UNIVERSITÉ LIBRE DE BRUXELLES

Project part 1 : Lexical Analyzer

Subject : introduction to language theory and compiling

TRANGANIDAS Orestis
BOUGLAM Sara

Contents

1 Introduction

In computing, compilers refer to the idea of transforming a code written in a source code to an object code which is low level language that can be understood by the computer at a hardware stage, namely Assembly.

Although writing software in higher level language rather than assembly, may have a positive impact on the productivity; however, compilers took a considerable time for their establishment due to technical issues caused by memory restriction in early computers.

By the 1950s, compilers witnessed enormous evolution; starting from the PhD thesis introduced by Corrado Böhm, then the first implemented compiler by Grace Hopper, which contained the first compiler, entitled A-0 System and followed by the first commercial compiler developed by The Fortran team at IBM.

Compilers generally go through 3 distinguishable phases:

> *Frontend phase* is responsible for checking the lexical, grammar and syntax error after dividing the code into core parts.
>
> *Middle-end phase* which aims to improve the efficiency of a program through reducing execution time
>
> *Backend phase* is the phase where the code generation occurs, in another word the program is translated into target language.

The focus of this report would be mainly on the frontend phase more specifically the lexical analysis, where a scanner for the FORTRAN language (FORmula TRANslation) would be implemented using the JFlex language which is a scanner generator for java programming language as well as exploring further this part of the phase, its main content and the implementation phases.

2 Background

2.1 Lexical Analysis

Consists of the first stage of the compiler where it reads the sequence of characters contained in the input file and converts it into a sequence of tokens as known symbol table.2 phases of lexical analysis can be distinguished namely: Scanner, screener.

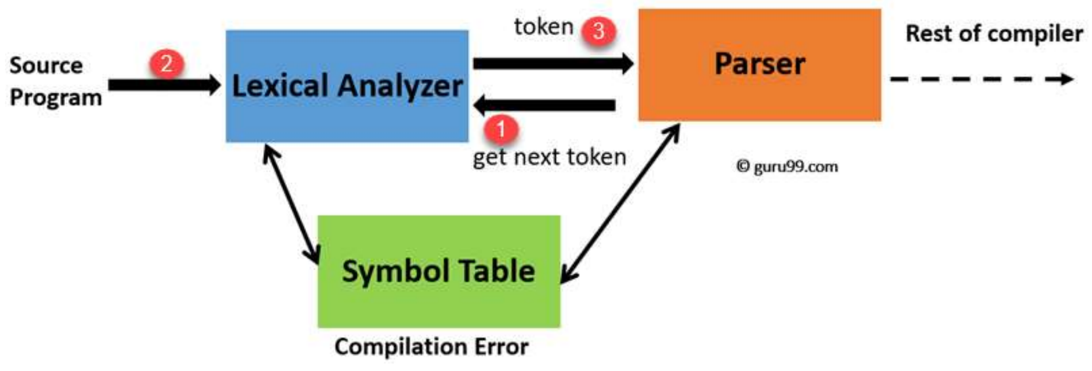


Figure 1: Lexical analyzer architecture (1)

2.2 Symbol Class

Mainly refers to the set of symbols that comprise an input file such as the constant, identifiers strings and are recognized by the parser, where *the symbol* is a word ∑* over the *alphabet* ∑ which is the set of characters occurring in the file. (2)

2.3 Scanner:

Conceptually, scanner represents a finite automaton, its main role is to deliver a sequence of symbols to the screener where each symbol is a substring of the input and labeled with its symbol class (2)

2.3.1 Regular Expression

Describes a sequence of characters that define a search pattern (3) also is used to specify the symbol class. Each regular language can be defined by a regular expression, e.g. the language **{a,b}** can be defined by the regular expression **a|b.** (2)

2.3.2 Finite automaton

Finite automaton is used to implement the recognizers, eventually it's an acceptor of regular language, and formally it can be represented by the tuple (2) M= (Q, ∑, Δ, Q0, F):

- **Q** is a finite set of state
- **∑** is finite alphabet
- **Q0 ⊆ Q** which represents the initial state
- **F ⊆ Q** represents the final state
- **Δ ⊆ Q *(∑ ∪ {ε})*Q** represents the transition relation

2.3.3 Character classes

Consists of grouping set of characters into classes where they can be exchanged against each other (2)

2.4 Screener

The screener pursues the processing of the symbols' sequence and achieves the following (2):

1. Removing the separators and comments
2. Removing reserved Symbols that characterize the target language
3. Convert the constant number into binary representation and storing string constant in an allocated memory
4. Storing the identifiers in a data structure in order to efficiently address them by their code.

