CSC 173 Project 2
Maryfrances Umeora and Carolina Lion He

Program Files Used

- rexp (contains main)
- Tree.c
- Tree.h
- RDP.c
- RDP.h
- TDP.c
- TDP.h
- Teacher Provided (from last project): LinkedList.c, LinkedList.h

How To Run My Project

In your terminal, type cd and navigate to where you stored the folder of my code.
Type make.
Type ./rexp, and the project will start running.

How The Code Runs (PLEASE READ)

Our code is separated into three distinct parts with the help of new lines and print statements.

When the program first runs, you will be informed that we will be parsing trees using Recursive Descent Parsing, and you will be prompted for a regular expression. If you type in your regular expression, you will either get a tree or an error message. You will continue to be prompted for regular expressions until you enter quit.

Once you have quit from the RDP section, you will be informed that we will now be parsing the Table Driven method. Like with the previous section, you can test regular expressions as your heart desires, until you enter quit.

Finally, you will get another prompt, much longer than the last two. It will inform you that we will be TDPing again, but this time, part 3 of the project will also be displayed, ie, after parsing your tree, you will get the expression string expressing your regular expression if it is valid. Of course, as with the others, you can do this as many times as you please till you decide to quit.

Then the run is complete.

PART 1

A recursive-descent parser (RDP.c) was implemented for the given context-free grammar expressions (which is already provided, see part 2 explanation for the grammar expressions). The parser is able to read in well-formed expressions as inputs and construct a parse tree. [Note:

Well-formed expressions means that it could take in suitable input (inputs that match the given grammar expressions)] If the parser reads in an input that is not well-formed, then the parser returns the error message: "Invalid input."
Some of the functions you may find used in our code are:

- `int matchTerminal()`
  - This function takes in a character and checks whether the following inputs are the same as the character taken in as a parameter. If the next input is the same, it will increase the pointer to check the following input characters while returning 1 as well. If the character in the parameter is not the same as the following inputs, then it would return 0.
- `void RDP()`
  - This function takes in a string and sets the string as the "nextInputChar".
  - Then, we create a tree named "res" and set it equal to the tree `E()` [Note: `E()` stands for expression - it is part of the context-free grammar provided]
  - We check if our input string is fully consumed. If the string is not "\0", it will be an invalid string. If the string is then it will print out our tree with the `printTree()` function.
- `int lookahead()`
  - This function sets the character c to be equal to the string ("*nextInputChar")
- `Type Tree`
  - Each syntactic category in the given context-free grammar is expressed using the type Tree. The following are the created Trees:
    - `E()` for expression
    - `C()` for concatenation
    - `ET()` for expression tail
    - `S()` for star(closure)
    - `CT()` for concatenation tail
    - `A()` for atomic expression
    - `ST()` for star tail
    - `X()` for input symbol
- `void printTree()`
  - When this function is called, checks how to print out the parse trees by using the arrow operator (->) to access either the leftChild, midChild, or rightChild from the root.
  - Later, it will call `printTree()` that will print out respective trees.
  - If none of the if statements are satisfied that means that there is no valid tree and it will print out "Invalid input."

PART 2

For this part (coded in TDP.c), we have defined the grammars as follows:

```
01.  <E> → <C><ET>
02.  <ET> → |<E>
03.  <ET> → ε
04.  <C> → <S><CT>
05.  <CT> → .<C>
06.  <CT> → ε
07.  <S> → <A><ST>
08.  <ST> → *<ST>
09.  <ST> → ε
10.  <A> → (<E>)
11.  <A> → <X>
12.  a|b|...|c
```

The resulting table is thus:

		0	**1**	**2**	**3**	**4**	**5**	**6**
		|	**.**	*****	**(**	**)**	**a|b|...|z**	**NUL**
0	**<E>**	∅	∅	∅	1	∅	1	∅
1	**<ET>**	2	3	3	3	3	3	3
2	**<C>**	∅	∅	∅	4	∅	4	∅
3	**<CT>**	6	5	6	6	6	6	6
4	**<S>**	∅	∅	∅	7	∅	7	∅
5	**<ST>**	9	9	8	9	9	9	9
6	**<A>**	∅	∅	∅	10	∅	11	∅
7	**<X>**	∅	∅	∅	∅	∅	12	∅

For this part of the question, we have constructed a transition table of sorts which we want to use to carry out table driven parsing.

