Syntactic Analyzer for the C Minus Language

Víctor Emmanuel Guerra Aguado - A01568075

Summary

The following document will report upon the development of a syntactic analyzer for the proposed C Minus programming language in the Python programming language, including in the complete phases of **Analysis**, **Design**, **Implementation**, and **Testing.** Each phase will contain its appropriate documentation and implementation.

Context & Notation

Context Free Grammars (CFGs)

A grammar is a set of rules and examples that deals with the syntax and word structure of a language via a set of Productions. In the context of compiler design and programming languages, it provides the rules of structure for a given programming language.

A Grammar is said to be Context-Free if all its productions are of the form $\alpha \rightarrow \beta$ where alpha consists of a single variable (non-terminal symbol) and beta is a string of language symbols (terminal symbols or tokens) and non-terminals. [1]

The notation used to specify grammars is known as the Backus Normal Form. A grammar is defined by a 4-tuple set (V, T, P, S) where V is the set of non-terminal symbols, T is the set of terminal symbols, P is the set of productions, and S is the start symbol. The set of terminal symbols is the list of Tokens produced by the lexical analyzer and are represented by:

- Lower case letters: a, b, c, etc.
- Operator Symbols: +, -, =, etc.
- Punctuation Symbols: ,, ., (, {, etc.
- Digits: 0 … 9
- Keywords: int, main, if, etc.

The set of non-terminals, sometimes called syntactic variables, represents a set of strings; the set of strings defines the language generated by the grammar. Non-terminal symbols are represented by:

- Upper case letters: A, B, C, etc.
- Letter S is the start symbol.
- Italic lower case string: *exp, statement,* etc.

One of the non-terminal symbols is designated as the start symbol, and the set of strings it specifies is the language generated by the grammar.
The set of productions of a grammar specifies the manner in which terminals and non-terminals can be combined to form strings. Each productions consists of [1]:

- A non-terminal known as the head or left-side of the production
- The arrow symbol :"→"
- A body or right-side, which consists of zero or more terminals and non-terminals

Issues with CFG(Ambiguity, Left Recursion, Left Factorization, Simplification of Grammars)

There are some aspects of a CFG that must be dealt with in order to be properly used for a programming language and the corresponding syntax analysis phase of its compiler:

- Ambiguity

  Parse Trees uniquely express the structure of the syntax as leftmost or rightmost derivations. Unfortunately it's possible for a grammar to permit a string to have more than one parse tree, if it does the grammar is called ambiguous. This represents a serious problem for a parser since the grammar does not specify precisely the syntactic structure of a program.

There is NO algorithm that converts an ambiguous grammar to an equivalent unambiguous one in a straightforward manner. The basic method for dealing with ambiguity is to change the grammar into a form that forces the construction of the correct parse tree, thus removing the ambiguity.

- Left Recursion

A grammar is left recursive if it has a non-terminal A such that there is a derivation A => Aα for some string α. If this kind of grammar is used in Top-Down parsers, the parser may go into an infinite loop. Top-Down parsers cannot handle left recursive grammars. Therefore, all left recursive grammars must be converted into an equivalent non-left recursive grammar. Left Recursion may appear in a single step of the derivation (immediate left recursion), or it may appear in more than one step of the derivation.

A grammar G is left-recursive if there is a rule of the form: A → Aα | β where β does not start with A, i.e., the leftmost terminal on the right hand side is the same as the non-terminal on the left hand side. To eliminate immediate left recursion rewrite the grammar as:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

This is an equivalent grammar G', which is free of left recursion. In General, for all productions of the form:

$$A \rightarrow A\alpha 1 \mid A\alpha 2 \mid \ldots \mid A\alpha m \mid \beta 1 \mid \beta 2 \mid \ldots \mid \beta n$$

where β1, β2, ... βn do not start with A, Eliminate immediate left recursion by:

$$A \rightarrow \beta 1A' \mid \beta 2A' \mid \ldots \mid \beta nA'$$
$$A' \rightarrow \alpha 1A' \mid \alpha 2A' \mid \ldots \mid \alpha mA' \mid \varepsilon$$

A grammar can be free of immediate left recursion, but it still can be left recursive. Eliminate complete left recursion:

1) Arrange non-terminals in some order: X1 ... Xn

2) Apply the following procedure:

```
for i=1 to n do{
        for j=1 to i-1 do{
        replace each production Xi → Xj β by
```

$X_i \to \alpha_1 \beta \mid \alpha_2 \beta \mid ... \mid \alpha_k \beta$

where

$X_j \to \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_k$

}

eliminate immediate left recursion among Xi.

}

- Left Factoring

Sometimes, the grammar may have common a prefix in many productions like $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_n$ where α is a common prefix. While processing α it can not be decided whether to expand A by $\alpha\beta_1$ or by $\alpha\beta_n$. So this needs Backtracking. Eliminating Left Factoring is a grammar transformation technique that is useful to produce grammars that are suitable for predictive or top-down parsing.

To Eliminate left factoring:

For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix:

$A \to \alpha \beta_1 \mid \alpha \beta_2 \mid ... \mid \alpha \beta_n \mid \gamma_1 \mid \gamma_2 \mid ... \mid \gamma_m$

where $\gamma_1, \gamma_2, ... \gamma_n$ do not begin with α convert it into:

$A \to \alpha A' \mid \gamma_1 \mid \gamma_2 \mid ... \mid \gamma_m$

$A' \to \beta_1 \mid \beta_2 \mid ... \mid \beta_n$

- Simplification of grammars

All Grammars are not always optimized. A grammar may contain extra or unnecessary symbols, which will increase the length of the grammar. Simplification of the grammar generally includes the following steps:

1) Elimination of useless symbols.
2) Elimination of ε-productions.
3) Elimination of unit productions.

1) A symbol is useless if it cannot derive a terminal or if it is not reachable from the start symbol. To eliminate such a symbol: eliminate all productions in which it appears on either side.

2) If a CFL contains the word ε, then the CFG must have an ε-production. However, if a CFG has an ε-production, then the CFL does not necessarily contain the word ε.

In a given CFG, a non-terminal X is nullable if:

1. There is a production $X \to \varepsilon$, or
2. There is a derivation that starts at X and leads to ε, i.e., $X \Rightarrow \varepsilon$

To eliminate ε-Productions:

1. Construct the set of all Nullable Variables Vn
2. For each production A → B, if B is a Nullable Variable, replace it by ε and add it with all possible combinations on the RHS.
3. Take out the production B → ε.

3) A Unit Production is a production of the form A → B. To eliminate Unit Productions:

1. For each pair of non-terminals A and B such that:
    a. There is a production A→B and
    b. The productions of B are:
       B → s1 | s2 | s1 | ... | sn , where at least one si ∈ (T + V)*, i.e., si is a string of terminals and non-terminals.
2. Create the new productions:
    a. A → s1 | s2 | s3 | … | sn
3. Remove the unit production A → B.
4. Repeat for all pairs A and B.

Parsing Algorithms & Top-Down Parsers

Parsing algorithms can be classified into three types: Universal Parsers, which perform the syntax analysis with any grammar, however, this kind of parser is inefficient and therefore it is not used in commercial compilers; Top-down Parsers (TDP), which begin from the start symbol of a grammar, and applies production rules for each non-terminal until it gets a string formed only by terminals; and Bottom-Up Parsers (BUP), in general more powerful than Top-Down parsers, but extremely difficult to write and develop by hand, and are therefore only used in most of the automatic parser generators; bottom-up parsers have none of the restrictions of a top-down parser previously discussed with the exception of an unambiguous grammar; Bottom-Up parsing constructs a parse tree from an input string beginning at the leaves and working towards the root, or start symbol of the grammar.[2]

Top-Down Parsers can be subdivided into Brute Force Parsers and Predictive Parsers. The Brute Force Technique uses full Backtracking, meaning, whenever a non-terminal has more than one alternative, it always expands the first option the first time it encounters it, if the parser does not successfully finish the analysis, the parser backtracks all the way to the first execution of the first alternative, and now tries with the second alternative, the parser repeats this until there is a match with the input string, or all combinations are verified.

Predictive Parsers are capable of deciding the correct alternative in order to expand a non-terminal; the prediction relies on information about what first symbols can be generated from a production or rule: if the first symbol of a production is a non-terminal, then the non-terminal has to be expanded until a set of terminal symbols are obtained; the grammar upon which a predictive parser is designed has three restrictions:

1. It has to be unambiguous.
2. It has to be free of left recursion.
3. It has to be free of left factors.

Predictive Parsers can be divided into two types:

1. Recursive Descent Parser
2. Non-recursive Descent (or LL(1)) Parser

Recursive Descent Parsers are the simplest type of predictive parsers. A Predictive Recursive Descent parser is constructed by writing recursive procedures for each non-terminal; it is a Top-Down process in which the parser attempts to verify whether the syntax of the input string is correct as it is read from left to right, and it always expands a non-terminal by its right hand side of the rule until it gets a string of terminals. The parser reads tokens from the scanner and matches them with terminals from the grammar that describes the syntax of the input, token by token from left to right.

A Non-Recursive Predictive parser can be built by explicitly maintaining a stack instead of implicitly using recursive calls, and a parsing table to dictate the parsing decisions. The LL(1) parser consists of the following elements:

1. The input buffer that contains the string to be parsed, i.e. The scanners' output
2. The stack that contains a sequence of grammar symbols
3. The parsing table M[X, a] that dictates the parsing decisions
4. The parsing program that takes the symbol X at the top of the stack together with "a" the current input symbol from the scanner. If X is a non-terminal then an X-production is chosen according to M[X, a]. If X is a token then it checks for a match between X and a.

The Parsing Algorithm Used

For the purpose of the C Minus language compiler, the LL(1) parsing algorithm was selected. In order to be able to implement the algorithm, the following steps will be necessary:

1. Elimination of Ambiguity.
2. Elimination of Left Recursion.
3. Elimination of Left Factors.
4. Elimination of Useless Symbols.
5. Elimination of ε-productions.
6. Elimination of Unit Productions.
7. Calculation of the FIRST Set.
8. Calculation of the FOLLOW Set.
9. Calculation of the FIRST+ Set.
10. Construction of the Parsing Table.
11. Implementation of the LL(1) Stack.
12. Implementation of the LL(1) parsing program.

Programming Language

The chosen programming language for the compiler, lexical analyzer, and syntactic analyzer is Python. It was chosen for its simplicity and straightforwardness regarding the management of data structures, memory, and data types; its flexibility would make it possible to take some load from the implementation phase that could be used during other phases.

Analysis

Requirements

1. The Parser must correctly recognize the C minus grammar.
2. The Parser must recognize and report the following Syntactical Errors:
    a. The program does not contain at least one declaration.
    b. void type used for a variable.
    c. Expected a (
    d. Expected a {
    e. Expected a [
    f. Expected a ;
    g. Expected a)
    h. Expected a }

i. Expected a]
j. Expected an identifier
k. Expected a number constant
l. Expected an expression

3. The Parser must recognize and report the following Semantic Errors:
   a. The programs' last declaration is not main.
   b. A variable has been used before declaration.
   c. A function has been used before declaration.
4. The Parser must modify the existing Symbol Tables to add the following columns:
   a. isVar (is a variable)
   b. isFun (is a function)
   c. dataType (data type that it is associated to)
   d. noArgs (if it is a function, how many arguments does it receive)
   e. isGlobal (if it is a global declaration)
   f. isLocal (if it is a local declaration)
5. The Error Reports must provide the number of the line in which the error was encountered.

The Definition of the final Grammar

The Parser will be made to follow the Top-Down LL(1) [2] algorithm, which requires said grammar to comply with several requirements before its implementation into code such as no Left Recursion, no Left Factors, and is unambiguous, while the simplification of the grammar is not necessary for a functional Top-Down parser, some simplification will be done in order to make the process of calculating the FIRST, FOLLOW, and FIRST+ Sets less arduous.

Firstly, the provided grammar for the C minus language contains several inconsistencies with the provided semantics of the language; it must be reworked to make it fit its description:

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. Original Provided Grammar

Making the grammar fit its description

1) *The grammar cannot identify whether an identifier for a function or variable is being used before its declaration.* This problem cannot be solved by a context free grammar, and will be solved later in the coding step, by making sure the first time an identifier is found, it is during its declaration.