2.5 Parser:

Takes as an input stream of tokens and produces a parse tree data structure based on the grammar rules (4)

2.6 Lexical Analyzer generator:

Takes as an input the specification with a set of regular expressions and actions, therefor it generates a program called lexer that reads an input and matches it to the specific file, eventually run the corresponding action if a regular expression matches. (5)

3 Implementation

The Lexical analyzer has been implemented using a scanner generator JFlex for the FORTRAN programming language which is a language that dominated the scientific computing for decades, originally, developed by IBM (6) .

The implementation files will consist of:

LexicalAnalyzer.flex: represents the scanner generator used.

LexicalAnalyzer.java: is the class that is automatically generated by the scanner generator.

LexicalUnit.java: consists of the different units that characterize the language as well as the reserved words.

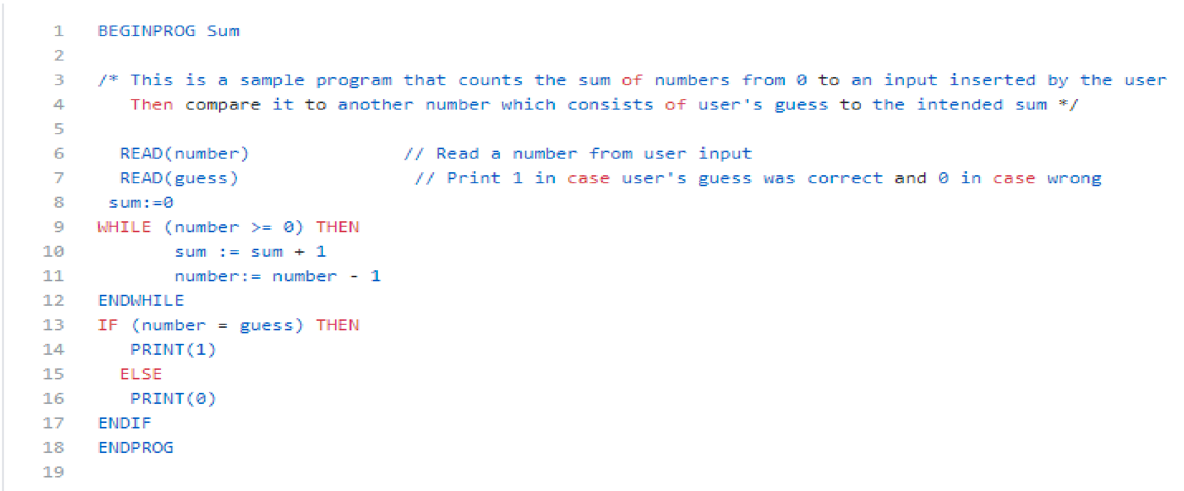
Symbol.java: a class which takes into one of its constructor's parameters a LexicalUnit and does the display of a single symbol table column.

3.1 JFlex

The lexical analyzer has been implemented using JFlex which a scanner generator written in java.

3.2 Part 1

The first step of the implementation process is to analyze a sample program in FORTRAN Language, in order to extract the different Symbols, reserved words and important characteristics of the language that would serve to generate the regular expressions and eventually the symbol table that would be delivered to the parser later on, during the syntactic analysis



```
BEGINPROG Sum

/* This is a sample program that counts the sum of numbers from 0 to an input inserted by the user
   Then compare it to another number which consists of user's guess to the intended sum */

  READ(number)              // Read a number from user input
  READ(guess)                // Print 1 in case user's guess was correct and 0 in case wrong
 sum:=0
WHILE (number >= 0) THEN
       sum := sum + 1
       number:= number - 1
ENDWHILE
IF (number = guess) THEN
   PRINT(1)
  ELSE
   PRINT(0)
ENDIF
ENDPROG
```

Figure 2: A sample program in FORTRAN

3.2.1 Observation

From the figure, different lexical units can be distinguished:

- Program structure: BEGINPROG ENDPROG.
- Conditions and loops: WHILE, THEN, IF, THEN, ELSE, ENDIF.
- Reading and printing tools: READ, PRINT.
- Different relational operator :), (, =, < ...etc.
- Variables and program names: Sum, number, guess.
- Comments: /* ,*/ ,//

Moreover, every lexical unit is separated by a white space.

