Character Classes

Our specification of the reserved word if, as shown earlier, is incomplete. We don't (yet) handle upper or mixed-case.

To extend our definition, we'll use a very useful feature of Lex and JLex—*character classes*.

Characters often naturally fall into classes, with all characters in a class treated identically in a token definition. In our definition of identifiers all letters form a class since any of them can be used to form an identifier. Similarly, in a number, any of the ten digit characters can be used.

Character classes are delimited by **`[`** and **`]`**; individual characters are listed without any quotation or separators. However **`\`**, **`^`**, **`]`** and **`-`**, because of their special meaning in character classes, must be escaped. The character class **`[xyz]`** can match a single **`x`**, **`y`**, or **`z`**.

The character class **`[\])]`** can match a single **`]`** or **`)`**.

(The **`]`** is escaped so that it isn't misinterpreted as the end of character class.)

Ranges of characters are separated by a **`-`**; **`[x-z]`** is the same as **`[xyz]`**. **`[0-9]`** is the set of all digits and **`[a-zA-Z]`** is the set of all letters, upper- and lower-case. **`\`** is the escape character, used to represent

unprintables and to escape special symbols.

Following C and Java conventions, **`\n`** is the newline (that is, end of line), **`\t`** is the tab character, **`\\`** is the backslash symbol itself, and **`\010`** is the character corresponding to octal 10.

The **`^`** symbol complements a character class (it is JLex's representation of the **Not** operation).

`[^xy]` is the character class that matches any single character *except* **`x`** and **`y`**. The **`^`** symbol applies to all characters that follow it in a character class definition, so **`[^0-9]`** is the set of all characters that aren't digits. **`[^]`** can be used to match all characters.

Here are some examples of character classes:

Character Class	Set of Characters Denoted
`[abc]`	Three characters: a, b and c
`[cba]`	Three characters: a, b and c
`[a-c]`	Three characters: a, b and c
`[aabbcc]`	Three characters: a, b and c
`[^abc]`	All characters except a, b and c
`[\^\-\]]`	Three characters: ^, - and]
`[^]`	All characters
`"[abc]"`	Not a character class. This is one five character *string*: `[abc]`

Regular Operators in JLex

JLex provides the standard regular operators, plus some additions.

- Catenation is specified by the juxtaposition of two expressions; no explicit operator is used.
  Outside of character class brackets, individual letters and numbers match themselves; other characters should be quoted (to avoid misinterpretation as regular expression operators).

Regular Expr	**Characters Matched**
`a b cd`	Four characters: `abcd`
`(a)(b)(cd)`	Four characters: `abcd`
`[ab][cd]`	Four different strings: `ac` or `ad` or `bc` or `bd`
`while`	Five characters: `while`
`"while"`	Five characters: `while`
`[w][h][i][l][e]`	Five characters: `while`

Case *is* significant.

- The alternation operator is **|**. Parentheses can be used to control grouping of subexpressions.
  If we wish to match the reserved word **while** allowing any mixture of upper- and lowercase, we can use
  (w|W)(h|H)(i|I)(l|L)(e|E)
  or
  [wW][hH][iI][lL][eE]

Regular Expr	Characters Matched
ab|cd	Two different strings: ab or cd
(ab)|(cd)	Two different strings: ab or cd
[ab]|[cd]	Four different strings: a or b or c or d

- Postfix operators:

  ***** Kleene closure: 0 or more matches

  (ab)* matches λ or **ab** or **abab** or **ababab** ...

  + Positive closure: 1 or more matches

  (ab)+ matches **ab** or **abab** or **ababab** ...

  ? Optional inclusion:

  expr?

  matches **expr** zero times or once.

  expr? is equivalent to **(expr) |** λ and eliminates the need for an explicit λ symbol.

  [-+]?[0-9]+ defines an optionally signed integer literal.

- Single match:
  The character "**.**" matches any single character (other than a newline).
- Start of line:
  The character **^** (when used outside a character class) matches the beginning of a line.
- End of line:
  The character **$** matches the end of a line. Thus,

  ^A.*e$

  matches an entire line that begins with **A** and ends with **e**.

