1 CSE340 Spring 2021 Project 2: Generating a lexical analyzer automatically!

Due: **Friday, March 5, 2021 by 11:59 pm MST**

2 Introduction

I will start with a high-level description of the project in this section. In subsequent sections, I will go into a detailed description of the requirements and how to go about implementing a solution that satisfies them.

The goal of this project is to implement a lexical analyzer automatically for any list of tokens that are specified using regular expressions. The input to your program will have two parts:

1. The first part of the input is a list of tokens separated by commas and terminated with the `#` (hash) symbol. Each token in the list consists of a token name and a token description. The token description is a regular expression for the token. The list has the following form:

   ```
   t1_name t1_description , t2_name t2_description , ... , tk_name tk_description #
   ```

2. The second part of the input is a *input string* of letters and digits and space characters.

Your program will read the list of tokens, represent them internally in appropriate data structures, and then do lexical analysis on the *input string* to break it down into a sequence of tokens and lexeme pairs from the provided list of tokens. The output of the program will be this sequence of tokens and lexemes. If during the processing of the input string, your program cannot identify a token to match from the list, it outputs ERROR and stops.

If the input to the program has a syntax error, then your program should not do any lexical analysis of the input string and instead it should output a syntax error message and exits.

More details about the input format and the expected output of your program are given in what follows.

The remainder of this document is organized as follows.

1. The second section describes the input format.
2. The third section describes the expected output.
3. The fourth section describes the requirements on your solution and the grading criteria.
4. The fifth and largest section is a detailed explanation how to go about implementing a solution. This section also includes a description of regular expressions.

3 Input Format

The input of your program is specified by the following context-free grammar:

```
input            →   tokens_section INPUT_TEXT
tokens_section   →   token_list HASH
token_list       →   token
token_list       →   token COMMA token_list
token            →   ID expr
expr             →   CHAR
expr             →   LPAREN expr RPAREN DOT LPAREN expr RPAREN
expr             →   LPAREN expr RPAREN OR LPAREN expr RPAREN
expr             →   LPAREN expr RPAREN STAR
expr             →   UNDERSCORE
```

Where

```
CHAR        =   a | b | ... | z | A | B | ... | Z  | 0 |  1 | ... | 9
LETTER      =   a | b | ... | z | A | B | ... | Z
SPACE       =  ' ' | \n | \t
INPUT_TEXT  =  "  (CHAR | SPACE)*  "
COMMA       = ','
LPAREN      = '('
RPAREN      = ')'
STAR        = '*'
DOT         =  '.'
OR          =  '|'
UNDERSCORE  =  '_'
ID          =  LETTER . CHAR*
```

Like the first project, you are provided with a lexer to read the input, but you are asked to write the parser. Compared to the first project, writing the parser should be easy.

In the description of regular expressions, UNDERSCORE represents epsilon (more about that later).

3.1 Examples

The following are examples of input.

1. ```
   t1 (a)|(b) , t2 (a).((a)*) , t3 (((a)|(b))*).(c) #
   "a aa bb aab"
   ```

   This input specifies three tokens `t1`, `t2`, and `t3` and an INPUT_TEXT “a aa bb aab”.
   ```

2. ```
   t1 (a)|(b) , t2 ((c)*).(b) #
   "a aa bb aad aa"
   ```
   This input specifies two tokens `t1`, `t2`, and an INPUT_TEXT "a aa bb aad aa".

3. ```
   t1 (a)|(b) , t2 (c).((a)*) , t3 (((a)|(b))*).(((c)|(d))*)#
   "aaabbcaaaa"
   ```
   This input specifies three tokens `t1`, `t2` and texttt3 and an INPUT_TEXT "aaabbcaaaa".

