IJCSI International Journal of Computer Science Issues, Vol. 7, Issue 3, No. 4, May 2010
ISSN (Online): 1694-0784
ISSN (Print): 1694-0814

Evaluation of JFlex Scanner Generator Using Form Fields Validity Checking

Ezekiel Okike[1] and **Maduka Attamah**[2]

[1] **School of Computer Studies, Kampala International University, Kampala, 256, Uganda**

[2] **Department of Computer Science, University of Ibadan, Ibadan, Oyo State 02 234, Nigeria**

Abstract

Automatic lexical analyzer generators have been available for decades now with implementations targeted for C/C++ languages. The advent of such a tool targeted to the Java language opens up new horizons, not just for the language users but for the Java platform and technology as a whole. Work has also been done by the Java language developers to incorporate much of the theory of lexical analysis into the language. This paper focuses on the application of the JFlex tool (a Java based lexical analyzer) towards the generation of a standalone class that will work as one more entity in a solution space. The study application is a lexer that serves to check the validity of fields in a form.

Keywords: *Lexical Analyzer, Complier Generators, Java Language, Validity Checking*

1. Introduction

A frequently encountered problem in real life application is that of checking the validity of field entries in a form. For example, a form field may require a user to enter a strong password which usually must contain at least one lower case letter, an upper case letter and a digit. If the user fails to enter password matching such specification, the program should respond by alerting the user with appropriate message such as "Your password is not strong".

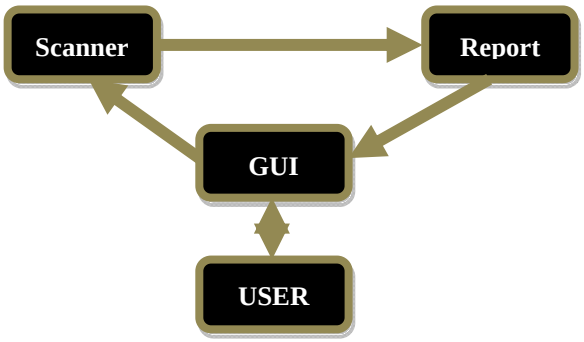

Figure 1: Model of GUI Lexical Analysis for a Web Directory Application

The job of checking the validity of fields in our application thus properly falls to the lexical analyzer. In this case, the Graphical User Interface (GUI) form collects the user inputs, constructs an input string from the input fields and user supplied values, and channels the input string to the scanner. The scanner matches each segment (field) of the input string against a regular expression and reports its observation. The report is thus generated and fed back to the GUI for the user to see. The user is allowed to correct any erroneous field as long as it appears. The model for this application is shown in figure 1 above.

Specifically, the application in question (see figure 2) is a Web Directory Form. Web administrators could use it to organize the different administrative information they may have on the web sites they administer. The web directory form thus forms but one module in a potential software. This form could check the validity of the fields before they are forwarded to the database. The fields checked in this case are as follows:

Web URL, IP Address, E-Mail Address, GSM Phone Number (Global), Strong Password, Security Question, Country, Time Zone.

Thus a regular expression for the above fields in a single lexical specification file generates a single scanner. Using Jflex (Java Flex), a scanner implementation in Java following a given lexical specification could be achieved.

It is also possible to achieve the implementation in other languages such as C or C++ with Lex, or Flex which is a faster version of Lex [3][10].

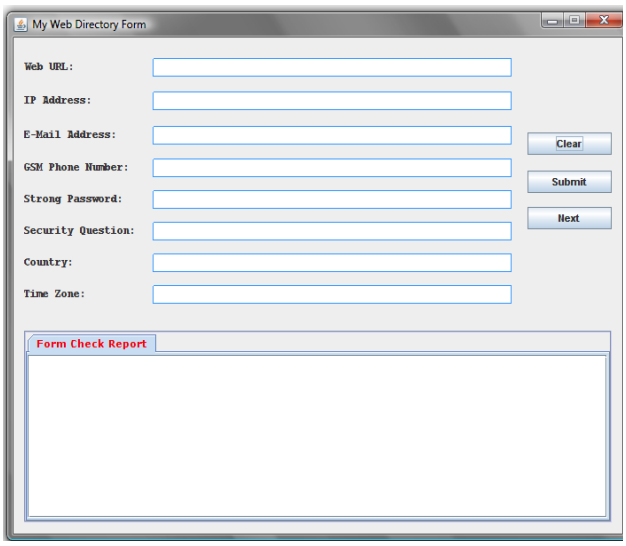

Figure 2: Snapshot of the Web Directory Application

This paper is thus focused on the use of the JFlex tool to generate a scanner for a language defined by the extended regular expression and supported by the JAVA programming language.

