Q1- A- Differentiate between the Analysis phase and Synthesis phase.

Answer:

There are two main phases in the compiler.

1. Analysis - Front end of a compiler
2. Synthesis - Back end of a compiler

Analysis phase of compiler:

Analysis phase reads the source program and splits it into multiple tokens and constructs the intermediate representation of the source program.

And also checks and indicates the syntax and semantic errors of a source program.

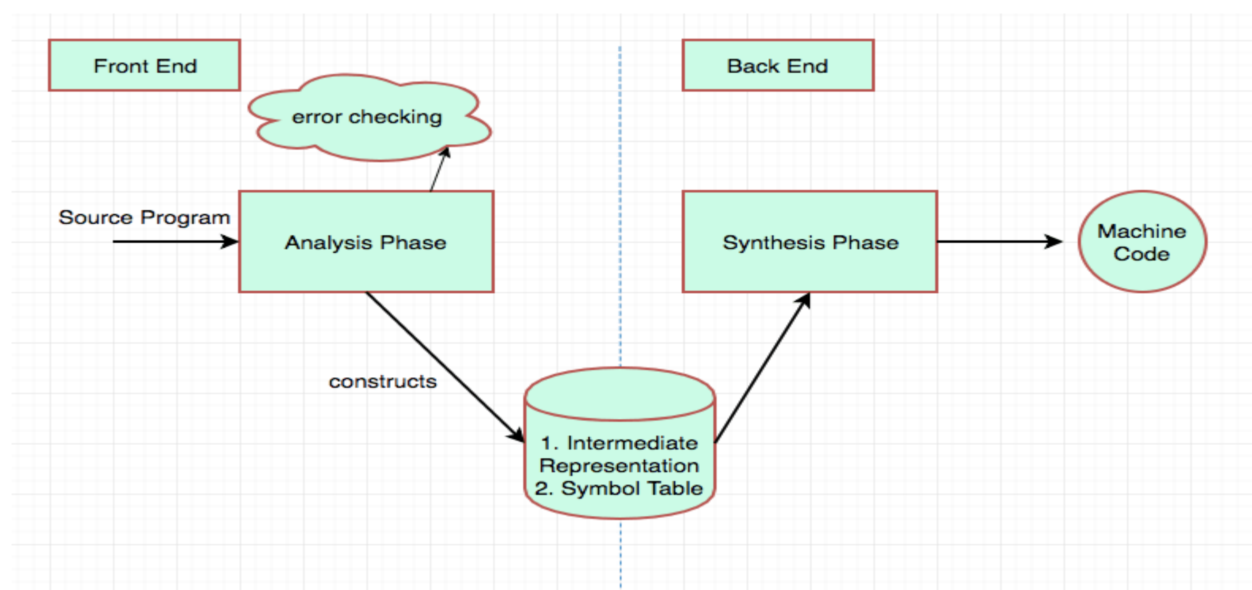
It collects information about the source program and prepares the symbol table. Symbol table will be used all over the compilation process.

This is also called as the front end of a compiler.

Synthesis phase of compiler:

It will get the analysis phase input (intermediate representation and symbol table) and produces the targeted machine level code.

This is also called as the back end of a compiler.



Q1-B- Draw the model identifying the phases of complier and briefly define each Phase.

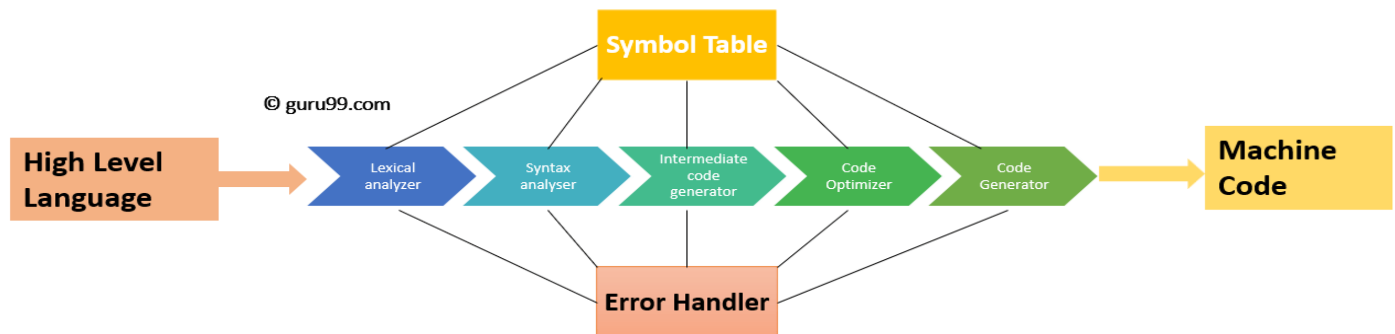
Answer:

Compiler operates in various phases each phase transforms the source program from one representation to another. Every phase takes inputs from its previous stage and feeds its output to the next phase of the compiler.

There are 6 phases in a compiler. Each of this phase help in converting the high-level langue the machine code.

The phases of a compiler are:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generator
5. Code optimizer
6. Code generator



1. Lexical Analysis

Lexical Analysis is the first phase when compiler scans the source code. This process can be left to right, character by character, and group these characters into tokens.

Here, the character stream from the source program is grouped in meaningful sequences by identifying the tokens. It makes the entry of the corresponding tickets into the symbol table and passes that token to next phase.

