The Fast Lexical Analyser Generator

JFlex User's Manual

Version 1.8.1, February 28, 2020

Contents

1 Introduction

JFlex is a lexical analyser generator for Java written in Java. It is also a rewrite of the tool JLex [3] which was developed by Elliot Berk at Princeton University. As Vern Paxson states for his C/C++ tool flex [8]: they do not share any code though.

A lexical analyser generator takes as input a specification with a set of regular expressions and corresponding actions. It generates a program (a *lexer*) that reads input, matches the input against the regular expressions in the spec file, and runs the corresponding action if a regular expression matched. Lexers usually are the first front-end step in compilers, matching keywords, comments, operators, etc, and generating an input token stream for parsers. They can also be used for many other purposes.

1.1 Design goals

The main design goals of JFlex are:

- **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 topic of lexical analysis in parsing. The references [1] and [2] provide a good introduction.

The next section of this manual describes installation procedures for JFlex. Working with JFlex - an example runs through an example specification and explains how it works. The section on Lexical specifications presents all JFlex options and the complete specification syntax; Encodings, Platforms, and Unicode provides information about Unicode and scanning text vs. binary files. A few words on performance gives tips on how to write fast scanners. The section on porting scanners shows how to port scanners from JLex, and from the `lex` and `flex` tools for C. Finally, working together discusses interfacing JFlex scanners with the LALR parser generators CUP, CUP2, BYacc/J, Jay.

2 Installing and Running JFlex

2.1 Installing JFlex

2.1.1 Windows

To install JFlex on Windows, follow these three steps:

1. Unzip the file you downloaded into the directory you want JFlex in. If you unzipped it to say `C:\`, the following directory structure should be generated:

```
C:\jflex-1.8.1\
    +--bin\                       (start scripts)
    +--doc\                       (FAQ and manual)
    +--examples\
        +--byaccj\                (calculator example for BYacc/J)
        +--cup-maven\             (calculator example for cup and maven)
        +--interpreter\           (interpreter example for cup)
        +--java\                  (Java lexer specification)
        +--simple\                (example scanner with no parser)
        +--standalone-maven\      (a simple standalone scanner,
                                   built with maven)
        +--zero-reader\           (Readers that return 0 characters)
    +--lib\                       (precompiled classes)
    +--src\
        +--main\
            +--config\            (PMD source analyzer configuration)
            +--cup\               (JFlex parser spec)
            +--java\
                +--jflex\         (source code of JFlex)
                    +--anttask\   (source code of JFlex Ant Task)
                    +--gui\       (source code of JFlex UI classes)
                    +--unicode\   (source code for Unicode properties)
            +--jflex\             (JFlex scanner spec)
            +--resources\         (messages and default skeleton file)
        +--test\                  (unit tests)
```

2. Edit the file **`bin\jflex.bat`** (in the example it's `C:\jflex-1.8.1\bin\jflex.bat`) such that
   - **`JAVA_HOME`** contains the directory where your Java JDK is installed (for instance `C:\java`) and
   - **`JFLEX_HOME`** the directory that contains JFlex (in the example: `C:\jflex-1.8.1`)
3. Include the `bin\` directory of JFlex in your path. (the one that contains the start script, in the example: `C:\jflex-1.8.1\bin`).

2.1.2 Mac/Unix with tar

To install JFlex on a Mac or Unix system, follow these two steps:

- Decompress the archive into a directory of your choice with GNU tar, for instance to `/usr/share`:

  ```
  tar -C /usr/share -xvzf jflex-1.8.1.tar.gz
  ```

  (The example is for site wide installation. You need to be root for that. User installation works exactly the same way — just choose a directory where you have write permission)

- Make a symbolic link from somewhere in your binary path to `bin/jflex`, for instance:

  `ln -s /usr/share/jflex-1.8.1/bin/jflex /usr/bin/jflex`

  If the Java interpreter is not in your binary path, you need to supply its location in the script `bin/jflex`.

You can verify the integrity of the downloaded file with the SHA1 checksum available on the JFlex download page. If you put the checksum file in the same directory as the archive, and run:

`shasum --check jflex-1.8.1.tar.gz.sha1`

it should tell you

`jflex-1.8.1.tar.gz: OK`

2.2 Running JFlex

You run JFlex with:

`jflex <options> <inputfiles>`

It is also possible to skip the start script in `bin/` and include the file `lib/jflex-1.8.1.jar` in your `CLASSPATH` environment variable instead.

Then you run JFlex with:

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

or with:

`java -jar jflex-1.8.1.jar <options> <inputfiles>`

The input files and options are in both cases optional. If you don't provide a file name on the command line, 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>`

`--encoding <name>`
uses the character encoding `<name>` (e.g. `utf-8`) to read lexer specifications and write java files.

`--skel <file>`
uses external skeleton `<file>` in UTF-8 encoding. This is mainly for JFlex maintenance and special low level customisations. Use only when you know what you are doing! JFlex comes with a skeleton file in the `src` directory that reflects exactly the internal, pre-compiled skeleton and can be used with the `-skel` option.

`--nomin`
skip the DFA minimisation step during scanner generation.

`--jlex`
tries even harder to comply to JLex interpretation of specs.

`--dot`
generate graphviz dot files for the NFA, DFA and minimised DFA. This feature is still in alpha status, and not fully implemented yet.

`--dump`
display transition tables of NFA, initial DFA, and minimised DFA

`--legacydot`
dot (`.`) meta character matches `[^\n]` instead of
`[^\n\r\u000B\u000C\u0085\u2028\u2029]`

`--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)

`--warn-unused`
warn about unused macros (by default true in verbose mode and false in quiet mode)

`--no-warn-unused`
do not warn about unused macros (by default true in verbose mode and false in quiet mode)

`--time`
display time statistics about the code generation process (not very accurate)

`--version`
print version number

`--info`
print system and JDK information (useful if you'd like to report a problem)

`--unicodever <ver>`
print all supported properties for Unicode version `<ver>`

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

2.3 Maven plugin

The plugin reads JFlex grammar specification files (`.jflex`) and generates a corresponding Java parser (in `target/generated-source/jflex` by default).

2.3.1 Usage

Minimal configuration This configuration generates java code of a parser for all grammar files (`*.jflex`, `*.jlex`, `*.lex`, `*.flex`) found in `src/main/jflex/` and its sub-directories.

The name and package of the generated Java source code are the ones defined in the grammar. The generated Java source code is placed in `target/generated-source/jflex`, in sub-directories following the Java convention on package names.

