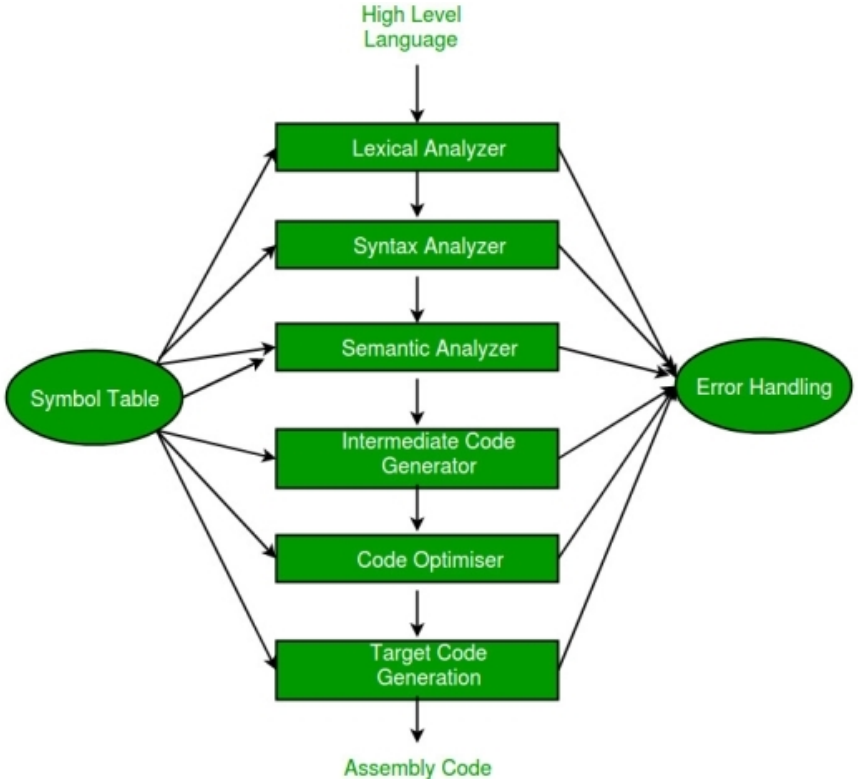
TOY COMPILER

Group Members:

Harsh Kumar **IIT2017098**
Tejas Ramesh Pawar **IIT2017109**
Kunal Kumar Prasad **IIT2017112**
Gaurav Kumar **IIT2017115**
Mirza Mohd. Aadil Beg **IIT2017145**
Aman Gupta **IWM2017006**

Introduction: Phases of Compilation



Lexical Analysis

The process, **lexing or tokenization,** is converting of the given strings into tokens/sequence of tokens. It is the first phase of compiler, called **Scanner/lexer/tokenizer**. The input provided is high level language and is converted into tokens.

What is a token ? - It is a sequence of characters that are treated as a unit in the grammar of programming languages.

Lexical Analysis: Tokens and Lexemes

Examples of token values

Token name	Sample token values
identifier	`x`, `color`, `UP`
keyword	`if`, `while`, `return`
separator	`}`, `(`, `;`
operator	`+`, `<`, `=`
literal	`true`, `6.02e23`, `"music"`

Examples of Non-Tokens: Comments, preprocessor directive, macros, blanks, tabs, newline, etc

Lexemes: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. **eg**- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

Lexemes and tokens:

- scanner is based on grammar of that programming language
- lexemes: lexemes are instance of token
- Tokens: It is structured as a pair consisting of a token name and an optional token value.
- tokens are then inserted into symbol table

Examples of Tokens created

Lexeme	Token
int	Keyword
maximum	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
Y	Identifier
)	Operator
{	Operator
If	Keyword

When a source program is fed into the lexical analyzer, it begins by breaking up the characters into sequences of lexemes. The lexemes are then used in the construction of tokens, in which the lexemes are mapped into tokens. A variable called **myVar** would be mapped into a token stating <**id**, "num">, where "num" should point to the variable's location in the symbol table.

Shortly put:

- Lexemes are the words derived from the character input stream.
- Tokens are lexemes mapped into a token-name and an attribute-value.

An example includes:
x = a + b * 2
Which yields the lexemes: {x, =, a, +, b, *, 2}
With corresponding tokens: {<**id**, 0>, <=>, <**id**, 1>, <+>, <**id**, 2>, <*>, <**id**, 3>}

Lexical Analysis in our Toy Compiler

- We take the high level language as the input and the first phase of compilation is to perform lexical analysis, the class **InputReader** is called to tokenize our input string. The phase is also called scanning/lexeing or tokenizing.
- In **InputReader**, we define attributes like lineMap, tokens, tokensOfEachLine with an upper limit to 100, that means this toy compiler support tokenization of at max 100 lines.
- We generate the tokens using **tokensGenerator()** method by taking buffer stream into a string called 'inputString'. (*We then divide the program into tokens.*)
- We parallely update tokens on each line by **mapFiller()** method and we hash all subtoken to their respective line numbers. We also delete **whitespaces** by *.split*(\ \).
- We have three comments mode that are double slash, open bracket and closed bracket for multiple lines which are removed as they are unnecessary. *i.e., removing the comments.*
- To get all the tokens, we call **getTokens()** method and to get tokens on each line we call **getTokensOfEachLine()** method so we can get count of words on that line number.
- The lexical analysis function also help in giving error message, if any using base case, etc.

Syntax Analysis

Syntax Analysis or Parsing is the second phase of the compiler followed by Lexical Analysis. In this process the Syntax Analyzer takes the input from Lexical Analyzer in form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code.

A parse tree is constructed by using the predefined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. If not, error is reported by syntax analyzer.

Syntax Analysis in our Toy Compiler

- The tokens generated from the previous step are used as the input for the Syntax Analyser along with a couple more helper functions.
- The Syntax Analyzer uses class **SyntaxStateMachine** and **syntaxHandler** as whole. The latter class is used to create a flag whether to advance to the next phase i.e. Semantic Analyzer, while the other one creates a parser by assigning integer values to specific tokens and then storing it into a 2-D matrix. The reason for the 2 dimensions are that each row corresponds to each line in the code and an entry in each row resemble a corresponding token number.
- The **getTokensOfEachLine()** stores tokens line-by-line and in an array and checks syntax of that particular line. If error occurs in a line, the function breaksand returns.
- The error handling is done using switch-case statements such that the case corresponds to the integer value given to the token error. This is implemented using the function **errorHandler().**

Semantic Analysis

Semantic Analysis is the third phase of Compiler. Semantic Analysis makes sure that declarations and statements of program are semantically correct.It is a collection of procedures which is called by parser as and when required by grammar. Both syntax tree of previous phase and symbol table are used to check the consistency of the given code.

Semantic Analyzer uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition. It gathers type information and stores it in either syntax tree or symbol table. This type information is subsequently used by compiler during intermediate-code generation.

Code Generation

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language.We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

Code Generation in our Toy Compiler:-

- We have a class called **CodeGenerator** that has all the functions of Code Generation implemented in it.So, we have generated the assembly code from input code using this class. The assembly code that we have generated uses 16 bit for representation of every call, in which first 4 bits are for operators, next 4 bits for register numbers or memory location and others for memory locations used or any other work according to operator.
- We subcategorized Code Generation into Statement Code Generation and Expression Code Generation and formed functions like **statementCodeGeneratorHandler** and **expressionCodeGeneratorHandler** in CodeGenerator.
- We formed functions like **memoryFiller** to fill the memory, **uselessRegisterIndexFinder** to find useless Register and various other such functions.
- So, we have implemented various functions for each operator of assembly that we are using for writing its assembly code.