# 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 may 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<br>(only \ is special within strings). | Represents itself. |
| \ | Escapes individual characters.<br>Also used to specify a character by its octal code. | Escapes individual characters.<br>Also used to specify a character by its octal code. |
| . | Matches any one character except \n. | Represents itself. |
| \| | Alternation (or) operator. | Represents itself. |

| Symbol | Meaning in Regular Expressions | Meaning in Character Classes |
|---|---|---|
| * | 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. Unprintables 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

**`www.cs.wisc.edu/~fischer/`**
**`cs536.s15/course/proj2/`**
**`startup/Jlex_test/`**

## 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

**`www.cs.wisc.edu/~fischer/cs536.s15/course/proj2/startup/java`**

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.");
}}}
```

# Other Scanner Issues

We will consider other practical issues in building real scanners for real programming languages.

Our finite automaton model sometimes needs to be augmented. Moreover, error handling must be incorporated into any practical scanner.

# Identifiers vs. Reserved Words

Most programming languages contain *reserved words* like **`if`**, **`while`**, **`switch`**, etc. These tokens look like ordinary identifiers, but aren’t.

It is up to the scanner to decide if what looks like an identifier is really a reserved word. This distinction is vital as reserved words have different token codes than identifiers and are parsed differently.

How can a scanner decide which tokens are identifiers and which are reserved words?

- We can scan identifiers and reserved words using the same pattern, and then look up the token in a special “reserved word” table.
- It is known that any regular expression may be *complemented* to obtain all strings not in the original regular expression. Thus $\overline{A}$, the complement of A, is regular if A is. Using complementation we can write a regular expression for nonreserved

  identifiers: $(\overline{\overline{\text{ident}}|\text{if}|\text{while}|\ldots)}$
  Since scanner generators don’t usually support complementation of regular expressions, this approach is more of theoretical than practical interest.

- We can give distinct regular expression definitions for each reserved word, and for identifiers. Since the definitions overlap (**`if`** will match a reserved word *and* the general identifier pattern), we give *priority* to reserved words. Thus a token is scanned as an identifier if it matches the identifier pattern *and* does not match any reserved word pattern. This approach is commonly used in scanner generators like Lex and JLex.

# Converting Token Values

For some tokens, we may need to convert from string form into numeric or binary form.

For example, for integers, we need to transform a string a digits into the internal (binary) form of integers.

We know the format of the token is valid (the scanner checked this), but:

- The string may represent an integer too large to represent in 32 or 64 bit form.
- Languages like CSX and ML use a non- standard representation for negative values (**~123** instead of **-123**)

We can safely convert from string to integer form by first converting the string to double form, checking against max and min int, and then converting to int form if the value is representable.

Thus **`d`** = **`new Double(str)`** will create an object **`d`** containing the value of **`str`** in double form. If **`str`** is too large or too small to be represented as a double, plus or minus infinity is automatically substituted.

**`d.doubleValue()`** will give **`d`**’s value as a Java double, which can be compared against **`Integer.MAX_VALUE`** or **`Integer.MIN_VALUE`**.

If **`d.doubleValue()`** represents a valid integer,
**`(int) d.doubleValue()`**
will create the appropriate integer value.

If a string representation of an integer begins with a “~” we can strip the “~”, convert to a double and then negate the resulting value.

# Scanner Termination

A scanner reads input characters and partitions them into tokens.

What happens when the end of the input file is reached? It may be useful to create an **`Eof`** pseudo-character when this occurs. In Java, for example, **`InputStream.read()`**, which reads a single byte, returns - 1 when end of file is reached. A constant, **`EOF`**, defined as - 1 can be treated as an “extended” ASCII character. This character then allows the definition of an **`Eof`** token that can be passed back to the parser.

An **`Eof`** token is useful because it allows the parser to verify that the logical end of a program corresponds to its physical end.

Most parsers *require* an end of file token.

Lex and Jlex automatically create an **`Eof`** token when the scanner they build tries to scan an **`EOF`** character (or tries to scan when **`eof()`** is true).

# Multi Character Lookahead

We may allow finite automata to look beyond the next input character.

This feature is necessary to implement a scanner for FORTRAN.

In FORTRAN, the statement