3.2.1.1 Note

given the fact that JFlex has been used to generate the lexical analyzer, it resulted on easing the implementation process, considering that separation of the characters contained in the input file as well as state transition is automatically yielded and handled by the scanner generator, therefor the main focus is on generating the appropriate regular expression that would match the language given.

3.3 Part 2: Regular Expressions

3.3.1 Program and variables names:

3.3.1.1 Variable names restrictions:

- They can only be composed of sequence of digits and letters in the range of a-z
- They start by a letter only and may end by a digit
- Uppercase letters are not allowed

The following regular expression matches the restriction:

```
[a-z]{1}([0-9]|[a-z])*
```

3.3.1.2 Program names restrictions:

- They can only be composed of sequence of digits and letters in the range of a-z
- They start by a letter only and may end by a digit
- Uppercase letters are allowed, however, it cannot be composed entirely of uppercase letters
- The program name should start with an uppercase letter

The following regular expression matches the restriction:

`[A-Z]{1}([A-Z]|[0-9])*[a-z]+([A-Z]|[0-9]|[a-z])*`

3.3.2 Comments

Comments are supposed to be ignored; two types of comments can be distinguished:

3.3.2.1 Long comments

Long comments start by /* and end with */ and may take several lines, the regular expression regarding this restriction is:

`[/][*][^*]*[*]+([^*/][^*]*[*]+)*[/]`

3.3.2.2 Short comments

Short comments start by // and their end is marked by the end of the line, the regular expression as follows: `“//”.*`

3.3.3 Numbers

For numbers, there exist Integers and floats; numbers such 018 not allowed, however, 0.9 is accepted. From these two restrictions the following regular expression, can be extracted:

3.3.3.1 Character class

Since both of integers and floats are handled, the following character class is used to generate the number regular expression:

Integer = `[1-9][0-9]*|0`

Decimal = `\.[0-9]*`

Exponent = `[eE]{Integer}`

3.3.3.2 Number

From the character class provided previously, the number’s regular expression can simply be of the form: `{Integer}{Decimal}?{Exponent}?`

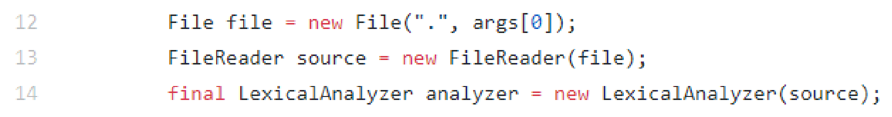
3.3.4 Reserved words and relational operators

For reserved words, the task was simpler; it consisted mainly on a manual iteration through a finite sequence of reserved words and comparing to each string contained in the input file

3.4 Part 3

This part which is also is the last one was mainly concerned with using the lexical analyzer that was generated automatically by JFlex.

For this purpose the implementation consisted of reading the provided file as an argument using the Java IO package more specifically the File reader.



```
        File file = new File(".", args[0]);
        FileReader source = new FileReader(file);
        final LexicalAnalyzer analyzer = new LexicalAnalyzer(source);
```

Figure 3 : excerpt from the Main.java class

Then extracting the variables, storing them in a data structure and finally displaying them in an alphabetical order in the standard output stream after eliminating duplicates.

4 Conclusion

To conclude, the whole implementation laid on 3 steps which consist of: analyzing the characteristics of the language then generating the regular expressions and finally linking the entire pieces together through the main class.

JFlex is a powerful a tool, not only it allowed easing and speeding up the process of generating the scanner for the target language but also it has a built in support for some parser generators that would be important for the upcoming phase of the compiler.

6 Bibliography

1. guru99. [Online] https://www.guru99.com/compiler-design-lexical-analysis.html.

2. **Seidl, Helmut, Hack, Sebastian and Wilhelm, Reinhard.** Compiler design: syntactic and semantic analysis. 2013.

3. *wikipedia.* [Online] https://en.wikipedia.org/wiki/Regular_expression.

4. [Online] https://datacadamia.com/code/compiler/parser#:~:text=A%20parser%20takes%20a%20token,a%20parse%20tree%20data%20structure..

5. JFlex. *JFlex.* [Online] https://jflex.de/.

6. Wikipedia. *Fortran.* [Online] https://en.wikipedia.org/wiki/Fortran.