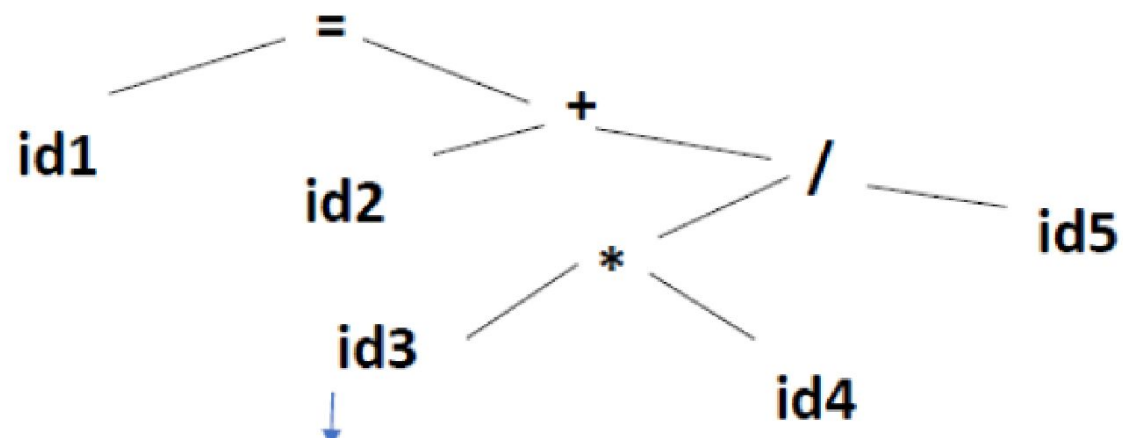
x = a + b * c / d

Lexical Analyzer

<id,1> <= > <id,2><+><id,3><*><id,4></><id,5>

Syntax Analyzer

```
              =
   id1                  +
            id2                    /
                       *                id5
                  id3       id4
```

Symbol Table

1	x		Float
2	a		Float
3	b		Float
4	c		Float
5	d		Float

Semantic Analyzer

```
              =
   id1                  +
            id2                    /
                       *                id5
                  id3       id4
```

```
                    │
                    ▼
┌─────────────────────────────────────┐
│   Intermediate Code Genarator        │
└─────────────────────────────────────┘
                    │
                    ▼
┌─────────────────────────────────────┐
│  t1 = id4 * id3                      │
│                                      │
│  t2 = t1 / id5                       │
│                                      │
│  t3 = t2 + id2                       │
│                                      │
│  id1 =t3                             │
└─────────────────────────────────────┘
                    │
                    ▼
┌─────────────────────────────────────┐
│         Code Optimization            │
└─────────────────────────────────────┘
                    │
                    ▼
┌─────────────────────────────────────┐
│  t1 = id4 * id3                      │
│                                      │
│  t2 = t1 / id5                       │
│                                      │
│  id1 = t2 + id2                      │
└─────────────────────────────────────┘
                    │
                    ▼
┌─────────────────────────────────────┐
│          Code Genarator              │
└─────────────────────────────────────┘
                    │
                    ▼
┌─────────────────────────────────────┐
│   LDF R1,id4                         │
│   LDF R2, id3                        │
│   MulF R3,R1,R2                      │
│   LDF R4,id5                         │
│   Div R4, R3, R4                     │
│   LDF R5, id2                        │
│   ADD R5, R4, R5                     │
│   STF id1, R5                        │
└─────────────────────────────────────┘
```

$$x = a + b * c / d$$

Lexical Analyzer

`<id,1> <= > <id,2><+><id,3><*><id,4></><id,5>`

Syntax Analyzer

```
        =
id1  ────┴────  +
        id2  ──┴──  /
              *      id5
          id3 ─┴─ id4
```

Semantic Analyzer

```
        =
id1  ────┴────  +
        id2  ──┴──  /
              *      id5
          id3 ─┴─ id4
```

Intermediate Code Genarator

```
t1 = id4 * id3
t2 = t1 / id5
t3 = t2 + id2
id1 = t3
```

Code Optimization

```
t1 = id4 * id3
t2 = t1 / id5
id1 = t2 + id2
```

Code Genarator

```
LDF R1,id4
LDF R2, id3
MulF R3,R1,R2
LDF R4,id5
Div R4, R3, R4
LDF R5, id2
ADD R5, R4, R5
STF id1, R5
```

Symbol Table

1	x	Float
2	a	Float
3	b	Float
4	c	Float
5	d	Float

Show the output of different phases of compiler for the following statement.

$$x = b * 10 + c / 20.$$

Write the symbol table entries.

Data Structures

Compilers are required to store all the identifiers defined in the source program.

Identifiers may be program variable or function name.

Program variables are to be checked for their place of declaration and their usage in the program, in order to avoid multiple definitions.

In C all variables must be declared before they are used.

It is also necessary to check for the variable being assigned values of same data type as declared.

Similarly functions to be defined before they are called. It is also necessary to check for the number of arguments and their data types and also the return values of the function are same as defined.

Hence compiler uses a data structure called Symbol table to store these information. Apart from identifiers source program also has constants whose values do not change during the execution of the program like const A=10. Source program may also have statements which are used to display constant strings.

During compilation some error like missing parenthesis, invalid function name or invalid operator may be found. In such cases, the location of errors like line no or function name or file name has to be displayed to the user along with the error message hence they should also be stored in symbol table as scope information. While displaying error message it is required to associate the message with the location of error. In order to store this location of error, error tables are used. This error table is also used by error handler routine. The error handler routine may performs some error corrections like adding missing parenthesis etc. This is mainly done in order to continue the compilation and generate code.

