JLex:
A lexical analyzer generator for Java(TM)

Elliot Berk
Department of Computer Science, Princeton University

Version 1.2, May 5, 1997

Manual revision October 29, 1997

Last updated September 6, 2000 for JLex 1.2.5

(latest version can be obtained from http://www.cs.princeton.edu/~appel/modern/java/JLex/)

Contents

---

1. Introduction

A lexical analyzer breaks an input stream of characters into tokens. Writing lexical analyzers by hand can be a tedious process, so software tools have been developed to ease this task.

Perhaps the best known such utility is Lex. Lex is a lexical analyzer generator for the UNIX operating system, targeted to the C programming language. Lex takes a specially-formatted specification file containing the details of a lexical analyzer. This tool then creates a C source file for the associated table-driven lexer.

The JLex utility is based upon the Lex lexical analyzer generator model. JLex takes a specification file similar to that accepted by Lex, then creates a Java source file for the corresponding lexical analyzer.

2. JLex Specifications

A JLex input file is organized into three sections, separated by double-percent directives (``%%"). A proper JLex specification has the following format.
user code
%%
JLex directives
%%
regular expression rules
The ``%%" directives distinguish sections of the input file and must be placed at the beginning of their line. The remainder of the line containing the ``%%" directives may be discarded and should not be used to house additional declarations or code.

The user code section - the first section of the specification file - is copied directly into the resulting output file. This area of the specification provides space for the implementation of utility classes or return types.

The JLex directives section is the second part of the input file. Here, macros definitions are given and state names are declared.

The third section contains the rules of lexical analysis, each of which consists of three parts: an optional state list, a regular expression, and an action.

2.1 User Code

User code precedes the first double-percent directive (``%%'). This code is copied verbatim into the lexical analyzer source file that JLex outputs, at the top of the file. Therefore, if the lexer source file needs to begin with a package declaration or with the importation of an external class, the user code section should begin with the corresponding declaration. This declaration will then be copied onto the top of the generated source file.

2.2 JLex Directives

The JLex directive section begins after the first ``%%" and continues until the second ``%%" delimiter. Each JLex directive should be contained on a single line and should begin that line.

2.2.1 Internal Code to Lexical Analyzer Class

The *%{...%}* directive allows the user to write Java code to be copied into the lexical analyzer class. This directive is used as follows.
%{
<code>
%}
To be properly recognized, the *%{* and *%}* should each be situated at the beginning of a line. The specified Java code in *<code>* will be then copied into the lexical analyzer class created by JLex.
class Yylex {
... <code> ...
}
This permits the declaration of variables and functions internal to the generated lexical analyzer class. Variable names beginning with *yy* should be avoided, as these are reserved for use by the generated lexical analyzer class.

2.2.2 Initialization Code for Lexical Analyzer Class

The *%init{ ... %init}* directive allows the user to write Java code to be copied into the constructor for the lexical analyzer class.
%init{
<code>
%init}
The *%init{* and *%init}* directives should be situated at the beginning of a line. The specified Java code in *<code>* will be then copied into the lexical analyzer class constructor.
class Yylex {
Yylex () {
... <code> ...
}
}
This directive permits one-time initializations of the lexical analyzer class from inside its constructor. Variable names beginning with *yy* should be avoided, as these are reserved for use by the generated lexical analyzer class.

The code given in the *%init{ ... %init}* directive may potentially throw an exception, or propagate it from another function. To declare this exception, use the *%initthrow{ ... %initthrow}* directive.
%initthrow{
<exception[1]>[, *<exception[2]>*, ...]
%initthrow}
The Java code specified here will be copied into the declaration of the lexical analyzer constructor.
Yylex ()
throws <exception[1]>[, *<exception[2]>*, ...]
{
... <code> ...
}
If the Java code given in the *%init{ ... %init}* directive throws an exception that is not declared, the resulting lexical analyzer source file may not compile successfully.

