Module 1

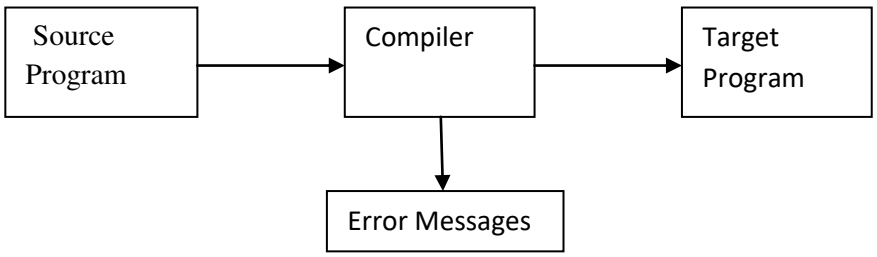
Introduction to compilers: Analysis of source program, Phases of a compiler, Grouping of phases, Compiler writing tools – bootstrapping

Lexical Analysis: The role of lexical analyzer, Input buffering, Specification of tokens using regular expressions, Review of Finite Automata, recognition of Tokens

Compiler

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

Fig: Compiler

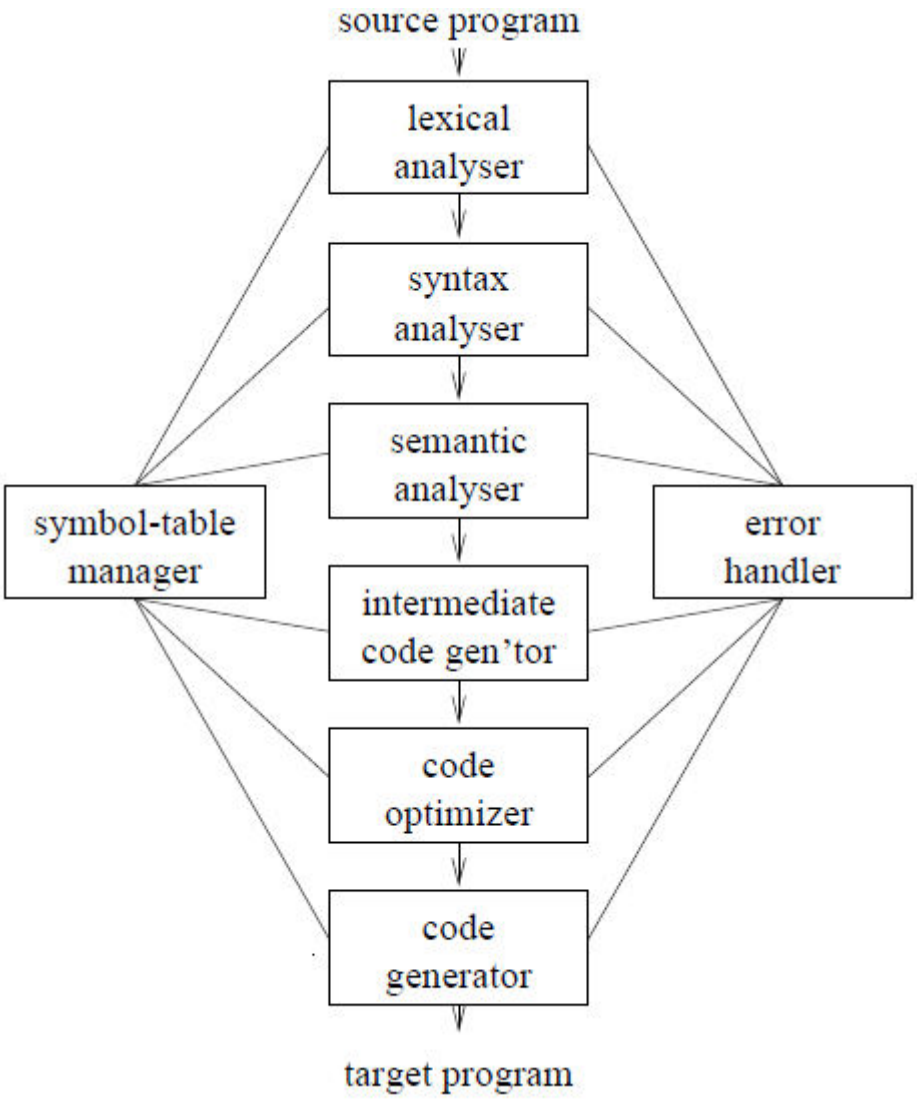


Phases of a Compiler (Structure of Compiler)

The phases include:

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Target Code Generation

Fig: Phases of Compiler



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 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 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 .

