YOU MAY RECEIVE A ZERO IF ANY PART OF YOUR CODE IS TO SIMILAR TO ANOTHER STUDENTS PAST OR PRESENT

1. (3 points) Develop a class/struct for Token, **must contain**:
   1. String for lexeme representation ✓
   2. Int for token code ✓
2. (2 Points) Develop a **Compiler Class** that ✓
   1. Has a method that takes in an ***INPUT FILE*** and converts it to one input str ✓
3. (15 points) Develop a **Lexer** class: (Code 10, definitions 5)
   1. An instant of this class should exist In the **Complier Class** ✓
   2. Takes in a string in its constructor ✓
   3. Converts a string into a list of Token object if there exist no errors ✓
      * 1. Should ignore block comments ✓
        2. Should ignore single line comments ✓
   4. Should contain the following tokens and ***clear patterns or automata to recognize them*** (in the comments each should be specified) ✓
      * 1. real\_literal represents fractional number ✓
        2. natural\_literal represent whole numbers and 0 ✓
        3. bool\_literal ✓
        4. char\_liter represents a single ascii charater including escape character ✓

1. Java rules for escape character ✓

v. string\_literal represents a any number of ascii charater including escape character ✓

1. Java rules for escape character vi. Keywords For

Selection statement ✓

Loop Statement ✓

Variable declaration for ✓

Strings ✓

Naturals ✓

Character ✓

Reals ✓

Booleans ✓

vii. Special Symbols for

Addition ✓

Subtraction ✓

Multiplication ✓

Division ✓

Exponentiation ✓

Symbol(s) to specify the breaking order of operations ✓

Greater than ✓

Less Than ✓

Greater Than or equal too (must be more than two characters) ✓

Less Than or equal too (must be more than two characters) ✓

Equal to ✓

Not equal too (must be more than two characters) ✓

unary negation operator ✓

Logical Not ✓

Logical And ✓

Logical Or ✓

Symbol(s) of grouping code blocks ✓

Parameter separator ✓

Symbol(s) to specify the parameters of a function ✓

viii. Variable/function identifier ✓

1. (20 points) Develop a **Parser** class: Code 10 definitions 10
   1. An instant of this class should exist In the **Complier Class**
   2. Takes in aa list of Token object in its constructor
   3. Outputs a parse tree of called functions that would recognize the input is syntactically correct
   4. **REQUIRE YOU CREATE GRAMMAR RULES THAT WOULD SATISFY A TOP DOWN**

**PARSER**

* 1. Should be coded in the style of a recursive decent parser
  2. Code should be able to handle multiple code statements

**i.** A valid code file should be able to have 0 or many valid statements

* 1. A statement should be able to be one of the following
     + 1. Code block
       2. Selection Statement
       3. Loop Statement i
       4. v. Assignment Statement

Should be able to assign an expression to a variable

Expressions should be able to have Boolean solutions

Operands in expressions can be variables, real\_literal, natural\_literal, bool\_literal, charl\_literal, string\_literal, function\_call

Expressions should allow for unary negation operator

a. If this symbol comes after any sumbol outside of the assignment operator or the opening symbol(s) to specify the breaking order of operations it should require it to be in the symbol(s) to specify the breaking order of operations

v. Declaration statement

Variables

Functions Definition

1. (5 points) use Denotational semantics to define your selection statement   
   1. (5 points) use Denotational semantics to define your loop statement   
      **<while> 🡪 while<bool> <stmt\_list>   
      m\_loop ( while <bool> <stmt\_list> , s ) 🡺  
      if M\_b (<bool> , s) == error**

**Return error   
if M\_b (<bool> , s) == false**

**Return s  
else   
if m\_sl (LS,s) == error   
return error**A hand holding a piece of paper

Description automatically generated with low confidence

Text, letter

Description automatically generated

1. (10 points) use Denotational semantics to define your Expr statement   
   **<expr> -> <real\_literal > | <natural\_literal > | <bool\_literal> | <binary\_expr>|<var>  
   M\_e (<expr> , s ) -🡪 if <expr> == <natural\_literal >**

