JFlex User's Manual

Version 1.2.2, August 23, 1999

Gerwin Klein

Contents

1 Introduction

JFlex is a lexical analyzer generator for Java[1] written in Java. It is also a rewrite of the very useful tool JLex [3] which was developed by Elliot Berk at Princeton University. As Vern Paxon states for his C/C++ tool flex [5]: They do not share any code though.

1.1 Design goals

The main design goals of JFlex are:

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

1.2 About this manual

This manual gives a brief but complete description of the tool JFlex. It assumes, that you are familiar with the issue of lexical analysis. [2], [1] and [11] provide a good introduction to this topic.

The next section of this manual describes installation procedures for JFlex. If you never worked with JLex or just want to compare a JLex and a JFlex scanner specification you should also read section 3. All options and the complete specification syntax are presented in "Lexical specifications" (section 4). If you are interested in performance considerations and comparing JLex vs. JFlex speed, "a few words on performance" (section 5) might be just right for you. Those, who want to use their old JLex specifications may want to check out section 6.1 to avoid possible problems with not portable or non standard JLex behavior that has been fixed in JFlex. Section 6.2 talks about porting scanners from the Unix tools lex and flex. Interfacing JFlex scanners with the LALR parser generator CUP is explained in section 7. Section 8 gives a list of currently known bugs. The manual continues with notes about "Copying and License" (section 9) and concludes with references.

2 Installing and Running JFlex

2.1 Installing JFlex

To install JFlex, follow these three steps:

[1] Java is a trademark of Sun Microsystems, Inc., and refers to Sun's Java programming language. JFlex is not sponsored by or affiliated with Sun Microsystems, Inc.

1. Unzip the file you downloaded into the directory you want JFlex in (using `tar/unzip` for Unix or WinZip[2] for W95/98). If you unzipped it to say `C:\`, the following directory structure should be generated:

```
C:\JFlex\
      +--bin\                    (start scripts)
      +--doc\                    (FAQ and this manual)
      +--examples\
               +--java\          (Java 1.1 lexer specification)
               +--simple\        (example scanner)
               +--standalone\    (a simple standalone scanner)
      +--lib\                    (the precompiled classes)
      +--src\
          +--JFlex\              (source code of JFlex)
          +--JFlex\gui           (source code of JFlex UI classes)
          +--java_cup\runtime\   (source code of cup runtime classes)
```

2. Edit the file `bin/jflex` for Unix or `bin\jflex.bat` for W95/98 (in the example above it's `C:\JFlex\bin\flex.bat`) such that

   - `JAVA_HOME` contains the directory where your Java JDK is installed (for instance `C:\java`, such that `C:\java\lib\classes.zip` is the file of the standard Java classes) and
   - `JFLEX_HOME` the directory that contains JFlex (in the example: `C:\JFlex`)

3. Include the `bin/` directory of JFlex in your path. (the one that contains the start scripts, in the example: `C:\JFlex\bin`).

2.2 Running JFlex

You run JFlex with:

```
jflex <options> <inputfiles>
```

If you have JDK 1.2, you can start up the JFlex GUI by a double click on the JAR-file.

It is also possible to skip steps 2 and 3 of the installation process and include the file `lib\JFlex.jar` in your `CLASSPATH` environment variable instead.

Then you run JFlex with:

```
java JFlex.Main <options> <inputfiles>
```

The input files and options are in both cases optional. If you don't provide a file name on the commandline, JFlex will pop up a window to ask you for one.

JFlex knows about the following options:

`-d <directory>`
: writes the generated file to the directory `<directory>`

[2] http://www.winzip.com

`-skel <file>`
: uses external skeleton `<file>`. This is mainly for JFlex maintenance and special low level customizations. Use only when you know what you are doing! JFlex comes with a skeleton file in the `src` directory that reflects exactly the internal, precompiled skeleton and can be used with the `-skel` option.

`-verbose` or `-v`
: display generation progress messages (enabled by default)

`-quiet` or `-q`
: display error messages only (no chatter about what JFlex is currently doing)

`-time`
: display time statistics about the code generation process. (Not very accurate)

`-help` or `-h`
: print a help message explaining options and usage of JFlex.

3 A simple Example: How to work with JFlex

To demonstrate what a lexical specification with JFlex looks like, this section presents a part of the specification for the Java language. The example does not describe the whole lexical structure of Java programs, but only a small and simplified part of it (some keywords, some operators, comments and only two kinds of literals). It also shows how to interface with the LALR parser generator CUP [8] and therefore uses a class `sym` (generated by CUP), where integer constants for the terminal tokens of the CUP grammar are declared. JFlex comes with a directory `examples`, where you can find a small standalone scanner that doesn't need other tools like CUP to give you a running example. The "`examples`" directory also contains a *complete* JFlex specification of the lexical stucture of Java programs together with the CUP parser specification for Java 1.1 by C. Scott Ananian, obtained from the CUP [8] website (it was modified to interface with the JFlex scanner). Both specifications adhere strictly to the Java Language Specification [7].

```
/* JFlex example: part of Java 1.0/1.1 language lexer specification */
import java_cup.runtime.*;

%%


%class Lexer
%unicode
%cup
%line
%column


%{
  StringBuffer string = new StringBuffer();
```

```
  private Symbol symbol(int type) {
    return new Symbol(type, yyline, yycolumn);
  }
  private Symbol symbol(int type, Object value) {
    return new Symbol(type, yyline, yycolumn, value);
  }
%}


LineTerminator = \r|\n|\r\n
InputCharacter = [^\r\n]
WhiteSpace     = {LineTerminator} | [ \t\f]

/* comments */
Comment = {TraditionalComment} | {EndOfLineComment} | {DocumentationComment}

TraditionalComment   = "/*" [^*] {CommentContent} "*"+ "/"
EndOfLineComment     = "//" {InputCharacter}* {LineTerminator}
DocumentationComment = "/**" {CommentContent} "*"+ "/"

CommentContent       = ( [^*] | \*+ [^/*] )*

Identifier = [:jletter:] [:jletterdigit:]*

DecIntegerLiteral = 0 | [1-9][0-9]*


%state STRING

%%


/* keywords */
<YYINITIAL> "abstract"          { return symbol(sym.ABSTRACT); }
<YYINITIAL> "boolean"           { return symbol(sym.BOOLEAN); }
<YYINITIAL> "break"             { return symbol(sym.BREAK); }


<YYINITIAL> {
  /* identifiers */
  {Identifier}                  { return symbol(sym.IDENTIFIER); }

  /* literals */
  {DecIntegerLiteral}           { return symbol(sym.INTEGER_LITERAL); }
  \"                            { string.setLength(0); yybegin(STRING); }

  /* operators */
  "="                           { return symbol(sym.EQ); }
  "=="                          { return symbol(sym.EQEQ); }
  "+"                           { return symbol(sym.PLUS); }

  /* comments */
  {Comment}                     { /* ignore */ }

  /* whitespace */
  {WhiteSpace}                  { /* ignore */ }
}
```

```
<STRING> {
  \"                             { yybegin(YYINITIAL);
                                   return symbol(sym.STRINGLITERAL,
                                   string.toString()); }
  [^\n\r\"\]+                    { string.append( yytext() ); }
  \\t                            { string.append('\t'); }
  \\n                            { string.append('\n'); }

  \\r                            { string.append('\r'); }
  \\"                            { string.append('\"'); }
  \\                             { string.append('\'); }
}


/* error fallback */
.|\n                             { throw new Error("Illegal character <"+
                                                   yytext()+">"); }
```

As with JLex, the specification consists of three parts, divided by %%:

- usercode,
- options and declarations and
- lexical rules.

