



Tecnológico de Monterrey

Syntax Analysis Project

Estefania Jimenez Garcia A01635062

15 de Junio del 2023

Introduction

In report highlights my project for a syntax analyzer(parser), the report has four segments in which I will make a extend analysis of the all the context free grammar pre-processing I did to the grammar and explain each modification that I made, the design of the algorithm of my parser, the implementation of the parser and finally a section that will include the testing that was used to ensure that the parser obtained the desired results based on the metrics provided for the project.

My parser uses the tokens given by my lexer(scanner) and applies the grammar rules that I modified to check if the code follows the correct syntax. For my parser I decided to go for a top down recursive descent parser, the reason for this is that I decided after doing the context free grammar pre processing, the first sets, the follow sets and the first+ sets and taking into account the time I had left to deliver my project and that would benefit my processed syntax conventions.

When considering a programming language for the implementation of my parser, I chose Python due to its attributes such as readability, simplicity, and versatility. The clear syntax presented by Python allows one to program quickly, safeguarding stability long-term as well.

Analysis

Based on the top down recipe I decided that the first step was doing: a context free grammar pre-processing, this process is needed since the grammar upon a predictive parser is design has to be: unambiguous, free of left recursion and free of left factor.

First I was dedicated to solving the ambiguity of the grammar. In this step I review the productions of the CFG that deal with arithmetic, logic and comparison operators with the goal of using the precedence and association rules are applied.

In this review, I noted the rules that could be ambiguous. I did this by making an input that could fit the grammar, and implementing a leftmost derivation and a right most derivation. In the leftmost derivation, at every step the leftmost non-terminal is replaced by one of its productions while in the right most derivation the rightmost non-terminal is the one being replaced by one of its productions. If the grammar produces two different parse trees, it means the grammar is ambiguous meaning I have to choose between the LMD or the RMD in order to fix this.

Here are some example of the process of a grammar that was unambiguous, and in which I didn't had to modified them:

1. $program \rightarrow declaration_list$
2. $declaration_list \rightarrow declaration_list\ declaration \mid declaration$
3. $declaration \rightarrow var_declaration \mid fun_declaration$

input: `int a; void function() {}`

LMD:

1. $program \rightarrow declaration_list$
2. $declaration_list \rightarrow declaration_list\ declaration$
3. $declaration_list \rightarrow declaration\ declaration$
4. $declaration_list \rightarrow var_declaration\ declaration$
5. $declaration_list \rightarrow int\ a;\ declaration$
6. $declaration_list \rightarrow int\ a;\ fun_declaration$
7. $declaration_list \rightarrow int\ a;\ void\ function()\ \{\}$

RMD:

1. $program \rightarrow declaration_list$
2. $declaration_list \rightarrow declaration_list\ declaration$
3. $declaration_list \rightarrow declaration_list\ fun_declaration$
4. $declaration_list \rightarrow declaration_list\ void\ function()\ \{\}$
5. $declaration_list \rightarrow declaration_list\ declaration$
6. $declaration_list \rightarrow var_declaration\ void\ function()\ \{\}$
7. $declaration_list \rightarrow int\ a;\ void\ function()\ \{\}$

Since it generated the string, we conclude that **the syntax conventions 1, 2, and 3 are unambiguous.**

4. $\text{var_declaration} \rightarrow \text{type_specifier ID} ; | \text{type_specifier ID [NUMBER]} ;$

5. $\text{type_specifier} \rightarrow \text{int} | \text{float} | \text{string} | \text{void}$

input: int x[10]

LMD:

1. $\text{var_declaration} \rightarrow \text{type_specifier ID [NUMBER]} ;$
2. $\text{var_declaration} \rightarrow \text{int ID [NUMBER]} ;$
3. $\text{var_declaration} \rightarrow \text{int x [NUMBER]} ;$
4. $\text{var_declaration} \rightarrow \text{int x [10]} ;$
5. $\text{var_declaration} \rightarrow \text{int x[10]} ;$

RMD:

1. $\text{var_declaration} \rightarrow \text{type_specifier ID [NUMBER]} ;$
2. $\text{var_declaration} \rightarrow \text{type_specifier ID [10]} ;$
3. $\text{var_declaration} \rightarrow \text{type_specifier x [10]} ;$
4. $\text{var_declaration} \rightarrow \text{int x [10]} ;$
5. $\text{var_declaration} \rightarrow \text{int x[10]} ;$

Since it generated the string, we conclude that **the syntax conventions 4 and 5 are unambiguous.**

6. $\text{fun_declaration} \rightarrow \text{type_specifier ID (params) compound_stmt}$

7. $\text{params} \rightarrow \text{param_list} | \text{void}$

8. $\text{param_list} \rightarrow \text{param_list , param} | \text{param}$

9. $\text{param} \rightarrow \text{type_specifier ID} | \text{type_specifier ID []}$

input: int function(int a, float b) {}

LMD:

1. $\text{fun_declaration} \rightarrow \text{type_specifier ID (params) compound_stmt}$
2. $\text{fun_declaration} \rightarrow \text{int ID (params) compound_stmt}$
3. $\text{fun_declaration} \rightarrow \text{int function (params) compound_stmt}$
4. $\text{fun_declaration} \rightarrow \text{int function (param_list) compound_stmt}$
5. $\text{fun_declaration} \rightarrow \text{int function (param_list , param) compound_stmt}$
6. $\text{fun_declaration} \rightarrow \text{int function (param , param) compound_stmt}$
7. $\text{fun_declaration} \rightarrow \text{int function (type_specifier ID , param) compound_stmt}$
8. $\text{fun_declaration} \rightarrow \text{int function (int ID , param) compound_stmt}$
9. $\text{fun_declaration} \rightarrow \text{int function (int a , param) compound_stmt}$
10. $\text{fun_declaration} \rightarrow \text{int function (int a , type_specifier ID) compound_stmt}$
11. $\text{fun_declaration} \rightarrow \text{int function (int a , float ID) compound_stmt}$
12. $\text{fun_declaration} \rightarrow \text{int function (int a , float b) compound_stmt}$
13. $\text{fun_declaration} \rightarrow \text{int function (int a , float b) \{\}}$

RMD:

1. $\text{fun_declaration} \rightarrow \text{type_specifier ID (params) compound_stmt}$
2. $\text{fun_declaration} \rightarrow \text{type_specifier ID (params) \{\}}$
3. $\text{fun_declaration} \rightarrow \text{type_specifier ID (param_list) \{\}}$
4. $\text{fun_declaration} \rightarrow \text{type_specifier ID (param_list , param) \{\}}$
5. $\text{fun_declaration} \rightarrow \text{type_specifier ID (param , param) \{\}}$
6. $\text{fun_declaration} \rightarrow \text{type_specifier ID (param , type_specifier ID) \{\}}$
7. $\text{fun_declaration} \rightarrow \text{type_specifier ID (param , float ID) \{\}}$
8. $\text{fun_declaration} \rightarrow \text{type_specifier ID (param , float b) \{\}}$
9. $\text{fun_declaration} \rightarrow \text{type_specifier ID (type_specifier ID , float b) \{\}}$
10. $\text{fun_declaration} \rightarrow \text{type_specifier ID (int ID , float b) \{\}}$
11. $\text{fun_declaration} \rightarrow \text{type_specifier ID (int a , float b) \{\}}$
12. $\text{fun_declaration} \rightarrow \text{int ID (int a , float b) \{\}}$
13. $\text{fun_declaration} \rightarrow \text{int function (int a , float b) \{\}}$

Since it generated the string, we conclude that **the syntax conventions 6, 7, 8 and 9 are unambiguous.**

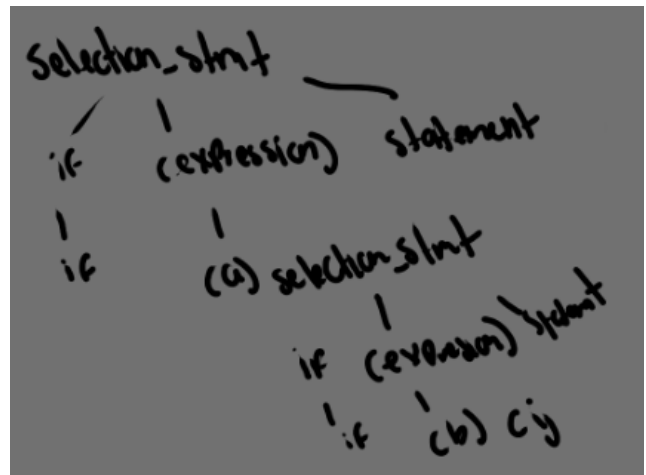
And there is the process of the grammars that were ambiguous and I decided to modified:

16. $\text{selection_stmt} \rightarrow \text{if (expression) statement} \mid \text{if (expression) statement else statement}$

input: if (a) if (b) c; else d;

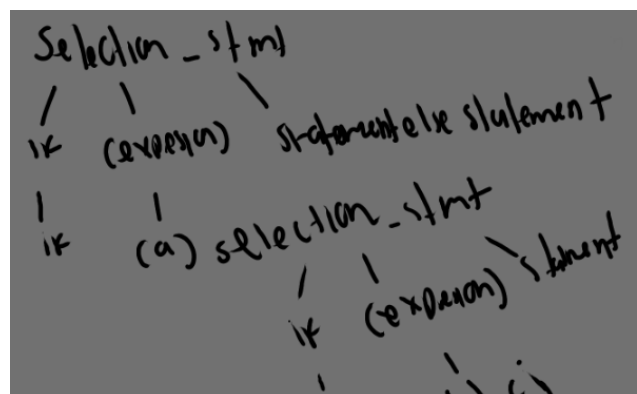
LMD:

1. $\text{selection_stmt} \rightarrow \text{if (expression) statement else statement}$
2. $\text{selection_stmt} \rightarrow \text{if (a) statement else statement}$
3. $\text{selection_stmt} \rightarrow \text{if (a) if (expression) statement else statement}$
4. $\text{selection_stmt} \rightarrow \text{if (a) if (b) statement else statement}$
5. $\text{selection_stmt} \rightarrow \text{if (a) if (b) c ; else statement}$
6. $\text{selection_stmt} \rightarrow \text{if (a) if (b) c ; else d ;}$



RMD:

1. $\text{selection_stmt} \rightarrow \text{if (expression) statement}$
2. $\text{selection_stmt} \rightarrow \text{if (a) statement}$
3. $\text{selection_stmt} \rightarrow \text{if (a) selection_stmt}$



4. $\text{selection_stmt} \rightarrow \text{if} (a) \text{if} (\text{expression}) \text{statement else statement}$
5. $\text{selection_stmt} \rightarrow \text{if} (a) \text{if} (b) \text{statement else statement}$
6. $\text{selection_stmt} \rightarrow \text{if} (a) \text{if} (b) c ; \text{else statement}$
7. $\text{selection_stmt} \rightarrow \text{if} (a) \text{if} (b) c ; \text{else } d ;$

Since we have different parse trees, we conclude **the syntax convection 16 is ambiguous**

In order to solve the ambiguity problem, we rewrite the syntax convention to using the dangling else problem, I divided the selection statements into two types: one for the **matched statements** and one for the **unmatched statements**:

1. $\text{matched_stmt} \rightarrow \text{if} (\text{expression}) \text{matched_stmt else matched_stmt} \mid \text{other_stmt}$
2. $\text{unmatched_stmt} \rightarrow \text{if} (\text{expression}) \text{statement} \mid \text{if} (\text{expression}) \text{matched_stmt else unmatched_stmt}$
3. $\text{statement} \rightarrow \text{matched_stmt} \mid \text{unmatched_stmt}$

In these new syntax conventions, **matched_stmt** represents an if statement where every if has a corresponding else, **unmatched_stmt** represents an if statement where there might be an if without a corresponding else and finally **other_stmt** represents all the other types of statements in the language. With this modification my parser would be able to match an else statement with the closest if, only if the if statement does not already have a matched else statement.

22. $\text{expression} \rightarrow \text{arithmetic_expression relop arithmetic_expression} \mid \text{arithmetic_expression}$
23. $\text{relop} \rightarrow <= \mid < \mid > \mid >= \mid == \mid !=$
24. $\text{arithmetic_expression} \rightarrow \text{arithmetic_expression addop term} \mid \text{term}$
25. $\text{addop} \rightarrow + \mid -$
26. $\text{term} \rightarrow \text{term mulop factor} \mid \text{factor}$
27. $\text{mulop} \rightarrow * \mid /$
28. $\text{factor} \rightarrow (\text{arithmetic_expression}) \mid \text{var} \mid \text{call} \mid \text{NUMBER}$

input: $a + b * c$

LMD:

- $\text{arithmetic_expression} \rightarrow \text{arithmetic_expression addop term}$
- $\text{arithmetic_expression} \rightarrow \text{term addop term}$
- $\text{arithmetic_expression} \rightarrow \text{factor addop term}$
- $\text{arithmetic_expression} \rightarrow \text{var addop term}$
- $\text{arithmetic_expression} \rightarrow a \text{ addop term}$
- $\text{arithmetic_expression} \rightarrow a + \text{term}$
- $\text{arithmetic_expression} \rightarrow a + \text{term mulop factor}$
- $\text{arithmetic_expression} \rightarrow a + \text{factor mulop factor}$
- $\text{arithmetic_expression} \rightarrow a + \text{var mulop factor}$
- $\text{arithmetic_expression} \rightarrow a + b \text{ mulop factor}$
- $\text{arithmetic_expression} \rightarrow a + b * \text{factor}$
- $\text{arithmetic_expression} \rightarrow a + b * \text{var}$
- $\text{arithmetic_expression} \rightarrow a + b * c$

RMD:

arithmetic_expression \rightarrow arithmetic_expression addop term
arithmetic_expression \rightarrow arithmetic_expression addop term mulop factor
arithmetic_expression \rightarrow arithmetic_expression addop factor mulop factor
arithmetic_expression \rightarrow arithmetic_expression addop var mulop factor
arithmetic_expression \rightarrow arithmetic_expression addop b mulop factor
arithmetic_expression \rightarrow arithmetic_expression addop b * factor
arithmetic_expression \rightarrow arithmetic_expression addop b * var
arithmetic_expression \rightarrow arithmetic_expression addop b * c
arithmetic_expression \rightarrow term addop b * c
arithmetic_expression \rightarrow factor addop b * c
arithmetic_expression \rightarrow var addop b * c
arithmetic_expression \rightarrow a addop b * c
arithmetic_expression \rightarrow a + b * c

Since we have different parse trees, we conclude **the syntax conventions 22, 23, 24, 25, 26, 27 and 28 are ambiguous**

The ambiguity can be solved by precedence and associativity; the precedence of the operator is encoded by the level of the grammar rule. The operator **mulop** has a higher precedence than **addop**, and these two have a higher precedence than **relop**. For the associativity, it is encoded by the recursion in the grammar rules, the recursion on the left makes the operator left associative while the recursion on the right makes them right associative. For this I decided that all operators would be left associative, since it is the standard.