4. ```
   tok (a).((b)|(_)) , toktok (a)|(_), tiktok ((a).(a)).(a) #
   "aaabbcaaaa"
   ```
   This input specifies three tokens whose names are `tok`, `toktok`, and `tiktok` and an INPUT_TEXT "aaabbcaaaa". Recall that in the description of regular expressions, underscore represents epsilon, so the regular expressions for the token `tok` is equivalent to $(a).((b)|(\epsilon))$ and the regular expressions for the token `toktok` is equivalent to $(a)|(\epsilon)$

Note 1 The code we provided breaks down the input to the program into tokens like ID, LPAREN, RPAREN and so on. Like the first project, to read the input, the code we provide has an object called `lexer` and a function `GetToken()` used in reading the input according to the fixed list of tokens for the input to the program. Your program will then have to break down the INPUT_TEXT string into a sequence of tokens according to the list of token in the input to the program. In order not to confuse the function that breaks down the INPUT_TEXT from the function `GetToken()` in the code we provided, you should call your function something else like `my_GetToken()`

4 Output Format

The output will be either SYNTAX ERROR if the input has a syntax error or a message indicating that one or more of the tokens have expressions that are not valid (see below) or a sequence of tokens and their corresponding lexemes according to the list of tokens provided if there are no errors. More specifically, the following are the output requirements.

1. if the input to your program is not in the correct format (not according to the grammar in Section 2), your parser should output SYNTAX ERROR and nothing else, so you should make sure not to print anything before the complete parsing of the input is completed.

2. if the input to your program is syntactically correct, then there are two cases to consider:

   (a) If any of the regular expressions of the tokens in the list of tokens in the input to your program can generate the empty string, then your program should output

   ```
   EPSILON IS NOOOOOT A TOKEN !!! tok_1 tok_2 ... tok_k
   ```

   where `tok_1`, `tok_2`, `...`, `tok_k` is the list of tokens whose regular expressions can generate the empty string.

   (b) If there is no syntax error and none of the expressions of the tokens can generate the empty string, your program should do lexical analysis on `INPUT_TEXT` and produce a sequence of tokens and lexemes in `INPUT_TEXT` according to the list of tokens specified
   ```

in the input to your program. Each token and lexeme should be printed on a separate line. The output on a given line will be of the form

```
t ,  "lexeme"
```

where t is the name of a token and lexeme is the actual lexeme for the token t. If during lexical analysis of `INPUT_TEXT`, a syntax error is encountered then ERROR is printed on a separate line and the program exits.

In doing lexical analysis for `INPUT_TEXT`, `SPACE` is treated as a separator and is otherwise ignored.

Note 2 The `my_GetToken()` that you will write is a general function that takes a list of token representations and does lexical analysis according to those representations. In later sections, I explain how that can be done, so do not worry about it yet, but keep in mind that you will be writing a general `my_GetToken()` function.

Examples Each of the following examples gives an input and the corresponding expected output.

1.
```
t1 (a)|(b) , t2 ((a)*).(a) , t3 (((a)|(b))*).(((c)*).(c)) #
"a aac bbc aabc"
```

This input specifies three tokens `t1`, `t2`, and `t3` and an INPUT_TEXT "a aac bbc aabc". Since the input is in the correct format and none of the regular expressions generates epsilon, the output of your program should be the list tokens in the INPUT_TEXT:

```
t1 , "a"
t3 , "aac"
t3 , "bbc"
t3 , "aabc"
```

2.
```
t1 (a)|(b) , t2 ((a)*).(a) , t3 (((a)|(b))*).(c) #
"a aa bbc aad aa"
```

This input specifies three tokens `t1`, `t2`, and `t3` and an INPUT_TEXT "a aa bbc aad aa". Since the input is in the correct format and none of the regular expressions generates epsilon, the output of your program should be the list tokens in the INPUT_TEXT the output of the program should be

```
t1 , "a"
t2 , "aa"
t3 , "bbc"
t2 , "aa"
ERROR
```

Note that doing lexical analysis for INPUT_TEXT according to the list of tokens produces ERROR after the second `t2` token because there is no token that starts with `'d'`.

3. `t1a (a)|(b) , t2bc (a).((a)*) , t34 (((a)|(b))*).((c)|(d))#`
   `"aaabbcaaaa"`

   This input specifies three tokens whose names are `t1a`, `t2bc`, and `t34` and an input text "aaabbcaaaa". Since the input is in the correct format, the output of your program should be the list tokens in the INPUT_TEXT:

   ```
   t34 , "aaabbc"
   t2bc , "aaaa"
   ```

4. `t1 (a)|(b) , t2 ((a)*).(a) , t3 (a)*, t4 b , t5 ((a)|(b))* #`
   `"a aac bbc aabc"`

