Appendix 2 A Grammar for Regular Expressions

(Refer GREP)

The following program takes a regular expression, and a file name, and searches the file for all lines that match the regular expression. Only entire line are matched, not partial lines. Thus it implements something similar to grep with the -x option. The matching algorithm is highly recursive, and very inefficient.

This example shows that it is possible to use JFlex and CUP to implement our own version of JFlex. All versions of JFlex itself, (apart from the very first version) could have been written in this way, although in fact it is not.

Shell Scripts

To compile the JFlex code:

```
#! /bin/bash

rm -f Source/grammar/Yylex.java jflex.error

CJFLEXJAR=`toNative.bash "$LIB330/JFlex.jar"`
CYYLEX=`toNative.bash "Source/grammar/Yylex.jflex"`
java -jar "$CJFLEXJAR" "$CYYLEX" &> jflex.error
```

To compile the CUP code:

```
#! /bin/bash
rm -f Source/grammar/parser.java Source/grammar/sym.java parser.states
cup.error

CUPINPUT=`toNative.bash "Source/grammar/parser.cup"`
CCUPJAR=`toNative.bash "$LIB330/java_cup.jar"`
STATES=`toNative.bash "parser.states"`

#    java -jar "$CCUPJAR" -nonterms \
#    -expect 0 -progress -dump -dumpto "$STATES" -input "$CUPINPUT" &>
cup.error

java -jar "$CCUPJAR" -nonterms \
-expect 0 -progress -input "$CUPINPUT" &> cup.error

if [ -e parser.java -a -e sym.java ]
then
    mv parser.java sym.java Source/grammar
fi
```

To compile the Java code:

```
#! /bin/bash

rm -rf Classes/*
if [ ! -e Classes ]
then
    mkdir Classes
fi

CCUPJAR=`toNative.bash "$LIB330/java_cup_runtime.jar"`
CMAIN=`toNative.bash "Source/Main.java"`
javac -d Classes -classpath "$CCUPJAR" -sourcepath "Source" "$CMAIN" &>
javac.error
```

To run the resultant program, we have the shell script myGrep:

```
#! /bin/bash

INPUT=`toNative.bash "$2"`
CCUPJAR=`toNative.bash "$LIB330/java_cup_runtime.jar"`

java -classpath "run.jar$CPSEP$CCUPJAR" Main "$1" "$INPUT"
```

The lexical Analyser

```
package grammar;

import java.io.*;
import java_cup.runtime.*;
import node.*;

%%

%public
%type         Symbol
%char

%{
     public Symbol token( int tokenType ) {
          return new Symbol( tokenType,
               yychar, yychar + yytext().length(), "" );
          }

     public Symbol token( int tokenType, Object value ) {
          return new Symbol( tokenType,
               yychar, yychar + yytext().length(), value );
          }
%}

%init{
     yybegin( NORMAL );
%init}

%eofval{
     return token( sym.EOF );
%eofval}

escchar   =    (\\[^0-3xX]|\\[0-3][0-7][0-7]|\\[xX][0-9a-fA-F][0-9a-fA-F])

cchar     =    ([^\"\\\r\n\$\(\)\|\?\+\*\.\[\]]|{escchar})
ichar     =    ([^\^\\\r\n\[\]\-]|{escchar})
schar     =    ([^\"\\\r\n]|{escchar})

%state NORMAL INSIDESQ

%%

<NORMAL>"("                 { return token( sym.LEFT ); }
<NORMAL>")"                 { return token( sym.RIGHT ); }
<NORMAL>"|"                 { return token( sym.OR ); }
<NORMAL>"?"                 { return token( sym.QUEST ); }
<NORMAL>"+"                 { return token( sym.PLUS ); }
<NORMAL>"*"                 { return token( sym.STAR ); }
<NORMAL>"."                 { return token( sym.DOT ); }

<NORMAL>"["                 {
                            yybegin( INSIDESQ );
```

```
                              return token( sym.LEFTSQ );
                              }
<NORMAL>\"{schar}*\"               {
                              String s = yytext();
                              return token( sym.STRING,  Text.parseString(
                                   s.substring( 1, s.length() - 1 ) ) );
                              }
<NORMAL>{cchar}                    {
                              String s = yytext();
                              return token( sym.CHAR, Text.parseChar( s ) );
                              }
<INSIDESQ>"]"                 {
                              yybegin(  NORMAL );
                              return token( sym.RIGHTSQ );
                              }
<INSIDESQ>"^"                 { return token( sym.CARET ); }
<INSIDESQ>"-"                 { return token( sym.MINUS ); }
<INSIDESQ>{ichar}             {
                              String s = yytext();
                              return token( sym.CHAR, Text.parseChar( s ) );
                              }
<NORMAL>.|\n                  { return token( sym.error ); }
<INSIDESQ>.|\n                { return token( sym.error ); }
```

