Implementing a Scanner, Recognizer, and Parser for Regular Expressions

Professor Maida
CMPS 450.1 and CMPS 450.2
Third Draft

October 18, 2024

1 Introduction

This project uses the Python programming language to write a parser for regular expressions. The project has one part:

1. Build a recursive descent recognizer that prints concrete syntax trees without building them (a parse tree can be printed without building it b/c the parse tree represents the recursive descent control flow). (30%)

2 Grammar

The grammer specifies the syntax of regular expressions. The grammar has three unary operators, ‘*’, ‘+’, and ‘?’. These are all quantifiers of equal precedence. The precedence of all operators, from highest to lowest, is as follows: parentheses, quantifiers, character sequence (or concatenation), and alternation (disjunction).

The BNF for regular expressions is given below and specifies the precedence hierarchy. The BNF grammar specifies the structure of concrete syntax trees, but with one exception. We allow operators such as “ | ” to take an arbitrary number of arguments. Therefore the arguments will all be at the same level in the hierarchy (more on this later).

```
BNF
===
1. <re>            ::= <re> "|" <simple-re | <simple-re>
2. <simple-re>     ::= <simple-re> <basic-re> | <basic-re>
3. <basic-re>      ::= <elementary-re> |
                       <elementary-re> "*" |
                       <elementary-re> "+" |
                       <elementary-re> "?" |
4. <elementary-re> ::=  "(" <re> ")"        |
                        "."                 |
                        <char-or-meta>      |
                        "[" <set-items> "]" |
                        "[^" <set-items> "]"
5. <char-or-meta>  ::= any NON-METACHAR | any METACHAR except "\" | "\" METACHAR
6. <set-items>     ::= <char-or-meta> | <char-or-meta> <set-items>
```

	Token	Type
1.	"|"	VERT
2.	"*"	STAR
3.	"+"	PLUS
4.	"?"	QMARK
5.	"("	LPAREN
6.	")"	RPAREN
7.	"."	PERIOD
8.	"[^"	LNEGSET
9.	"["	LPOSSET
10.	"]"	RSET
11.	"<"	LANGLE
12.	">"	RANGLE
13.	"\"	BSLASH
14.	"\n"	EOL
15.	A-Za-z_\	CHAR
16.	anything not recognized	ERROR

Table 1: Specification of tokens and token types for regular expressions.

The EBNF is given below and is translated from the BNF. The EBNF is used to generate syntax diagrams that determine the flow of control of the parser.

```
EBNF
====
1. <re>            ::=  <simple-re> { "|" <simple-re> }
2. <simple-re>     ::=  <basic-re>  { <basic-re> }
3. <basic-re>      ::=  <elementary-re> [ "*" | "+" | "?" ]
4. <elementary-re> ::=  "(" <re> ")"        |
                        "."                 |
                        <char-or-meta>      |
                        "[" <set-items> "]" |
                        "[^" <set-items> "]"
5. <char-or-meta>  ::= any NON-METACHAR | any METACHAR except "\" | "\" METACHAR
6. <set-items>     ::=  <char-or-meta> { <char-or-meta> }
```

The terminal constituents `NON-METACHAR` and `METACHAR` represent token types. This grammar requires two-step look ahead in order to process the last clause in line 4. Because of this, we need to be able to unread a character. A reliable way to do this is to read the characters in a list, and then process the list.

3 Scanner (Part 1)

You can use the scanner provided on Moodle, Week 8, Item 4. You are also allowed to write your own scanner. The token types are shown in Table 1 if you choose the write your own scanner.

4 Using the Scanner from the Recognizer

Put the scanner on file `regexTokenizer` and import it as `tk`. Import the `peekable` function from the `more_itertools` library as shown below.

```
    import regexTokenizer as tk
    from more_itertools import peekable
```

The line below shows how to get the tokens out of the scanner. The peekable function accepts a function that generates output via the `yield` command and makes it peekable.

```
    tokens = peekable(tk.tokenize(’t(oo?|wo)’))
```

You can now use the commands `peek` and `next` on `tokens`. The command `peek` looks at the next character in the input stream but does not advance it. You can use `peek` as many times as you want without changing the input stream. It’s use is shown below.

```
peek_tok = tokens.peek(None)
```

In the above `peek` is used as a method. We give the parameter `None` to serve as a sentinel to signal the end of input. In other works, if `peek` returns `None`, you are at the end of the input stream. Otherwise, it returns the token at the front of the input stream.