3.1 Code to include

Let's take a look at the first section, "user code": The text up to the first line starting with %% is copied verbatim to the top of the generated lexer class (before the actual class declaration). Beside `package` and `import` statements there is usually not much to do here.

3.2 Options and Macros

The second section "options and declarations" is more interesting. It consists of a set of options, code that is included inside the generated scanner class, lexical states and macro declarations. Each JFlex option must begin a line of the specification and starts with a %. In our example the following options are used:

- `%class Lexer` tells JFlex to give the generated class the name "Lexer" and to write the code to a file "`Lexer.java`".
- `%unicode` defines the set of characters the scanner will work on.
- `%cup` switches to CUP compatibility mode to interface with a CUP generated parser.
- `%line` switches line counting on (the current line number can be accessed via the variable `yyline`)
- `%column` switches column counting on (current column is accessed via `yycolumn`)

The code included in %{...%} is copied verbatim into the generated lexer class source. Here you can declare member variables and functions that are used inside scanner actions. In our example we declare a `StringBuffer` "`string`" in which we will store parts of string literals and two helper functions "`symbol`" that create `java_cup.runtime.Symbol` objects with position information of the current token (see section 7 for how to interface with the parser generator CUP). As JFlex options, both %{ and %} must begin a line.

The specification continues with macro declarations. Macros are abbreviations for regular expressions, used to make lexical specifications easier to read and understand. A macro declaration consists of a macro identifier followed by =, then followed by the regular expression it represents. This regular expression may itself contain macro usages. Although this allows a grammar like specification style, macros are still just abbreviations and not non terminals -- they cannot be recursive or mutually recursive. Cycles in macro definitions are detected and reported at generation time by JFlex.

Here some of the example macros in more detail:

- `LineTerminator` stands for the regular expression that matches an ASCII CR, an ASCII LF or an CR followed by LF.
- `InputCharacter` stands for all characters that are not a CR or LF.
- `TraditionalComment` is the expression that matches the string `"/*"` followed by a character that is not a * followed by anything that matches the macro `CommentContent` followed by any number of * followed by /.
- `CommentContent` matches zero or more occurrences of any character except a * or any number of * followed by a character that is not a /
- `Identifier` matches each string that starts with a character of class `jletter` followed by zero or more characters of class `jletterdigit`. `jletter` and `jletterdigit` are predefined character classes. `jletter` includes all characters for which the Java function `Character.isJavaIdentifierStart` returns `true` and `jletterdigit` all characters for that `Character.isJavaIdentifierPart` returns `true`.

The last part of the second section in our lexical specification is a lexical state declaration: `%state STRING` declares a lexical state `STRING` that can be used in the "lexical rules" part of the specification. A state declaration is a line starting with `%state` followed by a space or comma separated list of state identifiers. There can be more than one line starting with `%state`.

3.3 Rules and Actions

The "lexical rules" section of a JFlex specification contains regular expressions and actions (Java code) that are executed when the scanner matches the associated regular expression. As the scanner reads its input, it keeps track of all regular expressions and activates the action of the expression that has the longest match. Our specification above for instance would with input "`breaker`" match the regular expression for `Identifier` and not the keyword "`break`" followed by the Identifier "`er`", because rule `{Identifier}` matches more of this

input at once (i.e. it matches all of it) than any other rule in the specification. If two regular expressions both have the longest match for a certain input, the scanner chooses the action of the expression that appears first in the specification. In that way, we get for input "`break`" the keyword "`break`" and not an Identifier "`break`".

Additional to regular expression matches, one can use lexical states to refine a specification. A lexical state acts like a start condition. If the scanner is in lexical state `STRING`, only expressions that are preceded by the start condition `<STRING>` can be matched. A start condition of a regular expression can contain more than one lexical state. It is then matched when the lexer is in any of these lexical states. The lexical state `YYINITIAL` is predefined and is also the state in which the lexer begins scanning. If a regular expression has no start conditions it is matched in *all* lexical states.

Since you often have a bunch of expressions with the same start conditions, JFlex allows the same abbreviation as the Unix tool `flex`:

```
<STRING> {
  expr1   { action1 }
  expr2   { action2 }
}
```

means that both `expr1` and `expr2` have start condition `<STRING>`.

The first three rules in our example demonstrate the syntax of a regular expression preceded by the start condition `<YYINITIAL>`.

```
<YYINITIAL> "abstract"  { return symbol(sym.ABSTRACT); }
```

matches the input "`abstract`" only if the scanner is in its start state "`YYINITIAL`". When the string "`abstract`" is matched, the scanner function returns the CUP symbol `sym.ABSTRACT`. If an action does not return a value, the scanning process is resumed immediatly after executing the action.

The rules enclosed in

```
<YYINITIAL> {
...
}
```

demonstrate the abbreviated syntax and are also only matched in state `YYINITIAL`.

Of these rules, one may be of special interest:

```
\"  { string.setLength(0); yybegin(STRING); }
```

If the scanner matches a double quote in state `YYINITIAL` we have recognized the start of a string literal. Therefore we clear our `StringBuffer` that will hold the content of this string literal and tell the scanner with `yybegin(STRING)` to switch into the lexical state `STRING`. Because we do not yet return a value to the parser, our scanner proceeds immediately.

In lexical state `STRING` another rule demonstrates how to refer to the input that has been matched:

```
[^\n\r\"]+  { string.append( yytext() ); }
```

The expression `[^\n\r\"]+` matches all characters in the input up to the next backslash

(indicating an escape sequence such as `\n`), double quote (indicating the end of the string), or line terminator (which must not occur in a string literal). The matched region of the input is referred to with `yytext()` and appended to the content of the string literal parsed so far.

The last lexical rule in the example specification is used as an error fallback. It matches any character in any state that has not been matched by another rule. It doesn't conflict with any other rule because it has the least priority (because it's the last rule) and because it matches only one character (so it can't have longest match precendence over any other rule).

3.4 How to get it going

- Install JFlex (see section 2)
- If you have written your specification file (or chosen one from the `examples` directory), save it (say under the name `java-lang.flex`).
- Run JFlex with

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

4 Lexical Specifications

As shown above, a lexical specification file for JFlex consists of three parts divided by a single line starting with %%:

```
UserCode
%%
Options and declarations
%%
Lexical rules
```

In all parts of the specification comments of the form `/* comment text */` and the Java style end of line comments starting with `//` are permitted. JFlex comments do nest - so the number of `/*` and `*/` should be balanced.

4.1 User code

The first part contains Usercode that is copied verbatim into the beginning of the source file of the generated lexer before the scanner class is declared. As shown in the example above, this is the place to put `package` declarations and `import` statements. It is possible, but not considered as good Java programming style to put own helper class (such as token classes) in this section. They should get their own `.java` file instead.

4.2 Options and declarations

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

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

```
%class "classname"
```

means that you start a line with `%class` followed by a space followed by the name of the class for the generated scanner (the double quotes are not to be entered, see section 3).

4.2.1 Class options and user class code

These options regard the name, the constructor and related parts of the generated scanner class.

