REPORT

for

Compiler Syntax Analysis

Version 1

Prepared by

Giovanni Guzmán
A01566171

Revisions

Version	Primary Author(s)	Description of Version	Date Completed
1	Giovanni Guzmán	The parser was implemented	11/06/21

1 Introduction

1.1 Summary

This report documents the Analysis, Design, Implementation, Testing, and Deployment of the compiler C--. The document will provide examples and design documents that will be useful to understand the syntax analysis deliverable of the project.

1.2 Notation

Syntax analysis utilizes the token lists generated in phase one of the compiler (the lexical analysis). This tokens now go through a process to validate if they are properly sequenced in accordance with the grammar of the language. The grammar used in the elaboration of this compiler is a context free grammar. A context free grammar, or CFG, is where a non terminal symbol (meaning that it can lead to more symbols) points to a string conformed of terminal and non terminal symbols. Its important to note that CFGs can be recursive (a symbol can call itself).
To analyze and go through the token list a Top-Down Parser (TDP) is going to be implemented. This parser begins from the start symbol of a given grammar and applies production rules for each individual non terminal until it gets to a string formed by terminals. If it can’t reach this string it throws an error. TDPs have several subcategories. The one used will be a predictive parser. This parser relies on information about the symbols that can be generated from a production or rule.
The predictive parser needs to know the FIRST set of all the productions and rules of the grammar. A FIRST set refers to all the terminal symbols that can appear in some string. The parser also needs a FOLLOW set which is the same as the FIRST set but for the next terminals in line after the FIRST set. Lastly, the parser needs a FIRST+ Set. Which is the notation that the parser accepts that takes into account both the FIRST and FOLLOW sets.
For the analysis a class diagram was decided as the main design document of the project. This diagram provides with the interactions between components and the functionality of each.
Python 3 will be used to develop this compiler. It may not be the fastest or the most robust language but its ease of use will allow for all the resources to be working on putting together the compiler and not how to do it. The focus is more on the concepts and not on how to use the programming language. [1]

2 Analysis

2.1 Requirements

Develop a parser for the C-Minus programming language, which is essentially a subset of the C language. The grammatical conventions of the language are:

2.1.1 Syntax Conventions

1. *program* → *declaration_list*
2. *declaration_list* → *declaration_list declaration* | *declaration*
3. *declaration* → *var_declaration* | *fun_declaration*
4. *var_declaration* → *type_specifier* **ID ;** | *type_specifier* **ID [NUM] ;**
5. *type_specifier* → **int** | **void**
6. *fun_declaration* → *type_specifier* **ID (** *params* **)** *compound_stmt*
7. *params* → *param_list* | **void**
8. *param_list* → *param_list* **,** *param* | *param*
9. *param* → *type_specifier* **ID** | *type_specifier* **ID [ ]**
10. *compound_stmt* → **{** *local_declarations statement_list* **}**
11. *local_declarations* → *local_declarations var_declaration* | ε
12. *statement_list* → *statement_list statement* | ε
13. *statement* → *assignment_stmt* | *call_stmt* | *compound_stmt* | *selection_stmt* | *iteration_stmt* | *return_stmt* | *input_stmt* | *output_stmt*
14. *assignment_stmt* → *var* **=** *expression* **;**
15. *call_stmt* → *call* **;**
16. *selection_stmt* → **if (** *expression* **)** *statement* | **if (** *expression* **)** *statement* **else** *statement*
17. *iteration_stmt* → **while (** *expression* **)** *statement*
18. *return_stmt* → **return ;** | **return** *expression* **;**
19. *input_stmt* → **input** *var* **;**
20. *output_stmt* → **output** *expression* **;**
21. *var* → **ID** | **ID [** *arithmetic_expression* **]**

22. *expression* → *arithmetic_expression relop arithmetic_expression* | *arithmetic_expression*
23. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**
24. *arithmetic_expression* → *arithmetic_expression addop term* | *term*
25. *addop* → **+** | **-**
26. *term* → *term mulop factor* | *factor*
27. *mulop* → ***** | **/**
28. *factor* → **(** *arithmetic_expression* **)** | *var* | *call* | **NUM**
29. *call* → **ID (** *args* **)**
30. *args* → *args_list* | ε
31. *args_list* → *args_list* **,** *arithmetic_expression* | *arithmetic_expression*

Figure 1: Syntax Given

2.1.2 Semantics:

The following images describe the entire semantic given.

A short explanation of the meaning of each grammar rule is provided.

1. *program* → *declaration_list*
2. *declaration_list* → *declaration_list declaration* | *declaration*
3. *declaration* → *var_declaration* | *fun_declaration*

A program consists of a sequence of declarations. The sequence may have function or variable declaration, and the order does not matter. There has to be at least one declaration.

The following restrictions MUST be complied:

- All variables and functions must be declared before they are used.
- The last declaration in a program MUST be a function declaration of the form **void** main(**void**).
- Observe that C Minus does not have "prototypes", therefore, there is no distinction between declarations and prototypes.

4. *var_declaration* → *type_specifier* **ID ;** | *type_specifier* **ID [NUM] ;**
5. *type_specifier* → **int** | **void**

A variable declaration states either a simple variable of integer type or an array variable of integer type with indices from 0 to NUM-1. In C Minus, the only basic types are **int** and **void**.

Figure 2: Semantics Part 1

The following restrictions MUST be complied:

- In a variable declaration, only the type specifier **int** can be used, **void** is for function declarations.
- Only one variable can be declared per declaration.

6. *fun_declaration* → *type_specifier* **ID** **(** *params* **)** *compound_stmt*
7. *params* → *param_list* | **void**
8. *param_list* → *param_list* **,** *param* | *param*
9. *param* → *type_specifier* **ID** | *type_specifier* **ID [ ]**

A function declaration consists of a return type specifier, an identifier, and a comma-separated list of parameters inside parentheses, followed by a compound statement that contains the code of the function. If the return type of the function is **void**, then the function returns no value. Parameters of a function are either **void** (i.e., there are no parameters) or a list that represents the function's parameters. Parameters followed by brackets are array parameters whose size can vary.