   This input specifies five tokens and an INPUT_TEXT "a aac bbc aabc". Since some of the regular expressions can generate epsilon, the output:

   ```
   EPSILON IS NOOOOOT A TOKEN !!! t3 t5
   ```

5 Requirements and Grading

You should write a program to produce the correct output for a given input as described above. You will be provided with a number of test cases. Since this is the second project, the number of test cases provided with the project will be small relative to the number of test cases provided for project 1. **In your solution, you are not allowed to use any built-in or library support for regular expressions in C/C++. This requirement will be enforced by checking your code.**

The grade is broken down as follows

1. Submission compiles and every function has comments and every file has your name: **10 points**
2. Submission does not compile or some functions have no comments or some submitted file does not have your name: **no credit for the submission**.
3. Syntax checking: **10 points** (no partial credit for this)
4. EPSILON IS NOOOOOT A TOKEN !!! error: **15 points** (grade is strictly proportional to the number of test cases that your program successfully passes)
5. Lexical analysis of `INPUT_TEXT`: **65 points** (grade is strictly proportional to the number of test cases that your program successfully passes)

The compiler and environment used is the same as for project 1, so refer to project 1 document for the information.

Note 3 If your code does not compile on the submission website, you will not receive any points, not even for documentation.

6 How to Implement a Solution

The main difficulty in coming up with a solution is to transform a given list of token names and their regular expression descriptions into a `my_GetToken()` function for the given list of tokens. This transformation will be done in three high-level steps:

1. Transform regular expressions into REGs. The goal here is to parse a regular expression description and generate a graph that represents the regular expression[1]. The generated graph will have a specific format and I will describe below how to generate it. I will call it a *regular expression graph*, or REG for short.

2. Write a function `match(r,s,p)`, where `r` is a REG , `s` is a string and `p` is a position in the string `s`. The function match will return the longest possible lexeme starting from position `p` in the string `s` that matches the regular expression of the graph `r`.

3. Write a class `my_LexicalAnalyzer(list,s)` where `list` is a list of structures of the form {`token_name`, *reg_pointer*} and `s` is an input string. `my_LexicalAnalyzer` stores the list of structures and keeps track of the part of the input string that has been processed. The class `my_LexicalAnalyzer` has a method `my_GetToken()`. For every call of `my_GetToken()`, `match(r,s,p)` is called for every REG r in the list starting from the current position `p` maintained in `my_LexicalAnalyzer`. `my_GetToken()` returns the token with the longest matching prefix together with its lexeme and updates the current position. If the longest matching prefix matches more than one token, the matched token that is listed first in the list of tokens is returned.

In what follows I describe how a regular expression description can be transformed into a REG and how to implement the function `match(r,s,p)`. But first, I will give an overview of regular expressions and the sets of strings they represent.

6.1 Set of Strings Represented by Regular Expressions

A regular expression is a compact representation of a set, possibly infinite, of strings. For a given regular expression, we say that expression can *generate* a string if the string is in set that is represented by the regular expression. We start with a general description, then we give examples.

6.1.1 General description

We start with the simple expressions (the base cases)

- (**One-character strings**) The regular expression `a` represents the set of strings $\{a\}$, that is the set consisting of only the string `"a"`.

- (**Epsilon**) The regular expression _ represents the set of strings $\{\epsilon\}$, that is the set consisting of only the string ϵ (which is the empty string).

For the inductive step (recursion of your parser), there are four cases:

[1]The graph is a representation of a non-deterministic finite state automaton

- (**Parentheses**) If `R` is a regular expression, the regular expression `(R)` represents the same set of strings that `R` represents. The parentheses are used for grouping and to facilitate parsing and do not have a meaning otherwise.
- (**Concatenation**) If `R1` and `R2` are regular expressions that represents sets of strings `S1` and `S2` respectively, then `(R1).(R2)` represents a new set of strings that can be obtained by concatenating one string from `S1` with one string from `S2` (order matters).
- (**Union**) If `R1` and `R2` are regular expressions that represents sets of strings `S1` and `S2` respectively, then `(R1)|(R2)` represents the union of the two sets of strings `S1` and `S2`.
- (**Kleene star**) The last case is the most interesting because it allows us unlimited number of repetition. If `R` is a regular expression that represents the set of strings `S`, then `(R)*` represents the set of strings that can be obtained by concatenating any number of strings from `S`, including zero strings (which gives us epsilon).

