Regular Expressions a sequence of characters that define a search pattern.

POSIX or "Portable Operating System Interface for uniX" is a collection of standards that define some of the functionality that a (UNIX) operating system should support.

Basic Regular Expressions or BRE flavor standardizes a flavour similar to the one used by the traditional UNIX grep command.

"Extended" is relative to the original UNIX grep, which only had bracket expressions, dot, caret, dollar and star.

Flex's regular expression language is essentially POSIX-extended regular expressions

. Matches any single character except the newline character (\n)

[] A character class that matches any character within the brackets.

If the first char- acter is a circumflex (^), it changes the meaning to match any character except the ones within the brackets.

A dash inside the square brackets indicates a character range; for example, [0-9] means the same thing as [0123456789] and [a-z] means any lowercase letter.

A - or] as the first character after the [is interpreted literally to let you include dashes and square brackets in character classes.

Other metacharacters do not have any special meaning within square brackets except that C escape sequences starting with \ are recognized.

Character ranges are interpreted relative to the character coding in use,
so the range [A-z] with ASCII character coding would
match all uppercase and lowercase letters, as well as six punctuation characters
whose codes fall between the code for Z and the code for a .

In practice, useful ranges are ranges of digits, of uppercase letters, or of lowercase letters.

C escape sequences

\a	Alarm or Beep
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Tab (Horizontal)
\v	Vertical Tab
\\	Backslash
\'	Single Quote
\"	Double Quote
\?	Question Mark
\ooo	octal number
\xhh	hexadecimal number
\0	Null

[a-z]{-}[jv]
A differenced character class, with the characters in the first class omitting the characters in the second class (only in recent versions of flex).

^ Matches the beginning of a line as the first character of a regular expression. Also used for negation within square brackets.

$ Matches the end of a line as the last character of a regular expression.

{} If the braces contain one or two numbers, indicate the minimum and maximum number of times the previous pattern can match. For example, A{1,3} matches one to three occurrences of the letter A, and 0{5} matches 00000.

Quantification

\ Used to escape metacharacters and as part of the usual C escape sequences; for example, \n is a newline character, while * is a literal asterisk.

* Matches zero or more copies of the preceding expression. For example, [\t]* is a common pattern to match optional spaces and tabs, that is, whitespace, which Matches " ", " <tab><tab>", or an empty string.

+ Matches one or more occurrences of the preceding regular expression. For example, [0-9]+ matches strings of digits such as 1 , 111 , or 123456 but not an empty string.

? Matches zero or one occurrence of the preceding regular expression. For example, -?[0-9]+ matches a signed number including an optional leading minus sign.

|The alternation operator; matches either the preceding regular expression or the following regular expression. For example, faith|hope|charity matches any of the three virtues.

"…" Anything within the quotation marks is treated literally. Metacharacters other Than C escape sequences lose their meaning. As a matter of style, it's good practice To quote any punctuation characters intended to be matched literally.

() Groups a series of regular expressions together into a new regular expression. For example, (01) matches the character sequence 01, and a(bc|de) matches abc or ade. Parentheses are useful when building up complex patterns with *, +, ?, and |.

/ Trailing context, which means to match the regular expression preceding the slash but only if followed by the regular expression after the slash. For example, 0/1 matches 0 in the string 01 but would not match anything in the string 0 or 02 . The material matched by the pattern following the slash is not "consumed" and remains to be turned into subsequent tokens. Only one slash is permitted per pattern.

The repetition operators affect the smallest preceding expression, so abc+ matches ab followed by one or more c's. Use parentheses freely to be sure your expressions match what you want, such as (abc)+ to match one or more repetitions of abc.

Regular Expression Example Fortran-style numbers

Fortran-style numbers consist of an optional sign, a string of digits that may contain a decimal point, optionally an exponent that is the letter E , an optional sign, and a string of digits.

A pattern for an optional sign and a string of digits is simple enough:

```
[-+]?[0-9]+
```

Note that we put the hyphen as the first thing in [-+] so it wouldn't be taken to mean a character range.

Writing the pattern to match a string of digits with an optional decimal point is harder, because the decimal point can come at the beginning or end of the number. Here’s a few near misses:

[-+]?[0-9.]+	matches too much, like 1.2.3.4
[-+]?[0-9]+\.?[0-9]+	matches too little, misses .12 or 12.
[-+]?[0-9]*\.?[0-9]+	doesn't match 12.
[-+]?[0-9]+\.?[0-9]*	doesn't match .12
[-+]?[0-9]*\.?[0-9]*	matches nothing, or a dot with no digits at all

It turns out that no combination of character classes, ?, *, and + will match a number with an optional decimal point.

Fortunately, the alternation operator | does the trick by allowing the pattern to combine two versions, each of which individually isn't quite sufficient:

```
[-+]?([0-9]*\.?[0-9]+|[0-9]+\.)
[-+]?([0-9]*\.?[0-9]+|[0-9]+\.[0-9]*) This is overkill but also works
```

The second example is internally ambiguous, because there are many strings that match either of the alternates, but that is no problem for flex's matching algorithm.

Flex also allows two different patterns to match the same input, which is also useful but Requires more care by the programmer.

Now we need to add on the optional exponent, for which the pattern is quite simple:
E(+|-)?[0-9]+

Now we glue the two together to get a Fortran number pattern:
[-+]?([0-9]*\.?[0-9]+|[0-9]+\.)(E(+|-)?[0-9]+)?

Since the exponent part is optional, we used parens and a question mark to make it an optional part of the pattern.

Note that our pattern now includes nested optional parts, which work fine and as shown here are often very useful.

This is about as complex a pattern as you'll find in most flex scanners.
It's worth reiterating that complex patterns do not make the scanner any slower.

Write your patterns to match what you need to match, and trust flex to handle them.

Most flex programs are quite ambiguous, with multiple patterns that can match the same input. Flex resolves the ambiguity with two simple rules:

- Match the longest possible string every time the scanner matches input.
- In the case of a tie, use the pattern that appears first in the program.

These turn out to do the right thing in the vast majority of cases. Consider this snippet from a scanner for C source code:

```
"+"      { return ADD; }
"="      { return ASSIGN; }
"+="     { return ASSIGNADD; }
"if"     { return KEYWORDIF; }
"else"   { return KEYWORDELSE; }
[a-zA-Z_][a-zA-Z0-9_]* { return IDENTIFIER; }
```

For the first three patterns, the string += is matched as one token, since += is longer than + .

For the last three patterns, so long as the patterns for keywords precede the pattern that matches an identifier, the scanner will match keywords correctly.