The following restrictions MUST be complied:

- Simple integer parameters are passed by value.
- Array parameters are passed by reference (i.e., as pointers), and MUST be matched by an array variable during a call.
- There are no parameters of type function.
- The parameters of a function have scope equal to the compound statement of the function declaration.
- Functions may be recursive.

10. *compound_stmt* → **{** *local_declarations* *statement_list* **}**

A compound statement consists of curly brackets surrounding a set of declarations and statements. A compound statement is executed by executing the statement sequence in the order provided. The local declarations have scope equal to the statement list of the compound statement and supersede any global declarations.

11. *local_declarations* → *local_declarations* *var_declaration* | **ε**
12. *statement_list* → *statement_list* *statement* | **ε**

Both, the local declarations and the statement list may be empty.

Figure 3: Semantics Part 2

13. *statement* → *assignment_stmt*
| *call_stmt*
| *compound _stmt*
| *selection _stmt*
| *iteration _stmt*
| *return _stmt*
| *input _stmt*
| *output _stmt*
14. *assignment_stmt* → *var* **=** *expression* **;**

An assignment statement assigns the value of an expression to a variable followed by semicolon. The expression to the right side of the assignment is evaluated, and its result is stored in the location that the variable represents.

15. *call_stmt* → *call* **;**

A call statement executes a function and ends with semicolon.

16. *selection _stmt* → **if** **(** *expression* **)** *statement*
| **if** **(** *expression* **)** *statement* **else** *statement*

The if-statement has the usual semantics. The expression is evaluated, a non-zero value causes execution of the first part of the statement, while a zero value causes the execution of the **else** part of statement if it exists.

17. *iteration _stmt* → **while** **(** *expression* **)** *statement*

The while-statement is the only iteration statement for C Minus. It is executed by repeatedly evaluating the expression and then executing the statement if the expression evaluates to non-zero. It ends its execution when the expression evaluates to zero.

18. *return _stmt* → **return** **;** | **return** *expression* **;**

A return statement may either return a value or not. Functions not declared as void MUST return a value. Functions declared as voids MUST not return a value. A return causes the transfer of control back to the caller (or termination of the program if it is inside **main**).

19. *input _stmt* → **input** *var* **;**
20. *output _stmt* → **output** *expression* **;**

Figure 4: Semantics Part 3

The output-statement is used to display into standard output (screen) either the value of a variable, the result of a logical or arithmetic expression, or a constant. The input-statement assigns the string obtained from the standard input into the location represented by variable.

The following restriction MUST be complied:

- Only integer values can be assigned to var trough the usage of the input-statement.
- If a non-integer value is obtained from standard input, the program MUST stop.

21. *var* → **ID** | **ID [** arithmetic_*expression* **]**

A variable is either a simple (integer) variable, or a subscripted array variable. The following restrictions MUST be complied:

- A negative subscript value MUST cause the program to stop.
- The upper bound of the subscript MUST be checked, the value of the subscript exceeds the upper bound, then the program MUST stop.

22. *expression* → arithmetic_*expression relop* arithmetic_*expression*
    | arithmetic_*expression*
23. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**

An expression is usually evaluated, and the result used on the behavior of the program. An expression consists of relational operators that do not associate (i.e., an un-parenthesized expression can only have one relational operator). The value of an expression is either the value of its arithmetic expression, if it contains no relational operators, or 1 if the relational operator evaluates to true; 0 if it evaluates to false.

24. *arithmetic_expression* → *arithmetic_expression addop term* | *term*
25. *addop* → **+** | **-**
26. *term* → *term mulop factor* | *factor*
27. *mulop* → ***** | **/**

Arithmetic expressions represent the typical associative and precedence of arithmetic operators. The / symbol represents integer division; hence, any reminder is truncated.

28. *factor* → **(** *arithmetic_expression* **)** | *var* | *call* | **NUM**

A factor is an *arithmetic_expression* enclosed in parentheses. It can also be a variable, which evaluates to the value that it holds. It can also be a call to a function, which evaluates to its the returned value. Finally, it can be a number, whose value is computed by the scanner and it is held in the corresponding Symbol Table.

Figure 5: Semantics Part 4

29. *call* → **ID (** *args* **)**
30. *args* → *args_list* | **ε**
31. *args_list* → *args_list* **,** *arithmetic_expression* | *arithmetic_expression*

A function call consists of an ID which represents the name of the function, followed by parentheses enclosing its arguments. Arguments are either empty or consist of a comma-separated list of *arithmetic_expression*, representing the values to be assigned as parameters during a call.

The following restrictions MUST be complied:

- Functions MUST be declared before they are called
- An array parameter in a *function declaration* MUST be matched with an expression consisting of a single identifier that represents an array variable.

Figure 6: Semantics Part 5

3 Design

While designing the parser some changes needed to be made to the grammar. The new grammar and the steps taken are as follow:

Grammar

1. declaration_list = declaration declaration_list’
2. declaration_list’ = declaration declaration_list’ | ε
3. declaration = var_declaration | fun_declaration
4. var_declaration = int ID var_declaration’
5. var_declaration’ = ; | [NUM] ;
6. type_specifier = int | void
7. fun_declaration = type_specifier ID (params) compound_stmt
8. params = param_list | void
9. param_list = param param_list’
10. param_list’ = , param param_list’ | ε
11. param = int ID param’
12. param’ = [] | ε
13. compound_stmt = { local_declarations statement_list }
14. local_declarations = var_declaration local_declarations | ε
15. statement_list = statement statement_list'
16. statement_list' = statement | ε
17. statement = var = expression ; | call ; | compound _stmt | if (expression) statement selection _stmt_else | while (expression) statement | return return_stmt’ | input var ; | output output_stmt’
18. selection _stmt_else = else statement | ε
19. return_stmt’ = ; | expression ;
20. output_stmt’ = expression ; | var ;
21. var = ID var’
22. var’ = [aritmetic_expression] | ε
23. expression = arithmetic_expression expression’
24. expression’ = relop arithmetic_expression | ε
25. relop = <= | < | > | >= | == | !=