2. The Method

The development and evolution of Lex and Flex is described in [7][10]. Originally these tools were developed for the UNIX operating system to generate scanners in the C programming language but afterwards versions for other operating systems and programming languages began to surface. Popular Java versions of lex include JLex and JFlex. Flex can be downloaded along with its documentation at [1] whereas JFlex can be downloaded at [2].

JFlex was written in Java. One of the consequences of this is that in order to use the tool you must have your Java Runtime Environment (JRE) well setup. Other setup procedures for the JFlex tool are also required.

The setup of Jflex involves two major steps, namely: Installation of the JRE [3][4] and installation of JFlex [2][5]. Following appropriate installation and configuration procedures, the screenshot showing that Java is properly configured on the local machine is shown in figure 3, while the screenshot which shows that JFlex tool is well configured is shown in figure 4.

Figure 3: Snapshot That Shows That Java Is Well Configured On The Local Machine

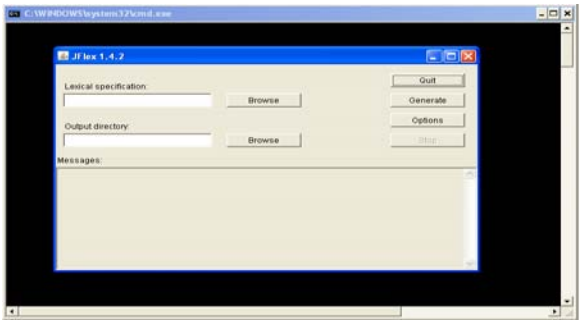

Figure 4: Snapshot Which Shows That Java JFlex Tool is Well Configured and Ready

2.1 The Lexical Specification for JFlex

A lexical specification file was created from a text file (.txt) by changing the extension of the file from ".txt" to ".flex". This implies that a specification file can be edited by simple text editors like notepad.exe or wordpad.exe (both on Microsoft Windows operating system).

2.2 The structure of the lexical specification file

The lexical specification file for JFlex consists of three parts divided by a single line starting with `%%` [8][10]:

```
User code
%%
Options and declarations
%%
Lexical rules
```

2.2.1 User Code

The first part contains user code that is copied literally into the beginning of the source file of the generated scanner, just before the scanner class is declared in the java code. This is the place to insert package declarations and import statements.

2.2.2 Options and Declarations

The second part of the lexical specification contains ***options*** to customize the generated scanner (JFlex directives and Java code to include in different parts of the scanner), declarations of ***lexical states*** and ***macro definitions*** for use in the third section (***Lexical Rules***). Each JFlex directive must be placed at the beginning of a line starting with the `%` character.

2.2.3 Lexical Rules

This section is introduced by another set of `'%%'` characters on a new line after all the intended macro definitions. It contains a set of regular expressions and actions (Java code) that are executed when the scanner matches the associated regular expression [5].

The lexical rules are also called transition rules. An example is shown below:

```
{EXPR} {
       System.out.print("%s", "Valid > " +
       yytext());
}
```

The above example means that if you match an expression, display the expression such that the string `Valid >` e*xpression* is on the output screen. Thus the transition or lexical rule is what happens when a pattern, defined by a regular expression is matched.

2.3 Lexical Specification for the Web Directory Scanner

The file containing the lexical specifications for this study is `WebDirectoryScanner.flex`. The ***user code*** section of this file is empty as there was no need for the package statement and the import statement. The classes were made standalone and simply copied to an application directory. Thus the specification file begins as follows:

```
%%
%public
%class WebDirectoryScanner
%standalone
%unicode
```

To make the generated class accessible by other classes in separate applications, the `%public` option was used. The `%class` option specifies `WebDirectoryScanner` as the name of the generated scanner class. The `%standalone` option was used because the scanner would not be plugged into a parser. The scanner is also required to throw out token information on the console (at the background, since the main view port for the user is the Graphical User Interface) to be used as debugging information.

