# Investigation of Lexical Analysis Phase with JFlex

Thet Myint
*Computer University (Monywa)*
thetmyint.29@gmail.com

## Abstract

*Lexical analysis is the initial part of reading and analyzing the program text in the compilation process. Its main task is to read input characters and produce as output a sequence of tokens that parser uses for syntax analysis. To aid the language developer, generating lexers for general purpose programming languages by using the fast lexical analyser generator, JFlex is investigated in this paper. This paper is described how to write a lexical specification file by using the regular expression notations of JFlex. And then, the results of scanner generation for the whole lexical structure of Java and C++ languages are described.*

## 1. Introduction

A compiler is a program that takes as input a program written in the source language and translates it into a functionally equivalent program in the target language.

Although it is focused on constructing a compiler for a programming language, these techniques can be valuable and useful for a wide variety of parsing and translating tasks such as translating javadoc comments to HTML, generating a table from the results of an SQL query, parsing word document into PDF format, and using for web indexing service.

There are two main stages in the compiling process, analysis and synthesis. The analysis stage breaks up the source program into pieces, and creates a language-independent intermediate representation of the program. Then, the synthesis stage constructs the desired target program from the intermediate representation.

Typically, a compiler's analysis stage is called its front-end and the syntheses stage its back-end. Each of the stages is broken down into a set of phases that handle different parts of the tasks. One of the analysis phases is lexical analysis.

## 2. Related Work

Most general-purpose programming systems include a lexer and parser generator modeled after the design of the LEX and YACC tools from UNIX. To start the development of a new language, a general-purpose programming language or a domain-specific language, one of the first steps is the selection of the tool, lexer generator that will be used to build the language processor [2].

One use of regular expressions that used to be very common was in web search engines. Archie, one of the first search engines, used regular expressions exclusively to search through a database of filenames on public FTP servers [11]. Once the World Wide Web started to take form, the first search engines for it also used regular expressions to search through their indexes. Regular expressions were chosen for these early search engines because of both their power and easy implementation. Also, Google search engine uses C++ tool Flex for web indexing service to get maximum speed [8].

A domain-specific language (DSL) is a specialized and problem-oriented language. Successful application of a DSL largely depends on provided tools, so called language workbenches that support programmers' creating, editing, and maintaining programs written in a DSL.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, and IBM OS systems. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system [8]. Lex is distributed with the UNIX operating system while Flex is a product of the Free Software Foundation, Inc.

Regular expressions are an extremely powerful tool for manipulating text and data. They are now standard features in a wide range of languages and popular tools, including Perl, Python, Ruby, Java, VB.NET and C#, PHP, and MySQL [4].

## 3. Lexical Analysis

The lexical analyzer takes a stream of characters and produces a stream of names, keywords, and punctuation marks; it discards white space and comments between the tokens. The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase or parser.

### 3.1. Tokens in Programming Languages

Lexical analysis performs the decomposition of the input into tokens. A token is usually described by an integer representing the kind of token, possibly together with an attribute, representing the value of the token.

In most programming languages the kinds of tokens can be found that are identifiers, reserved or keywords, integer constants, floating point constants, string constants, character constants, special symbols, comments, compiler directives, line information, and white space.

Each reserved word or special symbol is considered to be a different kind of token, as far as the parser is concerned. They are distinguished by a different integer to represent their kind.

All identifiers are likely to be considered as being the same kind of token, as far as the parser is concerned. Different identifiers have the same kind, and are distinguished by having a different attribute.

All integer constants are considered as being the same kind of token, as far as the parser is concerned. They are distinguished by their value. Similarly, floating point constants, string constants, and character constants will represent three different kinds of token, and will have an attribute representing their value.

For some constants, such as string constants, the translation from the text that makes up the constant. The surrounding quote marks have to be deleted, and escaped characters have to be translated into internal form.

Some tokens, while important for lexical analysis, are irrelevant for the parser such as layout tokens, white space, newlines, and comments are processed by the lexical analyser, and then discarded, since they are ignored by the parser.

### 3.2. Lexical Errors

The lexical analyser must be able to cope with text that may not be lexically valid. The compiler must produce an error message and somehow continue the lexical analysis. It would appear to be relatively easy to correct lexical errors, but it should be pointed out that poor error recovery in the lexical analysis phase can produce spurious errors at the parsing phase. A solution to this is terminating the rest of the compiler if a lexical error occurs, and only continues performing the lexical analysis.

### 3.3. Regular Expressions and Finite Automata

Regular expressions aim to solve a simple problem: defining strings to match other strings. In most programming languages, lexical tokens seem to have a remarkably simple structure that can be described by patterns called regular expressions. They are also used by Lex, Flex, JLex and JFlex, four very similar special purpose computer languages used for writing lexical analysers.

Finite automata can be used to decide if an input string is a member in some particular set of strings. It is a machine that has a finite number of states and finite number of transitions between these. There are two classes of FA: non-deterministic finite automata (NFA) and deterministic finite automata (DFA).