26. arithmetic_expression = term arithmetic_expression’
27. arithmetic_expression’ = addop term arithmetic_expression’ | ε
28. addop = + | -
29. term = factor term’
30. term’ = mulop factor term’ | ε
31. mulop = * | /
32. factor = (arithmetic_expression) | var | call | NUM
33. call = ID (args)
34. args = args_list | ε
35. args_list = arithmetic_expression args_list’
36. args_list’ = , arithmetic_expression args_list’ | ε

Changes in the grammar

- The following rule was changed to eliminate the use of void in the variables and eliminate ambiguity

  var_declaration = type_specifier ID ; | type_specifier ID [NUM]

  it was changed for

  var_declaration = int ID var_declaration’

  var_declaration’ = ; | [NUM] ;
- The following rule was changed to eliminate left recursion

  declaration_list = declaration_list declaration | declaration

  it was changed for

  declaration_list = declaration declaration_list’

  declaration_list’ = declaration declaration_list’ | ε
- The following rule was changed to eliminate left recursion

  param_list = param_list , param | param

  it was changed for

  param_list = param param_list’

  param_list’ = , param param_list’ | ε
- The following rule was changed to eliminate left recursion

  local_declarations = local_declarations var_declaration | ε

  it was changed for

  local_declarations = var_declaration local_declarations | ε
- The following rule was changed to eliminate left recursion

statement_list = statement_list statement | ε

it was changed for

statement_list = statement_list’

statement_list’ = statement statement_list’ | ε

- The following rule was changed to eliminate left recursion

  param = type_specifier ID | type_specifier ID[ ]

  it was changed for

  param = int ID param’

  param’ = [] | ε

- The following rules were deleted because they were not needed
  - program = declaration_list
  - call_stmt = call;
  - iteration_stmt = while (expression) statement
  - input_stmt = input var ;
  - assignment_stmt = var = expression ;
  - return_stmt = return return_stmt’
  - output_stmt = output output_stmt’
- La siguiente regla fue cambiada para eliminar left factoring

  return_stmt = return ; | return expression;

  it was changed for

  return_stmt = return return_stmt’

  return_stmt' = ; | expression ;

- The following rule was changed so the output can handle expressions and variables

  output_stmt = output expression;

  it was changed for

  output_stmt = output output_stmt’

  output_stmt’ = expression ; | var ;

- The following rule was changed to eliminate left factoring

  var = ID | ID[arithmetic_expression]

  it was changed for

  var = ID var’

  var’ = [aritmetic_expression] | ε

- The following rule was changed to eliminate left factoring

  expression = arithmetic_expression relop arithmetic_expression | arithmetic_expression it was changed for

expression = arithmetic_expression expression’

expression’ = relop arithmetic_expression | ε

- The following rule was changed to eliminate left recursion

  arithmetic_expression = arithmetic_expression addop term | term

  it was changed for

  arithmetic_expression = term arithmetic_expression’

  arithmetic_expression’ = addop term arithmetic_expression’ | ε

- The following rule was changed to eliminate left recursion

  term = term mulop factor | factor

  it was changed for

  term = factor term’

  term’ = mulop factor term’ | ε

- The following rule was changed to eliminate left recursion

  args_list = args_list , arithmetic_expression | arithmetic_expression

  it was changed for

  args_list = arithmetic_expression args_list’

  args_list’ = , arithmetic_expression args_list’ | ε

The next step in the design process was to create the FIRST set to know the terminal symbols that appear in each rule. Also the FOLLOW set was calculated using the FIRST set data. The process and the final table are as follow:

First

Procedure:

- var_declaration’ = **;** | **[NUM] ;**
- type_specifier = **int | void**
- param’ = **[]** | ε
- relop = **<= | < | > | >= | == | !=**
- addop = **+ | -**
- mulop = ***** | /**

First(var_declaration’) = { ;, [}

First(type_specifier) = { int, void }

First(param’) = { [, ε }

First(relop) = { <=, <, >, >=, ==, != }

First(addop) = { +, - }

First(mulop) = { *, / }

Calculamos ahora por "grupos" los first:

- params = param_list | **void**
- param_list = param param_list'
- param_list' = **,** param param_list' | ε
- param = **int ID** param'

First(param) = First(**int ID** param') = { int }

First(param_list') = {,, ε }

First(param_list) = First(param) = { int }

First(params) = First(param_list) + void = { int, void }

- var_declaration = ***int ID*** *var_declaration'*

First(var_declaration) = { int }

- var = **ID** var'
- var' = [aritmetic_expression] | ε
- call = **ID (** args **)**

First(var) = { ID }

First(var') = { [. ε }

First(call) = { ID }

- expression = arithmetic_expression expression'
- expression' = relop arithmetic_expression | ε
- arithmetic_expression = term arithmetic_expression'
- arithmetic_expression' = addop term arithmetic_expression' | ε
- term = factor term'
- term' = mulop factor term' | ε
- factor = **(** arithmetic_expression **)** | var | call | **NUM**

First(arithmetic_expression') = { +, -, ε }

First(term') = { *, / , ε }

First(arithmetic_expression) = First(term) + First(arithmetic_expression') = { (, ID, NUM }

First(factor) = First(**(** arithmetic_expression **))** + First(var) + First(call) + NUM = { (, ID, NUM }

First(term) = First(factor) = { (, ID, NUM }

First(expression') = { <=, <, >, >=, ==, !=, ε }

First(expression) = { (, ID, NUM }

- return_stmt’ = **;** | expression **;**
- output_stmt’ = expression **;** | var **;**
- args = args_list | ε
- args_list = arithmetic_expression args_list’
- args_list’ = **,** arithmetic_expression args_list’ | ε

First(return_stmt’) = ; + First(expression) = { (, ID, NUM, ; }

First(args) = First(args_list) + ε = { (, ID, NUM, ε }

First(args_list) = First(arithmetic_expression) = { (, ID, NUM }

Fist(args_list’) = {,, ε }

First(output_stmt’) = First(expression) + First(var) = { ID, (, NUM }

- selection _stmt_else = else statement | ε

First(selection _stmt_else) = {else, ε }

- statement_list = statement statement_list'
- statement_list' = statement | ε
- statement = assignment_stmt | *call **;*** | compound _stmt | if (expression) statement selection _stmt_else | **while** **(** expression **)** statement | return | **input** var **;** | output
- compound_stmt = **{** local_declarations statement_list **}**