Update the `pom.xml` to add the plugin:

```
<project>
  <!-- ... -->
  <build>
    <plugins>
```

```
      <plugin>
        <groupId>de.jflex</groupId>
        <artifactId>jflex-maven-plugin</artifactId>
        <version>1.8.1</version>
        <executions>
          <execution>
            <goals>
              <goal>generate</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
    <!-- ... -->
  </build>
  <!-- ... -->
</project>
```

More complex configuration This example generates the source for the two grammars `src/main/lex/preprocessor.jflex` and `/pub/postprocessor.jflex`, as well as all grammar files found in `src/main/jflex` (and its sub-directories). The generated Java code is placed into `src/main/java` instead of `target/generated-sources/jflex`.

```
      <plugin>
        <groupId>de.jflex</groupId>
        <artifactId>jflex-maven-plugin</artifactId>
        <version>1.8.1</version>
        <executions>
          <execution>
            <goals>
              <goal>generate</goal>
            </goals>
            <configuration>
              <outputDirectory>src/main/java</outputDirectory>
              <lexDefinitions>
                <lexDefinition>src/main/jflex</lexDefinition>
                <lexDefinition>src/main/lex/preprocessor.jflex</lexDefinition>
                <lexDefinition>/pub/postprocessor.jflex</lexDefinition>
              </lexDefinitions>
            </configuration>
          </execution>
        </executions>
      </plugin>
```

Even more complex configuration, using several executions This generates the source for

- all files found in `src/main/lex/`, using strict JLex compatibility.
- and all files found in `src/main/jflex`, in verbose mode.

```
      <plugin>
```

```
<groupId>de.jflex</groupId>
<artifactId>jflex-maven-plugin</artifactId>
<version>1.8.1</version>
<executions>
  <execution>
    <id>strict jlex</id>
    <goals>
      <goal>generate</goal>
    </goals>
    <configuration>
      <lexDefinitions>
        <lexDefinition>src/main/lex</lexDefinition>
      </lexDefinitions>
      <jlex>true</jlex>
    </configuration>
  </execution>
  <execution>
    <id>jflex</id>
    <goals>
      <goal>generate</goal>
    </goals>
    <configuration>
      <lexDefinitions>
        <lexDefinition>src/main/jflex</lexDefinition>
      </lexDefinitions>
      <verbose>true</verbose>
    </configuration>
  </execution>
</executions>
</plugin>
```

2.3.2 More information

- jflex:generate for more information about the configuration options of the jflex-maven-plugin.
- POM reference guide on plugins for more information about using plugins in a project.

2.4 JFlex Ant Task

JFlex can easily be integrated with the Ant build tool. To use JFlex with Ant, simply copy the `lib/jflex-1.8.1.jar` file to the `$ANT_HOME/lib/` directory or explicitly set the path to `lib/jflex-1.8.1.jar` in the task definition (see example below).

The JFlex Ant Task invokes JFlex on a grammar file.

To use the JFlex task, place the following line in the Ant build file:

```
<taskdef classname="jflex.anttask.JFlexTask" name="jflex" />
```

Or, setting the path to the JFlex jar explicitly:

```
<taskdef classname="jflex.anttask.JFlexTask" name="jflex"
         classpath="path-to-jflex.jar" />
```

The JFlex task requires the `file` attribute to be set to the source grammar file (`*.flex`). Unless the target directory is specified with the `destdir` option, the generated class will be saved to the same directory where the grammar file resides. Like `javac`, the JFlex task creates subdirectories in `destdir` according to the generated class package.

This task only invokes JFlex if the grammar file is newer than the generated files.

2.4.1 Parameters

The following attributes are available for invoking the JFlex task.

- `file="file"`
  The grammar file to process. This attribute is required.
- `destdir="dir"`
  The directory to write the generated files to. If not set, the files are written to the directory containing the grammar file. Note that unlike JFlex's `-d` command line option, `destdir` causes the generated file to be written to `{destdir}/{packagename}`. This behaviour is similar to 'javac -d dir.
- `outdir="dir"`
  The directory to write the generated files to. If not set, the files are written to the directory containing the grammar file. This options works exactly like JFlex's `-d` command line option, it causes the output file to be written to `dir` regardless of the package name.
- `verbose` (default `"off"`)
  Display generation process messages.
- `encoding` (if unset uses the JVM default encoding)
  The character encoding to use when reading lexer specifications and writing java files.
- `dump` (default `"off"`)
  Dump character classes, NFA and DFA tables.
- `time` or `timeStatistics` (default `"off"`)
  Display generation time statistics.
- `nomin` or `skipMinimization` (default `"off"`)
  Skip DFA minimisation step.
- `skel="file"` or `skeleton="file"`
  Use external skeleton file in UTF-8 encoding.
- `dot` or `generateDot` (default `"off"`)
  Write graphviz `.dot` files for the generated automata.
- `nobak` (default `"off"`)
  Do not make a backup if the generated file exists.
- `jlex` (default `"off"`)
  Use JLex compatibility mode.

- `legacydot` (default `"off"`)
  The dot `.` meta-character matches `[^\n]` instead of `[^\n\r\u000B\u000C\u0085\u2028\u2029]`
- `unusedwarning` (default `"true"`)
  Warn about unused macro definitions in the lexer specification.

2.4.2 Example

After the task definition, the `<jflex ..>` task is available in Ant. For example:

```
<jflex
    file="src/parser/Parser.flex"
    destdir="build/generated/"
/>
```

JFlex generates the scanner for `src/parser/Scanner.flex` and saves the result to `build/generated/parser/`, providing `Scanner.flex` is declared to be in package `parser`.

```
<jflex
    file="src/parser/Scanner.flex"
    destdir="build/generated/"
/>
<javac
    srcdir="build/generated/"
    destdir="build/classes/"
/>
```

The same as above plus compile generated classes to `build/classes`

3 A simple Example: How to work with JFlex

To demonstrate how 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 [6] and therefore uses a class `sym` (generated by CUP), where integer constants for the terminal tokens of the CUP grammar are declared.

You can find this example in `examples/cup-java-simplified`.

The `examples/cup-java` directory also contains a *complete* JFlex specification of the lexical structure of Java programs together with the CUP parser specification for Java by C. Scott Ananian, obtained from the CUP [6] web site (modified to interface with the JFlex scanner). Both specifications adhere to the Java Language Specification [5].

In `examples/standalone`, you can find a small standalone scanner that doesn't need other dependencies or tools like CUP to give you working code.

```
/* JFlex example: partial Java language lexer specification */
import java_cup.runtime.*;

/**
 * This class is a simple example lexer.
 */
%%

%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   = "/*" [^*] ~"*/" | "/*" "*"+ "/"
// Comment can be the last line of the file, without line terminator.
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.STRING_LITERAL,
                                  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 */
[^]                             { throw new Error("Illegal character <"+
                                                  yytext()+">"); }
```

From this specification JFlex generates a `.java` file with one class that contains code for the scanner. The class will have a constructor taking a `java.io.Reader` from which the input is read. The class will also have a function `yylex()` that runs the scanner and that can be used to get the next token from the input (in this example the function actually has the name `next_token()` because the specification uses the `%cup` switch).

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).

Next to `package` and `import` statements there is usually not much to do here. If the code ends with a `javadoc` class comment, the generated class will get this comment, if not, JFlex will generate one automatically.

