1. *Introduction*

This document contains a breakdown/report in reference to the contents of Project 2 “Building a Parser”. The parser was written in Java due to its structurability and object-oriented capabilities. The contributors include ***Nafiz Imtiaz, Chris Dihenia, and Axel Alvarez.*** All rights are reserved to Texas Tech University.

2. *Data Structures*

- **tokenList**: *For any final state s, tokenList[s] is the ArrayList to hold the processed tokens types concludable from the final state, s.*
- **valueList**: *For any final state s, valueList[s] is the ArrayList to hold the actual value of the processed tokens concludable from the final state, s.*
- **err:** *err[] is the ArrayList to hold the “error.” statement.*
    - **ArrayList:** *The ArrayList is a data structure in java that is a resizable array. The elements can be added and removed from an ArrayList whenever you want.*
- **parse _stack**: *The parse_stack is the Stack utilized to hold the parser grammar, elements from the* ***tokenList*** *(when match was found), and elements from the* ***valueList*** *(when match was found).*
- **temporaryStack**: *A temporary stack that takes in popped items from the parse_stack to format and print the elements in reverse order* ***(True Output)***
    - **Stack:** *The Stack is a linear data structure that is used to store the collection of objects. It is based on Last-In-First-Out (LIFO).*

3. *Algorithms (In Pseudocode)*

What outcome do we want from the program?:

We want to construct a parser that takes in the scanned tokens by the scanner and outputs a parse tree in an XML format following the provided context free grammar.

MAIN:

Name: *main*
Input: *N/A*
Data: *Scanner ( )*
Output: *tokenList or Error.*

```
file content = ' '
//File Open Close
try
   File open = new file
   Scanner reader = new Scanner
```

```
  while nextline is true
      increment file content
case file == NULL
  return file_not_found_error()

otherwise
  return TokenList[i]
```

SCANNER:

Name: *DFA_Scanner*

Input: *fileContent, valueList*

Data: *tokenList: the list to hold the processed token types*
valueList: the list to hold the processed token values
err: the list to hold error.

Output: *tokenList<ArrayList> - (STRING)*
valueList<ArrayList> - (STRING)

Objective: *To scan the entire file, locate the individual tokens*

Side Effects: *N/A*

```
//Array Lists
tokenList [] // Stores final state token
valueList [] // Stores token types

while index != End of File

from File
Case input ' ' , '\n' , '\t'
    continue

  Case input ' / ' append to valueList
  append token type "div" to tokenList


  Case input ' ( ' append to valueList
  append "lparen" to tokenList

  Case input ' ) ' append to valueList
  append "rparen" to tokenList

  Case input ' + ' append to valueList
```

```
    append "plus" to tokenList

    Case input ' - ' append to valueList
    append "minus" to tokenList

    Case input ' * ' append to valueList
    append "times" to tokenList

    Case input ' : ' and ' = ' append ' := '  to valueList
    append "assign" to tokenList

    otherwise case input ' : ' return error


    case input ' . ' and ' '
        return error
    otherwise case input '.' followed by 'number'
        if isDigit == True
            append "number" to tokenList
            String isDigit += character at index
        append isDigit to valueList

    case input "id" append to tokenList
        while input isLetter || isDigit
        String idCheck += character at index

        case "read" append to TokenList
            append idCheck to valueList
        case "write" append to TokenList
            append idCheck to valueList
        otherwise
            append "id" to tokenList
            append idCheck to valueList
    otherwise return non-valid-token-error


Case End of File

return tokenList //Returns our scanner output
```

PARSER:

Name: *reversePrint()*
Input: *parse_stack*
Data: *temporaryStack: A temporary stack that takes in popped items from the parse_stack to reverse them into correct order*
Output: *reversed parsed stack which is the final stack with correct order*
Objective: *To format the stack elements in correct order.*
Side Effects: *Prints the elements of stack in reverse order(Actual Output).*

```
{temporary stack initialized
traverse the parse_stack till the end of stack
        Pop the elements from parse_stack
        Push the elements in the temporary stack
        tabbing is done when pushing each element
}
```

Name: *match()*
Input: *tokenList, valueList, parse_stack*
Data: *val : to get the value from valueList*
Output: *output <ArrayList>*
Objective: *To match identified tokens when its specific parse block is reached*
Side Effects: *get the next token into input_token when output is ok.*

```
{
 if (expectedToken == input_token)
 // get next token from the input program input_token = scan();
        return ok;
else return parse_error }
```