So the first thing we do is hard code the table. The rows are the syntactic categories and the columns are the terminals. As per the instructions, this table is spit out by a function.

Next, we add strings representing the bodies of the twelve productions listed above to an array called `Productions`. For example, we have listed above production (1) as <E> → <C><ET>. Thus, in our array Productions, at the first index, we store the string "<C><ET>."

So here is the hard part.

- We create a tree called result. This will be our ending parse tree.
- Next, we initialize our stack. We keep track of two stacks in our code. First we have `S` and we use a Linked List to represent it. We also have `Snodes`, which will be our stack of nodes, implemented in a similar way. Since our pop method returns the *first* element of the linked list, our push method adds to the front of the linked list.
- Since the head of every tree is <E>, we push E into `S`. Note that angle brackets (<>) will not be included in our stack elements for clarity of reading. Similarly we add rooty, a node with no children and label "E" which will represent the root of our parse tree, into `Snodes`.
- While our stack is not empty, we pop from our stack and save the element we get in a string variable called `stackTop`. We also pop the top of `Snodes` and store it in a tree named `result`. For the sake of explaining this code, let's assume that the top of our stack was the string "E" and our current input character is 'a'. Likewise, the current value stored in result is the root node E. Since it is not a terminal, it skips the for-loop that specifies that terminals get matched immediately.
- So now we derive a string called `toPush`.
    - To describe how `toPush` comes to be, we must first understand my method called `getProduction`. This function returns a string that represents the production associated with the integer passed to it. For example, if you sent in the integer 5, it would return the string `".<C>"`.
    - To get `toPush`, we get the production associated with the integer in row E and column 'a' of our hard-coded table. That is, if you look at our table conveniently typed up above, `tt[<E>]['a']` will return getProduction(1) which is `"<C><ET>"`.
- Now we have the next string that will be pushed to the stack. Of course, we would want to push ET before we push C, so I have yet another method to do that. This is my trusty function method `pushToStack`.
    - This function, depending on what `toPush` is, will push to the stacks (strings for S, nodes for Snodes) the specified terminals and syntactic categories in the specified order. So in our case where toPush is `<C><ET>`, the function will first push `<ET>`, then `<C>`. As we do this, we are also building our tree as we go along. We make nodes in result to represent its children.
- After everything, we print our tree (we print `rooty`) with the same method as with RDP!

PART 3

The function that does part 3 is known as explore, and it is coded in rexp.c which contains our main method.

Since we wanted the parts to be distinct, TDP is called again before we then call `explore()`. Well, technically we call startExplore, because we wanted to keep our explore function, which is recursive, neat.

The basic, very simplified pseudocode for explore is this:

```
explore(Tree v)     {
     call explore() recursively on all children
     handle the operations
     handle the base cases
     return string
}
```

So in order to call `explore()` for all the kids, we have to check each individual child if they exist call explore again. Also, we check if each child returns a string. If they do, we make space in our string subsection, which is the string we will be returning.

After this, we handle the operations, by which I mean concatenation, union and closure. This will only happen if the node we are currently exploring is C, E or S because those are the only nodes which handle those operations. Depending on which of the letters is our node's label, we save an integer size accordingly, to make space for the words we're going to type, then add CONCAT, UNION or CLOSURE, depending, again, on what our node's label is.

Now we handle the base cases, which are the non-operation terminals. Basically, the lowercase letters. For this, we simply type ATOMIC, followed by the letter.

And we return our completed string!

Ferguson does have the whole "but if you want it to come all together…" extra part, but since we don't get extra credit for that, we didn't bother :)