6.1.2 Examples

1. The set of strings represented by `a` is
$$\{a\}$$
2. The set of strings represented by `b` is
$$\{b\}$$
3. The set of strings represented by `(a)|(b)` is
$$\{a, b\}$$
4. The set of strings represented by `((a)|(b)).(c)` is
$$\{ac, bc\}$$
5. The set of strings represented by `((a)|(b)).((c)|(d))` is
$$\{ac, ad, bc, bd\}$$
6. The set of strings represented by `((c)|(d)).((a)|(b))` is
$$\{ca, cb, da, db\}$$
7. The set of strings represented by `(a)*` is
$$\{\epsilon, a, aa, aaa, aaaa, \ldots\}$$
8. The set of strings represented by `(b)*` is
$$\{\epsilon, b, bb, bbb, bbbb, \ldots\}$$

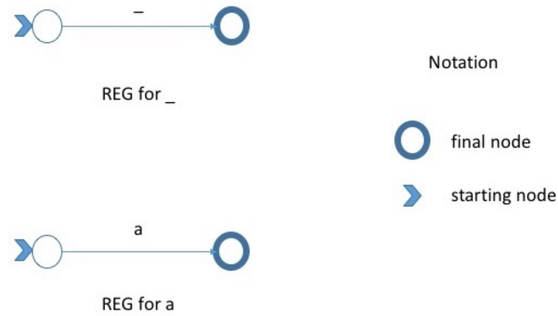

Figure 1: Regular expressions graphs for the base cases

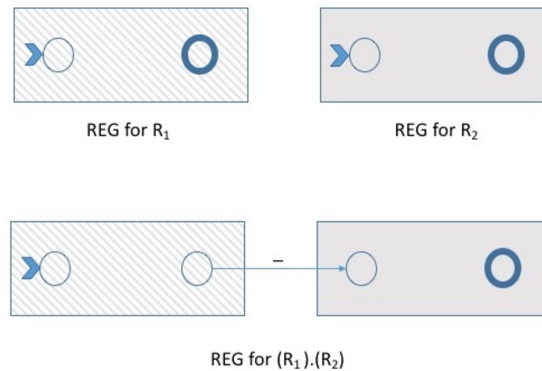

Figure 2: Regular expression graph for the an expression obtained using the dot operator

9. The set of strings represented by `(a)|((b)*)` is

$$\{a,\ \epsilon,\ b,\ bb,\ bbb,\ bbbb,\ \ldots\}$$

10. The set of strings represented by `((a)*)|((b)*)` is

$$\{\epsilon, a, b, aa, bb, aaa, bbb, aaaa, bbbb, \ldots\}$$

11. The set of strings represented by `((a)|(b))*` is

$$\{\epsilon,\ a,\ b,\ aa,\ ab,\ ba,\ bb,\ aaa,\ aab,\ aba,\ abb,\ baa,\ bab,\ bba,\ bbb,\ \ldots\}$$

6.2 Constructing REGs

The construction of REGs is done recursively. The construction we use is called Thompson's construction. Each REG has a one `start` node and one `accept` node. For the base cases of epsilon and `a`, where `a` is a character of the alphabet, the REGs are shown in Figure 1. For the recursive cases, the constructions are shown in Figures 2, 3, and 4. An example REG for the regular expression `((a)*).((b)*)` is shown in Figure 5.

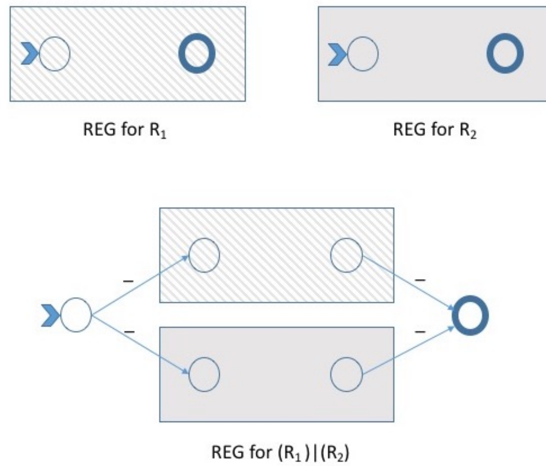

Figure 3: Regular expression graph for the an expression obtained using the or operator

