CS 440/540 Spring 2022

Assignment #2: Recursive Descent Parsing

Due date on Blackboard

The goal of this assignment is to use Lex (flex, jflex) write a recursive descent parser in C or Java. Your parser is going to read in an input that describes a NFA using a restricted context free grammar (remember the slide that shows how any NFA can be expressed as a CFG.) and then gives one or more strings. You are going to read in and store the NFA and then for each string, determine whether or not it is in the language. An example input file might look like:

```
A1 -> a A2
A2 -> a A1
A1 ->
aaaa

aaa
```

The first three lines describe an NFA with two states that accepts strings of 'a' where there are an even number of 'a's. The last three lines are a string of 4 'a's, a null string, and a string of three 'a's. The NFA given would accept the first two strings and reject the last one so your output would be:

```
Accept aaaa
Accept
Reject aaa
```

The grammar for the input files:

```
input      --> grammar strings
grammar    --> grammar production
            |  production
production --> NT ARROW T NT EOL
            |  NT ARROW NT EOL
            |  NT ARROW EOL
strings    --> strings string EOL
            |  string EOL
string     --> string T
            |               /* epsilon */
```

To implement this, there are a number of steps. Be sure each step works before going on to the next step.

1. Create a lexer that captures the lexical elements in the grammar above. The lexical elements you need to capture:
   - T - terminal symbols. This will be single lower case alphabetic characters.
   - NT - non-terminal symbols. These are alpha-numeric strings starting with an upper case alphabetic character.
   - ARROW - this is just the two-character sequence "->"
   - EOL - the end of line marker (\n).
   - Discard all other white space.
2. Convert the above grammar to make it LL(1)
3. Write the recursive descent parser for the LL grammar of step 2 - have it use the lexical elements returned by your lexer of step 1. You know this is working if you can parse the given input files, as well as input files you create.

4. Decide what data structure you want to use to represent a NFA. Note that your data structure will be a little more complicated than just building a table since there may be more than one transition associated with a given state and input symbol. Add functionality to your recursive parser to build the data structure that stores the NFA during the parse. Be sure the generated table is correct before going to the next step. As in class, the first production defines the start state for the NFA.
5. Add functionality to parse the input strings with respect to the NFA you have created. The algorithm, given in class on the NFA slides, will be recursive and will require backtracking to get it correct. Converting the NFA to a DFA is possible but probably more work than it is worth.
   Because you will need to backtrack on the input string, you will need to read the entire string to be checked before processing it. You can assume that the input strings will be no more than 255 characters in length. In the output for each string, say 'Accept' or 'Reject' and give the string.

NOTE - do not use YACC (or one of its variants) for this assignment. You will not receive credit for an assignment implemented in YACC.

Submitting

Your parser must read input from `stdin` and write to `stdout` just as you did in your first assignment.

Submit all files needed to build your parser (zipped). Include either a Makefile or a README that has instructions for building your executable on zeus. Leaving out this step will result in a grade penalty. CHECK your file to be sure that you have included all needed elements. If you accidently submit an empty file or miss some file, you will only be allowed to correct this problem if the files in your home directory have a timestamp before the due time.