To ensure the character set constituting the terminals of the scanner is sufficient to consider any special character, the unicode was formally activated. This is especially important if the **.**(dot) character class is used in some of the regular expressions. The 'dot' character class refers to all characters.

Class variables were not declared, hence the `%{` (and the counterpart `%}`) was not necessary.

2.4 Macro Definitions

These are included in the second section, usually after the options. A macro definition has the form

macroidentifier = regular expression

A macro definition is a macro identifier (letter followed by a sequence of letters, digits or underscores), that can later be used to reference the macro. This is followed by optional white-space, followed by an "=", followed by optional white-space, followed by a regular expression. The regular expression on the right hand side must be well formed and must not contain the ^, / or $ operators [5][8][9].

2.4.1 The Macros of WebDirectoryScanner.flex File

Macros were defined for the Web URL, IP Address, E-Mail Address, GSM Number, Strong Password, Security Question, Country and Time Zone.

3. Analysis of the Lexical Specification for the Web Directory Scanner

The breakdown of the associated macros is as follows:

3.1 Web URL

The URL (Universal Resource Locator) must point to a web page or a definite resource to be valid. The following is a listing of the web URL module from the lex specification file.

```
/*****WebURL Module ****************/
WEBURL = ((http:\/\/)?www|(https:\/\/)
?www|(ftp:\/\/)?ftp|(tftp:\/\/)?tftp)
(\.)(.)+(\.)[a-z]{2,6}
```

To the left of the assignment (=) lies the name of the macro (WEBURL), to the right lies the regular expression defining this macro.

This regular expression defines all strings beginning with an optional “http://www.”, “https://www.”, “ftp://ftp.” or “tftp://tftp.”, then followed by any string of one or more length, followed by another dot and terminating with two to six characters. Some of the strings in this language include:

www.yahoo.com
https://www.allmp3.net
ftp://ftp.glide.com/personal/readme.html
www.operators.org/review.html

The front slash (`/`) is a meta symbol (meaning that JFlex converts it as one of its own operating symbols). The only way to use this symbol as a terminal in a regular expression is to escape it as follows - `\/` that is, a backslash (the escape character) followed by a front slash. The same applies to the dot “.”, a meta symbol which stands for the universal set of all characters available to the particular flex specification (this universal set is defined with the %unicode, %ascii, etc, in the options section). To use the dot character as a terminal, there is need to escape it. All meta characters must receive the same treatment if they are intended to be used as terminals in the language in question.

The question mark is an optional closure which indicates that its operand may be (once) or not be there at all. The square brackets [] are used to define character sets in shorthand. For example [a-z] which implies one of a or b up to z. The {2, 6} is the range specifier which indicates that its operand (in this case the character set [a-z]) can occur for a minimum of two times and a maximum of six times. The ‘+’ meta symbol indicates iteration (or closure). This means that there can be at least one or more iterations (the empty string is not possible). The ‘*’ meta symbol indicates the Kleene closure which means that the iteration can occur zero or more times up to infinity (the empty string is possible with the Kleene closure).

JFlex also uses the Java format for comments and as such comments can be contained in between `/*` and `*/` (for multiline comments) or after `//` (for single line comments).

3.2 IP Address

The listing is as shown below.

```
/****IP Address Module ***************/
OCTET = ([0-1][0-9][0-9])|
       (2[0-5][0-5])|([0-9][0-9])|[0-9]

IPADDRESS ={OCTET}\.{OCTET}\.{OCTET}\.{OCTET}
```

An IP address has four octets, each containing a number between 0 and 255 inclusive. The OCTET macro defines this range and explained as follows: when the first digit is a zero or one, then the second and third digits can range from 0 through 9. When the first digit is two, then the second and third digits can only range from zero through five. When there are only two digits, then they can both range from zero to nine. When there is only one digit, it can range from zero to nine. The semantics of the `IPADDRESS` macro follows intuitively. Examples of strings in this language include:

127.0.0.0
192.168.5.6
255.255.78.10
10.0.1.11

3.3 E-Mail Address

The listing is as shown below:

```
EMAILADDRESS = [a-z0-9_]+ @[a-z0-9_]+
               (\. [a-z] {2,6} )+
```

An e-mail address is made up of a string consisting of a possible mix of lower case letters, digits, underscore, separated by the @ symbol and ending with a dot and a string of lowercase letters of length ranging from two to six. A closure (i.e. `{2,6}`) is placed after the "`\.[a-z]`" to show that the extension could repeat. Examples of strings in this language include:

pfizer@yahoo.co.uk, james@gmail.com,
bluecollar@37.com, emeks_rex@tim.net

3.4 GSM Number

```
/****** Global GSM Number ********/
GSMNUMBER = (\+[0-9]{3}|[0-9])[0-9]{10}
```

A regular expression that can match any GSM (Global System for Mobile Communication) number was implemented. GSM numbers usually comprise of ten to eleven digits (or twelve to thirteen digits if country code is included).