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>

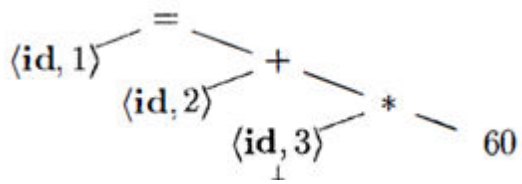
Blanks separating the lexemes would be discarded by the lexical analyzer. The representation of the assignment statement **position = initial + rate * 60** after lexical analysis as the sequence of tokens as:

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

2.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.

The syntax tree for above token stream is:

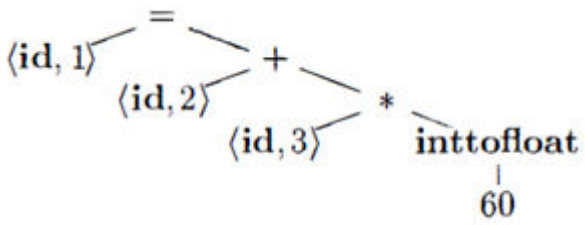


The tree has an interior node labeled with (id, 3) as its left child and the integer 60 as its right child. The node (id, 3) represents the identifier rate. The node labeled * makes it explicit that we must first multiply the value of rate by 60. The node labeled + indicates that we must add the result of this multiplication to the value of init ial. The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier posit ion.

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. 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 inttofloat , which explicitly converts its integer argument into a floating-point number.



4. 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:

1. It should be simple and easy to produce
2. 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 Optimization

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.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. If the target language is assembly language, this phase generate the assembly code as its output. In our example, the code generated is:

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
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 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.**