2) *The grammar does not specify that there is always a final function declaration with the identifier 'main'.* This problem can be partially solved by changing the grammar to have at least one declaration; later, during the coding of the parser, code can be added to make sure that said declaration uses the identifier 'main' and is, in fact, a function.
3) *The grammar allows for a variable to be assigned a* ***void*** *type.* This issue can be solved by creating separate type specifier variables for variables and functions.
4) *The grammar allows for parameters of type* ***void*** *to be assigned to a function.* This can be solved by using only the type specifier variable for variables in the productions of the non-terminal *param.*

Any other remaining specification cannot be verified during the syntax analysis, and correspond to further steps in the compilation process. The resulting grammar after the previous modifications is as follows, the corrections made in the previous step highlighted in green:

	Produccion				
No. P	LHS	→	RHS	RHS	RHS
1	program	→	declaration_list **void ID (void)** compound_stmt	**void ID (void)** compound_stmt	
2	declaration_list	→	declaration_list declaration	**epsilon**	
3	declaration	→	var_declaration	fun_declaration	
4	var_declaration	→	var_type_specifier **ID ;**	var_type_specifier **ID [NUM];**	
5	func_type_specifier	→	**int**	**void**	
6	var_type_specifier	→	**int**		
7	fun_declaration	→	func_type_specifier **ID (** params **)** compound_stmt		
8	params	→	param_list	**void**	
9	param_list	→	param_list **,** param	param	
10	param	→	var_type_specifier **ID**	var_type_specifier **ID [ ]**	
11	compound_stmt	→	**{** local_declarations statement_list **}**		
12	local_declarations	→	local_declarations var_declaration	**epsilon**	
13	statement_list	→	statement_list statement	**epsilon**	
14	statement	→	assignment_stmt	call_stmt	compound_stmt
			selection_stmt	iteration_stmt	return_stmt
			input_stmt	output_stmt	
15	assignment_stmt	→	var **=** expression **;**		
16	call_stmt	→	call **;**		
17	selection_stmt	→	**if (** expression **)** statement	**if (** expression **)** statement **else** statement	
18	iteration_stmt	→	**while (** expression **)** statement		
19	return_stmt	→	**return ;**	**return** exression **;**	
20	input_stmt	→	**input** var **;**		
21	output_stmt	→	**output** expression ;		
22	var	→	**ID**	**ID [** arithmetic_expression **]**	
23	expression	→	arithmetic_expression relop arithmetic_expression	arithmetic_expression	
24	relop	→	**<=**	**<**	**>**
			>=	**==**	**!=**
25	arithmetic_expression	→	arithmetic_expression addop term	term	
26	addop	→	**+**	**-**	
27	term	→	term mulop factor	factor	
28	mulop	→	*****	**/**	
29	factor	→	**(** arithmetic_expression **)**	var	call
			NUM		
30	call	→	**ID (** args **)**		
31	args	→	args_list	**epsilon**	
32	args_list	→	args_list **,** arithmetic_expression	arithmetic_expression	

Figure 2. The grammar after corrections for semantic compliance.

Elimination of Left Recursion

Now that the grammar appropriately represents the language described by the semantics, it must be refactored to, firstly, eliminate the left-recursion present, highlighted in red in Figure 2:

1) In the productions of non-terminal **No. 2** ‘*declaration_list*’:

   declaration_list →declaration_list declaration

   So it is replaced by the productions:

   declaration_list → declaration declaration_list’

   declaration_list’ →declaration declaration_list’ | *ε*

2) In the productions of non-terminal **No. 9** ‘*param_list’*:

   param_list → param_list, param | *param*

   So it is replaced by the productions:

   param_list → param param_list’

param_list → , param param_list' | ε

3) In the productions of non-terminal **No. 12** '*local_declarations':*

   local_declarations → local_declarations var_declaration | ε

   So they are replaced by the productions:

   local_declarations → local_declarations'

   local_declarations' → var_declarations local_declarations' | ε

4) In the productions of non-terminal **No. 13** 'statement_list':

   statement_list → statement_list statement | ε

   So they are replaced by the productions:

   statement_list → statement_list'

   statement_list' → statement statement_list' | ε

5) In the productions of non-terminal **No. 25** '*arithmetic_expression'*:

   arithmetic_expression → arithmetic_expression addop term | term

   So they are replaced by the productions:

   arithmetic_expression → term arithmetic_expression'

   arithmetic_expression' → addop term arithmetic_expression' | ε

6) In the productions of non-terminal **No. 27** '*term*':

   term → term mulop factor | factor

   So they are replaced by the productions:

   term → factor term'

   term' → mulop factor term | ε

7) In the productions of non-terminal **No. 32** '*args_list*':

   args_list → args_list , arithmetic_expression | arithmetic_expression

   So they are replaced by the productions:

   args_list → arithmetic_expression args_list'

   args_list' → , arithmetic_expression args_list' | ε

With the previous modifications the grammar results as follows:

	Produccion				
No. P	LHS	→	RHS	RHS	RHS
1	program	→	declaration_list **void ID (void)** compound_stmt	**void ID (void)** compound_stmt	
2	declaration_list	→	declaration declaration_list	**epsilon**	
4	declaration	→	var_declaration	fun_declaration	
5	var_declaration	→	var_type_specifier **ID ;**	var_type_specifier **ID [NUM];**	
6	func_type_specifier	→	**int**	**void**	
7	var_type_specifier	→	**int**		
8	fun_declaration	→	func_type_specifier **ID (** params **)** compound_stmt		
9	params	→	param_list	**void**	
10	param_list	→	param param_list'		
11	param_list'	→	**,** param param_list'	**epsilon**	
12	param	→	var_type_specifier **ID**	var_type_specifier **ID [ ]**	
13	compound_stmt	→	**{** local_declarations statement_list **}**		
14	local_declarations	→	local_declarations'		
15	local_declarations'		var_declaration local_declarations'	**epsilon**	
16	statement_list	→	statement_list'		
17	statement_list'	→	statement statement_list'	**epsilon**	
18	statement	→	assignment_stmt	call_stmt	compound_stmt
			selection_stmt	iteration_stmt	return_stmt
			input_stmt	output_stmt	
19	assignment_stmt	→	var **=** expression **;**		
20	call_stmt	→	call **;**		
21	selection_stmt	→	**if (** expression **)** statement	**if (** expression **)** statement **else** statement	
22	iteration_stmt	→	**while (** expression **)** statement		
23	return_stmt	→	**return ;**	**return** exression **;**	
24	input_stmt	→	**input** var **;**		
25	output_stmt	→	**output** expression ;		
26	var	→	**ID**	**ID [** arithmetic_expression **]**	
27	expression	→	arithmetic_expression relop arithmetic_expression	arithmetic_expression	
28	relop	→	**<=**	**<**	**>**
			>=	**==**	**!=**
29	arithmetic_expression	→	term arithmetic_expression'		
30	arithmetic_expression'	→	addop term arithmetic_expression'	**epsilon**	
31	addop	→	**+**	**-**	
32	term	→	factor term'		
33	term'	→	mulop factor term'	**epsilon**	
34	mulop	→	*****	**/**	
35	factor	→	**(** arithmetic_expression **)**	var	call
			NUM		
36	call	→	**ID (** args **)**		
37	args	→	args_list	**epsilon**	
38	args_list	→	arithmetic_expression args_list'		
39	args_list'	→	**,** arithmetic_expression args_list'	**epsilon**	

Figure 3. The grammar after rework for elimination of left recursion.

Elimination of Left Factors

Next, the grammar must be reworked in order to eliminate the left-factorization. Left factorization issues detected are highlighted in red in **Figure 3**:

1) In the productions of non-terminal **No. 5** '*var_declaration*':

   var_declaration →var_type_specifier ***ID*** **;** | *var_type_specifier* ***ID [NUM] ;***

   So it is replaced by the productions:

   var_declaration → var_type_specifier ***ID*** *var_declaration'* *var_declaration'* → ***;*** | ***[NUM] ;***

2) In the productions of non-terminal **No. 12** '*param'*:

   param → *var_type_specifier* ***ID*** | *var_type_specifier* ***ID []***

   So it is replaced by the productions:

   param → *var_type_specifier* ***ID*** *param'*

   param_list → ***[]*** | *ε*

3) In the productions of non-terminal **No. 21** '*selection_stmt':*

   selection_stmt → ***if (*** *expression* ***)*** *statement* | ***if (*** *expression* ***)*** *statement* ***else*** *statement*

   So they are replaced by the productions:

   selection_stmt → ***if (*** *expression* ***)*** *statement selection_stmt'*

   selection_stmt' → ***else*** *statement* | ***ε***

4) In the productions of non-terminal **No. 23** 'return_stmt':

   return_stmt→ ***return ;***| ***return*** *expression* ***;***

   So they are replaced by the productions:

   return_stmt → ***return*** *return_stmt'*

   return_stmt' → ***;*** | *expression* ***;***

5) In the productions of non-terminal **No. 26** '*var'*:

   var → ***ID*** | ***ID [*** *arithmetic_expression* ***]***

   So they are replaced by the productions:

   var → ***ID*** *var'*

   var'→ ***[*** *arithmetic_expression* ***]*** | *ε*

6) In the productions of non-terminal **No. 27** 'expression':

   expression → *arithmetic_expression relop arithmetic_expression* | *arithmetic_expression*

   So they are replaced by the productions:

   expression → *arithmetic_expression expression'*

   expression' → *relop arithmetic_expression* | *ε*

With all previous corrections, the grammar results as follows:

	Produccion				
No.	LHS	→	RHS	RHS	RHS
1	program	→	declaration_list **void ID (void)** compound_stmt	**void ID (void)** compound_stmt	
2	declaration_list	→	declaration declaration_list	**epsilon**	
3	declaration	→	var_declaration	fun_declaration	
4	var_declaration	→	var_type_specifier **ID** var_declaration'		
5	var_declaration'	→	**;**	**[NUM] ;**	
6	func_type_specifier	→	**int**	**void**	
7	var_type_specifier	→	**int**		
8	fun_declaration	→	func_type_specifier **ID (** params **)** compound_stmt		
9	params	→	param_list	**void**	
10	param_list	→	param param_list'		
11	param_list'	→	**,** param param_list'	**epsilon**	
12	param	→	var_type_specifier **ID** param'		
13	param'	→	**[ ]**	**epsilon**	
14	compound_stmt	→	**{** local_declarations statement_list **}**		
15	local_declarations	→	local_declarations'		
16	local_declarations'		var_declaration local_declarations'	**epsilon**	
17	statement_list	→	statement_list'		
18	statement_list'	→	statement statement_list'	**epsilon**	
19	statement	→	assignment_stmt	call_stmt	compound_stmt
			selection_stmt	iteration_stmt	return_stmt
			input_stmt	output_stmt	
20	assignment_stmt	→	var **=** expression **;**		
21	call_stmt	→	call **;**		
22	selection_stmt	→	**if (** expression **)** statement selection_stmt'		
23	selection_stmt'	→	**else** statement	**epsilon**	
24	iteration_stmt	→	**while (** expression **)** statement		
25	return_stmt	→	**return** return_stmt'		
26	return_stmt'	→	**;**	expression **;**	
27	input_stmt	→	**input** var **;**		
28	output_stmt	→	**output** expression ;		
29	var	→	**ID** var'		
30	var'	→	**[** arithmetic_expression **]**	**epsilon**	
31	expression	→	arithmetic_expression expression'		
32	expression'	→	relop arithmetic_expression	**epsilon**	
33	relop	→	**<=**	**<**	**>**
			>=	**==**	**!=**
34	arithmetic_expression	→	term arithmetic_expression'		
35	arithmetic_expression'	→	addop term arithmetic_expression'	**epsilon**	
36	addop	→	**+**	**-**	
37	term	→	factor term'		
38	term'	→	mulop factor term'	**epsilon**	
39	mulop	→	*****	**/**	
40	factor	→	**(** arithmetic_expression **)**	var	call
			NUM		
41	call	→	**ID (** args **)**		
42	args	→	args_list	**epsilon**	
43	args_list	→	arithmetic_expression args_list'		
44	args_list'	→	**,** arithmetic_expression args_list'	**epsilon**	

Figure 4. The grammar after elimination of left factors.