3.5 Strong Password

The definition of a strong password requires a string which contains at least one lowercase letter, one uppercase letter and one digit. A strong password module is shown below for `FLAGONE`, `FLAGTWO` and `FLAGTHREE` respectively. Possible strings in this language would include:

234loRd, socK3vbn*, YmeL01d, shu^b5b1

```
/*****  Strong Password Module  *****/
FLAGONE = [a-z]
FLAGTWO = [A-Z]
FLAGTHREE = [0-9]

STRONGPWD = (.)*{FLAGONE}+(.)*
{FLAGTWO}+(.)
*{FLAGTHREE}+(.)*|(.)*{FLAGONE}
+(.)*{FLAGTHREE}+(.)*{FLAGTWO}+
(.)*|(.)*{FLAGTWO}+(.)*{FLAGONE}
+(.)*{FLAGTHREE}+(.)*|(.)*{FLAGTWO}
+(.)*{FLAGTHREE}+(.)*{FLAGONE}+(.)
*|(.)*{FLAGTHREE}+(.)*{FLAGTWO}+(.)
*{FLAGONE}+(.)*|(.)*{FLAGTHREE}+
```

3.6 Security Question

A string is considered to be a question when it ends with a question mark. A regular expression for this is as follows:

```
SECURITYQUESTION = (.)+\?
```

3.7 Country

The only requirement for a country name in the regular expression is that it contains only characters from A through Z whether uppercase or lowercase. The listing is as follows.

```
/*..Country Module..........*/
COUNTRY = [a-zA-Z ]+
```

Since the language is English, special characters involving modifiers like dieresis would not be accepted (at least in this version). The space character is included in the character range**.**

3.8 Time Zone

```
/*.........Time Zone Module.............*/
TIMEZONE = ((g|G)(m|M)(t|T))[ ]
*((\+|-)[ ]*([1-9]|10|11|12):(00|30|45)|
(\+13:00))?[ ]*([a-zA-Z]| [ ]|,|;)+
```

The time zone whose regular expression listing is shown above expects a string beginning with "gmt" (the case mixtures does not matter), followed by an optional space or spaces, followed by an optional plus or minus sign, space or spaces, and a number ranging from one through twelve. Additional +13 is added for completeness (the time zone dating is not symmetrical). Finally the string ends with a country or region name with possible commas or semi-colon as defined by the following part of the regular expression: `([a-zA-Z]|[ ]|,|;)+`

The pair of square brackets with a space in between [ ] is the syntax for adding the space character to our regular expression.

3.9 Analysis of the Lexical Rules for the Web Directory Scanner

The lexical or transition rules section forms the heart of the real usefulness of this scanner. For any of the patterns matched, the lexical rule updates a custom class (`Report.java`) which keeps a record of whether each field is valid or not. The report class has one class variable (each is a boolean type) for each field in the web directory form. The class variable declarations in `Report.java` are shown below:

```
public static String report = "";
public static boolean webUrlOk = false;
public static boolean ipAddressOk = false;
public static boolean emailAddressOk = false;
public static boolean gsmPhoneNoOk = false;
public static boolean strongPwdOk = false;
public static boolean securityQuestionOk =
     false;
public static boolean countryOk = false;
public static boolean timeZoneOk = false;
public static String SATISFACTORY =
"All Fields Are Satisfactory!";
```

From all the fields in the form, a single string is constructed and passed to the scanner for analysis.

