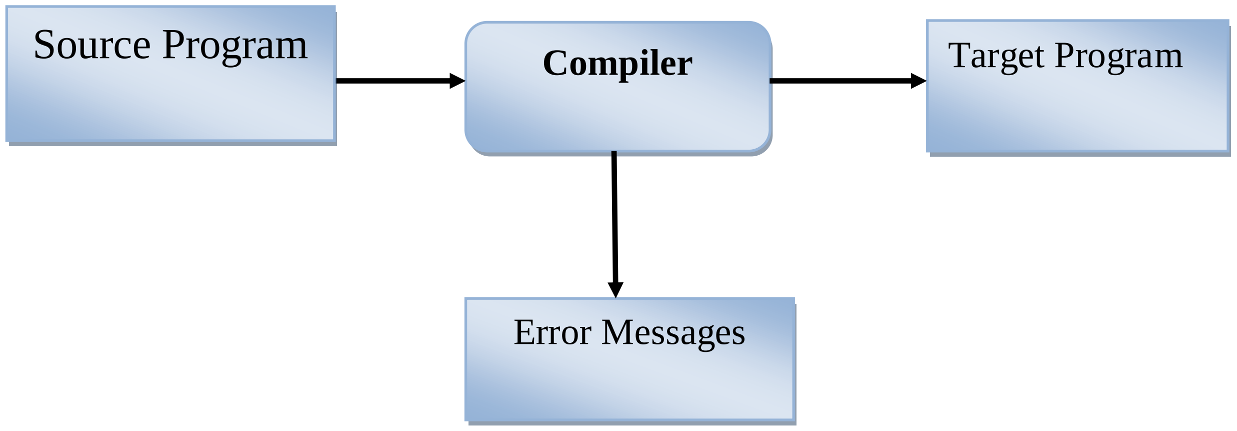
MODULE I

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),ie it is a software that converts the source code to the object code. In other words, we can say that it converts the high-level language to machine/binary language. Moreover, it is necessary to perform this step to make the program executable. This is because the computer understands only binary language. Some compilers convert the high-level language to an assembly language as an intermediate step, whereas some others convert it directly to machine code. This process of converting the source code into machine code is called **compilation.**



Compilers are sometimes classified as single pass, multi-pass, load-and-go, debugging, or optimizing, depending on how they have been constructed or on what function they are supposed to perform.

Analysis of a Source Program

We can analyze a source code in three main steps. These steps are further divided into different phases. The three steps are:

1. Linear Analysis

Here, it reads the character of the code from left to right. The characters having a collective meaning are formed. We call these groups tokens. In a compiler linear analysis is called lexical analysis or scanning. The lexical analysis phase reads the characters in the source program and grouped into tokens that are sequence of characters having a collective meaning.

EXAMPLE

position : = initial + rate * 60

This can be grouped into the following tokens;