Name: *program()*
Input: *tokenList, valueList, parse_stack*
Output: *ok if the input program follows the production on ; and parse_error otherwise.*
Objective: *To decompose the program calling stmt_List*
Side Effects: *value of input_token may be changed*

```
{ //   <program> -> <stmt_list>  $$
// $$ is the end of the program token
        case input_token of id, read, write, $$:
 // if part of the input program is stmt_list
         if (stmt_list() == ok)
```

```
    //rest of the program must be “end of the program”
     return match($$);
    else return parse_error
 otherwise: return parse_error
}
```

Name: *stmt_List()*
Input: *tokenList, valueList, parse_stack*
Output: *ok if the input stmt_List follows the production on ; and parse_error otherwise.*
Objective: *To decompose the stmt_List calling stmt , stmt_List*
Side Effects: *if output is ok, input_token will be the token after the input token sequence that forms <stmt> .*

```
{ //<stmtlist> → <stmt> <stmtlist> | ε
case input_token of id, read, write:
    if (stmt() == ok)
        return (stmt_list())
    else return parse_error $$:
        return ok
    // otherwise return parse_error }
```

Name: *stmt()*
Input: *tokenList, valueList, parse_stack*
Output: *ok if the input stmt follows the production on ; and parse_error otherwise.*
Objective: *To decompose the stmt*
Side Effects: *if output is ok, input_token will be the token after the input token sequence that forms <expr>*

```
{//<stmt> → id assign <expr> | read id | write <expr>
 case input_token of
    id:
        match(id);
        if match(:=)
            return expr()
        else return parse_error
    read :
        match(read);
         return match(id)
    write :
        match(write);
        return expr()
        otherwise return parse_error
}
```

Name: *expr()*
Input: *tokenList, valueList, parse_stack*
Output: *ok if the input stmt follows the production on ; and parse_error otherwise.*
Objective: *To decompose the expr*
Side Effects: *if output is ok, input_token will be the token after the input token sequence that forms <term_tail>*

```
{//<expr> → <term> <termtail>
case input token of id, number, ( :
        if (term() == ok)
                return term tail()
         otherwise parse error
}
```

Name: *term_tail()*
Input: *tokenList, valueList, parse_stack*
Output: *ok if the input term_tail follows the production on ; and parse_error otherwise.*
Objective: *To decompose the term_tail*
Side Effects: *if output is ok, input_token will be the token after the input token sequence that forms <term>*

```
{//<termtail> → <addop> <term> <termtail> | ε
case input token of
        +, - :
                if  (add op() == ok)
                        if (term() == ok)
                        return term tail()
        id, read, write, $$ :
                return ok
        otherwise parse error
}
```

Name: *term()*
Input: *tokenList, valueList, parse_stack*
Output: *ok if the input term follows the production on ; and parse_error otherwise.*
Objective: *To decompose the term*
Side Effects: *if output is ok, input_token will be the token after the input token sequence that forms <factor_tail>*

```
{//<term> → <factor> <facttail>
case input token of id, number, ( :
    if( factor() == ok)
        return  factor tail()
     otherwise parse error
}
```

Name: *factor_tail()*
Input: *tokenList, valueList, parse_stack*
Output: *ok if the input factor_tail follows the production on ; and parse_error otherwise.*
Objective: *To decompose the factor_tail*
Side Effects: *if output is ok, input_token will be the token after the input token sequence that forms <factor>*

```
{//<factortail> → <multop> <factor> <factortail> | ε
case input token of *, / :
    if (mult op() == ok)
        if (factor() == ok)
            return factor tail()
     +, -, ), id, read, write, $$ :
        return ok
    otherwise parse error
}
```

Name: *factor()*
Input: *tokenList, valueList, parse_stack*
Output: *ok if the input factor follows the production on ; and parse_error otherwise.*
Objective: *To decompose the factor*
Side Effects: *if output is ok, input_token will be the token after the input token sequence that forms <factor_tail>*

```
{//<factor> → lparen <expr> rparen | id | number
case input token of
    id :
        return match(id)
    number :
        return  match(number)
    ( :
        if ( match( ( ) == ok)
                    if (expr() == ok)
                    return match( ) )
        otherwise parse error
}
```

Name: *addop()*
Input: *tokenList, valueList, parse_stack*
Output: *ok if the input addop follows the production on ; and parse_error otherwise.*
Objective: *To decompose the addop*
Side Effects: *N/A*

```
{//<addop> → plus | minus
case input token of
        + :
        return match(+)
        - :
         return match(-)
otherwise parse error
}
```

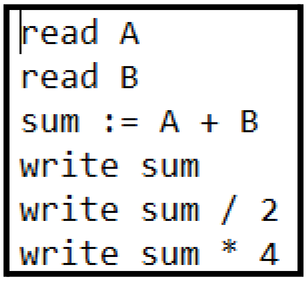
Name: *multop()*
Input: *tokenList, valueList, parse_stack*
Output: *ok if the input multop follows the production on ; and parse_error otherwise.*
Objective: *To decompose the multop*
Side Effects: *N/A*

```
{//<multop> → times | div
case input token of
        * :
        return match(*)
        / :
         return match(/)
otherwise parse error
}
```

