Chapter 1 Lexical Analysis Using JFlex

Tokens

The first phase of compilation is lexical analysis - 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. For example, in most programming languages we have the following kinds of tokens.

- Identifiers (x, y, average, etc.)
- Reserved or keywords (if, else, while, etc.)
- Integer constants (42, 0xFF, 0177 etc.)
- Floating point constants (5.6, 3.6e8, etc.)
- String constants ("hello there\n", etc.)
- Character constants ('a', 'b', etc.)
- Special symbols (( ) : := + - etc.)
- Comments (To be ignored.)
- Compiler directives (Directives to include files, define macros, etc.)
- Line information (We might need to detect newline characters as tokens, if they are syntactically important. We must also increment the line count, so that we can indicate the line number for error messages.)
- White space (Blanks and tabs that are used to separate tokens, but are otherwise not important).
- End of file

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 (perhaps the text that makes up the identifier, or an integer index into a table of identifiers).

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 - the numeric value of the integer. 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, to internal form can be moderately complex. The surrounding quote marks have to be deleted, and escaped characters have to be translated into internal form. For example “\n” (a “\” then an “n”) has to be translated by the compiler into a newline character, “\"” into a “"” character, “\177” into a delete character, etc.

Some tokens, while important for lexical analysis, are irrelevant for the parser (the portion of the compiler that analyses the structure of the program being compiled). For example, layout tokens, such as white space, newlines, and comments are processed by the lexical analyser, then discarded, since they are ignored by the parser. Nevertheless newlines will have to be counted, if we want to generate appropriate error messages, with a line number.

Lexical Errors

The lexical analyser must be able to cope with text that may not be lexically valid. For example

- A number may be too large, a string may be too long or an identifier may be too long.
- A number may be incomplete (e.g. 26., 26e, etc.).
- The final quote on a string may be missing.
- The end of a comment may be missing.
- A special symbol may be incomplete (e.g. If the special symbols included := :=: :<>: and we came across :<=: in the text, we may consider this to be incomplete).
- Invalid characters may appear in the text, for example if we accidentally attempt to lexically analyse a binary file.
- Compiler directives may be invalid.

The compiler must produce an error message and somehow continue the lexical analysis. It would appear to be relatively easy to correct lexical errors (mostly just by deleting characters), 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 terminate the rest of the compiler if a lexical error occurs, and only continue performing the lexical analysis.

Regular Expressions

In most programming languages, lexical tokens seem to have a remarkably simple structure, that can be described by patterns called regular expressions. Regular expressions are used in many editors, and by the UNIX command grep, to describe search patterns. They are also used by Lex, Flex, JLex and JFlex, four very similar special purpose computer languages used for writing lexical analysers. 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. We will use JFlex, because it is Java based and more sophisticated than JLex.

To indicate that we want to match simple text, we use a pattern equal to the text itself. For example

while Matches the text “while”.

We can match an identifier in most programming languages by using the pattern

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

The pattern [A-Za-z] represents any upper or lower case alphabetic letter (a range of characters is represented by writing the lower bound, a “-”, then the upper bound). The pattern [A-Za-z0-9] represents any upper or lower case alphabetic letter or decimal digit. Similarly, it is possible to put a “^” just after the “[” to match “any characters except” the following characters. For example the pattern [^\r\n\"\\] matches any character except a carriage return, linefeed, double quote or backslash character.

We can put a “*” after a pattern, to match text that corresponds to 0 or more occurrences of the pattern. Similarly we can put a “+” after a pattern to match text that corresponds to 1 or more occurrences of the pattern, a “?” after a pattern to match text that corresponds to an optional occurrence of the pattern, {n} (where n is a decimal integer) after a pattern to match text that

corresponds to n occurrences of the pattern, and {m, n} (where m and n are a decimal integers) after a pattern to match text that corresponds to between m and n occurrences of the pattern

We can write two patterns side by side, to match text corresponding to the first pattern, followed by text corresponding to the second pattern.

Hence the above pattern represents an alphabetic letter, followed by 0 or more letters or digits.

Similarly, the pattern

0	matches the integer 0.
[1-9][0-9]*	matches a nonzero decimal integer.
0[0-7]+	matches an octal integer.
0[xX][0-9A-Fa-f]+	matches a hexadecimal integer.

Some characters have a special meaning. If we want to use these characters with their normal meaning, we have to “escape” or “quote” the character. There are two ways of doing this. We can precede the character by a “\” or enclose a sequence of characters in "...". For example

[0-9]+ \. [0-9]+ [eE] [\+\-]? [0-9]+

matches one of the possible patterns for a floating point value. (It does not match all alternatives, since some of the portions can be omitted.)

In this case, we have escaped the “.”, “+” and “-” by prefacing them by a “\”, because they have a special meaning.

“.” is a pattern that matches any character, other than a newline. In fact it is only useful on UNIX systems, since it still matches carriage returns, which are used for line breaks on Macintoshes and PCs.

For example

"//".*	matches a comment in C++ or Java (since most pattern matchers match the longest possible text).

In fact, it is better to replace “.” by “[^\r\n]”, to also exclude a carriage return. JFlex is UNIX based, so it assumes line breaks are newlines.

“\r”, “\n”, “\t”, “\b”, etc have their usual meaning in C or Java, as carriage return, linefeed (newline), tab, backspace, etc.

We can also write patterns to represent matching of one of several alternatives, by using the “|” operator. For example

0 | [1-9][0-9]* | 0[0-7]+ | 0[xX][0-9A-Fa-f]+

represents the various styles of integer constant allowed in C or Java.

Regular expressions have different precedences. For example the postfix sequence operators “*”, “+”, and “?” have a higher precedence than concatenation, which has a higher precedence than “|”. Sometimes we need to overrule the precedence, by parenthesising the regular expressions. For example

```
{letter}({letter}|{digit})*
```

represents an identifier in most languages, if we define letter as [A-Za-z] and digit as [0-9].

It is quite difficult to understand complex regular expressions, so it is desirable to be able to name regular expressions, and use them in other regular expressions. In the JFlex language, used for writing lexical analysers, it is possible to associate identifiers with regular expressions, and use these identifiers later, by enclosing them in {}. For example the following definitions describe many of the tokens that occur in Java. (refer JAVA)

```
package grammar;

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

%%

%public
%type         Symbol
%char

%{
     public Symbol token( int tokenType ) {
          System.err.println( "Obtain token " + sym.terminal_name( tokenType )
               + " \"" + yytext() + "\"" );
          return new Symbol( tokenType, yychar,
               yychar + yytext().length(), yytext() );
          }

%}

InputChar      =    [^\n\r]
SpaceChar      =    [\ \t]
LineChar       =    \n|\r|\r\n

Zero           =    0
DecInt              =    [1-9][0-9]*
OctalInt       =    0[0-7]+
HexInt              =    0[xX][0-9a-fA-F]+

Integer             =    ( {Zero} | {DecInt} | {OctalInt} | {HexInt} )[lL]?
Exponent       =    [eE] [\+\-]? [0-9]+
Float1              =    [0-9]+ \. [0-9]+ {Exponent}?
Float2              =    \. [0-9]+ {Exponent}?
Float3              =    [0-9]+ \. {Exponent}?
Float4              =    [0-9]+ {Exponent}
Float               =    ( {Float1} | {Float2} | {Float3} | {Float4} ) [fFdD]? |
[0-9]+ [fFDd]
Ident               =    [A-Za-z_$] [A-Za-z_$0-9]*
CChar               =    [^\'\\\n\r] | {EscChar}
SChar               =    [^\"\\\n\r] | {EscChar}
EscChar             =    \\[ntbrf\\\'\"] | {OctalEscape}
OctalEscape         =    \\[0-7] | \\[0-7][0-7] | \\[0-3][0-7][0-7]

%%
abstract       { return token( sym.ABSTRACT ); }
boolean        { return token( sym.BOOLEAN ); }
break          { return token( sym.BREAK ); }
…
transient      { return token( sym.TRANSIENT ); }
try            { return token( sym.TRY ); }
void           { return token( sym.VOID ); }
volatile       { return token( sym.VOLATILE ); }
while          { return token( sym.WHILE ); }
```

```
"("             { return token( sym.LEFT ); }
")"             { return token( sym.RIGHT ); }
"{"             { return token( sym.LEFTCURLY ); }
"}"             { return token( sym.RIGHTCURLY ); }
"["             { return token( sym.LEFTSQ ); }
"]"             { return token( sym.RIGHTSQ ); }
…
"&"             { return token( sym.AMPERSAND ); }
"!"             { return token( sym.EXCLAIM ); }
"~"             { return token( sym.TILDE ); }

true            { return token( sym.BOOLEANLIT ); }
false           { return token( sym.BOOLEANLIT ); }
null            { return token( sym.NULLLIT ); }

{Integer}       { return token( sym.INTEGERLIT ); }

{Float}         { return token( sym.FLOATLIT ); }

\'{CChar}\'     { return token( sym.CHARLIT ); }
\"{SChar}*\"    {return token( sym.STRINGLIT ); }

{Ident}         { return token( sym.IDENT ); }

"//"{InputChar}*    { }

"/*"~"*/"       { }

{LineChar}      { }
{SpaceChar}     { }
<<EOF>>         { return token( sym.EOF ); }
.               { return token( sym.error ); }
```

Overview of JFlex

JFlex takes a JFlex program and creates a Java file. I give the JFlex program a suffix of ".jflex", although this is not compulsory. 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.

