Dennis Canar

9 – 13 – 20

CST – 405 – TOS101

Nasser Tadayon

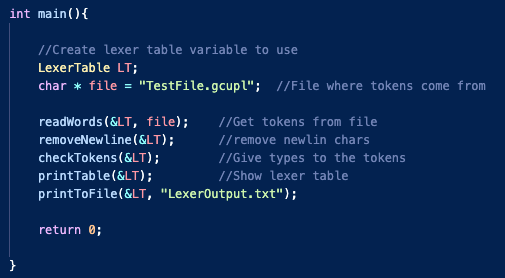
Lexical Analyzer

**Objective:**

The objective of the Lexical Analyzer is to read in a file and produce a table of all the parsed tokens along with its line number and possible error.

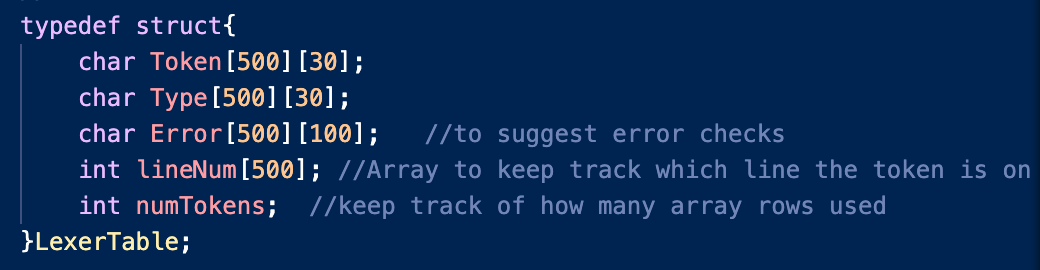
**Code Description:**

This project was started last semester and its initial design is still being used for the new C-- grammar the Lexer will be tokenizing for. Each type of token has its own DFA to recognize each specific type of token. Below in the DFA section I will either present all or most of the DFA so you have an idea.

The program will begin in main() like this:

The first step is to read the from the file and parse all the words into a 2d array of strings. The 2d array is stored in a LexerTable struct along with other variables to store type, error, line number, and number of tokens.

A screenshot of a cell phone

Description automatically generated

Once all the tokens have been parsed the tokens are all run through another function ‘RemoveNewline’. The function will go through each token and check if the last char is a newline character, if it is we replace it with a NULL char to terminate the string. This function is needed because the file is read line by line and the ‘\n’ is not stripped from the line when being read.

When we are sure all the tokens are in the correct format we can then check for their token and assign it to them. The table we created will then be used to loop through each token and run it through each check as shown on the right.

**Tokens:**

|  |  |
| --- | --- |
| Keywords | return, read, write, writeln, break, if, else, while, int, and char |
| Identifier | Follows C variable naming rules |
| Constant | any integer/float without leading 0’s  (unless 0.###) |
| String | any chars in between two quotes “example” |
| Char | any symbol in between two single quotes |
| UnaryOp | +, /, \*, ==, <=, >=, <, >, ||, &&, != |
| BinaryOp | -, ! |
| Equlas | = |
| Special Symbol | {, }, [, ], (, ), ; |
| Unidentified | any token that cannot be categorized as above |

**DFA:**

Each DFA is used within the C program to identify the type of token. The token will begin by going through each token check, if the token comes back as true then the rest of the checks will not proceed. Although each token has a check function there isn’t a DFA needed for each one because the checks are very straight forward. For example, the keywords DFA would just be a line of characters that spell each keyword.