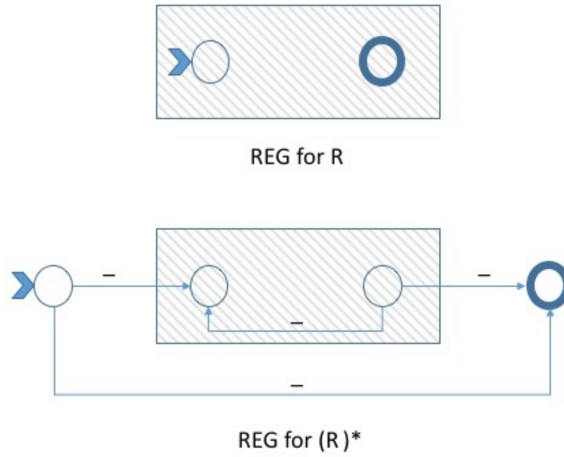

Figure 4: Regular expression graph for the an expression obtained using the star operator

6.2.1 Data Structures and Code for REGs

In the construction of REGs, every node has at most two outgoing arrows. This will allow us to use a simple representation of a REG node.

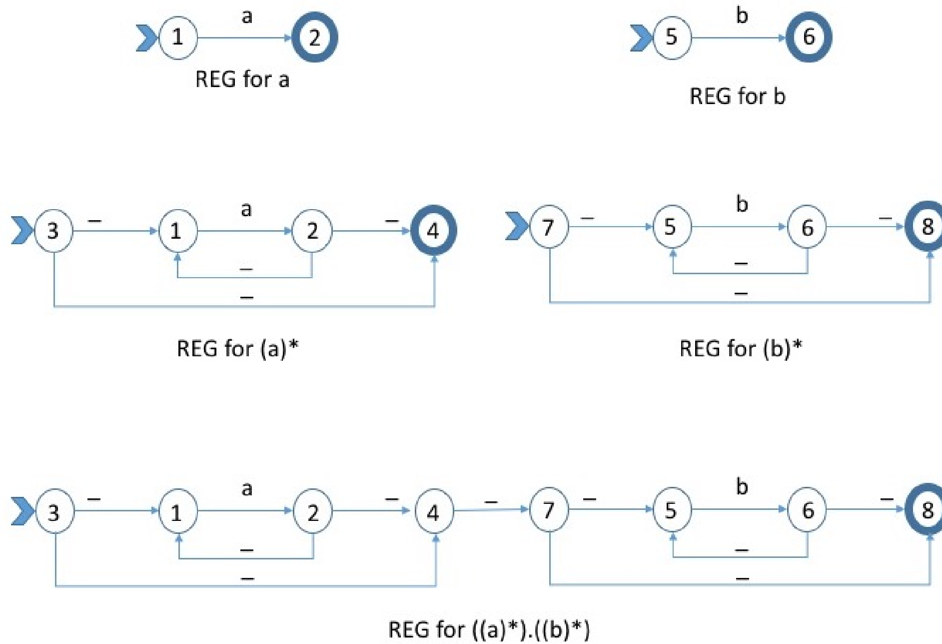

Figure 5: Regular expression graph for the an expression obtained using concatenation and star operators

```
 struct REG_node {
  struct REG_node * first_neighbor;
  char first_label;
  struct REG_node * second_neighbor;
  char second_label;
}
```

In the representation, `first_neighbor` is the first node pointed to by a node and `second_neighbor` is the second node pointed to by a node. `first_label` and `second_label` are the labels of the arrows from the node to its neighbors. If a node has only one neighbor, then `second_neighbor` will be NULL. If a node has no neighbors, then both `first_neighbor` and `second_neighbor` will be NULL.