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. For example, the parameter can be a StringReader (if we want to obtain the input from a String) or an InputStream (if we want to obtain the text from a file).

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. This method loops, matching the input to regular expressions, and performing the action associated with that regular expression. If the action contains a return statement, the method returns the value indicated.

An Example

(Refer SENTENCE.)

The following program takes text as input, and reformats it, one sentence to a line, with the first letter of the sentence capitalised, and only one space between words.

```
package grammar;

import java.io.*;

%%

%{
     static String capitalize( String s ) {
          return Character.toUpperCase( s.charAt( 0 ) ) + s.substring( 1 );
          }

%}

%public
%class Sentence
%type Void

%init{
    yybegin( FIRST );
%init}

letter         =    [A-Za-z]
word           =    {letter}+
endPunct       =    [\.\!\?]
otherPunct     =    [\,\;\:]
space          =    [\ \t\r\n]

%state FIRST, REST

%%
<FIRST> {
     {word}         {
                    System.out.print( capitalize( yytext() ) );
                    yybegin( REST );
                    }
     }

<REST> {
     {word}         {
                    System.out.print( " " + yytext() );
                    }
     {endPunct}          {
                    System.out.println( yytext() );
                    yybegin( FIRST );
                    }
     {otherPunct}   {
                    System.out.print( yytext() );
                    }
     }

{space}             {
                    }

.                   {
                    System.err.println(
                         "Invalid character \"" + yytext() + "\"" );
                    }
```

Our main() method in our Main class takes a directory as a parameter. This directory is meant to contain an input file “program.in”, and output and error files are created in this directory. Once we have created our lexical analyser instance, we can invoke the yylex() method (an instance method).

```
import grammar.*;
import java.io.*;

public class Main {

     public static void main( String[] argv ) {
          String dirName = null;

          try {
               for ( int i = 0; i < argv.length; i++ ) {
                    if ( argv[ i ].equals( "-dir" ) ) {
                         i++;
                         if ( i >= argv.length )
                              throw new Error( "Missing directory name" );
                         dirName = argv[ i ];
                         }
                    else {
                         throw new Error(
                              "Usage: java Main -dir directory" );
                         }
                    }

               if ( dirName == null )
                    throw new Error( "Directory not specified" );

               FileInputStream fileInputStream = new FileInputStream(
                    new File( dirName, "program.in" ) );
               System.setErr( new PrintStream( new FileOutputStream(
                    new File( dirName, "program.err" ) ) ) );
               System.setOut( new PrintStream( new FileOutputStream(
                    new File( dirName, "program.out" ) ) ) );

               Sentence lexer = new Sentence( fileInputStream );
               lexer.yylex();
               }
          catch ( Exception exception ) {
               System.err.println( "Exception in Main " + exception.toString() );
               exception.printStackTrace();
               }
          }

     }
```

The method yylex() loops, obtaining characters from the input stream, and matching patterns. Whenever it matches a pattern, it executes the action associated with that pattern. In our example, there is no return statement in the action, so yylex() just eats up all the input and performs the action for each token. Eventually, it reaches end of file, and returns. Most sensible lexical analysers make yylex() return a value when it matches a token other than white space or a comment.

What does our sample program do?

If it matches a word at the beginning of a sentence (represented by the FIRST state), it prints it out again, with the first letter in upper case. The line

```
yybegin( REST );
```

causes the current state to change to the REST state.

If it matches a word within a sentence (represented by the REST state), it prints it out again, preceded by a space.

If it matches a “.” or “!”or “?”, it prints it out, followed by a newline. It then changes to the FIRST state.

If it matches a “,” or “;”or “:”, it prints it out.

It just eats up white space and line breaks.

Anything else causes an error message to be printed.

Lexical Structure of JFlex

Comments

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

Spaces and line breaks

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

The Syntax of JFlex

It is a little early to give you a grammar definition, but 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. I have sometimes specified the grammar as it should be, rather than as it is. (However, the grammar of JFlex is far from regular, due to its historical origins, and could be much improved.) Moreover, I have not specified some things, such as “Java code” or “Directive”.

Overall structure

```
specification::=
          “Java code”
          “%%”
          macroList
          “%%”
          ruleList
     ;
```

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. For example

```
package grammar;
%%
%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. It generates a class

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

which can be invoked from 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();
               }
          }
     }
```

Directives and macros

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

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

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, 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]*".

Rules

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

rule::=
          statesOpt startLineOpt regExpr followOpt endLineOpt action
```

```
    |    statesOpt "<<EOF>>" action
    |    "<" stateList ">" "{" ruleList "}"
    ;

statesOpt::=
         "<" stateList ">"
    |    /* Empty */
    ;

stateList::=
         IDENT "," stateList
    |    IDENT
    ;

startLineOpt::=
         "^"
    |    /* Empty */
    ;

followOpt::=
         "/" regExpr
    |    /* Empty */
    ;

endLineOpt::=
         "$"
    |    /* Empty */
    ;

action::=
         "{ Java code }"
    |    "|\n"
    ;
```

A rule specifies what actions to perform when a regular expression is matched. It is composed of:

- An optional list of start states indicating that the rule should only be matched if the lexical analyser is in one of the specified start states. The start states are enclosed in "<...>".
- An optional "^", indicating that the rule should only be matched if the text occurs at the beginning of a line. This is often useful in languages in which "#" at the beginning of a line means a macro, while "#" in any other position is just a comment.

  For example, we could write

  ```
  <NORMAL>^#          { yybegin( MACRO ); }
  ```

  to match a line starting with a "#" and change into a new state for processing a macro.
- A regular expression indicating the text to be matched.
- An optional "/ regular expression" indicating that the rule should only be matched if the following text matches the specified regular expression. The input matched by the regular expression after the "/" is not considered to be part of the token itself, and is not consumed.

  For example, sometimes we might not want to match white space as a token (because to a large extent the language is in free format), but we might want to recognise a token followed by white space differently from one not followed by white space.
- An optional "$", indicating that the rule should only be matched if the text is at the end of a line.

- An action, indicating the Java code to perform if the rule is matched.

The action can be replaced by "|", at the end of line, to indicate that the action is the same as that of the following rule. It effectively allows alternatives in a regular expression to be split across a line. For example

```
0                   |
0[0-7]+             |
[1-9][0-9]*         |
0[xX][0-9a-fA-F]+   { return token( sym.INTCONST ); }
```

The pattern "<<EOF>>" can be used to match end of file.

```
<<EOF>>         { return token( sym.EOF ) }
```

Several rules can be grouped together, with the same start state list. The syntax "<stateList> { ruleList }" has the same effect as putting the stateList in front of each rule individually, but it is more concise and provides better structuring of the program. If subrules have states, they will be matched if the lexical analyser is in either the states of the enclosing group of rules, or the states of the specific rule.

For example, the following program processes text, and when it finds a string constant, builds the string, converting escape characters, etc. It reprints the string when it finds the end of the string.

```
package grammar;
%%

%{
     String text;

     void append( char c ) {
          text += c;
          }

%}

%public
%type Void
%state STRING
newline       =    \r|\n|\r\n
%%
<YYINITIAL> {
     \"                       { yybegin( STRING ); text = ""; }
     {newline}                { }
     .                        { }
     }

<STRING> {
     \"                       {
                              yybegin( YYINITIAL );
                              System.out.println( text );
                              }
     {newline}                {
                              yybegin( YYINITIAL );
                              System.out.println( text
                                   + " <<< Incomplete string" );
                              }
     \\b                      { append( '\b' ); }
     \\t                      { append( '\t' ); }
     \\f                      { append( '\f' ); }
     \\r                      { append( '\r' ); }
     \\n                      { append( '\n' ); }
     \\[0-3][0-7][0-7]        |
     \\[0-7][0-7]             |
     \\[0-7]                  {
                              append( ( char ) Integer.parseInt(
                                   yytext().substring( 1 ), 8 ) );
                              }
     \\x[0-9a-fA-F][0-9a-fA-F]    |
     \\x[0-9a-fA-F]           {
                              append( ( char ) Integer.parseInt(
                                   yytext().substring( 2 ), 16 ) );
                              }
     \\.                      { append( yytext().charAt( 1 ) ); }
     .                        { append( yytext().charAt( 0 ) ); }
     }
```

Regular expressions

The different operators that can be used in regular expressions have different precedences. The lowest precedence operator is "|", then concatenation, then the sequence operators. So "a | bc*" matches either "a", or ("b" followed by a sequence of 0 or more "c"s).

```
regExpr::=
          altExpr
     ;
```

```
altExpr::=
          altExpr "|" concatExpr
     |    concatExpr
     ;
```

We can combine a number of alternative patterns we want to match, by writing the alternatives down with "|" (pronounced "or") between them. For example "ab | cd | ef" can match any one of the strings "ab", "cd" or "ef".

```
concatExpr::=
          concatExpr prefixExpr
     |    prefixExpr
     ;
```

We can specify that we want to match several patterns in sequence, by writing the patterns one after the other. For example "[abc][def]" can match any one of the strings "ad", "ae", "af", "bd", "be", "bf", "cd", "ce", or "cf".