First(statement_list) = First(statement) = { ID, {, if, while, return, input, output }

First(statement_list’) = First(statement) + ε = { ID, {, if, while, return, input, output, ε }

First(statement) = First(var) + First(call) + First(compound _stmt) + if + while + return + input + output = { ID, {, if, while, return, input, output }

First(compound_stmt) = { { }

- declaration_list = declaration declaration_list’
- declaration_list’ = declaration declaration_list’ | ε
- declaration = var_declaration | fun_declaration
- fun_declaration = type_specifier **ID (** params **)** compound_stmt
- local_declarations = var_declaration local_declarations | ε

First(declaration_list) = First(declaration) = { int, void }

First(declaration_list’) = First(declaration) + ε = {int, void, ε }

Fist(declaration) = First(var_declaration) + First(fun_declaration) = { int, void }

First(fun_declaration) = First(type_specifier) = { int, void }

First(local_declarations) = First(var_declaration) + ε = { int, ε }

First and Follow Table

Non terminal	First	Follow
Declaration_list	{ int, void }	{ $ }
Declaration_list’	{int, void, ε }	{ $ }
Declaration	{ int, void }	{int, void, $ }
var_declaration	{ int }	{int, void, $, int, } }
var_declaration’	{ ;, [}	{int, void, $, int, } }
type_specifier	{ int, void }	{ ID }
Fun_declaration	{ int, void }	{int, void, $ }
Params	{ int, void }	{) }
param_list	{ int }	{) }
param_list’	{,, ε }	{) }
Param	{ int }	{,,) }
param’	{ [, ε }	{,,) }
compound_stmt	{ { }	{int, void, $ }
local_declarations	{ int, ε }	{ ID, {, if, while, return, input, output }
statement_list	{ ID, {, if, while, return, input, output }	{ } }
statement_list’	{ ID, {, if, while, return, input, output, ε }	{ } }
statement	{ ID, {, if, while, return, input, output }	{ ID, {, if, while, return, input, output, }, else }
selection _stmt_else	{else, ε }	{ ID, {, if, while, return, input, output, }, else }
return_stmt’	{ (, ID, NUM, ; }	{ ID, {, if, while, return, input, output, }, else }
output_stmt’	{ ID, (, NUM }	{ ID, {, if, while, return, input, output, }, else }

var	{ ID }	{ *, / , +, -, <=, <, >, >=, ==, !=, ;,), ,, }
var’	{ [. ε }	{ *, / , +, -, <=, <, >, >=, ==, !=, ;,), ,, }
expression	{ (, ID, NUM }	{ ;,) }
expression’	{ <=, <, >, >=, ==, !=, ε }	{ ;,) }
Relop	{ <=, <, >, >=, ==, != }	{ (, ID, NUM }
arithmetic_expression	{ (, ID, NUM }	{ <=, <, >, >=, ==, !=, ;,), ,, }
arithmetic_expression’	{ +, -, ε }	{ <=, <, >, >=, ==, !=, ;,), , }
addop	{ +, - }	{ (, ID, NUM }
term	{ (, ID, NUM }	{ +, -, <=, <, >, >=, ==, !=, ;,), ,, }
term’	{ *, / , ε }	{ +, -, <=, <, >, >=, ==, !=, ;,), ,, }
mulop	{ *, / }	{ (, ID, NUM }
factor	{ (, ID, NUM }	{ *, / , +, -, <=, <, >, >=, ==, !=, ;,), ,, }
call	{ ID }	{ *, / , +, -, <=, <, >, >=, ==, !=, ;,), ,, }
args	{ (, ID, NUM, ε }	{) }
args_list	{ (, ID, NUM }	{) }
args_list’	{,, ε }	{) }

First+ is the final step to be able to implement the grammar. This set needs to take into account the FIRST set and the FOLLOW set.

First+

First+(declaration_list = declaration declaration_list’) = { int, void }

First+(declaration_list’ = declaration declaration_list’) = { int, void }

First+(declaration_list’ = ε) = { $, ε }

First+(declaration = var_declaration) = { int }

First+(declaration = fun_declaration) = { int, void }

First+(var_declaration = int ID var_declaration’) = { int }

First+(var_declaration’ = ;) = { ; }

First+(var_declaration’ = [NUM] ;) = { [}

First+(type_specifier = void) = { int }

First+(type_specifier = void) = { void }

First+(fun_declaration = type_specifier ID (params) compound_stmt) = { int, void }

First+(params = param_list) = { int }

First+(params = void) = { void }

First+(param_list = param param_list’) = { int }

First+(param_list’ = , param param_list’) = { , }

First+(param_list’ = ε) = {), ε }

First+(param = int ID param’) = { int }

First+(param’ = []) = { [}

First+(param’ = ε) = {,,), ε }

First+(compound_stmt = { local_declarations statement_list }) = { } }

First+(local_declarations = var_declaration local_declarations) = { int }

First+(local_declarations = ε) = { ID, {, if, while, return, input, output, ε }

First+(statement_list = statement statement_list') = { ID, {, if, while, return, input, output }

First+(statement_list' = statement) = { ID, {, if, while, return, input, output }

First+(statement_list' = ε) = { }, ε }

First+(statement = var = expression ;) = { ID }

First+(statement = call ;) = { ID }

First+(statement = compound _stmt) = { { }

First+(statement = if (expression) statement selection _stmt_else) = { if }

First+(statement = while (expression) statement) = { while }

First+(statement = return return_stmt’) = { return }

First+(statement = input var ;) = { input }

First+(statement = output output_stmt’) = { output }

First+(selection _stmt_else = else statement) = else

First+(selection _stmt_else = ε) = { ID, {, if, while, return, input, output, }, else, ε }

