



SYMBIOSIS INSTITUTE OF TECHNOLOGY (SIT)

Constituent of Symbiosis International (Deemed University), Pune

(Established under Section 3 of the UGC Act of 1956 vide notification number F-9-12/2001-U-3 of the Government of India)
Re-Accredited by NAAC with 'A' Grade

SP CASE STUDY ASSIGNMENT

JFLEX

NAME: TRIPTI SINGH

PRN:18070124074

B. TECH IT 18-22

TABLE OF CONTENTS

INTRODUCTION	3
LEXICAL ANALYZER	3
MAIN DESIGN GOALS FOR jFlex:	4
FEATURE HIGHLIGHTS OF jFlex.....	4
LEXICAL SYNTAX OF JFLEX	6
USER CODE.....	6
OPTIONS & DECLARATIONS	6
LEXICAL RULES	8
LIMITATIONS	10
BUILDING COMPILER FROM jFLEX	10
CODE EXAMPLE FOR jFLEX	12
REFERENCES.....	14

INTRODUCTION

There are two complementary compiler construction tools which are traditionally available in the UNIX world:

- one to build lexical analysers (often called 'lexers' or 'scanners') e.g. lex, JLex, JFlex
- one to build syntactic analysers (often called 'parsers') e.g. yacc, bison, CUP

The primary goal of both lexers and parsers differ. Jflex is a lexical analyser or scanner generator for JAVA, written in JAVA. It is a rewrite of JLex ,a very useful tool which was developed by Elliot Berk at Princeton University.

JFlex takes a JFlex program and creates a Java file. The default name for the Java class generated is Yylex, and the code is written to a file called Yylex.java, although this can be changed, using the %class directive. JFlex supports JDK 1.8 or above for build and JDK 7 and above and it is open source released under the BSD licence.

LEXICAL ANALYZER

A lexical analyser takes in a specific input with a set of regular expressions and performs corresponding actions. It generates a program(lexer) which reads input, matches it with regular expression and if matched executes a series of corresponding actions.

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.

Lexers usually are the first front-end step in compilers, matching keywords, comments, operators, etc, and generating an input token stream for parsers. Lexers can also be used for many other purposes. JFlex lexers are based on deterministic finite automata (DFAs). They are fast, without expensive backtracking.

JFlex is designed to work together with the LALR parser generator CUP by Scott Hudson, and the Java modification of Berkeley Yacc BYacc/J by Bob Jamison. It can also be used together with other parser generators like ANTLR or as a standalone tool.

MAIN DESIGN GOALS FOR jFlex:

- ✓ Full unicode support
- ✓ Fast generated scanners
- ✓ Fast scanner generation
- ✓ Convenient specification syntax
- ✓ Platform independence
- ✓ JLex compatibility

FEATURE HIGHLIGHTS OF jFlex

1. Compatibility with JLex

Jflex specification Is based on JLex, it is inclusive of all its features and more.

2. JFlex is fast

Lexers of Jflex are DFA based. Moreover, they avoid expensive backtracking.

3. Regexp, Macros and Predefined character classes

Regular expressions in JFlex are powerful, they support repetition, negation, union, intersection, set difference, symmetric difference, etc.

Macros are abbreviations for regular expressions, they can be used to make lexical specifications easier to read and understand. Cycles in macro definitions are detected and reported at generation time.

4. Unicode and Platform independent

Jflex supports Unicode 7.0 to 12.1 including emoticons.

The end-of-line operator “\$” is platform independent, and detects end-of-line on Windows or Unix automatically.

5. Easy integration with parsers

JFlex has builtin support for several parser generators like CUP, CUP2, BYacc/J, Jay with possibility of ANTLR integration.

6. Diverse uses of Jflex

Via Maven plugin, ant task, bazel rule, standalone binary, it can also be used as a library.

