The Fast Lexical Analyser Generator

JFlex User's Manual

Version 1.4.1, November 7, 2004

Contents

1 Introduction

JFlex is a lexical analyzer generator for Java[1] written in Java. It is also a rewrite of the very useful tool JLex [3] which was developed by Elliot Berk at Princeton University. As Vern Paxson states for his C/C++ tool flex [11]: They do not share any code though.

1.1 Design goals

The main design goals of JFlex are:

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

1.2 About this manual

This manual gives a brief but complete description of the tool JFlex. It assumes that you are familiar with the issue of lexical analysis. The references [1], [2], and [13] provide a good introduction to this topic.

The next section of this manual describes *installation procedures* for JFlex. If you never worked with JLex or just want to compare a JLex and a JFlex scanner specification you should also read *Working with JFlex - an example* (section 3). All options and the complete specification syntax are presented in *Lexical specifications* (section 4); *Encodings, Platforms, and Unicode* (section 5) provides information about scannig text vs. binary files. If you are interested in performance considerations and comparing JLex with JFlex speed, *a few words on performance* (section 6) might be just right for you. Those who want to use their old JLex specifications may want to check out section 7.1 *Porting from JLex* to avoid possible problems with not portable or non standard JLex behavior that has been fixed in JFlex. Section 7.2 talks about porting scanners from the Unix tools lex and flex. Interfacing JFlex scanners with the LALR parser generators CUP and BYacc/J is explained in *working together* (section 8). Section 9 *Bugs* gives a list of currently known active bugs. The manual concludes with notes about *Copying and License* (section 10) and references.

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

2 Installing and Running JFlex

2.1 Installing JFlex

2.1.1 Windows

To install JFlex on Windows 95/98/NT/XP, follow these three steps:

1. Unzip the file you downloaded into the directory you want JFlex in (using something like WinZip[2]). If you unzipped it to say `C:\`, the following directory structure should be generated:

```
C:\JFlex\
      +--bin\                   (start scripts)
      +--doc\                   (FAQ and manual)
      +--examples\
               +--binary\       (scanning binary files)
               +--byaccj\       (calculator example for BYacc/J)
               +--cup\          (calculator example for cup)
               +--interpreter\  (interpreter example for cup)
               +--java\         (Java lexer specification)
               +--simple\       (example scanner)
               +--standalone\   (a simple standalone scanner)
      +--lib\                   (the precompiled classes)
      +--src\
          +--JFlex\             (source code of JFlex)
          +--JFlex\gui          (source code of JFlex UI classes)
          +--java_cup\runtime\  (source code of cup runtime classes)
```

2. Edit the file `bin\jflex.bat` (in the example it's `C:\JFlex\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`)

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

2.1.2 Unix with tar archive

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

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

  ```
  tar -C /usr/share -xvzf jflex-1.4.1.tar.gz
  ```

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

(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/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 MD5 checksum available on the JFlex download page. If you put the checksum file in the same directory as the archive, you run:

```
md5sum --check jflex-1.4.1.tar.gz.md5
```

It should tell you

```
jflex-1.4.1.tar.gz:  OK
```

2.1.3 Linux with RPM

- become root
- issue
  ```
  rpm -U jflex-1.4.1-0.rpm
  ```

You can verify the integrity of the downloaded `rpm` file with

```
rpm --checksig jflex-1.4.1-0.rpm
```

This requires my pgp public key. If you don't have it, you can use

```
rpm --checksig --nopgp jflex-1.4.1-0.rpm
```

or you can get it from `http://www.jflex.de/public-key.asc`.

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.jar` in your `CLASSPATH` environment variable instead.

Then you run JFlex with:

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

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

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

`--nomin`
: skip the DFA minimization 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 minimized DFA. This feature is still in alpha status, and not fully implemented yet.