In construction of the lexical analyser generator tool, firstly, it converts regular expressions to NFA, and then translates NFA to an efficient DFA. Some lexer tools consist of the DFA minimization.

The programmer specifies the tokens to match using regular expressions, and the action to perform in a conventional programming language. Lex/Flex are C based. JLex/JFlex are the Java based equivalents. The Lex/Flex or JLex/JFlex compiler generates a C or Java program, which can be combined with other C or Java code. Flex and JFlex more or less represent GNU extended versions of Lex and JLex. JFlex should be used because it is Java based and more sophisticated than JLex. To indicate that matching the simple text, a pattern equal to the text can be used itself.

## 4. Overview of JFlex

Fast lexical analyser generator, JFlex is a source-to-source translator. It is a program that translates source code from one language into another language. It translates its source code into Java. It recognises lexical patterns in text and then breaks input character stream into tokens. It generates a scanner, a java class by using its input file, scanner specification file. It can be ported specification from the other tools such as JLex, Lex and flex. It can work together with the free Java parser generator CUP and also has support for the Java extension BYacc/J parser generator.

There are two provided constructors for the lexical analyser class. The primary one takes a Reader object as a parameter. The secondary one takes an InputStream, which it converts into a Reader and invokes the primary constructor. The parameter

represents an object that provides the input to be lexically analysed.

The lexical analyser class has a method for getting a token. The default name for this method is yylex(), although this can be changed, using the %function directive. The default return type is Yytoken, although this can be changed, using the %type directive.

Both /* ... */ and // style comments are permitted in all parts of a JFlex program. /* ... */ style comments can be nested.

Generally, with some exceptions, JFlex programs can be laid out in free format, without regard to spaces and line breaks.

## 5. The Syntax of JFlex

The following gives a rough indication of the syntax of JFlex. It is a bit more complex than indicated by the grammar, because line breaks and white space are sometimes significant, and sometimes not.

A JFlex program is composed of three sections, separated by "%%" directive, which must occur at the beginning of a line.

```
user code
 %%
 options and declarations
%%
 lexical rules
```

The first section is Java code that is just copied into the Java program to be generated. The second section is composed of a list of macro declarations and directives. The third section is composed of a list of rules. For example,

```
package mylexer;
%%
%public
%type Void
letter = [A-Za-z]
newline = \r|\n|\r\n
%%
{letter}+ { System.out.println( yytext() ); }
{newline} { }
```

is a simple JFlex program that reprints the words that appear in its input, one to a line, and discards the rest of the input.

### 5.1 User Code

To add Java code outside the class declaration to the Java file generated by JFlex, place the code before the first directive. The code is transferred across to the generated Java file, without any analysis by JFlex. If it is not valid Java, the errors will not be detected until the Java compiler is run on the generated Java program. The purpose of this code is to specify the package, and import other packages.

It is possible to add Java code to the class declaration, by enclosing it in %{ ... %} in the directives section. For example, you can declare your own fields and methods in this section.

A rule is followed by Java code enclosed in {...}. This code becomes part of the lexical analyser method, and is executed when the rule is matched.

### 5.2. Directives and Macros

There are lots of directives. Directives generally start with a "%" at the beginning of a line, and are used to specify options such as the name of the class generated to perform lexical analysis.

A macro can be used to name a regular expression. For example,

Identifier = [A-Za-z][A-Za-z0-9]*

and later use "{Identifier}" to represent the pattern "[A-Za-z][A-Za-z0-9]*". Macros expressions can be used in JFlex are shown in the table (1).

Table 1. Regular expressions in JFlex

| | |
|---|---|
| a \| b | Union – match a or b. |
| Ab | concatenation- match a followed by b |
| a* | closure-match zero or more repetitions |
| a+ | iteration- one or more repetitions |
| a? | option- zero or one repetitions |
| !a | Negation- match all but not a |
| ~a | Upto-match everything up to a |
| a{n} | repeat- match n times of a |
| (a) | Match the same input as a |
| .(dot) | Any character except the newline |
| "…" | Verbatim string |
| {name} | Macro expansion |
| (……) | Grouping within regular expressions |
| [……] | class of characters - any one character enclosed in brackets |
| a-b | Range of characters |
| [^….] | negated class – any one not enclosed in brackets |

### 5.3. Lexical Rules

A rule specifies what actions to perform when a regular expression is matched. The lexical rules section of a JFlex specification contains a set of regular expressions, specification of a token pattern

and actions, java codes that are executed when the scanner matches the associated regular expression.

A rule consists of a regular expression pattern and then followed by code for rule's action within curly braces { }. Action usually includes a return statement. The rules written in a JFlex specification file should match all possible inputs.

As the scanner reads its input, it keeps track of all regular expressions and activates the action of the expression that has the longest match. A lexical state acts like a start condition. A start condition of a regular expression can contain more than one lexical state. It is then matched when the lexer is in any of these lexical states. The lexical state YYINITIAL is predefined and is also the state in which the lexer begins scanning. If a regular expression has no start conditions it is matched in all lexical states. Lexical states can be used to further restrict the set of regular expressions that match the current input.