Precedence:

Multiplication and division have a higher precedence than addition and subtraction
So multiplication and division should be performed before addition and subtraction

Associativity:

Both addition and subtraction should be associative

For addition: $(a+b) + c == a + (b+c)$

For multiplication: $(a*b) * c == a * (b*c)$

Using these two, we rewrite them to:

expression \rightarrow simple_expression relop simple_expression | simple_expression
relop \rightarrow <= | < | > | >= | == | !=
simple_expression \rightarrow simple_expression addop term | term
addop \rightarrow + | -
term \rightarrow term mulop factor | factor
mulop \rightarrow * | /
factor \rightarrow (expression) | var | call | NUMBER

These new syntax conventions will be able to be used in the parser to parse expression based on the standar rules of operator precedence and associativity.

After solving the ambiguity of the grammar, I reviewed the CFG to comply correctly with the syntax and semantics specifications, and I decided to add some restrictions as syntax conventions that didn't seem to be already implemented in the grammar.

I decided to add into the first syntax conventions **main_declaration**, and then added it as a new non-terminal with this new syntax conventions:

1. $program \rightarrow declaration_list\ main_declaration$
2. $main_declaration \rightarrow void\ ID\ (void)\ compound_stmt$

With these modifications, the parser ensures the main function at the end of the program and now compiles with the restriction that states “*The last declaration in a program MUST be a function declaration of the form void main(void)*”.

After doing this modification, I checked if the new syntax conventions are unambiguous in the same way I did before:

Input: `int a; void main(void) {}`

LDM:

1. $program \rightarrow declaration_list\ main_declaration$
2. $program \rightarrow declaration\ main_declaration$
3. $program \rightarrow type_specifier\ ID\ ;\ main_declaration$
4. $program \rightarrow int\ ID\ ;\ main_declaration$
5. $program \rightarrow int\ a\ ;\ main_declaration$
6. $program \rightarrow int\ a\ ;\ void\ ID\ (void)\ compound_stmt$
7. $program \rightarrow int\ a\ ;\ void\ main\ (void)\ compound_stmt$
8. $program \rightarrow int\ a\ ;\ void\ main\ (void)\ {}$

RDM:

1. $program \rightarrow declaration_list\ main_declaration$
2. $program \rightarrow declaration_list\ void\ ID\ (void)\ compound_stmt$
3. $program \rightarrow declaration_list\ void\ main\ (void)\ compound_stmt$
4. $program \rightarrow declaration_list\ void\ main\ (void)\ {}$
5. $program \rightarrow declaration\ void\ main\ (void)\ {}$
6. $program \rightarrow type_specifier\ ID\ ;\ void\ main\ (void)\ {}$
7. $program \rightarrow int\ ID\ ;\ void\ main\ (void)\ {}$
8. $program \rightarrow int\ a\ ;\ void\ main\ (void)\ {}$

Since they both produce the same string, **they are unambiguous**.

After finishing fixing the ambiguous grammar, I check for grammars that could be left recursive, based on the rule that states the grammar is left recursive if it has a non-terminal A such that there is a derivation on $A \rightarrow +A\alpha$ for some string α .

The reason why I did this is because if a left recursive grammar is used in a top-down parser, the parser may go into an infinite loop. Here are the grammars that I modified in order to prevent this from happening in my parser:

9. $param_list \rightarrow param_list, param \mid param$

The non-terminal **param_list** appears as the first symbol on the right side of the production, which makes it left-recursive.

In order to eliminate the left recursive, I added the new non-terminal **param_list'** to handle the recursion and also added ϵ , with another occurrence of **param_list'**.

In the end I modify the synthetic convection to:

9. $param_list \rightarrow param_list'$

10. $param_list' \rightarrow \epsilon, param_list' \mid \epsilon$

18. $matched_stmt \rightarrow if(expression) matched_stmt else matched_stmt \mid other_stmt$

The non-terminal **matched_stmt** appears as the first symbol on the right side of the production, so it is left-recursive

In order to eliminate the left recursive I added the new non-terminal **matched_stmt'** to handle the recursion, making this new non-terminal represent any additional match statements.

In the end the conventions changed to:

18. $matched_stmt \rightarrow if(expression) matched_stmt else matched_stmt matched_stmt' \mid other_stmt$

19. $unmatched_stmt \rightarrow if(expression) statement \mid if(expression) matched_stmt else unmatched_stmt$

The non-terminal **unmatched_stmt** appears as the first symbol on the right side of the production, so it is left-recursive.

In order to eliminate the left recursive I added the new non-terminal **unmatched_stmt'** to handle the recursion, making this new non-terminal represent any additional unmatched statements that follow the if statement

In the end the conventions changed to:

19. $unmatched_stmt \rightarrow if(expression) statement unmatched_stmt' \mid if(expression) matched_stmt else unmatched_stmt unmatched_stmt'$

25. $var \rightarrow ID \mid ID [arithmetic_expression]$

The non-terminal **var** appears as the first symbol on the right side of the second alternative, so it is left-recursive. In order to eliminate the left recursive I added the new non-terminal **var'** to handle the optional part.

In the end the conventions changed to:

25. $var \rightarrow ID\ var'$

26. $var' \rightarrow \varepsilon \mid [arithmetic_expression]$

29. $simple_expression \rightarrow simple_expression\ addop\ term \mid term$

The non-terminal **simple_expression** appears as the first symbol on the right side of the production, so it is left-recursive.

In order to eliminate the left recursive I added the new non-terminal **simple_expression'** to handle the recursion, making this new non-terminal represent any additional terms and adobe expression.

In the end the convention changed to:

29. $simple_expression \rightarrow term\ simple_expression'$

30. $simple_expression' \rightarrow \varepsilon \mid addop\ term\ simple_expression'$

32. $term \rightarrow term\ mulop\ factor \mid factor$

The non-terminal **term** appears as the first symbol on the right side of the first alternative, so it is left-recursive.

In order to eliminate the left recursive I added the new non-terminal **term'** to handle the recursion, making this new non-terminal represent any additional factors and mulop expression.

In the end the convention changed to:

32. $term \rightarrow factor\ term'$

33. $term' \rightarrow \varepsilon \mid mulop\ factor\ term$

38. $arg_list \rightarrow arg_list, arithmetic_expression \mid arithmetic_expression$

The non-terminal **arg_list** appears as the first symbol on the right side of the first alternative, so it is left-recursive.

In order to eliminate the left recursive I added the new non-terminal **arg_list'** to handle the recursion, making this new non-terminal represent any additional arithmetic expressions separated by commas.

In the end the convention changed to:

38. $arg_list \rightarrow arithmetic_expression\ arg_list'$

39. $arg_list' \rightarrow ,\ arithmetic_expression\ arg_list' \mid \varepsilon$

The next step for the context free grammar pre-processing is the left factoring technique, this technique is used to produce grammars that are suitable for a top-down parsing. This technique is used when a grammar has many common prefixes, which may lead to back tracking since while processing the common prefix it cannot be decided which production to expand.

After reviewing my syntax conventions, I decided that there was not any of them that would produce backtracking in my parser, and would only make the grammar more complex so I decided to not use this technique on my grammar.

Next and final step is to do simplification of grammars, which include doing the elimination of useless symbols, eliminating ϵ productions and the elimination of unit products of the form $A \rightarrow B$. This process is useful to optimize the grammar that may contain extra or unnecessary symbols, which will increase the length of the grammar. Here are the modifications that I made when I applied this techniques into the syntax conventions that needed it:

10. $param_list' \rightarrow , param param_list' \mid \epsilon$

Param_list' is nullable since it has an ϵ production at $param_list' \rightarrow \epsilon$

So we can modify it by replacing it with the other possible combination, so it can be empty using just param, so I modified by:

1. First I removed the ϵ -production from **param_list'**:
 $param_list' \rightarrow , param param_list'$
2. Then, for every production where **param_list'** appears on the right-hand side, we add a version of that production without **param_list'**:
 $param_list' \rightarrow , param param_list' \mid , param$

26. $var' \rightarrow \epsilon \mid [arithmetic_expression]$

Var' is nullable since it has an ϵ production at $var' \rightarrow \epsilon$

So I can modified by:

1. First I removed the ϵ -production from **var'**:
 $var' \rightarrow [arithmetic_expression]$
2. Then since **var'** appears on $var \rightarrow ID var'$, we add a new possible combination without **var'**:
 $var \rightarrow ID var' \mid ID$

30. $simple_expression' \rightarrow \epsilon \mid addop term simple_expression'$

Simple_expression' is nullable since it has an ϵ production at $simple_expression' \rightarrow \epsilon$

So I can modified by replacing it with the other possible combination:

1. First I remove the ϵ production from **simple_expression'**:
 $simple_expression' \rightarrow addop term simple_expression'$
2. Then for every production where **simple_expression'** appears on the right we add a version of the production without $simple_expression'$:
From:
 $simple_expression \rightarrow term simple_expression'$
To:
 $simple_expression \rightarrow term simple_expression' \mid term$
3. And then we do the same for $simple_expression' \rightarrow addop term simple_expression'$ to:
 $simple_expression' \rightarrow addop term simple_expression' \mid addop term$

33. $term' \rightarrow \epsilon \mid mulop factor term'$

Term' is nullable since it has an ϵ production at $\text{term}' \rightarrow \epsilon$
So I can modified by:

1. First I remove the ϵ production from **term'**:
 $\text{term}' \rightarrow \text{mulop factor term}'$
2. Then for every production where the **term'** appears on the right-hand side, we add a version of that production without **term'**. Since **term'** appears on $\text{term} \rightarrow \text{factor term}'$, we add a new possible combination without **term'**:
 $\text{term} \rightarrow \text{factor term}' \mid \text{factor}$
3. And finally I also added a new possible combination on $\text{simple_expression}' \rightarrow \text{addop term simple_expression}'$ | addop term :
 $\text{simple_expression}' \rightarrow \text{addop term simple_expression}' \mid \text{addop term} \mid \text{addop factor simple_expression}' \mid \text{addop factor}$

39. $\text{arg_list}' \rightarrow , \text{arithmetic_expression arg_list}' \mid \epsilon$

Arg_list' is nullable since it has an ϵ production at $\text{arg_list}' \rightarrow \epsilon$
So I can modified by:

1. First I remove the ϵ production from **arg_list'**:
 $\text{arg_list}' \rightarrow , \text{arithmetic_expression arg_list}'$
2. Then for every production where **arg_list'** appears on the right-hand side, we add a version of that production without **term'**. Since **arg_list'** appears on $\text{arg_list} \rightarrow \text{arithmetic_expression arg_list}'$, we add a new possible combination without **arg_list'**:
 $\text{arg_list} \rightarrow \text{arithmetic_expression arg_list}' \mid \text{arithmetic_expression}$
3. And then I add the new combinations to $\text{arg_list}' \rightarrow , \text{arithmetic_expression arg_list}'$ and it becomes:
 $\text{arg_list}' \rightarrow , \text{arithmetic_expression arg_list}' \mid , \text{arithmetic_expression}$

With these process we finish doing the context free grammar pre-processing, which resulted in a grammar that is unambiguous, free of left recursion and free of left factor:

1. $\text{program} \rightarrow \text{declaration_list main_declaration}$
2. $\text{main_declaration} \rightarrow \text{void ID (void) compound_stmt}$
3. $\text{declaration_list} \rightarrow \text{declaration_list declaration} \mid \text{declaration}$
4. $\text{declaration} \rightarrow \text{var_declaration} \mid \text{fun_declaration}$
5. $\text{var_declaration} \rightarrow \text{type_specifier ID ;} \mid \text{type_specifier ID [NUMBER] ;}$
6. $\text{type_specifier} \rightarrow \text{int} \mid \text{float} \mid \text{string} \mid \text{void}$
7. $\text{fun_declaration} \rightarrow \text{type_specifier ID (params) compound_stmt}$
8. $\text{params} \rightarrow \text{param_list} \mid \text{void}$
9. $\text{param_list} \rightarrow \text{param param_list'}$
10. $\text{param_list'} \rightarrow , \text{param param_list'} \mid , \text{param}$
11. $\text{param} \rightarrow \text{type_specifier ID} \mid \text{type_specifier ID []}$
12. $\text{compound_stmt} \rightarrow \{ \text{local_declarations statement_list} \}$
13. $\text{local_declarations} \rightarrow \text{local_declarations var_declaration} \mid \epsilon$
14. $\text{statement_list} \rightarrow \text{statement_list statement} \mid \epsilon$
15. $\text{statement} \rightarrow \text{assignment_stmt} \mid \text{call} \mid \text{compound_stmt} \mid \text{selection_stmt} \mid \text{iteration_stmt} \mid \text{return_stmt} \mid \text{input_stmt} \mid \text{output_stmt}$
16. $\text{assignment_stmt} \rightarrow \text{var} = \text{expression} ; \mid \text{STRING} ;$
17. $\text{call_stmt} \rightarrow \text{call} ;$
18. $\text{matched_stmt} \rightarrow \text{if (expression) matched_stmt else matched_stmt matched_stmt'}$
 other_stmt
19. $\text{unmatched_stmt} \rightarrow \text{if (expression) statement unmatched_stmt'}$
 $\text{matched_stmt else unmatched_stmt unmatched_stmt'}$
20. $\text{statement} \rightarrow \text{matched_stmt} \mid \text{unmatched_stmt}$
21. $\text{iteration_stmt} \rightarrow \text{while (expression) statement}$
22. $\text{return_stmt} \rightarrow \text{return} ; \mid \text{return expression} ;$
23. $\text{input_stmt} \rightarrow \text{input var} ;$
24. $\text{output_stmt} \rightarrow \text{output expression} ;$
25. $\text{var} \rightarrow \text{ID var'} \mid \text{ID}$
26. $\text{var'} \rightarrow [\text{arithmetic_expression}]$
27. $\text{expression} \rightarrow \text{simple_expression relop simple_expression} \mid \text{simple_expression}$
28. $\text{relop} \rightarrow \leq \mid < \mid > \mid \geq \mid == \mid !=$
29. $\text{simple_expression} \rightarrow \text{term simple_expression'}$
 $\mid \text{term}$
30. $\text{simple_expression'}$
 $\rightarrow \text{addop term simple_expression'}$
 $\mid \text{addop term} \mid \text{addop factor}$
 $\text{simple_expression'}$
 $\mid \text{addop factor}$
31. $\text{addop} \rightarrow + \mid -$
32. $\text{term} \rightarrow \text{factor term'}$
 $\mid \text{factor}$
33. $\text{term'} \rightarrow \text{mulop factor term'}$
34. $\text{mulop} \rightarrow * \mid /$
35. $\text{factor} \rightarrow (\text{expression}) \mid \text{var} \mid \text{call} \mid \text{NUMBER}$
36. $\text{call} \rightarrow \text{ID (args)}$
37. $\text{args} \rightarrow \text{arg_list} \mid \epsilon$
38. $\text{arg_list} \rightarrow \text{arithmetic_expression arg_list'}$
 $\mid \text{arithmetic_expression}$
39. $\text{arg_list'} \rightarrow , \text{arithmetic_expression arg_list'}$
 $\mid , \text{arithmetic_expression}$