A close up of a map

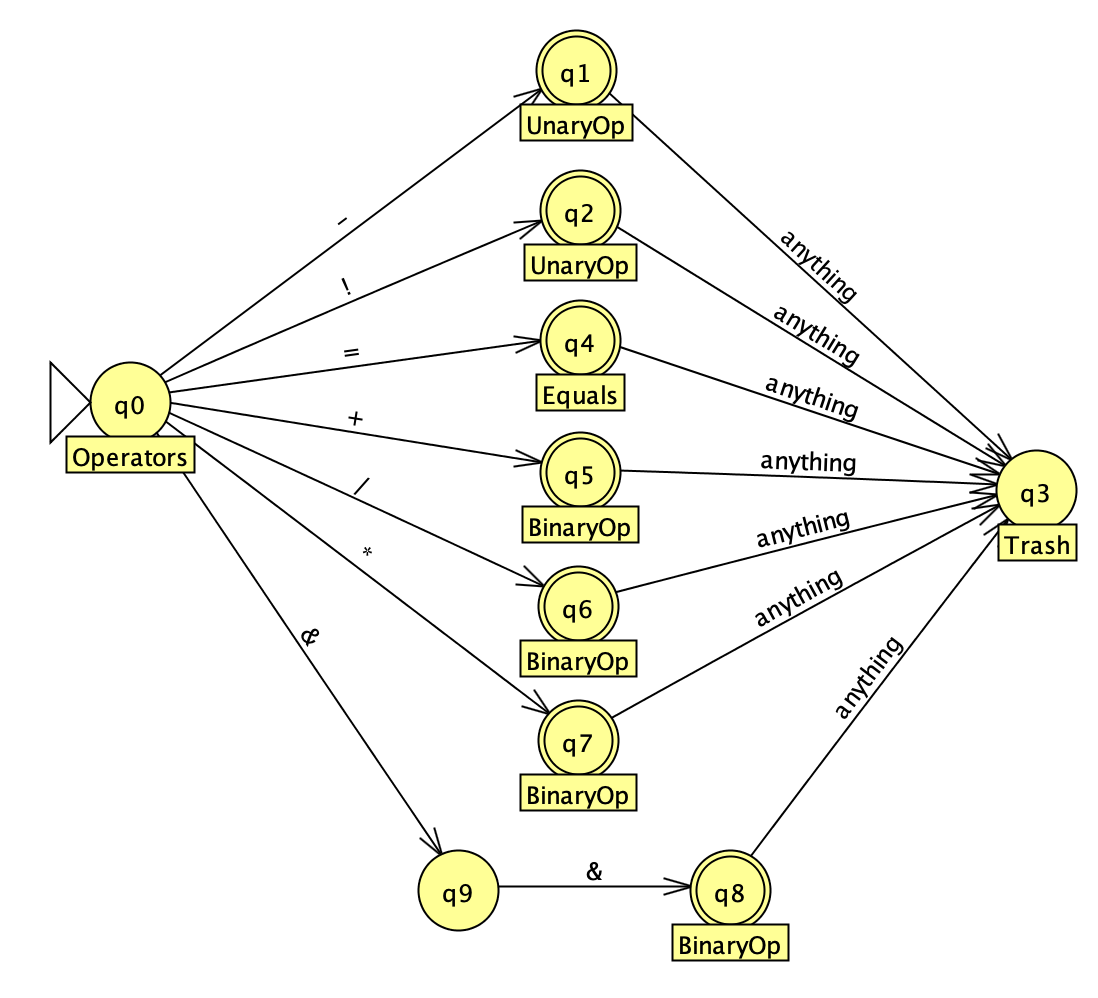
Description automatically generated

As seen above the keyword check goes through each character to check if it would match any keyword. Also, if there is no transition for the current state and a certain char is read, then we can consider it going to the trash because the keyword no longer matches. Thus, the implementation of the DFA in the code isn’t really a DFA because we could instead just store these keywords in an array and check if the token matches any of them.

A close up of a map

Description automatically generatedThe same is applied to our small range of tokens for the unary operator, binary operators, and special symbols. If they had a DFA it would look like the ones below, still following the rule that if there is no transition then the transition is to go to the trash.

Below is the DFA for all our operators and their types, the accepting state shows which type it will be if the string ends in an accepted state. Note, not all of the operators are there because with this example it is rather straightforward to see how each operator could be added to the DFA.