Simplification of the Grammar

Now the grammar is appropriate for the LL(1) algorithm, however, some additional simplifications have been made in order to better optimize the grammar, such as removal of useless symbols, removal of unit productions, and elimination of as many **ε**-productions as possible. In **Figure 4**, all ε-Productions are highlighted in red, however, most of these cannot be eliminated, since by doing this a left factor will be re-introduced in such a way that, by re-eliminating the left factor will re-introduce an ε-production. All such ε-productions are highlighted in blue in **Figure 5**:

	Produccion				
No. F	LHS	→	RHS	RHS	RHS
1	*program*	→	*declaration_list* **void ID (void)** *compound_stmt*	**void ID (void)** *compound_stmt*	
2	*declaration_list*	→	*declaration declaration_list*	**epsilon**	
4	*declaration*	→	*var_declaration*	*fun_declaration*	
5	*var_declaration*	→	*var_type_specifier* **ID** *var_declaration'*		
6	*var_declaration'*	→	**;**	**[NUM] ;**	
7	*func_type_specifier*	→	**int**	**void**	
8	*var_type_specifier*	→	**int**		
9	*fun_declaration*	→	*func_type_specifier* **ID (** *params* **)** *compound_stmt*		
10	*params*	→	*param_list*	**void**	
11	*param_list*	→	*param param_list'*		
12	*param_list'*	→	**,** *param param_list'*	**epsilon**	
13	*param*	→	*var_type_specifier* **ID** *param'*		
14	*param'*	→	**[ ]**	**epsilon**	
15	*compound_stmt*	→	**{** *local_declarations statement_list* **}**		
16	*local_declarations*	→	*local_declarations'*		
17	*local_declarations'*		*var_declaration local_declarations'*	**epsilon**	
18	*statement_list*	→	*statement_list'*		
19	statement_list'	→	statement statement_list'	**epsilon**	
20	*statement*	→	*assignment_stmt*	*call_stmt*	*compound_stmt*
			selection_stmt	*iteration_stmt*	*return_stmt*
			input_stmt	*output_stmt*	
21	*assignment_stmt*	→	*var* **=** *expression* **;**		
22	*call_stmt*	→	*call* **;**		
23	*selection_stmt*	→	**if (** *expression* **)** *statement selection_stmt'*		
24	*selection_stmt'*	→	**else** *statement*	**epsilon**	
25	*iteration_stmt*	→	**while (** *expression* **)** *statement*		
26	*return_stmt*	→	**return** *return_stmt'*		
27	*return_stmt'*	→	**;**	*expression* **;**	
28	*input_stmt*	→	**input** *var* **;**		
29	*output_stmt*	→	**output** *expression ;*		
30	*var*	→	**ID** *var'*		
31	*var'*	→	**[** arithmetic_expression **]**	**epsilon**	
32	*expression*	→	*arithmetic_expression expression'*		
33	*expression'*	→	*relop arithmetic_expression*	**epsilon**	
34	*relop*	→	**<=**	**<**	**>**
			>=	**==**	**!=**
35	*arithmetic_expression*	→	*term arithmetic_expression'*		
36	*arithmetic_expression'*	→	addop term arithmetic_expression'	**epsilon**	
37	*addop*	→	**+**	**-**	
38	*term*	→	*factor term'*		
39	term'	→	*mulop factor term'*	**epsilon**	
40	*mulop*	→	*	**/**	
41	*factor*	→	**(** *arithmetic_expression* **)**	*var*	*call*
			NUM		
42	*call*	→	**ID (** *args* **)**	**ID ( )**	
43	*args*	→	*args_list*		
44	*args_list*	→	*arithmetic_expression args_list'*		
45	args_list'	→	**,** *arithmetic_expression args_list'*	**epsilon**	

Figure 5. All ε-productions in the grammar.

However there is one ε-production that can be eliminated, seen as productions of non-terminal No. 42 '*args*' in **Figure 4**, and solved in **Figure 5** with the following method:

1) *call* → ***ID (args)*** can also be *call* → ***ID ( )***, so by adding this production to '*call*' the production *args* → *ε* can be eliminated.

This does re-introduce a left factor in non-terminal, but eliminating this does not produce a new ε-production, thus avoiding an infinite loop of elimination.

2) In the productions of non-terminal **No. 42** 'call':

   call→ ***ID (*** *args* ***)*** | ***ID ( )***

   So they are replaced by the productions:

   call → ***ID (*** *call'*

   call' → ***)*** | *args* ***)***

These changes result in the following grammar:

	Produccion				
No. Pr	LHS	→	RHS	RHS	RHS
1	program	→	declaration_list void ID (void) compound_stmt	void ID (void) compound_stmt	
2	declaration_list	→	declaration declaration_list	epsilon	
3	declaration	→	var_declaration	fun_declaration	
4	var_declaration	→	var_type_specifier ID var_declaration'		
5	var_declaration'	→	;	[NUM] ;	
6	func_type_specifier	→	int	void	
7	var_type_specifier	→	int		
8	fun_declaration	→	func_type_specifier ID (params) compound_stmt		
9	params	→	param_list	void	
11	param_list	→	param param_list'		
12	param_list'	→	, param param_list'	epsilon	
13	param	→	var_type_specifier ID param'		
14	param'	→	[ ]	epsilon	
15	compound_stmt	→	{ local_declarations statement_list }		
16	local_declarations	→	local_declarations'		
17	local_declarations'		var_declaration local_declarations'	epsilon	
18	statement_list	→	statement_list'		
19	statement_list'	→	statement statement_list'	epsilon	
20	statement	→	assignment_stmt	call_stmt	compound_stmt
			selection_stmt	iteration_stmt	return_stmt
			input_stmt	output_stmt	
21	assignment_stmt	→	var = expression ;		
22	call_stmt	→	call ;		
23	selection_stmt	→	if (expression) statement selection_stmt'		
24	selection_stmt'	→	else statement	epsilon	
25	iteration_stmt	→	while (expression) statement		
26	return_stmt	→	return return_stmt'		
27	return_stmt'	→	;	expression ;	
28	input_stmt	→	input var ;		
29	output_stmt	→	output expression ;		
30	var	→	ID var'		
31	var'	→	[arithmetic_expression]	epsilon	
32	expression	→	arithmetic_expression expression'		
33	expression'	→	relop arithmetic_expression	epsilon	
34	relop	→	<=	<	>
			>=	==	!=
35	arithmetic_expression	→	term arithmetic_expression'		
36	arithmetic_expression'	→	addop term arithmetic_expression'	epsilon	
37	addop	→	+	-	
38	term	→	factor term'		
39	term'	→	mulop factor term'	epsilon	
40	mulop	→	*	/	
41	factor	→	(arithmetic_expression)	var	call
			NUM		
42	call	→	ID (call'		
43	call'	→	args)	)	
44	args	→	args_list		
45	args_list	→	arithmetic_expression args_list'		
46	args_list'	→	, arithmetic_expression args_list'	epsilon	

Figure 6. The grammar after the correction of the ε-production at non-terminal 42.

Next, the elimination of unit productions. In **Figure 6**, all unit productions are highlighted in red; most of them will be eliminated, however, some will remain, since the variable is also used in other productions and their elimination might make the grammar more difficult to understand:

1) The production '*declaration → fun_declaration*' is replaced with the production '*declaration → func_type_specifier* ***ID (params) compound_stmt'***. And, in this case the non-terminal '*fun_declaration'* becomes a useless symbol and is eliminated from the grammar.
2) The production '*params → param_list*' is replaced with the production '*params → param param_list''*. And, in this case the non-terminal '*param_list'* becomes a useless symbol and is eliminated from the grammar.
3) The production '*local_declarations → local_declarations''* is replaced with the production '*local_declarations → var_declaration local_declaration''* AND '*local_declarations → ε*'. To avoid redundant productions the calls to '*local_declarations*' are changed into '*local_declarations*', the symbol '*local_declarations''* becomes useless, and is eliminated.
4) The production '*statement_list → statement_list''* is replaced with the production '*statement_list → statement statement_list*' ' AND '*statement_list → ε*'. To avoid redundant productions the calls to '*statement_list''* are changed into '*statement_list*', the symbol '*statement_list*'' becomes useless, and is eliminated.
5) The production '*statement → assignment_stmt'* is replaced with the production '*statement → var = expression* ***;***'. And, in this case, the non-terminal '*assignment_stmt'* becomes a useless symbol and is eliminated from the grammar.
6) The production '*statement → call_stmt'* is replaced with the production '*statement → call ;*'. And, in this case, the non-terminal '*call_stmt'* becomes a useless symbol and is eliminated from the grammar.
7) The production '*statement → iteration_stmt'* is replaced with the production '*statement →* ***while (*** *expression* ***)*** *statement*'. And, in this case, the non-terminal '*iteration_stmt'* becomes a useless symbol and is eliminated from the grammar.
8) The production '*statement → return_stmt'* is replaced with the production '*statement →* ***return*** *return_stmt''*. And, in this case, the non-terminal '*return_stmt'* becomes a useless symbol and is eliminated from the grammar.

9) The production '*statement* → *input_stmt*' is replaced with the production '*statement* → ***input*** *var* ***;***'. And, in this case, the non-terminal '*input_stmt*' becomes a useless symbol and is eliminated from the grammar.
10) The production '*statement* → *output_stmt*' is replaced with the production '*statement* → ***output*** *expression* ***;***'. And, in this case, the non-terminal '*output_stmt*' becomes a useless symbol and is eliminated from the grammar.
11) The production '*args* → *args_list*' is replaced with the production '*args* → *arithmetic_expression args_list*''. Now the symbol '*args_list*' is a useless symbol and is eliminated from the grammar, afterwards, for the sake of readability, the symbol '*args_list''* is changed into '*args_list*' in all its appearances.

With these changes, the resulting grammar is the following:

	Produccion				
No. Pr	LHS	→	RHS	RHS	RHS
1	program	→	declaration_list **void ID (void)** compound_stmt	**void ID (void)** compound_stmt	
2	declaration_list	→	declaration declaration_list	**epsilon**	
3	declaration	→	var_declaration	func_type_specifier **ID (** params **)** compound_stmt	
4	var_declaration	→	**int ID** var_declaration'		
5	var_declaration'	→	;	**[NUM] ;**	
6	func_type_specifier	→	**int**	**void**	
7	params	→	param param_list	**void**	
8	param_list	→	**,** param param_list	**epsilon**	
9	param	→	**int ID** param'		
10	param'	→	**[ ]**	**epsilon**	
11	compound_stmt	→	**{** local_declarations statement_list **}**		
12	local_declarations	→	var_declaration local_declarations	**epsilon**	
13	statement_list	→	statement statement_list	**epsilon**	
14	statement	→	var = expression ;	call ;	compound_stmt
			selection_stmt	**while (** expression **)** statement	**return** return_stmt'
			input var ;	**output** expression ;	
15	selection_stmt	→	**if (** expression **)** statement selection_stmt'		
16	selection_stmt'	→	**else** statement	**epsilon**	
17	return_stmt'	→	;	expression ;	
18	var	→	**ID** var'		
19	var'	→	**[** arithmetic_expression **]**	**epsilon**	
20	expression	→	arithmetic_expression expression'		
21	expression'	→	relop arithmetic_expression	**epsilon**	
22	relop	→	<=	<	>
			>=	==	!=
23	arithmetic_expression	→	term arithmetic_expression'		
24	arithmetic_expression'	→	addop term arithmetic_expression'	**epsilon**	
25	addop	→	+	-	
26	term	→	factor term'		
27	term'	→	mulop factor term'	**epsilon**	
28	mulop	→	*	/	
29	factor	→	**(** arithmetic_expression **)**	var	call
			NUM		
30	call	→	**ID (** call'		
31	call'	→	args **)**	**)**	
32	args	→	arithmetic_expression args_list		
33	args_list	→	**,** arithmetic_expression args_list	**epsilon**	

Figure 7. The grammar after eliminating Unit Productions.

An unexpected visitor

The next step is to eliminate deep left factors. Why just now, as opposed to doing so along with the normal left factors? Deep left factor is not covered in class material, so students had to find out this problem by themselves. Accordingly, this step is covered after several attempts of constructing the Parsing Table and encountering multiple productions in a single cell. The method is not guaranteed to not affect the grammar, since it was an educated guess:

1) The production '*program* → *declaration_list* ***void ID (void)*** *compound_stmt* | ***void ID (void)*** *compound_stmt'* is impossible to fix without modifying the grammar outside of its definition, so it is replaced with '*program* → *declaration_list'* and '*declaration_list* →*declaration declaration_list''* and '*declaration_list'* → *declaration declaration_list'* and '*declaration_list'* → *ε'*. This way there is still at least one declaration in the program, and the ID and parameters will be checked via code.
2) The productions '*declaration* → *var_declaration* | *func_type_specifier* ***ID (*** *params* ***)*** *compound_stmt*' are replaced by the productions '*declaration* →***int ID*** *declaration'* | ***void ID (*** *params* ***)*** *compound_stmt'* and '*declaration'* → *var_declaration'* | ***(*** *params* ***)*** *compound_stmt'*. This makes the symbol '*var_declaration*' a useless symbol, and is eliminated from the grammar.
3) The productions '*statement* → *var* = *expression* ***;*** | *statement* → *call* ***;'*** are replaced with a single production: '*statement* → ***ID*** *var_or_call_stmt'* and the following productions were added: '*var_or_call_stmt* → *var'* = *expression* ***;*** | ***(*** *call'* ***;'***.
4) The productions '*factor* → *var* | *call'* were replaced with a single production: '*factor* → ***ID*** *var_or_call'* and the following productions were added: '*var_or_call* → *var'* | ***(*** *call''*.

