CSE 340 Fall 2023 Project 1:
Generating a lexical analyzer automatically!!

Assigned: Friday, 25 August 2023.

Due: **Thursday, September 21, 2023 by 11:59 pm** on Gradescope.

1 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 (if you do not know what a regular expression is, do not worry, it will be defined in this document). The input to your program will have two parts:

1. The first part of the input is a list of token definitions separated by commas and terminated with the `#` (hash) symbol. Each token definition 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 an *input string* which is a sequence of letters and digits and space characters.

Your program will read the list of token definitions, represent the list 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 find a token in the list that matches part of the remaining input (cannot find a matching prefix), 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. Your program should also output error messages if the same name is used for multiple tokens or if one of the regular expressions generates the empty string. More specifics about the input format and expected output are given in Sections 2 and 3.

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 of how to go about implementing a solution. This section also includes a description of regular expressions.
5. The last section includes general instructions that apply to all programming assignments in this class.

2 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       =  ','
DOT         =  '.'
HASH        =  '#'
ID          =  LETTER . CHAR*
LPAREN      =  '('
OR          =  '|'
RPAREN      =  ')'
STAR        =  '*'
UNDERSCORE  =  '_'
```

You are provided with a lexer to read the input, but you are asked to write the parser. In the description of regular expressions, UNDERSCORE represents epsilon (more about that later).

2.1 Examples

The following are four 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 `t3` 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 $(a)|(\epsilon)$ (if this is not clear, it should become clear when you read the section on regular expressions).

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

3 Output Format

The output will be either a syntax error message, 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 (I start by describing the requirements, but below I give examples to clarify them).

1. (**Syntax Checking: 15 points**, but no partial credit) If the input to your program is not in the correct format (not according to the grammar in Section 2), there are two cases to consider

(a) If the first syntax error that your program encounters occurs while parsing a regular expression, then your program should output the following and nothing else and stops

```
SYNTAX ERROR IN EXPRESSION OF token_name
```

where `token_name` is the name of the token that your program parses before parsing the regular expression (such a name must exist because if it is missing, your program must have encountered a syntax error before parsing the expression).

(b) If the first syntax error that your program encounters in not in the regular expression of a token, then your program should output the following and nothing else

```
SNYTAX ERORR
```

You should make sure that your program doesn't print anything before it completely parses the input. No partial credit on syntax checking means that you should pass all test cases on the submission site in order to get the 15 points. If your program fails any of the test cases for this category, you will get no credit for syntax checking. Only syntax checking has no partial credit.

2. (**Semantic Checking: 15 points**) If the input to your program is syntactically correct (no syntax error), but some token name is declared more than once, then for every instance of the token name, except for the first instance, your program should output

```
Line line_number1: token_name already declared on line line_number2
```

Where `line_number1` is the line in which the duplicate `token_name` appears and `line_number2` is the line in which the first appearance of `token_name` occurs.

3. (**Epsilon error: 20 points**) If there are no syntax errors and all tokens have distinct names, then, if any regular expressions of the tokens in the list of tokens can generate the empty string, then your program should output

```
EPSILON IS NOOOOOOOT A TOKEN !!! token_name1 token_name2 ... token_namek
```

where `token_name1`, `token_name2`, `...`, `token_namek` is the list of names of the tokens whose regular expressions can generate the empty string.

4. (**Lexical Analysis : 60 points**) 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 lexical 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 (we already discussed what that means in class).

The total of these points is 110 points. In addition, there are 10 points for programs that compile correctly and are documented (it is very important that you read the "requirements and grading" below). So, the overall total is 120 points which will count as 12% of the course grade. Remember that if you do not document your code, you will not get any credit for the submission (see below).

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 you can implement this functionality, 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"
```

   Notice how `t3` is defined across multiple lines. Your program will not be aware of this because it will use the provided `GetToken()` function which ignores space.

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'`, so when `my_GetToken()` is called for the fifth time, it will not be able to match any of the tokens in the list, so an `ERROR` output is produced and your program exits.