First+(return_stmt’ = ;) = { ; }

First+(return_stmt’ = expression ;) = { (, ID, NUM }

First+(output_stmt’ = expression ;) = { (, ID, NUM }

First+(output_stmt’ = var ;) = { ID }

First+(var = ID var’) = { ID }

First+(var’ = [aritmetic_expression]) = { [}

First+(var’ = ε) = { *, / , +, -, <=, <, >, >=, ==, !=, ;,), ,, ε }

First+(expression = arithmetic_expression expression’) = { (, ID, NUM }

First+(expression’ = relop arithmetic_expression) = { <=, <, >, >=, ==, != }

First+(expression’ = ε) = { ;,), ε }

First+(relop = <=) = { <= }

First+(relop = <) = { < }

First+(relop = >) = { > }

First+(relop = >=) = { >= }

First+(relop = ==) = { == }

First+(relop = !=) = { != }

First+(arithmetic_expression = term arithmetic_expression’) = { (, ID, NUM }

First+(arithmetic_expression’ = addop term arithmetic_expression’) = { +, - }

First+(arithmetic_expression’ = ε) = { <=, <, >, >=, ==, !=, ;,), ,, ε }

First(addop = +) = { + }

First(addop = -) = { - }

First+(term = factor term’) = { (, ID, NUM }

First+(term’ = mulop factor term’) = { *, / }

First+(term’ = ε) = { +, -, <=, <, >, >=, ==, !=, ;,), ,, ε }

First+(mulop = *) = { * }

First+(mulop = /) = { / }

First+(factor = (arithmetic_expression)) = { (}

First+(factor = var) = { ID }

First+(factor = call) = { ID }

First+(factor = NUM) = { NUM }

First+(call = ID (args)) = { ID }

First+(args = args_list) = { (, ID, NUM }

First+(args = ε) = {), ε }

First+(args_list = arithmetic_expression args_list’) = { (, ID, NUM }

First+(args_list’ = , arithmetic_expression args_list’) = { , } First+(args_list' = ε) = {), ε }

For the design, a class diagram was used to check the different methods the various components were going to have. On the syntax analyzer there are missing the production methods because of the extensive quantity of methods.

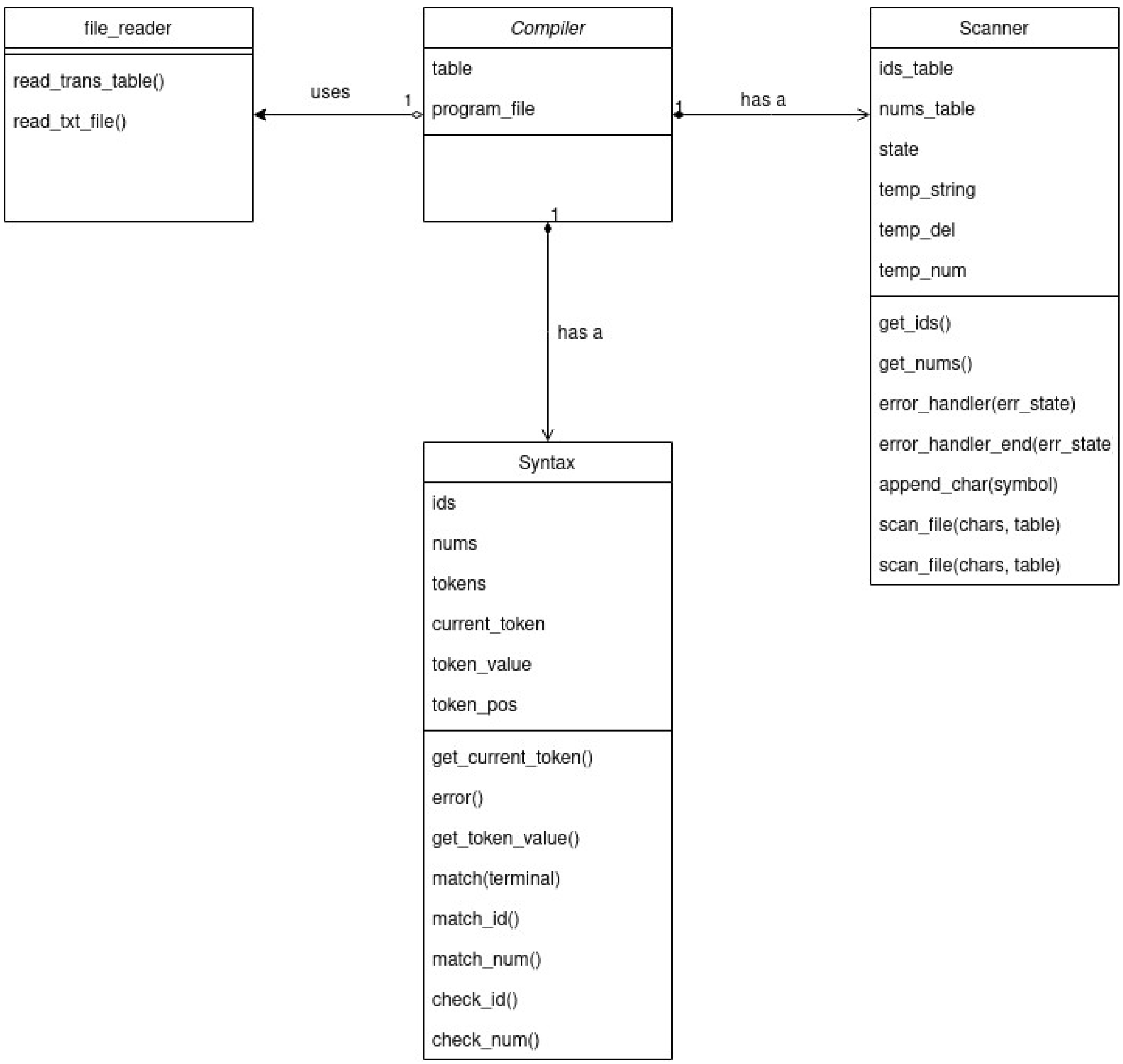

Figure 7: Compiler Class Diagram

This time the pseudo code used for the creation of the syntax analyzer was the one provided by Dr. Castello on one of his lectures.