Overlapping Definitions

Regular expressions map overlap (match the same input sequence).

In the case of overlap, two rules determine which regular expression is matched:

- The *longest possible* match is performed. JLex automatically buffers characters while deciding how many characters can be matched.
- If two expressions match *exactly* the same string, the earlier expression (in the JLex specification) is preferred. Reserved words, for example, are often special cases of the pattern used for identifiers. Their definitions are therefore placed before the

expression that defines an identifier token.

Often a “catch all” pattern is placed at the very end of the regular expression rules. It is used to catch characters that don’t match any of the earlier patterns and hence are probably erroneous. Recall that "." matches any single character (other than a newline). It is useful in a catch-all pattern. However, avoid a pattern like .* which will consume all characters up to the next newline.

In JLex an unmatched character will cause a run-time error.

The operators and special symbols most commonly used in JLex are summarized below. Note that a symbol sometimes has one meaning in a regular expression and an *entirely different* meaning in a character class (i.e., within a pair of brackets). If you find JLex behaving unexpectedly, it's a good idea to check this table to be sure of how the operators and symbols you've used behave. Ordinary letters and digits, and symbols not mentioned (like @**)** represent themselves. If you're not sure if a character is special or not, you can always escape it or make it part of a quoted string.

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 signal macro-expansion.	Represents itself.
}	Matches with { to signal macro-expansion.	Represents itself.
"	Matches with " to delimit strings (only \ is special within strings).	Represents itself.
\	Escapes individual characters. Also used to specify a character by its octal code.	Escapes individual characters. Also used to specify a character by its octal code.
.	Matches any one character except \n.	Represents itself.

Symbol	Meaning in Regular Expressions	Meaning in Character Classes
|	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 (one or zero matches).	Represents itself.
/	Context sensitive matching operator.	Represents itself.
^	Matches only at beginning of a line.	Complements remaining characters in the class.
$	Matches only at end of a line.	Represents itself.
-	Represents itself.	Range of characters operator.

Potential Problems in Using JLex

The following differences from "standard" Lex notation appear in JLex:

- Escaped characters within quoted strings are not recognized. Hence **`"\n"`** is *not* a new line character. Escaped characters outside of quoted strings (**`\n`**) and escaped characters within character classes (**`[\n]`**) are OK.
- A blank should not be used within a character class (i.e., **`[`** and **`]`**). You may use **`\040`** (which is the character code for a blank).

- A doublequote must be escaped within a character class. Use **`[\"]`** instead of **`["]`**.
- Unprintables are defined to be all characters before blank as well as the last ASCII character. These can be represented as: **`[\000-\037\177]`**

JLex Examples

A JLex scanner that looks for five letter words that begin with “P” and end with “T”.

This example is in
`~cs536-1/public/jlex`

The JLex specification file is:

```
class Token {
   String text;
   Token(String t){text = t;}
}
%%
Digit=[0-9]
AnyLet=[A-Za-z]
Others=[0-9'&.]
WhiteSp=[\040\n]
// Tell JLex to have yylex() return a
Token
%type Token
// Tell JLex what to return when eof of
file is hit
%eofval{
return new Token(null);
%eofval}
%%
[Pp]{AnyLet}{AnyLet}{AnyLet}[Tt]{WhiteSp}+
     {return new Token(yytext());}

({AnyLet}|{Others})+{WhiteSp}+
     {/*skip*/}
```

The Java program that uses the scanner is:

```
import java.io.*;

class Main {

public static void main(String args[])
   throws java.io.IOException {

 Yylex lex  = new Yylex(System.in);
 Token token = lex.yylex();

 while ( token.text != null ) {
   System.out.print("\t"+token.text);
   token = lex.yylex(); //get next token
 }
}}
```

In case you care, the words that are matched include:

Pabst
paint
petit
pilot
pivot
plant
pleat
point
posit
Pratt
print

An example of CSX token specifications. This example is in **`~cs536-1/public/proj2/startup`**