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

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

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

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

`--version`
: print version number

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

`--pack`
: use the %pack code generation method by default

`--table`
: use the %table code generation method by default

`--switch`
: use the %switch code generation method by default

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

3 A simple Example: How to work with JFlex

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

```
/* JFlex example: part of 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   = "/*" [^*] ~"*/" | "/*" "*"+ "/"
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 */
.|\n                            { 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). Beside `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. See also section 5 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 (current column is accessed via `yycolumn`)

The code included in `%{...%}` is copied verbatim into the generated lexer class source. Here you can declare member variables and functions that are used inside scanner actions. In our example we declare a `StringBuffer` "`string`" in which we will store parts of string literals and two helper functions "`symbol`" that create `java_cup.runtime.Symbol` objects with position information of the current token (see section 8.1 *JFlex and CUP* for how to interface with the parser generator CUP). As JFlex options, both `%{` and `\%}` must begin a line.

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

Here some of the example macros in more detail:

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

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

3.3 Rules and Actions

The "lexical rules" section of a JFlex specification contains regular expressions and actions (Java code) that are executed when the scanner matches the associated regular expression. As the scanner reads its input, it keeps track of all regular expressions and activates the action of the expression that has the longest match. Our specification above for instance would with input "`breaker`" match the regular expression for `Identifier` and not the keyword "`break`" followed by the Identifier "`er`", because rule `{Identifier}` matches more of this input at once (i.e. it matches all of it) than any other rule in the specification. If two regular expressions both have the longest match for a certain input, the scanner chooses the action of the expression that appears first in the specification. In that way, we get for input "`break`" the keyword "`break`" and not an Identifier "`break`".

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

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

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

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

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

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

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

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

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

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

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

The expression `[^\n\r\"]+` matches all characters in the input up to the next backslash (indicating an escape sequence such as `\n`), double quote (indicating the end of the string), or line terminator (which must not occur in a string literal). The matched region of the input is referred to with `yytext()` and appended to the content of the string literal parsed so far.

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

3.4 How to get it going

- Install JFlex (see section 2 *Installing 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 report some progress messages about generating the scanner and write the generated code to the directory of your specification file.
- Compile the generated `.java` file and your own classes. (If you use CUP, generate your parser classes first)
- That's it.

4 Lexical Specifications

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

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

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

4.1 User code

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

4.2 Options and declarations

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

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

`%class "classname"`

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

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

- `%extends "classname"`

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

- `%public`

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

- `%final`

  Makes the generated class final.

- `%abstract`

  Makes the generated class abstract.

- `%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 occurences 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 genarated 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 initialized. If more than one initializer option is present, the code is concatenated in order of appearance in the specification.

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

  or (on a single line) just

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

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

- `%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. This feature is still experimental. It works, but error reporting can be strange if a syntax error occurs on the last token in the included file.

4.2.2 Scanning method

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

- `%function "name"`

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

- `%integer`
  `%int`

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

- `%intwrap`

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

- `%type "typename"`

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

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

  or (on a single line) just

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

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

4.2.3 The end of file

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

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

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

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

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

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

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

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

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

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

  or (on a single line) just

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

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

- `%eofclose`

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

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

4.2.5 CUP compatibility

You may also want to read section 8.1 *JFlex and CUP* if you are interested in how to interface your generated scanner with CUP.

- `%cup`

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

  ```
  %implements java_cup.runtime.Scanner
  %function next_token
  %type java_cup.runtime.Symbol
  %eofval{
    return new java_cup.runtime.Symbol(<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.

- `%cupsym "classname"`

  Customizes 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`, but 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 section 8.2 *JFlex and BYacc/J* if you are interested in how to interface your generated scanner with Byacc/J.

- `%byacc`

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

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

4.2.7 Code generation

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

- `%switch`

  With `%switch` JFlex will generate a scanner that has the DFA hard coded into a nested switch statement. This method gives a good deal of compression in terms of the size of the compiled `.class` file while still providing very good performance. If your scanner gets to big though (say more than about 200 states) performance may vastly degenerate and you should consider using one of the `%table` or `%pack` directives. If your scanner gets even bigger (about 300 states), the Java compiler `javac` could produce corrupted code, that will crash when executed or will give you an `java.lang.VerifyError` when checked by the virtual machine. This is due to the size limitation of 64 KB of Java methods as described in the Java Virtual Machine Specification [10]. In this case you will be forced to use the `%pack` directive, since `%switch` usually provides more compression of the DFA table than the `%table` directive.