The string is constructed as follows:

Each value supplied by the user is concatenated to a string which stands for the title (or identification) for the field in which the entry was made. For example if the user enters mathi@yahoo.com in the E-Mail Address field, then the string “mathi@yahoo.com” is concatenated to "Email Address:" and terminated by a “\n”, to show the end of that segment or field. And this yields "Email Address:mathi@yahoo.com\n" as the resulting string. For the form shown below (figure 5), the string passed to the scanner would be

```
"Web Url:http://www.google.com\n
IP Address:10.0.11.98\nEmail
Address:gmailadmin@gmail.com\n
GSMNumber:+8938097865342\nStrong
Password:goGetM1cors0ft\nSecurity
Question:How are you today?\n
Country:Moscow\nTimeZone:GMT +
3:00 St. Petersburg, Volgograd\n”
```

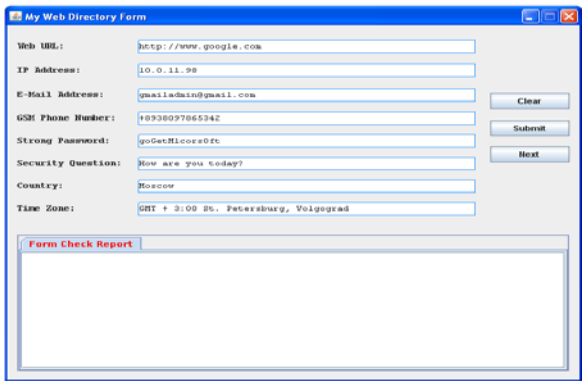

Figure 5: Snapshot of Web Directory Application under Testing

“\n” is the escape character for new line. When the scanner sees a new line it stops trying to recognize the pattern and compares what it already has to see if it is sufficient.

4. Discussion

Consider the transition rules below:

```
%%
"Web Url:"{WEBURL}"\n"
{Report.webUrlOk = true;}
"IP Address:"{IPADDRESS}"\n"
 {Report.ipAddressOk = true;}
"Email Address:"{EMAILADDRESS}
"\n" {Report.emailAddressOk = true;}
"GSM Number:"{GSMNUMBER}"\n"
{Report.gsmPhoneNoOk = true;}
"Strong Password:"{STRONGPWD}"\n"
 {Report.strongPwdOk = true;}
"Security Question:"{SECURITYQUESTION}
"\n" {Report.securityQuestionOk = true;}
"Country:"{COUNTRY}"\n"
{Report.countryOk = true;}
"TimeZone:"{TIMEZONE}"\n"
{Report.timeZoneOk = true;}
```

When the scanner ‘sees’ the constant string “Web URL” followed by another string that matches the pattern for WebURL `{WEBURL}`, and a newline character, then there is a valid WebURL field. The webURL field of the Report class (`Report.webUrlOk`) will be set to true – indicating that that field is valid. This happens to the rest of the string. At the end of the string (`zzAtEOF`) (see also 4.1), all the fields of the Report class are inspected, and if any of them is still false (they are false by default), then the scanner assumes that there is an error with the particular field. An error message tied to the field or fields is displayed.

The following code snippet compiles the report string to be displayed on the Form Check Report pane.

```
if(!webUrlOk)
   report = report +
"Invalid Web URL field\n";
```

If the value of `webUrlOk` is false, the report string should be filled-in so as to reflect the situation. And the process goes on as in the listing below.

```
public static String getReport(){

 report = ""; // clearing the string first of
initial values

if(!webUrlOk)
 report = report + "Invalid Web URL field\n";

 if(!ipAddressOk)
   report = report + "Invalid IP Address field\n";

 if(!emailAddressOk)
   report = report + "Invalid E-Mail Address
field\n";

 if(!gsmPhoneNoOk)
   report = report + "Invalid GSM Number field\n";
 if(!strongPwdOk)
   report = report + "Your Password Is Not Strong.
Have At Least One Digit, One Uppercase
          And One Lowercase Letter\n";
 if(!securityQuestionOk)
   report = report + "Your Question Is Not Complete,
Check It\n";
 if(!countryOk)
   report = report + "Invalid Country Name field\n";
 if(!timeZoneOk)
   report = report + "Invalid Time Zone field\n";
 if(report == "")
  report = report + SATISFACTORY;

 // reset all fields before exiting

  webUrlOk = false;
  ipAddressOk = false;
  emailAddressOk = false;
  gsmPhoneNoOk = false;
  strongPwdOk = false;
  securityQuestionOk = false;
  countryOk = false;
  timeZoneOk = false;
 return report;

}
```