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 the regular expressions of `t3` and `t5` can generate epsilon, the output is:

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

4 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 but these test cases are not meant to be exhaustive. They are only meant to complement the specification document (this document). **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 for a **total of 120 points**

1. Submission compiles and code properly documented **10 points**. To get the 10 points,
   - the submission must compile **and**
   - every function must have comments **and**
   - every submitted file must have your name.
2. Submission does not compile or some functions have no comments or submitted file does not have your name: **no credit for the submission**.
3. Syntax checking: **15 points**: There is no partial credit for this part.
4. Semantic checking: **15 points** (grade is strictly proportional to the number of test cases that your program successfully passes)
5. "EPSILON IS NOOOOOOOT A TOKEN !!!" error cases: **20 points** (grade is strictly proportional to the number of test cases that your program successfully passes)
6. Lexical analysis of `INPUT_TEXT`: **60 points** (grade is strictly proportional to the number of test cases that your program successfully passes)

Refer to sections 6 and 7 below for information about the compiler that we will use and the execution environment as well as for general instructions that are relevant for all programming assignments.

Note 3 If your code does not compile on the submission website, you will not receive any points, not even for documentation. Do not wait until the last minute to submit because there can be unexpected issues when you submit especially if you are not developing your solution in an environment that is compatible with the environment of the submission website.

5 How to Implement a Solution

The parser for this project is relatively simple, but this is the first time you write a parser. So, it is important that you finish your parser completely before you attempt to do anything else. You should make sure that the parser is correctly parsing and producing syntax error messages when needed.

The main difficulty in the project is in transforming 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 a new position `p'` such that the substring of `s` between `p` and `p'` is 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 by updating a variable `p` which is the position of the next character in the input string that has not 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 value of the current position `p`. 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.

5.1 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 s if the string s is in set that is represented by the regular expression. We start with a general description, then we give examples.

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

5.1.1 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:

- (**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 the 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 provides notation for an unlimited number of repetitions. 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).

5.1.2 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
$$\{ac, ad, bc, bd\}$$

   This requires some explanation. the set of strings represented by `((a)|(b))` is $\{a, b\}$and the set of strings represented by `((c)|(d))` is $\{c, d\}$, so the set of strings represented by `((a)|(b)).((c)|(d))`consists of strings that can be obtained by taking one string from the set $\{a, b\}$ and one string from the set $\{c, d\}$ and concatenating them together. The possibilities are

$$\{ac, ad, bc, bd\}$$

6. The set of strings represented by `((c)|(d)).((a)|(b))` is

   $$\{ca, cb, da, db\}$$

   Notice how this set is different from that of the previous example; order matters for concatenation!

7. The set of strings represented by `(a)*` is

   $$\{\epsilon, a, aa, aaa, aaaa, \ldots\}$$

8. The set of strings represented by `(b)*` is

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

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

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

    This also requires some explanation. A string is in the set of strings represented by `((a)|(b))*` if it is the concatenation of zero or more strings from the set of strings represented by (a)|(b) (the set $\{a, b\}$), so any string consisting of a sequence of zero or more a's and b's is in the set of strings represented by `((a)|(b))*`.

5.2 Constructing REGs

Recall that we want to construct graphs that represent the regular expressions. Those graphs are called REGs (regular expression graphs). The construction of REGs is done recursively. The construction we use is called Thompson's construction. Each REG has a exactly one `start` node and exactly 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.

5.2.1 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.