Upon reviewing my grammar at my oral exam for this project with my professor, it became apparent that I needed to solve some errors in order for my parser work correctly. First I shouldn't have solve the ambiguity problem of this syntax convention:

$$\text{selection_stmt} \rightarrow \text{if}(\text{expression}) \text{statement} \mid \text{if}(\text{expression}) \text{statement} \text{else statement}$$

By dividing the selection statements into two types, and modifying the syntax convention to:

$$\begin{aligned} \text{matched_stmt} &\rightarrow \text{if}(\text{expression}) \text{matched_stmt} \text{else matched_stmt matched_stmt}' \mid \\ &\text{other_stmt} \\ \text{unmatched_stmt} &\rightarrow \text{if}(\text{expression}) \text{statement unmatched_stmt}' \mid \text{if}(\text{expression}) \\ &\text{matched_stmt} \text{else unmatched_stmt unmatched_stmt}' \end{aligned}$$

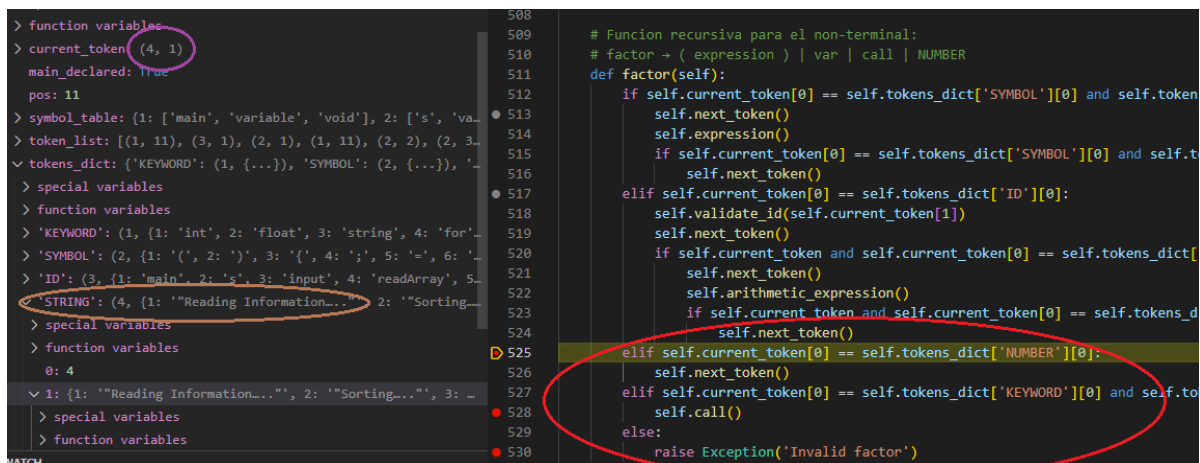
This process that I made introduced additional complexity into my grammar, prompting my parser to return this error:

Parsing failed: Invalid factor

When in fact there was not any factor in the code, which indicates there is an error in the way the statements are being parsed. Yet another parsing error was identified concerning the syntax convention of the non-terminal factor:

```
32. term → factor term' | factor
33. term' → mulop factor term'
34. mulop → * | /
35. factor → ( expression ) var | call | NUMBER
36. call → ID ( args )
37. args → arg_list | ε
38. arg_list → arithmetic_expression arg_list' | arithmetic_expression
```

Following my meeting with the professor, I decided to debug the code, which revealed that the error occur when the parser attempted to parse asignment_statemnt the of the variable S



```
> function variables
> current_token (4, 1)
  main_declared: true
  pos: 11
> symbol_table: {1: ['main', 'variable', 'void'], 2: ['s', 'va...
> token_list: [(1, 11), (3, 1), (2, 1), (1, 11), (2, 2), (2, 3...
> tokens_dict: {'KEYWORD': (1, {...}), 'SYMBOL': (2, {...}), '-
> special variables
> function variables
> 'KEYWORD': (1, {1: 'int', 2: 'float', 3: 'string', 4: 'for'...
> 'SYMBOL': (2, {1: '(', 2: ')', 3: '{', 4: '}', 5: '=', 6: '_...
> 'ID': (3, {1: 'main', 2: 's', 3: 'input', 4: 'readArray', 5...
> 'STRING': (4, {1: "Reading Information...", 2: "Sorting...
> special variables
> function variables
0: 4
1: {1: "Reading Information...", 2: "Sorting...", 3: ...
> special variables
> function variables
```

```
508
509
510 # Funcion recursiva para el non-terminal:
511 # factor → ( expression ) | var | call | NUMBER
512 def factor(self):
513     if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.token
514         self.next_token()
515         self.expression()
516     elif self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.t
517         self.next_token()
518     elif self.current_token[0] == self.tokens_dict['ID'][0]:
519         self.validate_id(self.current_token[1])
520         self.next_token()
521     elif self.current_token and self.current_token[0] == self.tokens_dict[
522         self.next_token()
523         self.arithmetic_expression()
524     elif self.current_token and self.current_token[0] == self.tokens_d
525         self.next_token()
526     elif self.current_token[0] == self.tokens_dict['NUMBER'][0]:
527         self.next_token()
528     elif self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.to
529         self.call()
530     else:
531         raise Exception('Invalid factor')
```

As you can see on the debugger, the purple circle points out the current token that is being parsed, the brown circle points out the tokens directory and the red circle points out the current loop set to trigger an error in the factor function.

```
> function variables
> current_token (4, 1)
  main_declared: true
  pos: 11
> symbol_table: {1: ['main', 'variable', 'void'], 2: ['s', 'va...
> token_list: [(1, 11), (3, 1), (2, 1), (1, 11), (2, 2), (2, 3...
✓ tokens_dict: {'KEYWORD': (1, {...}), 'SYMBOL': (2, {...}), '...
  > special variables
  > function variables
  > 'KEYWORD': (1, {1: 'int', 2: 'float', 3: 'string', 4: 'for'...
  > 'SYMBOL': (2, {1: '(', 2: ')', 3: '{', 4: ';', 5: '=', 6: '...
  > 'ID': (3, {1: 'main', 2: 's', 3: 'input', 4: 'readArray', 5...
  ✓ 'STRING': (4, {1: "Reading Information....", 2: "Sorting...."
  > special variables
  > function variables
    0: 4
  ✓ 1: {1: "Reading Information....", 2: "Sorting....", 3: ...
  > special variables
  > function variables
```

As it is being shown on the image, the current token is situated at (4,1) which is referencing the position within the tokens dictionary which we can use to determine the exact token that it is being parsed. Based on the token_dict, we can verify the parser is processing a token which type is STRING(4) with the value(1) "Reading Information....".

```
def test_parser8(self):
    testCode = '''
    void main(void){
        string s;
        s = "Reading Information....";
        input(s);
        readArray();
        s = "Sorting....";
        input(s);
        sort(f2,0,10);
        s = "Sorted Array:";
        output(s);
        writeArray();
        return;
    }/* END of main() */
    ...
```

With this new information, I went back to the code test that was being parsed and confirmed that the string bearing the value was being assigned to variable S.

```

508
509 # Funcion recursiva para el non-terminal:
510 # factor -> ( expression ) | var | call | NUMBER
511 def factor(self):
512     if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.token
513         self.next_token()
514         self.expression()
515     if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.t
516         self.next_token()
517     elif self.current_token[0] == self.tokens_dict['ID'][0]:
518         self.validate_id(self.current_token[1])
519         self.next_token()
520     if self.current_token and self.current_token[0] == self.tokens_dict[
521         self.next_token()
522         self.arithmetic_expression()
523         if self.current_token and self.current_token[0] == self.tokens_d
524             self.next_token()
525     elif self.current_token[0] == self.tokens_dict['NUMBER'][0]:
526         self.next_token()
527     elif self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.to
528         self.call()
529     else:
530         raise Exception('Invalid factor')

```

I can hence deduce that the error stemmed from the fact that my grammar did not allow the `assignment_stmt` to be `var = STRING`; . Consequently, when my parser processed `var =`, it directed it to `asignment_statement`, which then it passed it to `expression`.

Subsequently, an unsuccessful attempt to parse it under `expression` directed it to `factor` hence triggering the invalid factor error. Therefore, a potential solution to this would be to modify the syntax convention of:

$$\text{assignment_stmt} \rightarrow \text{var} = \text{expression} ; \mid \text{STRING} ;$$

To:

$$\text{assignment_stmt} \rightarrow \text{var} = \text{expression} ; \mid \text{var} = \text{STRING} ;$$

Which would allow a `assignment_stmt` to be parsed as a variable that is assigned a string. Additionally, we realize that I had inadvertently overlooked the replacement of `output_stm` and `input_stm` with their correct keywords, `write` and `read`. As a result, the parser incorrectly identified them as IDs, failing to parse them accurately as they appear on the code.

After the pre-processing of the context-free grammar, I initially decided to implement a Non-recursive descent (LL(1) parser). However, when I created the sets needed to implement the parsing table I transitioned to a Recursive descent parser. I will explain the reasons behind this change after I explained the process I did to create the sets.

The initial phase of implementing the LL(1) parser requires to formulate the First sets, Follow sets and First+ Sets, to later on use them to create the parsing table. First I formulated the First set for each terminal and non-terminal within my grammar. These sets are the set of terminal symbols and possible ϵ production that can appear as the first terminal in some string derived from the non-terminals and terminals.

The following are the First sets I formulated:

1. $\text{FIRST}(\text{program}) = \text{FIRST}(\text{declaration_list}) = \{\text{type_specifier}\}$
2. $\text{FIRST}(\text{main_declaration}) = \{\text{void}\}$
3. $\text{FIRST}(\text{declaration}) = \text{FIRST}(\text{var_declaration}) = \text{FIRST}(\text{fun_declaration}) = \{\text{type_specifier}\}$
4. $\text{FIRST}(\text{var_declaration}) = \{\text{type_specifier}\}$
5. $\text{FIRST}(\text{type_specifier}) = \{\text{int, float, string, void}\}$
6. $\text{FIRST}(\text{fun_declaration}) = \{\text{type_specifier}\}$
7. $\text{FIRST}(\text{params}) = \text{FIRST}(\text{param_list}) \cup \{\text{void}\} = \{\text{type_specifier, void}\}$
8. $\text{FIRST}(\text{param_list}) = \text{FIRST}(\text{param}) = \{\text{type_specifier}\}$
9. $\text{FIRST}(\text{param_list}') = \{\epsilon\}$
10. $\text{FIRST}(\text{param}) = \{\text{type_specifier}\}$
11. $\text{FIRST}(\text{compound_stmt}) = \{\epsilon\}$
12. $\text{FIRST}(\text{local_declarations}) = \text{FIRST}(\text{var_declaration}) \cup \{\epsilon\} = \{\text{type_specifier, } \epsilon\}$
13. $\text{FIRST}(\text{statement_list}) = \text{FIRST}(\text{statement}) \cup \{\epsilon\} = \{\text{var, if, while, return, input, output, } \epsilon\}$
14. $\text{FIRST}(\text{statement}) = \{\text{var, if, while, return, input, output, } \epsilon\}$
15. $\text{FIRST}(\text{assignment_stmt}) = \{\text{var, STRING}\}$
16. $\text{FIRST}(\text{call_stmt}) = \{\text{call}\}$
17. $\text{FIRST}(\text{matched_stmt}) = \{\text{if, other_stmt}\}$
18. $\text{FIRST}(\text{unmatched_stmt}) = \{\text{if}\}$
19. $\text{FIRST}(\text{iteration_stmt}) = \{\text{while}\}$
20. $\text{FIRST}(\text{return_stmt}) = \{\text{return}\}$
21. $\text{FIRST}(\text{input_stmt}) = \{\text{input}\}$
22. $\text{FIRST}(\text{output_stmt}) = \{\text{output}\}$
23. $\text{FIRST}(\text{var}) = \{\text{ID}\}$
24. $\text{FIRST}(\text{var}') = \{\epsilon\}$
25. $\text{FIRST}(\text{expression}) = \text{FIRST}(\text{simple_expression}) = \{\text{term}\}$
26. $\text{FIRST}(\text{relop}) = \{\leq, <, >, \geq, ==, !=\}$
27. $\text{FIRST}(\text{simple_expression}') = \{\text{addop}\}$
28. $\text{FIRST}(\text{addop}) = \{+, -\}$
29. $\text{FIRST}(\text{term}) = \text{FIRST}(\text{factor}) = \{(\text{, var, call, NUMBER}\}$
30. $\text{FIRST}(\text{term}') = \{\text{mulop}\}$
31. $\text{FIRST}(\text{mulop}) = \{*, /\}$
32. $\text{FIRST}(\text{factor}) = \{(\text{, var, call, NUMBER}\}$

33. $\text{FIRST}(\text{call}) = \{\text{ID}\}$
34. $\text{FIRST}(\text{args}) = \text{FIRST}(\text{arg_list}) \cup \{\epsilon\} = \{\text{arithmetic_expression}, \epsilon\}$
35. $\text{FIRST}(\text{arg_list}) = \{\text{arithmetic_expression}\}$
36. $\text{FIRST}(\text{arg_list}') = \{,\}$

In an effort to make them more understandable, I represented them in the following table:

First()	
program	{type_specifier}
main_declaration	{void}
declaration	{type_specifier}
var_declaration	{type_specifier}
type_specifier	{int, float, string, void}
fun_declaration	{type_specifier}
params	{type_specifier, void}
param_list	{type_specifier}
param_list'	{,}
param	{type_specifier}
compound_stmt	{ }
local_declarations	{type_specifier, ϵ }
statement_list	{var, if, while, return, input, output, {, ϵ }
statement	{var, if, while, return, input, output, { }
assignment_stmt	{var, STRING}
call_stmt	{call}
matched_stmt	{if, other_stmt}
unmatched_stmt	{if}
iteration_stmt	{while}
return_stmt	{return}
input_stmt	{input}
output_stmt	{output}

var	{ID}
var'	{[}
expression	{term}
relop	{<=, <, >, >=, ==, !=}
simple_expression'	{addop}
addop	{+, -}
term	{(, var, call, NUMBER}
term'	{mulop}
mulop	{*, /}
factor	{(, var, call, NUMBER}
call	{ID}
args	{arithmetic_expression, ϵ }
arg_list	{arithmetic_expression}
arg_list'	{,}

After formulating the first set, it was necessary to formulate the Follow sets for each non-terminal within my grammar.