- `%table`

  The `%table` direction causes JFlex to produce a classical table driven scanner that encodes its DFA table in an array. In this mode, JFlex only does a small amount of table compression (see [6], [12], [1] and [13] for more details on the matter of table compression) and uses the same method that JLex did up to version 1.2.1. See section 6 performance of this manual to compare these methods. The same reason as above (64 KB size limitation of methods) causes the same problem, when the scanner gets too big. This is, because the virtual machine treats static initializers of arrays as normal methods. You will in this case again be forced to use the `%pack` directive to avoid the problem.

- `%pack`

  `%pack` causes JFlex to compress the generated DFA table and to store it in one or more string literals. JFlex takes care that the strings are not longer than permitted by the class file format. The strings have to be unpacked when the first scanner object is created and initialized. After unpacking the internal access to the DFA table is exactly the same as with option `%table` — the only extra work to be done at runtime is the unpacking process which is quite fast (not noticeable in normal cases). It is in time complexity proportional to the size of the expanded DFA table, and it is static, i.e. it is done only once for a certain scanner class — no matter how often it is instantiated. Again, see section 6 performance on the performance of these scanners With `%pack`, there should be practically no limitation to the size of the scanner. `%pack` is the default setting and will be used when no code generation method is specified.

4.2.8 Character sets

- `%7bit`

  Causes the generated scanner to use an 7 bit input character set (character codes 0-127). Because this is the default value in JLex, JFlex also defaults to 7 bit scanners. If an input character with a code greater than 127 is encountered in an input at runtime, the scanner will throw an `ArrayIndexOutofBoundsException`. Not only because of this, you should consider using the `%unicode` directive. See also section 5 for information about character encodings.

- `%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 5 for more information about character encodings.

- `%unicode`
  `%16bit`

  Both options cause the generated scanner to use the full 16 bit Unicode input character set (character codes 0-65535). There will be no runtime overflow when using this set of input characters. `%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 5 for more information about character encodings.

- `%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 effect character classes. So `[a]` still only matches the character `a` and not `A`, too. Which letters are uppercase and which lowercase letters, is defined by the Unicode standard and determined by JFlex with the Java methods `Character.toUpperCase` and `Character.toLowerCase`. In JLex compatibility mode (`--jlex` switch on the command line), `%caseless` and `%ignorecase` also affect character classes.

4.2.9 Line, character and column counting

- `%char`

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

- `%line`

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

- `%column`

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

4.2.10 Obsolete JLex options

- `%notunix`

  This JLex option is obsolete in JFlex but still recognized 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 recognizes both styles of platform dependent line terminators.

- `%yyeof`

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

4.2.11 State declarations

State declarations have the following from:

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

There may be more than one line of state declarations, each starting with `%state` or `%xstate` (the first character is sufficient, `%s` and `%x` works, too). State identifiers are letters followed by a sequence of letters, digits or underscores. State identifiers can be separated by whitespace or comma.