1. The identifier position.

2. The assignment symbol : =

3. The identifier initial

4. The plus sign

5. The identifier rate

6. The multiplication sign

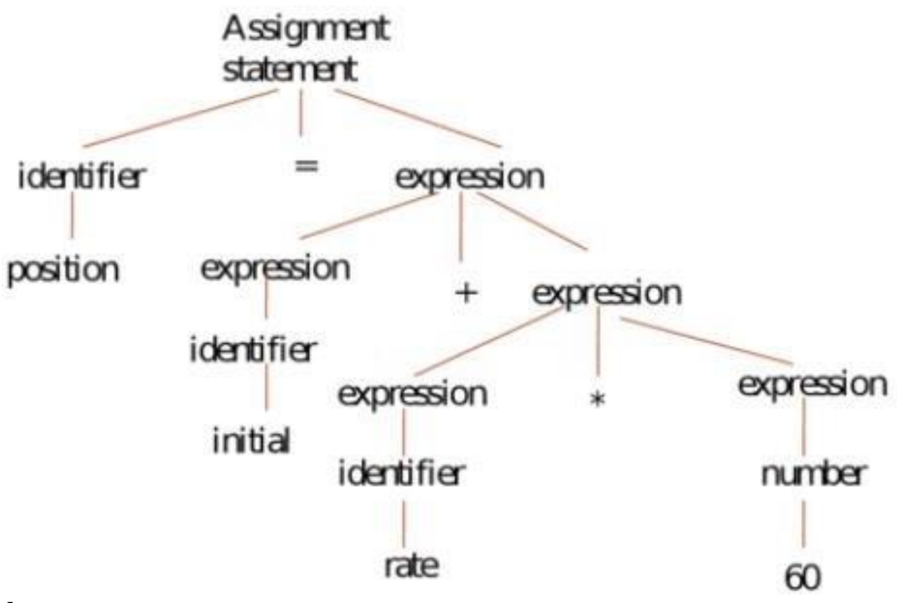
7. The number 60

Blanks separating characters of these tokens are normally eliminated during lexical analysis.

2. Syntax Analysis/Hierarchical Analysis

It involves grouping the tokens of the source program into grammatical phrases that are used by the complier to synthesize output. They are represented using a syntax tree.

A syntax tree is the tree generated as a result of syntax analysis in which the interior nodes are the operators and the exterior nodes are the operands. This analysis shows an error when the syntax is incorrect.

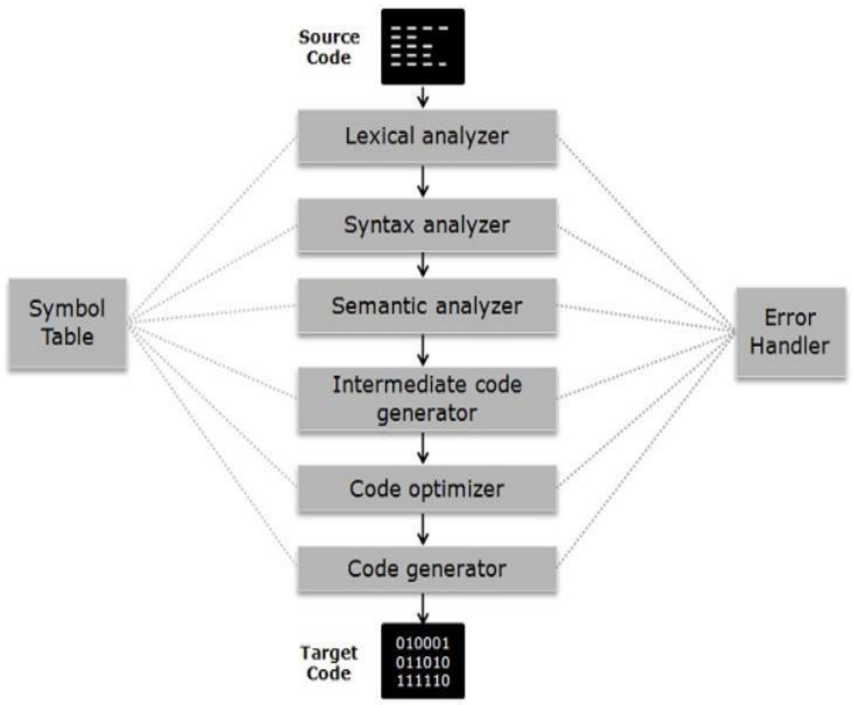


3. Semantic Analysis

In this step, we check if the components of the source code are appropriate in meaning. This phase checks the source program for semantic errors and gathers type information for subsequent code generation phase. An important component of semantic analysis is type checking. Here the compiler checks that each operator has operands that are permitted by thesource language specification.

Phases/Structure of Compiler

The compilation process takes place in several phases. Moreover, for each step, the output of one step acts as the input for the next step. The phases/structure of the compilation process are as follows



1. Lexical Analyzer

In a compiler linear analysis is called lexical analysis or scanning. Lexical analyzer scans the characters of source code from left to right. Hence, the name scanner also.

