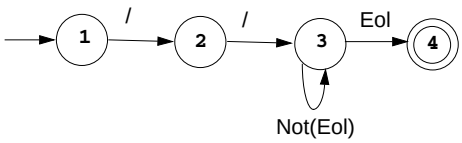


The corresponding transition table is:

State	Character				
	/	Eol	a	b	...
1	2				
2	3				
3	3	4	3	3	3
4					

A complete transition table contains one column for each character. To save space, *table compression* may be used. Only non-error entries are explicitly represented in the table, using hashing, indirection or linked structures.

All regular expressions can be translated into DFAs that accept (as valid tokens) the strings defined by the regular expressions. This translation can be done manually by a programmer or automatically using a scanner generator.

A DFA can be coded in:

- Table-driven form
- Explicit control form

In the table-driven form, the transition table that defines a DFA's actions is explicitly represented in a run-time table that is "interpreted" by a driver program.

In the direct control form, the transition table that defines a DFA's actions appears implicitly as the control logic of the program.

For example, suppose **`CurrentChar`** is the current input character. End of file is represented by a special character value, **`eof`**. Using the DFA for the Java comments shown earlier, a table-driven scanner is:

```
State = StartState
while (true){
  if (CurrentChar == eof)
      break
  NextState =
      T[State][CurrentChar]
  if(NextState == error)
      break
  State = NextState
  read(CurrentChar)
}
if (State in AcceptingStates)
     // Process valid token
else // Signal a lexical error
```


This form of scanner is produced by a scanner generator; it is definition-independent. The scanner is a driver that can scan *any* token if T contains the appropriate transition table.

Here is an explicit-control scanner for the same comment definition:

```
if (CurrentChar == '/'){
   read(CurrentChar)
   if (CurrentChar == '/')
     repeat
       read(CurrentChar)
     until (CurrentChar in
            {eol, eof})
   else //Signal lexical error
else // Signal lexical error
if (CurrentChar == eol)
  // Process valid token
else //Signal lexical error
```

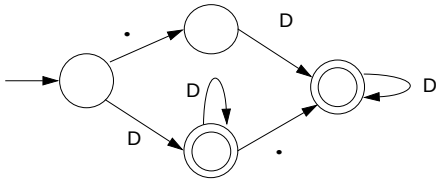
The token being scanned is "hardwired" into the logic of the code. The scanner is usually easy to read and often is more efficient, but is specific to a single token definition.

More Examples

- A FORTRAN-like real literal (which requires digits on either or both sides of a decimal point, or just a string of digits) can be defined as

  RealLit = $(D^+ (\lambda \mid .)) \mid (D^* . D^+)$

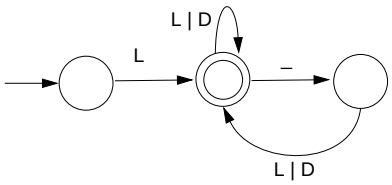
  This corresponds to the DFA



- An identifier consisting of letters, digits, and underscores, which begins with a letter and allows no adjacent or trailing underscores, may be defined as

  ID = L $(L \mid D)^*$ (_ $(L \mid D)^+)^*$

  This definition includes identifiers like `sum` or `unit_cost`, but excludes `_one` and `two_` and `grand___total`. The DFA is:



Lex/Flex/JLex

Lex is a well-known Unix scanner generator. It builds a scanner, in C, from a set of regular expressions that define the tokens to be scanned.

Flex is a newer and faster version of Lex.

JLex is a Java version of Lex. It generates a scanner coded in Java, though its regular expression definitions are very close to those used by Lex and Flex.

Lex, Flex and JLex are largely *non-procedural*. You don't need to tell the tools *how* to scan. All you need to tell it *what* you want scanned (by giving it definitions of valid tokens).

This approach greatly simplifies building a scanner, since most of the details of scanning (I/O, buffering, character matching, etc.) are automatically handled.

JLex

JLex is coded in Java. To use it, you enter

`java JLex.Main f.jlex`

Your **`CLASSPATH`** should be set to search the directories where JLex's classes are stored.
(The **`CLASSPATH`** we gave you includes JLex's classes).

After JLex runs (assuming there are no errors in your token specifications), the Java source file
`f.jlex.java` is created. (**`f`** stands for any file name you choose. Thus **`csx.jlex`** might hold token definitions for CSX, and **`csx.jlex.java`** would hold the generated scanner).

You compile **`f.jlex.java`** just like any Java program, using your favorite Java compiler.

After compilation, the class file

`Yylex.class` is created.

It contains the methods:

- **`Token yylex()`** which is the actual scanner. The constructor for **`Yylex`** takes the file you want scanned, so **`new Yylex(System.in)`** will build a scanner that reads from **`System.in`**. **`Token`** is the token class you want returned by the scanner; you can tell JLex what class you want returned.
- **`String yytext()`** returns the character text matched by the last call to **`yylex`**.

A simple example of the use of JLex is in
`~cs536-1/pubic/jlex`
Just enter
`make test`

Input to JLex

There are three sections, delimited by **%%**. The general structure is:

```
User Code
%%
Jlex Directives
%%
Regular Expression rules
```

The User Code section is Java source code to be copied into the generated Java source file. It contains utility classes or return type classes you need. Thus if you want to return a class **IntlitToken** (for integer literals that are scanned), you include its definition in the User Code section.

JLex directives are various instructions you can give JLex to customize the scanner you generate.

These are detailed in the JLex manual. The most important are:

- **%{**
  Code copied into the Yylex class (extra fields or methods you may want)
  %}
- **%eof{**
  Java code to be executed when the end of file is reached
  %eof}
- **%type classname**
  classname is the return type you want for the scanner method, **yylex()**

Macro Definitions

In section two you may also define *macros*, that are used in section three. A macro allows you to give a name to a regular expression or character class. This allows you to reuse definitions and make regular expression rule more readable.

Macro definitions are of the form

```
name = def
```

Macros are defined one per line.

Here are some simple examples:

```
Digit=[0-9]
AnyLet=[A-Za-z]
```

In section 3, you use a macro by placing its name within **{** and **}**. Thus **{Digit}** expands to the character class defining the digits 0 to 9.

Regular Expression Rules

The third section of the JLex input file is a series of token definition rules of the form

```
RegExpr     {Java code}
```

When a token matching the given **RegExpr** is matched, the corresponding Java code (enclosed in "{" and "}") is executed. JLex figures out what **RegExpr** applies; you need only say what the token looks like (using **RegExpr**) and what you want done when the token is matched (this is usually to return some token object, perhaps with some processing of the token text).

Here are some examples:

```
"+"     {return new Token(sym.Plus);}

(" ")+  {/* skip white space */}

{Digit}+ {return
   new IntToken(sym.Intlit,
    new Integer(yytext()).intValue());}
```


Regular Expressions in JLex

To define a token in JLex, the user to associates a regular expression with commands coded in Java.

When input characters that match a regular expression are read, the corresponding Java code is executed. As a user of JLex you don't need to tell it *how* to match tokens; you need only say *what* you want done when a particular token is matched.

Tokens like white space are deleted simply by having their associated command not return anything. Scanning continues until a command with a return in it is executed.

The simplest form of regular expression is a single string that matches exactly itself.

For example,

```
if      {return new Token(sym.If);}
```

If you wish, you can quote the string representing the reserved word (`"if"`), but since the string contains no delimiters or operators, quoting it is unnecessary.

For a regular expression operator, like +, quoting is necessary:

```
"+"     {return
           new Token(sym.Plus);}
```