**Compiler for a Toy Language - ERPLAG**

-Krittika (16103035)

Shivam (16103037)

1. **Phases of a Compiler**
   1. Lexical Analysis:

Lexical Analysis is the first phase when the 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 the 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

|  |  |
| --- | --- |
| *X* | *identifier* |
| *=* | *Assignment operator* |
| *Y* | *identifier* |
| *+* | *Addition operator* |
| *10* | *Number* |

*Example:*

*x = y + 10; Tokens:*

* 1. 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 programming 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.

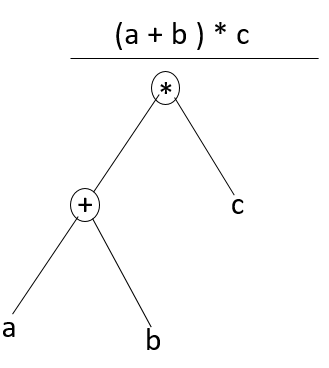
Here, is a list of tasks performed in this phase:

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

*Example:*

*Any identifier/number is an expression.*

*If x is an identifier and y+10 is an expression, then x= y+10 is a statement.*

*Consider parse tree for the following example:*

*(a + b) \* c;*

*In Parse Tree:*

* *Interior node: record with an operator filed and two files for children*
* *Leaf: records with 2/more fields; one for token and other information about the token*
* *Ensure that the components of the program fit together meaningfully*
* *Gathers type information and checks for type compatibility*
* *Checks operands are permitted by the source language*
  1. 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

*Example:*

*float x = 20.2;*

*float y = x\*30;*

*In the above code, the semantic analyser will typecast the integer 30 to float 30.0 before multiplication*

* 1. Intermediate code generator:

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

*Example:*

*total = count + rate \* 5;*

*Intermediate code with the help of address code method is:*

*t1 := int\_to\_float(5)*

*t2 := rate \* t1*

*t3 := count + t2*

*total := t3*

* 1. Code optimizer:

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:

* Helps you to translate the intermediate
* 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

*Example:*

*Consider the following code*

*a = intofloat(10);*

*b = c \* a;*

*d = e + b;*

*f = d;*

*Can become*

*b =c \* 10.0;*

*f = e + b;*

* 1. Code generator:

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 relocatable 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.

*Example:*

*a = b + 60.0;*

*Would be possibly translated to registers.*

*MOVF a, R1*

*MULF #60.0, R2*

*ADDF R1, R2*

1. **Project Update**

After understanding the above stages of the compiler, using the language constructs given in the pdf attached, **we were able to implement the lexical analysis phase of the said ERPLAG compiler**. The code for the same is also attached.

The analyser reads the input file into a buffer. From the buffer, it starts reading individual characters and forming valid tokens by analysing the state in which the current line is and spot which we are parsing. E.g. if we find an alphabet, we enter state 26. Now if the alphabet is just a single alphabet, we terminate it as a single token. Else, if the alphabet goes on to form an alphanumeric string (a word) then the state from the 2nd character of the string till we find any other symbol than an alphabet or a number becomes 27. Now, the word is matched with all the keywords. If a keyword is found, a keyword is assigned to it. Else, it is labelled as an ID. Similarly, if we find a word starting from an integer, the analyser goes into state 28 and the number is labelled as NUM. any alphabets following the number will again form an ID.

*Note: an alphabet followed by a number will form an ID because we can create variables starting with an alphabet. However, a number followed by any alphabets will not form an ID as variable names cannot start with numbers.*

*For example:*

*temp => ID*

*temp1 => ID*

*tem23p => ID*

*2temp => NUM, ID*

*23 => NUM*

Whenever a token is formed successfully, the code outputs the token token label, the token, and it’s line number.

*For example:*

*The line (assuming it appears on line number 12 in the code)*

*sum:= sum + k\*x;*

*will give:*

*------------- ID, sum, 12*

*------------- ASSIGNOP, :=, 12*

*------------- ID, sum, 12*

*------------- PLUS, +, 12*

*------------- ID, k, 12*

*------------- MUL, \*, 12*

*------------- ID, x, 12*

*------------- SEMICOL, ;, 12*

A few testcases, taken from <https://github.com/g31pranjal/erplag-compiler/tree/master/testcases> and <https://github.com/akankshadara/ERPLAG-Compiler/tree/master/testcases> have been tested and attached along with the code. More help from <https://github.com/g31pranjal/erplag-compiler/tree/master/testcases> was taken to understand the structure of the compiler.