4.1 The Scanning Process

The following code, embedded as the action code for clicking button "Submit" on the graphical user interface, starts off, sustains and terminates the process of scanning.

```
WebDirectoryScanner scanner = new
WebDirectoryScanner(new StringReader
(this.userInputs));
  try {
     while ( !scanner.zzAtEOF )
        scanner.yylex();
     } catch (IOException ex) {
        ex.printStackTrace();
  }
this.reportTextArea.
setText(Report.getReport());
```

The string passed to the scanner is assigned to a string object `userInputs` (a class variable of class `WebDirectory`). For the scanner class `WebDirectoryScanner`, its constructor takes an object of `InputReader` class. Since `StringReader` is a subclass of `InputReader`, it can substitute it for the more general `InputReader` since input to the scanner is just contained in a string. The object of `WebDirectoryScanner` created is `scanner`.

`Scanner.yylex()` is the method for actual scanning. When the end-of-file is reached, the state of the scanner, `scanner.zzAtEOF` is set to `true` to denote that the end of the string has been reached. At this point it stops scanning and displays the content of the Report class found on the lower part of the application window. This is achieved by calling:

```
this.reportTextArea.setText(
      Report.getReport());
```

4.2 Generating the Scanner (WebDirectoryScanner.java)

To generate the scanner class, the JFLex tool is executed by pointing to the file `WebDirectoryScanner.flex` as in figure 6.

Clicking the "Generate" button will display the following, at the same time as it drops the generated file in the folder indicated in the output directory. If errors were encountered, the source of the errors will instead be indicated on the Messages pane, and the user would go back and correct the ill-formed lexical specification.

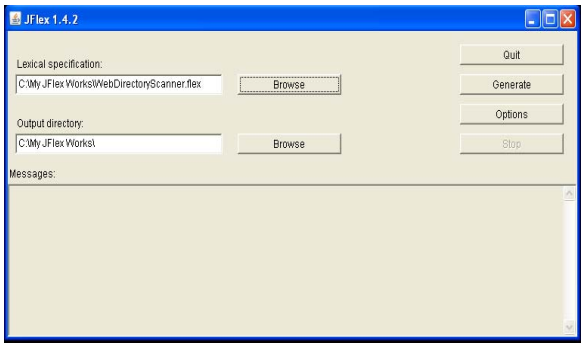

Figure 6: Snapshot of JFlex at Code Generation Time

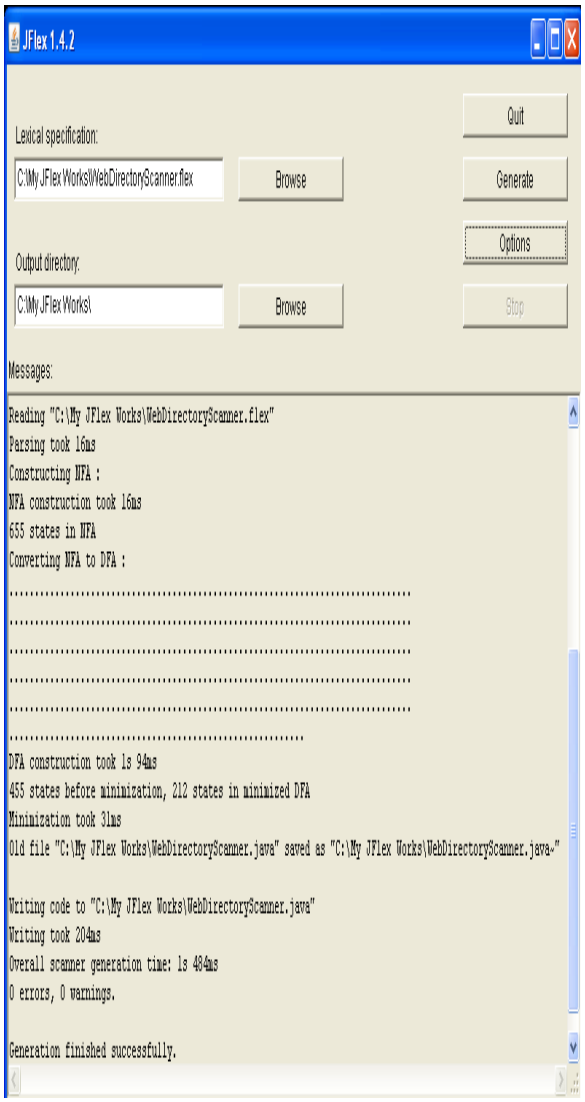

Figure 7: Snapshot Showing JFlex Messages after Generation of Code

JFlex generates exactly one file containing one class from the specification (except another class in the first section of the lexical specification is declared).

