Caige Middaugh
10/24/21
Programming Languages Paradigm

Project 1: Lexical Analyzer

Changes:

- elif and changed to where if, elif, and else can be used appropriately. Else is not required.
- Added simple arrays.
- For loop
- Functions (depending on complexity I may leave them out of implementation)
- Printf statement (the %s and %d only but does get tokenized as meta data.)
- General grammar changes.
- Added static semantic predicate for where a return value is required when calling a function.
- Added static semantic predicate for checking if %s and %d is correct and if the correct amount was assigned in a printf statement.

Explanation:

Grammar: I attempted to keep the Grammar as a LL Grammar, but I have noticed some issues with it that make it not a LL Grammar that will take a long time to change because I would have to then change all the static semantics which is a lot of manual editing of lines. Due to the sake of time and complexity of the project I will have to come back to this issue later.

First, I rearranged the grammar to separate integer and string values so I can easily separate the operations such as ',' where I want to concat a string or integer. However, this led to a lot of difficulties with maintaining a LL grammar and as said before may be revisited later. After I finished my operational semantics the time it took to make changes to the grammar became exponential because I had to re number and change every rule in the static semantics. For part 2 I will be reworking the grammar rules, so they will change, but the general syntax rules I want and changes I added above will remain.

In this document I provided the screenshots of what I wrote in VS Code because it did the numbering for me automatically by numbering the lines. However, for the static semantics I had to do the numbering manually because I would have to skip certain rules. I have included a copy paste version of the grammar at the bottom of the document in case you need that.

Static Semantics: I attempted to implement the predicate changes the best way I knew how. Simulating scope with the functions may cause a bit of an issue with how I executed it in the static semantics. However, due to the complexity of the grammar I cannot see any obvious errors with it right now. The rules that go upward in the tree I intend to be executed after the branch has been parsed out, as per explained in lecture with inherited and Synthesized attributes.

Denotational Semantics: I found this process to be very time consuming and complicated for my specific grammar. I saved this step for last so I could have some time left to complete the lexer because the lexer code is more important. I put my best foot forward with how I think denotational semantics should work, but the grammar is large and complex. Overall, this process required the most time, but was on the lower end of the priority list for me. Due to the time constraints and the low priority of this section I had to rush through a lot of the definitions.

State Diagram: I designed this to be a general guide for how to write my lexer code. However, my lexer code does not follow it 100% because some things I wanted to do such as using regular expression to separate STRINGF and STRINGS I was not sure how to put on the diagram. On top of that I was using freeware to develop the state diagram and I was not aware the website had a limit of how many items we can but on the diagram. I eventually hit a pay wall, but I was already done and just used function calls to finish out the diagram because I was doing that in my lexer.

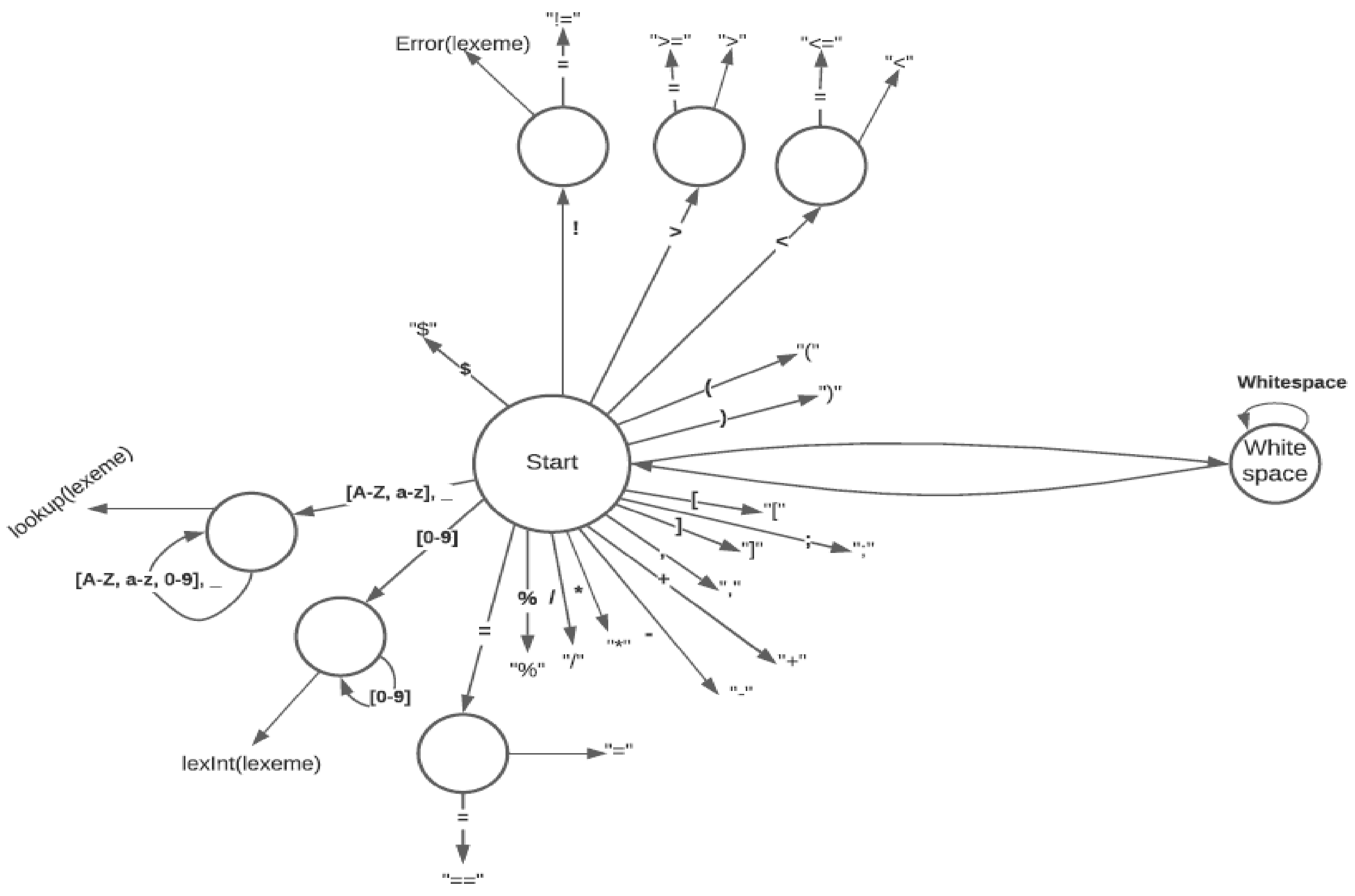
Lexer Code: Inside the lexer code there is 3 classes, Symbol, BuiltInTypeSumbol, and VarSymbol. I had started working on a symbol table, but due to time constraints I needed to work on the denotation semantics and was not able to finish the symbol table. I have included it just to show that I put effort towards making a symbol table. However, from my understanding a symbol table is optional. I just wanted to do it to help during part 2.