The command `next` advances the input stream by one token. It is used as a function instead of a method. Its use is shown below.

```
next(tokens)
```

In my implementation, I always used `peek` before `next`, so I always knew what I was looking at before advancing the input stream. Also, I never applied `next` to the token input stream when `peek` returned `None`.

5 Sample Input and Output for the Recognizer

This section discusses the recognizer. The recognizer prints a parse tree without building it.

Here is code that I used in my main module. You are free to use or modify. Notice that the input is provided. The last two lines are assuming the parser is an instance of class `RegexParser`. If you don’t take an object-oriented approach, you can modify this code appropriately.

```
    inputs = [’two’, ’t|w|o’, ’[two]’, ’[ˆtwo]’, ’t(oo?|wo)’, "(\<(/?[ˆ\>]+)\>)"]
    for regex in inputs:
        print(f’Proceesing expression: "{regex}"’)
        tokens = peekable(tk.tokenize(regex))
        parser = RegexParser(tokens)
        parser.parse_re(0)
```

For each statement in the file, you are to print a syntax tree as the parse proceeds. You can print this tree **without** actually **building the parse tree data structure inside the program**. The reason this is possible is that the control structure of the program mimics the parse tree.

The required output is shown below. The amount of horizontal indentation depicts the nesting level of the node within the tree. The node names “RE”, “S_RE”, “B_RE”, and “E_RE” stand for *regular expression*, *simple regular expression*, *basic regular expression*, and *elementary regular expression*, respectively.

```
Processing expression: "two"
RE
   S_RE
      B_RE
         E_RE
            CHAR_OR_META
               t CHAR
      B_RE
         E_RE
            CHAR_OR_META
               w CHAR
      B_RE
         E_RE
            CHAR_OR_META
               o CHAR

Processing expression: "t|w|o"
RE
   S_RE
      B_RE
         E_RE
            CHAR_OR_META
               t CHAR
   | VERT
   S_RE
      B_RE
         E_RE
            CHAR_OR_META
               w CHAR
   | VERT
   S_RE
      B_RE
         E_RE
            CHAR_OR_META
               o CHAR

Processing expression: "[two]"
RE
   S_RE
      B_RE
         E_RE
            [ LPOSSET
            SITEMS
               CHAR_OR_META
                  t CHAR
               CHAR_OR_META
                  w CHAR
               CHAR_OR_META
                  o CHAR
            ] RSET

Processing expression: "[ˆtwo]"
RE
   S_RE
      B_RE
         E_RE
            [ˆ LNEGSET
            SITEMS
               CHAR_OR_META
                  t CHAR
               CHAR_OR_META
                  w CHAR
               CHAR_OR_META
                  o CHAR
            ] RSET

Processing expression: "t(oo?|wo)"
RE
   S_RE
      B_RE
         E_RE
            CHAR_OR_META
               t CHAR
      B_RE
         E_RE
            ( LPAREN
            RE
               S_RE
                  B_RE
                     E_RE
                        CHAR_OR_META
                           o CHAR
                  B_RE
                     E_RE
                        CHAR_OR_META
                           o CHAR
                     ? QMARK
               | VERT
               S_RE
                  B_RE
                     E_RE
                        CHAR_OR_META
                           w CHAR
                  B_RE
                     E_RE
                        CHAR_OR_META
                           o CHAR
            ) RPAREN

Processing expression: "(\<(/?[^\>]+)\>)"
RE
   S_RE
      B_RE
         E_RE
            ( LPAREN
            RE
               S_RE
                  B_RE
                     E_RE
                        CHAR_OR_META
                           \ BSLASH
                           < LANGLE
                  B_RE
                     E_RE
                        ( LPAREN
                        RE
                           S_RE
                              B_RE
                                 E_RE
                                    CHAR_OR_META
                                       / CHAR
                                 ? QMARK
                              B_RE
                                 E_RE
                                    [^ LNEGSET
                                    SITEMS
                                       CHAR_OR_META
                                          \ BSLASH
                                          > RANGLE
                                    ] RSET
                                 + PLUS
                        ) RPAREN
                  B_RE
                     E_RE
                        CHAR_OR_META
                           \ BSLASH
                           > RANGLE
            ) RPAREN
```

For each statement in the file, you are to print a syntax tree as the parse proceeds. The parse tree for the first statement in the input given above is shown below. The amount of horizontal indentation depicts the nesting level of the node within the tree.