```
prefixExpr::=
          "!" prefixExpr
     |    "~" prefixExpr
     |    seqExpr
     ;
```

We can precede a pattern by "!" (pronounced "not"), to specify that we want to match anything except the pattern. The use of "!" can cause JFlex to generate a finite state automaton with exponential size, so it is not an operator to be used without due care. I am not convinced that it is all that useful. The only use I can think of is to get the intersection "`A&B`" of two regular expressions by writing "`!(!A|!B)`", or the difference "`A\B`" of two regular expressions by writing "`!(!A|B)`".

We can precede a pattern by "~" (pronounced "up to"), to specify that we want to match all text up to the first occurrence of the pattern. For example "/*"~"*/" can be used to match a C/Java comment, so long as we do not allow nested comments.

```
seqExpr::=
          simpleExpr "*"
     |    simpleExpr "+"
     |    simpleExpr "?"
     |    simpleExpr "{" INTCONST "}"
     |    simpleExpr "{" INTCONST "," INTCONST "}"
     |    simpleExpr
     ;
```

We can follow a regular expression by "*", to specify that we want to match zero or more repetitions of text matched by the regular expression. For example [ab]* matches "", "a", "b", "aa", "ab", "ba", "bb", "aaa", "aba", etc.

We can follow a regular expression by "+", to specify that we want to match one or more repetitions of text matched by the regular expression. For example [ab]+ matches "a", "b", "aa", "ab", "ba", "bb", "aaa", "aba", etc.

We can follow a regular expression by "?", to specify that we want to match an optional occurrence of text matched by the regular expression. For example [ab]? matches "", "a", "b".

We can follow a regular expression by "{ INTCONST }", to specify that we want to match the specified number of repetitions of the regular expression. For example, [ab]{3} matches "aaa", "aab", "aba", "abb", "baa", "bab", "bba", or "bbb".

We can follow a regular expression by "{ INTCONST1, INTCONST2 }", to specify that we want to match between INTCONST1 and INTCONST2 repetitions of the regular expression. For example a{3,5} matches "aaa", "aaaa", or "aaaaa".

```
simpleExpr::=
          "(" regExpr ")"
     |    "{"IDENT"}"
     |    charSet
     |    predefinedCharSet
     |    CHAR
     |    STRING
     |    "."
     ;
```

Regular expressions can be enclosed in parentheses, to avoid problems with precedences. For example a(bc|de)f matches “abcf” or “adef”.

Regular expressions that were named in the macro section can be referred to by enclosing the name in “{...}”. For example, if we define

```
zero          =    0
octal         =    0[0-7]+
decimal       =    [1-9][0-9]*
hexadecimal   =    0[xX][0-9a-fA-F]+
```

then we can write the pattern {zero}|{octal}|{decimal}|{hexadecimal} to match an integer constant.

A character set “[...]” can be used to match any one of the characters in the character set.

There are some predefined character sets, for matching letters, digits, etc.

A single character can be used to match itself. This is why something like “while” (without the quotes) can be used to match the text “while” (it is the concatenation of the individual characters “w”, “h”, “i”, “l”, “e”).

It is also possible to escape characters, by putting a \ in front. This needs to be done for characters that have special meanings, namely

~ ! ? * + | ( ) ^ $ / . < > [ ] { } " \

Almost any character can be escaped in this manner, and this is useful if you are unsure whether a character is special. However, like in C/java, some have a special meaning.

\b Backspace.

\n Linefeed (newline).

\t Tab.

\f Formfeed.

\r Carriage return.

\[0-3][0-7][0-7]

The character code corresponding to the number formed by the three octal digits.

\x[0-9a-fA-F][0-9a-fA-F]

The character code corresponding to the number formed by two hexadecimal digits.

\u[0-9a-fA-F][0-9a-fA-F][0-9a-fA-F][0-9a-fA-F]

The character code corresponding to the number formed by four hexadecimal digits.

\c A backslash followed by any other character c matches itself.

We can also eliminate the special meaning of text by enclosed it in double quotes ("..."). However, \ and “"” retain their special meaning. Spaces and tabs represent characters to be matched if they occur inside strings. A string can be used to match the text it contains.

A "." matches any character except "\n". It would be better if it also didn't match "\r", to allow machine independence.

Character sets

```
charSet::=
          "[" elementList "]"
     |    "[" "^" elementList "]"
     ;

elementList::=
          elementList element
     |    /* Empty */
     ;
element::=
          CHAR "-" CHAR
     |    CHAR
     |    STRING
     ;
```

A character set is represented by enclosing characters inside "[...]". It matches **any one of the characters** specified inside the "[...]". For example [0123456789] matches any one decimal digit.

The characters ^ [ ] { } " - and \ have a special meaning in character sets, and need to be escaped. Spaces and tabs represent characters to be matched inside character sets.

The symbol "^" means "any character except the characters listed in the character set". For example [^0123456789] matches any one character except a decimal digit.

A range of characters can be represented by writing lowerBound-upperBound. For example [A-Za-z] matches any one alphabetical character.

Characters inside a character set can be escaped using \, or double quotes.

```
predefinedCharSet::=
          "[:jletter:]"
     |    "[:jletterdigit:]"
     |    "[:letter:]"
     |    "[:digit:]"
     |    "[:upper:]"
     |    "[:lower:]"
     ;
```

There are predefined character sets for letters, digits, upper and lower case characters, etc. They all involve enclosing a name inside "[:...:]".

Adding Java Code to the Lexical analyser

Adding Java code outside the class declaration

To add Java code outside the class declaration to the Java file generated by JFlex, place the code before the first "%%". 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 declare helper classes in the user code section, but it is generally considered bad taste to do so. Other classes should be declared in their own files.

Adding Java code inside the class declaration

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.

Adding Java code to the constructor

It is possible to add Java code inside the constructor, by enclosing it in %init{ ... %init} in the directives section. For example, you might specify the initial start state for the lexical analyser.

Adding Java code to the actions

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.

Directives in JFlex

Customising the class

%class ClassName

The %class directive changes the the name of the class generated from Yylex to ClassName, and changes the file generated from Yylex.java to ClassName.java. The file is saved in the same directory as the JFlex specification, unless it is altered by the -d option to the JFlex command.

%public

The %public directive makes the class public. Otherwise it has package access.

%include Filename

The %include directive includes the specified file (like in C). The file name is not enclosed in quotes.

Customising the lexical analyser method

%function MethodName

The %function directive changes the name of the lexical analyser method from yylex to MethodName.

%int

The return type of the lexical analyser method is changed from Yytoken to int.

%type TypeName

The return type of the lexical analyser method is changed from Yytoken to TypeName.

Character sets

%unicode

The %unicode directive makes the lexical analyser use full UNICODE characters. It is recommended that you always use this option. The single byte default is only provided for compatibility with JLex.

%ignorecase

The %ignorecase directive causes the lexical analyser to ignore case.

Line, character and column counting

%char

%line

%column

These directives cause the lexical analyser to compute the character count (number of characters from the beginning of input to the start of the token), line count (number of line breaks from the beginning of input to the start of the token), and column count (number of characters from the beginning of line to the start of the token) for the token matched. (All start at 0).

The values of the variables yychar, yyline, yycolumn are set to the relevant information.

Compatibility

%cup

This directive sets a number of options, which makes the class suitable for use with CUP. The class implements java_cup.runtime.Scanner, names the lexical analyser method next_token, makes the return type java_cup.runtime.Symbol, makes the lexical analyser method return new Symbol(sym.EOF) on end of file, etc.

%byacc

This directive sets a number of options, which makes the class suitable for use with Byacc/J, a Java based parser generator, based on Yacc, and similar to CUP.

State directives

%state state0, state1, state2, ...

%xstate state0, state1, state2, ...

The %state directive declares the states specified in the comma separated list of identifiers.

The lexical analyser can match different rules depending on which state it is in. The %state and %xstate directives name the possible states. The initial state is YYINITIAL. The current state (an int value) can be obtained by the method yystate(), and can be altered by the method yybegin(int newState).

If a rule is preceded by a list of states, then it can only be matched if the lexical analyser is in one of these states.

States can be declared as inclusive (using the %state directive) or exclusive (using the %xstate directive). The only difference is that if a rule is not preceded by any states, then it is matched if the lexical analyser is in one of the inclusive states.

Macro definitions

Regular expressions may be named, by writing

Ident = RegExpr

The regular expression may be referred to later by enclosing the name in {...}.

Named regular expressions cannot be used recursively. They only provide a way of abbreviating regular expressions.

Built in fields and methods

`int yychar`

represents the number of characters processed since the start of input.

`int yyline`

represents the number of line breaks processed since the start of input.

`int yycolumn`

represents the number of characters processed since the start of the current line.

```
String yytext()
```

returns the text matched by the current rule.

```
int yylength()
```

returns the length of the text matched by the current rule.

```
int yystate()
```

returns the current state.

```
void yybegin( int lexicalState )
```

sets the current state.

```
void yypushback( int number )
```

deletes the specified number of characters from the end of the text matched, and pushes them back into the input, so that they can be re-read. After this, yylength() and yytext() will not include the characters pushed back.

```
void yyreset( Reader reader )
void yypushStream( Reader reader )
void yypopStream()
boolean yymoreStreams()
```

These methods can be used to implement include files. They are used to change where input is obtained from.

The structure of the Java Source Generated by JFlex

The JFlex program is translated into Java. A lot of the Java program is made up of tables to drive the lexical analyser finite state automaton, and we don't want to try and understand that part, but it is useful to see where the portions of the JFlex program turn up in the Java program, since it makes it clearer to us what is going on.