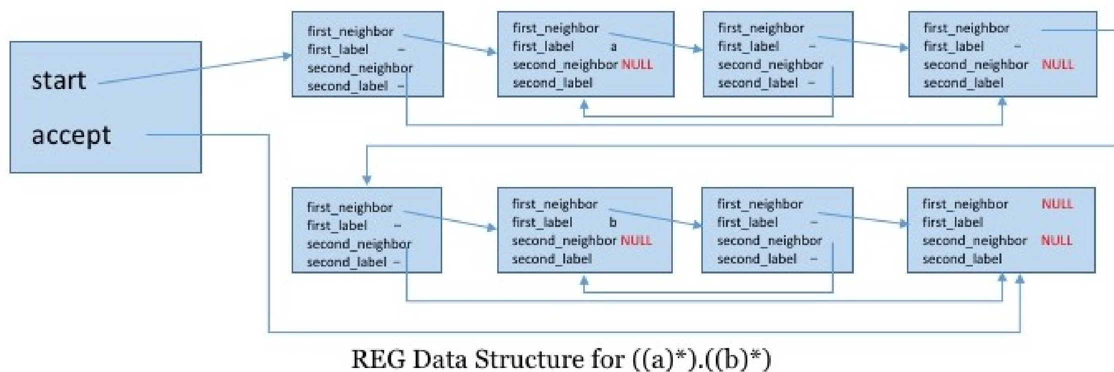

Figure 6: Data structure representation for the REG of `((a)*).((b*))`

```
struct REG {
   struct REG_node * start;
   struct REG_node * accept;
}
```

In the your parser, you should write a function `parse_expr()` that parses the regular expressions returns the REG of the regular expression that is parsed. The construction of REGs is done recursively. An outline of the process is shown on the next page.

```
struct REG * parse_expr()
{
   // if expression is UNDERSCORE or a CHAR, say 'a' for example
   // create a REG for the expression and return a pointer to it
   // (see Figure 1, for how the REG looks like)

   // if expression is (R1).(R2)
   //
   //       the program will call parse_expr() twice, once
   //       to parse R1 and once to parse R2
   //
   //       Each of the two calls will return a REG, say they are
   //       r1 and r2
   //
   //       construct a new REG r for (R1).(R2) using the
   //       two REGs r1 and r2
   //       (see Figure 2 for how the two REGs are combined)
   //
   //       return r
   //
   // the cases for (R1)|(R2) and (R)* are similar and
```

```
    // are omitted from the description
}
```

6.2.2 Detailed Examples for REG Construction

I consider the regular expression `((a)*).((b)*)` and explain step by step how its REG is constructed (Figure 5).

When parsing `((a)*).((b)*)`, the first expression to be fully parsed and its REG is constructed is `a` (Figure 1). In Figure 5, the nodes for the REG of the regular expression `a` have numbers 1 and 2 to indicate that they are the first two nodes to be created.

The second expression to be fully parsed and its REG constructed when parsing `((a)*).((b)*)` is `(a)*`. The REG for `(a)*` is obtained from the REG for the regular expression `a` by adding two more nodes (3 and 4) and adding the appropriate arrows as described in the general case in Figure 4. The starting node for the REG of `(a)*` is the newly created node 3 and the accepting node is the newly created node 4.

The third regular expression to be fully parsed while parsing `((a)*).((b)*)` is the regular expression `b`. The REG for regular expression `b` is constructed as shown in Figure 1. The nodes for this REG are numbered 5 and 6.

The fourth regular expression to be fully parsed while parsing `((a)*).((b)*)` is `(b)*`. The REG for `(b)*` is obtained from the REG for the regular expression `b` by adding two more nodes (7 and 8) and adding the appropriate arrows as described in the general case in Figure 4. The starting node for the REG of `(b)*` is the newly created node 7 and the accepting node is the newly created node 8.

Finally, the last regular expression to be fully parsed is the regular expression `((a)*).((b)*)`. The REG of `((a)*).((b)*)` is obtained from the REGs of `(a)*` and `(b)*` by creating a new REG whose initial node is node 3 and whose accepting node is node 8 and adding an arrow from node 4 (the accepting node of the REG of `(a)*`) to node 7 (the initial node for the REG of `(b)*`).

Another example for the REG of `(((a)*).((b).(b)))|((a)*)` is shown in Figures 8 and 9. In the next section, I will use REG of `(((a)*).((b).(b)))|((a)*)` to illustrate how `match(r,s,p)` can be implemented.

6.3 Implementing `match(r,s,p)`

Given an REG `r`, a string `s` and a position `p` in the string `s`, we would like to determine the longest possible lexeme that matches the regular expression for `r`.