2.2.3 End-of-File Code for Lexical Analyzer Class

The *%eof{ ... %eof}* directive allows the user to write Java code to be copied into the lexical analyzer class for execution after the end-of-file is reached.
%eof{
<code>
%eof}
The *%eof{* and *%eof}* directives should be situated at the beginning of a line. The specified Java code in *<code>* will be executed at most once, and immediately after the end-of-file is reached for the input file the lexical analyzer class is processing.

The code given in the *%eof{ ... %eof}* directive may potentially throw an exception, or propagate it from another function. To declare this exception, use the *%eofthrow{ ... %eofthrow}* directive.
%eofthrow{
<exception[1]>[, *<exception[2]>*, ...]
%eofthrow}
The Java code specified here will be copied into the declaration of the lexical analyzer function called to clean-up upon reaching end-of-file.
private void yy_do_eof ()
throws <exception[1]>[, *<exception[2]>*, ...]
{
... <code> ...
}
The Java code in <code> that makes up the body of this function will, in part, come from the code given

in the *%eof{ ... %eof}* directive. If this code throws an exception that is not declared using the *%eofthrow { ... %eofthrow}* directive, the resulting lexer may not compile successfully.

2.2.4 Macro Definitions

Macro definitions are given in the JLex directives section of the specification. Each macro definition is contained on a single line and consists of a macro name followed by an equal sign (=), then by its associated definition. The format can therefore be summarized as follows.
<name> = <definition>
Non-newline white space, e.g. blanks and tabs, is optional between the macro name and the equal sign and between the equal sign and the macro definition. Each macro definition should be contained on a single line.

Macro names should be valid identifiers, e.g. sequences of letters, digits, and underscores beginning with a letter or underscore.

Macro definitions should be valid regular expressions, the details of which are described in another section below.

Macro definitions can contain other macro expansions, in the standard
{<name>} format for macros within regular expressions. However, the user should note that these expressions are macros - not functions or nonterminals - so mutually recursive constructs using macros are illegal. Therefore, cycles in macro definitions will have unpredictable results.

2.2.5 State Declarations

Lexical states are used to control when certain regular expressions are matched. These are declared in the JLex directives in the following way.
%state state[0][, *state[1], state[2],* ...]
Each declaration of a series of lexical states should be contained on a single line. Multiple declarations can be included in the same JLex specification, so the declaration of many states can be broken into many declarations over multiple lines.

State names should be valid identifiers, e.g. sequences of letters, digits, and underscores beginning with a letter or underscore.

A single lexical state is implicitly declared by JLex. This state is called *YYINITIAL*, and the generated lexer begins lexical analysis in this state.

Rules of lexical analysis begin with an optional state list. If a state list is given, the lexical rule is matched only when the lexical analyzer is in one of the specified states. If a state list is not given, the lexical rule is matched when the lexical analyzer is in any state.

If a JLex specification does not make use of states, by neither declaring states nor preceding lexical rules with state lists, the resulting lexer will remain in state *YYINITIAL* throughout execution. Since lexical rules are not prefaced by state lists, these rules are matched in all existing states, including the implicitly declared state *YYINITIAL*. Therefore, everything works as expected if states are not used at all.

States are declared as constant integers within the generated lexical analyzer class. The constant integer declared for a declared state has the same name as that state. The user should be careful to avoid name conflict between state names and variables declared in the action portion of rules or elsewhere within the lexical analyzer class. A convenient convention would be to declare state names in all capitals, as a reminder that these identifiers effectively become constants.

2.2.6 Character Counting

Character counting is turned off by default, but can be activated with the *%char* directive.
%char
The zero-based character index of the first character in the matched region of text is then placed in the integer variable *yychar*.

2.2.7 Line Counting

Line counting is turned off by default, but can be activated with the *%line* directive.
%line
The zero-based line index at the beginning of the matched region of text is then placed in the integer variable *yyline*.