**Return M\_nat(<nat\_num>)**

**Else if( <expr> ==<bool\_lit>)  
Return M\_bool(<bool\_lit>)  
Else if( <expr> ==<real\_lit>)  
Return M\_real(<real\_lit>)**

**If VARMAP[<vat> ,s ] == undef  
return error**

**Else**

**return VARMAP[<var>, s]  
else**

**if M\_e(<left\_expr> , s ) == error**

**return error**

**if M\_e(<right \_expr> , s ) == error**

**return error**

Text, letter

Description automatically generated

1. (5 points) use Denotational semantics to redefine your Expr statement so it can return a Boolean solution   
   **M\_e (<expr> , s ) -🡪 if <expr> == <bool\_literal >**

Return m\_e (<bool\_literl>)

**If VARMAP[<bool> ,s ] == bool  
return error**

**Else**

**return VARMAP[<var>, s]**

Text, letter

Description automatically generated

1. (10 points) Define the attribute grammar for your assignment statement, make sure it follows the following rules
   * 1. String + String does concatenation   
        <assign> 🡪 <String> + <String>
     2. String \* Natural repeats the Natural   
        <assign> -> <String> \* <natural\_literal>
     3. Assign bool to natural is allowed   
        <assign> -> <bool\_literal> = <natural\_literal>
     4. Assign natural to bool is allowed   
        <assign> -> <natural\_literal> = <bool\_literal>
     5. Assign char to natural is allowed   
        <assign> -> <char\_literal> = <natural\_literal>
     6. Assign natural to char is allowed

<assign> -> <natural\_literal> = <char\_literal>

* + 1. Assign natural to real is allowed   
       <assign> -> <natural\_literal> = <real\_literal>
    2. No other types are allowed to be assigned to others outside of their own
    3. Dividing by zero is an error   
       <assign> -> <var> </>
    4. Modulo operating by zero is an error

Text, letter

Description automatically generated

 <assign> → <String>| <var> = <expr>

<expr>.expected\_type ← <var>.actual\_type

<expr> → <var>[2] + <var> [3]

<expr>.actual\_type ←

If (<var>[2],actual\_type = int) and

(<var> [3].actual\_type = int)

then int

else real

end if

<expr>.actual\_type == <expr>.expected\_type

1. (15 point) choose 3 syntactically valid assignment statements with at least 7 tokens to show these rules failing or passing semantic rules   
   Passing   
   int a = 10: // assign allowed

Int b= 20; // assign allowed

String y =”hello”;

Char c =’c’;  
If(a<b){

A+b=b;  
} else {

B=a;

}

Int x =0;

String y =”hello”;

Char c =’c’;

if (x < 1){  
x/0;  
System.out.print(“error");

int x =10;  
int y=5;

String y =”hello”;

Char c =’c’;

while (x>y){

system.out.print(“please I tried my best “);

1. ( 10 points ) Axiomatic Semantics ( find the weakest precondition) :

a.

a = 2 \* (b - 1) - 1 {a > 0}   
**a= 2\*(b-1)-1  
 +1  
---------------------  
a= 2\* (b-1) > 1  
 \2 \2  
---------------------  
a = (b-1) > ½  
 +1 +1  
---------------------  
b > 3/2**

b.

if (x < y) x = x +1 else

x = 3 \* x

{x < 0}

**If(x<y)  
x<-1  
else  
🡪 x->1  
x = 3\*x**

c.

y = a \* 2 \* (b - 1) - 1

if (x < y) x = y + 1 else x = 3 \* x

{x < 0}   
**y <-1**   
**if(x<y)  
y<-1  
x=y+1  
else  
🡪 y<- 1  
x=3\*x**

d.

a = 3 \* (2 \* b + a); b = 2 \* a - 1

{b > 5}   
**b=2\*a-1>5  
+1  
------------------  
2\*a > 6  
/2  
--------------  
a > 3  
3\*(2\*b+a) > 3  
2\*b > 1-a  
🡪 b> (1-a)/2**