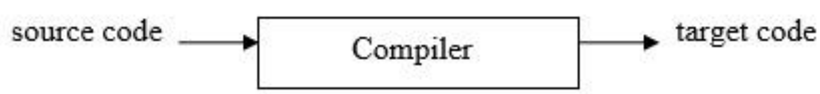
UNIT 2 INTRODUCTION TO COMPILERS

Compiler: Introduction - Analysis of the source program - phases of a compiler - Compiler construction tools-Lexical analysis - Role of the lexical analyzer - Specification of tokens – Recognition of tokens -Lexical analyzer generators- Design aspects of Lexical Analyzer

Compiler:

- ✓ It is a program that translates one language (source code) to another language (target code).



- ✓ It executes the whole program and then displays the errors.
    - ➢ Example: C, C++,COBOL, higher version of Pascal.
- ✓ A compiler translates the code written in one language to some other language without changing the meaning of the program.
- ✓ It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.
- ✓ Compiler design principles provide an in-depth view of translation and optimization process.
- ✓ Compiler design covers basic translation mechanism and error detection & recovery.
- ✓ It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end

Analysis of the source program:

Analysis consists of 3 phases:

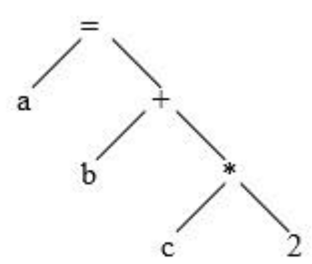
- ➢ Linear/Lexical Analysis
- ➢ Syntax Analysis
- ➢ Semantic Analysis

Linear/Lexical Analysis:

- ✓ It is also called scanning. It is the process of reading the characters from left to right and grouping into tokens having a collective meaning.
- ✓ For example, in the assignment statement a=b+c*2, the characters would be grouped into the following tokens:
    - ➢ The identifier1‘a’
    - ➢ The assignment symbol (=)
    - ➢ The identifier2 ‘b’
    - ➢ The plus sign (+)
    - ➢ The identifier3 ‘c’
    - ➢ The multiplication sign (*)
    - ➢ The constant ‘2’

Syntax Analysis:

- ✓ It is called parsing or hierarchical analysis. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.
- ✓ They are represented using a syntax tree as shown below:



- ✓ 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.

Semantic Analysis:

- ✓ It checks the source programs for semantic errors and gathers type information for the subsequent code generation phase. It uses the syntax tree to identify the operators and operands of statements.
- ✓ An important component of semantic analysis is type checking. Here the compiler checks that each operator has operands that are permitted by the source language specification.