The sequence

```
%state STATE1
%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 section 4.3.3 on the way lexical states influence how the input is matched.

4.2.12 Macro definitions

A macro definition has the form

`macroidentifier = regular expression`

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

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

This is a feature. It eliminates some very hard to find bugs in lexical specifications (such like not having parentheses around more complicated macros - which is not necessary with

JFlex). See section 7.1 *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. Macros however remain just abbreviations of the regular expressions they represent. They are not non terminals of a grammar and cannot be used recursively in any way. JFlex detects cycles in macro definitions and reports them at generation time. JFlex also warns you about macros that have been defined but never used in the "lexical rules" section of the specification.

4.3 Lexical rules

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

4.3.1 Syntax

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

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

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

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

The grammar uses the following terminal symbols:

- `JavaCode`
  a sequence of *`BlockStatements`* as described in the Java Language Specification [7], section 14.2.
- `Number`
  a non negative decimal integer.
- `Identifier`
  a letter `[a-zA-Z]` followed by a sequence of zero or more letters, digits or underscores `[a-zA-Z0-9_]`
- `Character`
  an escape sequence or any unicode character that is not one of these meta characters:
  `| ( ) { } [ ] < > \ . * + ? ^ $ / . " ~ !`
- `StringCharacter`
  an escape sequence or any unicode character that is not one of these meta characters:
  `\ "`
- An escape sequence
  - `\n \r \t \f \b`
  - a `\x` followed by two hexadecimal digits `[a-fA-F0-9]` (denoting a standard ASCII escape sequence),
  - a `\u` followed by four hexadecimal digits `[a-fA-F0-9]` (denoting an unicode escape sequence),
  - a backslash followed by a three digit octal number from 000 to 377 (denoting a standard ASCII escape sequence), or
  - a backslash followed by any other unicode character that stands for this character.

Please note that the `\n` escape sequence stands for the ASCII LF character - not for the end of line. If you 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` if you want to be fully Unicode compliant (see also [5]).

As of version 1.1 of JFlex the whitespace characters `" "` (space) and `"\t"` (tab) can be used to improve the readability of regular expressions. They will be ignored by JFlex. In character classes and strings however, whitespace characters keep standing for 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 presented above).

A regular expression that consists solely of

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

  `a 0 1 2 3 \n`

  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 may be useful in combination with the negation operator `'!'`.
- a negated character class `'[^' (Character|Character'-'Character)* ']'` 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.
- a string `'"' StringCharacter+ '" '` matches the exact text enclosed in double quotes. All meta characters but `\` and `"` loose 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:

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

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

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

`[:letter:]`	`isLetter()`
`[:digit:]`	`isDigit()`
`[:uppercase:]`	`isUpperCase()`
`[:lowercase:]`	`isLowerCase()`

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

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

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

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

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

`a+` (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 [5]) and at the beginning of input. The preceding line terminator in the input is not consumed and can be matched by another rule.

In a lexical rule, a regular expression `r` may be followed by a lookahead expression. A lookahead expression is either a '`$`' (the end of line operator) or a `'/'` followed by an arbitrary regular expression. In both cases the lookahead is not consumed and not included in the matched text region, but it *is* considered while determining which rule has the longest match (see also 4.3.3 *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 a bit different to the situation described in [5]: since in JFlex `$` is a true trailing context, the end of file does **not** count as end of line.

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

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

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

is very similar to the `%eofval` directive (section 4.2.3). The difference lies in the optional `StateList` that may precede the `<<EOF>>` rule. The action code will only be executed when the end of file is read and the scanner is currently in one of the lexical states listed in `StateList`. The same `StateGroup` (see section 4.3.3 *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 you work with trailing context expressions. The expression `a | (c / d) | b` is not syntactically legal, but can easily be expressed using the `|` action:

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

4.3.3 How the input is matched

When consuming its input, the scanner determines the regular expression that matches the longest portion of the input (longest match rule). If there is more than one regular expression that matches the longest portion of input (i.e. they all match the same input), the generated scanner chooses the expression that 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 at this 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 recognizes 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.

4.3.4 The generated class

JFlex generates exactly one file containing one class from the specification (unless you have declared another class in the first specification section).

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

The name of the class is by default `Yylex`, it is customizable with the `%class` directive (see also section 4.2.1). The input buffer of the lexer is connected with an input stream over the `java.io.Reader` object which is passed to the lexer in the generated constructor. If you want to provide your own constructor for the lexer, you should always call the generated one in it to initialize the input buffer. The input buffer should not be accessed directly, but only over the advertised API (see also section 4.3.5). Its internal implementation may change between releases or skeleton files without notice.

The main interface to the outside world is the generated scanning method (default name `yylex`, default return type `Yytoken`). Most of its aspects are customizable (name, return type, declared exceptions etc., see also section 4.2.2). If it is called, it will consume input until one of the expressions in the specification is matched or an error occurs. If an expression is matched, the corresponding action is executed. It may return a value of the specified return type (in which case the scanning method return with this value), or if it doesn't return a value, the scanner resumes consuming input until the next expression is matched. If the end of file is reached, the scanner executes the EOF action, and (also upon each further call to the scanning method) returns the specified EOF value (see also section 4.2.3).

4.3.5 Scanner methods and fields accessible in actions (API)

Generated methods and member fields in JFlex scanners are prefixed with `yy` to indicate that they are generated and to avoid name conflicts with user code copied into the class. Since user code is part of the same class, JFlex has no language means like the `private` modifier to indicate which members and methods are internal and which ones belong to the API. Instead, JFlex follows a naming convention: everything starting with a `zz` prefix like `zzStartRead` is to be considered internal and subject to change without notice between JFlex releases. Methods and members of the generated class that do not have a `zz` prefix like `yycharat` belong to the API that the scanner class provides to users in action code of the specification. They will be remain stable and supported between JFlex releases as long as possible.

Currently, the API consists of the following methods and member fields:

- `String yytext()`
  returns the matched input text region
- `int yylength()`
  returns the length of the matched input text region (does not require a `String` object to be created)
- `char yycharat(int pos)`
  returns the character at position `pos` from the matched text. It is equivalent to `yytext().charAt(pos)`, but faster. `pos` must be a value from `0` to `yylength()-1`.
- `void yyclose()`
  closes the input stream. All subsequent calls to the scanning method will return the end of file value
- `void yyreset(java.io.Reader reader)`
  closes the current input stream, and resets the scanner to read from a new input stream. All internal variables are reset, the old input stream *cannot* be reused (content of the internal buffer is discarded and lost). The lexical state is set to `YY_INITIAL`.
- `void yypushStream(java.io.Reader reader)`
  Stores the current input stream on a stack, and reads from a new stream. Lexical state, line, char, and column counting remain untouched. The current input stream can be restored with `yypopstream` (usually in an `<<EOF>>` action).

  A typical example for this are include files in style of the C preprocessor. The corresponding JFlex specification could look somewhat like this:

  ```
  "#include" {FILE}  { yypushStream(new FileReader(getFile(yytext()))); }
  ..
  <<EOF>>         { if (yymoreStreams()) yypopStream(); else return EOF; }
  ```

  This method is only available in the skeleton file `skeleton.nested`. You can find it in the `src` directory of the JFlex distribution.

- `void yypopStream()`
  Closes the current input stream and continues to read from the one on top of the stream stack.

  This method is only available in the skeleton file `skeleton.nested`. You can find it in the `src` directory of the JFlex distribution.

- `boolean yymoreStreams()`
  Returns true iff there are still streams for `yypopStream` left to read from on the stream stack.

  This method is only available in the skeleton file `skeleton.nested`. You can find it in the `src` directory of the JFlex distribution.

- `int yystate()`
  returns the current lexical state of the scanner.

- `void yybegin(int lexicalState)`
  enters the lexical state `lexicalState`

- `void yypushback(int number)`
  pushes `number` characters of the matched text back into the inputstream. They will be read again in the next call of the scanning method. The number of characters to be read again must not be greater than the length of the matched text. The pushed back characters will after the call of `yypushback` not be included in `yylength` and `yytext()`. Please note that in Java strings are unchangeable, i.e. an action code like

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

  will return the whole matched text, while

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

  will return the matched text minus the last character.

- `int yyline`
  contains the current line of input (starting with 0, only active with the `%line` directive)

- `int yychar`
  contains the current character count in the input (starting with 0, only active with the `%char` directive)

- `int yycolumn`
  contains the current column of the current line (starting with 0, only active with the `%column` directive)

5 Encodings, Platforms, and Unicode

This section tries to shed some light on the issues of Unicode and encodings, cross platform scanning, and how to deal with binary data. My thanks go to Stephen Ostermiller for his input on this topic.

5.1 The Problem

Before we dive straight into details, let's take a look at what the problem is. The problem is Java's platform independence when you want to use it. For scanners the interesting part about platform independence is character encodings and how they are handled.

If a program reads a file from disk, it gets a stream of bytes. In earlier times, when the grass was green, and the world was much simpler, everybody knew that the byte value 65 is, of course, an A. It was no problem to see which bytes meant which characters (actually these times never existed, but anyway). The normal Latin alphabet only has 26 characters, so 7 bits or 128 distinct values should surely be enough to map them, even if you allow yourself the luxury of upper and lower case. Nowadays, things are different. The world suddenly grew much larger, and all kinds of people wanted all kinds of special characters, just because they use them in their language and writing. This is were the mess starts. Since the 128 distinct values were already filled up with other stuff, people began to use all 8 bits of the byte, and extended the byte/character mappings to fit their need, and of course everybody did it differently. Some people for instance may have said "let's use the value 213 for the German character ä". Others may have found that 213 should much rather mean é, because they didn't need German and wrote French instead. As long as you use your program and data files only on one platform, this is no problem, as all know what means what, and everything gets used consistently.

Now Java comes into play, and wants to run everywhere (once written, that is) and now there suddenly is a problem: how do I get the same program to say ä to a certain byte when it runs in Germany and maybe é when it runs in France? And also the other way around: when I want to say é on the screen, which byte value should I send to the operating system?

Java's solution to this is to use Unicode internally. Unicode aims to be a superset of all known character sets and is therefore a perfect base for encoding things that might get used all over the world. 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 like a breeze: 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` or some `InputStream` together with an `InputStreamReader`. `InputStreams` return the raw bytes, the `InputStreamReader` converts the bytes 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 want to 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 a 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 internationalization. The link `http://java.sun.com/j2se/1.3/docs/guide/intl/` leads to an online version of this for Sun's JDK 1.3.

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 (there is a constructor for `InputStream` instances, but it's just there for convenience and wraps an `InputStreamReader` around it to get characters, not bytes). You can still get a binary scanner when you write your own custom `InputStreamReader` class that does explicitly 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 little 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 properly 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.

To illustrate these points, the example in `examples/binary` contains a very small binary scanner that tries to detect if a file is a Java `class` file. For that purpose it looks if the file begins with the magic number `\xCAFEBABE`.