Figure 8 depicts the grammar after the previous changes:

	Produccion				
No. Pr	LHS	→	RHS	RHS	RHS
1	program	→	declaration_list		
2	declaration_list	→	declaration declaration_list'		
3	declaration_list'	→	declaration declaration_list'	**epsilon**	
4	declaration	→	**int ID** declaration'	**void ID (** params **)** compound_stmt	
5	declaration'	→	var_declaration'	**(** params **)** compound_stmt	
6	var_declaration'	→	**;**	**[NUM] ;**	
7	params	→	param param_list	**void**	
8	param_list	→	**,** param param_list	**epsilon**	
9	param	→	**int ID** param'		
10	param'	→	**[ ]**	**epsilon**	
11	compound_stmt	→	**{** local_declarations statement_list **}**		
12	local_declarations	→	**int ID** var_declaration' local_declarations	**epsilon**	
13	statement_list	→	statement statement_list	**epsilon**	
14	statement	→	**ID** var_or_call_stmt	compound_stmt	selection_stmt
			while (expression **)** statement	**return** return_stmt'	**input ID** var' **;**
			output expression **;**		
15	var_or_call_stmt	→	var' **=** expression **;**	**(** call' **;**	
16	var_or_call	→	var'	**(** call'	
17	selection_stmt	→	**if (** expression **)** statement selection_stmt'		
18	selection_stmt'	→	**else** statement	**epsilon**	
19	return_stmt'	→	**;**	expression **;**	
20	var'	→	**[** arithmetic_expression **]**	**epsilon**	
21	expression	→	arithmetic_expression expression'		
22	expression'	→	relop arithmetic_expression	**epsilon**	
23	relop	→	**<=**	**<**	**>**
			>=	**==**	**!=**
24	arithmetic_expression	→	term arithmetic_expression'		
25	arithmetic_expression'	→	addop term arithmetic_expression'	**epsilon**	
26	addop	→	**+**	**-**	
27	term	→	factor term'		
28	term'	→	mulop factor term'	**epsilon**	
29	mulop	→	*	**/**	
30	factor	→	**(** arithmetic_expression **)**	**ID** var_or_call	**NUM**
31	call'	→	args **)**	**)**	
32	args	→	arithmetic_expression args_list		
33	args_list	→	**,** arithmetic_expression args_list	**epsilon**	

Figure 8. The grammar after the elimination of deep left factors.

Finally one last simplification is made to the grammar:

1) The non-terminal symbol '*param*' will be eliminated by replacing all its calls with its contents, since '*param'* only appears in the left-hand-side of a production once, it is easy to expand its calls. The productions: '*params* → *param param_list*' and '*param_list* → **,** *param param_list'* will be replaced by the productions: 'params → ***int ID*** *param' param_list*' and '*param_list* → **,** ***int ID*** *param' param_list*' respectively, and the production '*param* → ***int ID*** *param'*' since the non-terminal '*param*' has become a useless symbol.

The grammar results as follows:

	Produccion				
No. Pro	LHS	→	RHS	RHS	RHS
1	program	→	declaration_list		
2	declaration_list	→	declaration declaration_list'		
3	declaration_list'	→	declaration declaration_list'	**epsilon**	
4	declaration	→	**int ID** declaration'	**void ID (** params **)** compound_stmt	
5	declaration'	→	var_declaration'	**(** params **)** compound_stmt	
6	var_declaration'	→	**;**	**[NUM] ;**	
7	params	→	**int ID** param' param_list	**void**	
8	param_list	→	**, int ID** param' param_list	**epsilon**	
9	param'	→	**[ ]**	**epsilon**	
10	compound_stmt	→	**{** local_declarations statement_list **}**		
11	local_declarations	→	**int ID** var_declaration' local_declarations	**epsilon**	
12	statement_list	→	statement statement_list	**epsilon**	
13	statement	→	**ID** var_or_call_stmt	compound_stmt	selection_stmt
			while (expression **)** statement	**return** return_stmt'	**input ID** var' **;**
			output expression **;**		
14	var_or_call_stmt	→	var' **=** expression **;**	**(** call' **;**	
15	var_or_call	→	var'	**(** call'	
16	selection_stmt	→	**if (** expression **)** statement selection_stmt'		
17	selection_stmt'	→	**else** statement	**epsilon**	
18	return_stmt'	→	**;**	expression **;**	
19	var'	→	**[** arithmetic_expression **]**	**epsilon**	
20	expression	→	arithmetic_expression expression'		
21	expression'	→	relop arithmetic_expression	**epsilon**	
22	relop	→	**<=**	**<**	**>**
			>=	**==**	**!=**
23	arithmetic_expression	→	term arithmetic_expression'		
24	arithmetic_expression'	→	addop term arithmetic_expression'	**epsilon**	
25	addop	→	**+**	**-**	
26	term	→	factor term'		
27	term'	→	mulop factor term'	**epsilon**	
28	mulop	→	*	/	
29	factor	→	**(** arithmetic_expression **)**	**ID** var_or_call	**NUM**
30	call'	→	args **)**	**)**	
31	args	→	arithmetic_expression args_list		
32	args_list	→	**,** arithmetic_expression args_list	**epsilon**	

Figure 9. The final grammar.

This grammar still has one issue which will become clearer during the creation of the parsing table, and can be resolved during that step.

FIRST, FOLLOW, & FIRST+ sets & Parsing Table

Once the grammar is ready, the FIRST and FOLLOW sets must be calculated, in order to calculate the FIRST+ set, which is the one that will allow the LL(1) algorithm to make the parsing decisions and accept or reject the string of C minus code provided via the Parsing Table.

The FIRST (FIR) set is calculated for the terminals, non-terminals, and ε, according to the following rules:

1) If $\mathbf{X} \rightarrow \mathbf{aY}$, then $\mathbf{First(X)} = \mathbf{First(X)} \cup \{\mathbf{a}\}$.

2) If $\mathbf{X} \rightarrow \boldsymbol{\varepsilon}$, then $\mathbf{First(X)} = \mathbf{First(X)} \cup \{\boldsymbol{\varepsilon}\}$.

3) If $\mathbf{X} \rightarrow \mathbf{Y_1 Y_2 \ldots Y_n}$, then :

   a. $\mathbf{First(X)} = \mathbf{First(X)} \cup \mathbf{First(Y_1)} - \{\boldsymbol{\varepsilon}\}$

   b. If $\mathbf{Y_1} \Rightarrow \boldsymbol{\varepsilon}$, then
   $\mathbf{First(X)} = \mathbf{First(X)} \cup \mathbf{First(Y_2)} - \{\boldsymbol{\varepsilon}\}$

   c. If $\mathbf{Y_1} \Rightarrow \boldsymbol{\varepsilon} \wedge \mathbf{Y_2} \Rightarrow \boldsymbol{\varepsilon} \wedge \ldots \wedge \mathbf{Y_i} \Rightarrow \boldsymbol{\varepsilon} \wedge i < n$, then
   $\mathbf{First(X)} = \mathbf{First(X)} \cup \mathbf{First(Y_{i+1})} - \{\boldsymbol{\varepsilon}\}$

   d. If $\mathbf{Y_1} \Rightarrow \boldsymbol{\varepsilon} \wedge \mathbf{Y_2} \Rightarrow \boldsymbol{\varepsilon} \wedge \ldots \wedge \mathbf{Y_n} \Rightarrow \boldsymbol{\varepsilon}$, then
   $\mathbf{First(X)} = \mathbf{First(X)} \cup \{\boldsymbol{\varepsilon}\}$

Figure 10. Rules of calculating the FIRST Set [2].

So for every terminal, non-terminal, and ε, the FIRST set is calculated:

TERMINALS & e	
else	{ else }
if	{ if }
input	{ input }
int	{ int }
output	{ output }
return	{ return }
void	{ void }
while	{ while }
+	{ + }
-	{ - }
*****	{ * }
/	{ / }
>	{ > }
>=	{ >= }
<	{ < }
<=	{ <= }
==	{ == }
!=	{ != }
=	{ = }
;	{ ; }
,	{ , }
(	{ (}
)	{) }
{	{ { }
}	{ } }
[	{ [}
]	{] }
IDENTIFIER	{ IDENTIFIER }
NUM_CONSTANT	{ NUM_CONSTANT }
epsilon	{ epsilon }

Figure 11. The FIRST set of the terminals and ε

For the non-terminals:

1) FIR(*program*) = FIR(*declaration_list*) = {**int**, **void**}
2) FIR(*declaration_list*) = FIR(*declaration*) = {**int**, **void**}
3) FIR(*declaration_list'*) = FIR(*declaration*) u FIR(ε) = {**ε**, **int**, **void**}
4) FIR(*declaration*) = FIR(int) u FIR(void) = {**int**, **void**}
5) FIR(*declaration'*) = FIR(*var_declaration'*) u FIR(( ) = { **;**, **[**, **(** }

6) FIR(*var_declaration’*) = FIR(;) u FIR([) = { ;, [}
7) FIR(*params*) = FIR(int) u FIR(void) = {int, void}
8) FIR(*param_list*) = FIR(,) u FIR(ε) = {ε, ,}
9) FIR(*param’*) = FIR([) u FIR(ε) = {ε, [}
10) FIR(*compound_stmt*) = FIR({) = { { }
11) FIR(*local_declarations*) = FIR(int) u FIR(ε) = {ε, int}
12) FIR(*statement_list*) = FIR(*statement*) u FIR(ε) = {ε, ID, {, if, while, return, input, output}
13) FIR(*statement*) = FIR(ID) u FIR(*compound_stmt*) u FIR(*selection_stmt*) u FIR(while) u FIR(return) u FIR(input) u FIR(output) = {ID, {, if, while, return, input, output}
14) FIR(*var_or_call_stmt*) = FIR(*var’*) u FIR(() - {ε} u FIR(=) = {[, (, =}
15) FIR(*var_or_call*) = FIR(*var’*) u FIR(() = {ε, [, (}
16) FIR(*selection_stmt*) = FIR(if) = { if }
17) FIR(*selection_stmt’*) = FIR(else) u FIR(ε) = {ε, else}
18) FIR(*return_stmt’*) = FIR(;) u FIR(*expression*) = { ;, (, ID, NUM }
19) FIR(*var’*) = FIR([) u FIR(ε) = {ε, [}
20) FIR(*expression*) = FIR(*arithmetic_expression*) = { (, ID, NUM }
21) FIR(*expression’*) = FIR(*relop*) u FIR(ε) = {ε, <=, <, >, >=, ==, !=}
22) FIR(*relop*) = FIR(<=) u FIR(<) u FIR(>) u FIR(>=) u FIR(==) u FIR(!=) = { <=, <, >, >=, ==, !=}
23) FIR(*arithmetic_expression*) = FIR(*term*) = {(, ID, NUM}
24) FIR(*arithmetic_expression’*) = FIR(ε) u FIR(addop) = {ε, +, -}
25) FIR(*addop*) = FIR(+) u FIR(-) = {+, -}
26) FIR(*term*) = FIR(*factor*) = {(, ID, NUM}
27) FIR(*term’*) = FIR(*mulop*) u FIR(ε) = {ε, *, /}
28) FIR(*mulop*) = FIR(*) u FIR(/) = {*, /}
29) FIR(*factor*) = FIR(() u FIR(ID) FIR(NUM) = {(, ID, NUM}
30) FIR(*call’*) = FIR(*args*) u FIR()) = { (,), ID, NUM}
31) FIR(*args*) = FIR(*arithmetic_expression*) = {(, ID, NUM}
32) FIR(*args_list*) = FIR(,) u FIR(ε) = {ε, ,}

Calculating the FIRST set:	
NON-TERMINALS	
program	{ **int**, **void** }
declaration_list	{ **int**, **void** }
declaration_list'	{ **int**, **void**, **epsilon** }
declaration	{ **int**, **void** }
declaration'	{ **;**, **[**, **(** }
var_declaration'	{ **;**, **[** }
params	{ **int**, **void** }
param_list	{ **,**, **epsilon** }
param'	{ **[**, **epsilon** }
compound_stmt	{ **{** }
local_declarations	{ **int**, **epsilon** }
statement_list	{ **ID**, **{**, **if**, **while**, **return**, **input**, **output**, **epsilon**
statement	{ **ID**, **{**, **if**, **while**, **return**, **input**, **output** }
var_or_call_stmt	{ **(**, **[**, **=** }
var_or_call	{ **(**, **[**, **epsilon** }
selection_stmt	{ **if** }
selection_stmt'	{ **else**, **epsilon** }
return_stmt'	{ **;**, **(**, **ID**, **NUM** }
var'	{ **[**, **epsilon** }
expression	{ **(**, **ID**, **NUM** }
expression'	{ **<**, **<=**, **>**, **>=**, **==**, **!=**, **epsilon** }
relop	{ **<**, **<=**, **>**, **>=**, **==**, **!=** }
arithmetic_expression	{ **(**, **ID**, **NUM** }
arithmetic_expression	{ **+**, **-**, **epsilon** }
addop	{ **+**, **-** }
term	{ **(**, **ID**, **NUM** }
term'	{ *****, **/**, **epsilon** }
mulop	{ *****, **/** }
factor	{ **(**, **ID**, **NUM** }
call'	{ **(**, **)**, **ID**, **NUM** }
args	{ **(**, **ID**, **NUM** }
args_list	{ **,**, **epsilon** }

Figure 12. The FIRST sets of the non-terminals of the grammar.