- `%class "classname"`

  Tells JFlex to give the generated class the name "`classname`" and to write the generated code to a file "`classname.java`". If the `-d <directory>` command line option is not used, the code will be written to the directory where the specification file resides. If no `%class` directive is present in the specification, the generated class will get the name "`Yylex`" and will be written to a file "`Yylex.java`". There should be only one `%class` directive in a specification.

- `%implements "interface 1"[, "interface 2", ..]`

  Makes the generated class implement the specified interfaces. If more than one `%implements` directive is present, all the specified interfaces will be implemented.

- `%extends "classname"`

  Makes the generated class a subclass of the class “`classname`”. There should be only one `%extends` directive in a specification.

- `%public`

  Makes the generated class public (the class is only accessible in its own package by default).

- `%final`

  Makes the generated class final.

- `%abstract`

  Makes the generated class abstract.

- ```
  %{
  ...
  %}
  ```
  ```

The code enclosed in %{ and %} is copied verbatim into the generated class. Here you can define your own member variables and functions in the generated scanner. As all options, both %{ and %} must start a line in the specification. If more than one class code directive %{...%} is present, the code is concatenated in order of appearance in the specification.

- %init{
  ...
  %init}

  The code enclosed in %init{ and %init} is copied verbatim into the constructor of the generated class. Here, member variables declared in the %{...%} directive can be initialized. If more than one initializer option is present, the code is concatenated in order of appearance in the specification.

- %initthrow{
  "exception1"[, "exception2", ...]
  %initthrow}

  or (on a single line) just

  %initthrow "exception1" [, "exception2", ...]

  Causes the specified exceptions to be declared in the throws clause of the constructor. If more than one %initthrow{ ... %initthrow} directive is present in the specification, all specified exceptions will be declared.

4.2.2 Scanning method

This section shows how the scanning method can be customized. You can redefine the name and return type of the method and it is possible to declare exceptions that may be thrown in one of the actions of the specification. If no return type is specified, the scanning method will be declared as returning values of class Yytoken.

- %function "name"

  Causes the scanning method to get the specified name. If no %function directive is present in the specification, the scanning method gets the name "yylex". This directive overrides settings of the %cup switch. Please note that the default name of the scanning method with the %cup switch is next_token. Overriding this name might lead to the generated scanner being implicitly declared as abstract, because it does not provide the method next_token of the interface java_cup.runtime.Scanner. It is of course possible to provide a dummy implemention of that method in the class code section, if you still want to override the function name.

- %integer
  %int

  Both cause the scanning method to be declared as of Java type int. Actions in the specification can then return int values as tokens. The default end of file value under this setting is YYEOF, which is a public static final int member of the generated class.

- `%intwrap`

  Causes the scanning method to be declared as of the Java wrapper type `Integer`. Actions in the specification can then return `Integer` values as tokens. The default end of file value under this setting is `null`.

- `%type "typename"`

  Causes the scanning method to be declared as returning values of the specified type. Actions in the specification can then return values of `typename` as tokens. The default end of file value under this setting is `null`. If `typename` is not a subclass of `java.lang.Object`, you should specify another end of file value using the `%eofval{ ... %eofval}` directive or the `<<EOF>>` rule. The `%type` directive overrides settings of the `%cup` switch.

- ```
  %yylexthrow{
  "exception1"[, "exception2", ...  ]
  %yylexthrow}
  ```

  or (on a single line) just

  ```
  %yylexthrow "exception1" [, "exception2", ...]
  ```

  The exceptions listed inside `%yylexthrow{ ... %yylexthrow}` will be declared in the throws clause of the scanning method. If there is more than one `%yylexthrow{ ... %yylexthrow}` clause in the specification, all specified exceptions will be declared.

4.2.3 The end of file

There is always a default value that the scanning method will return when the end of file has been reached. You may however define a specific value to return and a specific piece of code that should be executed when the end of file is reached.

The default end of file values depends on the return type of the scanning method:

- For `%integer`, the scanning method will return the value `YYEOF`, which is a `public static final int` member of the generated class.
- For `%intwrap`,
- no specified type at all, or a
- user defined type, declared using `%type`, the value is `null`.
- In CUP compatibility mode, using `%cup`, the value is

  ```
  new java_cup.runtime.Symbol(sym.EOF)
  ```

User values and code to be executed at the end of file can be defined using these directives:

- ```
  %eofval{
  ...
  %eofval}
  ```

The code included in `%eofval{ ... %eofval}` will be copied verbatim into the scanning method and will be executed *each time* when the end of file is reached (this is possible when the scanning method is called again after the end of file has been reached). The code should return the value that indicates the end of file to the parser. There should be only one `%eofval{ ... %eofval}` clause in the specification. The `%eofval{ ... %eofval}` directive overrides settings of the `%cup` switch. As of version 1.2 JFlex provides a more readable way to specify the end of file value using the `<<EOF>>` rule (see also section 4.3.2).

- `%eof{`
  `...`
  `%eof}`

  The code included in `%{eof ...%eof}` will be executed exactly once, when the end of file is reached. The code is included inside a method `void yy_do_eof()` and should not return any value (use `%eofval{...%eofval}` or `<<EOF>>` for this purpose). If more than one end of file code directive is present, the code will be concatenated in order of appearance in the specification.

- `%eofthrow{`
  `"exception1"[,"exception2", ... ]`
  `%eofthrow}`

  or (on a single line) just

  `%eofthrow "exception1" [, "exception2", ...]`

  The exceptions listed inside `%eofthrow{...%eofthrow}` will be declared in the throws clause of the method `yy_do_eof()` (see `%eof` for more on that method). If there is more than one `%eofthrow{...%eofthrow}` clause in the specification, all specified exceptions will be declared.

- `%eofclose`

  Causes JFlex to close the input stream at the end of file. The code `yyclose()` is appended to the method `yy_do_eof()` (together with the code specified in `%eof{...%eof}`) and the exception `java.io.IOException` is declared in the throws clause of this method (together with those of `%eofthrow{...%eofthrow}`)

4.2.4 Standalone scanners

- `%debug`

  Creates a main function in the generated class that expects the name of an input file on the command line and then runs the scanner on this input file by printing each returned token to the Java console until the end of file is reached.

- `%standalone`

  Creates a main function in the generated class that expects the name of an input file on the command line and then runs the scanner on this input file. The values returned by the scanner are ignored, but any unmatched text is printed to the Java console instead

(as the C/C++ tool flex does, if run as standalone program). To avoid having to use an extra token class, the scanning method will be declared as having default type `int`, not `YYtoken` (if there isn't any other type explicitly specified). This is in most cases irrelevant, but could be useful to know when making another scanner standalone for some purpose. You should also consider using the `%debug` directive, if you just want to be able to run the scanner without a parser attached for testing etc.

4.2.5 CUP compatibility

You may also want to read section 7 if you are interested in how to interface your generated scanner with CUP.

- `%cup`

  The `%cup` directive enables the CUP compatibility mode and is equivalent to the following set of directives:

```
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
%eofval{
  return new java_cup.runtime.Symbol(sym.EOF);
%eofval}
%eofclose
```

4.2.6 Code generation

The following options define what kind of lexical analyzer code JFlex will produce. `%pack` is the default setting and will be used, when no code generation method is specified.

- `%switch`

  With `%switch` JFlex will generate a scanner that has the DFA hard coded into a nested switch statement. This method gives a good deal of compression in terms of the size of the compiled `.class` file while still providing very good performance. If your scanner gets to big though (say more than about 200 states) performance may vastly degenerate and you should consider using one of the `%table` or `%pack` directives. If your scanner gets even bigger (about 300 states), the Java compiler `javac` could produce corrupted code, that will crash when executed or will give you an `java.lang.VerifyError` when checked by the virtual machine. This is due to the size limitation of 64 KB of Java methods as described in the Java Virtual Machine Specification [9]. In this case you will be forced to use the `%pack` directive, since `%switch` usually provides more compression of the DFA table than the `%table` directive.

- `%table`

  The `%table` direction causes JFlex to produce a classical table driven scanner that encodes its DFA table in an array. In this mode, JFlex only does a small amount of

table compression (see [6], [10], [1] and [11] for more details on the matter of table compression) and uses the same method that JLex did up to version 1.2.1. See section 5 of this manual to compare these methods. The same reason as above (64 KB size limitation of methods) causes the same problem, when the scanner gets too big. This is, because the virtual machine treats static initializers of arrays as normal methods. You will in this case again be forced to use the `%pack` directive to avoid the problem.

- `%pack`

  `%pack` causes JFlex to compress the generated DFA table and to store it in a string literal. This string has to be unpacked when the first scanner object is created and initialized. After unpacking, the internal access to the DFA table is exactly the same as with option `%table` - the only extra work to be done at runtime is the unpacking process which is quite fast (not noticable in normal cases), is done only once at startup, and is in time complexity proportional to the size of the expanded DFA table. The unpacking process is static, i.e. it is done only once for a certain scanner class - no matter how often it is instantiated. Again, see section 5 on the performance of these scanners With `%pack`, there should be practically no limitation to the size of the scanner. `%pack` is the default setting and will be used, when no code generation method is specified.

4.2.7 Character sets

- `%7bit`

  Causes the generated scanner to expect 7 bit ASCII input files (character codes 0-127). Because this is the default value in JLex, JFlex also defaults to 7 bit scanners. If an input character with a code greater than 127 is encountered in an input at runtime, the scanner will throw an `ArrayIndexOutofBoundsException`. Not only because of this, you should consider using one of the `%full` or `%unicode` directives.

- `%full`
  `%8bit`

  Both options cause the generated scanner to expect 8 bit ASCII input files (character codes 0-255). If an input character with a code greater than 255 is encountered in an input at runtime, the scanner will throw an `ArrayIndexOutofBoundsException`.

- `%unicode`
  `%16bit`

  Both options cause the generated scanner to expect 16 bit Unicode input files (character codes 0-65535). There will be no runtime overflow when using this set of input characters.

- `%caseless`
  `%ignorecase`

  This option causes JFlex to handle all characters and strings in the specification as if they were specified in both uppercase and lowercase form. This enables an easy way to specify a scanner for a language with case insensitive keywords. The string "`break`" in a specificaion is for instance handled like the expression `([bB][rR][eE][aA][kK])`.

The `%caseless` option does not change the matched text and does not effect character classes. So `[a]` still only matches the character `a` and not `A`, too. Which letters are uppercase and which lowercase letters, is defined by the Unicode standard and determined by JFlex with the Java methods `Character.toUpperCase` and `Character.toLowerCase`.

4.2.8 Line, character and column counting

- `%char`

  Turns character counting on. The `int` member variable `yychar` contains the number of characters (starting with 0) from the beginning of input to the beginning of the current token.

- `%line`

  Turns line counting on. The `int` member variable `yyline` contains the number of lines (starting with 0) from the beginning of input to the beginning of the current token.

- `%column`

  Turns column counting on. The `int` member variable `yycolumn` contains the number of characters (starting with 0) from the beginning of the current line to the beginning of the current token.

4.2.9 Obsolete JLex options

- `%notunix`

  This JLex option is obsolete in JFlex but still recognized as valid directive. It used to switch between W95 and Unix kind of line terminators (`\r\n` and `\n`) for the `$` operator in regular expressions. JFlex always recognizes both styles of platform dependent line terminators.

- `%yyeof`

  This JLex option is obsolete in JFlex but still recognized as valid directive. In JLex it declares a public member constant `YYEOF`. JFlex declares it in any case.

4.2.10 State declarations

State declarations have the following from:

```
%state "state identifier" [, "state identifier", ...  ]
```

There may be more than one line of state declarations, each starting with `%state`. State identifiers are letters followed by a sequence of letters, digits or underscores. State identifiers can be separated by whitespace or comma.

The sequence

```
%state STATE1
%state STATE3, XYZ, STATE_10
%state ABC STATE5
```

declares the set of identifiers `STATE1, STATE3, XYZ, STATE_10, ABC, STATE5` as lexical states.

4.2.11 Macro definitions

A macro definition has the form

```
macroidentifier = regular expression
```

That means, a macro definition is a macro identifier (letter followed by a sequence of letters, digits or underscores), that can later be used to reference the macro, followed by optional whitespace, followed by an "=", followed by optional whitespace, followed by a regular expression (see section 4.3 for more information about regular expressions).

Each macro must fit on a single line.

The regular expression on the right hand side must be well formed and must not contain the `^`, `/` or `$` operators. **Differently to JLex, macros are not just pieces of text that are expanded by copying** - they are parsed and must be well formed.

This is a feature. It eliminates some very hard to find bugs in lexical specifications (such like not having parentheses around more complicated macros - which is not necessary with JFlex). See section 6.1 for more details on the problems of JLex style macros.

Since it is allowed to have macro usages in macro definitions, it is possible to use a grammar like notation to specify the desired lexical structure. Macros however remain just abbreviations of the regular expressions they represent. They are not non terminals of a grammar and cannot be used recursively in any way. JFlex detects cycles in macro definitions and reports them at generation time. JFlex also warns you about macros that have been defined but never used in the “lexical rules” section of the specification.

4.3 Lexical rules

The “lexical rules” section of an JFlex specification contains a set of regular expressions and actions (Java code) that are executed when the scanner matches the associated regular expression.

4.3.1 Syntax

The syntax of the "lexical rules" section is described by the following BNF grammar (terminal symbols are enclosed in ’quotes’):

```
LexicalRules ::= Rule+
Rule         ::= [StateList] ['^'] RegExp [LookAhead] Action
               | [StateList] '<<EOF>>' Action
```

```
              | StateGroup
StateGroup   ::= StateList '{' Rule+ '}'
StateList    ::= '<' Identifier (',' Identifier)* '>'
LookAhead    ::= '$' | '/' RegExp
Action       ::= '{' JavaCode '}' | '|'

RegExp       ::= RegExp '|' RegExp
               | RegExp RegExp
               | '(' RegExp ')'
               | RegExp ('*'|'+'|'?')
               | RegExp "{" Number ["," Number] "}"
               | '[' ['^'] (Character|Character'-'Character)+ ']'
               | PredefinedClass
               | '{' Identifier '}'
               | '"' StringCharacter+ '"'
               | Character