Lexical Support Code

We need a support class, to convert escape sequences to control characters, and back again.

```
package node;

public class Text {
     public static String toString( char value ) {
          switch ( value ) {
               case '\b':
                    return "\\b";
               case '\f':
                    return "\\f";
               case '\n':
                    return "\\n";
               case '\r':
                    return "\\r";
               case '\t':
                    return "\\t";
               case ' ':
                    return "\\ ";
               case '(':
               case ')':
               case '|':
               case '?':
               case '+':
               case '*':
               case '.':
               case '[':
               case ']':
               case '^':
               case '"':
               case '-':
               case '\\':
                    return "\\" + value;
               default:
                    if ( value < 32 || value >= 127 )
                         return "\\" + toOctalString( value );
                    else
```

```
                    return "" + value;
               }
          }

     public static String toString( String s ) {
          String result = "";
          for ( int i = 0; i < s.length(); i++ )
               result += toString( s.charAt( i ) );
          return result;
          }

     public static String toOctalString( char value ) {
          String result = "";
          result = value % 8 + result;
          value /= 8;
          result = value % 8 + result;
          value /= 8;
          result = value % 8 + result;
          return result;
          }

     public static String parseString( String input ) {
          String output = "";
          int i = 0;
          char inputChar;
          char outputChar;
          while ( i < input.length() ) {
               inputChar = input.charAt( i );
               if ( inputChar == '\\' ) {
                    i++;
                    if ( i >= input.length() )
                         throw new Error( "Invalid Text String" );
                    inputChar = input.charAt( i );
                    switch ( inputChar ) {
                         case 'b':
                              output += '\b';
                              i++;
                              break;
                         case 'f':
                              output += '\f';
                              i++;
                              break;
                         case 'n':
                              output += '\n';
                              i++;
                              break;
                         case 'r':
                              output += '\r';
                              i++;
                              break;
                         case 't':
                              output += '\t';
                              i++;
                              break;
                         case 'x':
                         case 'X':
                              {
                              outputChar = 0;
                              i++;
                              for ( int j = 0;
                                   i < input.length() && j < 2; j ++ ) {
```

```
                              inputChar = input.charAt( i );
                              if ( inputChar >= '0'
                                   && inputChar <= '9' ) {
                                   outputChar = ( char ) ( outputChar
                                        * 0x10 + inputChar - '0' );
                                   i++;
                                   }
                              else if ( inputChar >= 'a'
                                   && inputChar <= 'f' ) {
                                   outputChar = ( char ) ( outputChar
                                        * 0x10 + inputChar - 'a' + 0xa );
                                   i++;
                                   }
                              else if ( inputChar >= 'A'
                                   && inputChar <= 'F' ) {
                                   outputChar = ( char ) ( outputChar
                                        * 0x10 + inputChar - 'A' + 0xa );
                                   i++;
                                   }
                              else
                                   break;
                              }
                         output += outputChar;
                         } // end while loop
                         break;
                    case '0':
                    case '1':
                    case '2':
                    case '3':
                    case '4':
                    case '5':
                    case '6':
                    case '7':
                         {
                         outputChar = 0;
                         for ( int j = 0; i < input.length()
                              && j < 3; j ++ ) {
                              inputChar = input.charAt( i );
                              if ( inputChar >= '0'
                                   && inputChar <= '7' ) {
                                   outputChar = ( char ) ( outputChar
                                        * 010 + inputChar - '0' );
                                   i++;
                                   }
                              else
                                   break;
                              } // end while loop
                         output += outputChar;
                         }
                         break;
                    default:
                         output += inputChar;
                         i++;
                    } // end cases
               } // end then
          else {
               output += inputChar;
               i++;
               }
          } // end while loop
     return output;
```

```
          }

     public static Character parseChar( String input ) {
          String output = parseString( input );
          if ( output.length() == 1 )
               return new Character( output.charAt( 0 ) );
          else
               throw new Error( "Invalid Text String" );
          }
     }
```