Phases of a Compiler

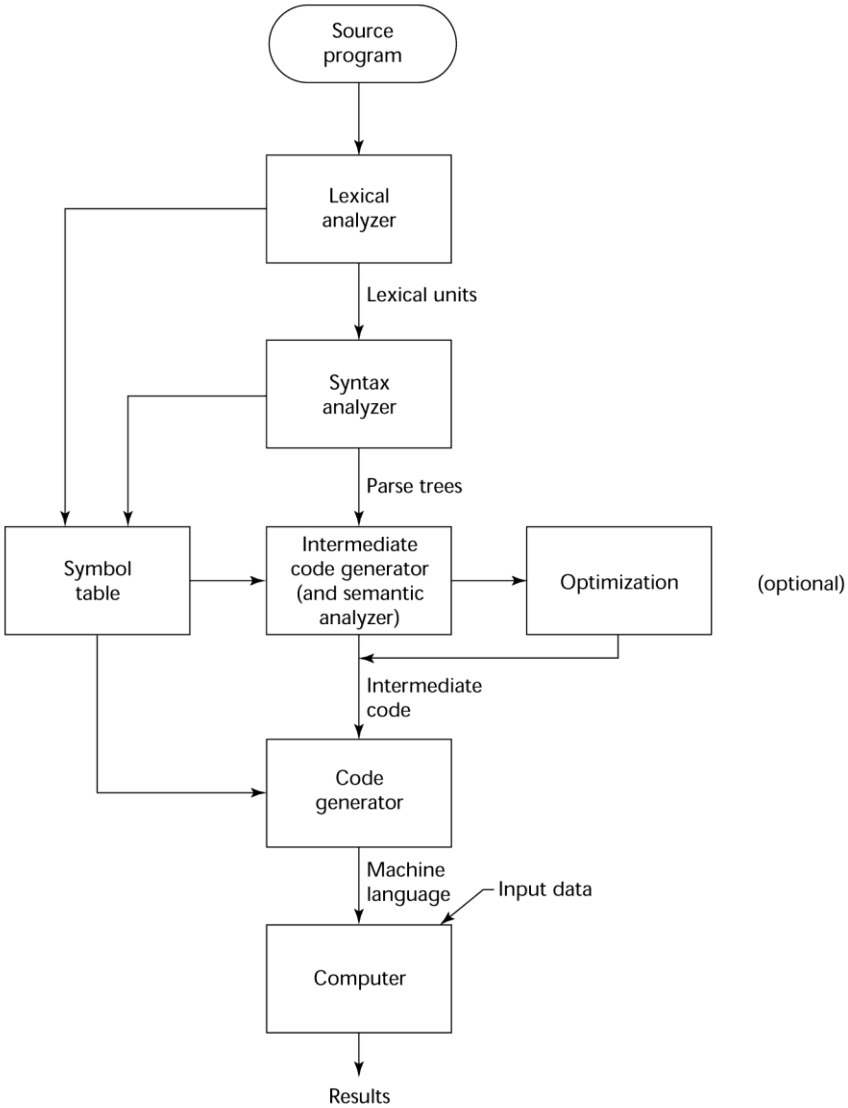
- ✓ A Compiler operates in phases, each of which transforms the source program from one representation into another.
- ✓ The following are the phases of the compiler:

Main phases:

- ➢ Lexical analysis
- ➢ Syntax analysis
- ➢ Semantic analysis
- ➢ Intermediate code generation
- ➢ Code optimization
- ➢ Code generation

Sub-Phases:

- ➢ Symbol table management
- ➢ Error handling



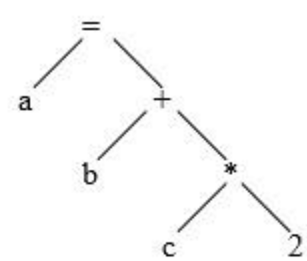
LEXICAL ANALYSIS:

- ✓ It is the first phase of the compiler.
- ✓ It gets input from the source program and produces tokens as output.
- ✓ It reads the characters one by one, starting from left to right and forms the tokens.
- ✓ Token: It represents a logically cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc.
    - ➢ Example: a+b=20 Here, a,b,+,=,20 are all separate tokens. Group of characters forming a token is called the Lexeme.

- ✓ The lexical analyzer not only generates a token but also enters the lexeme into the symbol table if it is not already there.

SYNTAX ANALYSIS:

- ✓ It is the second phase of the compiler.
- ✓ It is also known as parser.
- ✓ It gets the token stream as input from the lexical analyzer of the compiler and generates syntax tree as the output.
- ✓ Syntax tree: It is a tree in which interior nodes are operators and exterior nodes are operands.
    - ➢ Example: For a=b+c*2, syntax tree is



SEMANTIC ANALYSIS:

- ✓ It is the third phase of the compiler.
- ✓ It gets input from the syntax analysis as parse tree and checks whether the given syntax is correct or not.
- ✓ It performs type conversion of all the data types into real data types.

INTERMEDIATE CODE GENERATION:

- ✓ It is the fourth phase of the compiler.
- ✓ It gets input from the semantic analysis and converts the input into output as intermediate code such as three-address code.