The generated class contains (among other things) the DFA tables, an input buffer, the lexical states of the specification, a constructor, and the scanning method with the user supplied actions [5].

The message pane shows that the DFA has a total of 455 states before minimization and a total of 212 states after minimization. This is over 100% efficiency! The class Yylex by default could be customizable with the `%class` directive. The input buffer of the scanner is connected with an input stream via the `java.io.Reader` object which is passed to the scanner in the generated constructor.

The main interface of the scanner to the outside world is the generated scanning method (its default name `yylex()`, and its default return type `yytoken()`. Most of the generated class attributes are customizable (name, return type, declared exceptions etc.). If `yylex()` is called, it will "consume" input until one of the expressions in the specification is matched or an error occurs. If an expression is matched, the corresponding action is executed immediately. It may return a value of the specified return type (in which case the scanning method returns with this value), or if it does not return a value, the scanner resumes consuming input until the next expression is matched. When the end of file is reached, the scanner executes the EOF action, and (also upon each further call to the scanning method) returns the specified EOF value. Details of the scanner methods and fields accessible in the actions API can be found in [5].

4.3 Compiling the Generated Scanner

The generated scanner class is compiled using the Java compiler.

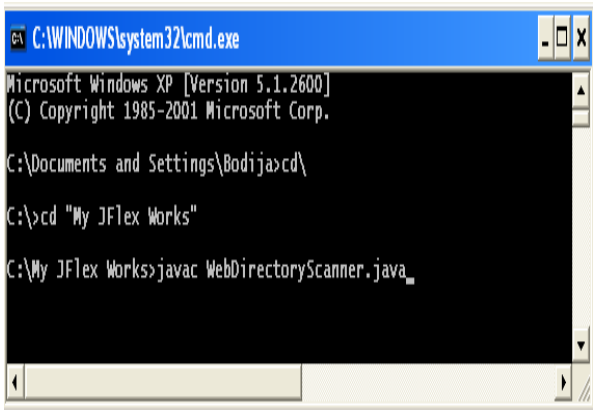

Figure 8: Screenshot Showing the Command for Compiling the Generated Code

Pressing the "Enter" key in the command line will produce the file "`WebDirectoryScanner.class`" which the Java Virtual Machine can then execute – this assumes there were no errors in the codes otherwise debugging would be done first.

Other class files needed for this case study were `Report.class` and `WebDirectory.class`. The entry point (Main class) to the entire application is in `WebDirectory.class`. Hence, this class was passed to the Java Virtual Machine at the command line during start of execution

4.4 Evaluation of JFlex tool

4.4.1 Some Limitations of JFlex 1.4.2

Some of the observed limitations of Jflex from this study are as follows.

There are currently no provisions for assertions. Assertions form part of the theory of the extended regular expression. Some assertions are:

- Lookahead assertion (?=)
- Negative lookahead (?!)
- Lookbehind assertion (?<=)
- Negative lookbehind (?!= or ?<!)
- Once-only subexpression (?>)
- Condition [if then] ?()
- Condition [if then else] ?()|

With facilities like this, some regular expressions would have been possible or easier to write.
An example is the regular expression for strong password which could have been written as:

```
(?=.*\[0-9])(?=.*[a-z])(?=.*[A-Z])
```

Using the lookahead assertion, it could have easily tracked the occurrence of digits, lowercase and uppercase letters and even special symbols. To add another character flag to our own version of regular expression using facilities provided by JFlex would have been very tedious. Facilities for lookahead assertions should be included in later versions of JFlex.

As a sequel to the above, this study found that trying to set the range for the length of the password field using the range specifier {x,y}, the JFlex tool was not terminating during code generation, and the processes of constructing the DFAs was taking an infinite time. Compared to the usual total generating time of about 500ms, JFlex did not finish even after 25 minutes. This happened after several trials until it was discovered that the range specifier {8, 20}

(which says that the password should be at least 8 characters in length and at most 20 characters) was the cause of the non termination. Hence, the range specifier was not useful in this case. JFlex also lacks the special range specifier `{x,}` which indicates at least x, but no upper limit. JFlex should therefore be optimized in subsequent versions so that the range specifier can be used in a practical situation.