As you will see in CSE355, a string `w` is in L(`R`) for a regular expression `R` with REG `r` if and only if there is a path from the starting node of `r` to the accepting node of `r` such that `w` is equal to the concatenation of all labels of the edges along the path. I will not go into the details of the equivalence in this document. I will describe how to find the longest possible substring `w` of `s` starting at position `p` such that there is a path from the starting node of `r` to the accepting node of `r` that can be labeled with `w`.

To implement `match(r,s,p)`, we need to be able to determine for a given input character `a` and a set of nodes `S` the set of nodes that can be reached from nodes in `S` by *consuming* `a`. To consume `a` we can traverse any number of edges labeled '_', traverse one edge labeled `a`, then traverse any number of edges labeled '_'. To match one character, you will implement a function called `Match_One_Char()` shown in Figure 7. For a given character `c` and a given set of nodes

```
Match_One_Char(set_of_nodes S, char c) returns set_of_nodes
{

      // 1. find all nodes that can be reached from S by consuming c
      //
      //    S’ = empty set
      //    for every node n in S
      //        if  ( (there is an edge from n to m labeled with c) &&
      //              ( m is not in S’) ) {
      //              add m to S’
      //        }
      //
      //     if (S’ is empty)
      //         return empty set
      //
      //    At this point, S’ is not empty and it contains the nodes that
      //    can be reached from S by consuming the character c directly
      //
      // 2. find all nodes that can be reached from the resulting
      //    set S’ by consuming no input
      //
      //     changed = true
      //     S’’ = empty set
      //     while (changed)  {
      //        changed = false
      //        for every node n in S’ {
      //             add n to S’’
      //             for ever neighbor m of n {
      //                 if  ( (the edge from n to m labeled with ’_’) &&
      //                       ( m is not in S’’) )
      //                     add m to S’’
      //             }
      //        }
      //        if (S’ not equal to S’’) {
      //             changed = true;
      //             S’ = S’’
      //             S’’ = empty set
      //         }
      //     }
      //
      //     at this point the set S’ contains all nodes that can be reached
      //     from S by first consuming C, then traversing 0 or more epsilon
      //     edges
      //
      //     return S’
 }
```

Figure 7: Pseudocode for matching one character

.

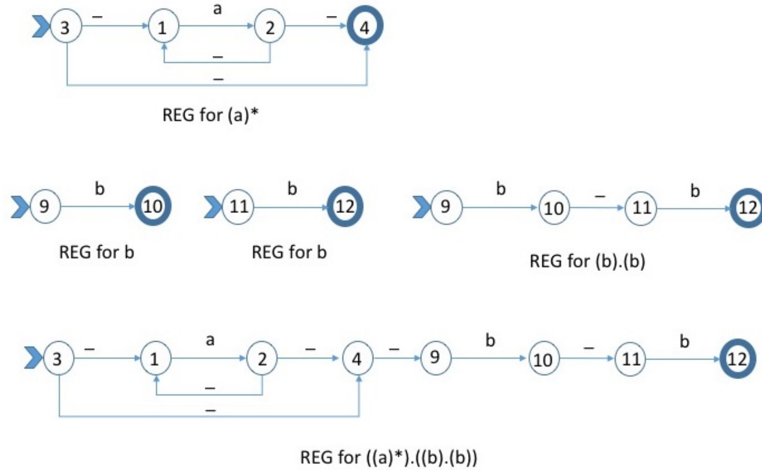

Figure 8: Regular expression graph `((a)*).((b).(b))`

`S`, `Match_One_Char()` will find all the nodes that can be reached from `S` by consuming the single character `c`.

In order to match a whole string, we need to match the characters of the strings one after another. At each step, the solution will keep track of the set of nodes `S` that can be reached by consuming the prefix of the input string that has been processed so far.

To implement `match(r,s,p)`, we start with the set of nodes that can be reached from the starting node of `r` by consuming no input. Then we repeatedly call `Match_One_Char()` for successive characters of the string `s` starting from position `p` until the returned set of nodes `S` is empty or we run out of input. If at any point during the repeated calls to `Match_One_Char()` the set `S` of nodes contains the accepting node, we note the fact that the prefix of string `s` starting from position `p` up to the current position is matching. At the end of the calls to `Match_One_Char()` when `S` is empty or the end of input is reached, the last matched prefix is the one returned by `match(r,s,p)`. If none of the prefixes are matched, then there is no match for `r` in `s` starting at `p`.