The parser

The constructor for the parser takes a pattern as a parameter, and builds a StringReader, which it passes to the constructor for the lexical analyser.

```
package grammar;

import java_cup.runtime.*;
import node.*;
import node.exprNode.*;
import node.exprNode.binaryNode.*;
import node.exprNode.postfixNode.*;
import node.exprNode.simpleNode.*;
import node.setComponentNode.*;
import java.io.*;

parser code
     {:

     private Yylex lexer;
     private String text;

     public parser( String text ) {
          this();
          this.text = text;
          lexer = new Yylex( new StringReader( text ));
          }

     public String terminal_name( int id ) {
          return sym.terminal_name( id );
          }

     public String non_terminal_name( int id ) {
          return sym.non_terminal_name( id );
          }

     public String rule_name( int id ) {
          return sym.rule_name( id );
          }

     public void report_error( String message, Object info ) {
          System.err.println( message );
          if ( info instanceof Symbol ) {
               Symbol symbol = ( Symbol ) info;
               System.err.println( text );
               for ( int i = 0; i < symbol.left; i++ )
                    System.err.print( " " );
               for ( int i = symbol.left; i < symbol.right; i++ )
                    System.err.print( "^" );
               System.err.println();
               }
```

```
          else
               System.err.println();
          }

     :};

scan with
     {:
     return lexer.yylex();
     :};

/* Terminal Symbols */
terminal Character
     CHAR;
terminal String
     STRING;
terminal
     CARET, LEFT, RIGHT, OR, QUEST, PLUS, STAR, DOT,
     LEFTSQ, RIGHTSQ, MINUS;

/* Nonterminals */

nonterminal ExprNode
     RegularExpr, OrExpr, ConcatExpr, SequenceExpr, SimpleExpr;
nonterminal SetComponentNode
     Set, Element;

start with RegularExpr;

RegularExpr::=
          OrExpr:expr
          {:
          RESULT = expr;
          :}
     ;

OrExpr::=
          OrExpr:expr1 OR ConcatExpr:expr2
          {:
          RESULT = new OrNode( expr1, expr2 );
          :}
     |
          ConcatExpr:expr
          {:
          RESULT = expr;
          :}
     ;

ConcatExpr::=
          ConcatExpr:expr1 SequenceExpr:expr2
          {:
          RESULT = new ConcatNode( expr1, expr2 );
          :}
     |
          SequenceExpr:expr
          {:
          RESULT = expr;
          :}
     ;

SequenceExpr::=
```

```
          SimpleExpr:expr QUEST
          {:
          RESULT = new OptNode( expr );
          :}
     |
          SimpleExpr:expr STAR
          {:
          RESULT = new OptSeqNode( expr );
          :}
     |
          SimpleExpr:expr PLUS
          {:
          RESULT = new SeqNode( expr );
          :}
     |
          SimpleExpr:expr
          {:
          RESULT = expr;
          :}
     ;

SimpleExpr::=
          DOT
          {:
          RESULT = new DotNode();
          :}
     |
          CHAR:chr
          {:
          RESULT = new CharNode( chr.charValue() );
          :}
     |
          STRING:string
          {:
          RESULT = new StringNode( string );
          :}
     |
          LEFTSQ Set:set RIGHTSQ
          {:
          RESULT = new SetNode( set );
          :}
     |
          LEFTSQ CARET Set:set RIGHTSQ
          {:
          RESULT = new NotSetNode( set );
          :}
     |
          LEFT OrExpr:expr RIGHT
          {:
          RESULT = expr;
          :}
     ;

Set::=
          Element:element
          {:
          RESULT = element;
          :}
     |
          Element:element Set:set
          {:
```

```
          RESULT = new SetComponentSeqNode( element, set );
          :}
     ;

Element::=
          CHAR:chr
          {:
          RESULT = new SetElementNode( chr.charValue() );
          :}
     |
          CHAR:chr1 MINUS CHAR:chr2
          {:
          RESULT = new SetRangeNode( chr1.charValue(), chr2.charValue() );
          :}
     ;
```