2.2.8 Java CUP Compatibility

Java CUP is a parser generator for Java originally written by Scott Hudson of Georgia Tech University, and maintained and extended by Frank Flannery, Dan Wang, and C. Scott Ananian. Details of this software tool are on the World Wide Web at http://www.cs.princeton.edu/~appel/modern/java/CUP/.
Java CUP compatibility is turned off by default, but can be activated with the following JLex directive.
%cup
When given, this directive makes the generated scanner conform to the `java_cup.runtime.Scanner` interface. It has the same effect as the following three directives:
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
See the next section for more details on these three directives, and the CUP manual for more details on using CUP and JLex together.

2.2.9 Lexical Analyzer Component Titles

The following directives can be used to change the name of the generated lexical analyzer class, the tokenizing function, and the token return type. To change the name of the lexical analyzer class from *Yylex*, use the *%class* directive.
%class <name>
To change the name of the tokenizing function from *yylex*, use the *%function* directive.
%function <name>
To change the name of the return type from the tokenizing function from *Yytoken*, use the *%type* directive.
%type <name>
If the default names are not altering using these directives, the tokenizing function is envoked with a call to *Yylex.yylex()*, which returns the *Ytoken* type.

To avoid scoping conflicts, names beginning with *yy* are normally reserved for lexical analyzer internal functions and variables.

2.2.10 Default Token Type

To make the 32-bit primitive integer type *int*, the return type for the tokenizing function (and therefore the token type), use the *%integer* directive.

%integer
Under default settings, *Yytoken* is the return type of the tokenizing function *Yylex.yylex()*, as in the following code fragment.
```
class Yylex { ...
public Yytoken yylex () {
... }
```
The *%integer* directive replaces the previous code with a revised declaration, in which the token type has been changed to *int*.
```
class Yylex { ...
public int yylex () {
... }
```
This declaration allows lexical actions to return integer codes, as in the following code fragment from a hypothetical lexical action.
```
{ ...
return 7;
... }
```

The integer return type forces changes the behavior at end of file. Under default settings, objects - subclasses of the java.lang.Object class - are returned by *Yylex.yylex()*. During execution of the generated lexer *Yylex*, a special object value must be reserved for end-of-file. Therefore, when the end-of-file is reached for the processed input file (and from then onward), *Yylex.yylex()* returns *null*.

When *int* is the return type of *Yylex.yylex()*, *null* can no longer be returned. Instead, *Yylex.yylex()* returns the value -1, corresponding to constant integer *Yylex.YYEOF*. The *%integer* directive implies *%yyeof*; see below.

2.2.11 Default Token Type II: Wrapped Integer

To make java.lang.Integer the return type for the tokenizing function (and therefore the token type), use the *%intwrap* directive.
%intwrap
Under default settings, *Yytoken* is the return type of the tokenizing function *Yylex.yylex()*, as in the following code fragment.
```
class Yylex { ...
public Yytoken yylex () {
... }
```
The *%intwrap* directive replaces the previous code with a revised declaration, in which the token type has been changed to java.lang.Integer.
```
class Yylex { ...
public java.lang.Integer yylex () {
```

... }
This declaration allows lexical actions to return wrapped integer codes, as in the following code fragment from a hypothetical lexical action.
{ ...
return new java.lang.Integer(0);
... }

Notice that the effect of *%intwrap* directive can be equivalently accomplished using the *%type* directive, as follows.
%type java.lang.Integer
This manually changes the name of the return type from *Yylex.yylex()* to *java.lang.Integer*.

2.2.12 YYEOF on End-of-File

The *%yyeof* directive causes the constant integer *Yylex.YYEOF* to be declared. If the *%integer* directive is present, *Yylex.YYEOF* is returned upon end-of-file.
%yyeof
This directive causes *Yylex.YYEOF* to be declared as follows:
public final int YYEOF = -1;
The *%integer* directive implies *%yyeof*.