Next, the FOLLOW (FOL) set is calculated for the non-terminals, according to the rules in **Figure 13**:

1) For the start symbol, **Follow**(**S**) = **Follow**(**S**) $\cup$ {**$**}.

2) If there is a production **X** $\rightarrow \alpha$**Y**β, then
Follow(**Y**) = **First**(β) - {ε}.

3) If there is a production **X** $\rightarrow \alpha$**Y**$\beta\theta$, where $\varepsilon \in$ **First**(β) then,
Follow(**Y**) = **First**(β) - {ε} $\cup$ **First**(θ) - {ε}

4) If there is a production **X** $\rightarrow \alpha$**Y** or **X** $\rightarrow \alpha$**Y**β and $\varepsilon \in$ **First**(β) then,
Follow(**Y**) = **Follow**(**Y**) $\cup$ **Follow**(**X**).

Figure 13. Rules for calculating the FOLLOW set of a set of non-terminals.

For every non-terminal, the FOLLOW set is calculated:

1) FOL(*program*) = FOL(*program*) u { $ } = { $ }
2) FOL(*declaration_list*) = FOL(*program*) = { $ }
3) FOL(*declaration_list'*) = FOL(*declaration_list*) u FOL(*declaration_list'*) = { $ }
4) FOL(*declaration*) = FIR(*declaration_list'*) - {ε} u FOL(*declaration_list*) u FOL(*declaration_list'*) = {int, void, $}
5) FOL(*declaration'*) = FOL(*declaration*) = {int, void, $}
6) FOL(*var_declaration'*) = FOL(*declaration'*) = {int, void, $}
7) FOL(*params*) = FIR()) = {) }
8) FOL(*param_list*) = FOL(*params*) u FOL(*param_list*) = {) }
9) FOL(*param'*) = FIR(*param_list*) - {ε} u FOL(*params*) u FOL(*params_list*) = {,,)}
10) FOL(*compound_stmt*) = FOL(*declaration*) u FOL(*declaration'*) u FOL(*statement*) = {int, void, $, ID, {, }, if, else, while, return, input, output}
11) FOL(*local_declarations*) = FIR(*statement_list*) - {ε} u FIR(}) u FOL(*local_declarations*) = {ID, {, }, if, while, return, input, output}
12) FOL(*statement_list*) = FIR(}) u FOL(*statement_list*) = {}}
13) FOL(*statement*) = FIR(*statement_list*) - {ε} u FOL(*statement_list*) u FOL(*statement*) u FIR(*selection_stmt'*) u FOL(*selection_stmt'*) = {ID, {, if, else, while, return, input, output}
14) FOL(*var_or_call_stmt*) = FOL(*statement*) = {ID, {, if, else, while, return, input, output, }}
15) FOL(*var_or_call*) = FOL(*factor*) = {*, /, +, -,], <, <=, >, >=, ==, !=,), ;, ,}

16) FOL(*selection_stmt*) = FOL(*statement*) = {ID, {, if, else, while, return, input, output, }}
17) FOL(*selection_stmt’*) = FOL(*selection_stmt*) = {ID, {, if, else, while, return, input, output, }}
18) FOL(*return_stmt’*) = FOL(*statement*) = {ID, {, }, if, else, while, return, input, output}
19) FOL(*var’*) = FIR(;) u FIR(=) u FOL(*var_or_call*) = {;, =, *, /, +, -,], <, <=, >, >=, ==, !=,), , }
20) FOL(*expression*) = FIR()) u FIR(;) = {), ,}
21) FOL(*expression’*) = FOL(*expression*) = {), ,}
22) FOL(*relop*) = FIR(*arithmetic_expression*) = {(, ID, NUM}
23) FOL(*arithmetic_expression*) = FIR(]) u FIR(*expression’*) u FOL(*expression*) u FOL(*expression’*) u FIR(*args_list*) u FOL(*args*) u FOL(*args_list*) = {], <=, <, >, >=, ==, !=,), ;, ,}
24) FOL(*arithmetic_expression’*) = FOL(*arithmetic_expression*) = {], <=, <, >, >=, ==, !=,), ;, ,}
25) FOL(*addop*) = FIR(*term*) = {(, ID, NUM}
26) FOL(*term*) = FIR(*arithmetic_expression’*) u FOL(*arithmetic_expression’*) = {+, -,], <=, <, >, >=, ==, !=,), ;, ,}
27) FOL(*term’*) = FOL(*term*) u FOL(*term’*) = {+, -,], <=, <, >, >=, ==, !=.), ;, ,}
28) FOL(*mulop*) = FIR(*factor*) = {(, ID, NUM}
29) FOL(*factor*) = FIR(*term’*) - {ε} u FOL(*term*) u FOL(*term’*) = {*, /, +, -,], <, <=, >, >=, ==, !=,), ;, ,}
30) FOL(*call’*) = FIR(;) u FOL(*var_or_call*) = {*, /, +, -,], <, <=, >, >=, ==, !=,), ;, ,}
31) FOL(*args*) = FIR()) = {) }
32) FOL(*args_list*) = FOL(*args*) u FOL(*args_list*) = {) }

Calculating the FOLLOW set:	
NON-TERMINALS	
program	{ **$** }
declaration_list	{ **$** }
declaration_list'	{ **$** }
declaration	{ int, void, **$** }
declaration'	{ int, void, **$** }
var_declaration'	{ int, void, **$** }
params	{) }
param_list	{) }
param'	{ ,,) }
compound_stmt	{ $, int, void, ID, {, }, if, while, return, input, output, else }
local_declarations	{ ID, {, }, if, while, return, input, output }
statement_list	{ } }
statement	{ ID, {, }, if, while, return, input, output, else }
var_or_call_stmt	{ ID, {, }, if, while, return, input, output, else }
var_or_call	{ *, /, +, -,], <, <=, >, >=, ==, !=,), ;, , }
selection_stmt	{ ID, {, }, if, while, return, input, output, else }
selection_stmt'	{ ID, {, }, if, while, return, input, output, else }
return_stmt'	{ ID, {, }, if, while, return, input, output, else }
var'	{ =, *, /, +, -,], <, <=, >, >=, ==, !=,), ;, , }
expression	{), ; }
expression'	{), ; }
relop	{ (, ID, NUM }
arithmetic_expression	{],<, <=, >, >=, ==, !=,), ;, , }
arithmetic_expression'	{],<, <=, >, >=, ==, !=,), ;, , }
addop	{ (, ID, NUM }
term	{ +, -,], <, <=, >, >=, ==, !=,), ;, , }
term'	{ +, -,], <, <=, >, >=, ==, !=,), ;, , }
mulop	{ (, ID, NUM }
factor	{ *, /, +, -,], <, <=, >, >=, ==, !=,), ;, , }
call'	{*, /, +, -,], <, <=, >, >=, ==, !=,), ;, ,}
args	{) }
args_list	{) }

Figure 14. The FOLLOW sets for the non-terminals of the grammar.

Finally, the FIRST+ (FIP) set is calculated for the non-terminals according to the rules in **Figure 15**:

For each production **X** → β, its ***augmented First Set*** **First⁺**(**X** → β) is defined as follows:

1) If $\varepsilon \notin$ **First**(β), then **First⁺**(**X** → β) = **First**(β).

2) If $\varepsilon \in$ **First**(β), then

$$\textbf{First}^{+}(\textbf{X} \rightarrow \beta) = \textbf{First}(\beta) \cup \textbf{Follow}(\textbf{X}).$$

Figure 15. Rules for calculating the FIRST+ set of a grammar [2].

For every production, the set is calculated:

1) FIP(1) = FIR(*declaration_list*) = {int, void}
2) FIP(2) = FIR(*declaration*) = {int, void}
3) FIP(3) = FIR(*declaration*) = {int, void}
4) FIP(4) = FIR(ε) u FOL(*declaration_list'*) = {ε, $}
5) FIP(5) = FIR(int) = {int}
6) FIP(6) = FIR(void) = {void}
7) FIP(7) = FIR(*var_declaration'*) = {;, [}
8) FIP(8) = FIR(() = { (}
9) FIP(9) = FIR(;) = { ; }

10) FIP(10) = FIR([) = {] }
11) FIP(11) = FIR(int) = {int}
12) FIP(12) = FIR(void) = {void}
13) FIP(13) = FIR(,) = { , }
14) FIP(14) = FIR(ε) u FOL(*param_list*) = {ε,)}
15) FIP(15) = FIR([) = { [}
16) FIP(16) = FIR(ε) u FOL(*param'*) = {ε, ,,)}
17) FIP(17) = FIR({) = { { }
18) FIP(18) = FIR(int) = {int}
19) FIP(19) = FIR(ε) u FOL(*local_declarations*) = {ε, ID, {, }, if, while, return, input, output}
20) FIP(20) = FIR(*statement*) = {ID, {, if, while, return, input, output}
21) FIP(21) = FIR(ε) u FOL(*statement_list*) = {ε, }}
22) FIP(22) = FIR(ID) = { ID }
23) FIP(23) = FIR(*compound_stmt*) = { { }
24) FIP(24) = FIR(*selection_stmt*) = {if}

25) FIP(25) = FIR(while) = { while }
26) FIP(26) = FIR(return) = { return }
27) FIP(27) = FIR(input) = { input }
28) FIP(28) = FIR(output) = { output }
29) FIP(29) = FIR(*var’*) u FIR(=) = {ε, [, =}
30) FIP(30) = FIR(() = { (}
31) FIP(31) = FIR(*var’*) u FOL(*var_or_call*) = {ε, [, *, /, +, -,], <, <=, >, >=, ==, !=,), ;, ,}
32) FIP(32) = FIR(() = { (}
33) FIP(33) = FIR(if) = { if }
34) FIP(34) = FIR(else) = {else}
35) FIP(35) = FIR(ε) u FOL(*selection_stmt’*) = {ε, ID, {, }, if, else, while, return, input, output}
36) FIP(36) = FIR(;) = { ; }
37) FIP(37) = FIR(*expression*) = { (, ID, NUM }
38) FIP(38) = FIR([) = { [}
39) FIP(39) = FIR(ε) u FOL(*var’*) = {ε, ;, =, *, /, +, -,], <, <=, >, >=, ==, !=,), ,}
40) FIP(40) = FIR(*arithmetic_expression*) = {(, ID, NUM}
41) FIP(41) = FIR(*relop*) = {<, <=, >, >=, ==, !=}
42) FIP(42) = FIR(ε) u FOL(*expression’*) = {ε,), ;}
43) FIP(43) = FIR(<=) = {<= }
44) FIP(44) = FIR(<) = { < }
45) FIP(45) = FIR(>) = { > }
46) FIP(46) = FIR(>=) = { >= }
47) FIP(47) = FIR(==) = { == }
48) FIP(48) = FIR(!=) = { != }
49) FIP(49) = FIR(*term*) = {(, ID, NUM}
50) FIP(50) = FIR(*addop*) = {+, -}
51) FIP(51) = FIR(ε) u FOL(*arithmetic_expression’*) = {ε,], <, <=, >, >=, ==, !=,), ;, ,}
52) FIP(52) = FIR(+) = { + }
53) FIP(53) = FIR(-) = { - }
54) FIP(54) = FIR(*factor*) = {(, ID, NUM}
55) FIP(55) = FIR(*mulop*) = {*, /}
56) FIP(56) = FIR(ε) u FOL(*term’*) = {ε, +, -,], <, <=, >, >=, ==, !=,), ;, ,}