4. *Test Cases*

Test Case 1:

Input:

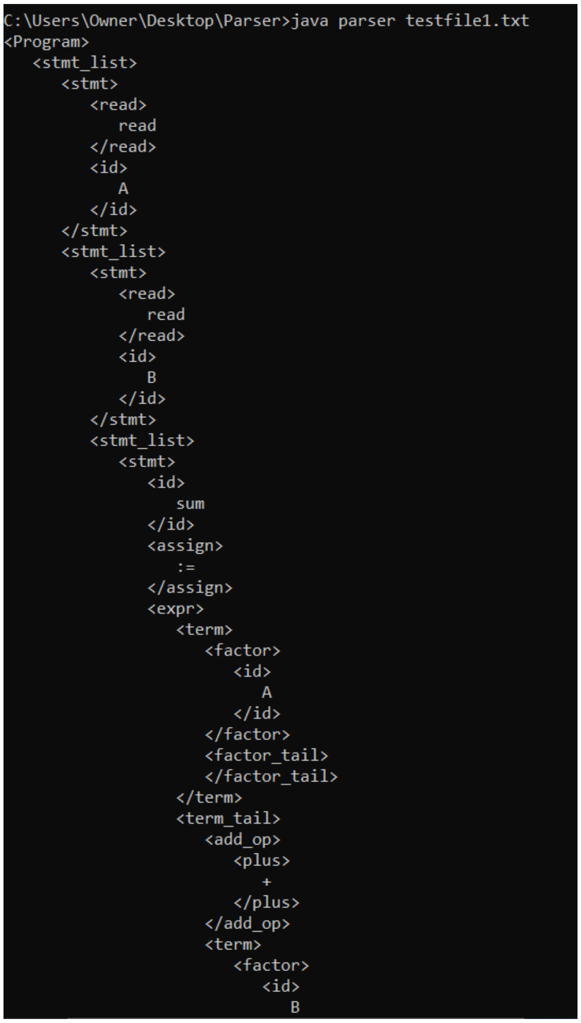


```
read A
read B
sum := A + B
write sum
write sum / 2
write sum * 4
```

<====== ***test1.txt***

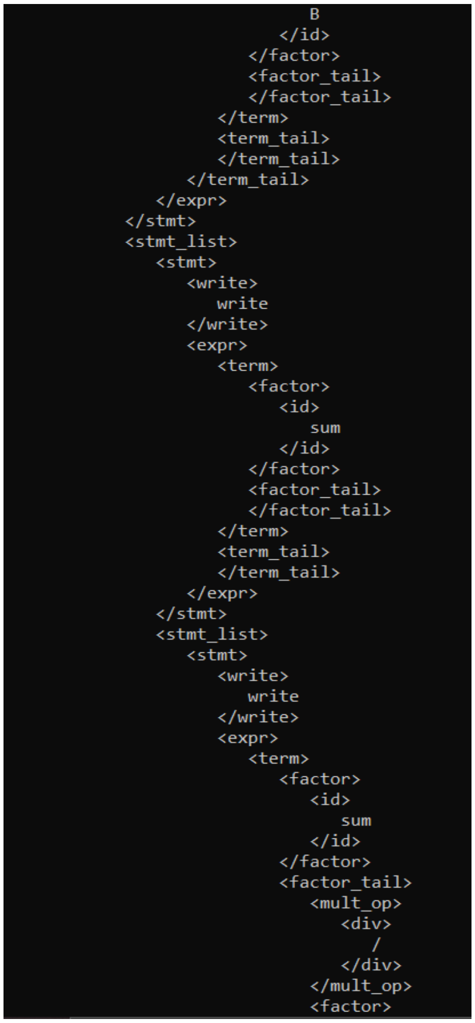
Output:

1.