```
// Code from the user section before the first %%
package grammar;

import java.io.*;

public class Sentence {

...
/** lexical states */
  final public static int YYINITIAL = 0;
  final public static int FIRST = 1;
  final public static int REST = 2;

...
// code from %{ ... %}
  static String capitalize( String s ) {
          return Character.toUpperCase( s.charAt( 0 ) ) + s.substring( 1 );
          }

// The two kinds of public constructor, with the name from %class
  public Sentence(java.io.Reader in) {
      yybegin( FIRST );
    this.yy_reader = in;
  }

  public Sentence(java.io.InputStream in) {
    this(new java.io.InputStreamReader(in));
  }
...
```

```
//   Provided methods
  final public void yybegin(int newState) {
    yy_lexical_state = newState;
  }

  final public String yytext() {
    return new String( yy_buffer, yy_startRead, yy_markedPos-yy_startRead );
  }

...
//   The yylex method
  public Void yylex() throws java.io.IOException {
...
    while (true) {
...
      switch (yy_action) {

        case 8:
          {
           System.out.print( yytext() );
           }
        case 10: break;
        case 7:
          {
           System.out.println( yytext() );
           yybegin( FIRST );
           }
        case 11: break;
        case 6:
          {
           System.out.print( " " + yytext() );
           }
        case 12: break;
        case 4:
          {
           }
        case 13: break;
        case 3:
          {
           System.out.println( "Invalid character \"" + yytext() + "\"" );
           }
        case 14: break;
        case 5:
          {
           System.out.print( capitalize( yytext() ) );
           yybegin( REST );
                                  }
        case 15: break;
        default:
          if (yy_input == YYEOF && yy_startRead == yy_currentPos) {
            yy_atEOF = true;
            return null;
          }
          else {
            yy_ScanError(YY_NO_MATCH);
          }
      }
    }
  }
```

```
}
```

Looking at this code makes the whole idea of what JFlex is doing much more concrete.

Running Java, Javac and Jar

Look on the department web pages, looking under References, then Java, Tool Reference, etc., for more information on running java and javac.

The UNIX Java interpreter is called java, and the Java compiler is called javac. On Windows, they are called java.exe and javac.exe. These files are in a directory with a name like /usr/local/java/bin on our UNIX machines, and in /cygdrive/c/jdk/bin when using cygwin (i.e., C:\jdk\bin in DOS) on our undergraduate laboratory Windows machines. As well as bin, there is also a subdirectory jre/lib, within the main java directory, containing jar files rt.jar and i18n.jar for the Java runtime environment (i.e., files containing the standard Java and internationalisation libraries).

Both java and javac expect source and class files to be stored with a directory structure that matches the package structure. Javac stores the created class files in the same manner.

Class files may also be packed together in a zip or jar file. The jar command can be used to create a jar file.

To run the java interpreter, type

```
java [options] <mainClassName> <parameters>
java -jar <jarFile.jar> <parameters>
```

In the second case, the jar file must include manifest information of the form "Main–Class: *mainClassName*" that specifies the name of the main class. The jar file must contain all the required user classes.

Java executes the main method in the main class and then exits unless main creates one or more threads. If any threads are created by main then java doesn't exit until the last thread exits.

Any parameters that appear after the main class name on the command line are passed to the main method of the class, as an array of Strings.

On many systems, it is also possible to execute the code in a jar file containing appropriate manifest information by double clicking on it.

Options for java include

-classpath path

> Specifies the path java uses to look up classes. It overrides the default or the CLASSPATH environment variable, if it is set. Directories containing class files and names of zip/jar files are separated by colons (":") on UNIX machines, and semicolons (";") on windows machines. The subdirectory structure of a classpath directories must match the package structure.
>
> When looking for classes, java first looks in the jre/lib directory for "bootstrap" classes (the standard libraries), then jre/lib/ext for local "extension" classes, then in the directories or files listed in the class path (in order from left to right). If there is no classpath option, the value of the CLASSPATH variable is used instead. If the CLASSPATH variable is also not defined, the class path is just the current directory.

Note that it is a good idea to specify the classpath as an option for each command, rather than placing commonly used libraries in some place like javaXXX/jre/lib/ext, or setting the CLASSPATH shell variable. On shared machines, you do not have the ability to place jar files in public places (unless you are superuser), and doing so might cause conflicts for other users. It is a bad idea to set the CLASSPATH variable in your .bash_profile file, because that assumes that you

want the same classpath under all circumstances. Maybe different courses use different class libraries, and they conflict with each other (for example, because they refer to different versions of the same package). This problem actually occurred a couple of years ago.

To run the java compiler, type

```
javac [options] <sourceFiles>
```

The java command expects the binary representation of the class ClassName to be in a file called ClassName.class. This class file is generated by compiling the corresponding source file with javac. All Java class files end with the filename extension .class which the compiler automatically adds when the class is compiled. The main class must contain a main() method defined as follows:

```
class ClassName {
     public static void main( String[] argv ) {
          ...
          }
     }
```

Options for javac include

-classpath path

Specifies the path javac uses to look up compiled classes being referenced by other classes you are compiling. If sourcepath is not specified, the class path is searched for source code as well.

-sourcepath path

Specifies the path javac uses to look up java source files being referenced by other classes you are compiling.

-d directory

Specifies the destination directory in which to place the class files. The subdirectory structure of the destination directory will match the package structure.

The compiler reads only from the class and source path directories, and writes to the destination directory. If a class occurs in both the class path and the source path, the modification dates are checked, to determine whether the source file needs to be recompiled.

-deprecation

Indicate any use or overriding of a deprecated member or class. Without –deprecation, javac shows the names of source files that use or override deprecated members or classes.

-nowarn

Disable warning messages.

-g

Generate full debugging information at run time, including the values of variables.

To run jar and create a jar archive, type

```
jar [options] <manifestFile> <destFile> <classFiles>
```

For example, we could write

```
cd Classes
jar cvmf ../manifest ../run.jar `find . -name "*.class"`
```

The order for the manifest file and destination file must agree with the order of m and f in the options.

We can run the resultant program using the java command, with the classpath including this jar file.

If the manifest file specifies the name of the main class by a line of the form "Main–Class: *mainClassName*", then we can run the program as

