REVISED: Implementing a Scanner, Recognizer, and Parser for Regular Expressions

Professor Maida
CMPS 450.2
Third draft

November 9, 2008

Changes from the October 19 version are:

The purpose of these changes is to address inconsistencies in the logic of the project as specified in the October 19 draft. These inconsistencies were brought to my attention by members of the class who were trying to implement the project. The inconsistencies fall into three categories:

1. The scanner has too much semantic knowledge.
2. The definition of NON–METACHAR was not fully specified.
3. The parse trees given in Section 4 of the specification are not completely reflective of the structure of the grammar and are not sufficiently detailed.
4. In the specification, the symbols VBAR and VERT were used interchangeably.

We now address these issues beginning with the scanner.

First issue. In the previous specification, if a meta character was preceded by a backslash character, the meta character was classified as a character. This reflects a level of semantic analysis. However, the role of the scanner is simply to return tokens. The semantic analysis is supposed to be postponed to the stage where abstract syntax trees are constructed. The old output specification for the token sequence “\<” was:

BSLASH
CHAR: <

Since the above reflects semantic analysis by the scanner, the new specification is:

BSLASH
LANGLE

In the new specification, the left angle bracket is not classified as an ordinary character until the abstract syntax trees are constructed.
Second issue. The second issue was that the definition of NON-METACHAR was not fully specified and several students asked about this. For the purposes of this project, the NON-METACHARs are the following:

1. the upper and lower case letters (A–Za–z).
2. the digits (0–9).
3. the underscore character (“_”).
4. and, the forward slash character (“/”).

The METACHARs (for the purposes of this project) are given in lines 1 – 14 of Table 1 in the specification. In conjunction with this, Rule 5 of the grammar has been revised. The old rule was:

5. `<char-or-meta> ::= any NON-METACHAR | "\" METACHAR`

The new rule is:

5. `<char-or-meta> ::= any NON-METACHAR | any METACHAR except "\" | "\" METACHAR`

The purpose of the above change is to remove the ambiguity from the rule.

Third issue. The third issue relates to the parse trees given in Section 4 of the specifications (The specs in Section 5 for the abstract syntax trees have not changed). These parse trees did not contain enough information. First, the grammar constituent `<char-or-meta>` was not represented as a node in the parse trees. The new specification fixes this. Second, parses involving a backslash character preceding a meta character were collapsed into one line. The new specification breaks this into two lines so that the information is more explicit. For example, in the old specification the subtrees for “\<” looked like:

```
E_RE
 \< LANGLE
```

According to the new specification, they look like:

```
E_RE
 CHAR_OR_META
 \ BSLASH
 < LANGLE
```

In the above, Line 5 of the grammar is explicitly represented as a node (as it should be). Also, the representation for “\<” has been split into two lines. This does not affect the abstract syntax trees. There are no changes in the abstract syntax tree specification.

Fourth issue. In regard to the fourth issue, the previous write-up inadvertently used the symbols VERT and VBAR interchangeably. The current write-up just uses just one symbol, which is VERT. In other words, replace all occurrences of VBAR with VERT.
1 Introduction

This project uses the Java programming language to write a parser for regular expressions. The project has the following three parts:

1. building a scanner for recognizing tokens in regular expressions. (10%)
2. building a recursive descent recognizer that prints concrete syntax trees without building them (Yes, it is possible to print a parse tree without building it). (30%)
3. building a recursive descent parser that builds concrete syntax trees and then prints them. (30%)
4. building a recursive descent parser that builds abstract syntax trees and then prints them. (30%)

These parts should be done in the sequence given. Each step prepares you to do the next step. The scanner will identify tokens that are used in regular expressions. The recognizer and parser will use the scanner as a method. Once you build the recognizer, you have a control structure that can be used to build parse trees. It is conceptually easier to build concrete syntax trees than abstract syntax trees. The reason is that abstract syntax trees require node filtering, which is explained in Section 7. Therefore step three is in front of step four. Once you have to fluency to build concrete syntax trees in step three, you should find it relatively easy to build abstract syntax trees.