57) FIP(57) = FIR(*) = { * }

58) FIP(58) = FIR(/) = { / }

59) FIP(59) = FIR(() = { (}

60) FIP(60) = FIR(ID) = { ID }

61) FIP(61) = FIR(NUM) = { NUM }

62) FIP(62) = FIR(*args*) = {(, ID, NUM}

63) FIP(63) = FIR()) = {) }

64) FIP(64) = FIR(*arithmetic_expression*) = {(, ID, NUM}

65) FIP(65) = FIR(,) = { , }

66) FIP(66) = FIR(ε) u FOL(*args_list*) = {ε,)}

Calculating the FIRST+ set:		
	PRODUCTIONS	
1	*program → declaration_list*	{ int, void }
2	*declaration_list → declaration declaration_list'*	{ int, void }
3	*declaration_list' → declaration declaration_list'*	{ int, void }
4	*declaration_list' → **epsilon***	{ **epsilon**, $ }
5	declaration → **int ID** declaration'	{ int }
6	declaration → **void ID (** params **)** compound_stmt	{ void }
7	declaration' → var_declaration'	{ ;, [}
8	declaration' → **(** params **)** compound_stmt	{ (}
9	var_declaration' → **;**	{ ; }
10	var_declaration' → **[NUM] ;**	{ [}
11	params → **int ID** param' param_list	{ int }
12	params → **void**	{ void }
13	param_list → **, int ID** param' param_list	{ , }
14	param_list → **epsilon**	{ **epsilon**,) }
15	param' → **[ ]**	{ [}
16	param' → **epsilon**	{ **epsilon**, ,,) }
17	compound_stmt → **{** local_declarations statement_list **}**	{ { }
18	local_declarations → **int ID** var_declaration' local_declarations	{ int }
19	local_declarations → **epsilon**	{ **epsilon**, ID, {, }, if, while, return, input, output }
20	statement_list → statement statement_list	{ ID, {, if, while, return, input, output }
21	statement_list → **epsilon**	{ **epsilon**, } }
22	statement → **ID** var_or_call_stmt	{ ID }
23	statement → compound_stmt	{ { }
24	statement → selection_stmt	{ if }
25	statement → **while (** expression **)** statement	{ while }
26	statement → **return** return_stmt'	{ return }
27	statement → **input ID** var' ;	{ input }
28	statement → **output** expression ;	{ output }
29	var_or_call_stmt → var' **=** expression **;**	{ **epsilon**, [, = }
30	var_or_call_stmt → **(** call' **;**	{ (}
31	var_or_call → var'	{ **epsilon**, [, *, /, +, -,], <, <=, >, >=, ==, !=,), ;, ,}}
32	var_or_call → **(** call'	{ (}
33	selection_stmt → **if (** expression **)** statement selection_stmt'	{ if }

34	selection_stmt' → **else** statement	{ else }
35	selection_stmt' → **epsilon**	{ **epsilon**, ID, {, }, if, else, while, return, input, output }
36	return_stmt' → **;**	{ ; }
37	return_stmt' → expression **;**	{ (, ID, NUM }
38	var' → **[** arithmetic_expression **]**	{ [}
39	var' → **epsilon**	{ **epsilon**, ;, =, (, ID, NUM }
40	expression → arithmetic_expression expression'	{ (, ID, NUM }
41	expression' → relop arithmetic_expression	{ <, <=, >, >=, ==, != }
42	expression' → **epsilon**	{ **epsilon**,), ; }
43	relop → **<=**	{ <= }
44	relop → **<**	{ < }
45	relop → **>**	{ > }
46	relop → **>=**	{ >= }
47	relop → **==**	{ == }
48	relop → **!=**	{ != }
49	arithmetic_expression → term arithmetic_expression'	{ (, ID, NUM }
50	arithmetic_expression' → addop term arithmetic_expression'	{ +, - }
51	arithmetic_expression' → **epsilon**	{ **epsilon**,], <, <=, >, >=, ==, !=,), ;, , }
52	addop → **+**	{ + }
53	addop → **-**	{ - }
54	term → factor term'	{ (, ID, NUM }
55	term' → mulop factor term'	{ *, / }
56	term' → **epsilon**	{ **epsilon**, +, -,], <, <=, >, >=, ==, !=,), ;, , }
57	mulop → *	{ * }
58	mulop → **/**	{ / }
59	factor → **(** arithmetic_expression **)**	{ (}
60	factor → **ID** var_or_call	{ ID }
61	factor → **NUM**	{ NUM }
62	call' → args **)**	{ (, ID, NUM }
63	call' → **)**	{) }
64	args → arithmetic_expression args_list	{ (, ID, NUM }
65	args_list → **,** arithmetic_expression args_list	{ , }
66	args_list → **epsilon**	{ **epsilon**,) }

Figure 16. The FIRST+ Sets of the grammar.

Next, the Parsing Table is calculated according to the algorithm:

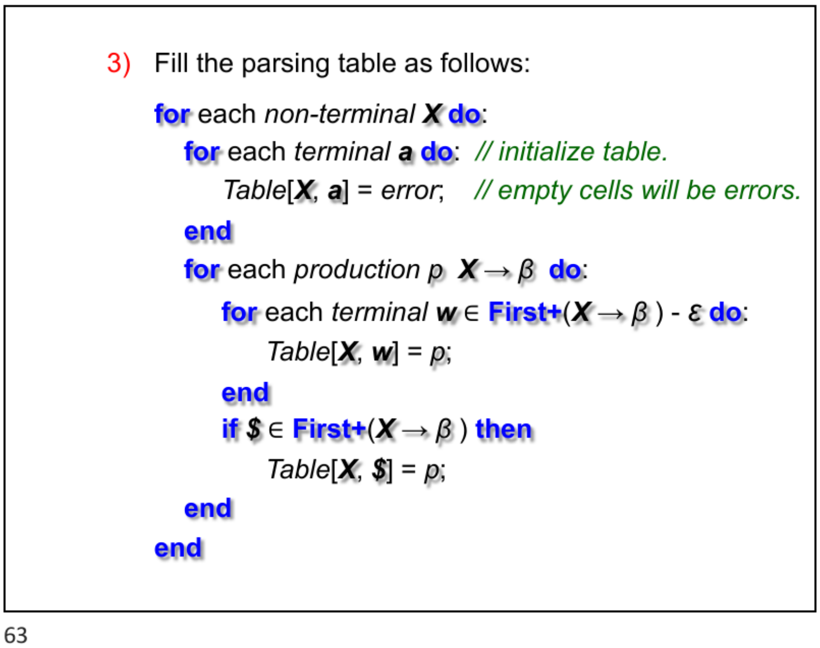

Figure 17. Algorithm for filling the Parsing Table.

The Parsing Table is filled using the FIRST+ sets, and results as follows:

	else	if	input	int	output	return	void	while	+	-	*	/	>	>=	<	<=	==	!=	=	;	,	(	)	{	}	[	]	ID	NUM	$
program				1			1																							
declaration_list				2			2																							
declaration_list'				3			3																							4
declaration				5			6																							
declaration'																				7		8				7				
var_declaration'																				9						10				
params				11			12																							
param_list																					13		14							
param'																					16		16			15				
compound_stmt																								17						
local_declarations		19	19	18	19	19		19																19	19			19		
statement_list		20	20		20	20		20																20	21			20		
statement		24	27		28	26		25																23				22		
var_or_call_stmt																			29			30				29				
var_or_call									31	31	31	31	31	31	31	31	31	31		31	31	32	31			31	31			
selection_stmt		33																												
selection_stmt'	34, 35	35	35		35	35		35																35	35			35		
return_stmt'																				36		37						37	37	
var'									39	39	39	39	39	39	39	39	39	39	39	39	39		39			38	39			
expression																						40						40	40	
expression'													41	41	41	41	41	41		42			42							
relop													45	46	44	43	47	48												
arithmetic_expression																						49						49	49	
arithmetic_expression'									50	50			51	51	51	51	51	51		51	51		51				51			
addop									52	53																				
term																						54						54	54	
term'									56	56	55	55	56	56	56	56	56	56		56	56		56				56			
mulop											57	58																		
factor																						59						60	61	
call'																						62	63					62	62	
args																						64						64	64	
args_list																					65		66							

Figure 18. The parsing table for the resulting C Minus grammar.

Note that one cell, corresponding to M[*selection_stmt'*, else] has two results, this is because of the 'dangling else' ambiguity. In this case the correct derivation for the grammar is the one where M[*selection_stmt'*, else] = 34 so the cell is changed accordingly. The following unambiguous Parsing Table will be translated into code:

	else	if	input	int	output	return	void	while	+	-	*	/	>	>=	<	<=	==	!=	=	;	,	(	)	{	}	[	]	ID	NUM	$
program				1			1																							
declaration_list				2			2																							
declaration_list'				3			3																							4
declaration				5			6																							
declaration'																				7		8				7				
var_declaration'																				9						10				
params				11			12																							
param_list																					13		14							
param'																					16		16			15				
compound_stmt																								17						
local_declarations		19	19	18	19	19		19																19	19			19		
statement_list		20	20		20	20		20																20	21			20		
statement		24	27		28	26		25																23				22		
var_or_call_stmt																			29			30				29				
var_or_call									31	31	31	31	31	31	31	31	31	31		31	31	32	31			31	31			
selection_stmt		33																												
selection_stmt'	34	35	35		35	35		35																35	35			35		
return_stmt'																				36		37						37	37	
var'									39	39	39	39	39	39	39	39	39	39	39	39	39		39			38	39			
expression																						40						40	40	
expression'													41	41	41	41	41	41		42			42							
relop													45	46	44	43	47	48												
arithmetic_expression																						49						49	49	
arithmetic_expression'									50	50			51	51	51	51	51	51		51	51		51				51			
addop									52	53																				
term																						54						54	54	
term'									56	56	55	55	56	56	56	56	56	56		56	56		56				56			
mulop											57	58																		
factor																						59						60	61	
call'																						62	63					62	62	
args																						64						64	64	
args_list																					65		66							

Figure 19. The unambiguous Parsing Table for the resulting C Minus grammar.

Design

During the design phase, diagrams made during the development of the Lexical Analyser were updated for compatibility with the design made for this phase of compilation, including: Class Diagram, Sequence Diagram, and the pseudo code provided by the professor.

A diagram of the architecture for an LL(1) Parser provided by the professor, shown in **Figure 20**, was used to create classes for expanding the original class diagram:

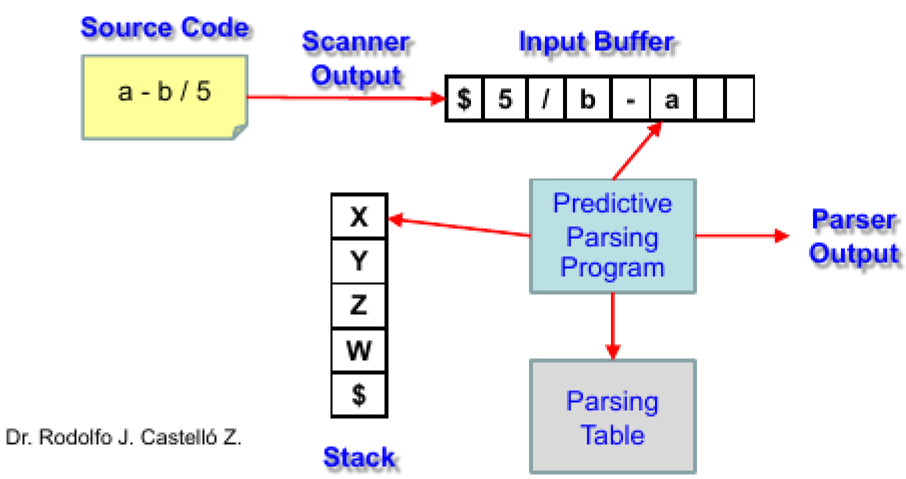

Figure 20. The basic architecture of an LL(1) Parser.

The diagram in **Figure 21** shall illustrate the interactions between the classes described previously; the classes for steps of compilation other than lexical analysis have been omitted as they are not relevant at this stage of compilation.