```
java -jar run.jar <parameters>
```

Options for jar include

c Create a jar file (as opposed to extract or list files)

t List the contents of the jar file.

x Extract the files from the jar file, into the current directory.

v Generate verbose output.

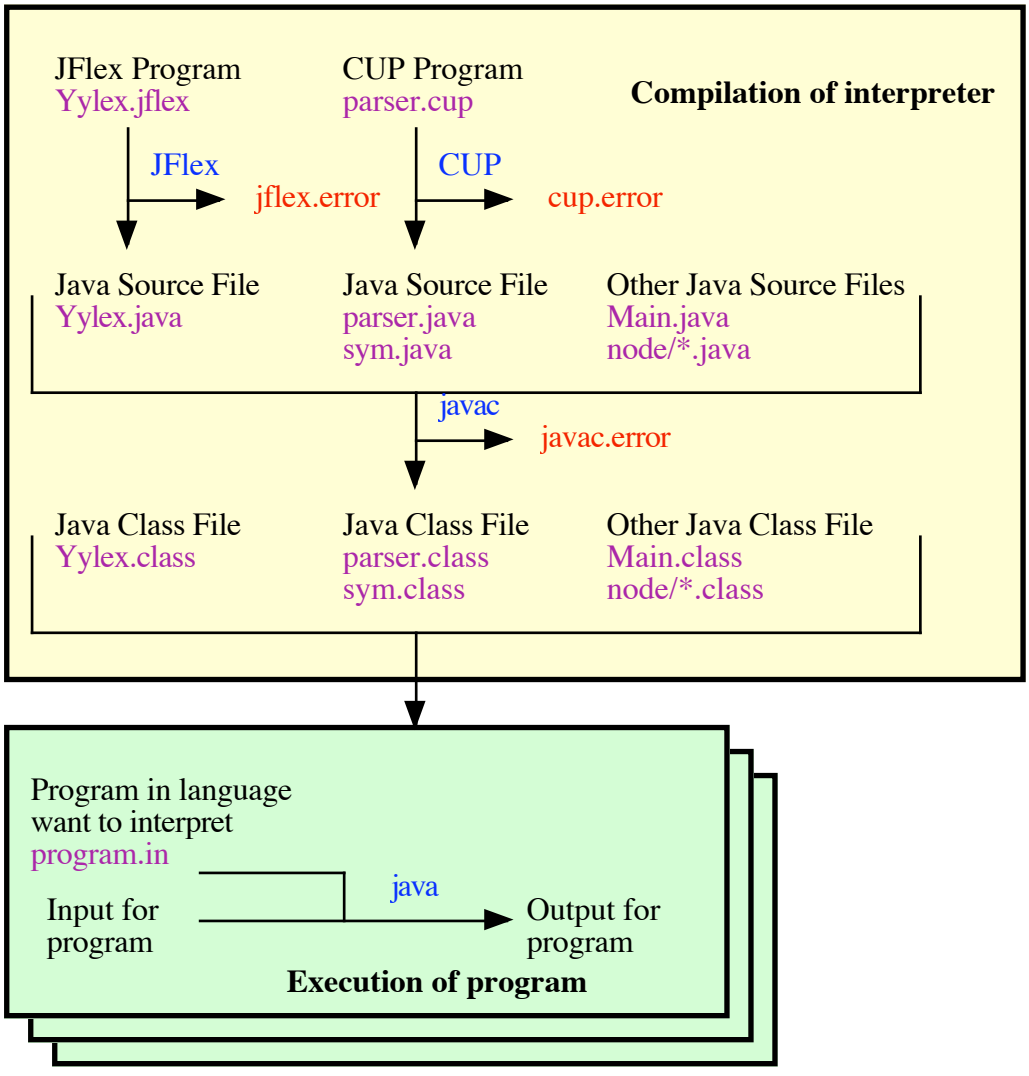
m <manifestFile>

Specifies the name of the manifest file to include.

f <destFile>

Specifies the name of the jar file to create.

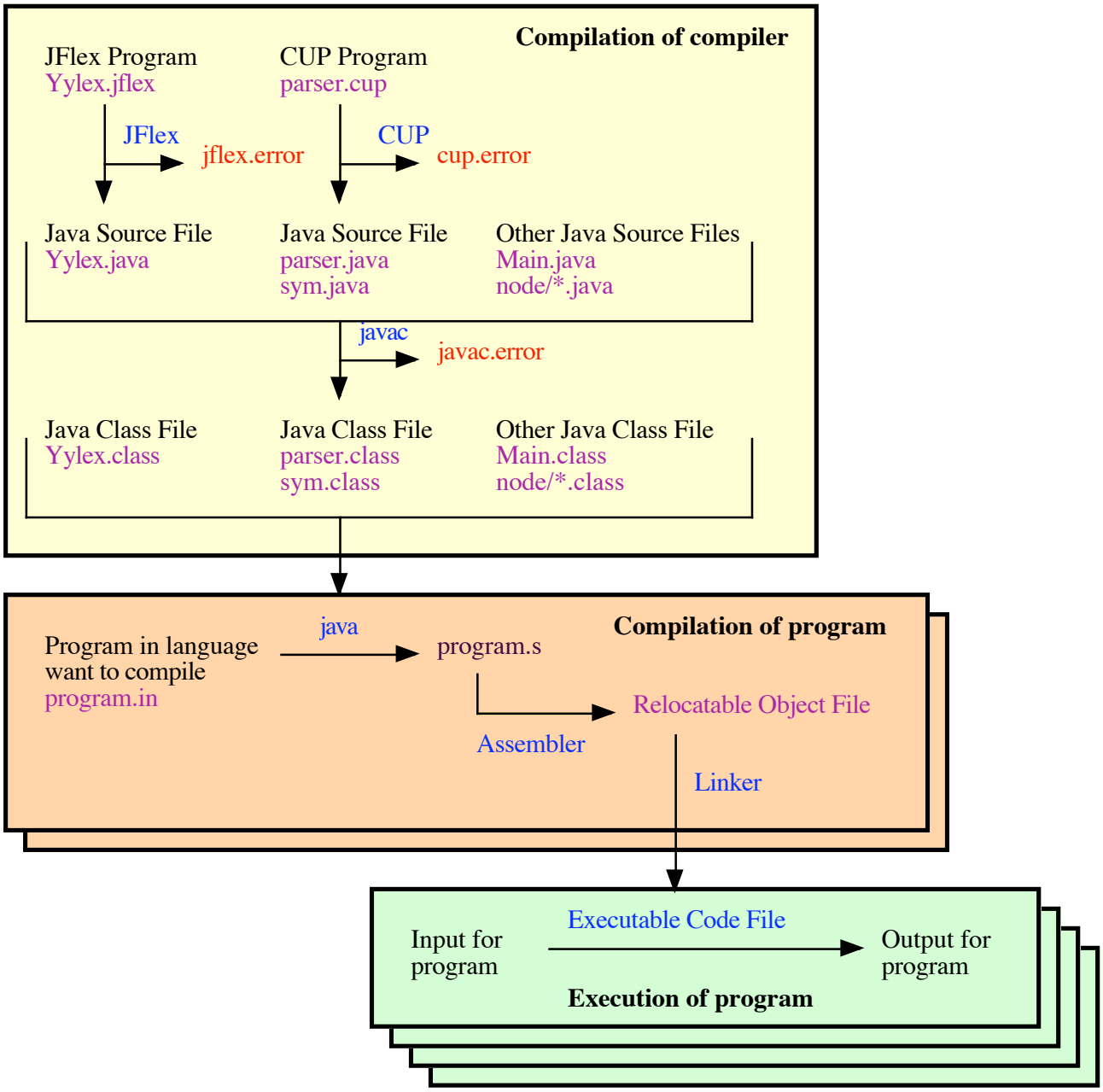
Using JFlex and CUP to implement an interpreter or compiler



Implementing an Interpreter

Suppose we want to implement an interpreter. We can write the interpreter in a combination of JFlex, CUP, and java. We run the JFlex and CUP compilers on the JFlex and CUP programs to generate Java. We run the Java compiler to generate class files.

Then, for every input for every program we wish to execute, we run the Java interpreter, to interpret the Java class files, which analyse the program we want to interpret, written in the language we have implemented, and then interpret this program, and process the input, and generate output.



Implementing a Compiler

Suppose we want to implement a compiler. We can write the compiler in a combination of JFlex, CUP, and java. We run the JFlex and CUP compilers on the JFlex and CUP programs to generate Java. We run the Java compiler to generate class files.

Then, for every program we wish to compile, 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.

Then we run an assembler to assemble the assembly language and generate a relocatable object file. Then we run a linker to combine the relocatable object file with library relocatable object files, and generate an executable code file.

Then, for every input for every program we wish to execute, we run the executable code file to process the input, and generate output.

Running JFlex and CUP

JFlex and CUP are written in Java. The class files that make up these programs can be packed into jar files, JFlex.jar and java_cup.jar. The manifests in these jar files specify the main class, so they can be run without specifying the main class. Moreover, because JFlex can be run without any parameters, it is often possible to run it by double clicking on it. The parser generated by CUP makes use of a sub-package of the java_cup package. These files are packed into a jar file java_cup_runtime.jar. They perform the parsing, using the tables generated by CUP. Install these jar files on your system, by obtaining the directory LIB330 from the 330 resources web page, containing JFlex.jar, java_cup.jar, and java_cup_runtime.jar (if you have not already done so by obtaining the gzipped tar file COPYTOHOME.tar.gz, which includes LIB330). If not already done, set a shell variable, LIB330, in your .bash_profile file to the path for this directory. For example,

```
LIB330=$HOME/LIB330
```

To run JFlex on UNIX, we could write

```
java -jar "$LIB330/JFlex.jar" Source/grammar/Ylex.jflex
```

To run CUP on UNIX, we could write

```
java -jar "$LIB330/java_cup.jar" \
-expect 0 -progress -source "Source/grammar" \
-input "parser.cup"
```

To compile the resultant Java source, together with Java source we wrote ourselves on UNIX, we could write

```
javac -d Classes -classpath "$LIB330/java_cup_runtime.jar" \
     -sourcepath "Source" Source/Main.java
```

This causes the compiled class files to be saved in the directory Classes.

To run the resultant program on UNIX, we could write

```
java -classpath "Classes:$LIB330/java_cup_runtime.jar" \
     Main -dir Programs/exampleDir
```

To create a jar file (excluding the java_cup runtime), we could write

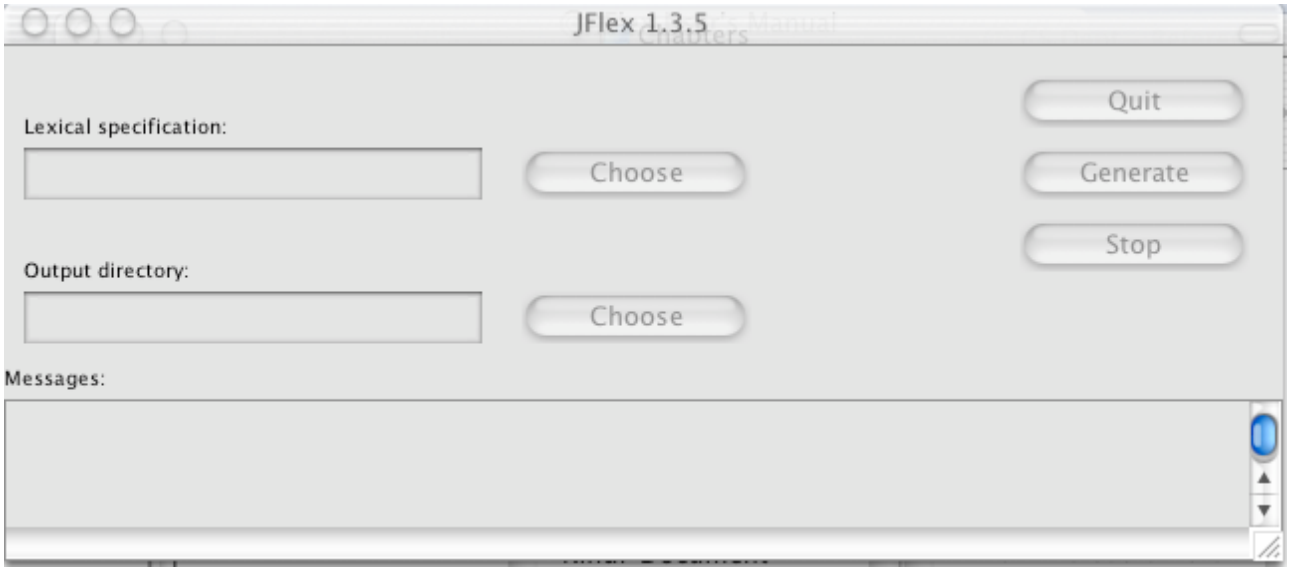
```
cd Classes
jar cvf ../run.jar `find . -name "*.class"`
```

To run the resultant program on UNIX, using the jar file, we could write