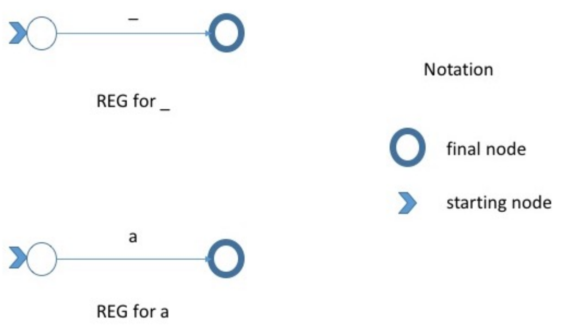

Figure 1: Regular expressions graphs for the base cases

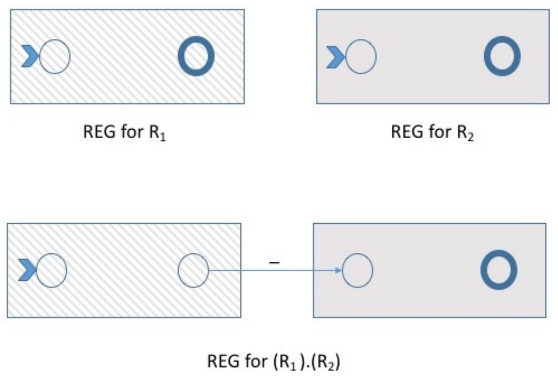

Figure 2: Regular expression graph for the an expression obtained using the dot operator

```
 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 REG node and `second_neighbor` is the second node pointed to by a REG 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.

An alternative representation uses a vector of neighbors instead two specific neighbors. That would make it easier to iterate over all neighbors of a REG node and avoid checking if the node has zero, one or two neighbors.

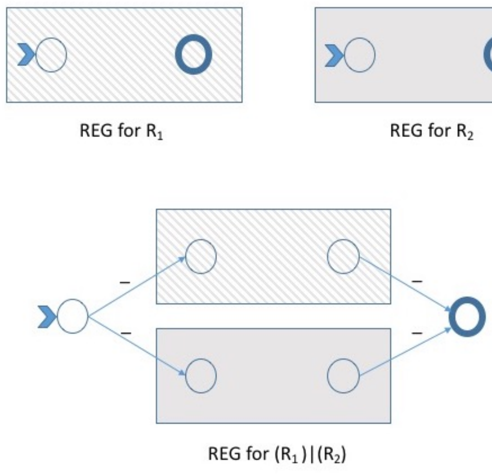

Figure 3: Regular expression graph for the an expression obtained using the or operator

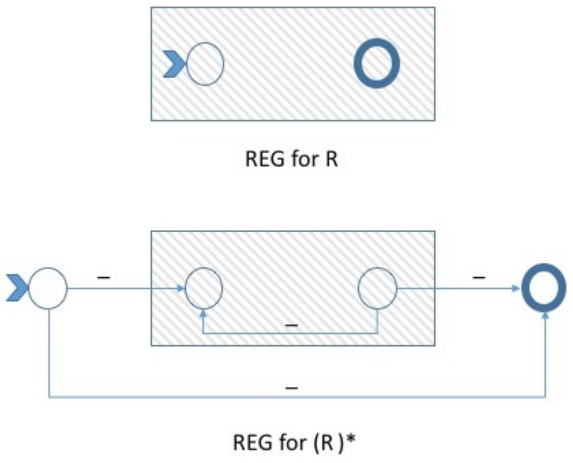

Figure 4: Regular expression graph for the an expression obtained using the star operator

The REG graph itself is represented as a structure with two pointers to two REG nodes:

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

The first pointer points to the (unique) `start` node and the second pointer points to the (unique) `accept` node. Later, when we discuss the `match()` function, we will explain what these nodes are used for.

In your parser, you should write a function `parse_expr()` that parses a regular expression and 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.

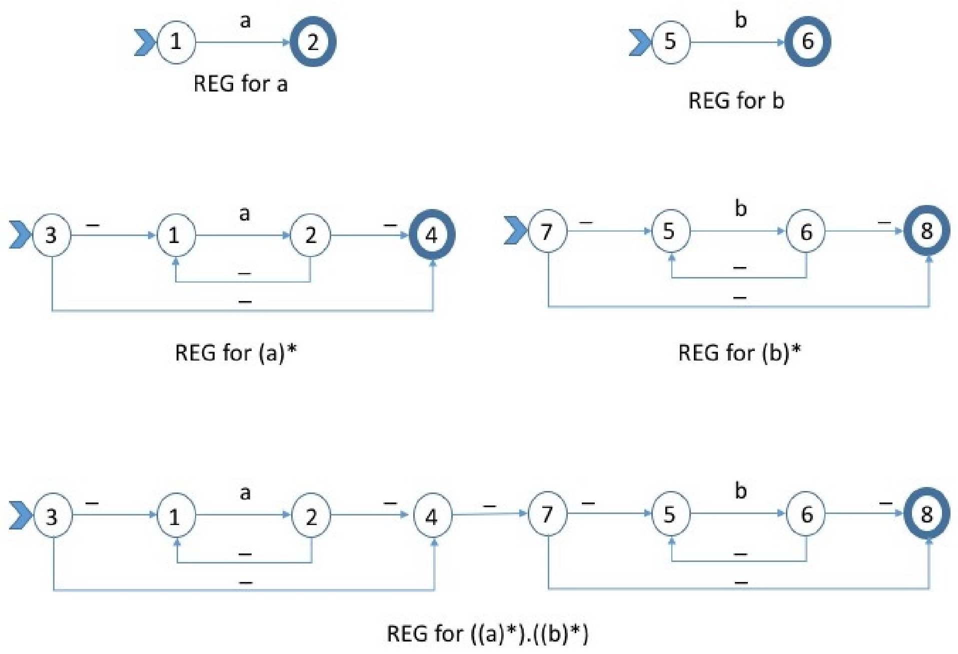

Figure 5: Regular expression graph for the an expression obtained using concatenation and star operators