2.2.13 Newlines and Operating System Compatibility

In UNIX operating systems, the character code sequence representing a newline is the single character ``\n". Conversely, in DOS-based operating systems, the newline is the two-character sequence ``\r\n" consisting of the carriage return followed by the newline. The *%notunix* directive results in either the carriage return or the newline being recognized as a newline.
%notunix
This issue of recognizing the proper sequence of characters as a newline is important in ensuring Java platform independence.

2.2.14 Character Sets

The default settings support an alphabet of character codes between 0 and 127 inclusive. If the generated lexical analyzer receives an input character code that falls outside of these bounds, the lexer may fail.

The *%full* directive can be used to extend this alphabet to include all 8-bit values.
%full
If the *%full* directive is given, JLex will generate a lexical analyzer that supports an alphabet of character codes between 0 and 255 inclusive.

The *%unicode* can be used to extend the alphabet to include the full 16-bit Unicode alphabet.
%unicode
If the *%unicode* directive is given, JLex will generate a lexical analyzer that supports an alphabet of character codes between 0 and 2^16-1 inclusive.

The *%ignorecase* directive can be given to generate case-insensitive lexers.
%ignorecase
If the *%ignorecase* directive is given, CUP will expand all character classes in a unicode-friendly way to match both upper, lower, and title-case letters.

2.2.15 Character Format To and From File

Under the status quo, JLex and the lexical analyzer it generates read from and write to Ascii text files, with byte sized characters. However, to support further extensions on the JLex tool, all internal processing of characters is done using the 16-bit Java character type, although the full range of 16-bit values is not supported.

2.2.16 Exceptions Generated by Lexical Actions

The code given in the action portion of the regular expression rules, in section three of the JLex specification, may potentially throw an exception, or propagate it from another function. To declare these exceptions, use the *%yylexthrow{ ... %yylexthrow}* directive.
%yylexthrow{
<exception[1]>[, *<exception[2]>*, ...]
%yylexthrow}
The Java code specified here will be copied into the declaration of the lexical analyzer tokenizing function *Yylex.yylex()*, as follows.

public Yytoken yylex ()
throws <exception[1]>[*, <exception[2]>, ...*]
{
...
}
If the code given in the action portion of the regular expression rules throws an exception that is not declared using the *%yylexthrow{ ... %yylexthrow}* directive, the resulting lexer may not compile successfully.

2.2.17 Specifying the Return Value on End-of-File

The *%eofval{ ... %eofval}* directive specifies the return value on end-of-file. This directive allows the user to write Java code to be copied into the lexical analyzer tokenizing function *Yylex.yylex()* for execution when the end-of-file is reached. This code must return a value compatible with the type of the tokenizing function *Yylex.yylex()*.
%eofval{
<code>
%eofval}
The specified Java code in *<code>* determines the return value of *Yylex.yylex()* when the end-of-file is reached for the input file the lexical analyzer class is processing. This will also be the value returned by *Yylex.yylex()* each additional time this function is called after end-of-file is initially reached, so *<code>* may be executed more than once. Finally, the *%eofval{* and *%eofval}* directives should be situated at the beginning of a line.

An example of usage is given below. Suppose the return value desired on end-of-file is *(new token(sym.EOF))* rather than the default value *null*. The user adds the following declaration to the specification file.
%eofval{
return (new token(sym.EOF));
%eofval}
The code is then copied into *Yylex.yylex()* into the appropriate place.
public Yytoken yylex () { ...
return (new token(sym.EOF));
... }
The value returned by *Yylex.yylex()* upon end-of-file and from that point onward is now *(new token(sym.EOF))*.

2.2.18 Specifying an interface to implement

JLex allows the user to specify an interface which the *Yylex* class will implement. By adding the following declaration to the input file:
%implements <classname>
the user specifies that Yylex will implement *classname*. The generated parser class declaration will look like:
`class Yylex implements classname { ...`

2.2.19 Making the Generated Class Public

The *%public* directive causes the lexical analyzer class generated by JLex to be a public class.
%public
The default behavior adds no access specifier to the generated class, resulting in the class being visible only from the current package.

