**JLex Notes**

**Overview**

JLex is a scanner generator that produces Java code. Here's a picture illustrating how to create and run a program using JLex:

+-----------+

JLex specification ---> | JLex.Main | ---> Java source code

(xxx.jlex) +-----------+ (xxx.jlex.java)

+------------------+

xxx.jlex.java ---> | jikes (or javac) | ---> Yylex.class

+------------------+

+--------+

Yylex.class-------> | java | ---> output of Main

Main.class--------> | |

+--------+

The input to Jlex is a specification that includes a set of regular expressions and associated actions. The output of Jlex is a Java source file that defines a class named **Yylex**. Yylex includes a constructor that is called with one argument: the input stream (an InputStream or a Reader). It also includes a method called **next\_token**, which returns the next token in the input.

The picture above assumes that a class named Main has been defined that contains the main program of interest. That program will declare an object of type Yylex, and will include calls to the Yylex constructor and its next\_token method.

**Format of a JLex Specification**

A JLex specification has three parts, separated by double percent signs:

1. **User code**: this part of the specification will not be discussed here.
2. **JLex directives**: This includes macro definitions (described below). See the [JLex Reference Manual](http://www.cs.princeton.edu/~appel/modern/java/JLex/manual.html) for more information about this part of the specification.
3. **Regular expression rules**: These rules specify how to divide up the input into tokens. Each rule includes an optional state list, a regular expression, and an associated action.

We will discuss the regular expression rules part first.

**Regular Expressions Rules**

The state-list part of a rule will not be discussed here. See the [JLex Reference Manual](http://www.cs.princeton.edu/~appel/modern/java/JLex/manual.html) if you are interested in learning about this part of a JLex specification. Ignoring state-lists, the form of a regular expression rule is:

regular-expression { action }

^ ^

| |

the pattern to java code to be executed

be matched when the pattern is matched

When the scanner's next\_token method is called, it repeats:

1. Find the longest sequence of characters in the input (starting with the current character) that matches a pattern.
2. Perform the associated action.

until an action causes the next\_token method to return. If there are several patterns that match the same (longest) sequence of characters, then the first such pattern is considered to be matched (so the order of the patterns can be important).

If an input character is not matched in any pattern, the scanner throws an exception (so it is important to make sure that there can be no such unmatched characters, since it is not good to have a scanner that can "crash" on bad input).

The regular expressions are similar to the ones discussed in the scanner notes. Here's how they are used to match the input:

* Most characters match themselves. For example:
* abc
* ==
* while

are three patterns that match exactly those sequences of characters (note that writing one character after another means "followed by" as usual).

* Characters (even special characters, except backslash followed by a double quote) enclosed in double quotes match themselves. For example, the following patterns are equivalent to the three given above:
* "abc"
* "=="
* "while"

And the following pattern:

"a|b"

matches the three-character sequence: a then | then b, rather than matching a single a or a single b.

* The following characters have the usual special meanings as regular expression operators:

|  |  |
| --- | --- |
| | | means "or" |
| \* | means zero or more instances of |
| + | means one or more instances of |
| ? | means zero or one instance of |
| () | are used for grouping |

* The dot character matches *any* character except the newline character. It is usually only used in the last rule in the specification, to match all "bad" characters (and the associated action issues an error message).
* The backslash is a special *escape* character:

|  |  |
| --- | --- |
| \n | newline |
| \t | tab |
| \" | double quote |

* So to match a backslash character, put it in quotes. (See the [JLex Reference Manual](http://www.cs.princeton.edu/~appel/modern/java/JLex/manual.html) for a complete list of the special characters escaped by a backslash.)
* The up-arrow and dollar-sign characters: ^ and $, are special cahracters. When the up-arrow is used as the first character in a pattern, it causes the pattern to match only at the beginning of a line (i.e., only if the previous character was a newline). When the dollar sign is used as the last character in a pattern, it causes the pattern to match only at the end of a line (i.e., only if the next character is a newline).
* The regular expression can include **character classes**, delimited by square brackets:
  + A character class will match *one* character.
  + If no special characters are used inside the character class, then the character class matches any of the characters it includes inside its square brackets. For example: [abc] matches an a, or a b, or a c, so it is the same as: a|b|c.
  + Here are the characters that are "special" inside a character class:

|  |  |
| --- | --- |
| - | means a range of characters; e.g., a-z means "a to z". |
| ^ | is only a special character if it is the *first* character in the square brackets; it means *not* any of the following characters. So for example, [^abc] matches any character other than an a, or a b, or a c. |
| \ | is used as an escape character with n, b, etc as usual; also can be used to escape the characters that are special inside a character class (e.g., [a\-z] matches an a or a - or a z, and [\\] matches a backslash. |

Note that whitespace only matches itself if it is inside quotes or in a character class; otherwise, it *ends* the current pattern. So the two rules:

[a bc] {}

a|" "|b|c {}

are equivalent; each matches an a, or a space, or a b, or a c, while the rule:

a bc {}

causes an error when you try to process your specification.