PredefinedClass ::= '[:jletter:]'
                  | '[:jletterdigit:]'
                  | '[:letter:]'
                  | '[:digit:]'
                  | '[:uppercase:]'
                  | '[:lowercase:]'
                  | '.'
```

The grammar uses the following terminal symbols:

- `JavaCode`
  a sequence of `BlockStatements` as described in the Java Language Specification [7], section 14.2.
- `Number`
  a non negative decimal integer.
- `Identifier`
  a letter `[a-zA-Z]` followed by a sequence of zero or more letters, digits or underscores `[a-zA-Z0-9_]`
- `Character`
  an escape sequence or any unicode character that is not one of these meta characters:
  `| ( ) { } [ ] < > \ . * + ? ^ $ / . "`
- `StringCharacter`
  an escape sequence or any unicode character that is not one of these meta characters:
  `\ "`
- An escape sequence

  – `\n \r \t \f \b`

- a `\x` followed by two hexadecimal digits `[a-fA-F0-9]` (denoting a standard ASCII escape sequence),
- a `\u` followed by four hexadecimal digits `[a-fA-F0-9]` (denoting an unicode escape sequence),
- a backslash followd by a three digit octal number from 000 to 377 (denoting a standard ASCII escape sequence), or
- a backslash followed by any other unicode character that stands for this character.

Please note, that the `\n` escape sequence stands for the ASCII LF character - not for the end of line. If you want to match the end of line, you should use the expression `\r|\n|\r\n` to take into account the different end of line standards on the platforms supported by Java.

As with version 1.1 of JFlex the whitespace characters `" "` (space) and `"\t"` (tab) can be used to improve the readability of regular expressions. They will be ignored by JFlex. In character classes and strings however, whitespace characters keep standing for themselfes (so the string `" "` still matches exactly one space character and `[ \n]` still matches an ASCII LF or a space character).

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

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

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

4.3.2 Semantics

This section gives an informal description of which text is matched by a regular expression (i.e. an expression described by the `RegExp` production of the grammar presented above).

A regular expression that consists solely of

- a `Character` matches this character.
- a character class `'[' (Character|Character'-'Character)+ ']'` matches any character in that class. A `Character` is to be considered an element of a class, if it is listed in the class or if its code lies within a listed character range `Character'-'Character`. So `[a0-3\n]` for instance matches the characters

  ```
  a 0 1 2 3 \n
  ```

- a negated character class `'[^' (Character|Character'-'Character)+ ']'` matches all characters not listed in the class.
- a string `'"' StringCharacter+ '" '` matches the exact text enclosed in double quotes. All meta characters but `\` and `"` loose their special meaning inside a string.

- a macro usage `'{' Identifier '}'` matches the input that is matched by the right hand side of the macro with name "`Identifier`".
- a predefined character class matches any of the characters in that class. There are the following predefined character classes:

  `.` contains all characters but `\n`.

  All other predefined character classes are defined in the Unicode specification or the Java Language Specification and determined by Java functions of class `java.lang.Character`.

```
[:jletter:]      isJavaIdentifierStart()
[:jletterdigit:] isJavaIdentifierPart()
[:letter:]       isLetter()
[:digit:]        isDigit()
[:uppercase:]    isUpperCase()
[:lowercase:]    isLowerCase()
```

  They are especially useful when working with the unicode character set.

If `a` and `b` are regular expressions, then

`a | b` (union) is the regular expression, that matches all input that is matched by `a` or by `b`.

`a b` (concatenation) is the regular expression, that matches the input matched by `a` followed by the input matched by `b`.

`a*` (kleene closure) matches zero or more repetitions of the input matched by `a`

`a+` is equivalent to `aa*`

`a?` matches the empty input or the input matched by `a`

`a{n}` is equivalent to `n` times the concatenation of `a`. So `a{4}` for instance is equivalent to the expression `a a a a`. The decimal integer `n` must be positive.

`a{n,m}` is equivalent to at least `n` times and at most `m` times the concatenation of `a`. So `a{2,4}` for instance is equivalent to the expression `a a a? a?`. Both `n` and `m` are non negative decimal integers and `m` must not be smaller than `n`.

`( a )` matches the same input as `a`.

In a lexical rule, a regular expression `r` may be preceded by a '^' (the beginning of line operator). `r` is then only matched at the beginning of a line in the input. A line begins after each `\r|\n|\r\n` and at the beginning of input. The preceding line terminator in the input is not consumed and can be matched by another rule.

In a lexical rule, a regular expression `r` may be followed by a lookahead expression. A lookahead expression is either a '$' (the end of line operator) or a `'/'` followed by an arbitrary regular expression. In both cases the lookahead is not consumed and not included

int the matched text region, but it *is* considered while determining which rule has the longest match (see also section 4.3.3).

In the '$' case `r` is only matched at the end of a line in the input. The end of a line is denoted by the regular expression `\r|\n|\r\n`. So `a$` is equivalent to `a / \r|\n|\r\n`

For arbitrary lookahead (also called *trailing context*) the expression is matched only when followed by input that matches the trailing context. Unfortunately the lookahead expression is not really arbitrary: In a rule `r1 / r2`, either the text matched by `r1` must have a fixed length (e.g. if `r1` is a string) or the beginning of the trailing context `r2` must not match the end of `r1`. So for example `"abc" / "a"|"b"` is ok because `"abc"` has a fixed length, `"a"|"ab" / "x"*` is ok because no prefix of `"x"*` matches a postfix of `"a"|"ab"`, but `"x"|"xy" / "yx"` is *not* possible, because the postfix `"y"` of `"x"|"xy"` is also a prefix of `"yx"`. JFlex will report such cases at generation time. The algorithm JFlex currently uses for matching trailing context expressions is the one described in [1] (leading to the deficiencies mentioned above).

As of version 1.2, JFlex allows lex/flex style `<<EOF>>` rules in lexical specifications. A rule

```
[StateList]  <<EOF>>    { some action code }
```

is very similar to the `%eofval` directive (section 4.2.3). The difference lies in the optional `StateList` that may precede the `<<EOF>>` rule. The action code will only be executed when the end of file is read and the scanner is currently in one of the lexical states listed in `StateList`. The same `StateGroup` (see section 4.3.3) and precedence rules as in the "normal" rule case apply (i.e. if there is more than one `<<EOF>>` rule for a certain lexical state, the action of the one appearing earlier in the specification will be executed). `<<EOF>>` rules override settings of the `%cup` option and should not be mixed with the `%eofval` directive.

An `Action` consists either of a piece of Java code enclosed in curly braces or is the special `|` action. The `|` action is an abbreviation for the action of the following expression.

Example:

```
expression1   |
expression2   |
expression3   { some action }
```

is equivalent to the expanded form

```
expression1   { some action }
expression2   { some action }
expression3   { some action }
```

They are useful when you work with trailing context expressions. The expression `a | (c / d) | b` is not syntactically legal, but can easily be expressed using the `|` action:

```
a       |
c / d   |
b       { some action }
```

4.3.3 How the input is matched

When consuming its input, the scanner determines the regular expression that matches the longest portion of the input (longest match rule). If there is more than one regular expression that matches the longest portion of input (i.e. they all match the same input), the generated scanner chooses the expression that appeares first in the specification. After determining the active regular expression, the associated action is executed. If there is no matching regular expression, the scanner terminates the program with an error message (if the `%standalone` directive has been used, the scanner prints the unmatched input to `java.lang.System.out` instead and resumes scanning).

Lexical states can be used to further restrict the set of regular expressions that match the current input.

- A regular expression can only be matched when its associated set of lexical states include the currently active lexical state of the scanner or if the set of associated lexical states is empty.
- The currently active lexical state of the scanner can be changed from within an action of a regular expression using the method `yybegin()`.
- The scanner starts in lexical state `YYINITIAL`, which is always declared by default.
- The set of lexical states associated with a regular expression is calculated as follows:

  The set of lexical states of a rule of the form