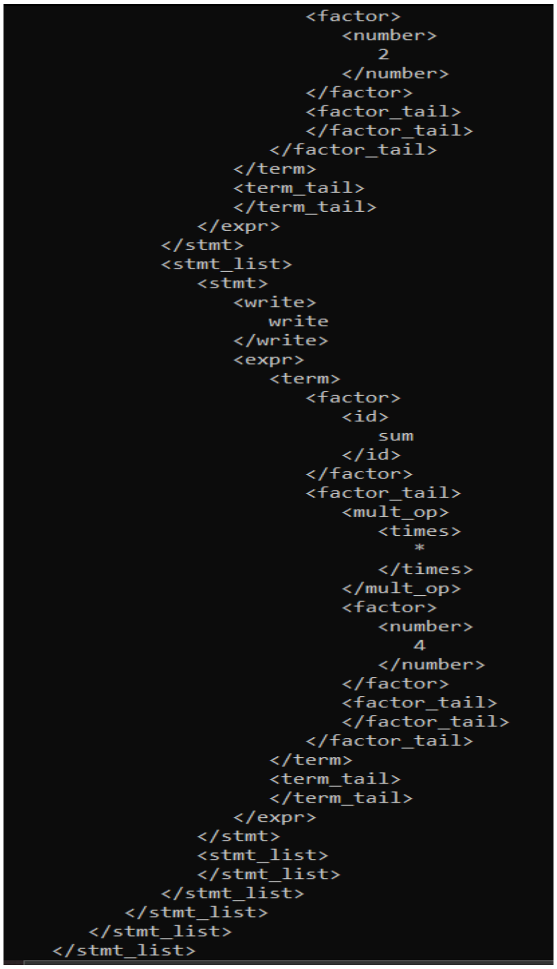
```
C:\Users\Owner\Desktop\Parser>java parser testfile1.txt
<Program>
   <stmt_list>
      <stmt>
         <read>
            read
         </read>
         <id>
            A
         </id>
      </stmt>
      <stmt_list>
         <stmt>
            <read>
               read
            </read>
            <id>
               B
            </id>
         </stmt>
         <stmt_list>
            <stmt>
               <id>
                  sum
               </id>
               <assign>
                  :=
               </assign>
               <expr>
                  <term>
                     <factor>
                        <id>
                           A
                        </id>
                     </factor>
                     <factor_tail>
                     </factor_tail>
                  </term>
                  <term_tail>
                     <add_op>
                        <plus>
                           +
                        </plus>
                     </add_op>
                     <term>
                        <factor>
                           <id>
                              B
```

2.

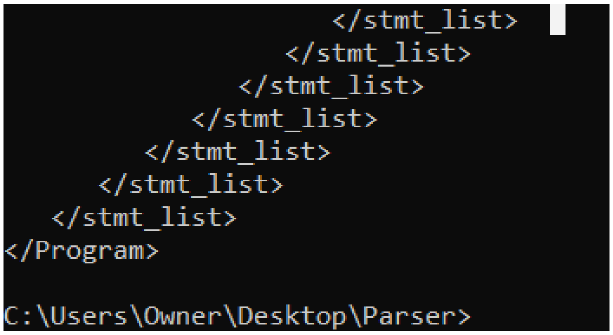


```
                              B
                           </id>
                        </factor>
                        <factor_tail>
                        </factor_tail>
                     </term>
                     <term_tail>
                     </term_tail>
                  </term_tail>
               </expr>
            </stmt>
            <stmt_list>
               <stmt>
                  <write>
                     write
                  </write>
                  <expr>
                     <term>
                        <factor>
                           <id>
                              sum
                           </id>
                        </factor>
                        <factor_tail>
                        </factor_tail>
                     </term>
                     <term_tail>
                     </term_tail>
                  </expr>
               </stmt>
               <stmt_list>
                  <stmt>
                     <write>
                        write
                     </write>
                     <expr>
                        <term>
                           <factor>
                              <id>
                                 sum
                              </id>
                           </factor>
                           <factor_tail>
                              <mult_op>
                                 <div>
                                    /
                                 </div>
                              </mult_op>
                              <factor>
```

3. ***4.***



```
                         <factor>
                            <number>
                               2
                            </number>
                         </factor>
                         <factor_tail>
                         </factor_tail>
                      </factor_tail>
                   </term>
                   <term_tail>
                   </term_tail>
                </expr>
             </stmt>
             <stmt_list>
                <stmt>
                   <write>
                      write
                   </write>
                   <expr>
                      <term>
                         <factor>
                            <id>
                               sum
                            </id>
                         </factor>
                         <factor_tail>
                            <mult_op>
                               <times>
                                  *
                               </times>
                            </mult_op>
                            <factor>
                               <number>
                                  4
                               </number>
                            </factor>
                            <factor_tail>
                            </factor_tail>
                         </factor_tail>
                      </term>
                      <term_tail>
                      </term_tail>
                   </expr>
                </stmt>
                <stmt_list>
                </stmt_list>
             </stmt_list>
          </stmt_list>
       </stmt_list>
    </stmt_list>
```

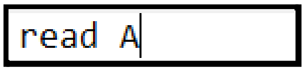


```
                     </stmt_list>
                  </stmt_list>
               </stmt_list>
            </stmt_list>
         </stmt_list>
      </stmt_list>
   </stmt_list>
</Program>

C:\Users\Owner\Desktop\Parser>
```