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. For scanning text files, `%unicode` should always be used. The Unicode version may be specified, e.g. `%unicode 4.1`. If no version is specified, the most recent supported Unicode version will be used - in JFlex 1.8.1, this is Unicode 12.1. See also Encodings for more information on character sets, encodings, and scanning text vs. binary files.
- `%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 (the current column is accessed via `yycolumn`)

The code between `%{` and `%}` 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 also JFlex and CUP for how to interface with the parser generator CUP). As with all 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. 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 a `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 does not contain, but ends in `*/`. As this would not match comments like `/****/`, we add `/*` followed by an arbitrary number (at least one) of `*` followed by the closing `/`. This is not the only, but one of the

simpler expressions matching non-nesting Java comments. It is tempting to just write something like the expression `/* .* */`, but this would match more than we want. It would for instance match the entire input `/* */ x = 0; /* */`, instead of two comments and four real tokens. See the macros `DocumentationComment` and `CommentContent` for an alternative.

- `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 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`.

In addition 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 there often are sets of expressions with the same start conditions, they can be grouped:

```
<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 immediately 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 is of special interest:

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

If the scanner matches a double quote in state `YYINITIAL` we have recognised 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 Java string literal). The matched region of the input is referred to by `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 precedence over any other rule).

3.4 How to get it building

- Install JFlex
- 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 show 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 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 user code that is copied verbatim to the beginning of the generated source file before the scanner class declaration. As shown in the example spec, this is the place to put `package` declarations and `import` statements. It is possible, but not considered good Java style to put helper classes, such as token classes, into this section; they are usually better declared in their own `.java` files.

4.2 Options and declarations

The second part of the lexical specification contains options and directives to customise the generated lexer, declarations of lexical states and macro definitions.

Each JFlex directive must sit 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 also the example specification).

4.2.1 Class options and user class code

These options regard name, constructor, API, 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 lexer class implement the specified interfaces. If more than one `%implements` directive is present, all 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.

- `%apiprivate`

  Makes all generated methods and fields of the class private. Exceptions are the constructor, user code in the specification, and, if `%cup` is present, the method `next_token`. All occurrences of `public` (one space character before and after `public`) in the skeleton file are replaced by `private` (even if a user-specified skeleton is used). Access to the generated class is expected to be mediated by user class code (see next switch).

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

  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. Like 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 initialised. If more than one initialiser 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.

- `%ctorarg "type" "ident"`

  Adds the specified argument to the constructors of the generated scanner. If more than one such directive is present, the arguments are added in order of occurrence in the specification. Note that this option conflicts with the `%standalone` and `%debug` directives, because there is no sensible default that can be created automatically for such parameters in the generated `main` methods. JFlex will warn in this case and generate an additional default constructor without these parameters and without user init code (which might potentially refer to the parameters).

- `%scanerror "exception"`

  Causes the generated scanner to throw an instance of the specified exception in case of an internal error (default is `java.lang.Error`). Note that this exception is only for internal scanner errors. With usual specifications it should never occur (i.e. if there is an error fallback rule in the specification and only the documented scanner API is used).

- `%buffer "size"`

  Set the initial size of the scan buffer to the specified value (decimal, in bytes). The default value is 16384.

- `%include "filename"`

  Replaces the `%include` verbatim by the specified file.

4.2.2 Scanning method

This section shows how the scanning method can be customised. 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. 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 implementation 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 returning 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 value 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`,
- for no specified type at all, or
- for 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* the end of file is reached (more than once 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 and `%byaccj` switch. There is also an alternative, more readable way to specify the end of file value using the `<<EOF>>` rule.

- `%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:

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

  The exceptions listed inside `%eofthrow{...%eofthrow}` will be declared in the throws clause of the method `yy_do_eof()`. 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}`)

- `%eofclose false`

  Turns the effect of `%eofclose` off again (e.g. in case closing of input stream is not wanted after `%cup`).

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 information about each returned token to the Java console until the end of file is reached. The information includes: line number (if line counting is enabled), column (if column counting is enabled), the matched text, and the executed action (with line number in the specification).

- `%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.

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 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 the CUP section if you are interested in how to interface your generated scanner with CUP.

- `%cup`

  The `%cup` directive enables 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(<CUPSYM>.EOF);
  %eofval}
  %eofclose
  ```

  The value of `<CUPSYM>` defaults to `sym` and can be changed with the `%cupsym` directive. In JLex compatibility mode (`--jlex` switch on the command line), `%eofclose` will not be turned on.

- `%cup2`

  The `%cup2` directive is similar to CUP mode, just for the CUP2 generator from TU Munich at `http://www2.in.tum.de/cup2`. It does the following:

  - adds CUP2 package import declarations
  - implements the CUP2 scanner interface
  - switches on line and column count
  - sets the scanner function to `readNextTerminal`
  - sets the token type to `ScannerToken<? extends Object>`
  - returns the special CUP2 EOF token at end of file
  - switches on unicode

- `%cupsym "classname"`

  Customises the name of the CUP generated class/interface containing the names of terminal tokens. Default is `sym`. The directive should not be used after `%cup`, only before.

- `%cupdebug`

  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. Prints line, column, matched text, and CUP symbol name for each returned token to standard out.

4.2.6 BYacc/J compatibility

You may also want to read JFlex and BYacc/J if you are interested in how to interface your generated scanner with Byacc/J.

- `%byaccj`

  The `%byaccj` directive enables BYacc/J compatibility mode and is equivalent to the following set of directives:

```
%integer
%eofval{
  return 0;
%eofval}
%eofclose
```

4.2.7 Input Character sets

- `%7bit`

  Causes the generated scanner to use an 7 bit input character set (character codes 0-127). 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 the `%unicode` directive. See also Encodings for information about character encodings. This is the default in JLex compatibility mode.

- `%full`
  `%8bit`

  Both options cause the generated scanner to use an 8 bit input character set (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`. Note that even if your platform uses only one byte per character, the Unicode value of a character may still be greater than 255. If you are scanning text files, you should consider using the `%unicode` directive. See also section Econdings for more information about character encodings.

- `%unicode`
  `%16bit`

  Both options cause the generated scanner to use the full Unicode input character set, including supplementary code points: 0-0x10FFFF. `%unicode` does not mean that the scanner will read two bytes at a time. What is read and what constitutes a character depends on the runtime platform. See also section Encodings for more information about character encodings. This is the default unless the JLex compatibility mode is used (command line option `--jlex`).

- `%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 specification is for instance handled like the expression `[bB][rR][eE][aA][kK]`. The `%caseless` option does not change the matched text and does not affect character classes. So `[a]` still only matches the character `a` and not `A`. Which letters are uppercase and which lowercase letters, is defined by the Unicode standard. In JLex compatibility mode (`--jlex` switch on the command line), `%caseless` and `%ignorecase` also affect character classes.

4.2.8 Line, character and column counting

- `%char`

  Turns character counting on. The `long` 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 recognised as valid directive. It used to switch between Windows and Unix kind of line terminators (`\r\n` and `\n`) for the `$` operator in regular expressions. JFlex always recognises both styles of platform dependent line terminators.

