Project 1 Revised:

1.) The general implementation for a lexical analyzer is a function that tokenizes the given input file into an array of strings while ignoring white spaces, new lines, and the contents of a comment. For this particular implementation of a scanner, we will rely on a table driven parser.

There are two major components necessary when constructing a table-driven parser: a parsing table and a production table. A parsing table is a two dimensional array indexed by T[X][CT], X being a nonterminal and [CT] being the current token to be matched. The production table is indexed by numbers for the production rules and the corresponding rule the top of the stack needs to be replaced with. These are typically calculated by using a parser generator or FIRST and FOLLOW functions, but since they have already been given for this assignment, they can be hardcoded in. So for example, given a program, the parse stack would identify whether the input is a <*stmt_list*> or the end of file. Assuming the input isn't an empty file, the parse stack moves from <*program*> to <*stmt_list*>, whose corresponding production rule is to check the statement followed by statement list; i.e. <*stmt*><*stmt_list*>. The parse stack passes the input to <*stmt_list*> which will identify the statement as either an id, read, or write. If the token is a match for either of the three, the program prints the appropriate token but if it is not, the program moves forward through the parse tree to check if the statement is an expression, following the same ideology as <*stmt*><*stmt_list*> to check if <*term*><*term_tail*> will continue to pass to <*factor*><*factor_tail*>, decomposing the token until we match with a particular operator. This represents the fundamental ideology of this scanner program, pushing the body of the stack until the current symbol is terminal.

2.) Pseudocode:

These represent the two tables the program will be using to parse through the tokens that have been identified:

```
parse_table = {'program' : {'id': 1, 'read': 1, 'write': 1, '$$': 1},
               'stmt_list' : {'id': 2, 'read': 2, 'write': 2, '$$': 3},
               'stmt' : {'id': 4, 'read': 5, 'write': 6},
               'expr' : {'id': 7, 'number': 7, '(': 7},
               'term_tail' : {'id': 9, 'read': 9, 'write': 9, ')': 9, '+': 8,
                  '-': 8, '$$': 9},
               'term' : {'id': 10, 'number': 10, '(': 10},
               'factor_tail' : {'id': 12, 'read': 12, 'write': 12, ')': 12,
                        '+': 12, '-': 12, '*': 11, '/': 11, '$$': 12},
               'factor' : {'id': 14, 'number': 15, '(': 13},
               'add_op' : {'+': 16, '-': 17},
               'mult_op' : {'*': 18, '/': 19}}
```

```
prod_rules = {1: ['stmt_list', '$$'],
              2: ['stmt', 'stmt_list'],
              3: [],
              4: ['id', ':=', 'expr'],
              5: ['read', 'id'],
              6: ['write', 'expr'],
              7: ['term', 'term_tail'],
              8: ['add_op', 'term', 'term_tail'],
              9: [],
              10: ['factor', 'factor_tail'],
              11: ['mult_op', 'factor', 'factor_tail'],
              12: [],
              13: ['expr'],
              14: ['id'],
              15: ['number'],
              16: ['+'],
              17: ['-'],
              18: ['*'],
              19: ['/']}
```

1. Pseudocode for recognizing tokens

function name: recognized_tokens(filename)

input

all characters in file disregarding newlines and tabs

output

list containing all tokens that have been recognized

data

tokens in array

plan

get all individual characters from input file disregarding newlines and tabs but not white spaces
string = join all characters together to form one string
source = list of strings split by removing '/* comments */'

for i in str(source)
split strings in list again at white spaces

now create list raw_input properly splitting each element individually

```
for inner_lst in source
      for ele in inner_lst
            append all elements to raw_input

source = raw_input

create final_list of inputs since source is a list of lists
for inner_lst in source
      for ele in inner_lst
            append all elements to final_list

now all tokens are properly identified and stored
tokens is a list with all the terminals identified to aid parsing
for element in final_list
      if element in terminals
            append to tokens
      elif str(element) isdigit
            append to tokens as 'number'
      elif element != 'read' or element != 'write'
            append to tokens as 'id'
      else
            'Token is not recognized'

return tokens
```