Test Case 2:

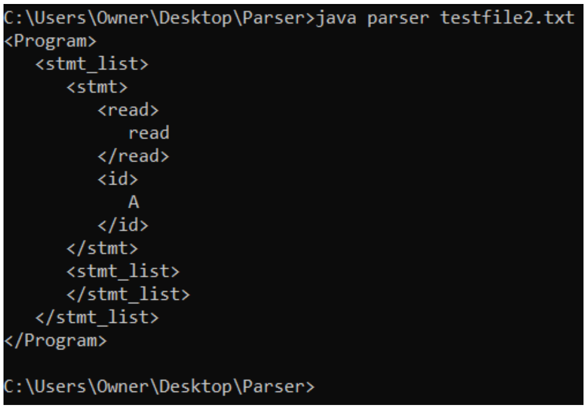
Input:



```
read A
```

<====== ***test2.txt***

Output:

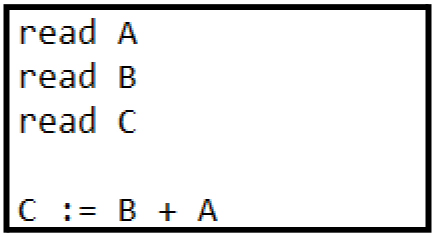


```
C:\Users\Owner\Desktop\Parser>java parser testfile2.txt
<Program>
   <stmt_list>
      <stmt>
         <read>
            read
         </read>
         <id>
            A
         </id>
      </stmt>
      <stmt_list>
      </stmt_list>
   </stmt_list>
</Program>

C:\Users\Owner\Desktop\Parser>
```

Test Case 3:

Input:



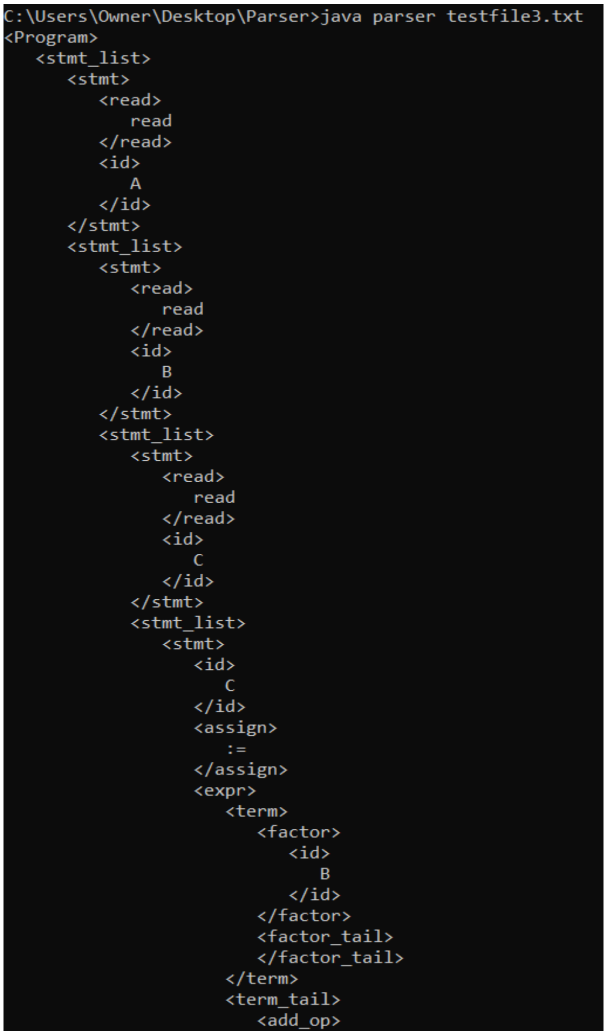
```
read A
read B
read C

C := B + A
```