- `%yyeof`

  This JLex option is obsolete in JFlex but still recognised 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 form:

`%s[tate] "state identifier" [, "state identifier", ... ]` for inclusive or
`%x[state] "state identifier" [, "state identifier", ... ]` for exclusive states

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

The sequence

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

declares the set of identifiers `STATE1, STATE3, XYZ, STATE_10, ABC, STATE5` as lexical states, `STATE1`, `ABC`, `STATE5` as inclusive, and `STATE3`, `XYZ`, `STATE_10` as exclusive. See also How the Input is Matched on the way lexical states influence how the input is matched.

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 white-space, followed by an `=`, followed by optional white-space, followed by a regular expression (see Lexical Rules for more information about the regular expression syntax).

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 Porting from JLex 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. However, macros remain just abbreviations of the regular expressions they represent. They are not non-terminals of a grammar and cannot be used recursively. 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 a JFlex specification contains a set of regular expressions and actions (Java code) that are executed when the scanner matches the associated regular expression.

The `%include` directive may be used in this section to include lexical rules from a separate file. The directive will be replaced verbatim by the contents of the specified file.

4.3.1 Syntax

The syntax of the *lexical rules* section is described by the following EBNF grammar (terminal symbols are enclosed in 'quotes'):

```
LexicalRules ::= (Include|Rule)+
Include      ::= '%include' (' '|'\t'|'\b')+ File
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 ('*'|'+'|'?')
               | RegExp "{" Number ["," Number] "}"
               | CharClass
               | PredefinedClass
               | MacroUsage
               | '"' StringCharacter+ '"'
               | Character

CharClass    ::= '[' ['^'] CharClassContent* ']'
               | '[' ['^'] CharClassContent+
                     CharClassOperator CharClassContent+ ']'

CharClassContent    ::= CharClass | Character |
                        Character'-'Character |
                        MacroUsage | PredefinedClass

CharClassOperator   ::= '||' | '&&' | '--' | '~~'

MacroUsage          ::= '{' Identifier '}'

PredefinedClass     ::= '[:jletter:]'
                      | '[:jletterdigit:]'
                      | '[:letter:]'
                      | '[:digit:]'
                      | '[:uppercase:]'
                      | '[:lowercase:]'
                      | '\d' | '\D'
                      | '\s' | '\S'
                      | '\w' | '\W'
                      | '\p{' UnicodePropertySpec '}'
                      | '\P{' UnicodePropertySpec '}'
                      | '\R'
                      | '.'

UnicodePropertySpec ::= BinaryProperty |
                        EnumeratedProperty (':' | '=') PropertyValue

BinaryProperty      ::= Identifier
```

```
EnumeratedProperty  ::= Identifier

PropertyValue       ::= Identifier
```

The grammar uses the following terminal symbols:

- `File`
  a file name, either absolute or relative to the directory containing the lexical specification.
- `JavaCode`
  a sequence of `BlockStatements` as described in the Java Language Specification [5], 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 an ASCII escape sequence);
  - a `\u` followed by four hexadecimal digits `[a-fA-F0-9]`, denoting a unicode escape sequence. Note that these are precisely four digits, i.e. `\u12345` is the character `\u1234` followed by the character `5`.
  - a `\U` (note that the 'U' is uppercase) followed by six hexadecimal digits `[a-fA-F0-9]`, denoting a unicode code point escape sequence;
  - `\u{H+( H+)*}`, where `H+` is one or more hexadecimal digits `[a-fA-F0-9]`, each `H+` denotes a code point - note that in character classes, only one code point is allowed;
  - a backslash followed by a three digit octal number from 000 to 377, denoting an 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 would like to match the line terminator, you should use the expression `\r|\n|\r\n` if you want the Java conventions, or `\r\n|[\r\n\u2028\u2029\u000B\u000C\u0085]` (provided as predefined class `\R`) if you want to be fully Unicode compliant (see also [4]).

The white-space 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,

white-space characters keep standing for themselves (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 postfix operators (`*`, `+`, `?`, `{n}`, `{n,m}`)
- unary prefix operators (`!`, `~`)
- concatenation (`RegExp::= RegExp Regexp`)
- union (`RegExp::= RegExp '|' RegExp`)

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

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 above.

A regular expression that consists solely of

- a `Character` matches this character.
- a character class `[...]` matches any character in that class. A `Character` is 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` or Macro or predefined character class. So `[a0-3\n]` for instance matches the characters

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

  If the list of characters is empty (i.e. just `[]`), the expression matches nothing at all (the empty set), not even the empty string. This can be useful in combination with the negation operator `!`.

  Character sets may be nested, e.g. `[[[abc]d[e]]fg]` is equivalent to `[abcdefg]`.

  Supported character set operations:

  - Union (`||`), e.g. `[[a-c]||[d-f]]`, equivalent to `[a-cd-f]`: this is the default character set operation when no operator is specified.
  - Intersection (`&&`), e.g. `[[a-f]&&[f-m]]`, equivalent to `[f]`.
  - Set difference (`--`), e.g. `[[a-z]--m]`, equivalent to `[a-ln-z]`.
  - Symmetric difference (`~~`): the union of two classes minus their intersection. For instance

    ```
    [\p{Letter}~~\p{ASCII}]
    ```

    is equivalent to

    ```
    [[\p{Letter}||\p{ASCII}]--[\p{Letter}&&\p{ASCII}]]
    ```

    the set of characters that are present in either `\p{Letter}` or in `\p{ASCII}`, but not in both.

- a negated character class `'[^...]'` matches all characters not listed in the class. If the list of characters is empty (i.e. `[^]`), the expression matches any character of the input character set, including unpaired Unicode surrogate characters.
- a string `" StringCharacter+ "` matches the exact text enclosed in double quotes. All meta characters apart from `\` and `"` lose their special meaning inside a string. See also the `%ignorecase` switch.
- 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:
  - two predefined character classes that are determined by Java library functions in class `java.lang.Character`:

    ```
    [:jletter:]       isJavaIdentifierStart()
    [:jletterdigit:]  isJavaIdentifierPart()
    ```

  - four predefined character classes equivalent to the following Unicode properties (described below):

    ```
    [:letter:]     \p{Letter}
    [:digit:]      \p{Digit}
    [:uppercase:]  \p{Uppercase}
    [:lowercase:]  \p{Lowercase}
    ```

  - the following meta characters, equivalent to these (sets of) Unicode Properties (described below):

    ```
    \d  \p{Digit}
    \D  \P{Digit}
    \s  \p{Whitespace}
    \S  \P{Whitespace}
    \w  [\p{Alpha}\p{Digit}\p{Mark}
         \p{Connector Punctuation}\p{Join Control}]
    \W  [^\p{Alpha}\p{Digit}\p{Mark}
          \p{Connector Punctuation}\p{Join Control}]
    ```

  - Unicode Properties are character classes specified by each version of the Unicode Standard. JFlex supports a subset of all defined Properties for each supported Unicode version. To see the full list of supported Properties, give the `-uniprops <ver>` option on the JFlex command line, where `<ver>` is the Unicode version. Some Properties have aliases; JFlex recognizes all aliases for all supported properties. JFlex supports loose matching of Properties: case distinctions, whitespace, hyphens, and underscores are ignored.