My lexer is a class that doesn't use splicing. It uses an index to iterate through the input source code. This solution is faster and better especially if there is long source code because you don't have to copy the input every single time. The lexer code is set up to give one token at a time.

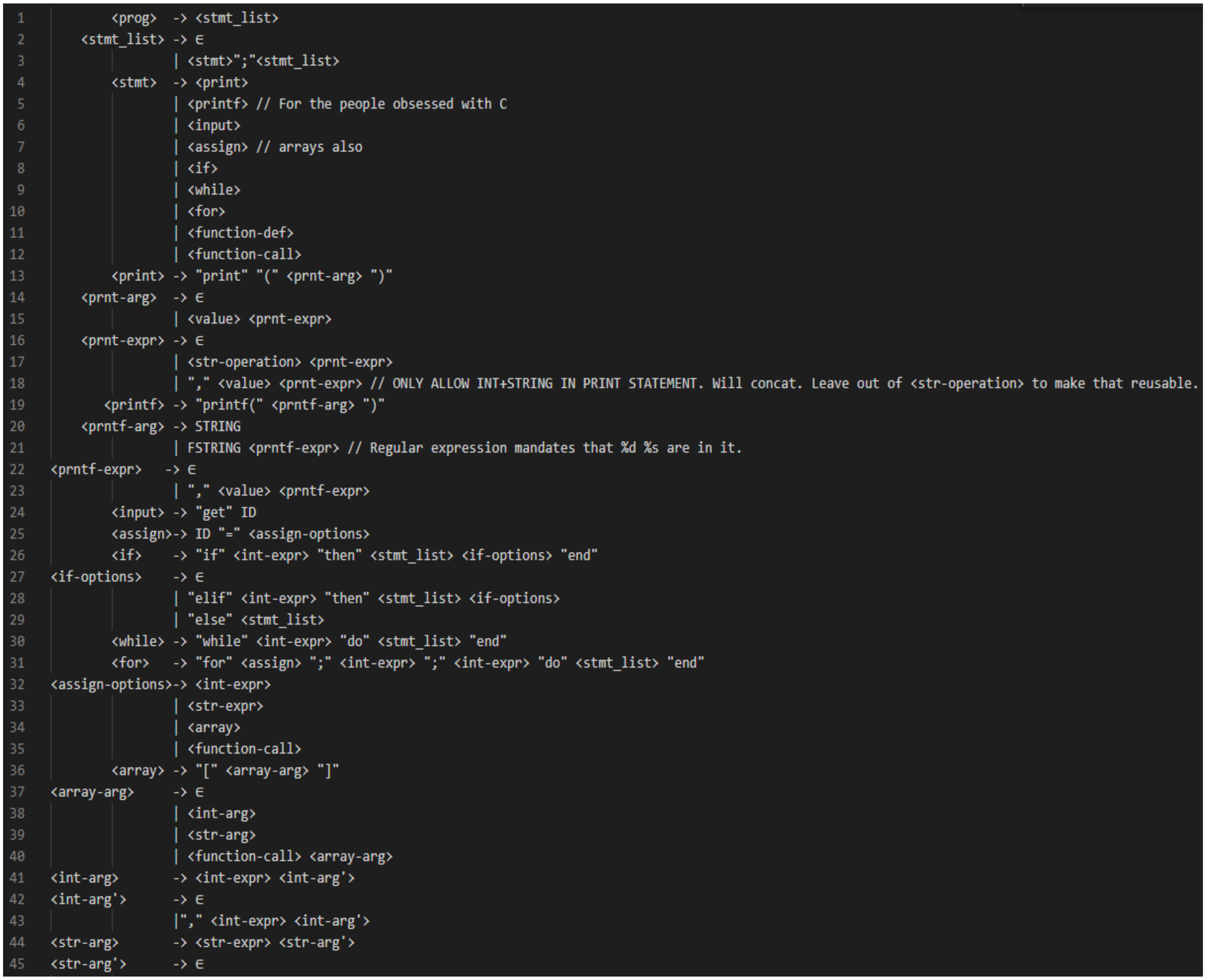
Conclusion:

Overall, the project took me a very long time. I personally struggled with keeping the grammars LL grammars and I did the static semantics early on. That means I had to keep changing the entire list of static semantics even if I made a single change in my grammar. That process is what took me the longest and wasted a lot of time. That is why I got to the point of turning it in as perfect because I had to move on. To summarize, I struggled the most with the semantic sections because they were very time consuming and tedious. It was easy to overlook a simple mistake that was then recursive throughout the whole semantics.