- ✓ The three-address code consists of a sequence of instructions, each of which has almost three operands.
    - ➢ Example: t1=t2+t3

CODE OPTIMIZATION:

- ✓ It is the fifth phase of the compiler.
- ✓ It gets the intermediate code as input and produces optimized intermediate code as output.
- ✓ This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
- ✓ During the code optimization, the result of the program is not affected.
- ✓ To improve the code generation, the optimization involves
    - ➢ Deduction and removal of dead code (unreachable code).
    - ➢ Calculation of constants in expressions and terms.
    - ➢ Collapsing of repeated expression into temporary string.
    - ➢ Loop unrolling.
    - ➢ Moving code outside the loop.
    - ➢ Removal of unwanted temporary variables.

CODE GENERATION:

- ✓ It is the final phase of the compiler.
- ✓ It gets input from code optimization phase and produces the target code or object code as result.
- ✓ Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- ✓ The code generation involves -allocation of register and memory -generation of correct references -generation of correct data types -generation of missing code

SYMBOL TABLE MANAGEMENT:

- ✓ Symbol table is used to store all the information about identifiers used in the program.

- ✓ It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- ✓ It allows to find the record for each identifier quickly and to store or retrieve data from that record.
- ✓ Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

ERROR HANDLING:

- ✓ Each phase can encounter errors.
- ✓ After detecting an error, a phase must handle the error so that compilation can proceed.
- ✓ In lexical analysis, errors occur in separation of tokens.
- ✓ In syntax analysis, errors occur during construction of syntax tree.
- ✓ In semantic analysis, errors occur when the compiler detects constructs with right syntactic structure but no meaning and during type conversion.
- ✓ In code optimization, errors occur when the result is affected by the optimization.
- ✓ In code generation, it shows error when code is missing etc.

COMPILER CONSTRUCTION TOOLS:

- ✓ These are specialized tools that have been developed for helping implement various phases of a compiler.
- ✓ The following are the compiler construction tools:
    - ➢ Parser Generators
    - ➢ Scanner Generator
    - ➢ Syntax-Directed Translation
    - ➢ Automatic Code Generators
    - ➢ Data-Flow Engines

Parser Generators:

- ✓ These produce syntax analyzers, normally from input that is based on a context-free grammar. -It consumes a large fraction of the running time of a compiler.

- ➢ Example-YACC (Yet Another Compiler-Compiler).

Scanner Generator:

- ✓ These generate lexical analyzers, normally from a specification based on regular expressions.
- ✓ The basic organization of lexical analyzers is based on finite automation.

Syntax-Directed Translation:

- ✓ These produce routines that walk the parse tree and as a result generate intermediate code.
- ✓ Each translation is defined in terms of translations at its neighbor nodes in the tree.

Automatic Code Generators:

- ✓ It takes a collection of rules to translate intermediate language into machine language.
- ✓ The rules must include sufficient details to handle different possible access methods for data.

Data-Flow Engines:

- ✓ It does code optimization using data-flow analysis, that is, the gathering of information about how values are transmitted from one part of a program to each other part.

LEXICAL ANALYSIS

- ✓ Lexical analysis is the process of converting a sequence of characters into a sequence of tokens.
- ✓ A program or function which performs lexical analysis is called a lexical analyzer or scanner.
- ✓ A lexer often exists as a single function which is called by a parser or another function.
- ✓ Upon receiving a “get next token” command from the parser, the lexical analyzer reads input characters until it can identify the next token.

ISSUES OF LEXICAL ANALYZER

- ✓ There are three issues in lexical analysis:
  - ➢ To make the design simpler.

- To improve the efficiency of the compiler.
- To enhance the computer portability.

SPECIFICATION OF TOKENS

There are 3 specifications of tokens:

- Strings
- Language
- Regular expression

Strings and Languages

- ✓ An alphabet or character class is a finite set of symbols.
- ✓ A string over an alphabet is a finite sequence of symbols drawn from that alphabet.
- ✓ A language is any countable set of strings over some fixed alphabet.
- ✓ In language theory, the terms "sentence" and "word" are often used as synonyms for "string."
- ✓ The length of a string s, usually written |s|, is the number of occurrences of symbols in s.
    - For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

Operations on strings:

The following string-related terms are commonly used:

- A prefix of string s is any string obtained by removing zero or more symbols from the end of strings.
- For example, ban is a prefix of banana.
- A suffix of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, nana is a suffix of banana.
- A substring of s is obtained by deleting any prefix and any suffix froms. For example, nan is a substring of banana.
- The proper prefixes, suffixes, and substrings of a string s are those prefixes, suffixes, and substrings, respectively of s that are not ε or not equal to s itself.