    To refer to a Unicode Property, use the `\p{...}` syntax, e.g. the Greek Block can be referred to as `\p{Block:Greek}`. To match all characters not included in a property, use the `\P{...}` syntax (note that the 'P' is uppercase), e.g. to match all characters that are **not** letters, use `\P{Letter}`.

See UTS#18 [4] for a description of and links to definitions of some supported Properties. UnicodeSet [11] is an online utility to show the character sets corresponding to Unicode Properties and set operations on them, but only for the most recent Unicode version.

- Dot (`.`) matches `[^\r\n\u2028\u2029\u000B\u000C\u0085]`.
  Use the `-legacydot` option to instead match `[^\n]`.
  Note that unpaired Unicode surrogate chars `[\uD800-\uDFFF]` are not matched by `.`.
- `\R` matches any newline: `\r\n|[\r\n\u2028\u2029\u000B\u000C\u0085]`.

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

- `a | b` (union)

  is the regular expression that matches all input 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+` (iteration)

  is equivalent to `aa*`

- `a?` (option)

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

- `!a` (negation)

  matches everything but the strings matched by `a`. Use with care: the construction of `!a` involves an additional, possibly exponential NFA to DFA transformation on the NFA for `a`. Note that with negation and union you also have (by applying DeMorgan) intersection and set difference: the intersection of `a` and `b` is `!(!a|!b)`, the expression that matches everything of `a` not matched by `b` is `!(!a|b)`

- `~a` (upto)

  matches everything up to (and including) the first occurrence of a text matched by `a`. The expression `~a` is equivalent to `!([^]* a [^]*) a`. A traditional C-style comment is matched by `"/*" ~"*/"`

- `a {n}` (repeat)

  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 occurrence of `\r|\n|\r\n|\u2028|\u2029|\u000B|\u000C|\u0085` (see also [4]) 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 look-ahead expression. A look-ahead expression is either `$` (the end of line operator) or `/` followed by an arbitrary regular expression. In both cases the look-ahead is not consumed and not included in the matched text region, but it **is** considered while determining which rule has the longest match (see also How the input is matched).

In the `$` case, `r` is only matched at the end of a line in the input. The end of a line is denoted by the regular expression `\r|\n|\r\n|\u2028|\u2029|\u000B|\u000C|\u0085`. So `a$` is equivalent to `a / \r|\n|\r\n|\u2028|\u2029|\u000B|\u000C|\u0085`. This is different to the situation described in [4]: since in JFlex `$` is a true trailing context, the end of file does **not** count as end of line.

For arbitrary look-ahead (also called *trailing context*) the expression is matched only when followed by input that matches the trailing context.

JFlex allows lex/flex style `<<EOF>>` rules in lexical specifications. A rule

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

is very similar to the `%eofval` directive. 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 How the input is matched) 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` and `%byaccj` options 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 working with trailing context expressions. The expression `a | (c / d) | b` is not a syntactically legal regular expression, but can 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 appears 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 includes the currently active lexical state of the scanner or if the set of associated lexical states is empty and the currently active lexical state is inclusive. Exclusive and inclusive states only differ in this one point: rules with an empty set of associated states.
- 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 the inclusive lexical state `YYINITIAL`, which is always declared by default.
- The set of lexical states associated with a regular expression is the `StateList` that precedes the expression. If a rule is contained in one or more `StateGroups`, then the states of these are also associated with the rule, i.e. they accumulate over `StateGroups`.

  Example:

  ```
  %states A, B
  %xstates C
  %%
  expr1                   { yybegin(A); action }
  <YYINITIAL, A> expr2    { action }
  <A> {
    expr3                 { action }
    <B,C> expr4           { action }
  }
  ```

  The first line declares two (inclusive) lexical states `A` and `B`, the second line an exclusive lexical state `C`. The default (inclusive) state `YYINITIAL` is always implicitly there and doesn't need to be declared. The rule with `expr1` has no states listed, and is thus matched in all states but the exclusive ones, i.e. `A`, `B`, and `YYINITIAL`. In its action, the

scanner is switched to state `A`. The second rule `expr2` can only match when the scanner is in state `YYINITIAL` or `A`. The rule `expr3` can only be matched in state `A` and `expr4` in states `A`, `B`, and `C`.

- 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. There is no guarantee that the values of these integer constants are distinct. They are pointers into the generated DFA table, and if JFlex recognises two states as lexically equivalent (if they are used with the exact same set of regular expressions), then the two constants will get the same value.

5 Encodings, Platforms, and Unicode

This section discusses Unicode and encodings, cross platform scanning, and how to deal with binary data.

5.1 The Problem

Java aims to be implementation platform independent, yet different platforms use different ways to encode characters. Moreover, a file written on one platform, say Windows, may later be read by a scanner on another platform, for instance Linux.

If a program reads a file from disk, what it really reads is a stream of bytes. These bytes can be mapped to characters in different ways. For instance, in standard ASCII, the byte value 65 stands for the character `A`, and in the encoding `iso-latin-1`, the byte value 213 stands for the umlaut character `ä`, but in the encoding `iso-latin-2` the value 213 is `é` instead. As long as one encoding is used consistently, this is no problem. Some characters may not be available in the encoding you are using, but at least the interpretation of the mapping between bytes and characters agrees between different programs.

When your program runs on more than one platform, however, as is often the case with Java, things become more complex. Java's solution to this is to use Unicode internally. Unicode aims to be able to represent all known character sets and is therefore a perfect base for encoding things that might get used all over the world and on different platforms. To make things work correctly, you still have to know where you are and how to map byte values to Unicode characters and vice versa, but the important thing is, that this mapping is at least possible (you can map Kanji characters to Unicode, but you cannot map them to ASCII or `iso-latin-1`).

5.2 Scanning text files

Scanning text files is the standard application for scanners like JFlex. Therefore it should also be the most convenient one. Most times it is.

The following scenario works fine: You work on a platform X, write your lexer specification there, can use any obscure Unicode character in it as you like, and compile the program. Your users work on any platform Y (possibly but not necessarily something different from X), they write their input files on Y and they run your program on Y. No problems.

Java does this as follows: If you want to read anything in Java that is supposed to contain text, you use a `FileReader`, which converts the bytes of the file into Unicode characters with the platform's default encoding. If a text file is produced on the same platform, the platform's default encoding should do the mapping correctly. Since JFlex also uses readers and Unicode internally, this mechanism also works for the scanner specifications. If you write an `A` in your text editor and the editor uses the platform's encoding (say `A` is 65), then Java translates this into the logical Unicode `A` internally. If a user writes an `A` on a completely different platform (say `A` is 237 there), then Java also translates this into the logical Unicode `A` internally. Scanning is performed after that translation and both match.