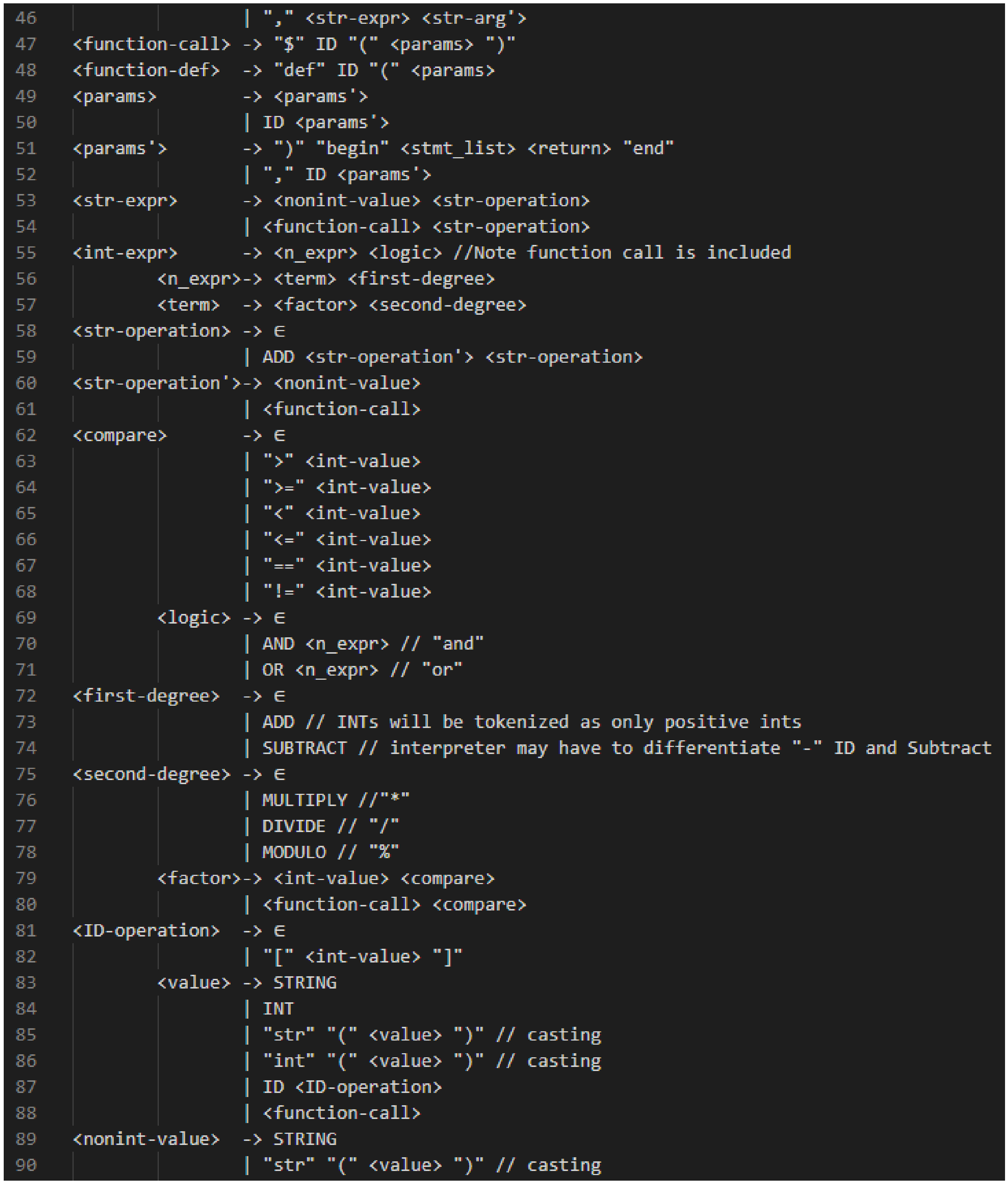
State Transition Diagram:



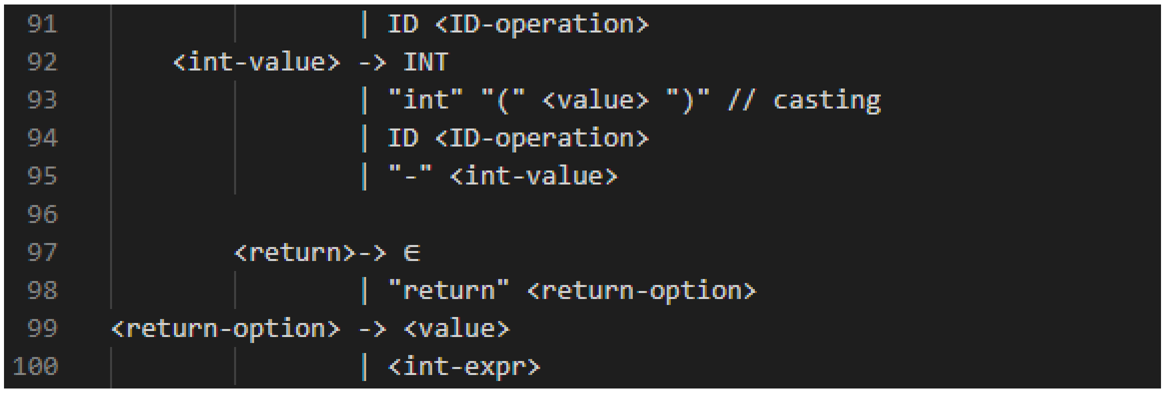
BNF GRAMMAR:



```
        <prog>  -> <stmt_list>
    <stmt_list> -> ∈
                | <stmt>";"<stmt_list>
        <stmt>  -> <print>
                | <printf> // For the people obsessed with C
                | <input>
                | <assign> // arrays also
                | <if>
                | <while>
                | <for>
                | <function-def>
                | <function-call>
        <print> -> "print" "(" <prnt-arg> ")"
    <prnt-arg>  -> ∈
                | <value> <prnt-expr>
    <prnt-expr> -> ∈
                | <str-operation> <prnt-expr>
                | "," <value> <prnt-expr> // ONLY ALLOW INT+STRING IN PRINT STATEMENT. Will concat. Leave out of <str-operation> to make that reusable.
       <printf> -> "printf(" <prntf-arg> ")"
    <prntf-arg> -> STRING
                | FSTRING <prntf-expr> // Regular expression mandates that %d %s are in it.
<prntf-expr>   -> ∈
                | "," <value> <prntf-expr>
        <input> -> "get" ID
        <assign>-> ID "=" <assign-options>
        <if>    -> "if" <int-expr> "then" <stmt_list> <if-options> "end"
<if-options>    -> ∈
                | "elif" <int-expr> "then" <stmt_list> <if-options>
                | "else" <stmt_list>
        <while> -> "while" <int-expr> "do" <stmt_list> "end"
        <for>   -> "for" <assign> ";" <int-expr> ";" <int-expr> "do" <stmt_list> "end"
<assign-options>-> <int-expr>
                | <str-expr>
                | <array>
                | <function-call>
        <array> -> "[" <array-arg> "]"
<array-arg>     -> ∈
                | <int-arg>
                | <str-arg>
                | <function-call> <array-arg>
<int-arg>       -> <int-expr> <int-arg'>
<int-arg'>      -> ∈
                |"," <int-expr> <int-arg'>
<str-arg>       -> <str-expr> <str-arg'>
<str-arg'>      -> ∈
```



```
                | "," <str-expr> <str-arg'>
<function-call> -> "$" ID "(" <params> ")"
<function-def>  -> "def" ID "(" <params>
<params>        -> <params'>
                | ID <params'>
<params'>       -> ")" "begin" <stmt_list> <return> "end"
                | "," ID <params'>
<str-expr>      -> <nonint-value> <str-operation>
                | <function-call> <str-operation>
<int-expr>      -> <n_expr> <logic> //Note function call is included
        <n_expr>-> <term> <first-degree>
        <term>  -> <factor> <second-degree>
<str-operation> -> ∈
                | ADD <str-operation'> <str-operation>
<str-operation'>-> <nonint-value>
                | <function-call>
<compare>       -> ∈
                | ">" <int-value>
                | ">=" <int-value>
                | "<" <int-value>
                | "<=" <int-value>
                | "==" <int-value>
                | "!=" <int-value>
        <logic> -> ∈
                | AND <n_expr> // "and"
                | OR <n_expr> // "or"
<first-degree>  -> ∈
                | ADD // INTs will be tokenized as only positive ints
                | SUBTRACT // interpreter may have to differentiate "-" ID and Subtract
<second-degree> -> ∈
                | MULTIPLY //"*"
                | DIVIDE // "/"
                | MODULO // "%"
        <factor>-> <int-value> <compare>
                | <function-call> <compare>
<ID-operation>  -> ∈
                | "[" <int-value> "]"
        <value> -> STRING
                | INT
                | "str" "(" <value> ")" // casting
                | "int" "(" <value> ")" // casting
                | ID <ID-operation>
                | <function-call>
<nonint-value>  -> STRING
                | "str" "(" <value> ")" // casting
```



```
                        | ID <ID-operation>
    <int-value> -> INT
                        | "int" "(" <value> ")" // casting
                        | ID <ID-operation>
                        | "-" <int-value>

        <return>-> ∈
                        | "return" <return-option>
<return-option> -> <value>
                        | <int-expr>
```

STATIC SEMANTICS:

1. <stmt_list>.ids = {}
3. <stmt_list>[0].ids = {<stmt>.id} U <stmt_list>[0].ids
    <stmt>.ids = <stmt_list>[0].ids
4. <print>.ids = <stmt>.ids
5. <printf>.ids = <stmt>.ids
6. <stmt>.id = <input>.id
7. <stmt>.id = <assign>.id
   <assign>.ids = <stmt>.ids
8. <if>.ids = <stmt>.ids
9. <while>.ids = <stmt>.ids
10. <for>.ids = <stmt>.ids
11. <function-def>.ids = {}
    <stmt>.id = <function-def>.id
12. <function-call>.ids = <stmt>.ids
    <function-call>.ret = False
13. <prnt-arg>.ids = <print>.ids
15. <value>.ids = <prnt-arg>.ids
    <prnt-expr>.ids = <prnt-arg>.ids
17. <str-operation>.ids = <prnt-expr>[0].ids
    <prnt-expr>[1].ids = <prnt-expr>[0].ids
18. <value>.ids = <prnt-expr>[0].ids
    <prnt-expr>[1].ids = <prnt-expr>[0].ids
19. <prntf-arg>.ids = <prntf>.ids
21. <printf-expr>.ids = <prntf-args>.ids
    <printf-expr>.fill_ins = {} U STRINGF.fill_ins // fill in is %s or %d and is a set of them.
22. **Predicate:** <printf-expr>.fill_ins = {}
23. <value>.ids = <prntf-expr>[0].ids
    <prntf-expr>[1].ids = <prntf-expr>[0].ids
    <value>.fill_in = <prntf-expr>[0].fill_ins[0]
    <printf-expr>[1].fill_ins = <prntf-expr>[0].fill_ins - <prntf-expr>[0].fill_ins[0]
24. <input>.id = ID.id
25. <assign>.id = ID.id
    <assign-options>.ids = <assign>.ids
26. <int-expr>.ids = <if>.ids
    <stmt_list>.ids = <if>.ids