The primary functions of this phase are:

- Identify the lexical units in a source code
- Classify lexical units into classes like constants, reserved words, and enter them in different tables. It will Ignore comments in the source program
- Identify token which is not a part of the language.

2. Syntax Analysis

Syntax analysis is all about discovering structure in code. It determines whether or not a text follows the expected format. The main aim of this phase is to make sure that the source code was written by the programmer is correct or not.

Syntax analysis is based on the rules based on the specific programing language by constructing the parse tree with the help of tokens. It also determines the structure of source language and grammar or syntax of the language.

Here, is a list of tasks performed in this phase:

- Obtain tokens from the lexical analyzer
- Checks if the expression is syntactically correct or not
- Report all syntax errors
- Construct a hierarchical structure which is known as a parse tree

3. Semantic Analysis

Semantic analysis checks the semantic consistency of the code. It uses the syntax tree of the previous phase along with the symbol table to verify that the given source code is semantically consistent. It also checks whether the code is conveying an appropriate meaning.

Semantic Analyzer will check for Type mismatches, incompatible operands, a function called with improper arguments, an undeclared variable, etc.

Functions of Semantic analyses phase are:

- Helps you to store type information gathered and save it in symbol table or syntax tree
- Allows you to perform type checking
- In the case of type mismatch, where there are no exact type correction rules which satisfy the desired operation a semantic error is shown
- Collects type information and checks for type compatibility
- Checks if the source language permits the operands or not

4. Intermediate Code Generation

Once the semantic analysis phase is over the compiler, generates intermediate code for the target machine. It represents a program for some abstract machine.

Intermediate code is between the high-level and machine level language. This intermediate code needs to be generated in such a manner that makes it easy to translate it into the target machine code.

Functions on Intermediate Code generation:

- It should be generated from the semantic representation of the source program
- Holds the values computed during the process of translation
- Helps you to translate the intermediate code into target language
- Allows you to maintain precedence ordering of the source language
- It holds the correct number of operands of the instruction.

5. Code Optimization

The next phase of is code optimization or Intermediate code. This phase removes unnecessary code line and arranges the sequence of statements to speed up the execution of the program without wasting resources. The main goal of this phase is to improve on the intermediate code to generate a code that runs faster and occupies less space.

The primary functions of this phase are:

- It helps you to establish a trade-off between execution and compilation speed
- Improves the running time of the target program
- Generates streamlined code still in intermediate representation
- Removing unreachable code and getting rid of unused variables
- Removing statements which are not altered from the loop.

6. Code Generation

Code generation is the last and final phase of a compiler. It gets inputs from code optimization phases and produces the page code or object code as a result. The objective of this phase is to allocate storage and generate locatable machine code.

It also allocates memory locations for the variable. The instructions in the intermediate code are converted into machine instructions. This phase coverts the optimize or intermediate code into the target language.

The target language is the machine code. Therefore, all the memory locations and registers are also selected and allotted during this phase. The code generated by this phase is executed to take inputs and generate expected outputs.

Q1-C- Explain Regular Expressions in Compilers.

Answer:

Regular Expressions

A sequence of symbols and characters expressing a string or pattern to be searched for within a longer piece of text. Regular Expressions

Regular Expressions we were able to describe identifiers by giving names to sets of letters and digits and using the language operators union, concatenation, and closure. This process is so useful that a notation called regular expressions has come into common use for describing all the languages that can be built from these operators applied to the symbols of some alphabet. In this notation, if letter- is established to stand for any letter or the underscore, and digit- is established to stand for any digit.

Regular Expressions The regular expressions are built recursively out of smaller regular expressions using the rules described below. Each regular expression r denotes a language L(r), which is also defined recursively from the languages denoted by r's sub expressions. Here are the rules that define the regular expressions over some alphabet C and the languages that those expressions denote.

BASIS

1. E is a regular expression, and L (E) is {E), that is, the language whose sole member is the empty string.

2. If a is a symbol in C, 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. Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.

INDUCTION

1. (r) 1 (9) is a regular expression denoting the language L(r) U L(s).

2. (r) (s) is a regular expression denoting the language L(r) L(s) .

3. (r) * is a regular expression denoting (L (r)) * .

4. (r) is a regular expression denoting L(r). This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

Precedence

As defined, regular expressions often contain unnecessary pairs of paren- theses. We may drop certain pairs of parentheses if we adopt the conventions that:

a) The unary operator * has highest precedence and is left associative.
b) Concatenation has second highest precedence and is left associative.
c) I has lowest precedence and is left associative.

Under these conventions, for example,

we may replace the regular expression (a) I ((b) * (c)) by a/ b*c. Both expressions denote the set of strings that are either a single a or are zero or more b's followed by one c.

Let C = {a, b}.

1. The regular expression a1 b denotes the language {a, b}.
2. (a1 b) (alb) denotes {aa, ab, ba, bb), the language of all strings of length two over the alphabet C. Another regular expression for the same language is aalablbaI bb.
3. a* denotes the language consisting of all strings of zero or more a's, that is, {E, a, aa, aaa, . . . }.

Let C = {a, b}

4. (alb)* denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: {e, a, b, aa, ab, ba, bb, aaa, . . .}. Another regular expression for the same language is (a*b*)*.

5. ala*b denotes the language {a, b, ab, aab,aaab,. . .), that is, the string a and all strings consisting of zero or more a's and ending in b.

THE END