Later parts of this write-up specify the input-output requirements for each part of the project and how you should approach it.

2 Grammar

The grammar 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.

The BNF 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.

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> ::= 
"
" | "\ ." | "[<char-or-meta> | "[<char-or-meta> <set-items> "]"] | "[<char-or-meta> <set-items> "]"
5. <char-or-meta> ::= any NON-METACHAR | any METACHAR except "" | "\" METACHAR
6. <set-items> ::= <char-or-meta> | <char-or-meta> <set-items>

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

<table>
<thead>
<tr>
<th>Token</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;*&quot;</td>
<td>STAR</td>
</tr>
<tr>
<td>&quot;+&quot;</td>
<td>PLUS</td>
</tr>
<tr>
<td>&quot;?&quot;</td>
<td>QMARK</td>
</tr>
<tr>
<td>&quot;(&quot;</td>
<td>LPAREN</td>
</tr>
<tr>
<td>&quot;)&quot;</td>
<td>RPAREN</td>
</tr>
<tr>
<td>&quot;.&quot;</td>
<td>PERIOD</td>
</tr>
<tr>
<td>&quot;[&quot;</td>
<td>LPOSSET</td>
</tr>
<tr>
<td>&quot;[ˆ&quot;</td>
<td>LNEGSET</td>
</tr>
<tr>
<td>&quot;]&quot;</td>
<td>RSET</td>
</tr>
<tr>
<td>&quot;&lt;&lt;&quot;</td>
<td>LANGLE</td>
</tr>
<tr>
<td>&quot;&gt;&quot;</td>
<td>RANGLE</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>BSLASH</td>
</tr>
<tr>
<td>&quot;\n&quot;</td>
<td>EOL</td>
</tr>
<tr>
<td>ordinary</td>
<td>CHAR</td>
</tr>
<tr>
<td>anything</td>
<td>ERROR</td>
</tr>
</tbody>
</table>

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. Java supports this using the PushbackReader class that has a method named unread() [1, p. 530].

3 Scanner

This section specifies the output requirements of the scanner. The token types are shown in Table 1. They can be declared in Java using the code below.

```java
public enum TokenType { VERT, STAR, PLUS, QMARK, LPAREN, RPAREN, PERIOD, BSLASH, LPOSSET, LNEGSET, RSET, LANGLE, RANGLE, EOL, CHAR, ERROR};
```

Consult [2] on p. 82 and p. 105 for more information on using a scanner with a parser.

The scanner should read the following input file.
For this file, it should read each line and print the following output.

**no change to this specification**
Processing expression: "t(oo?|wo)"
CHAR: t
LPAREN
CHAR: o
CHAR: o
QMARK
VERT
CHAR: w
CHAR: o
RPAREN
EOL

**this specification has changed**
Processing expression: "\(<\(\/?[^\">]+\)>\)"
LPAREN
BSLASH
LANGLE
BSLASH
LPAREN
CHAR: /
QMARK
LNEGSET
BSLASH
RANGLE
RSET
PLUS
RPAREN
BSLASH
RANGLE
RPAREN
EOL

# 4 Sample Input and Output for the Recognizer

This section discusses the recognizer. The recognizer will read input from a file consisting of a sequence of regular expressions. A sample input file appears below.

t(oo?|wo)
\(<\(\/?[^\">]+\)>\)

For each statement in the file, you are to print a syntax tree as the parse proceeds. 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.