    <if-options>.ids = <if>.ids
28. <int-expr>.ids = <if-options>[0].ids
    <stmt_list>.ids = <if-options>[0].ids
    <if-options>[1].ids = <if-options>[0].ids
29. <stmt_list>.ids = <if-options>.ids
30. <int-expr>.ids = <while>.ids
    <stmt_list>.ids = <while>.ids
31. <for>.ids = <assign>[0].id U <for>.ids
    <for>.ids = <assign>[1].id U <for>.ids
    <int-expr>.ids = <for>.ids
32. <int-expr>.ids = <assign-options>.ids
33. <str-expr>.ids = <assign-options>.ids
34. <array>.ids = <assign-options>.ids
35. <function-call>.ids = <array>.ids
    <funciton-call>.ret = True
36. <array-arg>.ids = <array>.ids
38. <int-arg>.ids = <array-arg>.ids
39. <str-arg>.ids = <array-arg>.ids
40. <function-call>.ids = <array-arg>[0].ids
    <function-call>.ret = True
    <array-arg>[1].ids = <array-arg>[0].ids
41. <int-expr>.ids = <int-arg>.ids
    <int-arg'>.ids = <int-arg>.ids
43. <int-expr>.ids = <int-arg'>[0].ids
    <int-arg'>[1].ids = <int-arg'>[0].ids
44. <str-expr>.ids = <str-arg>.ids
    <str-arg'>.ids = <str-arg>.ids
46. <str-expr>.ids = <str-arg'>[0].ids
    <str-arg'>[1].ids = <str-arg'>[0].ids
47. **Predicate:** ID.id ∈ <function-call>.ids
    Predicate: ID.ret = <function-call>.ret
    <params>.ids = <function-call>.ids
48. <function-def>.id = ID.id
    <params>.ids = <function-def>.ids U ID.id
    ID.ret = <params>.ret
49. <params'>.ids = <params>.ids
    <params'>.ret = <params>.ret
    <params>.ret = <params'>.ret
50. <params>.ids = <params>.ids U ID.id
    <params'>.ids = <params>.ids
    <params'>.ret = <params>.ret
    <params>.ret = <params'>.ret
51. <stmt_list>.ids = <params'>.ids
    <return>.ids = <params'>.ids
    <return>.ret = <params'>.ret
    <params'>.ret = <return>.ret //On the way up the tree
52. <params'>[0].ids = <params'>[0].ids U ID.id
    <params'>[1].ids = <params'>[0].ids
53. <nonint-value>.ids = <str-expr>.ids

<str-operation>.ids = <str-expr>.ids

54. <function-call>.ids = <str-expr>.ids
    <function-call>.ret = True
    <str-operation>.ids = <str-expr>.ids
55. <n_expr>.ids = <int-expr>.ids
    <logic>.ids = <int-expr>.ids
56. <term>.ids = <n_expr>.ids
    <first-degree>.ids = <n_expr>.ids
57. <factor>.ids = <term>.ids
    <second-degree>.ids = <term>.ids
59. <first-degree>.ids = <str-operation>[0].ids
    <str-operation'>.ids = <str-operation>[0].ids
    <str-operation>[1].ids = <str-operation>[0].ids
60. <nonint-value>.ids = <str-operation'>.ids
61. <function-call>.ids = <str-operation'>
    <function-call>.ret = True
63. <int-value>.ids = <compare>.ids
64. <int-value>.ids = <compare>.ids
65. <int-value>.ids = <compare>.ids
66. <int-value>.ids = <compare>.ids
67. <int-value>.ids = <compare>.ids
68. <int-value>.ids = <compare>.ids
70. <n_expr>.ids = <logic>.ids
71. <n_expr>.ids = <logic>.ids
79. <int-value>.ids = <factor>.ids
    <compare>.ids = <factor>.ids
80. <function-call>.ids = <factor>.ids
    <function-call>.ret = True
    <compare>.ids = <factor>.ids
82. <int-value>.ids = <ID-operation>.ids
83. **Predicate:** <value>.fill_in = %s OR <value>.fill_in = {} // empty if not printF line 22 proves it will
    // line 22 proves it will error out if blank value passed.
84. **Predicate:** <value>.fill_in = %d OR <value>.fill_in = {}
85. <value>[1].ids = <value> [0].ids
    <value>[1].fill_in = <value>[0].fill_in
86. <value> [1].ids = <value> [0].ids
    <value>[1].fill_in = <value>[0].fill_in
87. <ID-operation>.ids = <value>.ids
    Predicate: if ID.type = int then <value>.fill_in = %d else <value>.fill_in = %s
88. <function-call>.ids = <value.ids>
    <function-call>.ret = True
90. <value>.ids = <nonint-value>.ids
91. **Predicate:** ID.id ∈ <nonint-value>.ids
    <ID-operation>.ids = <nonint-value>.ids
93. <value>.ids = <int-value>.ids
94. **Predicate:** ID.id ∈ <int-value>.ids
    <ID-operation>.ids = <int-value>.ids
95. <int-value>[1].ids = <int-value>[0].ids
97. <return>.ret = False

98. <return options>.ids = <return>.ids
    <return>.ret = True
99. <value>.ids = <return-options>.ids
100. <int-expr>.ids = <return-options>.ids

