1. /\* SPECIAL SYMBOLS:

\*

\* + -> ADDITION

\* = -> ASSIGNMENT

\* - -> SUBTRACTION

\* / -> DIVISION

\* \* -> MULTIPLICATION

\* ++ -> INCREMENT

\* -- -> DECREMENT

\* % -> MODULO\_OPERATOR

\* && -> LOGICAL\_AND

\* || -> LOGICAL\_OR

\* ! -> LOGICAL\_NOT

\* { -> OPEN\_CODE\_BLOCK

\* } -> CLOSE\_CODE\_BLOCK

\* ( -> OPEN\_FUNC\_PARAM

\* ) -> CLOSE\_FUNC\_PARAM

\*

\*/

2. The results on my computer seem to differ each time I execute the code because the compute takes a slightly different amount of time to allocate the memory each time. Here is one example of my output:Text

Description automatically generated

It appears that on my machine, it takes the longest time in nanoseconds to allocate memory from the heap (the third result) with a time of 20,634,655 nanoseconds. The second slowest was the array from the stack at a time of 340,787 nanoseconds. The fastest was the static array with a time of 266,404 nanoseconds. I imagine that these times would differ depending on the machine running the code.

3. Arithmetic -> Arithmetic Operation | integer | (Arithmetic Operation)

Operation -> \* | / | + | - | %

The types of parsers that can be used to show the syntax for this are LL parsers and recursive decent parsers. An LL parser means that the parser is a left-to-right scan of the input and that a leftmost derivation is generated. A recursive descent parser is a type of LL parser that builds the parse tree in preorder/top down. Each node is visited before its branches are followed.

4. The part of the compilation process that allows us to determine the reference environment for any given line of code in the program is namely the scoping. Dynamic scoping is less widely used than static scoping because static scoping is easier to understand the context just by looking at a block of code. However, dynamic scoping is concerned more with how the program executes than how it is written. Each identifier has a global stack of bindings and the occurrence of an identifier is searched in the most recent binding. On the other hand, static scoping (also called lexical scoping) always refers to it top level environment. Static scoping is binding names to nonlocal variables known as static. The scope of a variable here is determined prior to execution and makes for easier code readability. The search begins in the subprogram containing the reference to the variable, then continues to the enclosing program, then to al the static ancestors of the subprogram.

5. When designing a lexical analyzer, reserved words should be given a higher rule priority over other tokens. You would have to change your code to implement this by throwing an error if the user tries to use a reserved work as an identifier, for example. In order to allow a user to choose a reserved word as an identifier, you could change the priority of keywords to be lower than identifiers so a user’s identifier choice is “more important” than a reserved word.

6.

EBNF Rules:

WHILE:

<while\_stmt> -> WHILE ‘(‘(<arithm\_expr> | <logical\_expr)’)’

<block><block> -> <stmt> | ‘{‘<stmt> {<stmt>} ‘}’

IF:

<if\_stmt> -> if (<boolean\_expr>) < stmt> {else <stmt}

LOGICAL/MATHEMATIC EXPRESSION:

<expression> -> <term> {(+|-) <term>}

<term> -> <factor> {(\*|/) <factor>}

<factor> -> ‘(‘ <expression> ‘)’ | name | number

ASSIGNMENT STATEMENT:

<assign> -> <id> = <expr>

<id> -> A|B|C

<expr> -> <expr> (\*|+|-|/) <expr> | <id> | (<expr>)

7. A recursive descent function’s biggest problem as an LL parser is left recursion. As long as the statement function has all of the information needed, correct parenthesis, and correct scoping using brackets, the statement will be able to be either a while, if, or an assignment statement. The left recursion problem that exists within top-down parsers is not present with bottom-up parsers. Removing direct and indirect left recursion could solve this problem in this case.

8. (Code is attached as Q8.txt)