The way the 'space' character was implemented in JFlex is truly not the best. The idea of defining a space character using [ ] is very obscure especially when it lies side-by-side other characters. For example `[a-zA-Z ]` (used in the COUNTRY macro). The space character following the capital Z is not very clear. The use of escape characters like \s or the posix `[:space:]` for the space character might be a better approach.

JFlex does not check the Java code included in the user code section and in the lexical rules for correctness. It is relatively straight forward to plug-in the Java Compiler (javac) into the jflex tool so as to do a static check on the java codes, routing the error messages to JFlex display so that the user can also correct the Java codes earlier on, instead of waiting for a separate session, at run time to start the debugging process. We recommend the integration of a lightweight debugger into JFlex.

The only uses well foreseen and provided for by the creators of the JFlex tools have been for either standalone scenarios (where the scanner can be run as a complete and separate application) or a scenario where the scanner is used by an overlying parser (in this case, the scanning method returns a token periodically to the parser). In our present use, the scanner has to be content with just being one of the utility classes in the entire application. The best bet is for it to standalone but then the field of the scanner that indicates that it has reached end-of-file (`zzAtEOF`) is a private member of the scanner class. And JFlex does not generate accessor methods for this field. Creating a non-standalone scanner even makes matters worse because the scanner will be expecting to scan, get a token, return the token and then stop (waiting for a call by a parser). The class acting as the parser will have to look out for a predetermined integer in the returned tokens which denote an end-of-file from the scanner. In our application, we desired that the scanner scans through the entire input stream before returning. And the calling class is not a parser!

The generated scanner had to be modified for this study application. An accessor method for the `zzAtEOF` field was implemented so as to be able to track the end of file (EOF) as shown in the following code snippet.

```
WebDirectoryScanner scanner =
new WebDirectoryScanner(new
StringReader(this.userInputs));
    try {
      while ( !scanner.getZzAtEOF() )
      scanner.yylex();
      } catch (IOException ex)
```

The while statement will continue to execute the scanning method `yylex()` while the end-of-file is not reached (EOF is known when `scanner.getZzAtEOF()` method returns `true`). We recommend that JFlex should generate public accessors for all the scanner fields so that applications that are neither standalone nor parser compatible can more easily utilize the scanner.

References

[1] http://www.gnu.org/software/flex/
[2] http://www.jflex.de/
[3] http://java.sun.com/javase/6/download.jsp
[4] http://java.sun.com/javase/6/webnotes/install/
[5] JFlex User's Manual, Version 1.4.2, May 27, 2008 – JFlex Installation Pack Accompanying Manual
[6] The Regular Expression Cheat Sheet From *www.ILoveJackDaniels.com*
[7] Compilers: Principles, Techniques and Tools, Alfred V. Aho et. al.; 2nd Ed., Pearson Education Inc., 2007.
[8] S. Chattopadhayay. "Compiler Design". New Delhi: Prentice- Hall, xvii +225pp; 2005
[9] K. D. Cooper and L. Torczon. "Engineering a Compiler". New York: MK, xxx + 801pp, 2008
[10] J. Levine. "Flex and Bison – text processing tools". New York: O'Reilly Media, pp304, 2009.

Ezekiel U. Okike received the B.Sc. degree in Computer Science from the University of Ibadan Nigeria in 1992, the Master of Information Science (MInfSc) in 1995 and PhD in Computer Science in 2007 all from the same University. He has been a lecturer in the Department of Computer Science, University of Ibadan since 1999 till date. Since September, 2008 to date, he has been on leave as a Senior Lecturer and Dean of the School of Computer Studies, Kampala International University, Uganda. His current research interests are in the areas of Software Engineering, Software Metrics, Compilers and Programming Languages. He is a member of IEEE Computer and Communication societies.

Maduka Attamah is currently a Researcher and Associate Lecturer in the Department of Computer Science, University of Ibadan. He is also serving in the ICT Unit of University of Ibadan as Systems Analyst and Software Engineer. He holds a B.Eng. (2006) in Electronic Engineering from the University of Nigeria, Nsukka and an M.Sc. (2009) in Computer Science from the University of Ibadan, Nigeria.