Dynamic Semantics:

```
densem(∈,(θ, i, p)) = (θ, i, p)
densem(<stmt> “;” <stmt_list>,(θ, i, p)) = densem(<stmt_list>, densem(<stmt>,(θ, i, p)))
densem("print" "(" <prnt-arg> ")", (θ, i, p)) = prntargsem(<prnt-arg>, (θ, i, p))
densem("printf" "(" <prntf-arg> ")", (θ, i, p)) = prntfargsem(<prntf-arg>, (θ, i, p))
densem(<while>, (θ, i, p)) = if intexprsem(<while.<expr>, θ) = 0
            then (θ, i, p)
            else densem(<while>, densem(<while.<stmt_list>, (θ, i, p)))
densem("get" ID, (θ, i, p)) = (θ', i', p)
            where (x, i') = getInt(clean(i))
            θ'(n) = if n = ID then x else θ(n)
densem(<if>, (θ, i, p)) = if intexprsem(<if>.<expr>, θ) != 0
            then densem(<if>.<stmt_list>[0], (θ, i, p))
            else ifoptionsem(<if>.<if-options>, (θ, i, p)))
densem(<function-def, (θ, i, p)) = funcSem("def" ID "(" <params>, (θ, i, p))
densem(<function-call>, (θ, i, p)) = funcSem(ID "(" <params> ")", (θ, i, p))
densem(<assign>, (θ, i, p)) = densem(ID "=" <assign-options>, (θ, i, p))
densem(ID "=" <assign-options>, (θ, i, p)) = (θ', i, p)
            where θ'(n) = if n = ID then assignoptionsem(<assign-option>, (θ, i, p)) else θ(n)
densem(<for>, (θ, i, p)) = forsem(<for>, (θ, i, p))
```

#PRINT

```
prntargsem(∈, (θ, i, p)) = (θ,i,append(p, "\n"))
prntargsem(<value> <prnt-expr>, (θ, i, p)) =
            prntargsem(<prnt-expr>, (θ,i,append(p, valsem(<value>, (θ, i, p)))))
```

#PRINTF

```
prntfargsem(STRING, (θ, i, p)) = (θ, i, append(p, STRING)
prntfargsem(FSTRING <prntf-expr>, (θ, i, p)) =
      prntfexprsem(<prntf-expr>, FSTRING, [], (θ, i, p)) // [] is empty stmt_list
prntfexprsem(∈, FSTRING, v, (θ, i, p)) =
      (θ, i, append(p, subFSTRING(FSTRING, v))
prntfexprsem("," <value> <prntf-expr>, FSTRING, v, (θ, i, p)) =
      prntfexprsem(<prntf-expr>, FSTRING, v.append(valsem(<value>, (θ, i, p))), (θ, i, p))
```

#IF

```
ifoptionsem(∈, (θ, i, p)) = ∈
ifoptionsem("elif" <int-expr> "then" <stmt_list> <if-options>, (θ, i, p)) =
            if intexprsem(<int-expr>, θ) != 0
            then densem(<stmt_list>, (θ, i, p))
            else ifoptionsem(<if-options>, (θ, i, p))
ifoptionsem("else" <stmt_list>,(θ, i, p)) = densem(<stmt_list>, (θ, i, p))
```

#ASSIGN

```
assignoptionsem(<int-expr>, (θ, i, p)) = intexprsem(<int-expr>, θ)
assignoptionsem(<str-expr>, (θ, i, p)) = strexprsem(<str-expr>, θ)
assignoptionsem(<array>, (θ, i, p)) = arraysem(<array>, (θ, i, p))
assignoptionsem(<function-call>, (θ, i, p)) = funcSem(<function-call>, (θ, i, p))
```

#ARRAY

```
arraysem("[" <array-arg> "]", (θ, i, p)) = arrayargsem(<array-arg>, (θ, i, p))
arrayargsem(∈, (θ, i, p)) = ∈
arrayargsem(<int-arg>, (θ, i, p)) = intargsem(<int-arg>, (θ, i, p))
arrayargsem(<str-arg>, (θ, i, p)) = strargsem(<str-arg>, (θ, i, p))
arrayargsem(<function-call> <array-arg>, (θ, i, p))=
        arrayargsem(<array-arg>, funcsem(<function-call, (θ, i, p)))
intargsem(∈, (θ, i, p) = ∈
intargsem(<int-expr> <int-arg'>, (θ, i, p)) =
        if intargsem(<int-arg'>, (θ, i, p)) = ∈
        then intexprsem(<int-expr>, θ)
        else intargsem(<int-arg'>, intexprsem(<int-expr>, θ))
intargsem("," <int-expr> <int-arg'>, (θ, i, p))=
        if intargsem(<int-arg'>, (θ, i, p)) = ∈
        then intexprsem(<int-expr>, θ)
        else intargsem(<int-arg'>, intexprsem(<int-expr>, θ))
strargsem(∈, (θ, i, p)) = ∈
strargsem(<str-expr> <str-arg'>, (θ, i, p)) =
        if strargsem(<str-arg'>, (θ, i, p)) = ∈
        then strexprsem(<str-expr>, θ)
        else strargsem(<str-arg'>, strexprsem(<str-expr>, θ))
strargsem("," <str-expr> <str-arg'>, (θ, i, p)) =
        if strargsem(<str-arg'>, (θ, i, p)) = ∈
        then strexprsem(<str-expr>, θ)
        else strargsem(<str-arg'>, strexprsem(<str-expr>, θ))
```

#FOR

```
forsem("for" <assign> ";" <int-expr> ";" <int-expr> "do" <stmt_list> "end" , (θ, i, p)) =
        θ' = densem(<assign>, (θ, i, p))
        if intexprsem(<int-expr>[0], θ') = 1
        then loop(<int-expr>[0], <int-expr>[1], <stmt_list>, (θ', i, p))
```