```
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
```

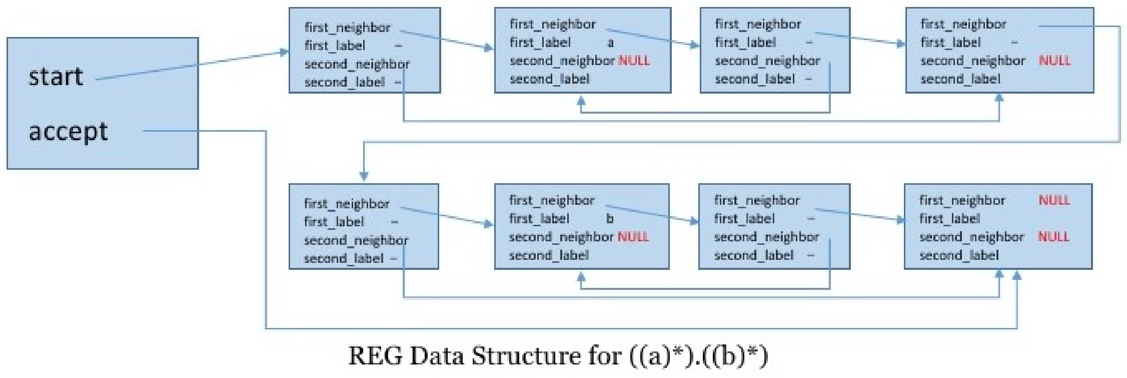

Figure 6: Data structure representation for the REG of `((a)*).((b*))`

```
}
```

5.2.2 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 accept 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 accept 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 start node is node 3 and whose accept node is node 8 and adding an arrow from node 4 (the accept node of the REG of `(a)*`) to node 7 (the start 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.

```
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 labeled c from n to a node m) &&
      //              ( 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 is 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

.

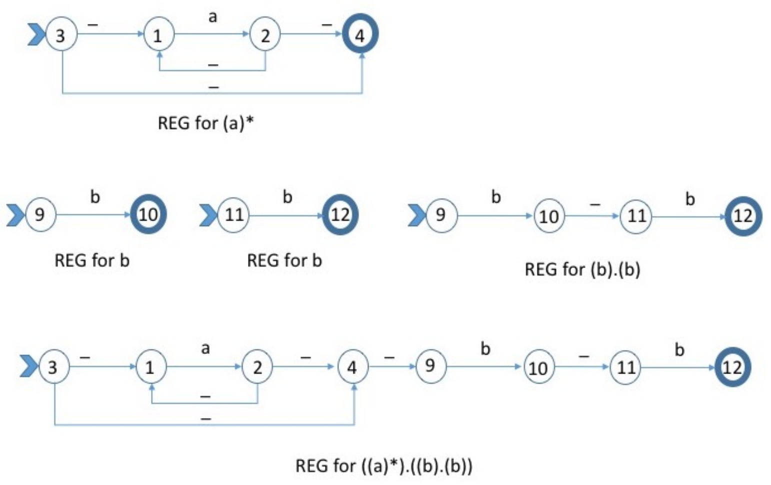

Figure 8: Regular expression graph `((a)*).((b).(b))`

5.3 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 `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`.

