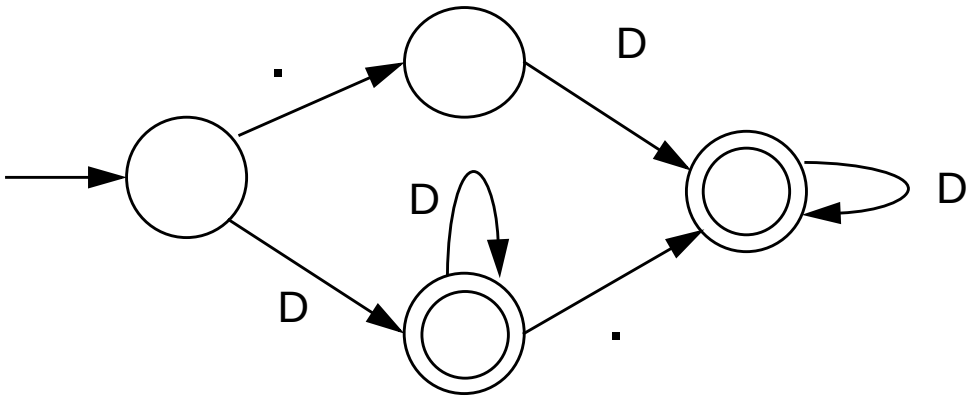
More Examples

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

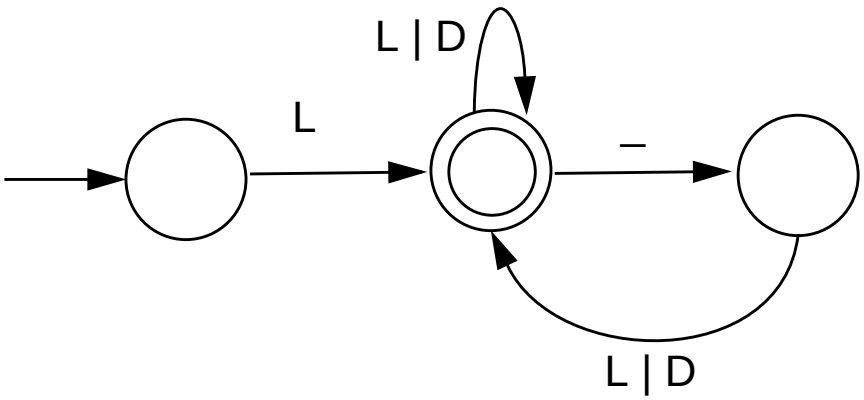
This corresponds to the DFA



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

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

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 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 new Token(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**.