```
#Now that you have lexed the string (or not), it's time to parse it to a
#tree, a so-called Abstract Syntax Tree.

#Parsing is NOT optional, you need the Abstract Syntax Tree of a regex to
#do anything useful with it

#An AST is a hierarchical representation of the code, showing the true
structure of any expression language, like Regex.

#You can use any parsing algorithm you fancy, but the easiest one to get
#started with is Recursive Descent

#Let us first define the AST structure



interface AST-Node {}

class Or-AST-Node is AST-Node {
   AST-Node left,
   AST-Node right
}
class Sequence-AST-Node is AST-Node {
    AST-Node left,
    AST-Node right
}
#etc for other types of recursive AST-Nodes
#-------------------------------------------------------------------------
class Literal-Character-AST-Node is AST-Node {
    char c
}
class Character-Class-AST-Node is AST-Node {
    set<char
       |pair<char>> chars
}



#What is the grammar we're parsing ? You can't write a parser without at
#least some idea of what you're parsing, and that idea has to be in the
#form of Backus-Naur form grammar
```

```
#Here's a nice one
#     regex-expr ::= regex-or-expr
#     regex-or-expr ::= regex-seq-expr (OR_TOKEN regex-seq-expr)*
#     regex-seq-expr ::= regex-quantified-expr (regex-quantified-expr)*

#     regex-quantified-expr ::= regex-base-expr (STAR_TOKEN | PLUS_TOKEN |
QUESTION_MARK_TOKEN)?
#     regex-base-expr ::= LITERAL_CHAR_TOKEN
#                       |  OPEN_SQUARE_BRACKET_TOKEN
#                          square-bracket-content
#                          CLOSED_SQUARE_BRACKET_TOKEN
#                       |  OPEN_PARENTHESIS_TOKEN
#                          regex-expr
#                          CLOSED_PARENTHESIS_TOKEN
#
#     square-bracket-content ::= square-bracket-element+
#     square-bracket-element ::= LITERAL_CHAR_TOKEN
#                              | LITERAL_CHAR_TOKEN DASH LITERAL_CHAR_TOKEN
#Or, in plain English, a well-formed regular expression according to our
#parser is one that :
#     1- Is a single big or expression
#
#     2- Where an or expression is just a sequence expression, possibly
#followed by 0 or more sequence expressions separated by the '|' character
#
#     3- Where a sequence expression is just a quantified expression,
#possibly followed by 0 or more quantified expressions
#
#     4- Where a quantified expression is just a base expression, possibly
#        followed by either '*', '+', or '?' characters
#
#     5- where a base expression is just
#                                a literal character ('a','b', etc...),
#                             or a square bracket,
#                             or a parenthesis followed by a whole other
#regular expression (goto 1 again) then a closing parenthesis
#
#And that's it, that's a complete description of all regular expressions we
accept
#The big idea of recursive descent is that the above description IS ALREADY
#code, let's see why
#-----------------------------------------------------------------------
```

```
#     regex-expr ::= regex-or-expr
subroutine Regex-Parse of
                  input token-stream
                  output AST

      return parse-or(token-stream)
#-------------------------------------------------------------------------
#regex-or-expr ::= regex-seq-expr (OR_TOKEN regex-seq-expr)*
subroutine parse-or of
                  input  token-stream
                  output AST

      left = parse-seq(token-stream)

      if token-stream ended then return left

      or-expr = left

      while token-stream.curr-token is OR_TOKEN then
            token-stream.advance-token-pointer()

            right = parse-seq(token-stream)

            or-expr = new Or-AST-Node(or-expr,right)

      return or-expr
#-------------------------------------------------------------------------
#regex-seq-expr ::= regex-quantified-expr (regex-quantified-expr)*
subroutine parse-seq of
                  input token-stream
                  output AST

      left = parse-quantified(token-stream)

      if token-stream.ended then return left

      seq-expr = left

      while token-stream.curr-token is not an OR_TOKEN
        and token-stream.curr-token is not a CLOSED_PAREN_TOKEN then
            right = parse-quantified(token-stream)

            seq-expr = new Sequence-AST-Node(seq-expr,right)
```

```
      return seq-expr
#-------------------------------------------------------------------------
#regex-quantified-expr ::= regex-base-expr (STAR_TOKEN | PLUS_TOKEN |
#                          QUESTION_MARK_TOKEN)?
subroutine parse-quantified
                  takes a token-stream
                  outputs an AST

      left = parse-base(token-stream)

      if token-stream ended then return left

      token-stream.advance-token-pointer() if token-stream.curr-token is
either STAR or PLUS or QUESTION_MARK

      if token-stream.curr-token is STAR then return Star-AST-Node(left)
      if token-stream.curr-token is PLUS then return Plus-AST-Node(left)
      if token-stream.curr-token is QUESTION_MARK then
           return Optional-AST-Node(left)

      return left
#-------------------------------------------------------------------------
#     regex-base-expr ::= LITERAL_CHAR_TOKEN
#                       |  OPEN_SQUARE_BRACKET_TOKEN
#                          square-bracket-content
#                          CLOSED_SQUARE_BRACKET_TOKEN
#                       |  OPEN_PARENTHESIS_TOKEN
#                          regex-expr
#                          CLOSED_PARENTHESIS_TOKEN
subroutine parse-base of
                  input  token-stream
                  output AST
      curr-token = token-stream.curr-token
      token-stream.advance-token-pointer()

      if curr-token is a LITERAL_CHARACTER then
             return Literal-Character-Node(curr-token.str)
      if curr-token is an OPEN_SQUARE_BRACKET then
             return parse-square-bracket(token-stream)
      if curr-token is an OPEN_PARENTHESIS then
             expr = Regex-Parse(token-stream)
             assert token-stream.curr-token is a CLOSED_PARENTHESIS,
```

```
raising an error otherwise
            return expr
```

Let's see an example of how "ab*c+de?(f|g|h)|mr|n|[pq]" would parse. For reference, the correct precedence is here : (https://regexper.com/#ab*c%2Bde%3F%28f%7Cg%7Ch%29%7Cmr%7Cn%7C%5Bpq%5D)

- Regex-Parse is the first function to be called, calls parse-or
    - parse-or calls parse-seq
        - parse-seq calls parse-quantified
            - parse-quantified calls parse-base
                - parse-base sees the current character is 'a',
                - so it returns an Literal-Character-Node representing it
            - now parse-quantified looks at the current character, it's not one of the 3 it expects, so it simply returns what parse-base returned (which is the Literal-Character-Node containing 'a')
        - now parse-seq looks at the current character, it's not an |, so it calls parse-quantified again, which parses the b* part of the string and returns it as a Star-AST-Node in the right variable
        - now parse-seq forms seq-expr = Sequence-AST-Node(
          left= Literal-Character-Node('a'),
          right= Star-AST-Node('b'))
        - and keeps calling parse-quantified again and again until ab*c+de?(f|g|h) is parsed completely
        - finally, it sees the current character is '|', so it returns what it has parsed
    - back at the parse-or, it sees the current character is '|', so it calls parse-seq again to parse mr, parse-seq returns when it sees the '|' token before 'n'
    - and so on, it parses 'n' and [pq] in the same way
    - finally returns to Regex-Parse
- Which finally returns what parse-or returns

The recursive structure of the grammar is reflected in the recursive subroutines and how they

call each other.

This is Recursive Descent, Extremely Elegant, you will never want to use a Ready-Made parser generator again.

(You will have to write the grammar anyway).