**all of these specifications have changed**
Processing expression: "two"
Processing expression: "t|w|o"

```
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: "[two]"

```
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: "[\^two]"

```
S_RE
  B_RE
    E_RE
      ^ LNEGSET
        SITEMS
          CHAR_OR_META
            t CHAR
          CHAR_OR_META
            w CHAR
          CHAR_OR_META
            o CHAR
    RSET
```

Processing expression: "t(oo?|wo)"

```
S_RE
  B_RE
    E_RE
      t(oo?|wo)
```

```
Processing expression: "\(\langle/\?[^\<>]+\rangle\)"
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.

5 Sample Input and Output for the Parser

This section discusses the input-output requirements for the REparser. This parser will come in two versions corresponding to steps three and four of the project. The output requirements for step three are the same as for the recognizer described in Section 4.

We now describe the requirements for step four. The parser will read input from a file consisting of a sequence of regular expressions. A sample input file appears below. It consists of six expressions, one on each line.

two
t|w|o
[two]
[^two]
t(oo?|wo)
(\</?[^>]+>)

For each statement in the file, you are to create an abstract syntax tree and print the tree when the parse is completed. The parse trees are shown below. The amount of horizontal indentation depicts the nesting level of the node within the tree. The parser builds a tree structure representation of the regular expression that takes a prefix form. All of the operators begin with the letter “K”, which stands for Kleene operator. Tree structure representations for lines 1 and 5 of the file are shown in Figures 1 and 2, respectively. When you look at the representation in Figure 1, you will see that the node for RE that represents the “K_cat” operation has been eliminated. This is because it did not serve a functional purpose, since the “K_cat” operation had only one argument. The actual output of the REparser is given below.

no change to these specifications
Processing expression: "two"
(K_cat
   (K_char
      t CHAR
   )
   (K_char
      w CHAR
   )
)
Processing expression: "t[w|o"
(K_cat
  (K_char
t CHAR
  )
  (K_char
  w CHAR
  )
  (K_char
  o CHAR
  )
)

Processing expression: "[two]"
(K_pos_set
  (K_char
t CHAR
  w CHAR
  o CHAR
  )
)

Processing expression: "[^two]"
(K_neg_set
  (K_char
t CHAR
  w CHAR
  o CHAR
  )
)

Processing expression: "t(oo?:wo)"
(K_cat
  (K_char
t CHAR
  )
  (K_alt
    (K_cat
      (K_char
        o CHAR
      )
      (K_qmark
        (K_char
          o CHAR
        )
      )
    )
    (K_cat
      (K_char
        w CHAR
      )
      (K_char
        o CHAR
      )
    )
  )
)

Processing expression: "(\(</?\[\^\>]+)\)>"
Figure 2: Abstract syntax tree for the regular expression “t(oo?|wo)”.

```java
(K_cat
  (K_qmark
    (K_char
      / CHAR
    )
  )
  (K_plus
    (K_neg_set
      (K_char
        > RANGLE
      )
    )
  )
) (K_char
  > RANGLE
)
```

6 Implementing the Recognizer

This section discusses implementation of the recognizer. Your main class should be called `RErecognizer`. You should be able to run your program with the command

```
java RErecognizer fileName
```

In the above, `fileName` consists of a list of regular expressions, one on each line. The code samples below are methods and variables in the `RErecognizer` class.

1. public static void main(String args[]) throws IOException {
2.     if (args.length < 1) throw new IllegalArgumentException("No arguments");
3.     FileReader fileIn = new FileReader(args[0]);
```
The method recognize_re() in line 8 is a method that you must implement. It will implement line 1 of the EBNF grammar. pbIn and curr_type are global variables.

```java
static PrintStream cout = System.out;
static TokenType curr_type; // for communication with the scanner
static int curr_char; // for communication with the scanner
static PushbackReader pbIn; // understands unread();
```

The code below implements recognize_re().

```java
static void recognize_re(int level) throws IOException {
    print_indentation(level);
    cout.println("RE");
    recognize_simple_re(level + 1);
    while (curr_type == TokenType.VERT) {
        print_indentation(level + 1);
        cout.printf("%c %s%n", curr_char, curr_type);
        match(TokenType.VERT);
        recognize_simple_re(level + 1);
    }
}
```

In the above, the communication with the scanner is via the match method. The match method is implemented below.

```java
static void match(TokenType ttype) throws IOException {
    if (ttype == curr_type) curr_type = getToken();
    else { System.out.println("Match error: "+ttype);
        System.exit(1);
    }
}
```

This method is modeled after the function with the same name on p. 106 of [2].