Note 4

- The algorithms given above are not the most efficient, but they are probably the simplest to implement the matching functions.
- The algorithm uses sets, so you need to have a representation for a set of nodes and to do operations on sets of nodes.

6.4 Detailed Example for Implementing `match(r,s,p)`

In this section, I illustrate the steps of an execution of `match(r,s,p)` on the REG of

```
(((a)*).((b).(b)))|((a)*)
```

shown in Figure 9. The input string we will consider is the string `s = "aaba"` and the initial position is `p = 0`.

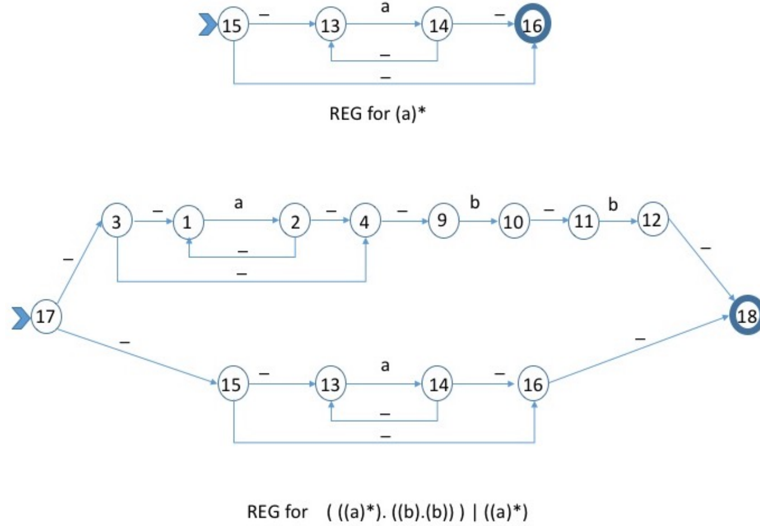

Figure 9: Regular expression graph `(((a)*).((b).(b)))|((a)*)`

1. Initially, the set of states that can reached by consuming no input starting from node 17 is

   $$S_0 = \{17, 3, 1, 4, 9, 15, 13, 16, \mathbf{18}\}$$

   Note that S_0 contains node **18** which means that the empty string is a matching prefix. This means that this expression should result in a “`EPSILON IS NOOOOOT A TOKEN !!!`” error if it is used in a token specification.

2. Consuming `a`.

   The set of states that can be reached by consuming `a` starting from S_0 is

   $$S_1 = \{2, 14\}$$

   The set of states that can be reached by consuming no input starting from S_1 is

   $$S_{1_} = \{2, 1, 4, 9, 14, 13, 16, \mathbf{18}\}$$

   Note that $S_{1_}$ contains node **18**, which means that the prefix `"a"` is a matching prefix.

3. Consuming `a`

   The set of states that can be reached by consuming `a` starting from $S_{1_}$ is

   $$S_2 = \{2, 14\}$$

   The set of states that can be reached by consuming no input starting from S_2 is

$$S_{2_} = \{2, 1, 4, 9, 14, 13, 16, \mathbf{18}\}$$

Note that $S_{2_}$ contains node **18**, which means that the prefix `"aa"` is a matching prefix.

4. Consuming `b`

   The set of states that can be reached by consuming `b` starting from $S_{2_}$ is

   $$S_3 = \{10\}$$

   The set of states that can be reached by consuming no input starting from S_3 is

   $$S_{3_} = \{10, 11\}$$

   Note that $S_{3_}$ does not contain node **18** which means that `"aab"` is not a matching prefix, but is still a viable prefix, which means that there is hope we can read more characters that will turn it into a matching prefix.

5. Consuming `a`

   The set of states that can be reached by consuming `a` starting from $S_{3_}$ is

   $$S_4 = \{\}$$

   Since S_4 is empty, `"aaba"` is not viable and we stop.

The longest matching prefix is `aa`. This is the lexeme that is returned. Note that the second call to `match(r,s,p)` starting after `"aa"` will return ERROR.