6 A few words on performance

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

6.1 Comparison of JLex and JFlex

Scanners generated by the tool JLex are quite fast. It was however possible to further improve the performance of generated scanners using JFlex. The following table shows the results that were produced by the scanner specification of a small toy programming language (the example from the JLex website). The scanner was generated using JLex 1.2.6 and JFlex version 1.3.5 with all three different JFlex code generation methods. Then it was run on a W98 system using Sun's JDK 1.3 with different sample inputs of that toy programming language. All test runs were made under the same conditions on an otherwise idle machine.

The values presented in the table denote the time from the first call to the scanning method to returning the EOF value and the speedup in percent. The tests were run both int the mixed (HotSpot) JVM mode and the pure interpreted mode. The mixed mode JVM brings about a factor of 10 performance improvement, the difference between JLex and JFlex only decreases slightly.

KB	JVM	JLex	`%switch`	speedup	`%table`	speedup	`%pack`	speedup
496	hotspot	325 ms	261 ms	24.5 %	261 ms	24.5 %	261 ms	24.5 %
187	hotspot	127 ms	98 ms	29.6 %	94 ms	35.1 %	96 ms	32.3 %
93	hotspot	66 ms	50 ms	32.0 %	50 ms	32.0 %	48 ms	37.5 %
496	interpr.	4009 ms	3025 ms	32.5 %	3258 ms	23.1 %	3231 ms	24.1 %
187	interpr.	1641 ms	1155 ms	42.1 %	1245 ms	31.8 %	1234 ms	33.0 %
93	interpr.	817 ms	573 ms	42.6 %	617 ms	32.4 %	613 ms	33.3 %

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

KB	JVM	JLex	`%switch`	speedup	`%table`	speedup	`%pack`	speedup
496	hotspot	204 ms	140 ms	45.7 %	138 ms	47.8 %	140 ms	45.7 %
187	hotspot	83 ms	55 ms	50.9 %	52 ms	59.6 %	52 ms	59.6 %
93	hotspot	41 ms	28 ms	46.4 %	26 ms	57.7 %	26 ms	57.7 %
496	interpr.	2983 ms	2036 ms	46.5 %	2230 ms	33.8 %	2232 ms	33.6 %
187	interpr.	1260 ms	793 ms	58.9 %	865 ms	45.7 %	867 ms	45.3 %
93	interpr.	628 ms	395 ms	59.0 %	432 ms	45.4 %	432 ms	45.4 %

Execution time of single instructions depends on the platform and the implementation of the Java Virtual Machine the program is executed on. Therefore the tables above cannot be used as a reference to which code generation method of JFlex is the right one to choose in general. The following table was produced by the same lexical specification and the same input on a Linux system also using Sun's JDK 1.3.

With actions:

KB	JVM	JLex	`%switch`	speedup	`%table`	speedup	`%pack`	speedup
496	hotspot	246 ms	203 ms	21.2 %	193 ms	27.5 %	190 ms	29.5 %
187	hotspot	99 ms	76 ms	30.3 %	69 ms	43.5 %	70 ms	41.4 %
93	hotspot	48 ms	36 ms	33.3 %	34 ms	41.2 %	35 ms	37.1 %
496	interpr.	3251 ms	2247 ms	44.7 %	2430 ms	33.8 %	2444 ms	33.0 %
187	interpr.	1320 ms	848 ms	55.7 %	958 ms	37.8 %	920 ms	43.5 %
93	interpr.	658 ms	423 ms	55.6 %	456 ms	44.3 %	452 ms	45.6 %

Without actions:

KB	JVM	JLex	`%switch`	speedup	`%table`	speedup	`%pack`	speedup
496	hotspot	136 ms	78 ms	74.4 %	76 ms	78.9 %	77 ms	76.6 %
187	hotspot	59 ms	31 ms	90.3 %	48 ms	22.9 %	32 ms	84.4 %
93	hotspot	28 ms	15 ms	86.7 %	15 ms	86.7 %	15 ms	86.7 %
496	interpr.	1992 ms	1047 ms	90.3 %	1246 ms	59.9 %	1215 ms	64.0 %
187	interpr.	859 ms	408 ms	110.5 %	479 ms	79.3 %	487 ms	76.4 %
93	interpr.	435 ms	200 ms	117.5 %	237 ms	83.5 %	242 ms	79.8 %

Although all JFlex scanners were faster than those generated by JLex, slight differences between JFlex code generation methods show up when compared to the run on the W98 system.

The following table compares a handwritten scanner for the Java language obtained from the website of CUP with the JFlex generated scanner for Java that comes with JFlex in the `examples` directory. They were tested on different `.java` files on a Linux machine with Sun's JDK 1.3.

lines	KB	JVM	handwritten scanner	JFlex generated scanner	
19050	496	hotspot	824 ms	248 ms	235 % faster
6350	165	hotspot	272 ms	84 ms	232 % faster
1270	33	hotspot	53 ms	18 ms	194 % faster
19050	496	interpreted	5.83 s	3.85 s	51 % faster
6350	165	interpreted	1.95 s	1.29 s	51 % faster
1270	33	interpreted	0.38 s	0.25 s	52 % faster

Although JDK 1.3 seems to speed up the handwritten scanner if compared to JDK 1.1 or 1.2 more than the generated one, the generated scanner is still up to 3.3 times as fast as the handwritten one. One example of a handwritten scanner that is considerably slower than the equivalent generated one is surely no proof for all generated scanners being faster than handwritten. It is clearly impossible to prove something like that, since you could always write the generated scanner by hand. From a software engineering point of view however, there is no excuse for writing a scanner by hand since this task takes more time, is more difficult and therefore more error prone than writing a compact, readable and easy to change lexical specification. (I'd like to add, that I do *not* think, that the handwritten scanner from the CUP website used here in the test is stupid or badly written or anything like that. I actually think, Scott did a great job with it, and that for learning about lexers it is quite valuable to study it or even to write a similar one for oneself.)

6.2 How to write a faster specification

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

- Avoid rules that require backtracking

  From the C/C++ flex [11] manpage: *"Getting rid of backtracking is messy and often may be an enormous amount of work for a complicated scanner."* Backtracking is introduced by the longest match rule and occurs for instance on this set of expressions:

  ```
   "averylongkeyword"
  .
  ```

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

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

While this is impractical in most scanners, there is still the 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.

- Avoid line and column counting

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

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

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

- Avoid the beginning of line operator '^'

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

- Match as much text as possible in a rule.

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

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

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

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

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

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 here. 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 could otherwise be a major problem – it can also help you find some disgusting bugs in your specification that didn't show up in the first place. In JLex, a right hand side of a macro is just a piece of text, that is copied to the point where the macro is used. With this, some weird kind of stuff like

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

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

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

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

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

  are only used to write rules like

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

  and not more complex expressions like

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

  where the kind of error presented above would show up.

7.2 Porting from lex/flex

This section tries to give an overview of activities and possible problems when porting a lexical specification from the C/C++ tools lex and flex [11] available on most Unix systems to JFlex.

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

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

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

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. A little attention is needed for some escape sequences present in flex (such as `\a`) that are not supported in JFlex. These escape sequences should be transformed into their octal or hexadecimal equivalent.

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

7.2.3 Lexical Rules

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

The trailing context algorithm of flex is better than the one used in JFlex. Therefore lookahead expressions could cause major headaches. JFlex will issue an error message at generation time, if it cannot generate a scanner for a certain lookahead expression. (sorry, I have no more tips here on that yet. If anyone knows how the flex lookahead algorithm works (or any better one) and can be efficiently implemented, again: please contact me).

8 Working together

8.1 JFlex and CUP

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

8.1.1 CUP version 0.10j

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

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

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

8.1.2 Using existing JFlex/CUP specifications with CUP 0.10j

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

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

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

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