The following are the resulting Follow sets:

1. FOLLOW(program) = {\$}
2. FOLLOW(main_declaration) = {\$}
3. FOLLOW(declaration_list) = {void}
4. FOLLOW(declaration) = FIRST(declaration_list) U FOLLOW(declaration_list) = {type_specifier, void}
5. FOLLOW(var_declaration) = FIRST(local_declarations) U FOLLOW(local_declarations) = {type_specifier, void, ϵ }
6. FOLLOW(type_specifier) = {ID}
7. FOLLOW(fun_declaration) = FOLLOW(declaration) = {type_specifier, void}
8. FOLLOW(params) = {)}
9. FOLLOW(param_list) = FIRST(param_list') U {} = {,,)}
10. FOLLOW(param_list') = FOLLOW(param_list) = {,,)}
11. FOLLOW(param) = FIRST(param_list) U FOLLOW(param_list) = {,, type_specifier, void,)}

12. FOLLOW(compound_stmt) = FOLLOW(main_declaration) U FOLLOW(statement) = {if, var, while, return, input, output, {, void, \$}
13. FOLLOW(local_declarations) = {type_specifier, ε, ID, {, if, while, return, input, output}
14. FOLLOW(statement_list) = FOLLOW(compound_stmt) = {if, var, while, return, input, output, {, void, \$}
15. FOLLOW(statement) = FIRST(statement_list) U FOLLOW(statement_list) = {if, var, while, return, input, output, {, void, \$, ε}
16. FOLLOW(assignment_stmt) = FOLLOW(statement) = {if, var, while, return, input, output, {, void, \$, ε}
17. FOLLOW(call_stmt) = FOLLOW(statement) = {if, var, while, return, input, output, {, void, \$, ε}
18. FOLLOW(matched_stmt) = FOLLOW(statement) = {if, var, while, return, input, output, {, void, \$, ε}
19. FOLLOW(unmatched_stmt) = FOLLOW(statement) = {if, var, while, return, input, output, {, void, \$, ε}
20. FOLLOW(iteration_stmt) = FOLLOW(statement) = {if, var, while, return, input, output, {, void, \$, ε}
21. FOLLOW(return_stmt) = FOLLOW(statement) = {if, var, while, return, input, output, {, void, \$, ε}
22. FOLLOW(input_stmt) = FOLLOW(statement) = {if, var, while, return, input, output, {, void, \$, ε}
23. FOLLOW(output_stmt) = FOLLOW(statement) = {if, var, while, return, input, output, {, void, \$, ε}
24. FOLLOW(var) = {=,]}
25. FOLLOW(var') = FOLLOW(var) = {=,]}
26. FOLLOW(expression) = {;,)}
27. FOLLOW(relop) = FIRST(simple_expression) = {term}
28. FOLLOW(simple_expression') = FOLLOW(simple_expression) = {relop, ;,)}
29. FOLLOW(addop) = FIRST(term) = {factor}
30. FOLLOW(term) = FIRST(simple_expression') U FOLLOW(simple_expression') = {addop, relop, ;,)}
31. FOLLOW(term') = FOLLOW(term) = {addop, relop, ;,)}
32. FOLLOW(factor) = FIRST(term') U FOLLOW(term') = {mulop, addop, relop, ;,)}
33. FOLLOW(call) = FOLLOW(factor) = {mulop, addop, relop, ;,)}
34. FOLLOW(args) = {}
35. FOLLOW(arg_list) = FIRST(arg_list') U FOLLOW(args) = {,,)}
36. FOLLOW(arg_list') = FOLLOW(arg_list) = {,,)}

Which I also represented them as a table:

Follow()	
program	{ \$ }
main_declaration	{ \$ }

declaration_list	void
declaration	{type_specifier, void}
var_declaration	{type_specifier, void, ε}
type_specifier	{ID}
fun_declaration	{type_specifier, void}
params	{}
param_list	{, }
param_list'	{, }
param	{, type_specifier, void, }
compound_stmt	{if, var, while, return, input, output, {, void, \$}
local_declarations	{type_specifier, ε, ID, {, if, while, return, input, output}
statement_list	{if, var, while, return, input, output, {, void, \$}
statement	{if, var, while, return, input, output, {, void, \$, ε}
assignment_stmt	{if, var, while, return, input, output, {, void, \$, ε}
call_stmt	{if, var, while, return, input, output, {, void, \$, ε}
matched_stmt	{if, var, while, return, input, output, {, void, \$, ε}
unmatched_stmt	{if, var, while, return, input, output, {, void, \$, ε}
iteration_stmt	{if, var, while, return, input, output, {, void, \$, ε}
return_stmt	{if, var, while, return, input, output, {, void, \$, ε}
input_stmt	{if, var, while, return, input, output, {, void, \$, ε}
output_stmt	{if, var, while, return, input, output, {, void, \$, ε}
var	{=, }
var'	{=, }
expression	{, }
relop	{term}
simple_expression'	{relop, :, }
addop	{factor}
term	{addop, relop, :, }

term'	{addop, relop, :,)}
factor	{mulop, addop, relop, :,)}
call	{mulop, addop, relop, :,)}
args	{}
arg_list	{, ,)}
arg_list'	{, ,)}

Lastly, using the First(**x**) and Follow(**x**), I formulated the First+ set to handle the ϵ production of a non-terminal symbol **x** from my grammar.

The following are the resulting First+ sets I formulated:

1. FIRST+(program \rightarrow declaration_list main_declaration) = FIRST(declaration_list) = {type_specifier}
2. FIRST+(main_declaration \rightarrow void ID (void) compound_stmt) = {void}
3. FIRST+(declaration_list \rightarrow declaration_list declaration) = FIRST(declaration_list) = {type_specifier}
4. FIRST+(declaration_list \rightarrow declaration) = FIRST(declaration) = {type_specifier}
5. FIRST+(declaration \rightarrow var_declaration) = FIRST(var_declaration) = {type_specifier}
6. FIRST+(declaration \rightarrow fun_declaration) = FIRST(fun_declaration) = {type_specifier}
7. FIRST+(var_declaration \rightarrow type_specifier ID ;) = {type_specifier}
8. FIRST+(var_declaration \rightarrow type_specifier ID [NUMBER] ;) = {type_specifier}
9. FIRST+(type_specifier \rightarrow int | float | string | void) = {int, float, string, void}
10. FIRST+(fun_declaration \rightarrow type_specifier ID (params) compound_stmt) = {type_specifier}
11. FIRST+(params \rightarrow param_list) = FIRST(param_list) = {type_specifier}
12. FIRST+(params \rightarrow void) = {void}
13. FIRST+(param_list \rightarrow param param_list') = {type_specifier}
14. FIRST+(param_list' \rightarrow , param param_list') = {,}
15. FIRST+(param_list' \rightarrow , param) = {,}
16. FIRST+(param \rightarrow type_specifier ID) = {type_specifier}
17. FIRST+(param \rightarrow type_specifier ID []) = {type_specifier}
18. FIRST+(compound_stmt \rightarrow { local_declarations statement_list }) = { }
19. FIRST+(local_declarations \rightarrow local_declarations var_declaration) = {type_specifier, ϵ }
20. FIRST+(local_declarations $\rightarrow \epsilon$) = FOLLOW(local_declarations) = {ID, {, if, while, return, input, output}}
21. FIRST+(statement_list \rightarrow statement_list statement) = {var, if, while, return, input, output, {, ϵ }
22. FIRST+(statement_list $\rightarrow \epsilon$) = FOLLOW(statement_list) = {if, var, while, return, input, output, {, void, \$}

23. $\text{FIRST}+(\text{statement} \rightarrow \text{assignment_stmt} \mid \text{call} \mid \text{compound_stmt} \mid \text{selection_stmt} \mid \text{iteration_stmt} \mid \text{return_stmt} \mid \text{input_stmt} \mid \text{output_stmt}) = \{\text{var, if, while, return, input, output, \{\}}\}$
24. $\text{FIRST}+(\text{assignment_stmt} \rightarrow \text{var} = \text{expression} ;) = \text{FIRST}(\text{var}) = \{\text{ID}\}$
25. $\text{FIRST}+(\text{assignment_stmt} \rightarrow \text{STRING} ;) = \{\text{STRING}\}$
26. $\text{FIRST}+(\text{call_stmt} \rightarrow \text{call} ;) = \text{FIRST}(\text{call}) = \{\text{ID}\}$
27. $\text{FIRST}+(\text{matched_stmt} \rightarrow \text{if} (\text{expression}) \text{matched_stmt} \text{ else } \text{matched_stmt} \text{ matched_stmt}') = \{\text{if}\}$
28. $\text{FIRST}+(\text{unmatched_stmt} \rightarrow \text{if} (\text{expression}) \text{statement} \text{ unmatched_stmt}') = \{\text{if}\}$
29. $\text{FIRST}+(\text{iteration_stmt} \rightarrow \text{while} (\text{expression}) \text{statement}) = \{\text{while}\}$
30. $\text{FIRST}+(\text{return_stmt} \rightarrow \text{return} ;) = \{\text{return}\}$
31. $\text{FIRST}+(\text{return_stmt} \rightarrow \text{return expression} ;) = \{\text{return}\}$
32. $\text{FIRST}+(\text{input_stmt} \rightarrow \text{input var} ;) = \{\text{input}\}$
33. $\text{FIRST}+(\text{output_stmt} \rightarrow \text{output expression} ;) = \{\text{output}\}$
34. $\text{FIRST}+(\text{var} \rightarrow \text{ID var}') = \{\text{ID}\}$
35. $\text{FIRST}+(\text{var} \rightarrow \text{ID}) = \{\text{ID}\}$
36. $\text{FIRST}+(\text{var}' \rightarrow [\text{arithmetic_expression}]) = \{\}$
37. $\text{FIRST}+(\text{expression} \rightarrow \text{simple_expression} \text{ relop } \text{simple_expression}) = \text{FIRST}(\text{simple_expression}) = \{\text{term}\}$
38. $\text{FIRST}+(\text{expression} \rightarrow \text{simple_expression}) = \text{FIRST}(\text{simple_expression}) = \{\text{term}\}$
39. $\text{FIRST}+(\text{relop} \rightarrow <= \mid < \mid > \mid >= \mid == \mid !=) = \{<=, <, >, >=, ==, !=\}$
40. $\text{FIRST}+(\text{simple_expression} \rightarrow \text{term simple_expression}') = \text{FIRST}(\text{term}) = \{\text{factor}\}$
41. $\text{FIRST}+(\text{simple_expression}' \rightarrow \text{addop term simple_expression}') = \{\text{addop}\}$
42. $\text{FIRST}+(\text{simple_expression}' \rightarrow \text{addop term}) = \{\text{addop}\}$
43. $\text{FIRST}+(\text{addop} \rightarrow + \mid -) = \{+, -\}$
44. $\text{FIRST}+(\text{term} \rightarrow \text{factor term}') = \text{FIRST}(\text{factor}) = \{ (, \text{var}, \text{call}, \text{NUMBER} \}$
45. $\text{FIRST}+(\text{term}' \rightarrow \text{mulop factor term}') = \{\text{mulop}\}$
46. $\text{FIRST}+(\text{mulop} \rightarrow * \mid /) = \{*, /\}$
47. $\text{FIRST}+(\text{factor} \rightarrow (\text{expression})) = \{ (\}$
48. $\text{FIRST}+(\text{factor} \rightarrow \text{var}) = \{\text{var}\}$
49. $\text{FIRST}+(\text{factor} \rightarrow \text{call}) = \{\text{call}\}$
50. $\text{FIRST}+(\text{factor} \rightarrow \text{NUMBER}) = \{\text{NUMBER}\}$
51. $\text{FIRST}+(\text{call} \rightarrow \text{ID} (\text{args})) = \{\text{ID}\}$
52. $\text{FIRST}+(\text{args} \rightarrow \text{arg_list}) = \text{FIRST}(\text{arg_list}) = \{\text{arithmetic_expression}\}$
53. $\text{FIRST}+(\text{args} \rightarrow \epsilon) = \text{FOLLOW}(\text{args}) = \{ \}$
54. $\text{FIRST}+(\text{arg_list} \rightarrow \text{arithmetic_expression arg_list}') = \{\text{arithmetic_expression}\}$
55. $\text{FIRST}+(\text{arg_list}' \rightarrow , \text{arithmetic_expression arg_list}') = \{ , \}$
56. $\text{FIRST}+(\text{arg_list}' \rightarrow , \text{arithmetic_expression}) = \{ , \}$