  ```
  StateList RegExp Action
  ```

  is the union of the set of all states listed in `StateList` and the set of lexical states of the enclosing `StateGroup`, or just the set of all states in `StateList`, if there is no enclosing `StateGroup`.

  The set of lexical states of a state group of the form

  ```
  StateList "{" Rule+ "}"
  ```

  is the union of the set of all states listed in `StateList` and the set of lexical states of the enclosing `StateGroup`, or just the set of all states in StateList, if there is no enclosing `StateGroup`.

  In short: lexical states cumulate over `StateGroups`.
- Lexical states are declared and used as Java `int` constants in the generated class under the same name as they are used in the specification.

4.3.4 Scanner methods and variables accessible in actions

The following methods and member variables of the generated scanner class are meant to be accessed by the user in lexical actions:

- `String yytext()`
  returns the matched input text region

- `int yylength()`
  returns the length of the matched input text region (does not require a `String` object to be created)
- `void yyclose()`
  closes the input stream. All subsequent calls to the scanning method will return the end of file value
- `int yystate()`
  returns the current lexical state of the scanner.
- `void yybegin(int lexicalState)`
  enters the lexical state `lexicalState`
- `void yypushback(int number)`
  pushes `number` characters of the matched text back into the inputstream. They will be read again in the next call of the scanning method. The number of characters to be read again must not be greater than the length of the matched text. The pushed back characters will after the call of `yypushback` not be included in `yylength` and `yytext()`. Please note, that in Java strings are unchangable, i.e. an action code like

  ```
  String matched = yytext();
  yypushback(1);
  return matched;
  ```

  will return the whole matched text, while

  ```
  yypushback(1);
  return yytext();
  ```

  will return the matched text minus the last character.
- `int yyline`
  contains the current line of input (starting with 0, only active whith the `%line` directive)
- `int yychar`
  contains the current character count in the input (starting with 0, only active whith the `%char` directive)
- `int yycolumn`
  contains the current column of the current line (starting with 0, only active whith the `%column` directive)

5 A few words on performance

This section gives some empirical results about the speed of JFlex generated scanners in comparison to those generated by JLex, compares a JFlex scanner with a handwritten one, and presents some tips on how to make your specification produce a faster scanner.

5.1 Comparison of JLex and JFlex

Scanners generated by the tool JLex are quite fast. It was however possible to further improve the performance of generated scanners using JFlex. The following table shows the results that were produced by the scanner specification of a small toy programming language (in fact the example from the JLex website). The scanner was generated using JLex and all three different JFlex code generation methods. Then it was run on a W95 system using JDK 1.2 with different sample inputs of that toy programming language. All test runs were made under the same conditions on an idle machine. The values presented in the table denote the time from the first call to the scanning method to returning the EOF value and the speedup in percent.

	JLex	`%switch`	speedup	`%table`	speedup	`%pack`	speedup
19050 lines, JIT	1.73 s	1.62 s	6.3 %	1.63 s	6.2 %	1.62 s	6.3 %
10010 lines, JIT	0.63 s	0.57 s	9.8 %	0.57 s	9.8 %	0.58 s	7.6 %
4982 lines, JIT	0.3 s	0.27 s	8.0 %	0.27 s	8.0 %	0.27 s	8.0 %
19050 lines, no JIT	8.37 s	7.92 s	5.7 %	7.97 s	5.0 %	7.94 s	5.4 %
10010 lines, no JIT	3.2 s	2.99 s	7.0 %	2.99 s	7.0 %	2.99 s	7.0 %
4982 lines, no JIT	1.58 s	1.47 s	7.5 %	1.47 s	7.5 %	1.47 s	7.5 %

Since the scanning time of the lexical analyzer examined in the table above includes lexical actions that often need to create new object instances, another table shows the execution time for the same specification with empty lexical actions.

	JLex	`%switch`	speedup	`%table`	speedup	`%pack`	speedup
19050 lines, JIT	0.56 s	0.46 s	21.2 %	0.46 s	21.2 %	0.46 s	21.7 %
10010 lines, JIT	0.24 s	0.19 s	30.1 %	0.19 s	30.1 %	0.19 s	30.1 %
4982 lines, JIT	0.12 s	0.1 s	20.0 %	0.1 s	20.0 %	0.11 s	9.1 %
19050 lines, no JIT	5.74 s	4.93 s	16.3 %	4.97 s	15.5 %	4.98 s	15.3 %
10010 lines, no JIT	2.27 s	1.9 s	19.7 %	1.92 s	18.2 %	1.91 s	18.9 %
4982 lines, no JIT	1.14 s	0.96 s	19.5 %	0.96 s	19.5 %	0.96 s	19.5 %

Execution time of single instructions depend on the platform and the implementation of the Java Virtual Machine the program is executed on. Therefore the tables above can not be used as a reference to which code generation method of JFlex is the right one to choose in general. The following table was produced by the same lexical specification and the same input on a Linux system using JDK 1.1.7 with the tya JIT compiler.

With actions:

	JLex	%switch	speedup	%table	speedup	%pack	speedup
19050 lines, JIT	3.32 s	3.12 s	6.2 %	3.11 s	6.5 %	3.11 s	6.6 %
10010 lines, JIT	1.2 s	1.12 s	6.8 %	1.11 s	7.9 %	1.12 s	7.4 %
4982 lines, JIT	0.6 s	0.56 s	7.7 %	0.55 s	8.9 %	0.55 s	8.5 %
19050 lines, no JIT	6.42 s	5.87 s	9.5 %	5.99 s	7.1 %	6.0 s	7.0 %
10010 lines, no JIT	2.45 s	2.19 s	11.5 %	2.23 s	9.5 %	2.23 s	9.7 %
4982 lines, no JIT	1.22 s	1.08 s	12.7 %	1.14 s	6.8 %	1.1 s	10.4 %

Without actions:

	JLex	%switch	speedup	%table	speedup	%pack	speedup
19050 lines, JIT	1.25 s	0.98 s	27.8 %	0.99 s	26.4 %	1.03 s	20.9 %
10010 lines, JIT	0.49 s	0.39 s	25.6 %	0.39 s	24.0 %	0.41 s	20.0 %
4982 lines, JIT	0.25 s	0.2 s	24.9 %	0.21 s	19.4 %	0.2 s	20.6 %
19050 lines, no JIT	3.68 s	2.93 s	25.3 %	2.98 s	23.5 %	2.99 s	22.9 %
10010 lines, no JIT	1.49 s	1.15 s	29.5 %	1.17 s	27.8 %	1.17 s	27.5 %
4982 lines, no JIT	0.75 s	0.57 s	30.5 %	0.58 s	28.9 %	0.58 s	28.7 %

Although all JFlex scanners were faster than those generated by JLex, slight differences between JFlex code generation methods show up when compared to the run on the W95 system.

The following table compares a handwritten scanner for the Java language obtained from the website of CUP with the JFlex generated scanner for Java that comes with JFlex in the `examples` directory. They were tested on different `.java` files on a Linux JDK 1.1.7 and the tya JIT compiler.

	handwritten scanner	JFlex generated scanner	
6350 lines, JIT	3.55 s	1.27 s	179 % faster
492 lines, JIT	0.23 s	86 ms	167 % faster
113 lines, JIT	134 ms	37 ms	262 % faster
6350 lines, no JIT	6.28 s	3.04 s	106 % faster
492 lines, no JIT	0.41 s	209 ms	96 % faster
113 lines, no JIT	201 ms	81 ms	148 % faster

As you can see, the generated scanner is up to 2.5 times faster than the handwritten one. One example of a handwritten scanner that is considerably slower than the equivalent generated one is surely no proof for all generated scanners being faster than handwritten. It is clearly impossible to prove something like that, since you could always write the generated scanner by hand. From a software engineering point of view however, there is no excuse for writing a scanner by hand since this task takes more time, is more difficult and therefore more error prone than writing a compact, readable and easy to change lexical specification. (I like to add, that I do *not* think, that the handwritten scanner from the CUP website used here in the test is stupid or badly written or anything like that. I actually think, Scott did a great job with it, and that for learning about lexers it is quite valuable to study it or even to write a similar one for oneself.)

5.2 How to write a faster specification

Although JFlex generated scanners show good performance without special optimizations, there are some heuristics that can make a lexical specification produce an even faster scanner. Those are (roughly in order of performance gain):

- Avoid rules that require backtracking

  From the C/C++ flex [5] manpage: *"Getting rid of backtracking is messy and often may be an enormous amount of work for a complicated scanner."* Backtracking is introduced by the longest match rule and occurs for instance on this set of expressions:

  ```
  "averylongkeyword"
  .
  ```

  With input `"averylongjoke"` the scanner has to read all charcters up to `'j'` to decide that rule `.` should be matched. All characters of `"verylong"` have to be read again for the next matching process. Backtracking can be avoided in general by adding error rules that match those error conditions

  ```
  "av"|"ave"|"avery"|"averyl"|..
  ```

  While this is impractical in most scanners, there is still the possiblity to add a "catch all" rule for a lengthy list of keywords

  ```
  "keyword1"  { return symbol(KEYWORD1); }
  ..
  "keywordn"  { return symbol(KEYWORDn); }
  [a-z]+      { error("not a keyword"); }
  ```

  Most programming language scanners already have a rule like this for some kind of variable length identifiers.

- Avoid line and column counting

  It costs one additional comparison per input character and the matched text has to be rescanned for counting. In most scanners it is possible to do the line counting in the specification by incrementing `yyline` each time a line terminator has been matched. Column counting could also be included in actions. This will be faster, but can in some cases become quite messy.

- Avoid lookahead expressions and the end of line operator '$'

  The trailing context will first have to be read and then (because it is not to be consumed) read again.

- Avoid the beginning of line operator '^'

  It costs two additional comparisons per match. In some cases one extra lookahead character is needed (when the last character read is `\r` the scanner has to read one character ahead to check if the next one is an `\n` or not).

- Match as much text as possible in a rule.

  One rule is matched in the innermost loop of the scanner. After each action some overhead for setting up the internal state of the scanner is necessary.

Note, that writing more rules in a specification does not make the generated scanner slower (except when you have to switch to another code generation method because of the larger size).

The two main rules of optimization apply also for lexical specifications:

1. **don't do it**
2. **(for experts only) don't do it yet**

Some of the performance tips above contradict a readable and compact specification style. When in doubt or when requirements are not or not yet fixed: don't use them - the specification can always be optimized in a later state of the development process.

6 Porting Issues

6.1 Porting from JLex

JFlex was designed to read old JLex specifications unchanged and to generate a scanner which behaves exactly the same as the one generated by JLex with the only difference of being faster.

This works as expected on all well formed JLex specifications.

Since the statement above is somewhat absolute, let's take a look at what "well formed" means here. A JLex specification is well formed, when it

- generates a working scanner with JLex
- doesn't contain the '^' operator

  The beginning of line operator has a non standard behavior in JLex: It consumes a preceding `\n` character. This does not happen in JFlex generated scanners. The problem can be worked around by writing an extra rule that matches the "new" `\n` characters.
- doesn't contain the '$' operator

  The end of line operator is not platform independent in JLex. It matches a `\n` by default or a `\r\n` when the `%notunix` directive is given. JFlex matches the expression `\r|\n|\r\n` for the end of line and ignores the `%notunix` directive. This should usually cause no problems with your old specification other than that it accepts other platforms too.
- doesn't contain the `%cup` switch

  The `%cup` switch has currently no meaning in JLex (this refers to JLex version 1.2.4, it is anticipated that future versions of JLex will support the `%cup` switch with a similar meaning as now in JFlex). In JFlex it is used to interface with CUP.
- has only complete regular expressions surrounded by parentheses in macro definitions

  This may sound a bit harsh, but could otherwise be a major problem -- it can also help you find some disgusting bugs in your specification that didn't show up in the first

place. In JLex, a right hand side of a macro is just a piece of text, that is copied to the point where the macro is used. With this, some weird kind of stuff like

```
macro1 = ("hello"
macro2 = {macro1})*
```

was possible (with `macro2` expanding to `("hello")*`). This is not allowed in JFlex and you will have to transform such definitions. There are however some more subtle kinds of errors that can be introduced by JLex macros. Let's consider a definition like `macro = a|b` and a usage like `{macro}*`. This expands in JLex to `a|b*` and not to the probably intended `(a|b)*`.

JFlex uses always the second form of expansion, since this is the natural form of thinking about abbreviations for regular expressions.

Most specifications shouldn't suffer from this problem, because macros often only contain (harmless) character classes like `alpha = [a-zA-Z]` and more dangerous definitions like

```
ident = {alpha}({alpha}|{digit})*
```

are only used to write rules like

```
{ident}       { .. action .. }
```

and not more complex expressions like

```
{ident}*      { .. action .. }
```

where the kind of error presented above would show up.

6.2 Porting from lex/flex

This section tries to give an overview of activities and possible problems when porting a lexical specification from the C/C++ tools lex and flex [5] available on most Unix systems to JFlex.

Most of the C/C++ specific features are naturally not present in JFlex, but most "clean" lex/flex lexical specifications can be ported to JFlex without very much work.

This section is by far not complete and is based mainly on a survey of the flex man page and very little personal experience. If you do engage in any porting activity from lex/flex to JFlex and encounter problems, have better solutions for points presented here or have just some tips you would like to share, please do contact me via email: `Gerwin Klein <lsf@jflex.de>`. I will incorporate your experiences in this manual (with all due credit to you, of course).

6.2.1 Basic structure

A lexical specification for flex has the following basic structure:

```
definitions
%%
rules
%%
user code
```

The `user code` section usually contains some C code that is used in actions of the `rules` part of the specification. For JFlex most of this code will have to be included in the class code `%{..%}` directive in the `options and declarations` section (after translating the C code to Java, of course).

6.2.2 Macros and Regular Expression Syntax

The `definitions` section of a flex specification is quite similair to the `options and declarations` part of JFlex specs.

Macro definitions in flex have the form:

```
<identifier>  <expression>
```

To port them to JFlex macros, just insert a = between `<identifier>` and `<expression>`.

The syntax and semantics of regular expressions in flex are pretty much the same as in JFlex. A little attention is needed for some escape sequences present in flex (such as `\a`) that are not supported in JFlex. These escape sequences should be transformed into their octal or hexdecimal equivalent.

Another point are predefined character classes. Flex offers the ones directly supported by C, JFlex offers the ones supported by Java. These classes will sometimes have to be listed manually (if there is need for this feature, it may be implemented in a future JFlex version).

6.2.3 Lexical States

Flex supports two kinds of lexical states or start conditions: inclusive states declared with `%s` and exclusive states declared using `%x`.

JFlex only supports inclusive lexical states (for which the `%s` just has to be replaced by `%state`).

6.2.4 Lexical Rules

Since flex ist mostly Unix based, the '^' (beginning of line) and '$' (end of line) operators, consider the `\n` character as only line terminator. This should usually cause not much problems, but you should be prepared for `\r` or `\r\n` occurences that are now considered as line terminators and therefore may not be consumed when ^ or $ is present in a rule.

The trailing context algorithm of flex is better than the one used in JFlex. Therefore lookahead expressions could cause major headaches. JFlex will issue an error message at generation time, if it cannot generate a scanner for a certain lookahead expression. (sorry, I have no

more tips here on that yet. If anyone knows how the flex lookahead algorithm works (or any better one) and can be efficiently implemented, again: please contact me).

7 Working together: JFlex and CUP

One of the main design goals of JFlex was to make interfacing with the free Java parser generator CUP [8] as easy as possibly. This has been done by giving the `%cup` directive a special meaning. An interface however always has two sides. This section concentrates on the CUP side of the story.

7.1 CUP version 0.10j

Since CUP version 0.10j, this has been simplified greatly by the new CUP scanner interface `java_cup.runtime.Scanner`. JFlex lexers now implement this interface automatically when then `%cup` switch is used. There are no special `parser code`, `init code` or `scan with` options any more that you have to provide in your CUP parser specification. You can just concetrate on your grammar.

If your generated Lexer has the class name `Scanner`, the parser is started from the a main program like this:

```
...
  try {
    parser p = new parser(new Scanner(new FileReader(fileName)));
    Object result = p.parse().value;
  }
  catch (Exception e) {
...
```

7.2 Using existing JFlex/CUP specifications with CUP 0.10j

If you already have an existing specification and you would like to upgrade both JFlex and CUP to their newest version, you will probably have to adjust your specification.

The main difference between the `%cup` switch in JFlex 1.2.1 and lower, and the current JFlex version is, that JFlex scanners now automatically implement the `java_cup.runtime.Scanner` interface. This means, that the scanning function now changes its name from `yylex()` to `next_token()`.

The main difference from older CUP versions to 0.10j is, that CUP now has a default constructor that accepts a `java_cup.runtime.Scanner` as argument and that uses this scanner as default (so no `scan with` code is necessary any more).

If you have an existing CUP specification, it will probably look somewhat like this:

```
parser code {:
  Lexer lexer;
```

```
  public parser (java.io.Reader input) {
    lexer = new Lexer(input);
  }
:};

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

To upgrade to CUP 0.10j, you could change it to look like this:

```
parser code {:
  public parser (java.io.Reader input) {
    super(new Lexer(input));
  }
:};
```

If you do not mind to change the method that is calling the parser, you could remove the constructor entirely (and if there is nothing else in it, the whole `parser code` section as well, of course). The calling main procedure would then construct the parser as shown in the section above.

The JFlex specification does not need to be changed.

7.3 Using older versions of CUP

For people, who like or have to use older versions of CUP, the following section explains "the old way". Please note, that the standard name of the scanning function with the `%cup` switch is not `yylex()`, but `next_token()`.

If you have a scanner specification that begins like this:

```
package PACKAGE;
import java_cup.runtime.*;   /* this is convenience, but not necessary */

%%

%class Lexer
%cup
..
```

then it matches a CUP specification starting like

```
package PACKAGE;

parser code {:
  Lexer lexer;

  public parser (java.io.Reader input) {
    lexer = new Lexer(input);
  }
:};
```

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

..
```

This assumes that the generated parser will get the name `parser`. If it doesn't, you have to adjust the constructor name.

The parser can then be started in a main routine like this:

```
..
  try {
    parser p = new parser(new FileReader(fileName));
    Object result = p.parse().value;
  }
  catch (Exception e) {
..
```

If you want the parser specification to be independent of the name of the generated scanner, you can instead write an interface Lexer

```
public interface Lexer {
  public java_cup.runtime.Symbol next_token() throws java.io.IOException;
}
```

change the parser code to:

```
package PACKAGE;

parser code {:
  Lexer lexer;

  public parser (Lexer lexer) {
    this.lexer = lexer;
  }
:};

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

..
```

tell JFlex about the Lexer interface using the `%implements` directive:

```
..
%class Scanner     /* not Lexer now since that is our interface! */
%implements Lexer
%cup
..
```

and finally change the main routine to look like

```
...
  try {
    parser p = new parser(new Scanner(new FileReader(fileName)));
    Object result = p.parse().value;
  }
  catch (Exception e) {
...
```

If you want to improve the error messages that CUP generated parsers produce, you can also override the methods `report_error` and `report_fatal_error` in the "parser code" section of the CUP specification. The new methods could for instance use `yyline` and `yycolumn` (stored in the `left` and `right` members of class `java_cup.runtime.Symbol`) to report error positions more conveniently for the user. The lexer and parser for the Java language in the `examples\java` directory of this JFlex distribution use this style of error reporting. These specifications also demonstrate the techniques above in action.

8 Bugs and Deficiencies

8.1 Deficiencies

The trailing context algorithm described in [1] and used in JFlex is incorrect. It does not work, when a postfix of the regular expression matches a prefix of the trailing context and the length of the text matched by the expression does not have a fixed size. JFlex will report these cases as errors at generation time.

8.2 Bugs

As of August 23, 1999, no bugs have been reported for JFlex version 1.2.2. All bugs reported for earlier versions have been fixed.

If you find new ones, please report them by email to Gerwin Klein `<lsf@jflex.de>`.

Please check the FAQ and currently known bugs at the JFlex website[3] before reporting a new bug.

9 Copying and License

JFlex is free software, published under the terms of the GNU General Public License[4].

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

The code generated by JFlex inherits the copyright of the specification it was produced from. If it was your specification, you may use the generated code without restriction.

See the file `COPYRIGHT` for more information.

[3] http://www.jflex.de/
[4] http://www.fsf.org/copyleft/gpl.html

References

[1] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques, and Tools*, 1986

[2] A.W. Appel, *Modern Compiler Implementation in Java: basic techniques*, 1997

[3] Elliot Berk, *JLex: A lexical analyser generator for Java*, `http://www.cs.princeton.edu/~appel/modern/java/JLex/`

[4] K. Brouwer, W. Gellerich,E. Ploedereder, *Myths and Facts about the Efficient Implementation of Finite Automata and Lexical Analysis*, in: Proceedings of the 7th International Conference on Compiler Construction (CC '98), 1998

[5] Vern Paxon, *flex - The fast lexical analyzer generator*, 1995

[6] P. Dencker, K. Durre, J. Henft, *Optimization of Parser Tables for portable Compilers*, in: ACM Transactions on Programming Languages and Systems 6(4), 1984

[7] J. Gosling, B. Joy, G. Steele, *The Java Language Specifcation*, 1996, `http://www.javasoft.com/docs/books/jls/`

[8] Scott E. Hudson, *CUP LALR Parser Generator for Java*, `http://www.cs.princeton.edu/~appel/modern/java/CUP/`

[9] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, 1996, `http://www.javasoft.com/docs/books/vmspec/`

[10] R.E. Tarjan, A. Yao, *Storing a Sparse Table*, in: Communications of the ACM 22(11), 1979

[11] R. Wilhelm, D. Maurer, *Ubersetzerbau*, Berlin 1997^2