```
parser code {:
  Lexer lexer;

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

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

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

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

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

The JFlex specification does not need to be changed.

8.1.3 Using older versions of CUP

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

If you have a scanner specification that begins like this:

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

%%
```

```
%class Lexer
%cup
..
```

then it matches a CUP specification starting like

```
package PACKAGE;

parser code {:
  Lexer lexer;

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

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

..
```

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

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

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

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

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

change the parser code to:

```
package PACKAGE;

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

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

..
```

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

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

and finally change the main routine to look like

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

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

8.2 JFlex and BYacc/J

JFlex has builtin support for the Java extension BYacc/J [9] 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 that bit.

A matching BYacc/J parser specification could 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 customized 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.

9 Bugs and Deficiencies

9.1 Deficiencies

The trailing context algorithm described in [1] and used in JFlex is incorrect. It does not work, when a postfix of the regular expression matches a prefix of the trailing context and the length of the text matched by the expression does not have a fixed size. In this case JFlex will still correctly use `r1 r2` (`r1` followed by a lookahead `r2`) to determine if the rule should be matched, but it might return too many characters in `yytext` (it will return the longest match of `r1` within `r1 r2`). JFlex attempts to report these cases as errors at generation time, but the warnings are overeager. A large number of safe lookaheads are reported as unsafe.

9.2 Bugs

As of November 7, 2004 the following bugs are known in JFlex:

- The check for unsafe lookahead expressions is overeager. The lookahead algorithm itself works as advertised, but JFlex will report a large number of lookahead expressions as unsafe although they are safe.

  Workaround: Check lookahead expressions manually. A lookahead expression `r1/r2` is ok, if no postfix of `r1` can match a prefix of `r2`.

If you find new ones, please use the bugs section of the JFlex website[3] to report them.

10 Copying and License

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

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

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

See the file `COPYRIGHT` for more information.

[3] http://www.jflex.de/

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

References

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

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

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

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

[5] M. Davis, *Unicode Regular Expression Guidelines*, Unicode Technical Report #18, 2000 `http://www.unicode.org/unicode/reports/tr18/tr18-5.1.html`

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

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

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

[9] B. Jamison, *BYacc/J*, `http://troi.lincom-asg.com/~rjamison/byacc/`

[10] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, 1996, `http://java.sun.com/docs/books/vmspec/`

[11] V. Paxson, *flex - The fast lexical analyzer generator*, 1995

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

[13] R. Wilhelm, D. Maurer, *Übersetzerbau*, Berlin 1997^2