The lexical analysis phase reads the characters in the source program and grouped into tokens that are sequence of characters having a collective meaning. 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

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 forposition. 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.
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 <60>

Blanks separating the lexemes would be discarded by the lexical analyzer. The representation of the assignment statement position = initial + rate * 60 after lexical analysis as the sequence of tokens as:

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

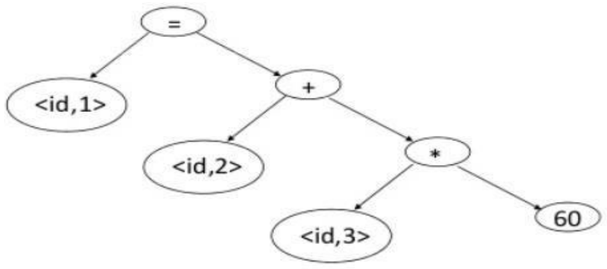
/*

- It takes the high-level language source code as the input.
- It scans the characters of source code from left to right. Hence, the name scanner also.
- It groups the characters into lexemes. Lexemes are a group of characters which has some meaning.
- Each lexeme corresponds to form a token.
- It removes white spaces and comments.
- It checks and removes the lexical errors.*/

2. Syntax Analyzer

- ‘Parser’ is the other name for the syntax analyzer.
- The output of the lexical analyzer is its input.
- It checks for syntax errors in the source code.
- It does this by constructing a parse tree of all the tokens.
- For the syntax to be correct, the parse tree should be according to the rules of source code grammar.
- The grammar for such codes is context-free grammar.

The syntax tree for above token stream is:



The tree has an interior node labeled with (id, 3) as its left child and the integer 60 asits 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 additioninto the location for the identifier position.

3. Semantic Analyzer

- It verifies the parse tree of the syntax analyzer.
- It checks the validity of the code in terms of programming language. Like, compatibility of data types, declaration, and initialization of variables, etc.
- It also produces a verified parse tree. Furthermore, we also call this tree an annotated parse tree.
- It also performs flow checking, type checking, etc.

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.

Some sort of type conversion is also done by the semantic analyzer.

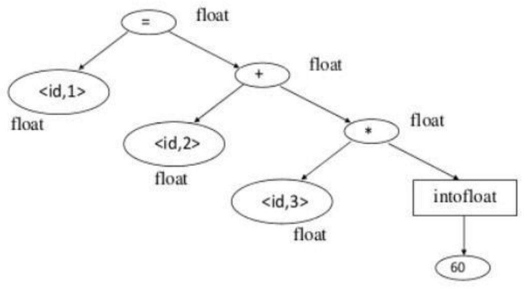
For example, if the operator is applied to a floating point number and an integer, the compiler may convert the integer into a floating point number.

In our example, suppose that position, initial, and rate have been declared to be floating-point numbers, and that the lexeme 60 by itself forms an integer.

The semantic analyzer 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 the following figure, 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 afloating-point number.



4. Intermediate Code Generator (ICG)

- It generates an intermediate code.
- This code is neither in high-level language nor in machine language. It is in an intermediate form.
- It is converted to machine language but, the last two phases are platform dependent.
- The intermediate code is the same for all the compilers. Further, we generate the machine code according to the platform.
- An example of an intermediate code is three address code.

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 simple and easy to produce
- It should be easy to translate into the target machine.

In our example, the intermediate representation used is three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction.

t1 = inttofloat(60)

t2 = id3 * t1

```
t3 = id2 + t2

id1 = t3
```

5. Code Optimizer

- It optimizes the intermediate code.
- Its function is to convert the code so that it executes faster using fewer resources (CPU, memory).
- It removes any useless lines of code and rearranges the code.
- The meaning of the source code remains the same.
- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
- The objectives for performing optimization are: faster execution, shorter code, or target code that consumes less power.
- In our example, the optimized code is:

```
t1 = id3 * 60.0

id1 = id2 + t1
```

6. Target Code Generator

- Finally, it converts the optimized intermediate code into the machine code.
- This is the final stage of the compilation.
- The machine code which is produced is relocatable.

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 foreach 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.

If the target language is assembly language, this phase generates the assembly code as its output.