A close up of a map

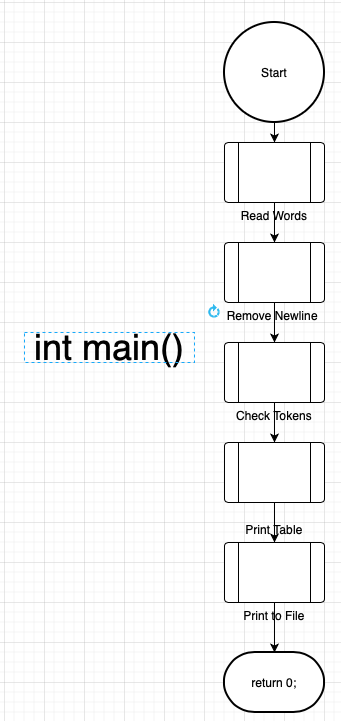
Description automatically generatedBelow on the left is the DFA for our integers and floats, but to call them one name we can call them constants. The DFA below works rather simple there can be two starts to the constants DFA with a 0 or a non 0 number. If the number begins with a zero then the next input must be a decimal because the language does not accept leading zeros. And if the number doesn’t begin with a zero then there is not any other checking to do rather than checking the input is a number 1-9. If the input reads a letter then the DFA will automatically transition to a not accepting trash state.

A close up of a map

Description automatically generated

The DFA on the right is for our identifiers. The DFA checks the identifier starts with the appropriate character. After that any alphanumeric or underscore will stay in the accepted state. Any character not allowed in identifiers will send the DFA to a non-accepting state.

Lastly I did not make DFA’s for the string or char because they are very easy to check. For a string all we need to check is that the last and first element are double quotes. Similarly, we check our chars, except are chars are limited to one character.

**Flowcharts:**

This section I will supply important flow charts to better model how the lexer will work.

Firstly, the flowchart to the right is the high-level flowchart to show our main and the different stages the tokens go through. The program reads the words from the file, removes newline chars, checks tokens, prints our table, then writes to a file where we can use the table in another program.

The read words function works simple so there will be no flow chart. The read words function opens the file we want to read then will read each line until the ‘getLine’ function returns a -1 (meaning we reached the end of the file). For each line iteration the line count variable is increased, and the tokens are parsed using strtok. The strtok parses the line by white space. I then take this parsed string and add it to my array while also adding the line number the token was found on.

The remove newline then will iterate through all of the tokens I recorded and check the last char of the string to see if there is a \n. If there is we remove it by replacing it with a null terminating char ‘\0’.

These next flowcharts will be for the check tokens part of main. This function is a large function made up of smaller individual function. The flowchart below is the same as the one in the description section. Next I will show how each function within check tokens works.

A close up of text on a white background

Description automatically generated

A close up of text on a white background

Description automatically generated

The checkConstant() flowchart is to check our numbers. This handles the case for leading zeros and can also verify float numbers. The way it works is by using Booleans to check if we are in certain state or not. Each state serves a different purpose, q1 will be the initial state and will determine where the DFA goes. State q1 will be used if the first input was a 0, state q2 will be used when the constant has had a non-zero input, and q3 will be used once a decimal input has been read. From these three states we are then limited to what character can be read next to make the constant valid and if the character does not follow our rules we return False.

A close up of a map

Description automatically generated

Above is the flowchart to identify identifiers, first we will check if the first char is allowable for an identifier. If it is, then we continue and check that each following character after that and make sure they are either an alphanumeric or \_. If the input from the token is not allowed then we return False. Because this only really checks if a string starts with [A-Z, a-z, \_] and if the string then contains alphanumerics throughout, and for that reason most tokens that are unrecognized could fall in this category.

A close up of a logo

Description automatically generated

Thus, the final flowchart is a simple flowchart to show how we check against the keywords. Here we just take the token and loop through the Keywords container making comparisons to each keyword. Note, the checkSpecials and checkOperators functions work in the same way so I will not be showing their flowchart.

**Important Screenshots:**

A screenshot of a cell phone

Description automatically generated

The screen shot above is a function that is used within the checkConstant() function. What this does is use the int ASCII value to check if the value falls within a specified range, if it does then we return true If not then false.