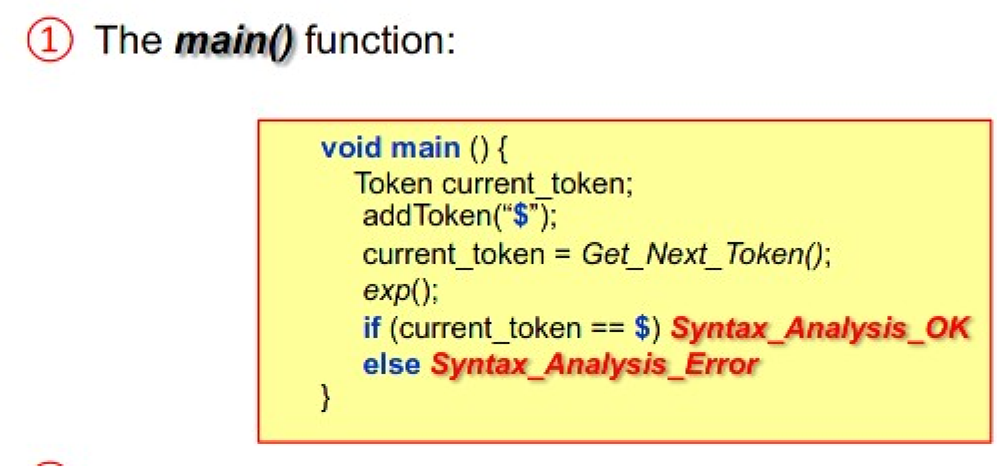

Figure 8: Dr. Castello's main() pseudo code

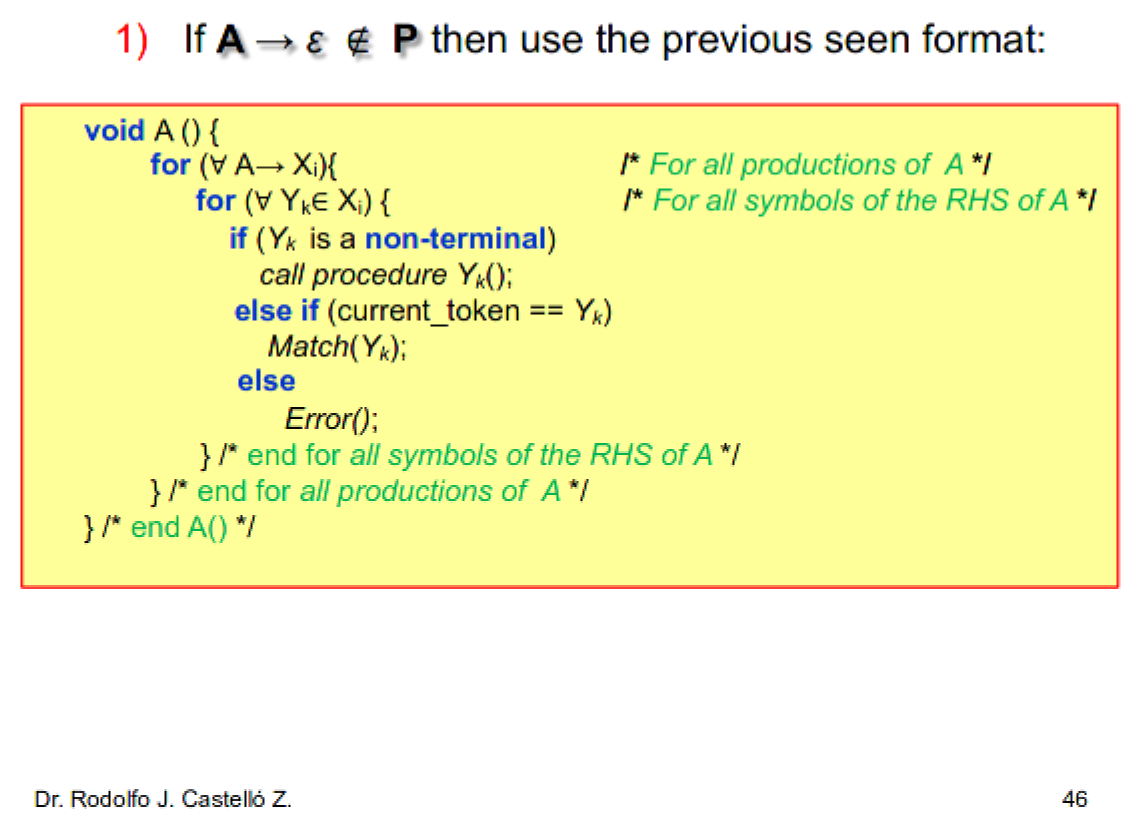

Figure 9: Dr. Castello's non empty function pseudo code

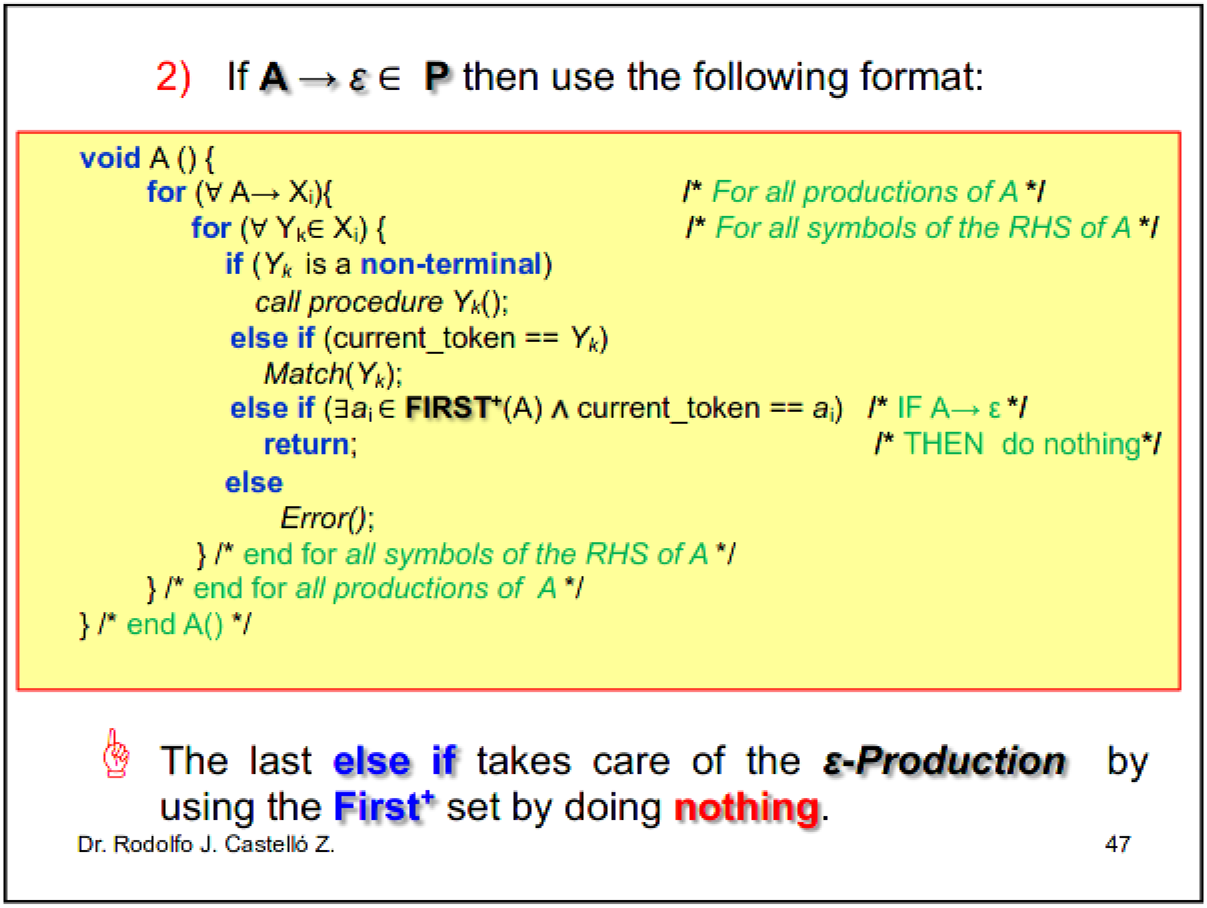

Figure 10: Dr. Castello's empty function pseudo code

4 Implementation

4.1 Code

Parser Implementation Code

```
class Parser:
def __init__(self, ids, nums, tokens):
self.ids = ids
self.nums = nums
self.tokens = tokens
self.current_token = tokens[0]
self.token_value = ""
self.token_pos = 0
self.res_list = ["err", "else", "if", "int", "return",
"void", "while", "input", "output"]
self.del_list = [
"+", "-", "*", "/", "<", "<=", ">", ">=", "==", "!=", "=", ";",
",", "(", ")", "[", "]", "{", "}"
]
self.tokens.append(["del", "$"])

def get_current_token(self):
return self.current_token

def error(self):
print("IMPLEMENTAR ERROR BIEN")
exit()

def get_token_value(self):
value = ""
if (self.current_token[0] == "res"):
value = self.res_list[self.current_token[1]]
elif (self.current_token[0] == "del"):
value = self.del_list[self.current_token[1]]
elif (self.current_token[0] == "id"):
```

```
value = self.ids[self.current_token[1]]
elif (self.current_token[0] == "nums"):
value = self.nums[self.current_token[1]]
else:
print("UNRECOGNIZED TOKEN TYPE")
exit()
return value

def match(self, terminal):
if (self.token_value == terminal):
self.token_pos += 1
self.current_token = self.tokens[0]
self.token_value = self.get_token_value()
else:
self.error()

def match_id(self):
if (self.current_token[0] == "id"):
self.token_pos += 1
self.current_token = self.tokens[0]
self.token_value = self.get_token_value()
else:
self.error()
def match_num(self):
if (self.current_token[0] == "num"):
self.token_pos += 1
self.current_token = self.tokens[0]
self.token_value = self.get_token_value()
else:
self.error()

def check_id(self):
if (self.current_token[0] == "id"):
return True
else:
return False

def check_num(self):
if (self.current_token[0] == "num"):
```

```
return True
else:
return False
def declaration_list(self):
self.token_value = self.get_token_value()
self.declaration()
self.declaration_list_prime()

def declaration_list_prime(self):
if (self.token_value == "int" or self.token_value == "void"):
self.declaration()
self.declaration_list_prime()
elif(self.token_value == "$"):
return
else:
self.error()

def declaration(self):
if (self.token_value == "int"):
self.var_declaration()
elif (self.token_value == "void"):
self.fun_declaration()
else:
self.error()
def var_declaration(self):
self.match("int")
self.match_id()
if (self.token_value == "("):
self.fun_declaration()
return
self.var_declaration_prime()
def var_declaration_prime(self):
if (self.token_value == ";"):
self.match(";")
elif (self.token_value == "["):
self.match("[")
self.match_num()
self.match("]")
self.match(";")
```

```
else:
self.error()
def fun_declaration(self):
if (self.token_value == "("):
self.match("(")
self.params()
self.match(")")
self.compound_stmt()
elif (self.token_value == "void"):
self.match("void")
self.match_id()
self.match("(")
self.params()
self.match(")")
self.compound_stmt()
else:
self.error()
def params(self):
if (self.token_value == "int"):
self.param_list()
elif (self.token_value == "void"):
self.match("void")
else:
self.error()
def param_list(self):
self.param()
self.param_list_prime()

def param_list_prime(self):
if (self.token_value == ","):
self.match(",")
self.param()
self.param_list_prime()
elif (self.token_value == ")"):