```
java -classpath "run.jar:$LIB330/java_cup_runtime.jar" \
     Main -dir Programs/exampleDir
```

Parameters for the JFlex command

JFlex can be run without any parameters. In this case it generates a GUI window. The GUI window prompts for a JFlex file to analyse, and an output directory in which to create the Java file.



JFlex can also be run with a list of parameters specifying options and input files to analyse. The parameters are placed after specification of the jar file or class path and main class.

Options for JFlex include

-d <directory>

Specifies the directory to place the generated files in. The default is the same directory as the input file.

Using Cygwin in the Laboratory

Start up Cygwin by selecting "Cygwin Bash Shell" from the start menu at the bottom left of the screen.

When you start up Cygwin in the undergraduate laboratory, it should start in your home directory, and execute your .bash_profile file, which should be in this directory. Exactly where your home directory is specified to be depends on the way the lab has been set up by the technical staff. You should have a personal directory on a server mounted as your "H:" drive. I think your home directory is currently the sfac_apps/cyghome subdirectory within your "H:" drive. You can refer to this as "/home/sfac_apps/cyghome" or "/cygdrive/h/sfac_apps/cyghome" in Cygwin, or "H:\sfac_apps\cyghome" in DOS.

Setting up your home directory for Cygwin

The value of the shell variable HOME indicates the name of your home directory. Start up Cygwin, and type

```
echo $HOME
```

to determine the name of your home directory. In the undergraduate lab, it should be "/home/sfac_apps/cyghome". It will be different on your own machine, where it will be "/home/YourLoginName".

Obtain the file COPYTOHOME.tar.gz from the resources web page for CompSci 210.

Save it somewhere. Change into this directory by typing

```
cd dirName
```

Untar the file COPYTOHOME.tar.gz using the command

```
tar -x -z -f COPYTOHOME.tar.gz
```

in Cygwin, when in the directory containing COPYTOHOME.tar.gz. This creates a subdirectory called COPYTOHOME. This directory contains: a suitable .bash_profile file for Bash setup; a .exrc file for vi editor setup; a bin directory with suitable files for running the Alpha simulator, converting between Windows and UNIX filenames, printing the pathname of commands, etc.; a directory with useful shell scripts for converting file between different formats, the jar files needed to run JFlex and CUP, etc.

Quit from Cygwin. Move the contents of COPYTOHOME (including .bash_profile and .exrc) into your actual home directory (/home/sfac_apps/cyghome). When you start up Cygwin later, so long as the disk containing your home directory is mounted, you should automatically start up in your home directory and run your .bash_profile file.

Changing the notion of home directory for Cygwin

If the network is down, you might need to temporarily change the directory you consider to be your home directory. You might even want to have a different home directory for each course, with a different setup for each.

First, ensure that a copy of .bash_profile exists in the directory you want to be your home directory.

To change where your home directory is, change to the appropriate directory. Then type

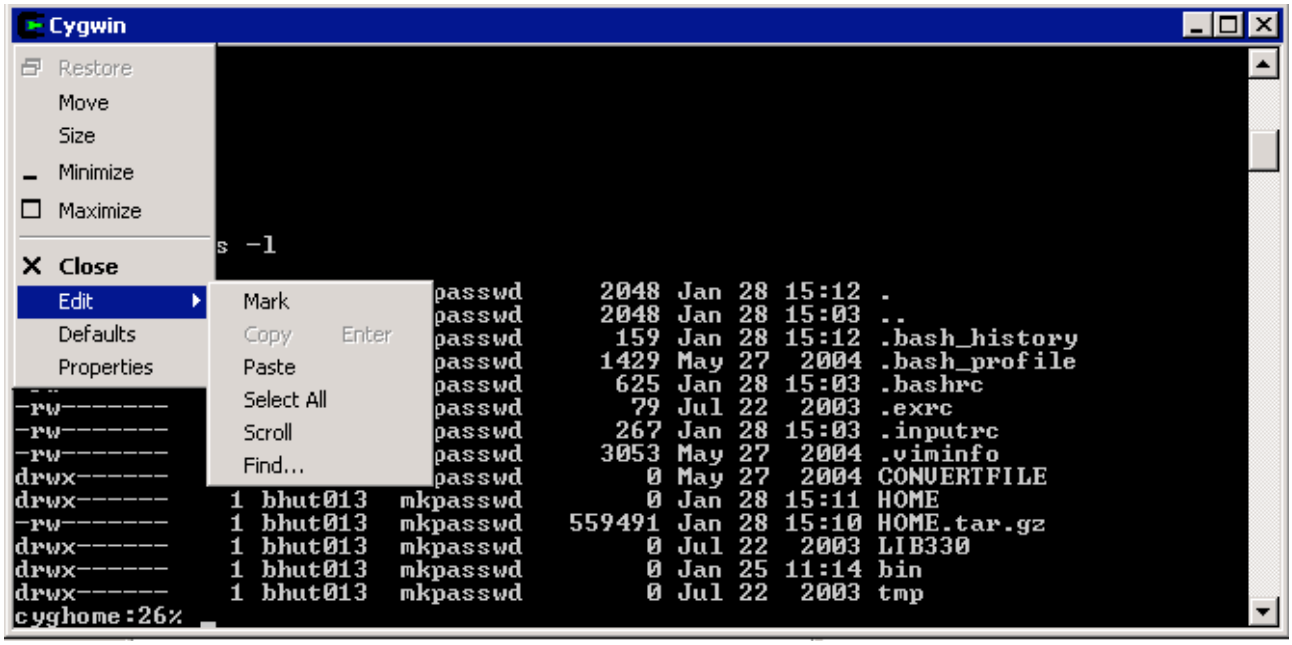
```
HOME=`pwd`
source .bash_profile
```

The first line sets the shell variable HOME to the appropriate place. The second line runs your .bash_profile file. You will have to do this every time you start up Cygwin, if you want a different notion of home directory from the standard one.

Copying and Pasting in Cygwin

When executing commands in UNIX, and hence also in Cygwin, it is possible to use file redirection to save the output as a file.

It is also possible to copy a rectangular area of the Cygwin “terminal” window then insert the text into a Windows application. Similarly, text can be pasted into the Cygwin “terminal” window. A menu associated with the Cygwin “terminal” window can be used to perform the copy or paste. When copying, you have to select the complete rectangular area you want to copy (including the right hand side), not just the lines you want. Select Edit/Mark, then drag across the rectangle, then type return.



Running Windows programs from Cygwin

There are additional problems when running Windows programs from Cygwin. Cygwin is essentially a version of UNIX built on top of Windows, and it uses UNIX style pathnames. However, Java and Javac are Windows commands, and hence they require Windows pathnames. It is possible to run Windows commands from Cygwin, but the pathnames have to be converted to Windows format. In Cygwin, we refer to drives such as the C drive by "/cygdrive/c", while in Windows we write "C:". In Cygwin, we use "/" as a path separator, while in Windows we use "\". So "/cygdrive/h/sfac_apps/cyghome/330PROGRAMS/LEX" in Cygwin becomes "H:\sfac_apps\cyghome\330PROGRAMS\LEX" in Windows. Fortunately, there is a conversion command, cygpath, provided to perform the conversion.

```
cygpath -w pathName
```

converts a UNIX pathname to Windows format, and

```
cygpath -u pathName
```

converts a Windows pathname to UNIX format. If you don't enclose the file name in '...', you will have to escape \ for it to work.

Another minor requirement for Cygwin shell scripts is that they have to start with a specification of the shell command to execute. For example, if the path for bash is /bin/bash we might have a first line

```
#! /bin/bash
```

If your machine does not have the bash command in this directory, and you have administrator access, create a link in /bin, that links to the actual bash executable file (bash.exe).

If we want shell scripts that work on both UNIX and Cygwin, we could have a toNative.bash command that just echoes its arguments on UNIX, but converts its argument pathnames to Windows format on Windows. It is necessary to avoid file names with spaces in them, because the quoting of file names is lost. Alternatively, set `IFS=$'\n'` in the shell script that uses it, so that it only uses line breaks as word separators.

```
#! /bin/bash

while (( $# > 0 ))
do
     if [[ "$OSTYPE" == "cygwin" ]]
     then
          cygpath -w "$1"
     else
          echo "$1"
     fi
     shift
done
```

Shell scripts to run JFlex, CUP, javac, etc

To decrease the amount of typing, and allow machine independent commands, we can use Bash shell scripts to run JFlex, CUP, javac, and java.

To run JFlex, we could have the command createlexer.bash

```
#! /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
```

Error messages and summary information will be placed in the file jflex.error, which you should view, to make sure that everything worked.

To run CUP, we could have the command createparser.bash

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

SOURCEDIR=`toNative.bash "Source/grammar"`
CCUPJAR=`toNative.bash "$LIB330/java_cup.jar"`

java -jar "$CCUPJAR" -nonterms \
-expect 0 -progress -source "$SOURCEDIR" \
-dump -dumpto "parser.states" \
-input "parser.cup" &> cup.error
```

Error messages and summary information will be placed in the file cup.error, which you should view, to make sure that everything worked.