2. Pseudocode for parsing function

```
function name: scanner(tokens)
      input
            tokens identified from the text file
      output
            all tokens properly identified
      data
            tokens stored in an array
      plan
            create empty stack and append first production rule 'program'
            initialize i to 0

            try:

                  loop while stack is empty, removing the top element of the
                  stack with each iteration
```

```
            while length of stack > 0
                  current_token = first element from tokens
                  top = first element in stack
                  if (isTerminal(top) and top == current_token)
                        pop stack
                        iterate i
                        continue
                  else
                        get new production rule from function
                        getProdrule
                        pop stack
                        rule_list = value from getProdrule
                        stack = rule_list + stack

      except IndexError
            return results
```

3. Pseudocode for checking if token is terminal

function name: isTerminal(top)

input

top element of the stack

output

bool True or False

data

N/A

plan

```
if the top element is not a key in the parse_table, it is not a
nonterminal
if top not in parse_table.keys()
      return True
```

4. Pseudocode for getting new production rule

function name: getProdrule(top, current_token)

input

top of stack and current token

output

next production rule that needs to get pushed to stack

data

list of new rules

plan

try

find nested dictionary with key nonterminal and value of the current token
next_prod_num = parse_table[top][current_token]
append list of applied production rules so far
new_rules = list obtained from table of prod_rules
return new_rules

except if top of stack is terminal

append list of applied production rules with 0
next_prod_num = recursive lookup function
new_rules = list found by recursive lookup function
return new_rules

5. Pseudocode for finding next production rule in parsing table

function name: recursive_lookup(key, table)

input

current key we're looking for

output

corresponding production rules

data

N/A

plan

recursively iterates through parse_table to find which nonterminal dict contains the current token to find the next production rule
if key in table
return table[key]
for value in table.values()
returns True if value is found in dict or calls function again if result is None
if isinstance(value, dict)
a = recursive_lookup(key, value)
if a is not None
return a

return None

6. Pseudocode to print results of scanner

```
function name: print_results()
      input
            none
      output
            formatted results
      data
            results from scanner
      plan

            iterate through list of results
                  if word == ‘:=’
                        results[i] = ‘assign’
                  elif word == '(':
                  results[i] = 'left parentheses'
                  elif word == ')':
                  results[i] = 'right parentheses'
                  elif word == '+':
                  results[i] = 'plus'
                  elif word == '-':
                  results[i] = 'minus'
                  elif word == '*':
                  results[i] = 'times'
                  elif word == '/':
                  results[i] = 'divide'

            formatted_results = results joined by a comma
            print formatted_results
```

3.) Test Cases

The first two cases were chosen for their brevity and to show the accuracy of the parse stack. The stack must parse through over a dozen production rules before exiting, so a long input file would be overwhelming.

Input: `sum := A + B write sum`

Output: `(id, assign, id, plus, id, write, id)`

This input was an assignment on the homework and the parse stack from my program is shown below. It directly matches the parse found on the homework.

1. [‘program’, ‘$$’]
2. ['stmt_list', '$$', '$$']
3. ['stmt', 'stmt_list', '$$', '$$']
4. ['id', ':=', 'expr', 'stmt_list', '$$', '$$']
    a. **Results** ['id']
    b. **Results** ['id', ':=']
5. ['term', 'term_tail', 'stmt_list', '$$', '$$']
6. ['factor', 'factor_tail', 'term_tail', 'stmt_list', '$$', '$$']
7. ['id', 'factor_tail', 'term_tail', 'stmt_list', '$$', '$$']
    a. **Results** ['id', ':=', 'id']
8. ['term_tail', 'stmt_list', '$$', '$$']
9. ['add_op', 'term', 'term_tail', 'stmt_list', '$$', '$$']
10. ['+', 'term', 'term_tail', 'stmt_list', '$$', '$$']
    a. **Results** ['id', ':=', 'id', '+']
11. ['factor', 'factor_tail', 'term_tail', 'stmt_list', '$$', '$$']
12. ['id', 'factor_tail', 'term_tail', 'stmt_list', '$$', '$$']
    a. **Results** ['id', ':=', 'id', '+', 'id']
13. ['term_tail', 'stmt_list', '$$', '$$']
14. ['stmt_list', '$$', '$$']
15. ['stmt', 'stmt_list', '$$', '$$']
16. ['write', 'expr', 'stmt_list', '$$', '$$']
    a. **Results** ['id', ':=', 'id', '+', 'id', 'write']
17. ['term', 'term_tail', 'stmt_list', '$$', '$$']
18. ['factor', 'factor_tail', 'term_tail', 'stmt_list', '$$', '$$']
19. ['id', 'factor_tail', 'term_tail', 'stmt_list', '$$', '$$']
    a. **Results** ['id', ':=', 'id', '+', 'id', 'write', 'id']
20. ['stmt_list', '$$', '$$']
21. ['$$']