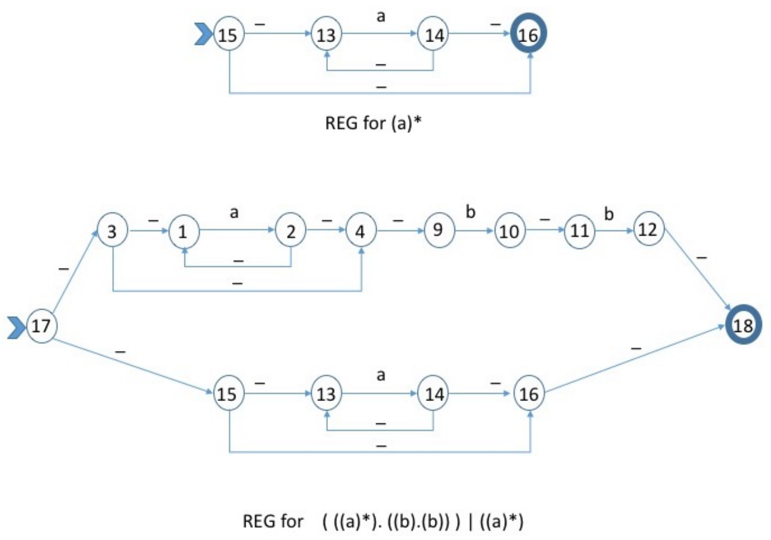

Figure 9: Regular expression graph `(((a)*).((b).(b)))|((a)*)`

Note 4

- The algorithm given above is not efficient, but it is probably the simplest to implement the matching functions. More efficient algorithms would require marking the nodes of the graphs, using a stack, and erasing the markings when match is done (think about it!).
- The algorithm uses sets, so you need to have a representation for a set of nodes and to do operations on sets of nodes.

5.4 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`.

1. Initially, we start with the set of states that can reached by consuming no input starting from the start node of the REG, node 17 in our example. The initial set of states 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 NOOOOOOOT A TOKEN !!!`" error if it is used in a token specification.

2. Consuming `a`

To find the set of states that can be reached from S_0 by consuming `a`, we first find the set S_1 of states that can be reached by consuming `a` directly, then we find the set of states $S_{1\_}$ that can be reached from S_1 by consuming nothing (traversing 0 or more epsilon edges).

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` directly 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 it 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.

6 Instructions

ollow these steps:

- Make sure that you have read this document carefully.
- Make sure that you have read the *project 1 presentation* document carefully. It has useful illustrations about how to approach the implementation.
- Download the `lexer.cc`, `lexer.h`, `inputbuf.cc` and `inputbuf.h` files accompanying this project description and familiarize yourself with the provided functions.
- Design a solution before you start coding. It is really very important that you have a clear overall picture of what the project will require before starting coding. Deciding on data structures and how you will use them is crucial. One possible exception is the parser, which you can and should write first before the rest of the solution.
- Write your code and make sure to compile your code using g++ (4:11.2.0-1ubuntu1) on **OS is ubuntu 22.4** (Ubuntu). These are the versions used on the submission website. If you want to test your code on your personal machine, you should install a virtual machine with Ubuntu 22.4 and the correct version of g++ on it. You will need to use the `g++` command to compile your code in a terminal window. See Section 7 for more details on how to compile using GCC. **You submission will be compiled using g++ (4:11.2.0-1ubuntu1) -std=c++11 option and executed on Ubuntu 22.4. It is your responsibility to make sure that your code correctly compiles and executes as specified**, but you are free to use any IDE or text editor on any platform while developing your code as long as you compile it and test it on Ubuntu/GCC before submitting it.
- Test your code to see if it passes the provided test cases. You will need to extract the test cases from the zip file and run the provided test script `test1.sh`. See section 7 for more details.
- Submit your code on the course submission website before the deadline. You can submit as many times as you need. Make sure your code is compiled correctly on the website, if you get a compiler error, fix the problem and submit again.
- **Only the last version you submit is graded. There are no exceptions to this. This will be the case for all programming assignments.** If you submit one version and get 90

on it and then submit a later version and you get 85 on it, it is your responsibility to resubmit or activate the earlier version to make sure that it counts.

Keep in mind that

- You should use C++11, no other programming languages or versions of C++ are allowed.
- All programming assignments in this course are individual assignments. Students must complete the assignments on their own.
- You should submit your code through the submission website (to be announced), no other submission forms will be accepted.
- You should familiarize yourself with the Ubuntu environment and the GCC compiler. Programming assignments in this course might be very different from what you are used to in other classes.

7 General instructions for all programming assignments

NOTE: This section applies to all programming assignments.

You should use the instructions in the following sections to compile and test your programs for all programming assignments in this course.