To compile the resultant Java source, together with Java source we wrote ourselves, we could have the command createclass.bash

```
#! /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
```

Error messages and summary information will be placed in the file javac.error, which you should view, to make sure that everything worked.

To create a jar file containing the contents of Classes we could have the command createjar.bash

```
#! /bin/bash
rm -f run.jar
cd Classes
MANIFEST=`toNative.bash ../manifest`
JARFILE=`toNative.bash ../run.jar`
UNIXCLASSES=`find . -name "*.class"`
NATIVECLASSES=`toNative.bash $UNIXCLASSES`
jar cmf $MANIFEST $JARFILE $NATIVECLASSES
```

To perform all of these tasks together, we could have the command createcompiler.bash

```
#! /bin/bash

createlexer.bash
createparser.bash
createclass.bash
createjar.bash
```

To run the resultant program, specifying a directory containing the input, error, and output file, we could have the command run.bash

```
#! /bin/bash

#    ulimit -t 10

DIR="$1"
echo "$DIR"
CCUPJAR=`toNative.bash "$LIB330/java_cup_runtime.jar"`
NATIVEDIR=`toNative.bash "$DIR"`
rm -f "$DIR/program.err" "$DIR/program.print" "$DIR/program.out"
java -classpath "run.jar$CPSEP$CCUPJAR" Main -dir "$NATIVEDIR"
```

The command ulimit is used to limit the total resources the command can use (in this case it limits the CPU usage to 10 seconds). In fact it doesn't work on Cygwin, because the -t option is not supported.

The shell variable $CPSEP is set to the class path separator, ":" on UNIX, or ";" on Windows.

To run the resultant program, on all subdirectories of the Programs directory, we could have the command runall.bash

```
#! /bin/bash

COMMAND="run.bash"
DIR="$1"
if [ "${DIR}" == "" ]
then
     DIR=Programs
fi
for subDir in "${DIR}"/*
do
     "${COMMAND}" ${subDir}
done
```

Error messages and output will be placed in files of the form program.err and program.out, within the subdirectory, which you should view, to make sure that everything worked.

Matching comments using JFlex (Refer STRIP1)

I often use JFlex to clean up a text file in some way. For example, I remove the bodies of methods, remove comments, etc. I wanted to extract the grammar from a Java CUP grammar definition. This involved deleting text enclosed within {:...:}, deleting comments, and deleting empty lines. The following JFlex program achieves this.

```
package grammar;

import java.io.*;

%%

%{
     boolean printed = false;

     void echo( String text ) {
          System.out.print( text );
          printed = true;
          }
%}

%init{
     yybegin( NORMAL );
%init}

%public
%type Void

%state NORMAL COMMENT CODESTRING

newline        =    (\r|\n|\r\n)
%%
<NORMAL> {
     "/*"           { yybegin( COMMENT ); }
     "{:"           { yybegin( CODESTRING ); }
     {newline}      {
                    if ( printed ) {
                         System.out.println();
                         printed = false;
                         }
                    }
     :[A-Za-z0-9]+ { }
     .              { echo( yytext() ); }
     }
<COMMENT> {
     "*/"           { yybegin( NORMAL ); }
     {newline}      { }
     .              { }
     }
<CODESTRING> {
     ":}"           { yybegin( NORMAL ); }
     {newline}      { }
     .              { }
     }
```

Matching comments is difficult in many lexical analyser generators, because of the way the lexical analyser generated matches text when there are several alternative ways of matching it. The lexical analyser matches the longest possible text. This is exactly what is wanted for numbers and identifiers, but the opposite of what is wanted for string constants and comments, where it is the shortest possible match that is wanted.

Ideally, we would like to describe a C/Java comment as

"/*"(.|\n)*"*/"

However, this would match from the beginning of the first comment to the end of the last comment.

The way to match comments is to match the beginning of the comment, namely "/*" as a token, then change into a comment state. In the comment state, if we match "*/", we change back to our normal state. Otherwise, we match a single character and do nothing. Again, because we match the longest pattern, we will match "*/" in preference to matching "*" and "/" as separate tokens.

In fact JFlex does have a special construct to support the matching of comments, but Lex and JLex do not have this feature.

We can precede a pattern by "~" (pronounced "up to"), to specify that we want to match all text up to the first occurrence of the pattern. For example "/*"~"*/" can be used to match a C/Java comment. This is one of the little features that make JFlex just a little better than its competitors. (Refer STRIP2)

```
package grammar;

import java.io.*;

%%

%{
     boolean printed = false;

     void echo( String text ) {
          System.out.print( text );
          printed = true;
          }
%}

%public
%type Void

newline        =    (\r|\n|\r\n)
%%
"/*"~"*/"      { }
"{:"~":}"      { }
{newline}      {
                    if ( printed ) {
                        System.out.println();
                        printed = false;
                        }
                    }
:[A-Za-z0-9]+ { }
.                   { echo( yytext() ); }
```

However, this feature is not as useful as it might appear. It does not cope with such things as nested comments (where we have to keep an indication of the nesting level, so that we know when to go back to normal processing). It also does not allow us to count line breaks (although the number of line breaks can be obtained from the yyline variable). (Refer STRIP3)

```
package grammar;

import java.io.*;

%%

%{
```

```
     int commentNest = 0;

     int lineCount =1;

     boolean printed = false;

     void echo( String text ) {
          System.out.print( text );
          printed = true;
          }
%}

%init{
     yybegin( NORMAL );
%init}

%public
%type Void

%state NORMAL COMMENT CODESTRING

newline        =    \r|\n|\r\n
%%

<NORMAL> {
     "/*"           {
                    yybegin( COMMENT );
                    commentNest++;
                    }
     "{:"           { yybegin( CODESTRING ); }
     {newline}      {
                    if ( printed ) {
                         System.out.println();
                         printed = false;
                         }
                    lineCount++;
                    }
     :[A-Za-z0-9]+ { }
     .              { echo( yytext() ); }
     }

<COMMENT> {
     "/*"           { commentNest++; }
     "*/"           {
                    --commentNest;
                    if ( commentNest == 0 )
                         yybegin( NORMAL );
                    }
     {newline}      { lineCount++; }
     .              { }
     }

<CODESTRING> {
     ":}"           { yybegin( NORMAL ); }
     {newline}      { lineCount++; }
     .              { }
     }
```

Matching Identifiers and Reserved Words and Interfacing JFlex with CUP

(Refer INTERP3)

Of course, lexical analysers are not normally used by themselves.

JFlex is normally used as part of a compiler. Usually the parser invokes yylex() whenever it needs the next token, and yylex() returns a single token, rather than consuming all the input. The standard way of using JFlex, is to put a return statement in the action for those tokens that are syntactically important (namely those other than white space, newlines, comments, etc). Tokens that are ignored by the parser, have an action that does not return. In our previous examples, there are no return statements, so yylex() consumes all tokens, and performs an action for each one.

The following code represents the lexical analyser portion of a program that analyses a simple language. The parser is written using Java CUP, a parser generator. CUP assumes that the lexical analyser returns a value of type Symbol. Symbol is a class with fields

sym	An integer representing the symbol type of the token.
value	The value of the token, of type Object. (The actual value can be of any type that extends Object, and hence any class).
left	The left position of the token in the original input file.
right	The right position of the token in the original input file.

We can return the value as a String, since String extends Object. If we want to return an int, char or double, we have to package it in a wrapper class such as Integer, Character or Double.

The symbol type is an integer constant. A class called sym is generated by CUP, with definitions of constants for each kind of token. (The designer of CUP obviously doesn't follow the Java conventions of upper case for the start of a class name.)

Reserved words are lexically the same as identifiers. One way of doing lexical analysis is to put the reserved words in a table, match everything as an identifier, and search the table to determine whether the token is really a reserved word. The action for an identifier can then return a token type that indicates the kind of reserved word, or IDENT if the token is an identifier. So long as we do not have an excessive number of reserved words, it is also possible to use JFlex to perform the separation.

The lexical analyser generated by JFlex resolves conflicts by matching the longest possible text. This is needed to guarantee that if we have the text for an identifier, the JFlex lexical analyser matches the whole text, and not just a portion of the text. It resolves conflicts between matches of the same length, by preferring the first rule. We do not want reserved words to be treated as identifiers. We can use JFlex's conflict resolution policy to do what we want, so long as we put the rules for the reserved words before the rules for identifiers. Thus "while" will be treated as a reserved word, rather than an identifier. The preference for the longest match also means that "integral" is interpreted as an identifier, rather than the reserved word "int" and the identifier "egral".