A regular expression can only be matched when its associated set of lexical states includes the currently active lexical state of the scanner or if the set of associated lexical states is empty and the currently active lexical state is inclusive. Exclusive and inclusive states only differ at this point: rules with an empty set of associated states.

## 6. Scanner Implementation and Result

A simple source editor with token coloring feature is implemented to investigate lexical analysis phase of a compiler with lexer generator tool, JFlex. The generated lexers are used to recognize tokens for Java language and C++ language respectively.

The types of token for these languages can be easily classified according to their specified colors in this editor. There are seven colors to show type of token and one for error reporting. They are shown in the following table.

Table 2. Kinds of tokens and their colors

| Token Type | Color |
|---|---|
| Keyword | Blue |
| Identifier | Black |
| Literal | Dark red |
| Separator | Dark blue |
| Operator | Dark magenta |
| Comment | Gray |
| Preprocessor | Dark green |
| Error | Red |

This system is developed by using reusable approach. The generated lexers must implement an interface, Tokenizer and create an abstract class, Token for storing recognized tokens in two classes JavaToken and CPlusToken respectively. The parser like CUP can check syntax easily by using these two classes. So, this will be too useful to extend the system for other languages and compiler's next phase, syntax analysis. This is depicted by a class diagram in the figure 1.

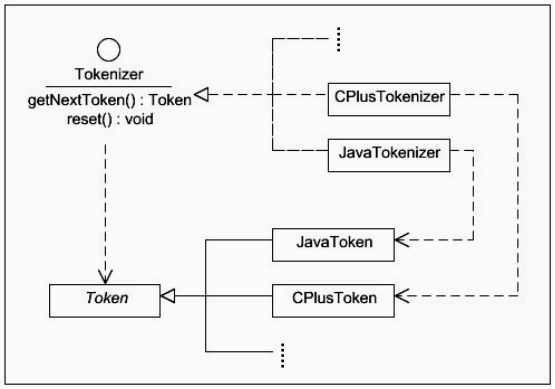


Figure 1. Class diagram

When generating scanners for the whole lexical structure of Java and C++ programming languages, resulting states and time statistics for NFA and DFA construction are shown in Table(3).

Table 3. Scanner generation results

| | C++ | Java |
|---|---|---|
| Generated class name | CplusTokenizer .Java | JavaTokenizer. java |
| NFA construction | 2203 states | 1251 states |
| DFA construction | 868 states | 461 states |
| DFA minimization | 546 states | 362 states |
| Overall generation time | 1s 906ms | 1s 42ms |

## 7. Conclusion

The techniques used to implement lexical analysers can also be applied to other areas such as query languages and information retrieval systems. In each application, the underlying problem is the specification and design of programs that execute actions triggered by patterns in strings.

Several tools have been built for constructing lexical analysers from special purpose notations based on regular expressions. Since pattern-directed programming is widely useful, one of a pattern-action language called JFlex for specifying lexical analysers.

The code generated by JFlex inherits the copyright of the specification it was produced from. Therefore, the generated codes can be used freely in developing open system.

This system can easily extend for other programming languages because it is designed with object-oriented approach. Learners who understand on writing specification file can create scanners for not only general purpose languages but also problem-oriented languages.

A software tool that automates the construction of lexical analysers allows people with different backgrounds to use pattern matching in their own application areas. A major advantage of a lexical analyser generator is that it can utilize the best-known pattern-matching algorithms and thereby create efficient lexical analysers for people who are not experts in pattern-matching techniques.

## 8. References


[1] A. Aaby *Compiler Construction using Flex and Bison*, Walla Walla College, cs.wwc.edu, Version of April 22, 2005

[2] V. Aho, R.Sethi, and J. D. Ullman, *Compilers;Principles, Techniques and Tools*, Addison-Wesley, 1986.

[3] W . Appel and J. Palsberg, *Modern Compiler Implementation in Java, Second Edition*, ISBN: 052182060x, Cambridge University Press, 2002.

[4] J. E. F. Friedl, *Mastering Regular Expressions, Third Edition*, ISBN-10: 0-596-52812-4, O'Reilly, August 2006.

[5] J. Gosling, B. Joy, G. Steele and G. Bracha, *The Java™ Language Specification Third Edition*, ISBN 0-321-24678-0, Addison-Wesley, May 2005.

[6] International Standard - ISO/IEC-14882, *Programming Language-C++, First Edition*, American National Standard Institute, 1998.

[7] G. Klein, *JFlex , Version 1.4.2, The Fast Lexical Analyser Generator*, www.jflex.de.

[8] T. E. Mogensen, *Basic of Compiler Design*, DIKU University of Copenhagen, Denmark, 2007.

[9] T. Niemann , *A Compact Guide to Lex & Yacc*, www.epaperpress.com.

[10] R. Wilhelm and D. Maurer, *Compiler Design, First Edition*, Addison-Wesley, 1996.

[11] www.searchenginehistory.com.