Note that because of this mapping from bytes to characters, you should always use the `%unicode` switch in you lexer specification if you want to scan text files. `%8bit` may not be enough, even if you know that your platform only uses one byte per character. The encoding `Cp1252` used on many Windows machines for instance knows 256 characters, but the character ’ with `Cp1252` code `\x92` has the Unicode value `\u2019`, which is larger than 255 and which would make your scanner throw an `ArrayIndexOutOfBoundsException` if it is encountered.

So for the usual case you don't have to do anything but use the `%unicode` switch in your lexer specification.

Things may break when you produce a text file on platform X and consume it on a different platform Y. Let's say you have a file written on a Windows PC using the encoding `Cp1252`. Then you move this file to a Linux PC with encoding `ISO 8859-1` and there you run your scanner on it. Java now thinks the file is encoded in `ISO 8859-1` (the platform's default encoding) while it really is encoded in `Cp1252`. For most characters `Cp1252` and `ISO 8859-1` are the same, but for the byte values `\x80` to `\x9f` they disagree: `ISO 8859-1` is undefined there. You can fix the problem by telling Java explicitly which encoding to use. When constructing the `InputStreamReader`, you can give the encoding as argument. The line

```
Reader r = new InputStreamReader(input, Cp1252);
```

will do the trick.

Of course the encoding to use can also come from the data itself: for instance, when you scan an HTML page, it may have embedded information about its character encoding in the headers.

More information about encodings, which ones are supported, how they are called, and how to set them may be found in the official Java documentation in the chapter about internationalisation. The link http://docs.oracle.com/javase/7/docs/technotes/guides/intl/ leads to an online version of this for Oracle's JDK 1.7.

5.3 Scanning binaries

Scanning binaries is both easier and more difficult than scanning text files. It's easier because you want the raw bytes and not their meaning, i.e. you don't want any translation. It's more difficult because it's not so easy to get “no translation” when you use Java readers.

The problem (for binaries) is that JFlex scanners are designed to work on text. Therefore the interface is the `Reader` class. You can still get a binary scanner when you write your

own custom `InputStreamReader` class that explicitly does no translation, but just copies byte values to character codes instead. It sounds quite easy, and actually it is no big deal, but there are a few pitfalls on the way. In the scanner specification you can only enter positive character codes (for bytes that is `\x00` to `\xFF`). Java's `byte` type on the other hand is a signed 8 bit integer (-128 to 127), so you have to convert them accordingly in your custom `Reader`. Also, you should take care when you write your lexer spec: if you use text in there, it gets interpreted by an encoding first, and what scanner you get as result might depend on which platform you run JFlex on when you generate the scanner (this is what you want for text, but for binaries it gets in the way). If you are not sure, or if the development platform might change, it's probably best to use character code escapes in all places, since they don't change their meaning.

5.4 Conformance with Unicode Regular Expressions UTS#18

This section gives details about JFlex 1.8.1's conformance with the requirements for Basic Unicode Support Level 1 given in UTS#18 [4].

5.4.1 RL1.1 Hex Notation

> *To meet this requirement, an implementation shall supply a mechanism for specifying any Unicode code point (from U+0000 to U+10FFFF), using the hexadecimal code point representation.*

JFlex conforms. Syntax is provided to express values across the whole range, via `\uXXXX`, where `XXXX` is a 4-digit hex value; `\Uyyyyyy`, where `yyyyyy` is a 6-digit hex value; and `\u{X+( X+)*}`, where `X+` is a 1-6 digit hex value.

5.4.2 RL1.2 Properties

> *To meet this requirement, an implementation shall provide at least a minimal list of properties, consisting of the following: General_Category, Script and Script_Extensions, Alphabetic, Uppercase, Lowercase, White_Space, Noncharacter_Code_Point, Default_Ignorable_Code_Point, ANY, ASCII, ASSIGNED.*
>
> *The values for these properties must follow the Unicode definitions, and include the property and property value aliases from the UCD. Matching of Binary, Enumerated, Catalog, and Name values, must follow the Matching Rules from [UAX44].*

JFlex conforms. The minimal set of properties is supported, as well as a few others. To see the full list of supported properties, use the JFlex command line option `--uniprops <ver>`, where `<ver>` is the Unicode version. Loose matching is performed: case distinctions, whitespace, underscores and hyphens in property names and values are ignored.

5.4.3 RL1.2a Compatibility Properties

> *To meet this requirement, an implementation shall provide the properties listed in Annex C: Compatibility Properties, with the property values as listed there. Such an*

implementation shall document whether it is using the Standard Recommendation or POSIX-compatible properties.

JFlex does not fully conform. The Standard Recommendation version of the Annex C Compatibility Properties are provided, with two exceptions: `\X` Extended Grapheme Clusters; and `\b` Default Word Boundaries.

5.4.4 RL1.3 Subtraction and Intersection

To meet this requirement, an implementation shall supply mechanisms for union, intersection and set-difference of Unicode sets.

JFlex conforms by providing these mechanisms, as well as symmetric difference.

5.4.5 RL1.4 Simple Word Boundaries

To meet this requirement, an implementation shall extend the word boundary mechanism so that:

1. *The class of* `<word_character>` *includes all the Alphabetic values from the Unicode character database, from UnicodeData.txt [UData], plus the decimals (General_Category = Decimal_Number, or equivalently Numeric_Type = Decimal), and the U+200C ZERO WIDTH NON-JOINER and U+200D ZERO WIDTH JOINER (Join_Control=True). See also Annex C: Compatibility Properties.*
2. *Nonspacing marks are never divided from their base characters, and otherwise ignored in locating boundaries.*

JFlex does not conform: `\b` does not match simple word boundaries.

5.4.6 RL1.5 Simple Loose Matches

To meet this requirement, if an implementation provides for case-insensitive matching, then it shall provide at least the simple, default Unicode case-insensitive matching, and specify which properties are closed and which are not.

To meet this requirement, if an implementation provides for case conversions, then it shall provide at least the simple, default Unicode case folding.

JFlex conforms. All supported Unicode Properties are closed.

5.4.7 RL1.6 Line Boundaries

To meet this requirement, if an implementation provides for line-boundary testing, it shall recognize not only CRLF, LF, CR, but also NEL (U+0085), PARAGRAPH SEPARATOR (U+2029) and LINE SEPARATOR (U+2028).

JFlex conforms.

5.4.8 RL1.7 Supplementary Code Points

> *To meet this requirement, an implementation shall handle the full range of Unicode code points, including values from U+FFFF to U+10FFFF. In particular, where UTF-16 is used, a sequence consisting of a leading surrogate followed by a trailing surrogate shall be handled as a single code point in matching.*

JFlex conforms.

6 A few words on performance

This section gives tips on how to make your specification produce a faster scanner.

In general, the regular expression matching generated by JFlex has very good performance. It is DFA-based (deterministic finite automata) and does not require backtracking over alternative as for instance perl-style regular expression matching does. In the optimal case, each character is only examined once, in some situations explained below, a small amount of backtracking is necessary to determine the longest match.

Even within the class of DFA-based scanners, JFlex generated scanners usually show very good performance without special optimisations. The following lists a few 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

  While there is no backtracking for expressions like `a|b` in JFlex, some backtracking is still 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 characters up to `'j'` to decide that rule `.` should be matched. All characters of `verylong` have to be read again for the next matching process.