These were also represented as a table:

Production	First+
$\text{program} \rightarrow \text{declaration_list main_declaration}$	$\{\text{type_specifier}\}$
$\text{main_declaration} \rightarrow \text{void ID} (\text{void}$	$\{\text{void}\}$

declaration_list \rightarrow declaration_list declaration	{type_specifier}
declaration_list \rightarrow declaration	{type_specifier}
declaration \rightarrow var_declaration	{type_specifier}
declaration \rightarrow fun_declaration	{type_specifier}
var_declaration \rightarrow type_specifier ID ;	{type_specifier}
var_declaration \rightarrow type_specifier ID [NUMBER] ;	{type_specifier}
type_specifier \rightarrow int float string void	{int, float, string, void}
fun_declaration \rightarrow type_specifier ID (params	{type_specifier}
params \rightarrow param_list	{type_specifier}
params \rightarrow void	{void}
param_list \rightarrow param param_list'	{type_specifier}
param_list' \rightarrow , param param_list'	{,}
param_list' \rightarrow , param	{,}
param \rightarrow type_specifier ID	{type_specifier}
param \rightarrow type_specifier ID []	{type_specifier}
compound_stmt \rightarrow { local_declarations statement_list }	{ {} }
local_declarations \rightarrow local_declarations var_declaration	{type_specifier, ϵ }
local_declarations \rightarrow ϵ	{ID, {, if, while, return, input, output}
statement_list \rightarrow statement_list statement	{var, if, while, return, input, output, {, ϵ }
statement_list \rightarrow ϵ	{if, var, while, return, input, output, {, void, \$}
statement \rightarrow assignment_stmt call compound_stmt selection_stmt iteration_stmt return_stmt input_stmt output_stmt	{var, if, while, return, input, output, {}
assignment_stmt \rightarrow var = expression ;	{ID}
assignment_stmt \rightarrow STRING ;	{STRING}
call_stmt \rightarrow call ;	{ID}
matched_stmt \rightarrow if (expression	{if}
unmatched_stmt \rightarrow if (expression	{if}
iteration_stmt \rightarrow while (expression	{while}
return_stmt \rightarrow return ;	{return}
return_stmt \rightarrow return expression ;	{return}
input_stmt \rightarrow input var ;	{input}
output_stmt \rightarrow output expression ;	{output}
var \rightarrow ID var'	{ID}
var \rightarrow ID	{ID}
var' \rightarrow [arithmetic_expression]	{ {} }
expression \rightarrow simple_expression relop simple_expression	{term}

expression \rightarrow simple_expression	{term}
relop \rightarrow <= < > >= == !=	{<=, <, >, >=, ==, !=}
simple_expression \rightarrow term simple_expression'	{factor}
simple_expression' \rightarrow addop term simple_expression'	{addop}
simple_expression' \rightarrow addop term	{addop}
addop \rightarrow + -	{+, -}
term \rightarrow factor term'	{(, var, call, NUMBER}
term' \rightarrow mulop factor term'	{mulop}
mulop \rightarrow * /	{*, /}
factor \rightarrow (expression	{(}
factor \rightarrow var	{var}
factor \rightarrow call	{call}
factor \rightarrow NUMBER	{NUMBER}
call \rightarrow ID (args	{ID}
args \rightarrow arg_list	{arithmetic_expression}
args \rightarrow ϵ	{)}
arg_list \rightarrow arithmetic_expression arg_list'	{arithmetic_expression}
arg_list' \rightarrow , arithmetic_expression arg_list'	{,}
arg_list' \rightarrow , arithmetic_expression	{,}

After I formulated all the sets needed, I began to code a function that would be able to create a parsing table in the format of a csv file. This function, however, proved to be more difficult than I had imagined, since multiple attempts I did were met with failure, one of these failures should be in the folder I sended for this project. Despite my efforts to resolve the difficulties I keep facing, the table proved too complex to construct via code within the time constraints. Given the strict deadline and the complexities of the project, I was forced to reconsider my plan. After careful thought, I felt that implementing a recursive descent parser would be a more realistic plan given the conditions. This decision was primarily motivated by the need of having a functional parser by the deadline.

Design

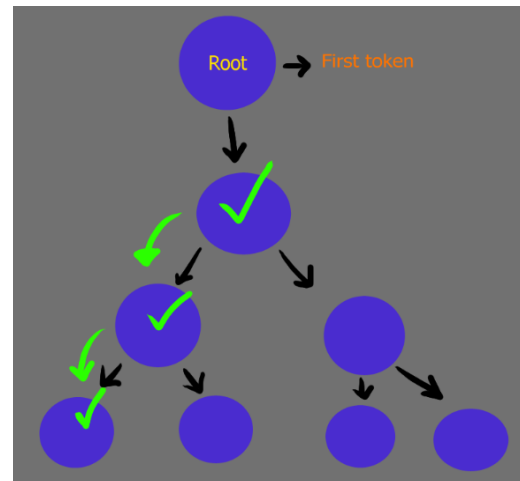
Algorithm

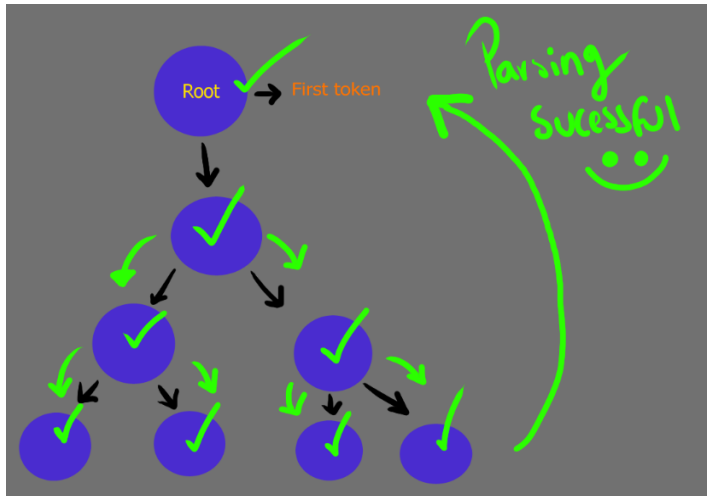
The reasons why I chose to implement a recursive descent parser besides the ones I mentioned earlier include:

1. Implementing a recursive descent parser is simpler than to implement a LL(1), since a recursive descent starts from the root and then proceeds down the parse tree in a depth-first manner, breaking down the parsing into simpler subproblems. This made it more intuitive for me to implement.
2. They tend to be more efficient for a grammar than the LL(1)
3. They are easier to debug since they follow the sequence of functions that mirror the grammar, and since I would probably need to do a lot of debugging to make sure my parser worked I consider this as an important pro to have.
4. And finally, as I mentioned earlier given my time constraints I had for this project, it was more feasible for me to implement a recursive descent parser.

The base for my design of the algorithm of my parser is based on the idea that a recursive descent parser start from the root (which would be the first token given by the scanner) and then proceed down the parse tree in a depth-first manner, recursively breaking down the parsing into subproblems, which I will illustrate in the following in order to explain my logic behind my algorithm.

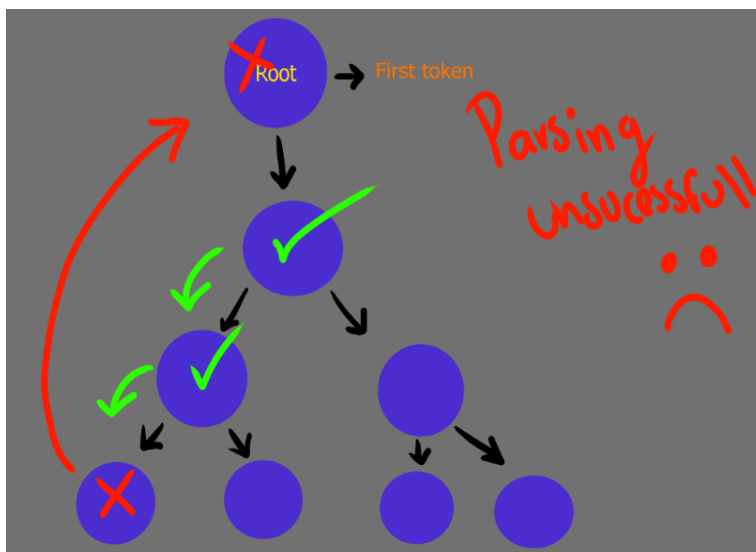
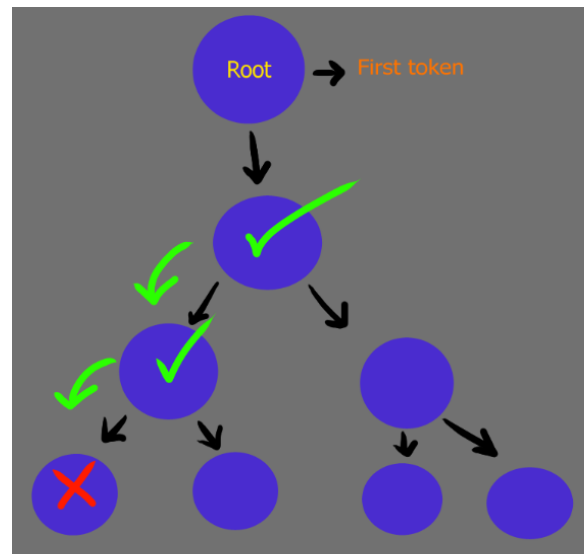
As I mentioned earlier, the parser starts by checking the root, which corresponds to the first token. It then proceeds to verify each subsequent token, using a depth-first approach to transverse the tree.





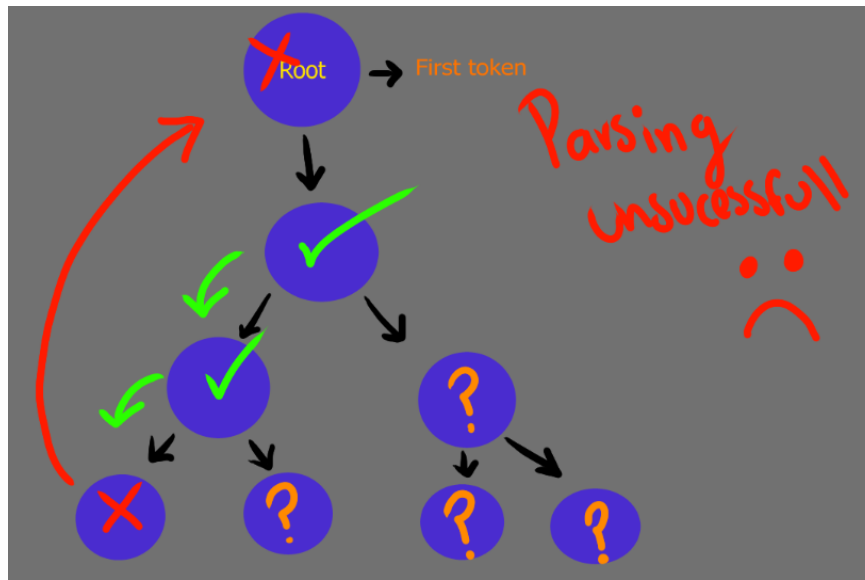
If all of the tokens under the root are successfully verified against the grammar then the parsing is deemed successful.

However if one of the tokens is found that it does not comply with the grammar



The parser will identify the error at its current location, and even if some tokens may have been correctly parsed before this point, the parser would cease its process upon encountering an error.

Consequently, any remaining tokens which were not processed before the error, will remain unparsed. The parser, upon identifying an error, terminates its parsing process and returns an error indicating which token was incorrect.



With this in mind and considering that a recursive is a kind of top-down parser built from a set of mutually recursive functions, where each function implements one of the non-terminals of the grammar. I design an overview of the algorithm of the my parser:

1. **Start:** It begins at the start symbol of my grammar rules (syntax conventions)
2. **Read the input: the parser reads the next token from the input stream**
3. **Decision of rule:** Based on the current token and grammar rule, the parser makes a decision on which rule to apply next. This decision is deterministic and is made through a parsing function associated with the current non-terminal.
4. **Recursive descent:** This step involves:
 - a. If the chosen rule's right hand side is a non-terminal symbol, the parser calls the function for that non-terminal
 - b. If it is a terminal symbol, the parser compares it with the next token from the token list given by the scanner, if they match, the parser moves to the next token in the rule's right-hand side.
 - c. If the right-hand side of the rule is empty(ϵ production), the parser does nothing and considers this rule processed.
5. **Verification:** the parser continues this process until all the tokens are consumed and all the call functions have successfully returned. If the parser has consumed all the tokens and the start function has finished processing, the tokens are accepted which means the parsing has been successful. If there is any error the parser rejects the parsing, meaning the parsing was unsuccessful.

