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.");
}}}
```

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 $\overline{\mathbf{A}}$, 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.

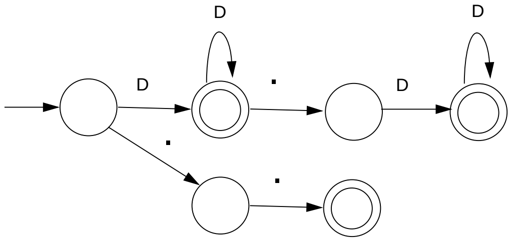
In FORTRAN, the statement
`DO 10 J = 1,100`
specifies a loop, with index **`J`** ranging from **`1`** to **`100`**.
The statement
`DO 10 J = 1.100`
is an assignment to the variable **`DO10J`**. (Blanks are not significant except in strings.)

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.