Once you have implemented the recognizer correctly, you can be sure that your implementation of the syntax diagrams that correspond to the EBNF is correct. You can then use these control structures when building the parser.

## 7 Implementing the Parser

The REparse implements abstract syntax trees. Trees will have two node types. These will be: AtomicNode, and ConsCellNode. Figures 1 and 2 show AtomicNodes and ConsCellNodes. The structure of these node types is shown in Figure 3. There is a third abstract node type. AbstractNode is a type that is the superclass of AtomicNodes and ConsCellNodes. Since both ConsCellNodes and AtomicNodes are also AbstractNodes, they are members of the same data type.
Figure 3: Structures of conscell nodes and atomic nodes. Field types are inside of the fields. Field names are below the fields.

The source code below corresponds to recognize_re() for the RErecognizer. The name of the parsing method is parse_re_alt(). The prefix “recognize” has been changed to “parse” because it builds a parse tree fragment. Finally, the suffix “alt” has been added. This indicates that this method builds “K_alt” nodes of the parse tree. Lines 2 and 3 build a ConsCellNode to represent the “K_alt” operation. However, if it turns out that the “K_alt” has only one argument (which means that it does not really have a function), then it is not returned. Instead, its child is returned. This technique is called node filtering. This decision is made in lines 10 through 12 of the method.

```java
1. static ConsCellNode parse_re_alt() throws IOException {
2.    ConsCellNode root = new ConsCellNode(parse_simple_re_cat(),
3.        K_op_type.K_alt);
4.    ConsCellNode last = root;

5.    while (curr_type == TokenType.VERT) {
6.        match(TokenType.VERT);
7.        last.setNext(new ConsCellNode(parse_simple_re_cat(),
8.            K_op_type.K_alt));
9.        last = last.getNext();
10.       if (root.getNext() != null)
11.          return root;
12.       else return (ConsCellNode)root.getFirst();
}
```

The fragment is returned as an instance of an ConsCellNode. Notice that Line 12 has a downcast operation to ConsCellNode. This is necessary because the getFirst() method returns the data type of AbstractNode as depicted in Figure 3. The source code below defines part of the structure of a ConsCellNode. It defines the fields, constructors, setters, and getters. You will also need to define a toString() method and a method to print the tree structure. You will also need to define AbstractNode and AtomicNode.

```java
public class ConsCellNode extends AbstractNode {
    AbstractNode first = null;
    ConsCellNode next = null;
    K_op_type k_op;

    public ConsCellNode() { super();}
    public ConsCellNode(AbstractNode first, K_op_type k_op) {
        this();
        this.first = first;
        this.k_op = k_op;
    }

    public AbstractNode getFirst() { return first;
```
public ConsCellNode getNext () { return next;}
public void setCar(AbstractNode nd) { first = nd;}
public void setNext(ConsCellNode node) { next = node;}
public void set_k_op (K_op_type k_op) { this.k_op = k_op;}
public K_op_type get_k_op() { return k_op;}

Finally, the code segment below defines the \texttt{K\_op\_type} and the associated operator values.

public enum K_op_type { K\_cat, // concatenation
  K\_alt, // alternation
  K\_star, // Kleene star quantifier
  K\_plus, // Kleene plus quantifier
  K\_qmark, // question quantifier
  K\_self, // exactly one implicit quantifier
  K\_pos\_set, //
  K\_neg\_set, // complement
  K\_paren, // parentheses
  K\_period, // match anything
  K\_char,
  ERROR;
}

References