7. Rule dependent

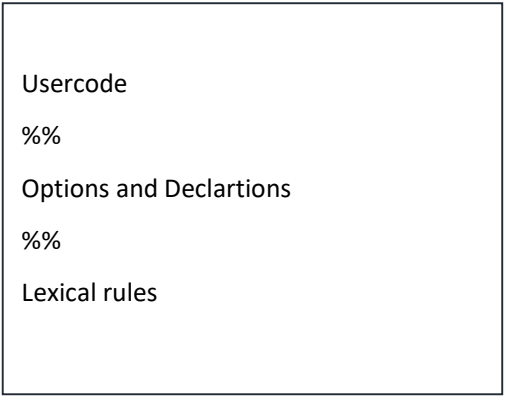
Works on rules and gives expected results with high accuracy rates.

8. Easy Debugging

Jflex has column and line counters with debugging support. It can generate a standalone-lexer, independent of any parser.

LEXICAL SYNTAX OF JFLEX

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



```
Usercode
%%
Options and Declartions
%%
Lexical rules
```

The diagram illustrates the structure of a JFlex program. It is a rectangular box containing three sections separated by double percent signs (%%). The first section is labeled 'Usercode', the second 'Options and Declartions' (note the typo), and the third 'Lexical rules'.

USER CODE

The first part contains user code that is copied verbatim into the beginning of the source file of the generated lexer before the scanner class is declared.

OPTIONS & DECLARATIONS

The second part of the lexical specification contains options to customize your generated lexer (JFlex directives and Java code to include in different parts of the lexer), declarations of lexical states and macro definitions for use in the third section “Lexical rules” of the lexical specification file.

Jflex Directive

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. Each JFlex directive must be situated at the beginning of a line and starts with the % character. Directives that have one or more parameters are described as follows:

`%class "classname"`

means that you start a line with %class followed by a space followed by the name of the class for the generated scanner.

jFlex Macro

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

`Ident = [A-Za-z][A-Za-z0-9]*` and later use “`{Ident}`” to represent the pattern “`[A-Za-z][A-Za-z0-9]*`”

A macro definition has the form **macroidentifier = regular expression**

```
macroList ::=
    macroList macro
    | /* Empty */
    ;

macro ::=
    "Directive"
    | IDENT "=" regExpr "\n"
    ;
```

LEXICAL RULES

A rule specifies what actions to perform when a regular expression is matched. The "lexical rules" section of an JFlex specification contains a set of regular expressions and actions (Java code) that are executed when the scanner matches the associated regular expression.

```
ruleList ::=  
    ruleList rule  
    ;
```

Some highlights on syntactical details:

[1] Jflex Grammar Norms

JFlex applies the following standard operator precedences in regular expression (from highest to lowest):

- unary postfix operators ('*', '+', '?', {n}, {n,m})
- unary prefix operators ('!', '~')
- concatenation (RegExp ::= RegExp Regexp)
- union (RegExp ::= RegExp '|' RegExp)

So the expression `a | abc | !cd*` for instance is parsed as `(a|(abc)) | ((!c)(d*))`.

[2] Jflex Naming Conventions

JFlex follows a naming convention: everything with an underscore after the yy prefix like yy_startRead is to be considered internal. Methods and members of the generated class that do not contain an underscore belong to the API that the scanner class provides to users in action code of the specification. They will remain stable and supported between JFlex releases as long as possible.

[3] Steps To Code In Jflex

- Install JFlex
- Save the specification file , save it as java-lang.flex(optional).
- Run JFlex with `jflex java-lang.flex`
- JFlex should then report some progress messages about generating the scanner and write the generated code to the directory of your specification file.
- Compile the generated .java file and your own classes. (If you use CUP, generate your parser classes first)

[4] Points To Write A Faster Specification In Jflex

1. Avoid rules that require backtracking
2. Avoid line and column counting
3. Avoid lookahead expressions and the end of line operator '\$'
4. Avoid the beginning of line operator '^'
5. Match as much text as possible in a rule.