In our example, the code generated is:

```
LDF    R2,   id3
MULF  R2, #60.0
LDF    R1,   id2
ADDF   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 above code 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 Rl and the fourth adds to it the value previously computed in register R2.

Finally, the value in register Rl is stored into the address of idl , so the code correctly implements the assignment statement

position = initial + rate * 60.

Symbol Table

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.

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.

GROUPING OF PHASES

The process of compilation is split up into following phases:

- **Analysis Phase(Front end/machine independent/language dependent)**
- **Synthesis phase(Back end/machine dependent/language independent)**

Analysis Phase

Analysis Phase performs 4 actions namely:

a. **Lexical analysis**

b. **Syntax Analysis**

c. **Semantic analysis**

d. **Intermediate Code Generation**

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 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.

Synthesis Phase

Synthesis Phase performs 2 actions namely:

a. **Code Optimization**
b. **Code Generation**

Compiler writing (Construction) tools

Compiler writers use software development tools and more specialized tools for implementing various phases of a compiler. Some commonly used compiler construction tools include the following.

- ➢ **Parser Generators**
- ➢ **Scanner Generators**
- ➢ **Syntax-directed translation engine**
- ➢ **Automatic code generators**
- ➢ **Data-flow analysis Engines**
- ➢ **Compiler Construction toolkits**

Parser Generators.

Input : Grammatical description of a programming language

Output : Syntax analyzers.

These produce syntax analyzers, normally from input that is based on a context-free grammar.

In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler.

This phase is one of the easiest to implement.

Scanner Generators

Input : Regular expression description of the tokens of a language

Output : Lexical analyzers.

These automatically generate lexical analyzers, normally from a specificalion based on regular expressions.The basic organization of the resulting lexical analyzer is in effect a finite automaton.

Syntax-directed Translation Engines

Input : Parse tree.

Output : Intermediate code.

These produce collections of routines that walk the parse tree, generating intermediate code.

The basic idea is that one or more "translations" are associated with each node of the parse tree, and each translation is defined in terms of translations at its neighbour nodes in the tree.

Automatic Code Generators

Input : Intermediate language.

Output : Machine language.

Such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine.

The rules must include sufficient detail that we can handle the different possible access methods for data.

Data-flow Analysis Engines

Data-flow analysis engine gathers the Information that is, the values transmitted from one part of a program to each of the other parts.

Data-flow analysis is a key part of code optimization.

Cousins of Compiler

1. Preprocessor
2. Assembler
3. Loader and Link-editor

Preprocessor

A preprocessor is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers.
They may perform the following functions :

i) Macro processing
ii) File Inclusion
iii)Language extension
iv)Rational preprocessors

i) Macro processing:

A macro is a rule or pattern that specifies how a certain input sequence should be mapped to an output sequence according to a defined procedure. The mapping process that instantiates a macro into a specific output sequence is known as macro expansion.

ii)File Inclusion:

Preprocessor includes header files into the program text. When the preprocessor finds an #include directive it replaces it by the entire content of the specified file.