#EXPRESSION SEMANTICS (INT AND STR EXPRESSIONS)

```
intexprsem(<expr>, θ) = if <expr>.<b_expr> = ∈
                then intexprsem(<expr>.<n_expr>, θ)
                else logicsem(<expr>.<logic>, intexprsem(<expr>.<n_expr>),θ)
intexprsem(<n_expr>, θ) = if <n_expr>.<t_expr> = ∈
                then intexprsem(<n_expr>.<term>, θ))
                else firstdegsem(<n_expr>.<first-degree>, intexprsem(<n_expr>.<term>), θ)
intexprsem(<term>, θ) = if <term>.<second-degree> = ∈
                then intexprsem(<term>.<factor>, θ)
                else seconddegsem(<term>.<second-degree>, intexprsem(<term>.<factor>),θ)
intexprsem(<factor>, θ) = if <factor>.<v_expr> = ∈
                then intexprsem(<factor>.<value>, θ)
                else comparesem(<factor>.<compare>, valsem(<factor>.<int-value), (θ, i, p))
logicsem( "and" <n_expr>, v, θ) = if v != 0 and intexprsem(<n_expr>, θ) != 0 then 1 else 0
logicsem("or" <n_expr>, v, θ) = if v !- 0 or intexprsem( <n_expr>, θ) != 0 then 1 else 0
firstdegsem("+" <n_expr>, v, θ) = v + intexprsem( <n_expr>, θ)
firstdegsem("-" <n_expr>, v, θ) = v - intexprsem(<term>, θ)
seconddegsem("*" <term>, v, θ) = v * intexprsem(<term>, θ)
seconddegsem("/" <term>, v, θ) = v/ intexprsem(<term>, θ)
seconddegsem("%" <term>, v, θ) = v mod intexprsem(<term>, θ)
comparesem(">" <int-value>, v, θ) = if v > intexprsem(<int-value>, θ) then 1 else 0
comparesem(">=" <int-value>, v, θ) = if v >= intexprsem(<int-value>, θ) then 1 else 0
comparesem("<" <int-value>, v, θ) = if v < intexprsem(<int-value>, θ) then 1 else 0
comparesem("<=" <int-value>, v, θ) = if v <= intexprsem(<int-value>, θ) then 1 else 0
comparesem("!=" <int-value>, v, θ) = if v != intexprsem(<int-value>, θ) then 1 else 0
comparesem("==" <int-value>, v, θ) = if v = intexprsem(<int-value>, θ) then 1 else 0
strexprsem(<nonint-value> <str-operation>, (θ, i, p)) = if <str-operation> = ∈
        then valsem(<nonint-value>, (θ, i, p))
        else stroperationsem(<str-operation>, valsem(<nonint-value>, (θ, i, p)), (θ, i, p))
strexprsem(<function-call> <str-operation>, (θ, i, p)) = if <str-operation> = ∈
        then funcSem(<function-call>, (θ, i, p))
        else stroperationsem(<str-opearation>, funcsem(<nonint-value>, (θ, i, p)), (θ, i, p))
stroperationsem(∈, (θ, i, p)) = ∈
stroperationsem(<nonint-value>, (θ, i, p)) = valsem(<nonint-value>, (θ, i, p))
stroperationsem(<function-call>, (θ, i, p)) = funcSem(<function-call>, (θ, i, p))
stroperationsem(ADD <str-operation'> <str-operation>, v, (θ, i, p)) =
    if stroperationsem(<str-operation>, (θ, i, p)) = ∈
    then append(v, stroperationsem(<str-operation'>, (θ, i, p)))
    else stroperationsem(<str-operation>, append(v, stroperationsem(<str-operation'>, (θ, i, p))), (θ, i, p))
```

#FUNCTION SEMANTICS

```
funcSem("$" ID "(" <params> ")", (θ, i, p)) = if(ID ∈ θ)
    then funcSem(ID "(" <params> ")", funcSem(<params>, (θ, i, p)))
    else then p.append("error: ID not initialized")
funcSem("def" ID "(" <params>, (θ, i, p))) = funcSem(<params>, (θ', i, p))
    where θ'(n) = if n = ID then x else θ(n)
funcSem(ID <params'>, (θ, i, p)) = if(ID ∈ θ)
    then funcSem(ID <params'>, funcSem(<params'>, (θ, i, p)))
    else then p.append("error: ID not initialized")

funcSem(")" "begin" <stmt_list> <return> "end", (θ, i, p)) =
    funcSem(")" "begin" <stmt_list> <return> "end",
    funcSem(")" "begin" <stmt_list> <return> "end", denSem(<stmt_list>)))
funcSem("," ID <params'>, (θ, i, p)) = if(ID ∈ θ)
    then funcSem("," ID <params'>, funcSem(<params'>, (θ, i, p)))
    else then p.append("error: ID not initialized")
funcSem(<params'>, (θ, i, p)) = if(empty) then ∈
    else funcSem(<params'>)
funcSem(<return>, (θ, i, p)) = if(empty) then ∈
    elif(valsem(<return>) != error or null)  funcSem(<return>, intexprsem(<return>.<value>))// return
options eval
    else then funcSem(<return>, intexprsem(<return>.<expr>))
```

#ID-OPERATION SEMANTICS

```
IDOperationsem(∈, (θ, i, p)) = ∈
IDOperationsem( "[" <int-value> "]", (θ, i, p)) =
    valsem(IDOperationsem.<int-value>, (θ, i, p))
```

#VALUE SEMANTICS

```
valsem(STRING) = STRING
valsem(INT) = INT
valsem("str" "(" <value> ")",(θ, i, p)) = if(valsem(<value>) == STRING) then STRING
    elif(valsem(<value>) == INT) then append('',INT,'')
    elif(valsem(<value>) == ID and valsem(<value>.<ID-operation>) == ∈) then
        if(ID ∈ θ) then append('',ID,'')
        else then p.append("error: ID not initialized") return p, error
    elif(valsem(<value>) == ID and valsem(<value>.<ID-operation>) != ∈) then
        append('',IDOperationsem(<value>.<ID-operation>, (θ, i, p)),'')
    elif(valsem(<value>.<function-call>) != error) then
        append('',funcSem(<value>.<function-call>, (θ, i, p)),'')
    else then error
```

```
valsem("int" "(" <value> ")",(θ, i, p)) =
        if(valsem(<value>) == STRING) then
                removeQ(STRING)
        elif(valsem(<value>) == INT) then INT
        elif(valsem(<value>) == ID and valsem(<value>.<ID-operation>) == ∈) then
                if(ID ∈ θ) then removeQ(ID)
                else then p.append("error: ID not initialized") return p, error
        elif(valsem(<value>) == ID and valsem(<value>.<ID-operation>) != ∈) then
                removeQ(IDOperationsem(<value>.<ID-operation>))
        elif(valsem(<value>.<function-call>) != error) then
                removeQ(funcSem(<value>.<function-call>))
        else then error
valsem(ID <ID-operation>, (θ, i, p)) = if(ID ∈ θ)
        then IDOperationsem(<value>.<ID-operation>, (θ, i, p))
                                else then p.append("error: ID not initialized") return p, error
valsem(<function-call>, (θ, i, p)) = funcSem(<value>.<function-call>, (θ, i, p))
valsem("-" <int-value>, (θ, i, p)) = -valsem(<int-value>, (θ, i, p))
```