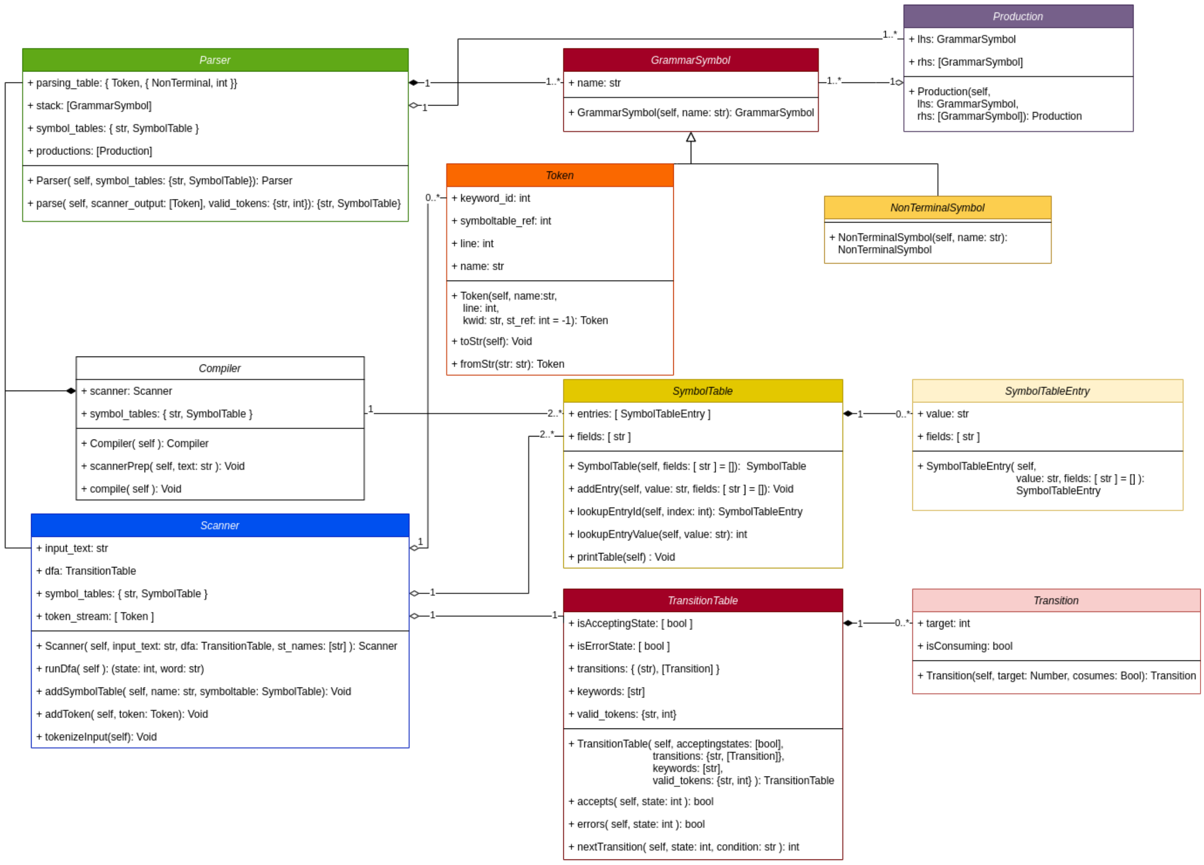

Figure 21. The entire Class Diagram for the compiler program, including updates.

As considered, the *Compiler* class is composed of a *Scanner* object, and now of a *Parser* object, too; a Production Class, GrammarSymbol class, and a NonTerminalSymbol class. Now, the Token class inherits from the GrammarSymbol class, along with NonTerminalSymbol, so that the stack of the parser can be treated as a homogenous list of GrammarSymbols. The list of productions from which the Parsing Table will take is a list of Production Objects whose right-hand-side is a list of GrammarSymbols: Token (terminal) objects and NonTerminalSymbol (non-terminal) objects that are pushed onto the stack.

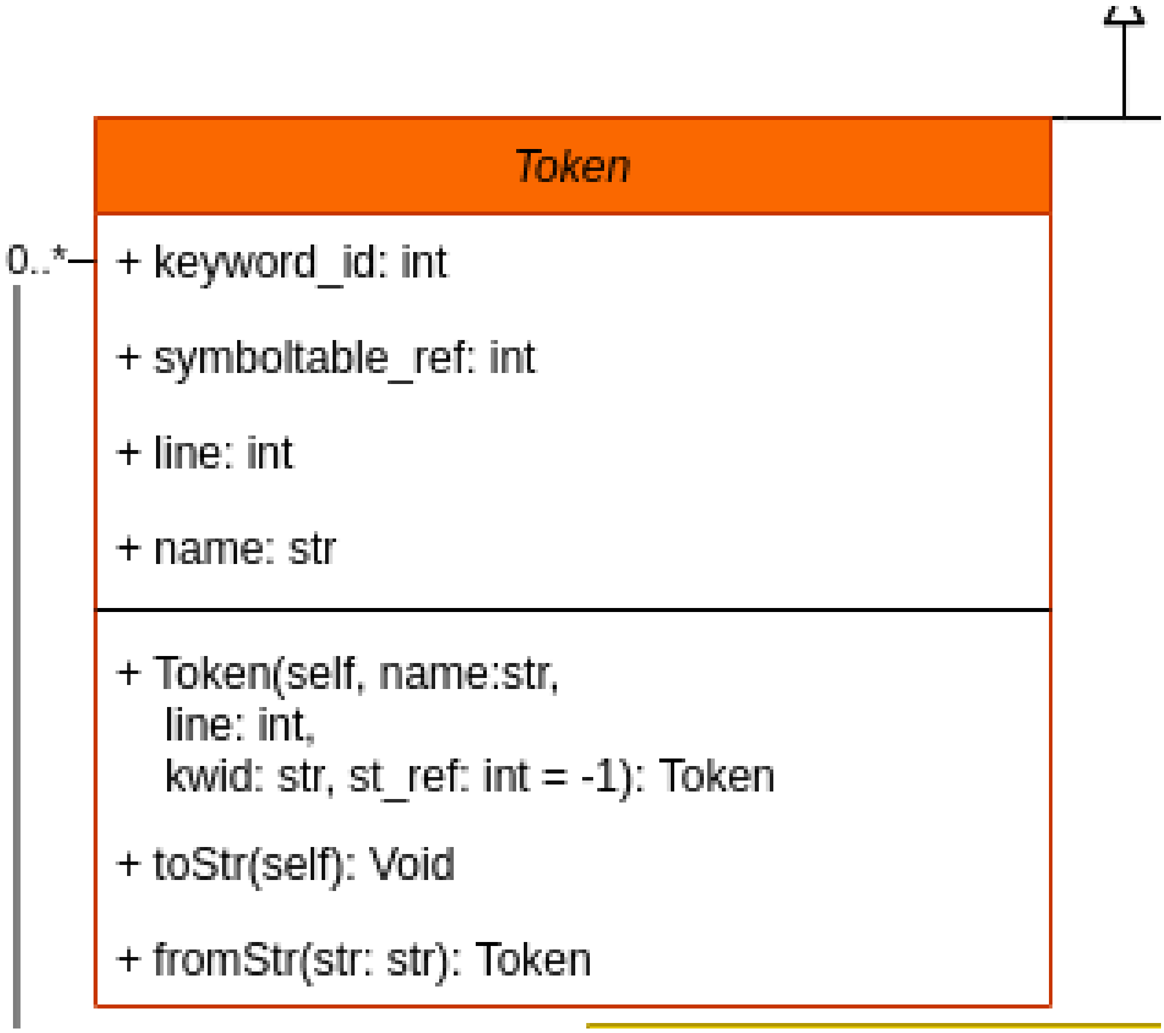

Figure 22. The updated Token class.

The Token class was updated to receive an additional two values in its constructor: the *name* field, and the *line* field. The *name* field serves as a shorthand when comparing the output from the scanner and the elements in the stack, as well as creating a point of comparison between other GrammarSymbol objects that are not necessarily Token objects. The *line* field is a new value passed from the scanner phase that allows the syntax analyser to know the line of the token in case an error is detected.

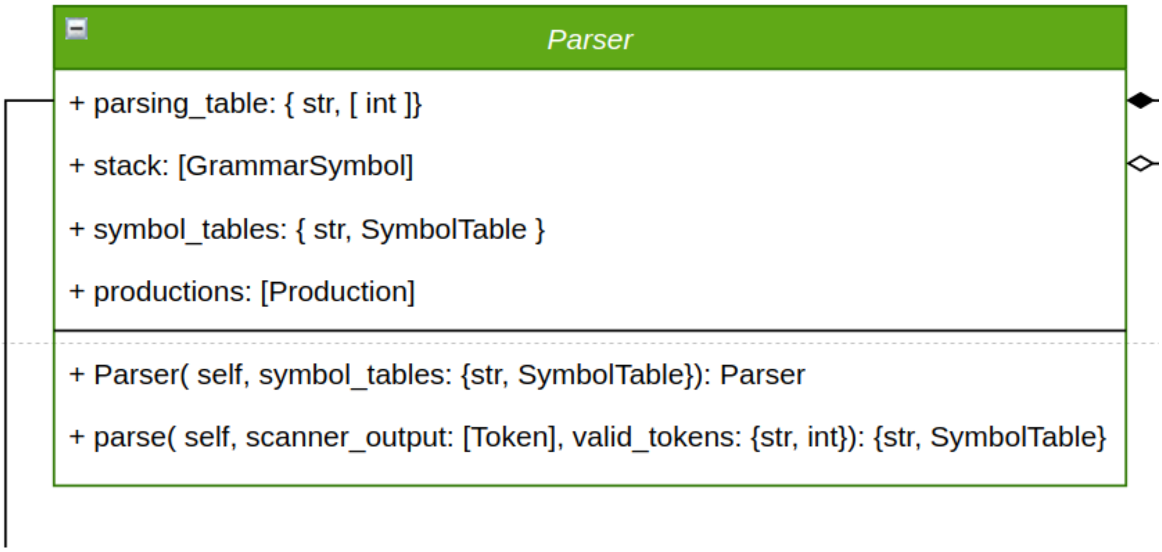

Figure 23. The Parser Class.

The Parser class contains the elements of a parser as described by the LL(1) parser algorithm: a stack, an input of Tokens, the parsing program in the form of the *parse* method, and the parsing table. The parsing table is a dictionary that receives the name of a NonTerminalSymbol in the form of a string, and with an index that corresponds to the Token ID - 1, returns a number corresponding to the number of a production. The Stack is a list which, as the parsing begins, is appended the grammar symbols ‘$’ and ‘program’ (the start symbol of the grammar). The symbol tables contain the information about the ID Tokens that have been found, and, during parsing, are updated with new information about them. The Production list is a list of productions, whose order corresponds to the output of the parsing table minus one (- 1), so that the parser knows which GrammarSymbols to add to the stack.

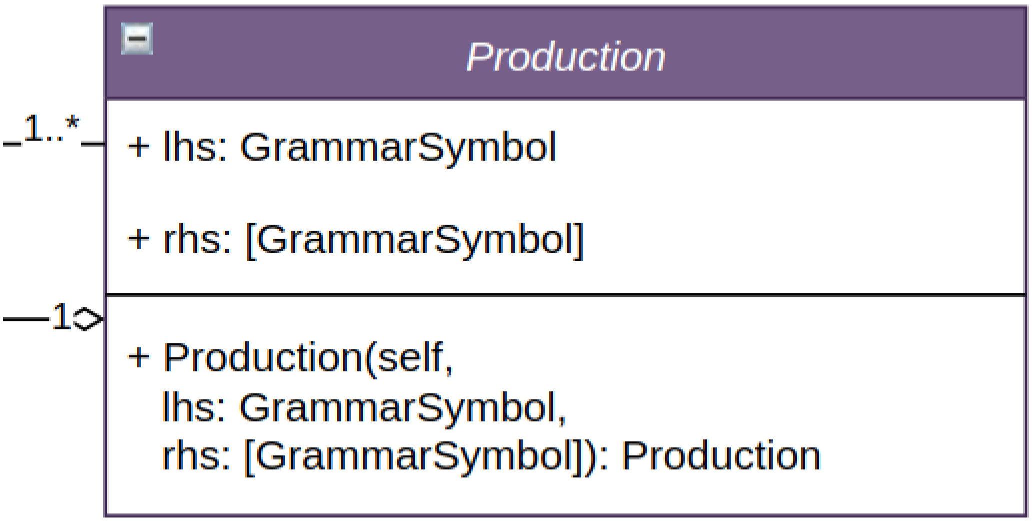

Figure 24. The Production class.

The production class serves as an abstraction for the productions of a grammar, in this case the list of productions from the resulting C Minus grammar are passed into code via the Production class.

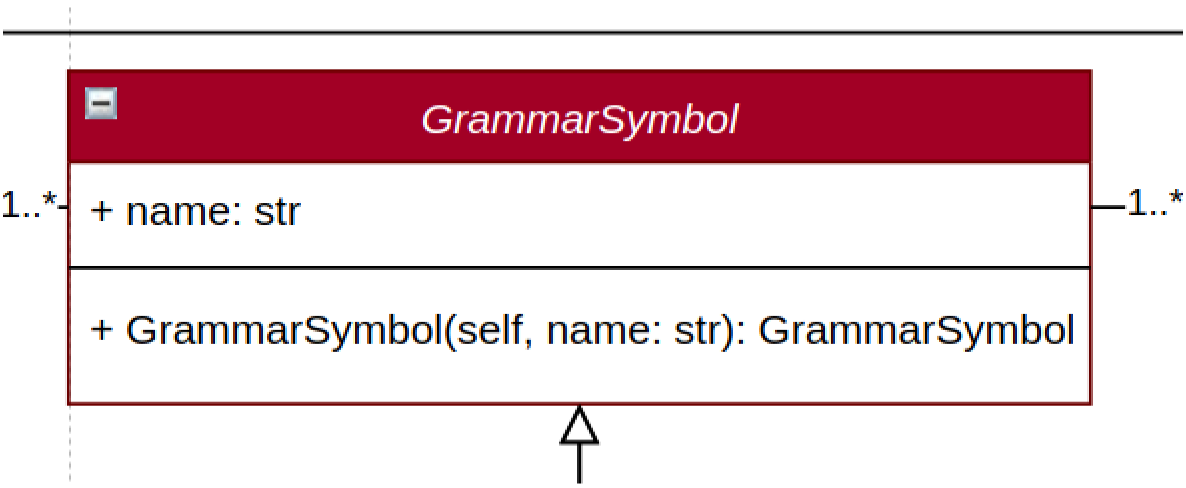

Figure 25. The GrammarSymbol class.