Now using this overview I made a pseudocode for each function that my parser will have, each function implements one of the non-terminal of the grammar.

program \rightarrow *declaration list main declaration*

```
Function program:
  Call declaration_list
  Call main_declaration
  If no more tokens left and main_declared flag is true:
    Return parsing successful
  Else if main_declared flag is false:
    Throw an exception
  Else:
    Return parsing unsuccessful
```

main declaration \rightarrow *void ID (void) compound_stmt*

```
Function main_declaration:
  Check for 'void' keyword
  Check for an ID
  Check for '(' symbol
  Check for 'void' keyword
  Check for ')' symbol
  Call compound_stmt
  Set main_declared flag to True
```

declaration list \rightarrow *declaration list declaration | declaration*

```
Function declaration_list:
  Call declaration
  While there are tokens left:
    If current token is ';' symbol:
      Call declaration
    Else if current token is a keyword:
      Call declaration
  Else:
    Break
```

declaration \rightarrow *var declaration | fun declaration*

```
Function declaration:
  If current token is a keyword:
    If keyword is a type specifier:
```

```
    Call var_declaration
Else:
    Call fun_declaration
Else:
    Throw an exception
```

var_declaration \rightarrow *type specifier ID ;* | *type specifier ID [NUMBER] ;*

```
Function var_declaration:
    Check for type specifier
    Check for an ID
    Add variable to symbol_table
    If current token is ';' symbol:
        Continue
    Else if current token is '[' symbol:
        Check for a number
        Check for ']' symbol
        Check for ';' symbol
    Else if current token is '(' symbol:
        Call fun_declaration_prime
    Else:
        Throw an exception
```

fun_declaration \rightarrow *type specifier ID (params) compound stmt*

```
Function fun_declaration:
    Check for type specifier
    Check for an ID
    Add function to symbol_table
    Check for '(' symbol
    Call params
    Check for ')' symbol
    Call compound_stmt
```

params \rightarrow *param list* | *void*

```
Function params:
    If current token is 'void' keyword:
        Continue
    Else:
        Call param_list
```

param list \rightarrow *param param list*

```
Function param_list:
  Call param
  While there are tokens left and current token is ',' symbol:
    Call param
```

param \rightarrow *type specifier ID* | *type specifier ID []*

```
Function param:
  If current token type is a keyword and is either 'int', 'float',
  'string', or 'void':
    Move to next token.
  If current token type is an ID:
    Add to symbol table with appropriate attributes.
    Move to next token.
  If current token is symbol and is '[':
    Move to next token.
    If current token is symbol and is ']':
      Move to next token.
    Else:
      Raise an error ('Expected "]"')
  Else:
    Raise an error ('Expected ID')
  Else:
    Raise an error ('Expected type specifier')
```

compound stmt \rightarrow { *local declarations statement list* }

```
Function compound_stmt:
  If current token type is symbol and is '{':
    Move to next token.
    Execute function local_declarations.
    Execute function statement_list.
  If current token type is symbol and is '}':
    Move to next token.
  Else:
    Raise an error ('Expected "]"')
  Else:
    Raise an error ('Expected "{"')
```

local_declarations \rightarrow *local_declarations* *var_declaration* | ϵ

```
Function local_declarations:
    While current token is a keyword and is either 'int', 'float',
    'string', or 'void':
        Execute function var_declaration.
```

statement_list \rightarrow *statement_list* *statement* | ϵ

```
Function statement_list:
    While current token is not a symbol and is not '}':
        Execute function Statement.
```

statement \rightarrow *assignment_stmt* | *call* | *compound_stmt* | *selection_stmt* | *iteration_stmt* |
return_stmt | *input_stmt* | *output_stmt*

```
Function statement:
    Depending on the current token, execute the corresponding function.
    Raise an error if none of the conditions are met ('Invalid
    statement').
```

assignment_stmt \rightarrow *var* = *expression* ; | *STRING* ;

```
Function assignment_stmt:
    Validate and handle assignment operation.
    Validate and handle string.
    If conditions are not met, raise an error ('Expected ID or STRING')
```

call_stmt \rightarrow *call* ;

```
Function call_stmt:
    Execute function call.
    Validate and move to next token.
    If conditions are not met, raise an error ('Expected ";"')
```

matched_stmt \rightarrow *if* (*expression*) *matched_stmt* *else matched_stmt matched_stmt* | *other_stmt*

```
Function matched_stmt:
    Handle matched statements following a specific sequence of symbols
    and tokens.
    If conditions are not met, raise errors accordingly.
```


unmatched_stmt \rightarrow *if (expression) statement unmatched_stmt' | if (expression) matched_stmt*
else unmatched_stmt unmatched_stmt'

```
Function unmatched_stmt:  
    Handle unmatched statements following a specific sequence of  
    symbols and tokens.  
    If conditions are not met, raise errors accordingly.
```

selection_stmt

```
Function selection_stmt:  
    If current token is keyword 'if':  
        Read next token  
        If current token is symbol '(':  
            Read next token  
            Call expression function  
            If current token is symbol ')':  
                Read next token  
                Call statement function  
                If current token exists and is keyword 'else':  
                    Read next token  
                    Call statement function  
            Else, raise exception 'Expected ")"'  
        Else, raise exception 'Expected "("'  
    Else, raise exception 'Expected "if"'
```

iteration_stmt

```
Function iteration_stmt:  
    If current token is keyword 'while':  
        Read next token  
        If current token is symbol '(':  
            Read next token  
            Call expression function  
            If current token is symbol ')':  
                Read next token  
                Call statement function  
            Else, raise exception 'Expected ")"'  
        Else, raise exception 'Expected "("'  
    Else, raise exception 'Expected "while"'
```

return_stmt

```
Function return_stmt:
    If current token is keyword 'return':
        Read next token
        If current token is symbol ';':
            Read next token
        Else:
            Call expression function
            If current token is symbol ';':
                Read next token
            Else, raise exception 'Expected ";"'
    Else, raise exception 'Expected "return"'
```

input_stmt

```
Function input_stmt:
    If current token is keyword 'input':
        Read next token
        If current token is an identifier:
            Read next token
            If current token is symbol ';':
                Read next token
            Else, raise exception 'Expected ";"'
        Else, raise exception 'Expected ID'
    Else, raise exception 'Expected "input"'
```

output_stmt

```
Function output_stmt:
    If current token is keyword 'output':
        Read next token
        Call expression function
        If current token is symbol ';':
            Read next token
        Else, raise exception 'Expected ";"'
    Else, raise exception 'Expected "output"'
```

var \rightarrow *ID* *var*' | *ID*

```
Function var:
```

```
If current token is an identifier:
    Read next token
    If current token is symbol '[':
        Read next token
        Call arithmetic_expression function
        If current token is symbol ']':
            Read next token
    Else, raise exception 'Expected ID'
```

$simple_expression \rightarrow term\ simple_expression' \mid term$

```
Function simple_expression:
    Call term function
    While current token exists and is either '+' or '-':
        Read next token
        Call Term function
```

$term \rightarrow factor\ term' \mid factor$

```
Function term:
    Call factor function
    While current token exists and is either '*' or '/':
        Read next token
        Call Factor function
```

$factor \rightarrow (expression) \mid var \mid call \mid NUMBER$

```
Function factor:
    If current token is symbol '(':
        Read next token
        Call expression function
        If current token is symbol ')':
            Read next token
    Else If current token is an identifier:
        Validate identifier
        Read next token
        If current token exists and is symbol '[':
            Read next token
            Call arithmetic_expression function
            If current token exists and is symbol ']':
```

```
        Read next token
    Else If current token is a number:
        Read next token
    Else If current token is keyword 'call':
        Call call function
    Else, raise exception 'Invalid factor'
```

arithmetic_expression

```
Function arithmetic_expression:
    Call term function
    While current token exists and is either '+' or '-':
        Read next token
        Call term function
```

args \rightarrow *arg_list* | ϵ

```
Function args:
    While current token is not symbol ')':
        Call arithmetic_expression function
    If current token is symbol ',':
        Read next token
        Call arithmetic_expression function
```

arg_list \rightarrow *arithmetic_expression arg_list'* | *arithmetic_expression*

```
Function arg_list:
    Call arithmetic_expression function
    While current token is symbol ',':
        Read next token
        Call arithmetic_expression function
```

arg_list' \rightarrow , *arithmetic_expression arg_list'* | , *arithmetic_expression*

```
Function arg_list_prime:
    If current token is symbol ',':
        Read next token
        Call arithmetic_expression function
        Call arg_list_prime function
```

call \rightarrow *ID* (*args*)

```
Function call:
  If current token is an identifier:
    Read next token
    If current token is symbol '(':
      Read next token
      Call args function
      If current token is symbol ')':
        Read next token
      Else, raise exception 'Expected ")"'
    Else, raise exception 'Expected "("'
  Else, raise exception 'Expected ID'
```

After finishing these pseudocodes for the non-terminals, I also created the pseudocodes of the other functions that while they don't represent a non-terminal they are essential in order to make the parser work correctly.

Function that starts the parser

```
Constructor function (__init__):
  Set tokens_dict to result of calling transform_tokens_dict
  function on input tokens_dict
  Set token_list to input token_list
  Set pos to 0
  Set current_token to token_list at position pos
  Initialize symbol_table as an empty dictionary
  Set main_declared to False
```

Function that transforms the dictionary obtained by the scanner, since the dictionary obtained by the scanner maps the type of token with a dictionary for the parser we need that dictionary to be the opposite

```
Function transform_tokens_dict:
  Initialize new_tokens_dict as an empty dictionary
  For each token_type, (token_id, token_map) pair in input
  tokens_dict:
    Create new_token_map by swapping keys and values of token_map
    Store (token_id, new_token_map) under token_type key in
  new_tokens_dict
  Return new_tokens_dict
```

Function that updates the current token to the next token

```
Function next_token:
    Increment pos by 1
    If pos is less than the length of token_list:
        Set current_token to token_list at position pos
    Else:
        Set current_token to None
```

Function that returns the previous token of the current token

```
Function last_token:
    Set lastPos to pos minus 1
    If lastPos is greater than or equal to 0:
        Return token_list at position lastPos
    Else:
        Return None
```

Function that checks if the symbol has already been declared based on the restriction of "Functions MUST be declared before they are called " searches the symbol table if the variable or function has already been declared

```
Function validate_id:
    If input token is not in symbol_table:
        Set val_name to the key corresponding to the input token in
the ID sub-dictionary of tokens_dict
        Raise Exception stating "Identifier val_name has not been
declared"
```

Data Structures

1. **tokens_dict:** It is the dictionary that we take from the scanner, which represents the symbol table without being updated by the parser
2. **token_list:** It is the list of tokens obtained from the scanner, the position that the tokens are found is saved to ensure the precedence of the tokens
3. **symbol_table:** Dictionary with position of the token in the token list mapped with name (token ID), if it is a variable or function and type of token. This dictionary represents the symbol table updated by the parser.

Architecture:

Scanner

- **Input source code:** The scanner will get the source code, this source code consists of various components such as identifiers, literals operators delimiters.
- **Character Stream:** The scanner then transforms the input by processed piece-by-piece of the input using characters' streams accessible to scanners.
- **Finite State Machine(FSM):** It is responsible for recognizing lexical components in the character stream. It has several states to represent specific token types or stages in intermediate token recognition. By analyzing the current state and present character, the FSM gracefully moves between states to achieve its objective.
- **Transition Table:** It is used to define how to move between states, the scanner employs a transition table. This chart maps present situations along with character classifications to identify subsequent moves or actions. The FSM relies heavily on this vital reference when navigating scans.
- **Token Generation:** Every time the scanner processes character streams through its FSM and transition table, it generates tokens for recognized lexical components. These tokens have both types, such as keyword or number, and their corresponding values.
- **Token Stream:** After the scanner has processed all the characters, it makes a token stream. The scanner delivers this output after consuming an entire reservoir of characters.

Parser

- **Token transformation and initialization** (init and transform_tokens_dict): The parser receives a dictionary of tokens and a token list. It transforms the token dictionary to have a token map where keys are the token values and the values are the original keys. It also sets the current token and initializes an empty symbol table.
- **Token Navigation** (next_token and last_token): These functions are responsible for moving forward (and checking the previous token if needed) in the token list.
- **Syntax Rules** (selection_stmt, iteration_stmt, return_stmt, etc): Each of these functions represents a grammar rule of the language. They are responsible for parsing different language constructs. They recursively call each other and the token navigation functions to parse the language construct they are responsible for. If the current token does not match the expected token for the grammar rule, an exception is thrown.