- A subsequence of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, baan is a subsequence of banana.

Operations on languages:

- The following are the operations that can be applied to languages:
- 1.Union 2.Concatenation 3.Kleene closure 4.Positive closure
- The following example shows the operations on strings:
- Let L={0,1} and S={a,b,c}
    - Union : L U S={0,1,a,b,c}
    - Concatenation: L.S= {0a,1a,0b,1b,0c,1c}
    - Kleene closure: L*= {ε,0,1,00….}
    - Positive closure: L+= {0,1,00….}

Regular Expressions

- ✓ Each regular expression r denotes a language L(r).
- ✓ Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:
- ✓ ε is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the empty string.
- ✓ If 'a' is a symbol in Σ, then 'a' is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with 'a' in its one position.
- ✓ Suppose r and s is regular expressions denoting the languages L(r) and L(s). Then,
    - (r)|(s) is a regular expression denoting the language L(r) U L(s).
    - (r)(s) is a regular expression denoting the language L(r)L(s).
    - (r)* is a regular expression denoting (L(r))*.
    - (r) is a regular expression denoting L(r).
- ✓ The unary operator * has highest precedence and is left associative.
- ✓ Concatenation has second highest precedence and is left associative.

- ✓ It has lowest precedence and is left associative.

RECOGNITION OF TOKENS

- ✓ Consider the following grammar fragment:
  - ➢ stmt→if expr then stmt

    |if expr then stmt else stmt

    |ε
  - ➢ expr→term relop term

    |term
  - ➢ term→id

    |num

where the terminals if , then, else, relop, id and num generate sets of strings given by the following regular definitions:

  - ➢ if → if
  - ➢ then → then
  - ➢ else → else
  - ➢ relop → <|<=|=|<>|>|>=
  - ➢ id → letter(letter|digit)*
  - ➢ num → digit+ (.digit+)?(E(+|-)?digit+)?

- ✓ For this language fragment the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id, and num.
- ✓ To simplify matters, we assume keywords are reserved; that is, they cannot be used as identifiers.

Lexical analyzer generators:

- ✓ There is a wide range of tools for constructing lexical analyzers.
  - ➢ Lex

➢ YACC

LEX:

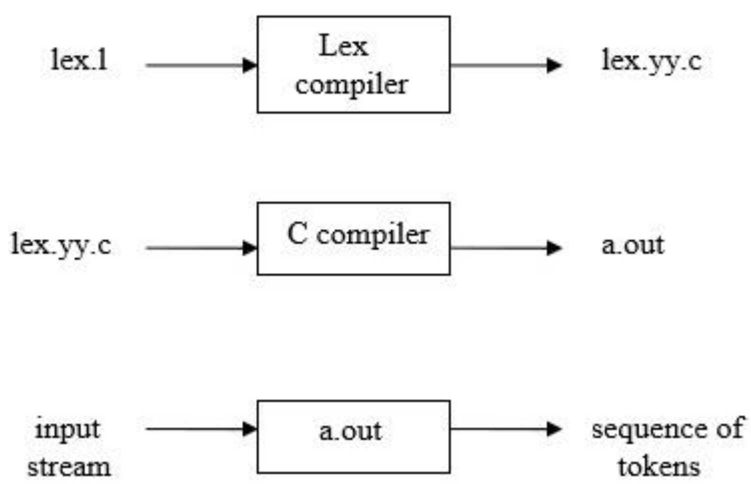
✓ Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator.

Creating a lexical analyzer

✓ First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language.

✓ Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.

✓ Finally, lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.



YACC-YET ANOTHER COMPILER-COMPILER:

✓ Yacc provides a general tool for describing the input to a computer program.

✓ The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized.

✓ Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.