2.3 Regular Expression Rules

The third part of the JLex specification consists of a series of rules for breaking the input stream into tokens. These rules specify regular expressions, then associate these expressions with actions consisting of Java source code.

The rules have three distinct parts: the optional state list, the regular expression, and the associated action. This format is represented as follows.
[*<states>*] *<expression> { <action> }*
Each part of the rule is discussed in a section below.

If more than one rule matches strings from its input, the generated lexer resolves conflicts between rules by greedily choosing the rule that matches the longest string. If more than one rule matches strings of the same length, the lexer will choose the rule that is given first in the JLex specification. Therefore, rules appearing earlier in the specification are given a higher priority by the generated lexer.

The rules given in a JLex specification should match all possible input. If the generated lexical analyzer receives input that does not match any of its rules, an error will be raised.

Therefore, all input should be matched by at least one rule. This can be guaranteed by placing the following rule at the bottom of a JLex specification:
. { java.lang.System.out.println("Unmatched input: " + yytext()); }
The dot (.), as described below, will match any input except for the newline.

2.3.1 Lexical States

An optional lexical state list preceeds each rule. This list should be in the following form:
<state[0][, *state[1], state[2], ...*]>
The outer set of brackets ([]) indicate that multiple states are optional. The greater than (<) and less than (>) symbols represent themselves and should surround the state list, preceding the regular expression. The state list specifies under which initial states the rule can be matched.

For instance, if *yylex()* is called with the lexer at state *A*, the lexer will attempt to match the input only against those rules that have *A* in their state list.

If no state list is specified for a given rule, the rule is matched against in all lexical states.

2.3.2 Regular Expressions

Regular expressions should not contain any white space, as white space is interpreted as the end of the current regular expression. There is one exception; if (non-newline) white space characters appear from within double quotes, these characters are taken to represent themselves. For instance, `` " " is interpreted as a blank space.

The alphabet for JLex is the Ascii character set, meaning character codes between 0 and 127 inclusive.

The following characters are metacharacters, with special meanings in JLex regular expressions.

**? * + | ( ) ^ $. [ ] { } " **

Otherwise, individual characters stand for themselves.

ef Consecutive regular expressions represents their concatenation.

e|*f* The vertical bar (|) represents an option between the regular expressions that surround it, so matches either expression *e* or *f*.

The following escape sequences are recognized and expanded:

\b	Backspace
\n	newline
\t	Tab
\f	Formfeed
\r	Carriage return
ddd	The character code corresponding to the number formed by three octal digits *ddd*
\x*dd*	The character code corresponding to the number formed by two hexadecimal digits *dd*
\u*dddd*	The Unicode character code corresponding to the number formed by four hexidecimal digits *dddd*.
\^*C*	Control character
c	A backslash followed by any other character *c* matches itself

$ The dollar sign ($) denotes the end of a line. If the dollar sign ends a regular expression, the expression is matched only at the end of a line.

. The dot (.) matches any character except the newline, so this expression is equivalent to [^\n].

"..." Metacharacters lose their meaning within double quotes and represent themselves. The sequence \" (which represents the single character ") is the only exception.

{name} Curly braces denote a macro expansion, with *name* the declared name of the associated macro.

* The star (*) represents Kleene closure and matches zero or more repetitions of the preceding regular expression.

+ The plus (+) matches one or more repetitions of the preceding regular expression, so *e*+ is equivalent to *ee**.

? The question mark (?) matches zero or one repetitions of the preceding regular expression.

(...) Parentheses are used for grouping within regular expressions.