LIMITATIONS

1. Scanning binaries is both easier and more difficult than scanning text files. It's easier because we want the raw bytes and not their meaning. The problem (for binaries) is that JFlex scanners are designed to work on text. Therefore, the interface is the Reader class (there is a constructor for InputStream instances, but it is just there for convenience and wraps an InputStreamReader around it to get characters, not bytes).

2. There is absolutely NO WARRANTY for JFlex, its code and its documentation.

3. The check, if a lookahead expression is legal, fails on some expressions. The lookahead algorithm itself works as advertised, but JFlex will not report all lookahead expressions that the algorithm cannot handle at generation time. Some cases are caught by the check, but not all.

4. The trailing context algorithm used in JFlex is incorrect. It does not work, when a postfix of the regular expression matches a prefix of the trailing context and the length of the text matched by the expression does not have a fixed size. JFlex will report these cases as errors at generation time.

BUILDING COMPILER FROM JFLEX

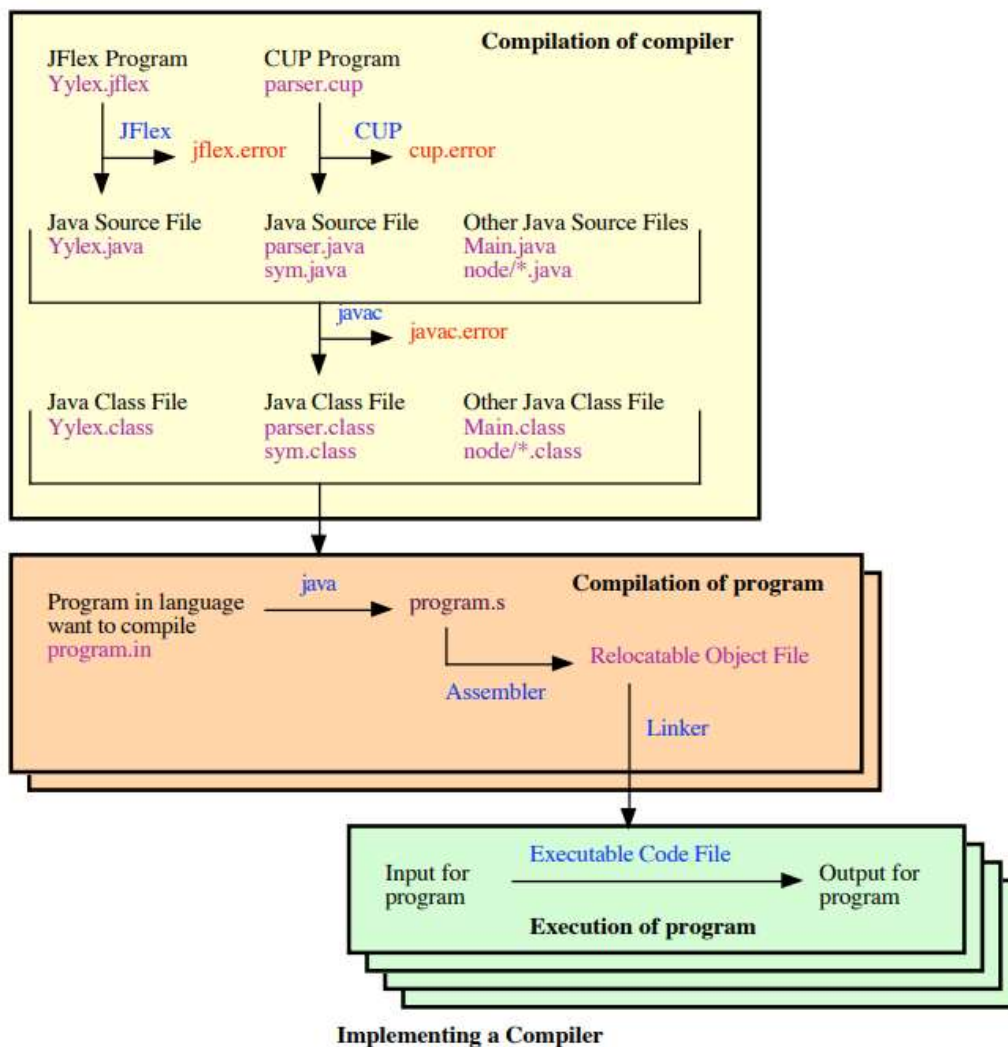
For implementing a compiler, we can write the compiler in a combination of JFlex, CUP, and java.