7.1 Compiling your code with GCC

You should compile your programs with the GCC compilers. GCC is a collection of compilers for many programming languages. There are separate commands for compiling C and C++ programs. Use the `g++` command to compile C++ programs

Here is an example of how to compile a simple C++ program:

```
$ g++ test_program.cpp
```

If the compilation is successful, it will generate an executable file named `a.out` in the same directory (folder) as the program. You can change the executable file name by using the `-o` option. For example, if you want the executable name to be `hello.out`, you can execute

```
$ g++ test_program.cpp -o hello.out
```

To enable C++11, with `g++`, which you should do for projects in this class, use the `-std=c++11` option:

```
$ g++ -std=c++11 test_program.cpp -o hello.out
```

The following table summarizes some useful compiler options for g++:

Option	Description
`-o path`	Change the filename of the generated artifact
`-g`	Generate debugging information
`-ggdb`	Generate debugging information for use by GDB

Option	Description
`-Wall`	Enable most warning messages
`-std=c++11`	Compile C++ code using 2011 C++ standard

Compiling projects with multiple files

If your program is written in multiple source files that should be linked together, you can compile and link all files together with one command:

```
$ g++ file1.cpp file2.cpp file3.cpp
```

Or you can compile them separately and then link:

```
$ g++ -c file1.cpp
$ g++ -c file2.cpp
$ g++ -c file3.cpp
$ g++ file1.o file2.o file3.o
```

The files with the `.o` extension are object files but are not executable. They are linked together with the last statement (g++ file1.o file2.o file3.o) and the final executable will be `a.out`.

7.2 Testing your code on Ubuntu

Your programs should not explicitly open any file. You can only use the **standard input** and **standard output** in C++. The provided lexical analyzer already reads the input from standard input and you should not modify it. In C++, standard input is `std::cin` and standard output is `std::cout`. In C++, any output that your program produces should be done with `cout`. To read input from a file or produce output to a file, we use IO redirection outside the program. The following illustrates the concept.

Suppose we have an executable program `a.out`, we can run it by issuing the following command in a terminal (the dollar sign is not part of the command):

```
$ ./a.out
```

If the program expects any input, it waits for it to be typed on the keyboard and any output generated by the program will be displayed on the terminal screen.

To get the input to the program from a file, we can redirect the standard input to a file:

```
$ ./a.out < input_data.txt
```

Now, the program will not wait for keyboard input, but rather read its input from the specified file as if the file `input_data.txt` is standard input. We can redirect the output of the program as well:

```
$ ./a.out > output_file.txt
```

In this way, no output will be shown in the terminal window, but rather it will be saved to the specified file[2].

[2]Programs have access to another standard stream which is called standard error e.g. `std::cerr` in C++. Any

Finally, it's possible to do redirection for standard input and standard output simultaneously. For example,

```
$ ./a.out < input_data.txt > output_file.txt
```

will read standard input from `input_data.txt` and produces standard output to `output_file.txt`.

Now that we know how to use standard IO redirection, we are ready to test the program with test cases.

Test Cases

For a given input to your program, there is an *expected* output which is the correct output that should be produced for the given input. So, a test case is represented by two files:

- `test_name.txt`
- `test_name.txt.expected`

The input is given in `test_name.txt` and the expected output is given in `test_name.txt.expected`.

To test a program against a single test case, first we execute the program with the test input data:

```
$ ./a.out < test_name.txt > program_output.txt
```

With this command, the output generated by the program will be stored in `program_output.txt`. To see if the program generated the correct expected output, we need to compare `program_output.txt` and `test_name.txt.expected`. We do that using the `diff` command which is a command to determine differences between two files:

```
$ diff -Bw program_output.txt test_name.txt.expected
```

If the two files are the same, there should be no difference between them. The options `-Bw` tell `diff` to ignore whitespace differences between the two files. If the files are the same (ignoring the whitespace differences), we should see no output from `diff`, otherwise, `diff` will produce a report showing the differences between the two files.

We consider that the test **passed** if `diff` could not find any differences, otherwise we consider that the test **failed**.

Our grading system uses this method to test your submissions against multiple test cases. In order to avoid having to type the commands shown above for running and comparing outputs for each test case manually, we provide you with a script that automates this process. The script name is `test1.sh`. `test1.sh` will make your life easier by allowing you to test your code against multiple test cases with one command.

Here is how to use `test1.sh` to test your program:

- Store the provided test cases zip file in the same directory as your project source files
- Open a terminal window and navigate to your project directory

such output is still displayed on the terminal screen. It is possible to redirect standard error to a file as well, but we will not discuss that here

- Unzip the test archive using the `unzip` command: `bash   $ unzip tests.zip`

This will create a directory called `tests`

- Store the `test1.sh` script in your project directory as well
- Make the script executable: `bash   $ chmod +x test1.sh`
- Compile your program. The test script assumes your executable is called `a.out`
- Run the script to test your code: `bash   $ ./test1.sh`

The output of the script should be self explanatory. To test your code after you make changes, you will just perform the last two steps (compile and run `test1.sh`).