  From the C/C++ flex [8] man page: *Getting rid of backtracking is messy and often may be an enormous amount of work for a complicated scanner.* 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 possibility 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, which means this kind of backtracking for programming language scanners often concerns only at most a single character.

- Avoid line and column counting

  It costs multiple additional comparisons per input character and the matched text has to be re-scanned 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 look-ahead expressions and the end of line operator `$`

  In the best case, the trailing context will first have to be read and then (because it is not to be consumed) re-read again. The cases of fixed-length look-ahead and fixed-length base expressions are handled efficiently by matching the concatenation and then pushing back the required amount of characters. This extends to the case of a disjunction of fixed-length look-ahead expressions such as `r1 / \r|\n|\r\n`. All other cases `r1 / r2` are handled by first scanning the concatenation of `r1` and `r2`, and then finding the correct end of `r1`. The end of `r1` is found by scanning forwards in the match again, marking all possible `r1` terminations, and then scanning the reverse of `r2` backwards from the end until a start of `r2` intersects with an end of `r1`. This algorithm is linear in the size of the input (not quadratic or worse as backtracking is), but about a factor of 2 slower than normal scanning. It also consumes memory proportional to the size of the matched input for `r1 r2`.

- Avoid the beginning of line operator `^`

  It costs multiple additional comparisons per match. In some cases one extra look-ahead 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, setting up the internal state of the scanner is necessary and induces a small overhead.

Note that writing more rules in a specification does *not* make the generated scanner slower.

The two main rules of optimisation 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 optimised in a later state of the development process.

7 Porting Issues

7.1 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 for this purpose. A JLex specification is well formed, when it

- generates a working scanner with JLex
- doesn't contain the unescaped characters `!` and `~`

  They are operators in JFlex while JLex treats them as normal input characters. You can easily port such a JLex specification to JFlex by replacing every `!` with `\!` and every `~` with `\~` in all regular expressions.
- has only complete regular expressions surrounded by parentheses in macro definitions

  This may sound a bit harsh, but is usually not a big problem – it can also help you find some disgusting bugs in your specification that went unnoticed so far. In JLex, the 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, things like

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

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

  Basically, JLex uses C-preprocessor style macros, whereas JFlex uses grammar definitions.

  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.

7.2 Porting from lex/flex

This section gives an incomplete overview of potential pitfalls and steps for porting a lexical specification from the C/C++ tools `lex` and `flex` [8] 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 too much work.

7.2.1 Basic structure

A lexical specification for flex has the following basic structure:

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

The `user code` section usually contains C code that is used in actions of the `rules` part of the specification. For JFlex, this code will have to be translated to Java, and most of it will then go into the class code `%{..%}` directive in the `options and declarations` section.

7.2.2 Macros and Regular Expression Syntax

The `definitions` section of a flex specification is quite similar 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. Some attention is needed for escape sequences present in flex (such as `\a`) that are not supported in JFlex. These escape sequences should be transformed into their unicode equivalent.

7.2.3 Character Classes

Flex offers the character classes directly supported by C, JFlex offers the ones supported by Java. These classes will sometimes have to be listed manually.

In flex more special characters lose their meaning in character classes. In particular[1]:

- in flex `[][]` is the character class containing `]` and `[`, whereas in JFlex, the expression means "empty expression" followed by "empty expression". To get `]` and `[` in JFlex, use for instance `[\]\[]`.
- the classes `[]` and `[^]` are illegal in flex, but have meaning in JFlex.
- in flex `["]` is legal, in JFlex you need `[\"]`.

7.2.4 Lexical Rules

Since flex is mostly Unix based, the '`^`' (beginning of line) and '`$`' (end of line) operators, consider the `\n` character as only line terminator. This should usually not cause much problems, but you should be prepared for occurrences of `\r` or `\r\n` or one of the characters `\u2028`, `\u2029`, `\u000B`, `\u000C`, or `\u0085`. They are considered to be line terminators in Unicode and therefore may not be consumed when `^` or `$` is present in a rule.

[1] Thanks to Dimitri Maziuk for pointing these out.

8 Working together

8.1 JFlex and CUP

One of the design goals of JFlex was to make interfacing with the parser generators CUP [6] and CUP2 [9] as easy as possible. This has been done by providing the `%cup` and `%cup2` directives in JFlex. However, each interface has two sides. This section concentrates on the CUP side of the story.

8.1.1 CUP2

Please refer to the CUP2 [9] documentation, which provides instructions on how to interface with JFlex. The CUP2 JFlex patch provided there is not necessary any more for JFlex versions greater than 1.5.0.

8.1.2 CUP version 0.10j and above

Since CUP version 0.10j, interfacing with JFlex has been simplified greatly by the new CUP scanner interface `java_cup.runtime.Scanner`. JFlex lexers now implement this interface automatically when the `%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 concentrate on your grammar.

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

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

8.1.3 Custom symbol interface

If you have used the `-symbol` command line switch of CUP to change the name of the generated symbol interface, you have to tell JFlex about this change of interface so that correct end-of-file code is generated. You can do so either by using an `%eofval{` directive or by using an `<<EOF>>` rule.

If your new symbol interface is called `mysym` for example, the corresponding code in the jflex specification would be either

```
%eofval{
  return mysym.EOF;
%eofval}
```

in the macro/directives section of the spec, or it would be

```
<<EOF>>  { return mysym.EOF; }
```

in the rules section of your spec.

8.1.4 Using existing JFlex/CUP specifications with CUP 0.10j and above

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 more recent versions is that JFlex scanners now automatically implement the `java_cup.runtime.Scanner` interface. This means the scanning function 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 don't mind changing 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). The main method calling the parser would then construct the parser as shown in the section above.

The JFlex specification does not need to be changed.

8.2 JFlex and BYacc/J

JFlex has built-in support for the Java extension BYacc/J [7] by Bob Jamison to the classical Berkeley Yacc parser generator. This section describes how to interface BYacc/J with JFlex. It builds on many helpful suggestions and comments from Larry Bell.

Since Yacc's architecture is a bit different from CUP's, the interface setup also works in a slightly different manner. BYacc/J expects a function `int yylex()` in the parser class that returns each next token. Semantic values are expected in a field `yylval` of type `parserval` where `parser` is the name of the generated parser class.

For a small calculator example, one could use a setup like the following on the JFlex side:

```
%%

%byaccj

%{
  /* store a reference to the parser object */
  private parser yyparser;

  /* constructor taking an additional parser object */
  public Yylex(java.io.Reader r, parser yyparser) {
    this(r);
    this.yyparser = yyparser;
  }
%}

NUM = [0-9]+ ("." [0-9]+)?
NL  = \n | \r | \r\n

%%

/* operators */
"+" |
..
"(" |
")"    { return (int) yycharat(0); }

/* newline */
{NL}   { return parser.NL; }

/* float */
{NUM}  { yyparser.yylval = new parserval(Double.parseDouble(yytext()));
         return parser.NUM; }
```

The lexer expects a reference to the parser in its constructor. Since Yacc allows direct use of terminal characters like `'+'` in its specifications, we just return the character code for single char matches (e.g. the operators in the example). Symbolic token names are stored as `public static int` constants in the generated parser class. They are used as in the `NL` token above. Finally, for some tokens, a semantic value may have to be communicated to the parser. The `NUM` rule demonstrates how.