Input:

```
read
/* foo
      bar */
*
five 5
```

Output: `(read, times, id, number)`

This input is the test case given in the assignment. The expected output was also given, but the corresponding parse stack has also been included below.

```
1. [‘program’, ‘$$’]
2. ['stmt_list', '$$', '$$']
3. ['stmt', 'stmt_list', '$$', '$$']
4. ['read', 'id', 'stmt_list', '$$', '$$']
      a. Results ['read']
5. ['mult_op', 'factor', 'factor_tail', 'stmt_list', '$$', '$$']
6. ['*', 'factor', 'factor_tail', 'stmt_list', '$$', '$$']
      a. Results ['read', '*']
7. ['id', 'factor_tail', 'stmt_list', '$$', '$$']
      a. Results ['read', '*', 'id']
8. ['term', 'term_tail', 'stmt_list', '$$', '$$']
9. ['factor', 'factor_tail', 'term_tail', 'stmt_list', '$$', '$$']
10.['number', 'factor_tail', 'term_tail', 'stmt_list', '$$', '$$']
      a. Results ['read', '*', 'id', 'number']
11.['stmt_list', '$$', '$$']
12.['$$']
```

Input:
`(5) purple r3ad`

`4+55`

Output: `(left parentheses, number, right parentheses, id, id, number, plus, number)`

This last example is used to show the accuracy of the parse stack and how it manages to match tokens when the first token is neither read, write, nor id.

```
1. ['term', 'term_tail', '$$']
2. ['factor', 'factor_tail', 'term_tail', '$$']
3. ['(', 'expr', ')', 'factor_tail', 'term_tail', '$$']
      a. Results ['(']
4. ['term', 'term_tail', ')', 'factor_tail', 'term_tail', '$$']
5. ['factor', 'factor_tail', 'term_tail', ')', 'factor_tail',
   'term_tail', '$$']
6. ['number', 'factor_tail', 'term_tail', ')', 'factor_tail',
   'term_tail', '$$']
      a. Results ['(', 'number']
7. ['term_tail', ')', 'factor_tail', 'term_tail', '$$']
```

8. [')', 'factor_tail', 'term_tail', '$$']
    a. **Results** ['(', 'number', ')']
9. ['term_tail', '$$']
10. ['$$']
11. ['stmt_list', '$$']
12. ['stmt', 'stmt_list', '$$']
13. ['id', ':=', 'expr', 'stmt_list', '$$']
    a. **Results** ['(', 'number', ')', 'id']
14. ['stmt', 'stmt_list', '$$']
15. ['id', ':=', 'expr', 'stmt_list', '$$']
    a. **Results** ['(', 'number', ')', 'id', 'id']
16. ['term', 'term_tail', '$$']
17. ['factor', 'factor_tail', 'term_tail', '$$']
18. ['number', 'factor_tail', 'term_tail', '$$']
    a. **Results** ['(', 'number', ')', 'id', 'id', 'number']
19. ['term_tail', '$$']
20. ['add_op', 'term', 'term_tail', '$$']
21. ['+', 'term', 'term_tail', '$$']
    a. **Results** ['(', 'number', ')', 'id', 'id', 'number', '+']
22. ['factor', 'factor_tail', 'term_tail', '$$']
23. ['number', 'factor_tail', 'term_tail', '$$']
    a. **Results** ['(', 'number', ')', 'id', 'id', 'number', '+',
       'number']
24. ['$$']