The Main class

The Main class passes the pattern to the constructor for the parser, and creates a BufferedReader to read the file. It then invokes the parse() method to generate a tree for the pattern, then uses the match() method to attempt to match the pattern to each line of the file.

```
import java.io.*;
import java_cup.runtime.*;
import grammar.*;
import node.exprNode.*;

public class Main {

     public static void main( String[] argv ) {
          try {
               if ( argv.length != 2 )
                    throw new Error( "Usage: java Main pattern filename" );
               // System.out.println( "argv[ 0 ] = " + argv[ 0 ] );
               // System.out.println( "argv[ 1 ] = " + argv[ 1 ] );
               parser p = new parser( argv[ 0 ] );

               ExprNode exprNode = ( ExprNode ) ( p.parse().value );
               System.err.println( "Parse tree " + exprNode );

               if ( exprNode != null ) {
                    int line = 1;
                    String buffer;
                    BufferedReader inputStream =
                         new BufferedReader(
                              new InputStreamReader(
                                   new FileInputStream( argv[ 1 ] ) ) );

                    while ( ( buffer = inputStream.readLine() ) != null ) {
                         if ( exprNode.match( buffer ) )
                              System.out.println( line+ ": " + buffer );
                         line++;
                         }
                    }
               else
                    System.out.println( "Null parse tree" );
               }
          catch ( Exception e ) {
               System.out.println( e.getMessage() );
               }
          }
```

```
    }
```

The Node classes

For every Node corresponding to a pattern, I want to be able to display the node.

```
package node;

public abstract class Node {

    public Node() {
        }

    public abstract String toString();
    }
```

Nodes corresponding to expressions need to have a precedence, and might need to be parenthesized, if the surrounding context has a higher precedence.

```
package node;

public abstract class ExprNode extends Node {

    int precedence;

    public String toString( int parentPrec ) {
        if ( precedence < parentPrec )
            return "(" + toString() + ")";
        else
            return toString();
        }

    public abstract boolean match( String s );

    }
```

Binary nodes have a textual representation for the operator, and left and right children. The method for converting them to a String can be written in terms of the operator text.

```
package node;

public abstract class BinaryNode extends ExprNode {

    String operator;
    ExprNode left, right;

    public BinaryNode( ExprNode left, ExprNode right ) {
        this.left = left;
        this.right = right;
        }

    public String toString() {
        return
            left.toString( precedence )
            + operator
            + right.toString( precedence + 1 );
        }
    }
```

When we get down to the level of specific operators, we can specify a method to determine whether the pattern matches an input String.

For the “|” operator, we match the String if either child matches.

```
package node;

public class OrNode extends BinaryNode {

     public OrNode( ExprNode left, ExprNode right ) {
          super( left, right );
          precedence = 1;
          operator = "|";
          }

     public boolean match( String s ) {
          return left.match( s ) || right.match( s );
          }

     }
```

For the concatenation operator, we match the String if there is a split of the string so that the left child matches the first part, and the right child matches the second part.

```
package node;

public class ConcatNode extends BinaryNode {

     public ConcatNode( ExprNode left, ExprNode right ) {
          super( left, right );
          precedence = 2;
          operator = "";
          }

     public boolean match( String s ) {
          for ( int i = 0; i < s.length() + 1; i++ )
               if ( left.match( s.substring( 0, i ) )
                    && right.match( s.substring( i ) ) )
                    return true;
          return false;
          }

     }
```

```
package node;
```

Postfix nodes have a textual representation for the operator, and a left child. The method for converting them to a String can be written in terms of the operator text.

```
public abstract class PostfixNode extends ExprNode {

     String operator;
     ExprNode left;

     public PostfixNode( ExprNode left ) {
          this.left = left;
          precedence = 3;
          }

     public String toString() {
          return
               left.toString( precedence )
               + operator;
```

```
        }
    }
```

When we get down to the level of specific operators, we can specify a method to determine whether the pattern matches an input String.

