Lexical classes and Scanning

REGEX :: The default alphabet we are going to use is ASCII printable characters. We can form character classes out of this alphabet as follows:

$DIGIT [0-9]

$NON-ZERO [^0] IN $DIGIT

$CHAR [a-zA-Z]

$UPPER [^a-z] IN $CHAR

The above examples show digits 0 through 9 out of ASCII printable characters define a character class DIGIT and thus, anywhere DIGIT appears it could mean any of the 0 through 9. We have formed a sub-class NON-ZERO out of this on the 2nd line – NON-ZERO removes 0 out of DIGIT and means it corresponds to 1 through 9 anywhere it appears. Similarly, we have CHAR class defined out of ASCII printable characters first and then a sub-class UPPER defined out of this…

The following characters must be *escaped* if they were to appear as a part of alphabet (i.e., preceded by a backslash): space, \, *, +, |, [,], (,), ., ' and ". If they are not escaped, they will be treated as regex operators as follows: a space without a preceding \ in a regex is ignored, | is the regex union, ( ) are for grouping, * + are for repetitions, [ ] and . are for character classes (see RegEx section below).

Note that first the scanner generator should analyze the specification of character classes and determine if it is a valid specification and then build the character classes. Following rules should be used :

Character Classes:

Within a REGEX, a dot indicates any character (i.e., the class of all strings of length one). Any other character class is delimited by [ ] and follows these rules:

- There can't be nested [ ]'s. Inside a [ ], you can specify the characters included in the class (e.g., [abc] to describe the class containing a, b, and c).
- To specify a character class by excluding characters, use ^ as the first element after the opening followed by the keyword IN followed by character class from which to exclude… [(e.g., [^abc] IN CHAR specifies we are excluding characters a, b, or c from a CHAR class)
- Inside [ ] or [^], you can use ranges of characters based on their ASCII numbers. For example, [a-z] represents all lowercase letters, whereas [^0-9] represents exclude the digits 0 through 9.
- It is an error if a range is empty (e.g., [1-0]).
- Inside [ ] or [^], you must escape ^ if you want to specify that character. Also, to exclude \, -, [and], you must escape them. For example, [^\\\^\-\]\[] represents all characters but \, ^, -, [and]. Those are the only characters that can be escaped within [ ]'s. Note that [*+.] represents the *, +, . and space characters (i.e. they don't need escaping).
- It is not an error if a character appears more than once inside a [ ] or [^], including [ccc] and [1-64-9].

Spaces, tabulations, line ends and carriage returns are considered white space and skipped by the scanner.

RegEx Specification

To build a regex, you must use characters and character classes as the basic building blocks, | for union of subexpressions, round parentheses to surround a subexpression for giving it a higher precedence, *

and + for zero-to-infinite and one-to-infinite repetitions (e.g., (ab)+ is ab, abab, etc.). Empty strings in larger regex's are implicit. For example, 'a(|b)' represents the language {a, ab}, and '(|b)' means "empty string or b".

When matching strings, * + are "greedy" up to **the end of the line (detected by \n)** . For example, searching for (ab)* in a line abababa followed by line abab matches this regex twice: once for the 3 ab in that first line and once for the 2 ab in the second line (as a separate, second match).

The precedences of regex operators are : ( ) highest followed by * and + followed by concatenation followed by | (union) – the full grammar for regex is given in the Appendix.

First the scanner generator will read the regex specification of a token and make sure it is legal – it will then use the precedences and generate primitive NFAs and put those together as per the precedences and make one giant NFA out of it. It will then convert the giant NFA to a DFA.

Appendix I: Formal Grammar for REGEX

To avoid any confusion with the format of regular expressions, this is the grammar:

<reg-ex> → <rexp>
<rexp> → RE_CHAR | <char-class> | (<rexp>) | <rexp> UNION <rexp> | <rexp> <rexp> |
(<rexp>) * | (<rexp>) + | RE_CHAR * | RE_CHAR + | ε
<char-class> → . | [<char-set-list>] | <exclude-set> | <defined-class>
<char-set-list> → <char-set> <char-set-list> | ε
<char-set> → CLS_CHAR | <range>
<range> → CLS_CHAR - CLS_CHAR
<exclude-set> → [^ <char-set>] IN [<char-set>] | [^<char-set>] IN <defined-class>

Following is the grammar reduced to a form that can be parsed by recursive descent parser, with operator precedence incorporated and left recursion removed (note : you will be writing a recursive descent parser for this grammar, please use the slides on how to build a recursive descent parser):

<reg-ex> → <rexp>
<rexp> → <rexp1> <rexp'>
<rexp'> → UNION <rexp1> <rexp'> | ε
<rexp1> → <rexp2> <rexp1'>
<rexp1'> → <rexp2> <rexp1'> | ε
<rexp2> → (<rexp>) <rexp2-tail> | RE_CHAR <rexp2-tail> | <rexp3>
<rexp2-tail> → * | + | ε
<rexp3> → <char-class> | ε
<char-class> → . | [<char-class1> | <defined-class>
<char-class1> → <char-set-list> | <exclude-set>
<char-set-list> → <char-set> <char-set-list> |]
<char-set> → CLS_CHAR <char-set-tail>
<char-set-tail> → – CLS_CHAR | ε
<exclude-set> → ^ <char-set>] IN <exclude-set-tail>
<exclude-set-tail> → [<char-set>] | <defined-class>

Spaces, tabulations, line ends and carriage returns are considered white space and skipped.

UNION is the | character representing the union of two (sub-) regular expressions.
RE_CHAR is the union of two sets:

1) the ASCII printable characters (see http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters) other than space, \, *, +, ?, |, [,], (,), ., ' and "
2) escaped characters: \ (backslash space), \\, *, \+, \?, \|, \[, \], \(, \), \., \' and \"

CLS_CHAR is the union of two sets:

1) the ASCII printable characters (see http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters) other than \, ^, -, [and]
2) escaped characters: \\, \^, \-, \[and \]

<defined-class> is a class defined in the definition section of the input format file shown above such as DIGIT or CHAR or UPPER etc.

Remember that it is not an error if a character appears more than once inside a [ ] or [^], whether it's explicit or in a range. See project description above for more details about the semantics of a REGEX.

Note that tabulations and newline characters (\n, carriage return, line feed) are not allowed in REGEX simply because they are not ASCII printable characters. Space, however, is allowed in REGEX (without escaping inside a [ ] or [^], and escaped otherwise).