```
 DO 10 J = 1,100
```

specifies a loop, with index **J** ranging from 1 to 100.
The statement

```
 DO 10 J = 1.100
```

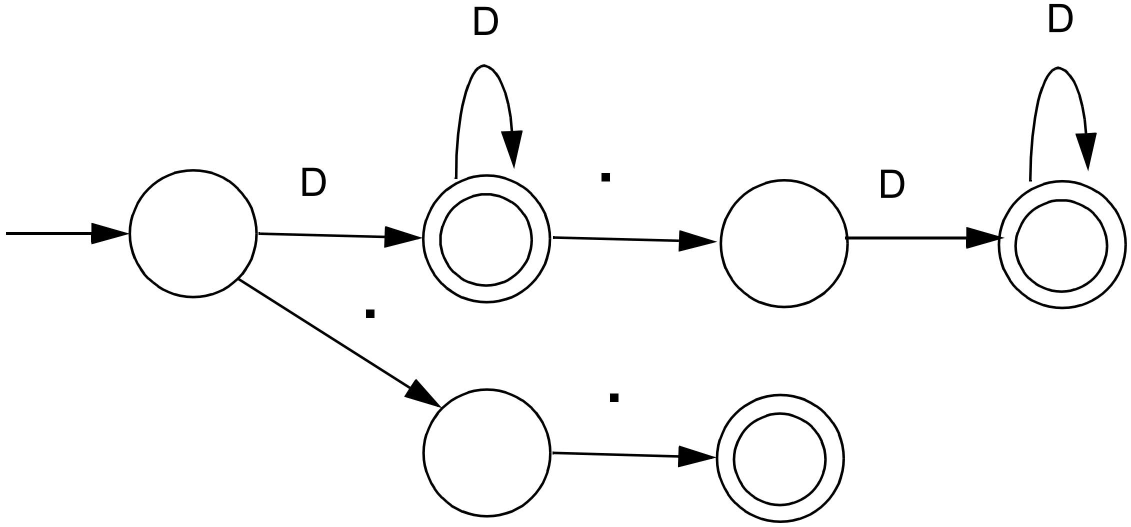
is an assignment to the variable **DO10J**. (Blanks are not significant except in strings.)

A FORTRAN scanner decides whether the **O** is the last character of a **DO** token only after reading as far as the comma (or period).

A milder form of extended lookahead problem occurs in Pascal and Ada.
The token **10.50** is a real literal, whereas **10..50** is three different tokens.

We need two- character lookahead after the **10** prefix to decide whether we are to return **10** (an integer literal) or **10.50** (a real literal).

Suppose we use the following FA.



Given **10..100** we scan three characters and stop in a non-accepting state.

Whenever we stop reading in a non- accepting state, we *back up* along accepted characters until an accepting state is found.

Characters we back up over are *rescanned* to form later tokens. If no accepting state is reached during backup, we have a lexical error.

# Performance Considerations

Because scanners do so much character- level processing, they can be a real performance bottleneck in production compilers.

Speed is not a concern in our project, but let’s see why scanning speed can be a concern in production compilers.

Let’s assume we want to compile at a rate of 5000 lines/sec. (so that most programs compile in just a few seconds).

Assuming 30 characters/line (on average), we need to scan 150,000 char/sec.

A key to efficient scanning is to group character- level operations whenever possible. It is better to do one operation on n characters rather than n operations on single characters.

In our examples we’ve read input one character as a time. A subroutine call can cost hundreds or thousands of instructions to execute—far too much to spend on a single character.

We prefer routines that do block reads, putting an entire block of characters directly into a buffer.

Specialized scanner generators can produce particularly fast scanners.

The *GLA scanner generator* claims that the scanners it produces run as fast as:

```
 while(c != Eof) {
    c = getchar();
}
```