To match a sequence, we try to match with a single occurrence, and if that doesn't work, we try to match some input with one occurrence, and the remainder with a sequence.

```
package node;

public class SeqNode extends PostfixNode {

    public SeqNode( ExprNode left ) {
        super( left );
        operator = "+";
        }

    public boolean match( String s ) {
        if ( left.match( s ) )
            return true;
        else {
            for ( int i = 1; i < s.length(); i++ )
                if ( left.match( s.substring( 0, i ) )
                    && match( s.substring( i ) ) )
                    return true;
            return false;
            }
        }

    }
```

To match an optional sequence, we try to match nothing, and if that doesn't work, we try to match some input with one occurrence, and the remainder with an optional sequence.

```
package node;

public class OptSeqNode extends PostfixNode {

    public OptSeqNode( ExprNode left ) {
        super( left );
        operator = "*";
        }

    public boolean match( String s ) {
        if ( s.length() == 0 )
            return true;
        else {
            for ( int i = 1; i < s.length() + 1; i++ )
                if ( left.match( s.substring( 0, i ) )
                    && match( s.substring( i ) ) )
                    return true;
            return false;
            }
        }

    }
```

To match an optional pattern, we try to match nothing or the pattern.

```
package node;

public class OptNode extends PostfixNode {

     public OptNode( ExprNode left ) {
          super( left );
          operator = "?";
          }

     public boolean match( String s ) {
          return s.length() == 0 || left.match( s );
          }

     }
```

Simple nodes have precedence 4.

```
package node;

public abstract class SimpleNode extends ExprNode {

     public SimpleNode() {
          precedence = 4;
          }

     }
```

Dot matches a single character other than ‘\n’.

```
package node;

public class DotNode extends SimpleNode {

     public DotNode() {
          }

     public String toString() {
          return ".";
          }

     public boolean match( String s ) {
          return s.length() == 1 && s.charAt( 0 ) != '\n';
          }

     }
```

A single character matches itself.

```
package node;

public class CharNode extends SimpleNode {

     char value;

     public CharNode( char value ) {
          super();
          this.value = value;
          }

     public String toString() {
          return Text.toString( value );
          }
```

```
     public boolean match( String s ) {
          return s.length() == 1 && s.charAt( 0 ) == value;
          }

     }
```

Strings match themselves.

```
package node;

public class StringNode extends SimpleNode {

     String value;

     public StringNode( String value ) {
          super();
          this.value = value;
          }

     public String toString() {
          String text;
          return "\"" + Text.toString( value ) + "\"";
          }

     public boolean match( String s ) {
          return value.equals( s );
          }

     }
```

A set matches a single character, if one of its components match.

```
package node;

public class SetNode extends SimpleNode {

     SetComponentNode components;

     public SetNode( SetComponentNode components ) {
          this.components = components;
          }

     public String toString() {
          return "[" + components.toString() + "]";
          }

     public boolean match( String s ) {
          return s.length() == 1 && components.match( s.charAt( 0 ) );
          }

     }
```

A negated set matches a single character, if none of its components match.

```
package node;

public class NotSetNode extends SimpleNode {

     SetComponentNode components;

     public NotSetNode( SetComponentNode components ) {
```

```
          this.components = components;
          }

     public String toString() {
          return "[^" + components.toString() + "]";
          }

     public boolean match( String s ) {
          return s.length() == 1 && ! components.match( s.charAt( 0 ) );
          }

     }
```

A set component sequence matches if one of its components match.

```
package node;

public class SetComponentSeqNode extends SetComponentNode {
     SetComponentNode head;
     SetComponentNode tail;

     public SetComponentSeqNode(
          SetComponentNode head,
          SetComponentNode tail ) {
          this.head = head;
          this.tail = tail;
          }

     public String toString() {
          return head.toString() + tail.toString();
          }

     public boolean match( char c ) {
          return head.match( c ) || tail.match( c );
          }

     }
```

```
package node;

public abstract class SetComponentNode extends Node {

     public SetComponentNode() {
          }

     public abstract boolean match( char c );

     }
```

A single character in a set matches if the character agrees with the input.

```
package node;

public class SetElementNode extends SetComponentNode {

     char value;

     public SetElementNode( char value ) {
          this.value = value;
          }

     public String toString() {
```

```
          return Text.toString( value );
          }

     public boolean match( char c ) {
          return value == c;
          }

     }
```

A character range in a set matches if the input character is in range.

```
package node;

public class SetRangeNode extends SetComponentNode {

     char low, high;

     public SetRangeNode( char low, char high ) {
          this.low = low;
          this.high = high;
          }

     public String toString() {
          return Text.toString( low ) + "-" + Text.toString( high );
          }

     public boolean match( char c ) {
          return low <= c && c <= high;
          }

     }
```