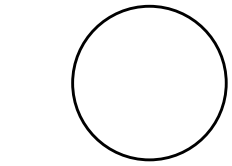
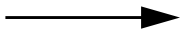
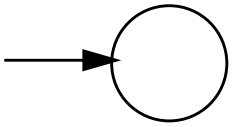
These four components of a finite automaton are often represented graphically:



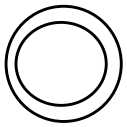
is a state



is a transition



is the start state

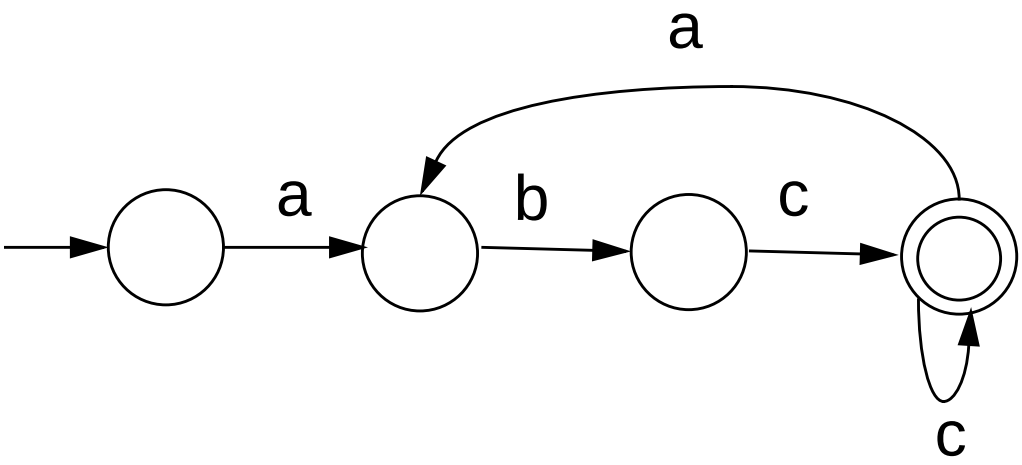


is an accepting state

Finite automata (the plural of automaton is automata) are represented graphically using *transition diagrams.* We start at the start state. If the next input character matches the label on

a transition from the current state, we go to the state it points to. If no move is possible, we stop. If we finish in an accepting state, the sequence of characters read forms a *valid* token; otherwise, we have not seen a valid token.

In this diagram, the valid tokens are the strings described by the regular expression $(a\ b\ (c)^+\ )^+$.



Deterministic Finite Automata

As an abbreviation, a transition may be labeled with more than one character (for example, Not(c)). The transition may be taken if the current input character matches any of the characters labeling the transition.

If an FA always has a *unique* transition (for a given state and character), the FA is *deterministic* (that is, a deterministic FA, or DFA). Deterministic finite automata are easy to program and often drive a scanner.

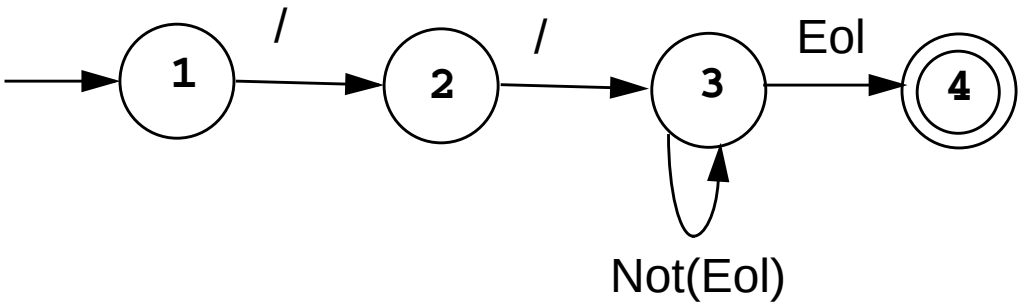
If there are transitions to more than one state for some character, then the FA is *nondeterministic* (that is, an NFA).

A DFA is conveniently represented in a computer by a *transition table.* A transition table, T, is a two dimensional array indexed by a DFA state and a vocabulary symbol.

Table entries are either a DFA state or an error flag (often represented as a blank table entry). If we are in state s, and read character c, then T[s,c] will be the next state we visit, or T[s,c] will contain an error marker indicating that c cannot extend the current token. For example, the regular expression

$$// \ \ \text{Not(Eol)}^* \ \text{Eol}$$

which defines a Java or C++ single-line comment, might be translated into



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
```

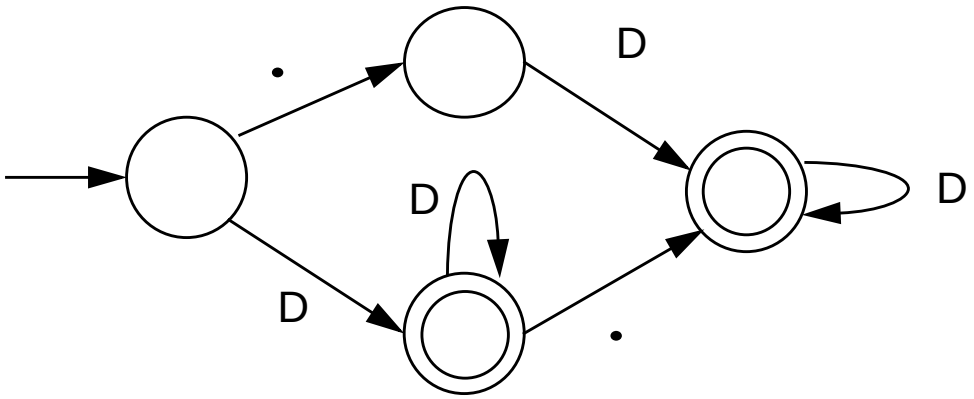
The token being scanned is "hardwired" into the logic of the code. The scanner is usually easy to read and often is more efficient, but is specific to a single token definition.

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

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

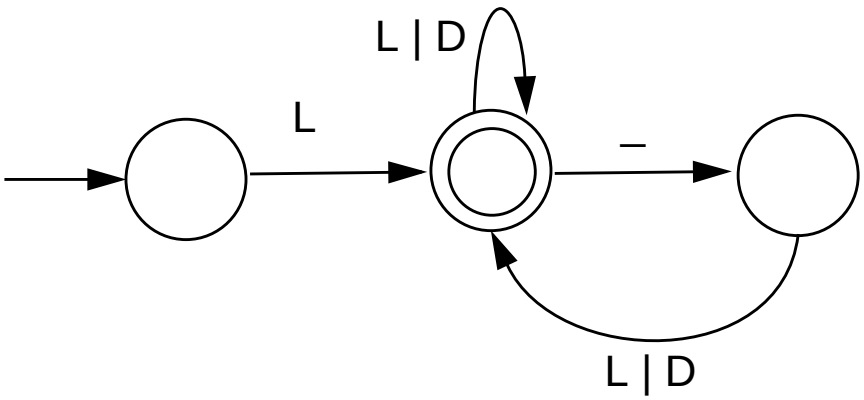
This corresponds to the DFA



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

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

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.