Implementation

```
class Parser:
    # Funcion que inicia el parser
    # self.token_dict: Es el diccionario que tomamos del scanner, que representaria el symbol table sin actualizar
    # self.token_list: Es la lista de los tokens obtenidos del scanner, guardando la posicion que se van encontrando los tokens
    # self.pos: La posicion actual que indica que token esta analizando el parser
    # self.symbol_table: Diccionario con posicion del token en el token_list mapeado con nombre (token ID), si es variable o funcion y tipo de token.
    # Este representa el symbol table actualizado por el parser.
    # self.main_declared: Flag que indica si el programa tiene main function, basandose en la restriccion de
    # "The last declaration in a program MUST be a function declaration of the form void main(void) ""
    def __init__(self, tokens_dict, token_list):
        self.tokens_dict = self.transform_tokens_dict(tokens_dict)
        self.token_list = token_list
        self.pos = 0
        self.current_token = self.token_list[self.pos]
        self.symbol_table = {}
        self.main_declared = False
```

```
# Funcion que transforma el diccionario obtenido por el scanner
# El diccionario obtenido por el scanner, mapea el tipo de token con un diccionario
# para el parser necesitamos que ese diccionario este al contrario
def transform_tokens_dict(self, tokens_dict):
    new_tokens_dict = {}
    for token_type, (token_id, token_map) in tokens_dict.items():
        new_token_map = {value: key for key, value in token_map.items()}
        new_tokens_dict[token_type] = (token_id, new_token_map)
    return new_tokens_dict

# Funcion que actualiza self.current_token al siguiente token
def next_token(self):
    self.pos += 1
    if self.pos < len(self.token_list):
        self.current_token = self.token_list[self.pos]
    else:
        self.current_token = None # ya no hay mas tokens
```



```
#Funcion que regresa el anterior token del self.current_token
def last_token(self):
    lastPos = self.pos
    lastPos -= 1
    if lastPos >= 0:
        return self.token_list[lastPos]
    else:
        return None # no existe un token anterior

# Funcion que valida si el symbol ya fue declarado
# basada en la restriccion de "Functions MUST be declared before they are called"
# Tal vez asumi de mas, pero tambien valida variables
def validate_id(self, token):
    if token not in self.symbol_table:
        val_name = self.tokens_dict['ID'][1][token]
        raise Exception("Identifier {} has not been declared".format(val_name))
```

```
# Funcion recursiva para el non-terminal:
# program + declaration_list main_declaration
def program(self):
    try:
        self.declaration_list()
        self.main_declaration()
    # Si ya no hay mas tokens entonces obtenemos este error, que significa que el parser parcio todos los tokens sin dar un error
    # Por lo cual asumimos que el parseo fue exitoso, atrapamos el error y regresamos True
    except TypeError as e:
        if str(e) == "'NoneType' object is not subscriptable":
            # Si ya no tenemos tokens, pero la funcion main no fue declarada entonces creamos una exeptions ya que no cumple con los requerimientos
            if self.main_declared:
                return True
            else:
                raise Exception('The last declaration in a program must be a function declaration of the form void main(void)')
        else:
            return False
```

```
# Funcion recursiva para el non-terminal:
# main_declaration + void ID ( void ) compound_stmt
def main_declaration(self):
    if self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'void':
        self.next_token()
    if self.current_token[0] == self.tokens_dict['ID'][0]:
        self.next_token()
    if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '(':
        self.next_token()
        if self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'void':
            self.next_token()
            if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ')':
                self.next_token()
                self.compound_stmt()
                # Si la funcion main es declarada por completo entonces modificamos el flag
                self.main_declared = True
            else:
                raise Exception('Expected ")"')
        else:
            raise Exception('Expected "void"')
    else:
        raise Exception('Expected "("')
    else:
        raise Exception('Expected ID')
    else:
        raise Exception('Expected "void"')
```

```
# Funcion recursiva para el non-terminal:
# declaration_list → declaration_list declaration | declaration
def declaration_list(self):
    self.declaration()
    while self.current_token:
        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ';':
            self.next_token()
            self.declaration()
        # En el caso que se este declarando variables afuera de una funcion
        elif self.current_token[0] == self.tokens_dict['KEYWORD'][0]:
            self.declaration()
        else:
            break

# Funcion recursiva para el non-terminal:
# declaration → var_declaration | fun_declaration
def declaration(self):
    if self.current_token[0] == self.tokens_dict['KEYWORD'][0]:
        if self.tokens_dict['KEYWORD'][1][self.current_token[1]] in ['int', 'float', 'string', 'void']:
            self.var_declaration()
        else:
            self.fun_declaration()
    else:
        raise Exception('Invalid declaration')
```

```
# Funcion recursiva para el non-terminal:
# var_declaration → type_specifier ID ; | type_specifier ID [ NUMBER ] ;
def var_declaration(self):
    if self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] in ['int', 'float', 'string', 'void']:
        self.next_token()
        if self.current_token[0] == self.tokens_dict['ID'][0]:
            # Checa en el caso que se este declarando la funcion de main, solo pasa si es la unica funcion en el codigo
            # Pueda que sea un error de mi gramatica pero no me baje puntos :,c
            if self.tokens_dict['ID'][1][self.current_token[1]] == "main":
                self.main_declared = True
            # Si se esta declarando una variable entonces tambien se agrega al diccionario self.symbol_table
            self.symbol_table[self.current_token[1]] = [self.tokens_dict['ID'][1][self.current_token[1]], 'variable', self.tokens_dict['KEYWORD'][1][self.last_token()[1]]]
            self.next_token()
            if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ';':
                self.next_token()
            elif self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '[':
                self.next_token()
                if self.current_token[0] == self.tokens_dict['NUMBER'][0]:
                    self.next_token()
                    if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ']':
                        self.next_token()
                        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ';':
                            self.next_token()
                        else:
                            raise Exception('Expected ";"')
                    else:
                        raise Exception('Expected "]"')
                else:
                    raise Exception('Expected NUMBER')
            else:
                raise Exception('Expected type specifier')
```

```
else:
    # En el caso que en realidad se este declarando fun_declaration y no var_declaration
    # Tambien esto causa que algunas funciones se pongan como variables en self.symbol_table
    if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '(':
        self.fun_declaration_prime()
        return
    else:
        raise Exception('Expected ";" or "["')
else:
    raise Exception('Expected ID')
else:
    raise Exception('Expected type specifier')
```

```
# Funcion recursiva para el non-terminal:
# fun_declaration → type_specifier ID ( params ) compound_stmt
def fun_declaration(self):
    if self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] in ['int', 'float', 'string', 'void']:
        self.next_token()
        if self.current_token[0] == self.tokens_dict['ID'][0]:
            # Si se esta declarando una funcion entonces tambien se agrega al diccionario self.symbol_table
            self.symbol_table[self.current_token[1]] = [self.tokens_dict['ID'][1][self.current_token[1]], 'function', self.tokens_dict['KEYWORD'][1][self.last_token()[1]]]
            self.next_token()
            if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '(':
                self.next_token()
                self.params()
                if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ')':
                    self.next_token()
                    self.compound_stmt()
                else:
                    raise Exception('Expected ")')
            else:
                raise Exception('Expected "("')
        else:
            raise Exception('Expected ID')
    else:
        raise Exception('Expected type specifier')
```

```
# Funcion que es usada en el caso que cuando se este a la mitad del proceso de declarar una variable en var_declaration
# nos damos cuenta que en realidad es una funcion, es similar a la anterior funcion solo que empeiza desde los parametros
def fun_declaration_prime(self):
    if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '(':
        self.next_token()
        self.params()
        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ')':
            self.next_token()
            self.compound_stmt()
        else:
            raise Exception('Expected ")')
    else:
        raise Exception('Expected "("')

# Funcion recursiva para el non-terminal:
# params → param_list | void
def params(self):
    if self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'void':
        self.next_token()
    else:
        self.param_list()
```

```
# Funcion recursiva para el non-terminal:
# param_list → param param_list'
# Tambien abarca: param_list' → , param param_list' | , param
def param_list(self):
    self.param()
    while self.current_token and self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ',':
        self.next_token()
        self.param()

# Funcion recursiva para el non-terminal:
# param → type_specifier ID | type_specifier ID [ ]
def param(self):
    if self.tokens_dict['KEYWORD'][1][self.current_token[1]] in ['int', 'float', 'string', 'void']:
        self.next_token()
        if self.current_token[0] == self.tokens_dict['ID'][0]:
            # Si se esta declarando una variable en los parametros
            # entonces tambien se agregan al diccionario self.symbol_table
            self.symbol_table[self.current_token[1]] = [self.tokens_dict['ID'][1][self.current_token[1]], 'variable', self.tokens_dict['KEYWORD'][1][self.last_token()[1]]]
            self.next_token()
            if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '[':
                self.next_token()
                if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ']':
                    self.next_token()
                else:
                    raise Exception('Expected "]"')
            else:
                raise Exception('Expected ID')
        else:
            raise Exception('Expected type specifier')
```

```
# Funcion recursiva para el non-terminal:
# compound_stmt -> { local_declarations statement_list }
def compound_stmt(self):
    if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '{':
        self.next_token()
        self.local_declarations()
        self.statement_list()
        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '}':
            self.next_token()
        else:
            raise Exception('Expected "}"')
    else:
        raise Exception('Expected "{"')
```

```
# Funcion recursiva para el non-terminal:
# local_declarations -> local_declarations var_declaration | ε
def local_declarations(self):
    while self.current_token and self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] in ['int', 'float', 'st']:
        self.var_declaration()

# Funcion recursiva para el non-terminal:
# statement_list -> statement_list statement | ε
def statement_list(self):
    while self.current_token and self.current_token[0] != self.tokens_dict['SYMBOL'][0] and self.current_token[1] != '}':
        self.statement()
```

```
# Funcion recursiva para el non-terminal:
# statement -> assignment_stmt | call | compound_stmt | selection_stmt | iteration_stmt | return_stmt | input_stmt | output_stmt
def statement(self):
    if self.current_token[0] == self.tokens_dict['ID'][0]:
        self.assignment_stmt()
    elif self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'call':
        self.call()
    elif self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '{':
        self.compound_stmt()
    elif self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'if':
        self.selection_stmt()
    elif self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'while':
        self.iteration_stmt()
    elif self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'return':
        self.return_stmt()
    #Esto esta incorrecto es Read
    elif self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'input':
        self.input_stmt()
    #Incorrecto es write
    elif self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'output':
        self.output_stmt()
    else:
        raise Exception('Invalid statement')
```

```
# Funcion recursiva para el non-terminal:
# assignment_stmt -> var = expression ; | STRING ;
def assignment_stmt(self):
    if self.current_token[0] == self.tokens_dict['ID'][0]:
        # Al momento de asignar un valor a una variable, verificamos que la variable ya fue declarada
        if self.current_token[1] not in self.symbol_table:
            # Usamos el current_token para ver el nombre de la variable, para poder regresar la excepcion con el ID de la variable
            val_name = self.tokens_dict['ID'][1][self.current_token[1]]
            raise Exception("Identifier {} has not been declared".format(val_name))
        self.next_token()
    if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '=':
        self.next_token()
        self.expression()
        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ';':
            self.next_token()
        else:
            raise Exception("Expected ';'")
    else:
        try:
            self.call_prime()
        except:
            raise Exception("Expected '='")
    elif self.current_token[0] == self.tokens_dict['STRING'][0]:
        self.next_token()
        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ';':
            self.next_token()
        else:
            raise Exception("Expected ';'")
    else:
        raise Exception("Expected ID or STRING")
```

```
# Funcion recursiva para el non-terminal:
# call_stmt -> call ;
def call_stmt(self):
    self.call()
    if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ';':
        self.next_token()
    else:
        raise Exception("Expected ';'")

# Funcion recursiva para el non-terminal:
# matched_stmt -> if ( expression ) matched_stmt else matched_stmt matched_stmt | other_stmt
def matched_stmt(self):
    if self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'if':
        self.next_token()
        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '(':
            self.next_token()
            self.expression()
            if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ')':
                self.next_token()
                self.matched_stmt()
            if self.current_token and self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'else':
                self.next_token()
                self.matched_stmt()
            else:
                raise Exception("Expected ')'")
        else:
            raise Exception("Expected '('")
    else:
        self.other_stmt()
```

```
# Funcion recursiva para el non-terminal:
# unmatched_stmt -> if ( expression ) statement unmatched_stmt' | if ( expression ) matched_stmt else unmatched_stmt unmatched_stmt'
def unmatched_stmt(self):
    if self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'if':
        self.next_token()
        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '(':
            self.next_token()
            self.expression()
            if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ')':
                self.next_token()
                self.statement()
                self.unmatched_stmt_prime()
            else:
                raise Exception('Expected ")"')
        else:
            raise Exception('Expected "("')
    else:
        raise Exception('Expected "if"')

# Funcion recursiva para el non-terminal: unmatched_stmt'
def unmatched_stmt_prime(self):
    if self.current_token and self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'else':
        self.next_token()
        self.statement()
    else:
        self.unmatched_stmt()
```

```
# Funcion recursiva para el non-terminal: selection_stmt
# Representa un if-else, y esta relacionado con:
# statement -> assignment_stmt | call | compound_stmt | selection_stmt | iteration_stmt | return_stmt | input_stmt | output_stmt
def selection_stmt(self):
    if self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'if':
        self.next_token()
        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '(':
            self.next_token()
            self.expression()
            if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ')':
                self.next_token()
                self.statement()
                if self.current_token and self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'else':
                    self.next_token()
                    self.statement()
            else:
                raise Exception('Expected ")"')
        else:
            raise Exception('Expected "("')
    else:
        raise Exception('Expected "if"')
```

```
# Funcion recursiva para el non-terminal:
# iteration_stmt -> while ( expression ) statement
def iteration_stmt(self):
    if self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'while':
        self.next_token()
        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '(':
            self.next_token()
            self.expression()
            if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ')':
                self.next_token()
                self.statement()
            else:
                raise Exception('Expected ")"')
        else:
            raise Exception('Expected "("')
    else:
        raise Exception('Expected "while"')
```

```
# Funcion recursiva para el non-terminal:
# return_stmt → return ; | return expression ;
def return_stmt(self):
    if self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'return':
        self.next_token()
    if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ';':
        self.next_token()
    else:
        self.expression()
        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ';':
            self.next_token()
        else:
            raise Exception('Expected ";"')
    else:
        raise Exception('Expected "return"')

# Funcion recursiva para el non-terminal:
# input_stmt → input var ;
def input_stmt(self):
    if self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'input':
        self.next_token()
    if self.current_token[0] == self.tokens_dict['ID'][0]:
        self.next_token()
    if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ';':
        self.next_token()
    else:
        raise Exception('Expected ";"')
    else:
        raise Exception('Expected ID')
    else:
        raise Exception('Expected "input"')
```

```
# Funcion recursiva para el non-terminal:
# output_stmt → output expression ;
def output_stmt(self):
    if self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'output':
        self.next_token()
    self.expression()
    if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ';':
        self.next_token()
    else:
        raise Exception('Expected ";"')
    else:
        raise Exception('Expected "output"')

# Funcion recursiva para el non-terminal:
# var → ID var' | ID
# var' → [ arithmetic_expression ]
def var(self):
    if self.current_token[0] == self.tokens_dict['ID'][0]:
        self.next_token()
        # En caso de ser var' se checa []
        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '[':
            self.next_token()
            self.arithmetic_expression()
            if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ']':
                self.next_token()
        else:
            raise Exception('Expected ID')
```

```
# Funcion recursiva para el non-terminal:
# simple_expression → term simple_expression' | term
def simple_expression(self):
    self.term()
    # La terminal addop: addop → + | -
    # esta en la segunda parte: self.tokens_dict['SYMBOL'][1][self.current_token[1]] in ['+', '-']
    while self.current_token and self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] in ['+', '-']:
        self.next_token()
        self.term()

# Funcion recursiva para el non-terminal:
# term → factor term' | factor
def term(self):
    self.factor()
    # La terminal mulop: mulop → * | /
    # esta en la segunda parte: self.tokens_dict['SYMBOL'][1][self.current_token[1]] in ['*', '/']
    while self.current_token and self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] in ['*', '/']:
        self.next_token()
        self.factor()
```

```
# Funcion recursiva para el non-terminal:
# factor → ( expression ) | var | call | NUMBER
def factor(self):
    if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '(':
        self.next_token()
        self.expression()
        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ')':
            self.next_token()
    elif self.current_token[0] == self.tokens_dict['ID'][0]:
        self.validate_id(self.current_token[1])
        self.next_token()
    elif self.current_token and self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '[':
        self.next_token()
        self.arithmetic_expression()
        if self.current_token and self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ']':
            self.next_token()
    elif self.current_token[0] == self.tokens_dict['NUMBER'][0]:
        self.next_token()
    elif self.current_token[0] == self.tokens_dict['KEYWORD'][0] and self.tokens_dict['KEYWORD'][1][self.current_token[1]] == 'call':
        self.call()
    else:
        raise Exception('Invalid factor')
```

```
# Funcion recursiva para el non-terminal:
# arithmetic_expression
def arithmetic_expression(self):
    self.term()
    while self.current_token and self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] in ['+', '-']:
        self.next_token()
        self.term()

# Funcion recursiva para el non-terminal:
# args → arg_list | ε
def args(self):
    if self.current_token[0] != self.tokens_dict['SYMBOL'][0] or self.tokens_dict['SYMBOL'][1][self.current_token[1]] != ')':
        self.arithmetic_expression()
        while self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ',':
            self.next_token()
            self.arithmetic_expression()

# Funcion recursiva para el non-terminal:
# arg_list → arithmetic_expression arg_list' | arithmetic_expression
def arg_list(self):
    self.arithmetic_expression()
    while self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ',':
        self.next_token()
        self.arithmetic_expression()
```



```
# Funcion recursiva para el non-terminal:
# arg_list' → , arithmetic_expression arg_list' | , arithmetic_expression
def arg_list_prime(self):
    if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ',':
        self.next_token()
        self.arithmetic_expression()
        self.arg_list_prime()

# Funcion recursiva para el non-terminal:
# call → ID ( args )
def call(self):
    if self.current_token[0] == self.tokens_dict['ID'][0]:
        self.next_token()
        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '(':
            self.next_token()
            self.args()
            if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ')':
                self.next_token()
            else:
                raise Exception('Expected ")')
        else:
            raise Exception('Expected "("')
    else:
        raise Exception('Expected ID')
```

```
# Funcion recursiva para el non-terminal: call'
def call_prime(self):
    if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == '(':
        self.next_token()
        self.args()
        if self.current_token[0] == self.tokens_dict['SYMBOL'][0] and self.tokens_dict['SYMBOL'][1][self.current_token[1]] == ')':
            self.next_token()
        else:
            raise Exception('Expected ")')
    else:
        raise Exception('Expected "("')
```