1. Run the JFlex and CUP compilers on the JFlex and CUP programs to generate Java.

2. Run the Java compiler to generate class files.

3. To compile the desired program, we run the Java interpreter, to interpret the Java class files, which analyse the program we want to compile, written in the language we have implemented, and generate assembly language.

4. We run an assembler to assemble the assembly language and generate a relocatable object file.
5. Run a linker to combine the relocatable object file with library relocatable object files and generate an executable code file.
6. For every input for every program we wish to execute, we run the executable code file to process the input and generate output.



Referenced from : [Microsoft Word - Chapt 01 Lexical Analysis.doc \(auckland.ac.nz\)](#)

CODE EXAMPLE FOR JFLEX

There 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. package grammar.

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

It generates a class,

```
package grammar;  
  
public class Yylex {  
  
    public Yylex( Reader reader ) {...}  
  
    public Yylex( InputStream in ) {...}  
  
    public Void yylex() {...}}
```

Which can be invoked with java code,

```
import grammar.*;  
  
import java.io.*;  
  
public class Main { public static void main( String[] argv ) {  
  
    try { If ( argv.length != 1 )  
  
        throw new Error( "Usage: java Main filename" );
```

```

FileInputStream fileInputStream =
new FileInputStream( argv[ 0 ] );
Yylex lexer = new Yylex( fileInputStream );
lexer.yylex();}

catch ( Exception exception ) {

System.out.println( "Exception in Main " + exception.toString() );

exception.printStackTrace();}

}

}

```

Some points to avoid error in writing Jflex codes:

- ✓ Don't modify the code in Driver.java, SrcLoc.java, or LEXSymbol.java. There should be no need for this project, however, looking over how they work will help in creating your parser and understanding how it works.
- ✓ Make sure you have your class path settings right or give the full path to the Java CUP jar like in the examples given.
- ✓ Make sure you have the java package set up correctly. All class and java files should be in the parser package and in a directory with the same name.
- ✓ Do not ignore warnings from Java CUP, they almost always indicate that something is wrong and needs fixing.
- ✓ If you use the given. jflex file the only file you need to submit is the .cup file, however, you may also include example outputs, test cases and screenshots.
- ✓ If you do not use the given. jflex make sure to include yours and any .java files it uses. Also ensure that you note any changes you made. Your lexer from project 1 may need some changes to work with Java CUP, use the given. jflex files as an example.
- ✓ The grammar given in the project for C- will need changes to work with Java CUP but should be equivalent.

REFERENCES

[Compiler Construction Tools, Part III LG #41 \(tldp.org\)](#)
[Compiler Construction Tools LG #39 \(tldp.org\)](#)
[JFlex - JFlex The Fast Scanner Generator for Java](#)
[HackerDan.com » Blog Archive » Using JFlex](#)
[Microsoft Word - Chapt 01 Lexical Analysis.doc \(auckland.ac.nz\)](#)
[JFlex User's Manual \(auckland.ac.nz\)](#)
[JFlex User's Manual \(unsw.edu.au\)](#)

PLAGIARISM REPORT

RESULTS

100%

Completed: 100% Checked

0%

Plagiarism

100%

Unique

Sentence Wise Result

Document View

Matched Sources

Share Report

Download Report

Start New Search

THANKYOU