return
else:
self.error()

def param(self):
```

```
self.match("int")
self.match_id()
self.param_prime()

def param_prime(self):
if (self.token_value == "["):
self.match("[")
self.match("]")
elif (self.token_value == "," or self.token_value == ")"):
return
else:
self.error()
def compound_stmt(self):
self.match("{")
self.local_declarations()
self.statement_list()
self.match("}")
def local_declarations(self):
if (self.token_value == "int"):
self.var_declaration()
self.local_declarations()
elif (self.check_id() or self.token_value == "{" or
self.token_value == "if" or self.token_value == "while"
or self.token_value == "return" or self.token_value == "input" or
self.token_value == "output" ):
return
else:
self.error()

def statement_list(self):
self.statement()
self.statement_list_prime()
def statement_list_prime(self):
if (self.check_id() or self.token_value == "{" or self.token_value
== "if" or self.token_value == "while"
or self.token_value == "return" or self.token_value == "input" or
self.token_value == "output" ):
self.statement()
elif (self.token_value == "}"):
```

```
return
else:
self.error()
def statement(self):
if (self.check_id()):
self.var()
self.match("=")
self.expression()
self.match(";")
elif (self.token_value == "{"):
self.compound_stmt()
elif (self.token_value == "if"):
self.match("if")
self.match("(")
self.expression()
self.match(")")
self.statement()
self.selection_stmt_else()
elif (self.token_value == "while"):
self.match("while")
self.match("(")
self.expression()
self.match(")")
self.statement()
elif (self.token_value == "return"):
self.match("return")
self.return_stmt_prime()
elif(self.token_value == "input"):
self.match("input")
self.var()
self.match(";")
elif(self.token_value == "output"):
self.match("output")
self.output_stmt_prime()
else:
self.error()
def selection_stmt_else(self):
if(self.token_value == "else"):
self.match("else")
```

```
self.statement()
elif (self.check_id() or self.token_value == "{" or
self.token_value == "if" or self.token_value == "while"
or self.token_value == "return" or self.token_value == "input" or
self.token_value == "output"
or self.token_value == "}" or self.token_value == "else" ):
return
else:
self.error()
def return_stmt_prime(self):
if self.token_value == ";":
self.match(";")
elif (self.token_value == "(" or self.check_id() or
self.check_num()):
self.expression()
self.match(";")
else:
self.error()
def output_stmt_prime(self):
if self.check_id():
self.var()
self.match(";")
elif (self.token_value == "(" or self.check_id() or
self.check_num()):
self.expression()
self.match(";")
else:
self.error()
def var(self):
self.match_id()
if(self.token_value == "("):
#call
self.match("(")
self.args()
self.match(")")
return
self.var_prime()

def var_prime(self):
```

```
if(self.token_value == "["):
self.match("[")
self.arithmetic_expression()
self.match("]")
elif(self.token_value == "*" or self.token_value == "/" or
self.token_value == "+"
or self.token_value == "-" or self.token_value == "<=" or
self.token_value == "<"
or self.token_value == ">" or self.token_value == ">=" or
self.token_value == "=="
or self.token_value == "!=" or self.token_value == ";" or
self.token_value == ")"
or self.token_value == ","):
return
else:
self.error()
def expression(self):
self.arithmetic_expression()
self.expression_prime()
def expression_prime(self):
if(self.token_value == "<=" or self.token_value == "<"
or self.token_value == ">" or self.token_value == ">=" or