A screenshot of a cell phone

Description automatically generated

This screenshot shows how the file is read from, the function begins by setting our variables to track the position in the file, line being read, and line count. After that the file is opened and read line by line and the counter is updated. Once the line is read we then use strtok to parse the string and iterate over the words. The words finally are added to the table and given the correct line number information.

A screenshot of a cell phone

Description automatically generated

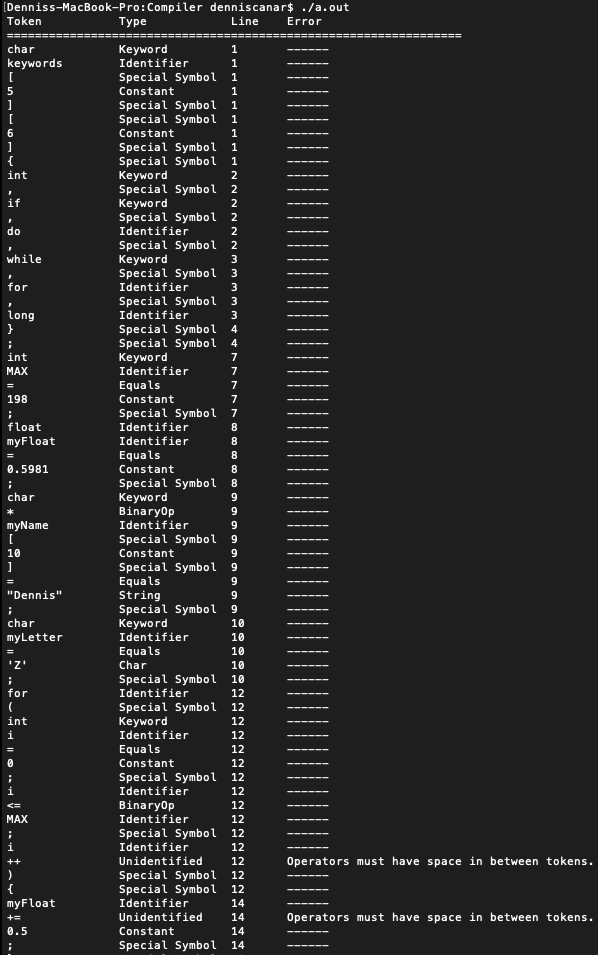
This screenshot is of the function we use to print to our out file. We pass in the table and the file name, then open the file and print each row of data from the table.

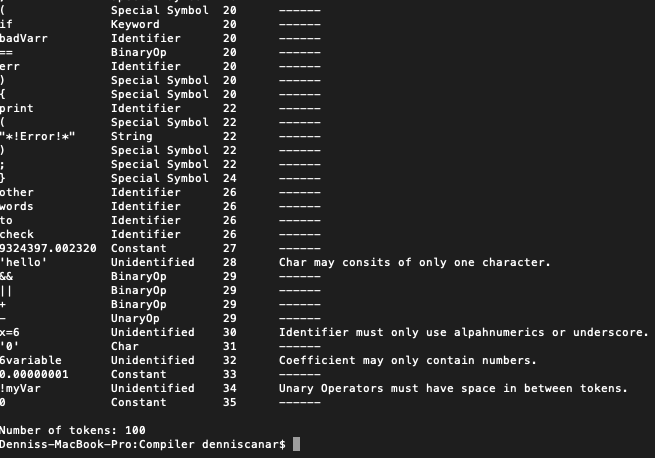
A screenshot of a cell phone

Description automatically generated

Above is how the out-file looks.

**Code Execution:**





* To execute have .C and .GCU-PL file within the same directory and use terminal to execute “gcc LexicalAnalyzer.C -w”. then run “./a.out” to see the output within the terminal.
* The testFile should be provided and if any changes want to be made to the file, it is ok it wont affect the lexer itself. And if there is a different filename to use then it can be changed in the main().

**Error Handling:**

* If tokens are too long then printing will be a bit off.
* If more than 500 tokens then can result in seg fault.
* There must be spaces between token types