#HELPER FUNCTIONS

```
append(val1, val2, val3) = val1val2val3 // merges them together
append(val1, val2) = va1val2
removeQ(val) = val.intify() // function to turn into integer or remove quotes
getInt(i) = (int(x),i') where (x,i') = getIntSeq(∈, i)
getIntSeq(i1, i2) = if digit(head(i2)) then getIntSeq(append(i1, head(i2)), tail(i2))
loop(<int-expr>[0], <int-expr>[1],<stmt_list>, (θ', i, p)) if intexprsem(<int-expr>[0], θ') = 1
        then loop(<int-expr>[0], <int-expr>[1],<stmt_list>, intexprsem(<int-expr>[1], densem(<stmt_list>, (θ', i, p))))
subFSTRING(FSTRING, v) // a function that goes through and finds the first %s or %d
                        // then replaces it with the first element in v then v.pop()
```

COPY PASTE GRAMMAR:

```
        <prog>  -> <stmt_list>
    <stmt_list> -> ∈
                | <stmt>";"<stmt_list>
        <stmt>  -> <print>
                | <printf> // For the people obsessed with C
                | <input>
                | <assign> // arrays also
                | <if>
                | <while>
                | <for>
                | <function-def>
                | <function-call>
        <print> -> "print" "(" <prnt-arg> ")"
    <prnt-arg>  -> ∈
                | <value> <prnt-expr>
    <prnt-expr> -> ∈
                | <str-operation> <prnt-expr>
                | "," <value> <prnt-expr> // ONLY ALLOW INT+STRING IN PRINT STATEMENT. Will concat. Leave out of <str-operation> to make that reusable.
       <printf> -> "printf(" <prntf-arg> ")"
    <prntf-arg> -> STRING
                | FSTRING <prntf-expr> // Regular expression mandates that %d %s are in it.
<prntf-expr>   -> ∈
                | "," <value> <prntf-expr>
        <input> -> "get" ID
        <assign>-> ID "=" <assign-options>
        <if>    -> "if" <int-expr> "then" <stmt_list> <if-options> "end"
<if-options>    -> ∈
                | "elif" <int-expr> "then" <stmt_list> <if-options>
                | "else" <stmt_list>
        <while> -> "while" <int-expr> "do" <stmt_list> "end"
        <for>   -> "for" <assign> ";" <int-expr> ";" <int-expr> "do" <stmt_list> "end"
<assign-options>-> <int-expr>
                | <str-expr>
                | <array>
                | <function-call>
        <array> -> "[" <array-arg> "]"
<array-arg>     -> ∈
                | <int-arg>
                | <str-arg>
                | <function-call> <array-arg>
<int-arg>       -> <int-expr> <int-arg'>
<int-arg'>      -> ∈
                |"," <int-expr> <int-arg'>
<str-arg>       -> <str-expr> <str-arg'>
<str-arg'>      -> ∈
                | "," <str-expr> <str-arg'>
<function-call> -> "$" ID "(" <params> ")"
<function-def>  -> "def" ID "(" <params>
<params>        -> <params'>
                | ID <params'>
<params'>       -> ")" "begin" <stmt_list> <return> "end"
                | "," ID <params'>
<str-expr>      -> <nonint-value> <str-operation>
                | <function-call> <str-operation>
<int-expr>      -> <n_expr> <logic> //Note function call is included
        <n_expr>-> <term> <first-degree>
        <term>  -> <factor> <second-degree>
<str-operation> -> ∈
                | ADD <str-operation'> <str-operation>
<str-operation'>-> <nonint-value>
                | <function-call>
<compare>       -> ∈
                | ">" <int-value>
                | ">=" <int-value>
                | "<" <int-value>
                | "<=" <int-value>
                | "==" <int-value>
                | "!=" <int-value>
        <logic> -> ∈
                | AND <n_expr> // "and"
                | OR <n_expr> // "or"
<first-degree>  -> ∈
                | ADD // INTs will be tokenized as only positive ints
                | SUBTRACT // interpreter may have to differentiate "-" ID and Subtract
<second-degree> -> ∈
                | MULTIPLY //"*"
                | DIVIDE // "/"
                | MODULO // "%"
        <factor>-> <int-value> <compare>
                | <function-call> <compare>
<ID-operation>  -> ∈
                | "[" <int-value> "]"
```

```
        <value> -> STRING
                | INT
                | "str" "(" <value> ")" // casting
                | "int" "(" <value> ")" // casting
                | ID <ID-operation>
                | <function-call>
<nonint-value>  -> STRING
                | "str" "(" <value> ")" // casting
                | ID <ID-operation>
    <int-value> -> INT
                | "int" "(" <value> ")" // casting
                | ID <ID-operation>
                | "-" <int-value>

        <return>-> ∈
                | "return" <return-option>
<return-option> -> <value>
                | <int-expr>
```