Symbol table

Symbol table is used to store the description of identifiers used in the source program.
Contents of symbol table are name of the identifier, data type, scope and value.
If the identifier is the name of the function along with name it also stores list of arguments their data type and return type of function.
Symbol table helps in avoiding multiple declarations of variables and keeping track of scope information.
It also has a pointer which points to the memory location where the identifier is stored.

Consider the following C code.

```
int main()
{
    int x=10;
    float y, z;
    Printf("enter value for z\n");
    scanf("%f",&z);
    y = (float) x +z;
    printf(" the value of y=%f\n",y);
}
```

Identifier name	Data type	Value	....
X	int	10	
Y	float		
Z	float		

Tab 1.1 Sample Symbol table

Literal table

Literal table stores the constant values and strings used in the source program. The main function of the literal table is to conserve memory by reusing the constant and the string.

In the following example there is constant value stored in 'a', the value of 'a' do not change during the execution of the program. This can be stored in the literal table as Literal1.

In same way printf statement displays string "Hello", this is also stored literal table as Literal 2. Tab 1.2 shows the sample literal table for the below example.

```
int main()

    {

    const a=7;

        printf("Hello");

    }
```

Literal no	Literal name	Value	....
Literal 1	A	10	
Literal 2	String	Hello	

Tab 1.2 Sample Literal table

Error table

Whenever error is found in any phase of compiler, it is recorded in error table.

Based on the type of error, error messages are displayed and if required some error corrections are done in order to continue the compilation.

Some of the common errors that can be detected in lexical analysis are incomplete token so the error message would be "misspelled identifier". Similarly errors during syntax analysis are "mismatched parenthesis".

It is said that most of the syntax error are detected during syntax analysis. Error table stores the type of the error like lexical error, syntax error etc, along with the string and the location of error.

Consider the following C code

Line no 1:   main( )

Line no 2:   {

Line no 3:   in abc;

Line no 4:   abc=10+(10*3;

Line no 5:   }

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 and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument 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.

The Grouping of Phases

- Compiler *front* and *back ends*:
  - Front end: *analysis* (*machine independent*)
  - Back end: *synthesis* (*machine dependent*)
- Compiler *passes:*
  - It deals with the logical organization of a compiler.
  - In an implementation, activities from several phases may be grouped together into a pass that reads an input file and writes an output file.
    - 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 might be an optional pass.
    - Then there could be a back-end pass consisting of code generation for a particular target machine.

Compiler-Construction Tools

- The compiler writer can 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.

Compiler-Construction Tools

Some commonly used compiler-construction tools include :

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

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

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

- 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.

Compiler-Construction Tools

- **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.

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

The Move to Higher-level Languages

- The first step towards more people-friendly programming languages was the development of mnemonic assembly languages in the early 1950's.

- Initially, the instructions in an assembly language were just mnemonic representations of machine instructions.

- A major step towards higher-level languages was made in the latter half of the 1950's with the development of Fortran for scientific computation, Cobol for business data processing, and Lisp for symbolic computation

- The philosophy behind these languages was to create higher-level notations with which programmers could more easily write numerical computations, business applications, and symbolic programs.

- These languages were so successful that they are still in use today.

- Today, there are thousands of programming languages.

The Move to Higher-level Languages

- Programming Languages can be classified in a variety of ways.

- One classification is by generation.

  - First-generation languages are the machine languages,

  - Second-generation the assembly languages,

  - Third-generation the higher-level languages like Fortran, Cobol, Lisp, C,
    C++, C#, and Java.

  - Fourth-generation languages are languages designed for specific applications like NOMAD for report generation, SQL for database queries, and Postscript for text formatting.

  - Fifth-generation language has been applied to logic- and constraint-based languages like Prolog and OPS5.

The Move to Higher-level Languages

- Another classification of languages uses the term imperative for languages in which a program specifies how a computation is to be done and declarative for languages in which a program specifies what computation is to be done.

- Languages such as C, C++, C#, and Java are imperative languages. In imperative languages there is a notion of program state and statements that change the state.

- Functional languages such as ML and Haskell and constraint logic languages such as Prolog are often considered to be declarative languages.

- The term von Neumann language is applied to programming languages whose computational model is based on the von Neumann computer architecture.

- Many of today's languages, such as Fortran and C are von Neumann languages.

The Move to Higher-level Languages

- An object-oriented language is one that supports object-oriented programming, a programming style in which a program consists of a collection of objects that interact with one another.
    - Simula 67 and Smalltalk are the earliest major object-oriented languages.
    - Languages such as C++, C#, Java, and Ruby are more recent object-oriented languages.

- Scripting languages are interpreted languages with high-level operators designed for "gluing together" computations.
    - These computations were original called "scripts."
    - Awk, JavaScript, Perl, PHP, Python, Ruby, and Tcl are popular examples of scripting languages.
    - Programs written in scripting languages are often much shorter than equivalent programs written in languages like C.

Impacts on Compilers

- Since the design of programming languages and compilers are intimately related, the advances in programming languages placed new demands on compiler writers.

- They had to devise algorithms and representations to translate and support the new language features.

-

- The compiler writers have to track new language features and design translation algorithm.

- Compilers can help promote the use of high-level languages by minimizing the execution overhead of the programs written in these languages.

- Compilers are also critical in making high-performance computer architectures effective on users' applications.

- The performance of a computer system is so dependent on compiler technology that compilers are used as a tool in evaluating architectural concepts before a computer is built.

Impacts on Compilers

- Compiler writing is challenging.

- A compiler by itself is a large program.

- A compiler must translate correctly the potentially infinite set of programs that could be written in the source language.

- The problem of generating the optimal target code from a source program is undecidable in general.

- So compiler writers must evaluate tradeoffs about what problems to tackle and what heuristics to use to approach the problem of generating efficient code.