The JLex specification file is:

```
import java_cup.runtime.*;

/*  Expand this into your solution for
project 2 */

class CSXToken {
 int linenum;
 int colnum;
 CSXToken(int line,int col){
 linenum=line;colnum=col;};
}

class CSXIntLitToken extends CSXToken {
 int intValue;
 CSXIntLitToken(int val,int line,
   int col){
   super(line,col);intValue=val;};
}

class CSXIdentifierToken extends
CSXToken {
String identifierText;
CSXIdentifierToken(String text,int line,
  int col){
  super(line,col);identifierText=text;};
}
```

```
class CSXCharLitToken extends CSXToken {
  char charValue;
CSXCharLitToken(char val,int line,
    int col){
   super(line,col);charValue=val;};
}

class CSXStringLitToken extends CSXToken
{
   String stringText;
   CSXStringLitToken(String text,
     int line,int col){
   super(line,col);
   stringText=text; };
}

// This class is used to track line and
column numbers
// Feel free to change to extend it
class Pos {
static int  linenum = 1;
/* maintain this as line number current
   token was scanned on */
static int  colnum = 1;
  /* maintain this as column number
    current token began at */
static int  line = 1;
/* maintain this as line number after
   scanning current token  */
```

```
static int  col = 1;
  /* maintain this as column number
    after scanning current token  */
static void setpos() {
 //set starting pos for current token
   linenum = line;
   colnum = col;}
}

%%
Digit=[0-9]

// Tell JLex to have yylex() return a
 Symbol, as JavaCUP will require
%type Symbol

// Tell JLex what to return when eof of
file is hit
%eofval{
return new Symbol(sym.EOF,
          new  CSXToken(0,0));
%eofval}

%%
"+"   {Pos.setpos(); Pos.col +=1;
        return new Symbol(sym.PLUS,
          new CSXToken(Pos.linenum,
                       Pos.colnum));}
```

```
"!="    {Pos.setpos(); Pos.col +=2;
         return new Symbol(sym.NOTEQ,
           new CSXToken(Pos.linenum,
                        Pos.colnum));}
";"     {Pos.setpos(); Pos.col +=1;
           return new Symbol(sym.SEMI,
             new CSXToken(Pos.linenum,
                          Pos.colnum));}
{Digit}+   {// This def doesn't check
            // for overflow
           Pos.setpos();
           Pos.col += yytext().length();
           return new Symbol(sym.INTLIT,
            new CSXIntLitToken(
       new Integer(yytext()).intValue(),
       Pos.linenum,Pos.colnum));}

\n      {Pos.line +=1; Pos.col = 1;}
" "     {Pos.col +=1;}
```

The Java program that uses this scanner (P2) is:

```
class P2 {
 public static void main(String args[])
    throws java.io.IOException {
  if (args.length != 1) {
    System.out.println(
    "Error: Input file must be named on
command line." );
    System.exit(-1);
  }
  java.io.FileInputStream yyin = null;
  try {
   yyin =
   new java.io.FileInputStream(args[0]);
  } catch (FileNotFoundException
           notFound){
    System.out.println(
   "Error: unable to open input file.");
    System.exit(-1);
  }

 // lex is a JLex-generated scanner that
 // will read from yyin
    Yylex lex = new Yylex(yyin);
```

```
    System.out.println(
       "Begin test of CSX scanner.");

  /************************************
   You should enter code here that
   thoroughly test your scanner.

   Be sure to test extreme cases,
   like very long symbols or lines,
   illegal tokens, unrepresentable
   integers, illegals strings, etc.
   The following is only a starting point.
  ************************************/
  Symbol token = lex.yylex();

  while ( token.sym != sym.EOF ) {
   System.out.print(
     ((CSXToken) token.value).linenum
     + ":"
     + ((CSXToken) token.value).colnum
     + " ");

    switch (token.sym) {
     case sym.INTLIT:
      System.out.println(
       "\tinteger literal(" +
       ((CSXIntLitToken)
       token.value).intValue + ")");
      break;
```

```
   case sym.PLUS:
     System.out.println("\t+");
     break;

   case sym.NOTEQ:
     System.out.println("\t!=");
     break;

   default:
     throw new RuntimeException();
  }

  token = lex.yylex(); // get next token
 }

 System.out.println(
    "End test of CSX scanner.");
 }}}
```