iii)Rational Preprocessors:

These processors change older languages with more modern flow-of-control and data-structuring facilities.

iv)Language extension:

These processors attempt to add capabilities to the language by what amounts to built-in macros.

Assembler

Assembler creates object code by translating assembly instruction mnemonics into machine code. There are two types of assemblers:

· One-pass assemblers go through the source code once and assume that all symbols will be defined before any instruction that references them.

· Two-pass assemblers create a table with all symbols and their values in the first pass, and then use the table in a second pass to generate code

Linker and Loader

A **linker** or **link editor** is a program that takes one or more objects generated by a compiler and combines them into a single executable program. Three tasks of the linker are

1. Searches the program to find library routines used by program, e.g. printf(), math routines.

2. Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references
3. Resolves references among files.

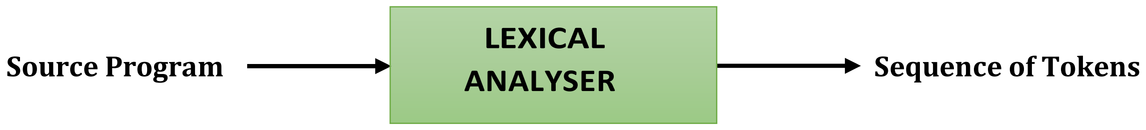
A **loader** is the part of an operating system that is responsible for loading programs in memory, one of the essential stages in the process of starting a program.

Lexical Analysis

Role of lexical analyzer

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.

The stream of tokens is sent to the parser for syntax analysis.



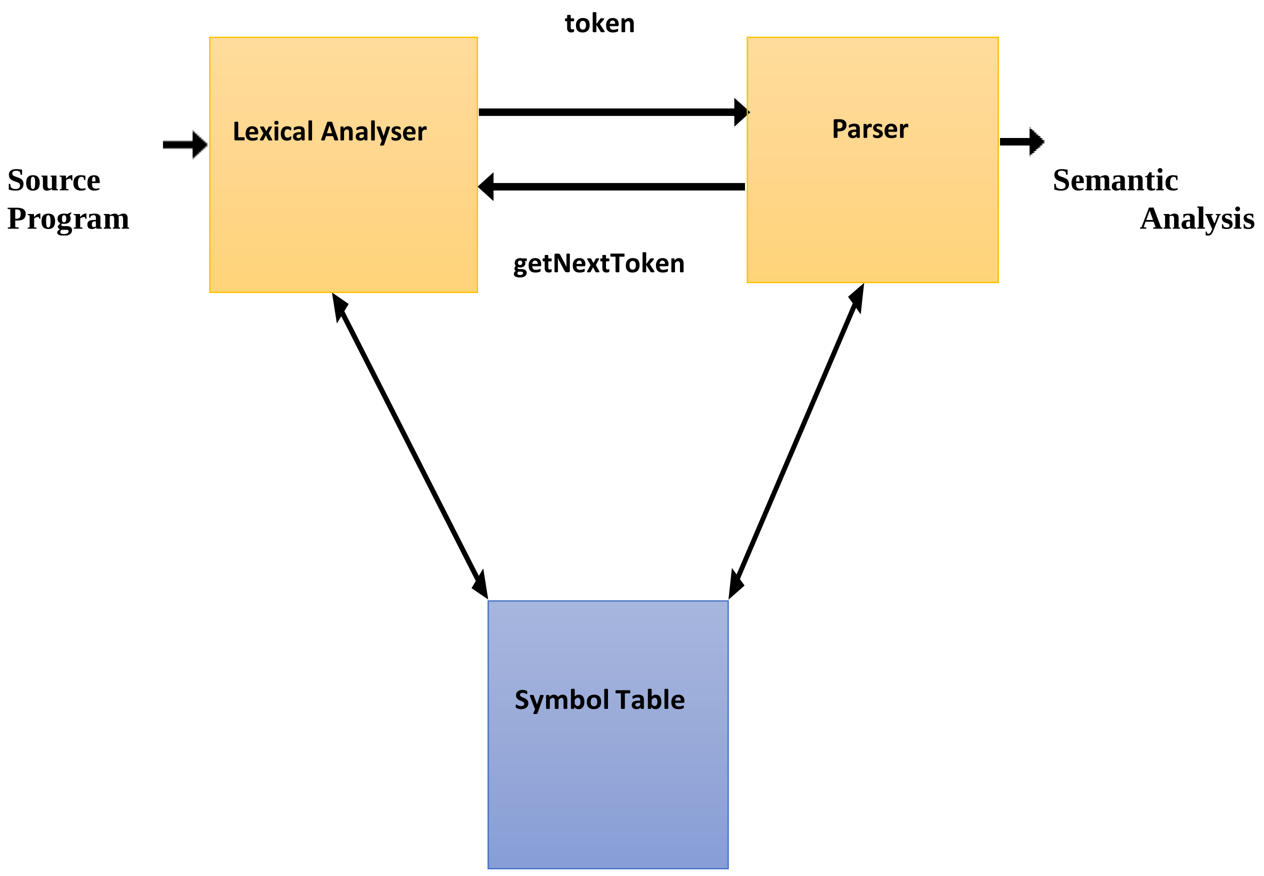
Lexical Analyzer also interacts with the symbol table.

When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.

In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

These interactions are given in following figure.

The call, suggested by the **getNextToken** command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it thenext token, which it returns to the parser.



Other tasks of Lexical Analyzer

1. Stripping out comments and whitespace (blank, newline, tab, and perhaps othercharacters that are used to separate tokens in the input).
2. Correlating error messages generated by the compiler with the source program.For instanc e, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.
3. If the source program uses a macro-pre-processor, the expansion of macros may alsobe performed by the lexical analyzer.

Advantages of having a lexical analyzer

Following are the reasons why lexical analysis is separated from syntax analysis

Simplicity Of Design

The separation of lexical analysis and syntactic analysis often allows us to simplify at least one of these tasks. The syntax analyzer can be smaller and cleaner by removing the low level details of lexical analysis

Efficiency

Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

Portability

Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer.

Attributes For Tokens

Sometimes a token need to be associate with several pieces of information.

The most important example is the token id, where we need to associate with the tokena great deal of information.

Normally, information about an identifier - e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) - is kept in the symbol table.

Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Lexical Errors

A character sequence that can't be scanned into any valid token is a lexical error.

Suppose a situation arises in which the lexical analyzer is unable to proceed becausenone of the patterns for tokens matches any prefix of the remaining input.

The simplest recovery strategy is "panic mode" recovery.

We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.

This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Other possible error-recovery actions are:

1. **Delete one character from the remaining input.**