**TEST YOURSELF #1**

**Question 1:** The character class [a-zA-Z] matches any letter. Write a character class that matches any letter or any digit.

**Question 2:** Write a pattern that matches any Pascal identifier (a sequence of one or more letters and/or digits, starting with a letter).

**Question 3:** Write a pattern that matches any Java identifier (a sequence of one or more letters and/or digits and/or underscores, starting with a letter or underscore.

**Question 4:** Write a pattern that matches any Java identifier that does *not* end with an underscore.

**JLex directives**

Recall that the second part of a JLex specification contains directives. This can include specifying the value that should be returned on end-of-file, specifying that line counting should be turned on, and specifying that the scanner will be used with the Java parser generator java cup. (See the [JLex Reference Manual](http://www.cs.princeton.edu/~appel/modern/java/JLex/manual.html) for more information about directives.)

The directives part also includes **macro definitions**. The form of a macro definition is:

name = regular-expression

where name is any valid Java identifier, and regular-expression is any regular expression as defined above. Here are some examples:

DIGIT= [0-9]

LETTER= [a-zA-Z]

WHITESPACE= [ \t\n]

Once a macro has been defined, it can be used in a regular expression (either to define another macro, or in the "Regular Expresson Rules" part of the JLex specification. To use a macro, just use its name inside curly braces. For example, given the above macro definitions, the following pattern could be used to match Pascal identifiers:

{LETTER}({LETTER}|{DIGIT})\*

**TEST YOURSELF #2**

Define a macro named NOTSPECIAL that matches any character except a newline, double quote, or backslash.

**Comments**

You can include comments in the first and second parts of your JLex specification, but not in the third part (because JLex will think they are part of a pattern).

**yyline and yytext**

If you turn line counting on (by including %line in the "directives" part of the specification), you can use the variable yyline in the actions that you write for the regular expressions. The value of yyline will be the current line number in the input file, counting from zero (so to use that number in error messages printed by your scanner, you will need to add one to yyline).

You can also use the method yytext() in your actions. This method returns a String -- the sequence of characters that was just matched.

**A Small Example**

Here is a small (complete) JLex specification:

%%

DIGIT= [0-9]

LETTER= [a-zA-Z]

WHITESPACE= [ \t\n] // space, tab, newline

// The next 3 lines are included so that we can use the generated scanner

// with java CUP (the Java parser generator)

%implements java\_cup.runtime.Scanner

%function next\_token

%type java\_cup.runtime.Symbol

// Turn on line counting

%line

%%

{LETTER}({LETTER}|{DIGIT}\*) {System.out.println(yyline+1 + ": ID " + yytext());}

{DIGIT}+ {System.out.println(yyline+1 + ": INT");}

"=" {System.out.println(yyline+1 + ": ASSIGN");}

"==" {System.out.println(yyline+1 + ": EQUALS");}

{WHITESPACE}\* { }

. {System.out.println(yyline+1 + ": bad char");}

**Quick Reference Guide**

**Operators and Special Symbols in JLex**

The following table summarizes the operators and special symbols used in JLex. Note that some characters have an entirely different meaning when used in a regular expression and in a character class. Character classes are always delimited by square brackets; they can be used in the regular expressions that define macros, as well as in the regular expressions used to specify a pattern to be matched in the input.

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Meaning in Regular Expressions** | **Meaning in Character Classes** |
| ( | Matches with ) to group sub-expressions. | Represents itself. |
| ) | Matches with ( to group sub-expressions. | Represents itself. |
| [ | Begins a character class. | Represents itself. |
| ] | Represents itself. | Ends a character class. |
| { | Matches with } to delimit a macro name. | Represents itself. |
| } | Matches with { to delimit a macro name. | Represents itself. |
| " | Matches with " to delimit strings (only \ is special within strings). | Matches with " to delimit a string of characters that belong to the character class.  Only \" is special within the string. |
| \ | Escapes special characters (n, t, etc). Also used to specify a character by its octal, hexadecimal, or unicode value. | Escapes characters that are special inside a character class. |
| . | Matches any one character except newline. | Represents itself. |
| | | Alternation (or) operator. | Represents itself. |
| \* | Kleene closure operator (zero or more matches). | Represents itself. |
| + | Positive closure operator (one or more matches). | Represents itself. |
| ? | Optional choice operator (zero or one matches). | Represents itself. |
| ^ | Matches only at beginning of a line. | When it is the first character in the character class, complements the remaining characters in the class. |
| $ | Matches only at end of a line. | Represents itself. |
| - | Represents itself. | Range of characters operator. |