[...] Square backets denote a class of characters and match any one character enclosed in the backets. If the first character following the left bracket ([) is the up arrow (^), the set is negated and the expression matches any character except those enclosed in the backets. Different metacharacter rules hold inside the

backets, with the following expressions having special meanings:

{name}	Macro expansion
a - *b*	Range of character codes from *a* to *b* to be included in character set
"..."	All metacharacters within double quotes lose their special meanings. The sequence \" (which represents the single character ") is the only exception.
\	Metacharacter following backslash(\) loses its special meaning

For example, [a-z] matches any lower-case letter, [^0-9] matches anything except a digit, and [0-9a-fA-F] matches any hexadecimal digit. Inside character class brackets, a metacharacter following a backslash loses its special meaning. Therefore, [\-\\] matches a dash or a backslash. Likewise ["A-Z"] matches one of the three characters A, dash, or Z. Leading and trailing dashes in a character class also lose their special meanings, so [+-] and [-+] do what you would expect them to (ie, match only '+' and '-').

2.3.3 Associated Actions

The action associated with a lexical rule consists of Java code enclosed inside block-delimiting curly braces.
{ action }
The Java code *action* is copied, as given, into the state-driven lexical analyzer produced by JLex.

All curly braces contained in *action* not part of strings or comments should be balanced.

2.3.3.1 Actions and Recursion:

If no return value is returned in an action, the lexical analyzer will loop, searching for the next match from the input stream and returning the value associated with that match.

The lexical analyzer can be made to recur explicitly with a call to *yylex()*, as in the following code fragment.
{ ...
return yylex();
... }
This code fragment causes the lexical analyzer to recur, searching for the next match in the input and returning the value associated with that match. The same effect can be had, however, by simply not returning from a given action. This results in the lexer searching for the next match, without the additional overhead of recursion.

The preceding code fragment is an example of tail recursion, since the recursive call comes at the end of the calling function's execution. The following code fragment is an example of a recursive call that is not tail recursive.

```
{ ...
next = yylex();
... }
```

Recursive actions that are not tail-recursive work in the expected way, except that variables such as *yyline* and *yychar* may be changed during recursion.

2.3.3.2 State Transitions:

If lexical states are declared in the JLex directives section, transitions on these states can be declared within the regular expression actions. State transitions are made by the following function call.

```
yybegin(state);
```

The void function *yybegin()* is passed the state name *state* and effects a transition to this lexical state.

The state *state* must be declared within the JLex directives section, or this call will result in a compiler error in the generated source file. The one exception to this declaration requirement is state *YYINITIAL*, the lexical state implicitly declared by JLex. The generated lexer begins lexical analysis in state *YYINITIAL* and remains in this state until a transition is made.

2.3.3.3 Available Lexical Values:

The following values, internal to the *Yylex* class, are available within the action portion of the lexical rules.

Variable or Method	ActivationDirective	Description
java.lang.String yytext();	Always active.	Matched portion of the character input stream.
int yychar;	*%char*	Zero-based character index of the first character in the matched portion of the input stream
int yyline;	*%line*	Zero-based line number of the start of the matched portion of the input stream

---

3. Generated Lexical Analyzers

JLex will take a properly-formed specification and transform it into a Java source file for the corresponding lexical analyzer.

The generated lexical analayzer resides in the class *Yylex*. There are two constructors to this class, both requiring a single argument: the input stream to be tokenized. The input stream may either be of type `java.io.InputStream` or `java.io.Reader` (such as `StringReader`). Note that the `java.io.Reader` constructor should be used if you are generating a lexer accepting unicode characters, as the JDK 1.0 `java.io.InputStream` class does not always read unicode correctly.

The access function to the lexer is *Yylex.yylex()*, which returns the next token from the input stream. The return type is *Yytoken* and the function is declared as follows.
class Yylex { ...
public Yytoken yylex () {
... }
The user must declare the type of *Yytoken* and can accomplish this conveniently in the first section of the JLex specification, the user code section. For instance, to make *Yylex.yylex()* return a wrapper around integers, the user would enter the following code somewhere preceding the first ``%%".
class Yytoken { int field; Yytoken(int f) { field=f; } }
Then, in the lexical actions, wrapped integers would be returned, in something like this way.
{ ...
return new Yytoken(0);
... }
Likewise, in the user code section, a class could be defined declaring constants that correspond to each of the token types.
class TokenCodes { ...
public static final STRING = 0;
public static final INTEGER = 1;
... }
Then, in the lexical actions, these token codes could be returned.
{ ...
return new Yytoken(STRING);
... }
These are simplified examples; in actual use, one would probably define a token class containing more information than an integer code.