Note that the JFlex program has to import java_cup.runtime.*, since that is where the Symbol class is declared.

```
package grammar;

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

%%
```

```
%public
%type         Symbol
%char

%{
     private int lineNumber = 1;
     public int lineNumber() { return lineNumber; }

     public Symbol token( int tokenType ) {
          Print.error().println( "Obtain token "
               + sym.terminal_name( tokenType )
               + " \"" + yytext() + "\"" );
          return new Symbol( tokenType, yychar,
               yychar + yytext().length(), yytext() );
          }

%}

number        =    [0-9]+
ident         =    [A-Za-z][A-Za-z0-9]*
space         =    [\ \t]
newline       =    \r|\n|\r\n

%%

"="           { return token( sym.ASSIGN ); }
"+"           { return token( sym.PLUS ); }
"-"           { return token( sym.MINUS ); }
"*"           { return token( sym.TIMES ); }
"/"           { return token( sym.DIVIDE ); }
"("           { return token( sym.LEFT ); }
")"           { return token( sym.RIGHT ); }
"<"           { return token( sym.LT ); }
"<="          { return token( sym.LE ); }
">"           { return token( sym.GT ); }
">="          { return token( sym.GE ); }
"=="          { return token( sym.EQ ); }
"!="          { return token( sym.NE ); }
"if"          { return token( sym.IF ); }
"then"        { return token( sym.THEN ); }
"else"        { return token( sym.ELSE ); }
"while"       { return token( sym.WHILE ); }
"do"          { return token( sym.DO ); }
"{"           { return token( sym.LEFTCURLY ); }
"}"           { return token( sym.RIGHTCURLY ); }
";"           { return token( sym.SEMICOLON ); }
{newline}     { lineNumber++; }
{space}       { }

{number}      { return token( sym.NUMBER ); }
{ident}       { return token( sym.IDENT ); }

<<EOF>>       { return token( sym.EOF ); }

.             { return token( sym.error ); }
```

Note how the rules corresponding to syntactically significant tokens have return statement, while the rules corresponding to comments and white space do not return. The lexical analyser loops until it gets a syntactically significant token, then returns that token.

<<EOF>> is a notation used to represent end of file. The default action generated by JFlex is to return a null value. However, CUP wants a Symbol returned, with the token kind of sym.EOF.

CUP generates the class sym, automatically. It invents numbers for the token kind. We can use the symbolic values in our lexical analyser.

```
package Parser;

/** CUP generated class containing symbol constants. */
public class sym {
     /* terminals */
     public static final int TIMES = 6;
     public static final int LT = 10;
     public static final int NE = 15;
     public static final int IDENT = 24;
     public static final int ELSE = 18;
     public static final int SEMICOLON = 9;
     public static final int PLUS = 4;
     public static final int THEN = 17;
     public static final int WHILE = 19;
     public static final int IF = 16;
     public static final int GT = 12;
     public static final int LE = 11;
     public static final int DO = 20;
     public static final int RIGHT = 3;
     public static final int LEFT = 2;
     public static final int NUMBER = 23;
     public static final int EOF = 0;
     public static final int DIVIDE = 7;
     public static final int GE = 13;
     public static final int MINUS = 5;
     public static final int error = 1;
     public static final int ASSIGN = 8;
     public static final int EQ = 14;
     public static final int RIGHTCURLY = 22;
     public static final int LEFTCURLY = 21;

     /* nonterminals */
     static final int Program = 1;
     static final int Factor = 7;
     static final int Term = 6;
     static final int Stmt = 3;
     static final int Expr = 5;
     static final int BoolExpr = 4;
     static final int $START = 0;
     static final int StmtList = 2;
     ...
}
```

The main program to go with this parser is as follows. It creates an instance of the parser. I have extended the parser, by adding a constructor that takes the input file as a parameter. I get the parser to perform parsing, by invoking the parse method.

```
import java.io.*;
import java_cup.runtime.*;
import runEnv.*;
import node.*;
import node.stmtNode.*;
import grammar.*;
import text.*;

public class Main {
```

```
    public static void main( String[] argv ) {
        String dirName = null;

        try {
            for ( int i = 0; i < argv.length; i++ ) {
                if ( argv[ i ].equals( "-debug" ) ) {
                    Print.DEBUG = true;
                    }
                else if ( argv[ i ].equals( "-dir" ) ) {
                    i++;
                    if ( i >= argv.length )
                        throw new Error( "Missing directory name" );
                    dirName = argv[ i ];
                    }
                else {
                    throw new Error(
                        "Usage: java Main [-debug] -dir directory" );
                    }
                }

            if ( dirName == null )
                throw new Error( "Directory not specified" );

            System.setErr( new PrintStream( new FileOutputStream(
                new File( dirName, "program.parse" ) ) ) );
            Print.setError( new File( dirName, "program.err" ) );
            Print.setReprint( new File( dirName, "program.print" ) );
            Print.setInterp( new File( dirName, "program.out" ) );

            parser p = new parser( new File( dirName, "program.in" ) );
            StmtListNode program = ( StmtListNode ) p.parse().value;
            Print.error().println( "Reprinting ... " );
            Print.reprint().println( program );
            Print.error().println( "Evaluate ... " );
            program.eval( new RunEnv() );

            }
        catch ( Exception e ) {
            Print.error().println( "Exception at " );
            e.printStackTrace();
            }
        }
    }
```

The parser is defined using CUP. The CUP compiler is used to translate the grammar definition into a Java program. The lexical analyser is invoked by the parser, every time it needs a new token. The connection between the parser and lexical analyser is in the lines

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

below, that specify that whenever the parser needs a new token, it should invoke the yylex() method of the lexical analyser. This occurs somewhere in the depths of the CUP runtime library code for the parser.

I open the input file once for the lexical analyser, and once for use when generating error messages, and wanting to reprint the text surrounding the error token.

```
package grammar;

import node.*;
```

```
import node.stmtNode.*;
import node.exprNode.*;
import node.exprNode.prefixNode.*;
import node.exprNode.valueNode.*;
import node.exprNode.binaryNode.*;
import node.exprNode.binaryNode.arithNode.*;
import node.exprNode.binaryNode.relationNode.*;
import text.*;

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

parser code
     {:
     private Yylex lexer;
     private File file;

     public parser( File file ) {
          this();
          this.file = file;
          try {
               lexer = new Yylex( new FileReader( file ) );
               }
          catch ( IOException exception ) {
               throw new Error( "Unable to open file \"" + file + "\"" );
               }
          }
...
scan with
     {:
     return  lexer.yylex();
     :};

terminal LEFT, RIGHT, PLUS, MINUS, TIMES, DIVIDE, ASSIGN, SEMICOLON;
terminal LT, LE, GT, GE, EQ, NE, IF, THEN, ELSE, WHILE, DO, LEFTCURLY,
RIGHTCURLY;
terminal String NUMBER;
terminal String IDENT;

nonterminal StmtListNode StmtList;
nonterminal StmtNode Stmt;
nonterminal ExprNode BoolExpr, Expr, Term, Factor;

start with StmtList;

StmtList::=
          {:
          RESULT = new StmtListNode();
          :}
     |
          StmtList:stmtList Stmt:stmt
          {:
          stmtList.addElement( stmt );
          RESULT = stmtList;
          :}
     ;

Stmt::=
          IDENT:ident ASSIGN Expr:expr SEMICOLON
          {:
          RESULT = new AssignStmtNode( ident, expr );
```

```
            :}
        |
            IF BoolExpr:expr THEN Stmt:stmt1 ELSE Stmt:stmt2
            {:
            RESULT = new IfThenElseStmtNode( expr, stmt1, stmt2 );
            :}
        |
            IF BoolExpr:expr THEN Stmt:stmt1
            {:
            RESULT = new IfThenStmtNode( expr, stmt1 );
            :}
        |
            WHILE BoolExpr:expr DO Stmt:stmt1
            {:
            RESULT = new WhileStmtNode( expr, stmt1 );
            :}
        |
            LEFTCURLY StmtList:stmtList RIGHTCURLY
            {:
            RESULT = new CompoundStmtNode( stmtList );
            :}
        |
            error SEMICOLON
            {:
            RESULT = new ErrorStmtNode();
            :}
        |
            error RIGHTCURLY
            {:
            RESULT = new ErrorStmtNode();
            :}
        ;

BoolExpr::=
            Expr:expr1 LT Expr:expr2
            {:
            RESULT = new LessThanNode( expr1, expr2 );
            :}
        |
            Expr:expr1 LE Expr:expr2
            {:
            RESULT = new LessEqualNode( expr1, expr2 );
            :}
        |
            Expr:expr1 GT Expr:expr2
            {:
            RESULT = new GreaterThanNode( expr1, expr2 );
            :}
        |
            Expr:expr1 GE Expr:expr2
            {:
            RESULT = new GreaterEqualNode( expr1, expr2 );
            :}
        |
            Expr:expr1 EQ Expr:expr2
            {:
            RESULT = new EqualNode( expr1, expr2 );
            :}
        |
            Expr:expr1 NE Expr:expr2
            {:
```

```
          RESULT = new NotEqualNode( expr1, expr2 );
          :}
     ;

Expr::=
          Expr:expr PLUS Term:term
          {:
          RESULT = new PlusNode( expr, term );
          :}
     |
          Expr:expr MINUS Term:term
          {:
          RESULT = new MinusNode( expr, term );
          :}
     |
          MINUS Term:term
          {:
          RESULT = new NegateNode( term );
          :}
     |
          Term:term
          {:
          RESULT = term;
          :}
     ;

Term::=
          Term:term TIMES Factor:factor
          {:
          RESULT = new TimesNode( term, factor );
          :}
     |
          Term:term DIVIDE Factor:factor
          {:
          RESULT = new DivideNode( term, factor );
          :}
     |
          Factor:factor
          {:
          RESULT = factor;
          :}
     ;

Factor::=
          LEFT Expr:expr RIGHT
          {:
          RESULT = expr;
          :}
     |
          NUMBER:value
          {:
          RESULT = new NumberNode( new Integer( value ) );
          :}
     |
          IDENT:ident
          {:
          RESULT = new IdentNode( ident );
          :}
     ;
```