<===== ***test3.txt***

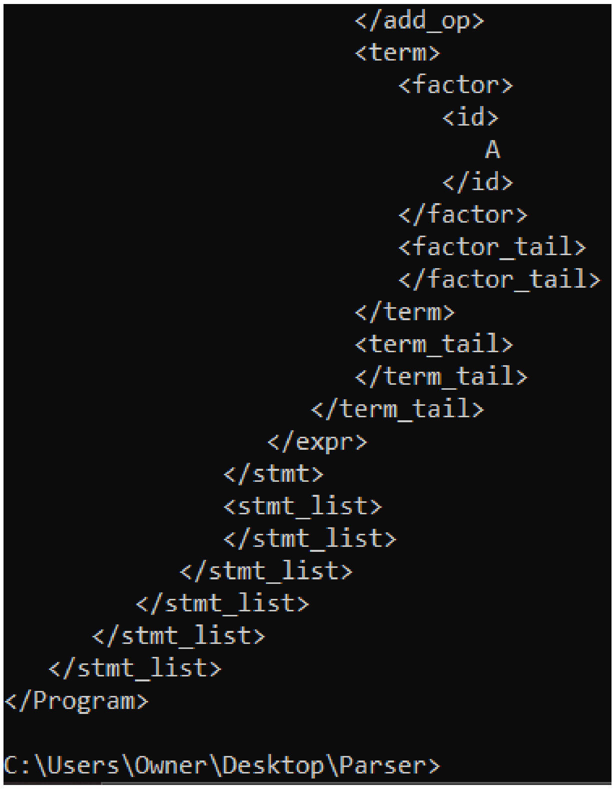
Output:

1.



```
C:\Users\Owner\Desktop\Parser>java parser testfile3.txt
<Program>
   <stmt_list>
      <stmt>
         <read>
            read
         </read>
         <id>
            A
         </id>
      </stmt>
      <stmt_list>
         <stmt>
            <read>
               read
            </read>
            <id>
               B
            </id>
         </stmt>
         <stmt_list>
            <stmt>
               <read>
                  read
               </read>
               <id>
                  C
               </id>
            </stmt>
            <stmt_list>
               <stmt>
                  <id>
                     C
                  </id>
                  <assign>
                     :=
                  </assign>
                  <expr>
                     <term>
                        <factor>
                           <id>
                              B
                           </id>
                        </factor>
                        <factor_tail>
                        </factor_tail>
                     </term>
                     <term_tail>
                        <add_op>
```

2.

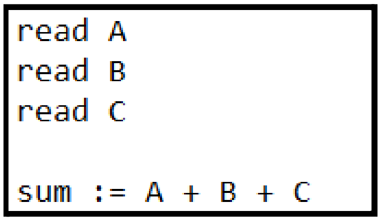


```
                        </add_op>
                        <term>
                           <factor>
                              <id>
                                 A
                              </id>
                           </factor>
                           <factor_tail>
                           </factor_tail>
                        </term>
                        <term_tail>
                        </term_tail>
                     </term_tail>
                  </expr>
               </stmt>
               <stmt_list>
               </stmt_list>
            </stmt_list>
         </stmt_list>
      </stmt_list>
   </stmt_list>
</Program>

C:\Users\Owner\Desktop\Parser>
```

Test Case 4:

Input:



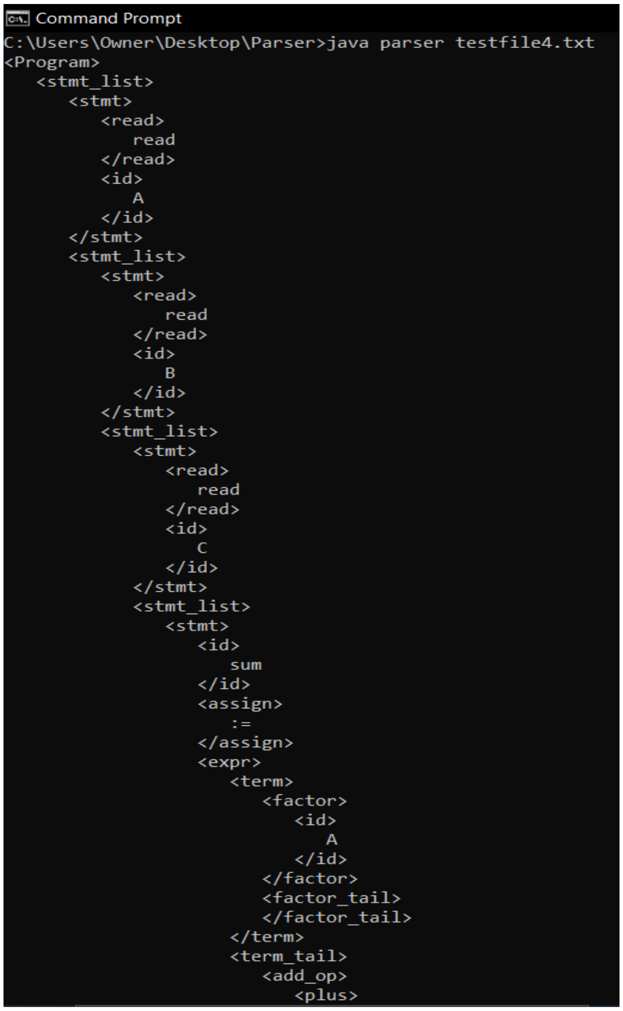
```
read A
read B
read C

sum := A + B + C
```