Testing

My test model for my parser consists a unit test with 6 different cases:

1. void main (void): In this test case I make sure that the parser is able to parse correctly even if the only function declared in the code is the main function

```
testCode = '''  
void main (void){}  
...  
'''
```

This test case creates the following symbol table:

```
✓ symbol_table: {1: ['main', 'variable', 'void']}  
  > special variables  
  > function variables  
✓ 1: ['main', 'variable', 'void']  
  > special variables  
  > function variables  
    0: 'main'  
    1: 'variable'  
    2: 'void'  
    len(): 3  
    len(): 1
```

And has the following token dictionary:

```
✓ tokens_dict: {'KEYWORD': (1, {...}), 'SYMBOL': (2, {...}), 'ID': (3, {...}), 'STRING': (4, {}), 'NUMBER': (5, {...})}  
  > special variables  
  > function variables  
  > 'KEYWORD': (1, {1: 'int', 2: 'float', 3: 'string', 4: 'for', 5: 'if', 6: 'else', 7: 'while', 8: 'return', 9: 'read', ...})  
  > 'SYMBOL': (2, {1: '(', 2: ')', 3: '{', 4: '}'})  
  > 'ID': (3, {1: 'main'})  
  > special variables  
  > function variables  
    0: 3  
  > 1: {1: 'main'}  
    len(): 2  
  > 'STRING': (4, {})  
  > 'NUMBER': (5, {})  
    len(): 5
```

The taste case when compiled produces this output:

```

C:\Users\Estef\OneDrive\Documents\Scanner_A01635062>python parserUnitTest.py
Parsing completed successfully
.
-----
Ran 1 test in 201.693s

OK

```

2. testFun: In this test case I make sure that the parser is able to parse correctly a case scenario where the code is completely correct.

```

testCode = '''
int testFun(int nums []){
    int a;
    int b;
    a = a + 1;
    if (a < b) {
        b = b * a;
    }
}

void main (void){
    int nums [3];
    testFun(nums)
}
'''

```

This test case creates the following symbol table:

```

symbol_table: {1: ['testFun', 'variable', 'int'], 2: ['nums', 'variable', 'int'], 3: ['a', 'variable', 'int'], 4: ['b', 'variable', 'int'], 5: ['main', 'variable', 'void']}
> special variables
> function variables
> 1: ['testFun', 'variable', 'int']
> 2: ['nums', 'variable', 'int']
> 3: ['a', 'variable', 'int']
> 4: ['b', 'variable', 'int']
> 5: ['main', 'variable', 'void']
len(): 5

```

And has the following token dictionary:

```

tokens_dict: {'KEYWORD': (1, {...}), 'SYMBOL': (2, {...}), 'ID': (3, {...}), 'STRING': (4, {}), 'NUMBER': (5, {...})}
> special variables
> function variables
> 'KEYWORD': (1, {1: 'int', 2: 'float', 3: 'string', 4: 'for', 5: 'if', 6: 'else', 7: 'while', 8: 'return', 9: 'read', ...})
> 'SYMBOL': (2, {1: '(', 2: '[', 3: ']', 4: ')', 5: '{', 6: ';', 7: '=', 8: '+', 9: '<', ...})
> 'ID': (3, {1: 'testFun', 2: 'nums', 3: 'a', 4: 'b', 5: 'main'})
> 'STRING': (4, {})
> 'NUMBER': (5, {1: '1', 2: '3'})
len(): 5

```

The taste case when compiled produces this output:

```
Parsing completed successfully
```

```
.
```

```
-----  
Ran 1 test in 96.300s
```

```
OK
```

```
PS C:\Users\Estefi\OneDrive\Documentos\Scanner_A01635062> []
```

3. funcCount: In this test case I make sure that the parser is able recognize if a variable is used before it is declared

```
testCode = '''  
int funcCount(void) {  
    int total = 0;  
    while (count < 10) {  
        if (count <= 2) {  
            total + count;  
        } else {  
            total - count;  
        }  
        count + 1;  
    }  
    return 0;  
}  
...'''
```

This test case creates the following symbol table:

```
✓ symbol_table: {1: ['funcCount', 'variable', 'int'], 2: ['total', 'variable', 'int']}  
> special variables  
> function variables  
> 1: ['funcCount', 'variable', 'int']  
> 2: ['total', 'variable', 'int']  
len(): 2
```

And has the following token dictionary:

```
✓ tokens_dict: {'KEYWORD': (1, {...}), 'SYMBOL': (2, {...}), 'ID': (3, {...}), 'STRING': (4, {}), 'NUMBER': (5, {...})}  
> special variables  
> function variables  
> 'KEYWORD': (1, {1: 'int', 2: 'float', 3: 'string', 4: 'for', 5: 'if', 6: 'else', 7: 'while', 8: 'return', 9: 'read', ...})  
> 'SYMBOL': (2, {1: '(', 2: ')', 3: '{', 4: '=', 5: ';', 6: '<', 7: '<=', 8: '+', 9: '}', ...})  
> 'ID': (3, {1: 'funcCount', 2: 'total', 3: 'count'})  
> 'STRING': (4, {})  
> 'NUMBER': (5, {1: '0', 2: '10', 3: '2', 4: '1'})  
len(): 5
```

The test case when compiled produces this output:

```
Parsing failed: Identifier miniloc has not been declared
.
-----
Ran 1 test in 306.113s

OK
PS C:\Users\Estefi\OneDrive\Documentos\Scanner_A01635062> 
```

4. miniloc: This case was provided by the professor, in this case we make sure that if the code does not have a main function the parser is able to detect that and return an exception

```
test_parser {
testCode = '''
/* Program that reads a 10 element array of
integers, and then multiply each element of
the array by a float, stores the result into an
array of floats. Subsequently, the array of
floats is sorted and display it into standard
output.*/

int x[10];
string s;
float f1;
float f2[10];

int miniloc(float a[], int low, int high){
    int i;
    float y;
    int k;

    k = low;
    y = a[low];
    i = low + 1;
    while (i < high){
        if (a[i] < x){
            y = a[i];
            k = i;
        }
        i = i + 1;
    }
    return k;
}
}/* END of miniloc() */
...

```

This test case creates the following symbol table:

```
✓ symbol_table: {1: ['x', 'variable', 'int'], 2: ['s', 'variable', 'string'], 3: ['f1', 'variable', 'float'], 4: ['f2', 'variabl...  
> special variables  
> function variables  
> 1: ['x', 'variable', 'int']  
> 2: ['s', 'variable', 'string']  
> 3: ['f1', 'variable', 'float']  
> 4: ['f2', 'variable', 'float']  
> 5: ['miniloc', 'variable', 'int']  
> 6: ['a', 'variable', 'float']  
> 7: ['low', 'variable', 'int']  
> 8: ['high', 'variable', 'int']  
> 9: ['i', 'variable', 'int']  
> 10: ['y', 'variable', 'float']  
> 11: ['k', 'variable', 'int']  
len(): 11
```

And has the following token dictionary:

```
✓ tokens_dict: {'KEYWORD': (1, {...}), 'SYMBOL': (2, {...}), 'ID': (3, {...}), 'STRING': (4, {}), 'NUMBER': (5, {...})}  
> special variables  
> function variables  
> 'KEYWORD': (1, {1: 'int', 2: 'float', 3: 'string', 4: 'for', 5: 'if', 6: 'else', 7: 'while', 8: 'return', 9: 'read', ...})  
> 'SYMBOL': (2, {1: '[', 2: ']', 3: ';', 4: '(', 5: ',', 6: ')', 7: '{', 8: '=', 9: '+', ...})  
> 'ID': (3, {1: 'x', 2: 's', 3: 'f1', 4: 'f2', 5: 'miniloc', 6: 'a', 7: 'low', 8: 'high', 9: 'i', ...})  
> 'STRING': (4, {})  
> 'NUMBER': (5, {1: '10', 2: '1'})  
len(): 5
```

The taste case when compiled produces this output:

```
Parsing failed: The last declaration in a program must be a function declaration of the form void main(void)  
.  
-----  
Ran 1 test in 98.155s  
  
OK  
PS C:\Users\Estefi\OneDrive\Documentos\Scanner_A01635062> []
```

5. sort: This case was provided by the professor, in this case we make sure that the parser is able to recognize if a function is called before it is declared

```
void sort(float a[], int low, int high){
    int i; int k;

    i = low;
    while (i < high - 1){
        float t;
        k = miniloc(a,i,high);
        t = a[k];
        a[k] = a[i];
        a[i] = t;
        i = i +1;
    }
    return;
}/* END of sort() */
...
```

This test case creates the following symbol table:

```
symbol_table: {1: ['sort', 'variable', 'void'], 2: ['a', 'variable', 'float'], 3: ['low', 'variable', 'int'], 4: ['high', 'var...
> special variables
> function variables
> 1: ['sort', 'variable', 'void']
> 2: ['a', 'variable', 'float']
> 3: ['low', 'variable', 'int']
> 4: ['high', 'variable', 'int']
> 5: ['i', 'variable', 'int']
> 6: ['k', 'variable', 'int']
> 7: ['t', 'variable', 'float']
len(): 7
```

And has the following token dictionary:

```
tokens_dict: {'KEYWORD': (1, {...}), 'SYMBOL': (2, {...}), 'ID': (3, {...}), 'STRING': (4, {}), 'NUMBER': (5, {...})}
> special variables
> function variables
> 'KEYWORD': (1, {1: 'int', 2: 'float', 3: 'string', 4: 'for', 5: 'if', 6: 'else', 7: 'while', 8: 'return', 9: 'read', ...})
> 'SYMBOL': (2, {1: '(', 2: '[', 3: ']', 4: ',', 5: ')', 6: '{', 7: ';', 8: '=', 9: '<', ...})
> 'ID': (3, {1: 'sort', 2: 'a', 3: 'low', 4: 'high', 5: 'i', 6: 'k', 7: 't', 8: 'miniloc'})
> 'STRING': (4, {})
> 'NUMBER': (5, {1: '1'})
len(): 5
token: 8
```

The taste case when compiled produces this output:

```
Parsing failed: Identifier miniloc has not been declared
.
-----
Ran 1 test in 306.113s

OK
PS C:\Users\Estefi\OneDrive\Documentos\Scanner_A01635062> []
```

6. writeArray: This case has been provided by the professor, in this case we make sure that the parser is able to recognize if a statement is invalid

```
void writeArray(void){
    int i;
    i = 0;
    while (i < 10){
        write f2[i];
        i = i + 1;
    }
    return;
}/* END of writeArray() *
...
```

This test case creates the following symbol table:

```
✓ symbol_table: {1: ['writeArray', 'variable', 'void'], 2: ['i', 'variable', 'int']}
> special variables
> function variables
> 1: ['writeArray', 'variable', 'void']
> 2: ['i', 'variable', 'int']
len(): 2
```

And has the following token dictionary:

```
✓ tokens_dict: {'KEYWORD': (1, {...}), 'SYMBOL': (2, {...}), 'ID': (3, {...}), 'STRING': (4, {}), 'NUMBER': (5, {...})}
> special variables
> function variables
> 'KEYWORD': (1, {1: 'int', 2: 'float', 3: 'string', 4: 'for', 5: 'if', 6: 'else', 7: 'while', 8: 'return', 9: 'read', ...})
> 'SYMBOL': (2, {1: '(', 2: ')', 3: '{', 4: ';', 5: '=', 6: '<', 7: '[', 8: ']', 9: '+', ...})
> 'ID': (3, {1: 'writeArray', 2: 'i', 3: 'f2'})
> 'STRING': (4, {})
> 'NUMBER': (5, {1: '0', 2: '10', 3: '1'})
len(): 5
```

The test case when compiled produces this output:


```
Parsing failed: Invalid statement
```

```
.
```

```
-----  
Ran 1 test in 161.186s
```

```
OK
```

```
PS C:\Users\Estefi\OneDrive\Documentos\Scanner_A01635062> 
```

References

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd Edition). Pearson.
- Cooper, K., & Torczon, L. (2011). *Engineering a Compiler* (2nd Edition). Morgan Kaufmann.
- Fischer, C. N., & LeBlanc, R. J. (1988). *Crafting a Compiler with C*. Benjamin-Cummings Publishing Company.
- Grune, D., & Jacobs, C. J. H. (2008). *Parsing Techniques: A Practical Guide* (2nd Edition). Springer.