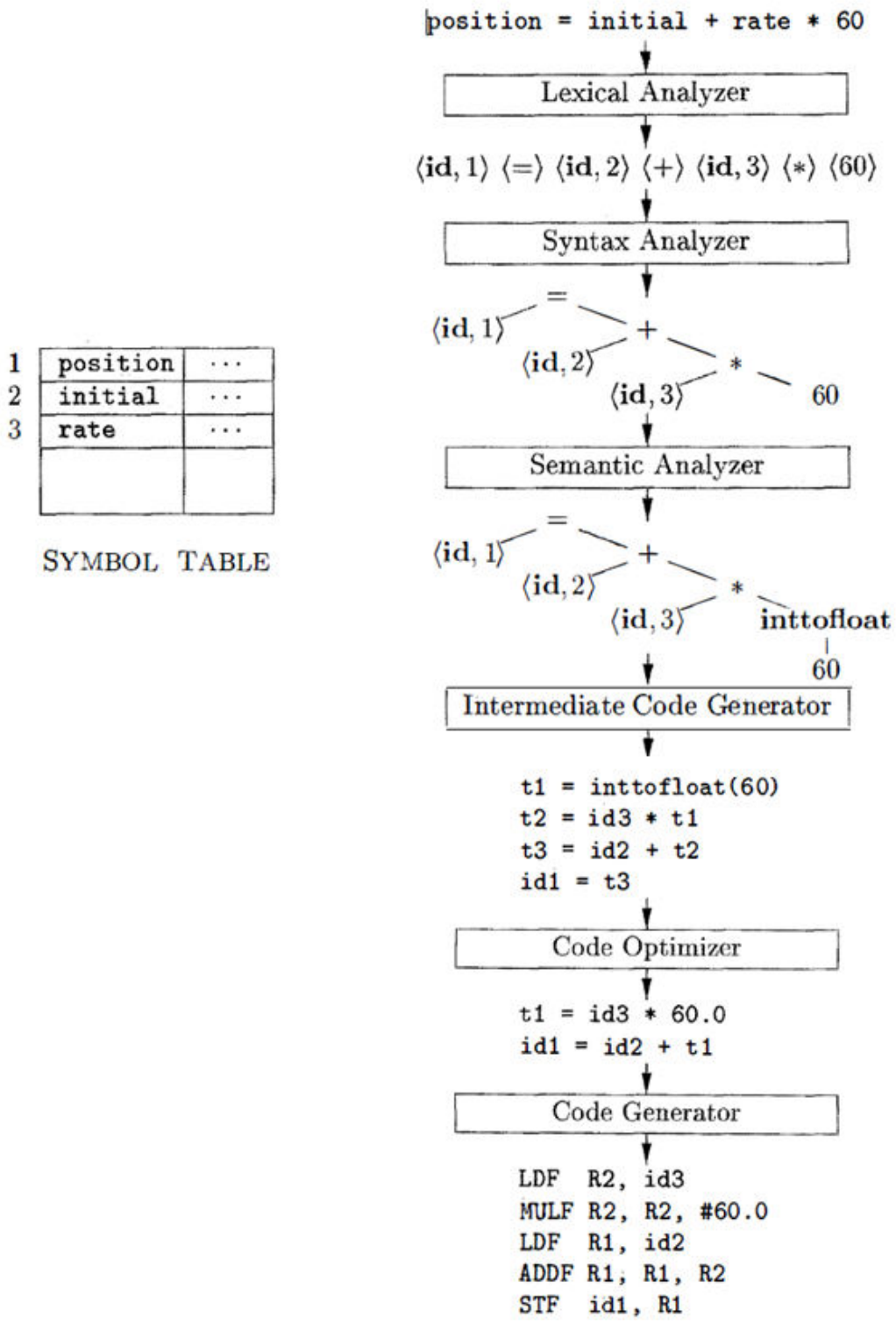
Symbol-Table Management

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.

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 a helpful one.

Fig: Translation of an assignment statement



The process of compilation is split up into following phases:

1.Analysis Phase

2.Synthesis phase

1.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.

2.Synthesis Phase :

Synthesis Phase performs 3 actions namely:

e)Code Optimization

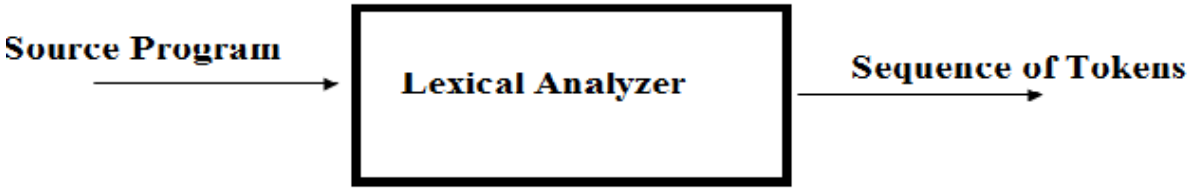
f) Code Generation

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**.

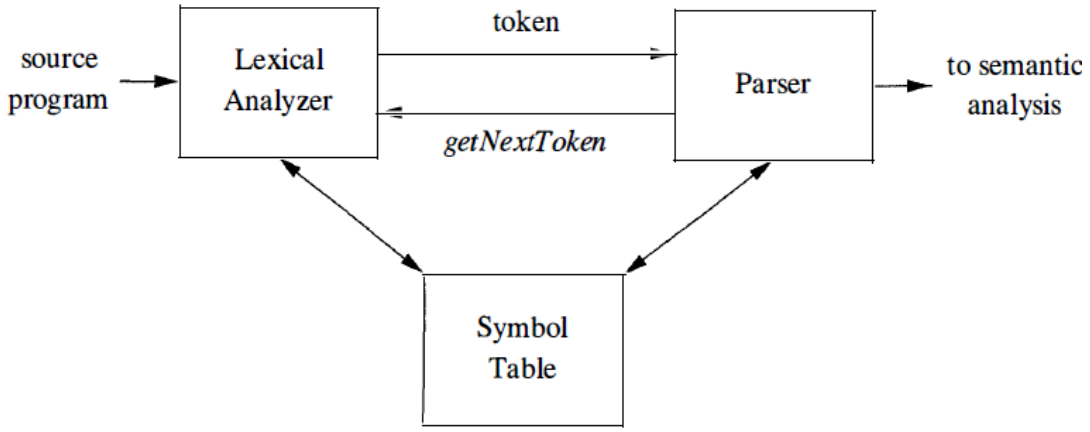
Lexical Analysis and its Role

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.

Fig: Lexical Analyser



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 . Commonly, the interaction is implemented by having the parser call the lexical analyzer. 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 the next token, which it returns to the parser.



Interactions between the lexical analyzer and the parser

Other tasks of Lexical Analyzer:

1. Stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
2. Correlating error messages generated by the compiler with the source program. For instance, 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-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

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 lowlevel 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.

Tokens, patterns and Lexemes

Token - A token is a pair consisting of a token name and an optional attribute value.

< token name, attribute value >

The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes.

Pattern - A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.

Lexeme-A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Figure : Examples of tokens

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 token a 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 because none 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 wprth the effort.