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.

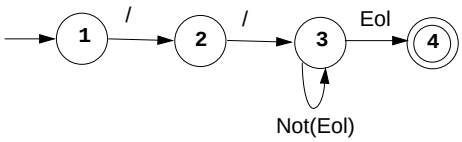
If there are transitions to more than one state for some character, then the FA is *nondeterministic* (that is, an NFA).

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

// **Not(Eol)* 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
```

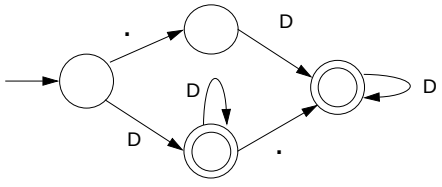

The token being scanned is "hardwired" into the logic of the code. The scanner is usually easy to read and often is more efficient, but is specific to a single token definition.

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

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

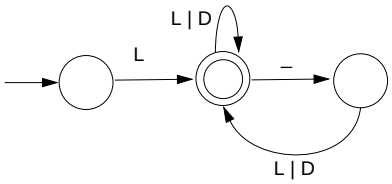
  This corresponds to the DFA



- 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 | D)* (_ (L | 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 using 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 newToken(sym.Plus);}
```


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]`**