<===== ***test4.txt***

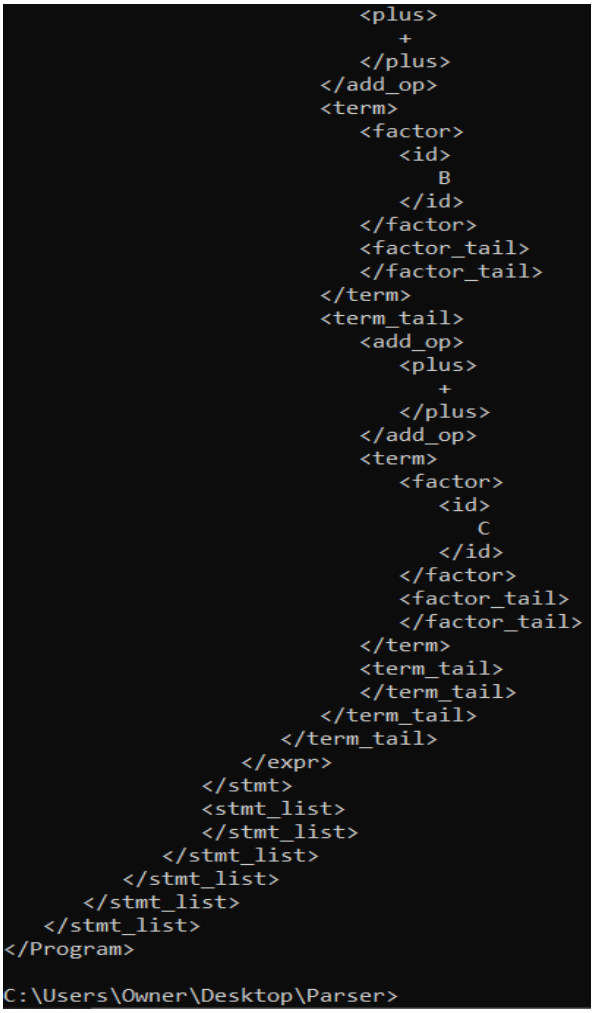
Output:

1.



```
C:\Users\Owner\Desktop\Parser>java parser testfile4.txt
<Program>
   <stmt_list>
      <stmt>
         <read>
            read
         </read>
         <id>
            A
         </id>
      </stmt>
      <stmt_list>
         <stmt>
            <read>
               read
            </read>
            <id>
               B
            </id>
         </stmt>
         <stmt_list>
            <stmt>
               <read>
                  read
               </read>
               <id>
                  C
               </id>
            </stmt>
            <stmt_list>
               <stmt>
                  <id>
                     sum
                  </id>
                  <assign>
                     :=
                  </assign>
                  <expr>
                     <term>
                        <factor>
                           <id>
                              A
                           </id>
                        </factor>
                        <factor_tail>
                        </factor_tail>
                     </term>
                     <term_tail>
                        <add_op>
                           <plus>
```

2.

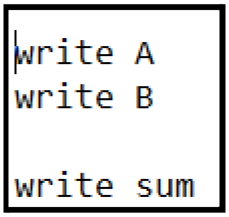


```
                           <plus>
                              +
                           </plus>
                        </add_op>
                        <term>
                           <factor>
                              <id>
                                 B
                              </id>
                           </factor>
                           <factor_tail>
                           </factor_tail>
                        </term>
                        <term_tail>
                           <add_op>
                              <plus>
                                 +
                              </plus>
                           </add_op>
                           <term>
                              <factor>
                                 <id>
                                    C
                                 </id>
                              </factor>
                              <factor_tail>
                              </factor_tail>
                           </term>
                           <term_tail>
                           </term_tail>
                        </term_tail>
                     </term_tail>
                  </expr>
               </stmt>
               <stmt_list>
               </stmt_list>
            </stmt_list>
         </stmt_list>
      </stmt_list>
   </stmt_list>
</Program>

C:\Users\Owner\Desktop\Parser>
```

<u>Test Case 5:</u>

Input:



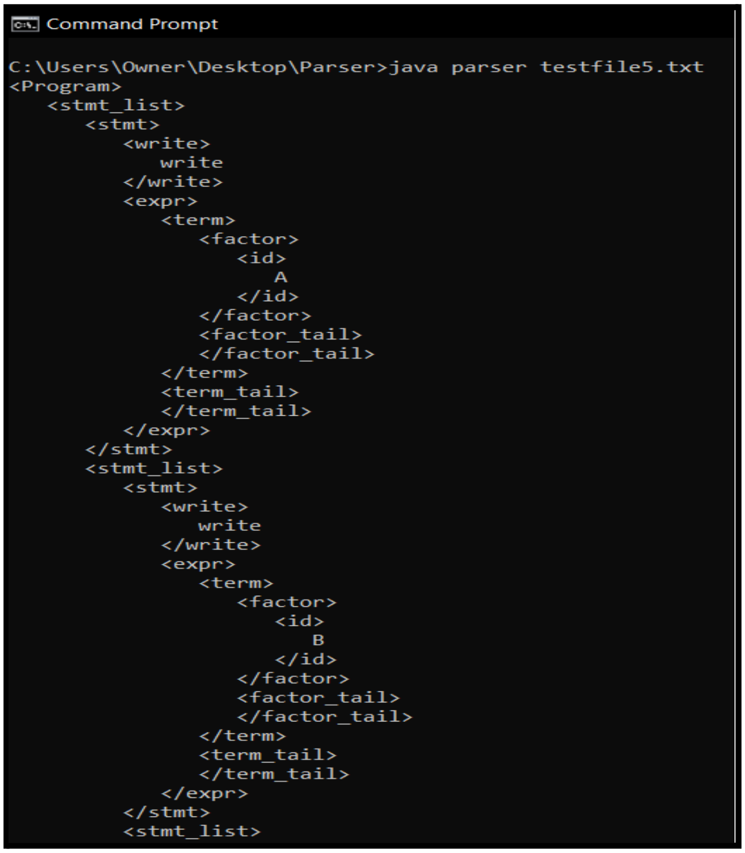
```
write A
write B

write sum
```

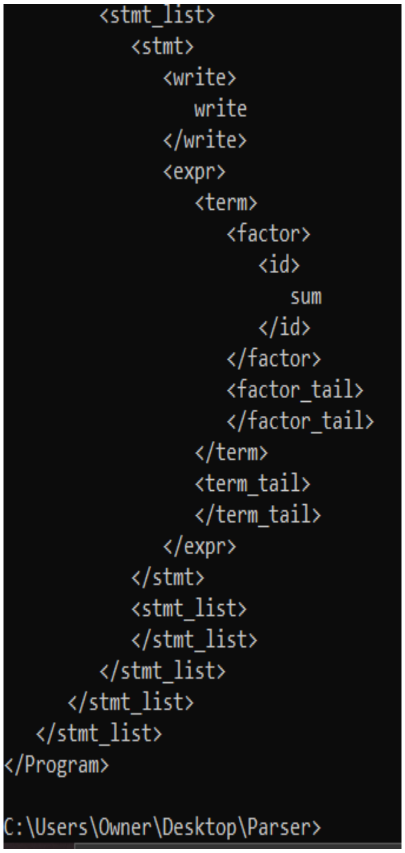
<====== ***testfile5.txt***

Output:

1. ***2.***



```
C:\Users\Owner\Desktop\Parser>java parser testfile5.txt
<Program>
   <stmt_list>
      <stmt>
         <write>
            write
         </write>
         <expr>
            <term>
               <factor>
                  <id>
                     A
                  </id>
               </factor>
               <factor_tail>
               </factor_tail>
            </term>
            <term_tail>
            </term_tail>
         </expr>
      </stmt>
      <stmt_list>
         <stmt>
            <write>
               write
            </write>
            <expr>
               <term>
                  <factor>
                     <id>
                        B
                     </id>
                  </factor>
                  <factor_tail>
                  </factor_tail>
               </term>
               <term_tail>
               </term_tail>
            </expr>
         </stmt>
         <stmt_list>
```



```
         <stmt_list>
            <stmt>
               <write>
                  write
               </write>
               <expr>
                  <term>
                     <factor>
                        <id>
                           sum
                        </id>
                     </factor>
                     <factor_tail>
                     </factor_tail>
                  </term>
                  <term_tail>
                  </term_tail>
               </expr>
            </stmt>
            <stmt_list>
            </stmt_list>
         </stmt_list>
      </stmt_list>
   </stmt_list>
</Program>

C:\Users\Owner\Desktop\Parser>
```

5. ***Acknowledgements***

Special thanks to,

- ***Yuanlin Zhang (Dr. Zhang)*** for his useful vocal discussions regarding the contents of our project, as well as providing us with constructive advice.
- ***Caleb Canales*** for his assistance on how to properly implement the current state and next state token identifier when we were constructing a new scanner in Java, so that its implementation into our parser was as accurate as possible.
- ***Eric Ward*** for his assistance on how to properly implement the current state and next state token identifier when we were constructing a new scanner in Java, so that its implementation into our parser was as accurate as possible.
- ***Ryan Lee*** for his assistance on how to properly implement the current state and next state token identifier when we were constructing a new scanner in Java, so that its implementation into our parser was as accurate as possible.