self.token_value == "=="
or self.token_value == "!="):
self.relop()
self.arithmetic_expression()
elif(self.token_value == ";" or self.token_value == ")"):
return
else:
self.error()
def relop(self):
if(self.token_value == "<="):
self.match("<=")
elif(self.token_value == "<"):
self.match("<")
elif(self.token_value == ">"):
self.match(">")
elif(self.token_value == ">="):
self.match(">=")
```

```
elif(self.token_value == "=="):
self.match("==")
elif(self.token_value == "!="):
self.match("!=")
else:
self.error()
def arithmetic_expression(self):
self.term()
self.arithmetic_expression_prime()

def arithmetic_expression_prime(self):
if(self.token_value == "+" or self.token_value == "-"):
self.addop()
self.term()
self.arithmetic_expression_prime()
elif(self.token_value == "<=" or self.token_value == "<"
or self.token_value == ">" or self.token_value == ">=" or
self.token_value == "=="
or self.token_value == "!=" or self.token_value == ";" or
self.token_value == ")"
or self.token_value == ","):
return
else:
self.error()
def addop(self):
if(self.token_value == "+"):
self.match("+")
elif(self.token_value == "-"):
self.match("-")
else:
self.error()
def term(self):
self.factor()
self.term_prime()
def term_prime(self):
if(self.token_value == "*" or self.token_value == "/"):
self.mulop()
self.factor()
self.term_prime()
```

```
elif(self.token_value == "+"
or self.token_value == "-" or self.token_value == "<=" or
self.token_value == "<"
or self.token_value == ">" or self.token_value == ">=" or
self.token_value == "=="
or self.token_value == "!=" or self.token_value == ";" or
self.token_value == ")"
or self.token_value == ","):
return
else:
self.error()
def mulop(self):
if(self.token_value == "*"):
self.match("*")
elif(self.token_value == "/"):
self.match("/")
else:
self.error()
def factor(self):
if(self.token_value == "("):
self.match("(")
self.arithmetic_expression()
self.match(")")
elif(self.check_id()):
self.var()
elif(self.check_num()):
self.match_num()
else:
self.error()

def args(self):
if(self.token_value == "(" or self.check_id() or self.check_num()):
self.args_list()
elif(self.token_value == ")"):
return
else:
self.error()

def args_list(self):
```

```
self.arithmetic_expression()
self.args_list_prime()

def args_list_prime(self):
if(self.token_value == ","):
self.match(",")
self.arithmetic_expression()
self.args_list_prime()
elif(self.token_value == ")"):
return
else:
self.error()
```

Tests

The tests used for the compiler are shown in Figure 11. This tests where manually checked after each new functionality added. The downside Is the great amount of time needed to do the tests but manually checking them each times gives certainty of the results

ID	Desc	Input	Output
1	The parser transitions into the correct “rule” function based on the grammar	ids, num, tokens	next function
2	The parser recognizes an error. It displays an error message	ids, num, tokens	error message Exit program
3	The parser recognizes the $ symbol. It displays successful message with the token list	ids, num, tokens	List with the correct token characters and a message saying the syntax analysis was a success

Figure 11: Tests

5 References

1. R. Castelló, Class Lecture, Topic: “Chapter 3 – Sintax Analysis Part I.”TC3048, School of Engineering and Science, ITESM, Chihuahua, Chih, June, 2020.
2. R. Castelló, Class Lecture, Topic: “Chapter 4 – Sintax Analysis Part II.”TC3048, School of Engineering and Science, ITESM, Chihuahua, Chih, June, 2020.