A matching BYacc/J parser specification would look like this:

```
%{
  import java.io.*;
%}

%token NL          /* newline  */
%token <dval> NUM  /* a number */

%type <dval> exp

%left '-' '+'
..
%right '^'         /* exponentiation */

%%

..

exp:     NUM          { $$ = $1; }
       | exp '+' exp  { $$ = $1 + $3; }
       ..
       | exp '^' exp  { $$ = Math.pow($1, $3); }
       | '(' exp ')'  { $$ = $2; }
       ;

%%
  /* a reference to the lexer object */
  private Yylex lexer;

  /* interface to the lexer */
  private int yylex () {
    int yyl_return = -1;
    try {
      yyl_return = lexer.yylex();
    }
    catch (IOException e) {
      System.err.println("IO error :"+e);
    }
    return yyl_return;
  }

  /* error reporting */
  public void yyerror (String error) {
    System.err.println ("Error: " + error);
  }

  /* lexer is created in the constructor */
  public parser(Reader r) {
    lexer = new Yylex(r, this);
  }

  /* that's how you use the parser */
```

```
  public static void main(String args[]) throws IOException {
    parser yyparser = new parser(new FileReader(args[0]));
    yyparser.yyparse();
  }
```

Here, the customised part is mostly in the user code section. We create the lexer in the constructor of the parser and store a reference to it for later use in the parser's `int yylex()` method. This `yylex` in the parser only calls `int yylex()` of the generated lexer and passes the result on. If something goes wrong, it returns -1 to indicate an error.

Runnable versions of the specifications above are located in the `examples/byaccj` directory of the JFlex distribution.

8.3 JFlex and Jay

Combining JFlex with the Jay Parser Generator [10] is quite simple. The Jay Parser Generator defines an interface called `<parsername>.yyInput`. In the JFlex source the directive

```
%implements <parsername>.yyInput
```

tells JFlex to generate the corresponding class declaration.

The three interface methods to implement are

- `advance()` which should return a boolean that is `true` if there is more work to do and `false` if the end of input has been reached,
- `token()` which returns the last scanned token, and
- `value()` which returns an Object that contains the (optional) value of the last read token.

The following shows a small example with Jay parser specification and corresponding JFlex code. First of all the Jay code (in a file `MiniParser.jay`):

```
%{
//
// Prefix Code like Package declaration,
// imports, variables and the parser class declaration
//

import java.io.*;
import java.util.*;

public class MiniParser
{

%}

// Token declarations, and types of non-terminals

%token DASH COLON
```

```
%token <Integer> NUMBER

%token <String> NAME

%type <Gameresult> game
%type <Vector<Gameresult>> gamelist

// start symbol
%start gamelist

%%

gamelist: game        { $$ = new Vector<Gameresult>();
                        $<Vector<Gameresult>>$.add($1);
                      }
  |  gamelist game    { $1.add($2); }

game: NAME DASH NAME NUMBER COLON NUMBER {
      $$ = new Gameresult($1, $3, $4, $6); }

%%

  // supporting methods part of the parser class
  public static void main(String argv[])
  {
    MiniScanner scanner = new MiniScanner(new InputStreamReader(System.in));
    MiniParser parser = new MiniParser();
    try {
      parser.yyparse (scanner);
    } catch (final IOException ioe) {
      System.out.println("I/O Exception : " + ioe.toString());
    } catch (final MiniParser.yyException ye) {
      System.out.println ("Oops : " + ye.toString());
    }
  }

} // closing brace for the parser class

class Gameresult {
  String homeTeam;
  String outTeam;
  Integer homeScore;
  Integer outScore;

  public Gameresult(String ht, String ot, Integer hs, Integer os)
  {
    homeTeam = ht;
    outTeam = ot;
    homeScore = hs;
    outScore = os;
  }
}
```

The corresponding JFlex code (MiniScanner.jflex) could be

```
%%

%public
%class MiniScanner
%implements MiniParser.yyInput
%integer

%line
%column
%unicode

%{
private int token;
private Object value;

// the next 3 methods are required to implement the yyInput interface

public boolean advance() throws java.io.IOException {
  value = new String("");
  token = yylex();
  return (token != YYEOF);
}

public int token() {
  return token;
}

public Object value() {
  return value;
}

%}

nl =     [\n\r]+
ws =     [ \t\b\015]+
number = [0-9]+
name =   [a-zA-Z]+
dash =   "-"
colon =  ":"

%%

{nl}      { /* do nothing */ }
{ws}      { /* happy meal */ }
{name}    { value = yytext(); return MiniParser.NAME; }
{dash}    { return MiniParser.DASH; }
{colon}   { return MiniParser.COLON; }
{number}  { try  {
              value = Integer.valueOf(Integer.parseInt(yytext()));
            } catch (NumberFormatException nfe) {
              // shouldn't happen
```

```
        throw new Error();
      }
      return MiniParser.NUMBER;
    }
```

This small example reads an input like

```
Borussia - Schalke 3:2
ACMilano - Juventus 1:4
```

9 Bugs and Deficiencies

9.1 Deficiencies

JFlex 1.8.1 conforms with Unicode Regular Expressions UTS#18 [4] Basic Unicode Support Level 1, with a few exceptions - for details see UTS # 18 Conformance.

9.2 Bugs

As of 28 February 2020, no major open problems are known for JFlex version 1.8.1.

Please use the JFlex github issue tracker for any problems that have been reported since then.

10 Copying and License

JFlex is free software, published under a BSD-style license.

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

See the file `COPYRIGHT` for more information.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java: Basic Techniques*. Cambridge University Press, 1998.

[3] Elliot Berk. JLex: A lexical analyzer generator for Java. http://www.cs.princeton.edu/~appel/modern/java/JLex/, 1996.

[4] Mark Davis and Andy Heninger. Unicode regular expressions. http://www.unicode.org/reports/tr18/tr18-17.html, 2013.

[5] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[6] Scott E. Hudson. CUP LALR parser generator for Java. http://www2.cs.tum.edu/projects/cup/, 1996.

[7] Bob Jamison. BYacc/J. http://byaccj.sourceforge.net.

[8] Vern Paxson. flex - the fast lexical analyzer generator. http://flex.sourceforge.net, 1995.

[9] Michael Petter. CUP2 user manual. http://www2.in.tum.de/cup2, 2008.

[10] Axel T. Schreiner. Jay parser generator. http://www.cs.rit.edu/~ats/projects/lp/doc/jay/package-summary.html, June 2006.

[11] Unicode utilities: UnicodeSet. http://unicode.org/cldr/utility/list-unicodeset.jsp, 2015.