2. **Insert a missing character into the remaining input.**
3. **Replace a character by another character.**
4. **Transpose two adjacent characters.**

Transformations like these may be tried in an attempt to repair the input.

The simplest such strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by a single transformation.

A more general correction strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes, but this approach is considered too expensive in practice to be worth the effort.

Types of Compiler

1. Cross Compilers

They produce an executable machine code for a platform but, this platform is not the one on which the compiler is running.

2. Bootstrap Compilers

These compilers are written in a programming language that they have to compile.

3. Source to source/transcompiler

These compilers convert the source code of one programming language to the source code of another programming language.

4. Decompiler

Basically, it is not a compiler. It is just the reverse of the compiler. It converts the machine code into high-level language.

Cross Compilers

A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example, a compiler that runs on a PC but generates code that runs on an Android smartphone is a cross compiler.

A cross compiler is useful to compile code for multiple platforms from one development host. Direct compilation on the target platform might be infeasible, for example on embedded systems with limited computing resources.

Cross compilers are distinct from source-to-source compilers. A cross compiler is for cross-platform software generation of machine code, while a source-to-source compiler translates from one coding language to another in text code. Both are programming tools.

The fundamental use of a cross compiler is to separate the build environment from target environment. This is useful in several situations:

- Embedded computers where a device has highly limited resources. For example, a microwave oven will have an extremely small computer to read its keypad and door sensor, provide output to a digital display and speaker, and to control the microwave for cooking food. This computer is generally not powerful enough to run a compiler, a file system, or a development environment.
- Compiling for multiple machines. For example, a company may wish to support several different versions of an operating system or to support several different operating systems. By using a cross compiler, a single build environment can be set up to compile for each of these targets.
- Compiling on a server farm. Similar to compiling for multiple machines, a complicated build that involves many compile operations can be executed across any machine that is free, regardless of its underlying hardware or the operating system version that it is running.
- Bootstrapping to a new platform. When developing software for a new platform, or the emulator of a future platform, one uses a cross compiler to compile necessary tools such as the operating system and a native compiler.

Use of virtual machines (such as Java's JVM) resolves some of the reasons for which cross compilers were developed. The virtual machine paradigm allows the same compiler output to be used across multiple target systems, although this is not always ideal because virtual machines are often slower and the compiled program can only be run on computers with that virtual machine.

Bootstrapping

Bootstrapping is widely used in the compilation development.

Bootstrapping is used to produce a self-hosting compiler. Self-hosting compiler is a type of compiler that can compile its own source code.
Bootstrap compiler is used to compile the compiler and then you can use this compiled compiler to compile everything else as well as future versions of itself.