These examples begin to illustrate the object-oriented techniques a user could employ to define an arbitrarily complex token type to be returned by *Yylex.yylex()*. In particular, inheritance permits the user to return more than one token type. If a distinct token type was needed for strings and integers, the user could make the following declarations.

class Yytoken { ... }
class IntegerToken extends Yytoken { ... }
class StringToken extends Yytoken { ... }
Then the user could return both *IntegerToken* and *StringToken* types from the lexical actions.

The names of the lexical analyzer class, the tokening function, and its return type each may be altered using the JLex directives. See the section 2.2.9 for more details.

---

4. Performance

A benchmark experiment was conducted, comparing the performance of a lexical analyzer generated by JLex to that of a hand-written lexical analyzer. The comparison was made for lexical analyzers of a simple ``toy'' programming language. The hand-written lexical analyzer, like the lexical analyzer generated by JLex, was written in Java.

The experiment consists of running each lexical analyzer on two source files written in the toy language, then measuring the time required to process these files. Each lexical analyzer was invoked by a dummy driver also written in Java.

The generated lexical analyzer proved to be quite quick, as the following results show.

Size of Source File	JLex-Generated Lexical Analyzer: Execution Time	Hand-Written Lexical Analyzer: Execution Times
177 lines	0.42 seconds	0.53 seconds
897 lines	0.98 seconds	1.28 seconds

The JLex lexical analyzer soundly outperformed the hand-written lexer.

One of the biggest complaints about table-driven lexical analyzers generated by programs like JLex is that these lexical analyzers do not perform as well as hand-written ones. Therefore, this experiment is particularly important in demonstrating the relative speed of JLex lexical analyzers.

---

5. Implementation Issues

5.1 Unimplemented Features

The following is a (possibly incomplete) list of unimplemented features of JLex.

1. The regular expression lookahead operator is unimplemented, and not included in the list of special regular expression metacharacters.
2. The start-of-line operator (^) assumes the following nonstandard behavior. A match on a regular expression that uses this operator will cause the newline that precedes the match to be discarded.

5.2 Unicode vs Ascii

In contrast to the 8-bit character type (char) mandated by Ansi C, Java supports a 16-bit char and the Unicode character set. Java provides a built-in String class to manipulate these Unicode characters.

As of version 1.2.5, JLex uses the JDK 1.1 `Reader` and `Writer` classes to read in the JLex specification file and write out the lexical analyzer source file. This means that all unicode characters are allowed in both of these. In order for the generated scanner to work with unicode characters, you must use the `java.io.Reader` constructor of the generated scanner, and the `Reader` you provide must properly handle the translation from OS-native format to unicode. You must also specify the *%unicode* directive in the specification; see section 2.2.14.

5.3 Commas in State Lists

Commas between state names in declaration lists and lexical rules are optional. These lists will be correctly parsed with white space between state names and without comma separators.

5.4 Wish List of Unimplemented Features

The following minor features would be nice to have as part of JLex, but have not been implemented due to their scope or their negative impact upon performance.

1. Detection of unbalanced braces within the comment portion of lexical actions.
2. Detection of cycles in macro definitions.

---

6. Credits and Copyrights

6.1 Credits

The treatment of lexical analyzer generators given in Alan Holub's *Compiler Design in C* (Prentice-Hall, 1990) provided a starting point for my implementation.

Discussions with Professor Andrew Appel of the Princeton University Computer Science Department provided guidance in the design of JLex.

Java is a trademark of Sun Microsystems Incorporated.

6.2 Copyright

merchantability and fitness. In no event shall Elliot Joel Berk be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

Frank Flannery
Wed Jul 24 00:27:39 EDT 1996