The GrammarSymbol class serves as an abstraction for all terminals, non-terminals, and pseudo-tokens that are used in the definition of a grammar. Pseudo tokens such as ‘$’ and ‘ε’ do not have a token id, since they are not Tokens, and are not non-terminal symbols, but can be represented as simple grammar symbols with their own unique name.

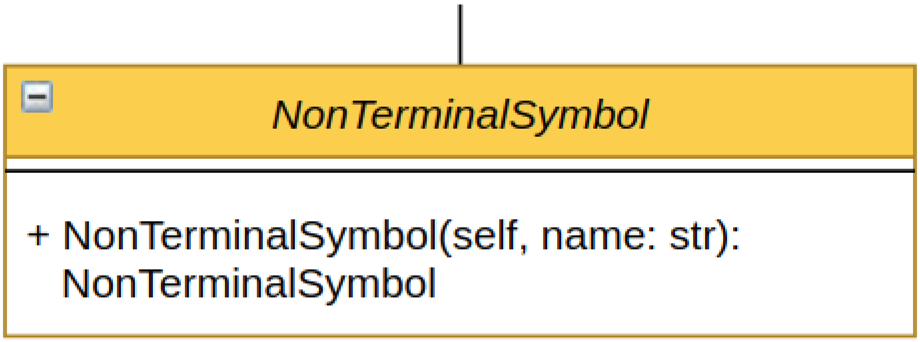

Figure 26. The NonTerminalSymbol class.

The NonTerminalSymbol class is a child class to the GrammarSymbol class, and its only purpose is to serve as an abstraction for the non-terminal symbols in the grammar. The *name* field is inherited from the GrammarSymbol class and is the only information needed from the non-terminal symbols themselves.

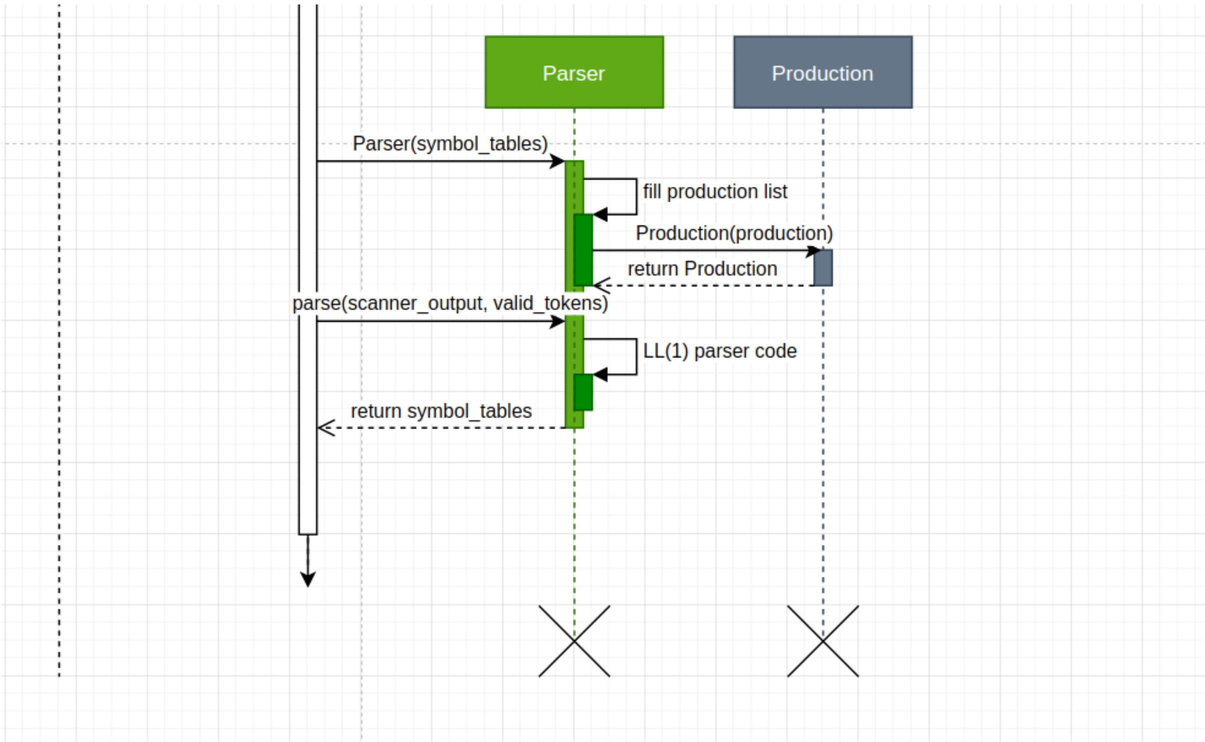

Figure 27. The update to the compilers' sequence diagram.

The section in **Figure 27** is added to the sequence diagram for the compiler, and depicts the behavior of classes involved.

Finally, the following is the pseudocode that was used for the implementation of the LL(1) parsing algorithm for the Syntactic Analyzer for the C Minus language:

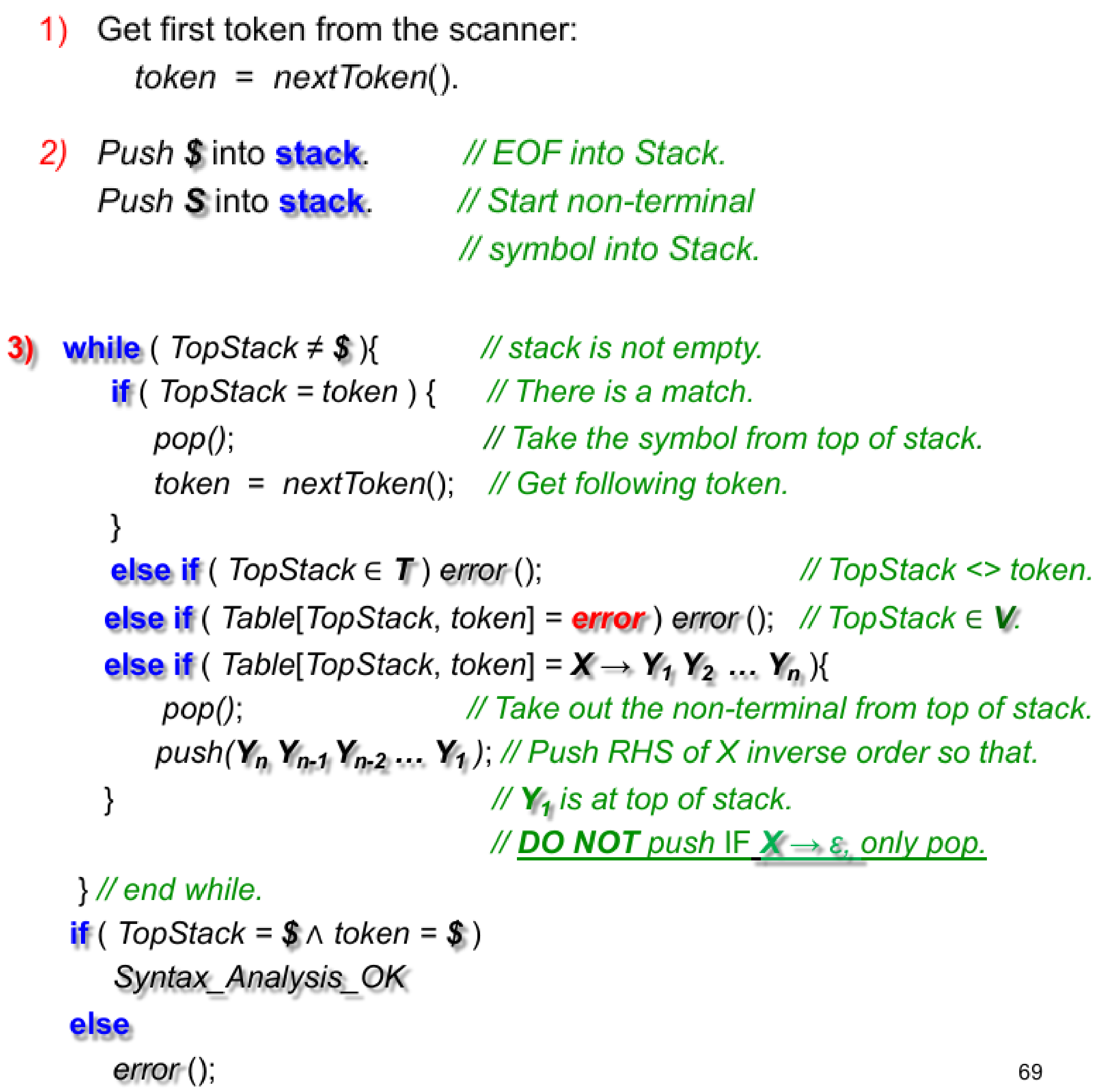

Figure 28. The pseudocode por the LL(1) parsing algorithm. [2]

Implementation

The following images will illustrate how the updates to the compiler were implemented:

```python
"""
GrammarSymbol Class
author: Victor Emmanuel Guerra Aguado

Description:
    The GrammarSymbol class provides an abstraction of the grammar symbols present in
    a Context Free Grammar.
Attributes:
    + name: str;
        The name of the GrammarSymbol object
"""
class GrammarSymbol:
    def __init__(self, name: str):
        self.name = name

    def __eq__(self, o: object):
        if isinstance(o, GrammarSymbol):
            return self.name == o.name
        return False

    """
    GrammarSymbol toStr method

    Description:
        Provides a way to transform the GrammarSymbol Object into a string of characters
        that can be printed onto the screen for visualization or written into a
        file for storage.

    toStr(self) -> str
    """
    def toStr(self) -> str:
        return f"({self.name})"

class NonTerminalSymbol(GrammarSymbol):
    def __init__(self, name: str):
        self.name = name
```

Figure 29. The definition of classes: Grammar Symbol and NonTerminalSymbol.

```python
from grammarSymbol import GrammarSymbol

"""
Production Class
author: Victor Emmanuel Guerra Aguado

Description:
    The Production Class provides a structure for the entries made into the list
    of Productions of the language grammar.

Attributes:
    + lhs: GrammarSymbol;
      The symbol at the left-hand-side of the production.
    + rhs: [GrammarSymbol];
      The list of symbols that appear at the right-hand-side of the production.
"""
class Production:
    def __init__(self, lhs: GrammarSymbol, rhs: [GrammarSymbol]):
        self.lhs = lhs
        self.rhs = rhs

```

Figure 30. The definition of Production class.

```python
from grammarSymbol import GrammarSymbol
"""
Token Class
author: Victor Emmanuel Guerra Aguado

Description:
    The Token Class provides a value object for the tokens created in the
    Lexical Analysis phase of a compiler.

Attributes:
    + name: str;
        The name of the token.
    + line: int;
        The line in which this token was detected during the lexical analysis
        phase.
    + keyword_id: int;
        The number that tells what type of valid word for the language the token
        represents.
    + symboltable_ref: int;
        The ID number that indicates an entry to a Symbol Table, if not an
        IDENTIFIER or NUMERICAL_CONST, it defaults to -1.

"""
class Token(GrammarSymbol):
    """
    Token Class Constructor

    __init__(self, kwid: int, st_ref: int) -> Token
    """
    def __init__(self, name: str, line: int, kwid: int, st_ref: int = None):
        self.name = name
        self.line = line
        self.keyword_id = kwid
        self.symboltable_ref = st_ref

```

Figure 31. The update to the Token class.

```
    """
    Compiler compile method

    Description:
        Executes all of the main components' methods to produce an output as
        indicated by the architecture diagram. Said output is usually passed
        as an argument as input to the next method.

    compile(self) -> None
    """
    def compile(self):

        # Lexical Analysis Phase
        self.scanner.tokenizeInput()
        token_stream = self.scanner.token_stream
        self.symbol_tables = self.scanner.symbol_tables

        token_stream.append(GrammarSymbol("$"))
        self.parser = Parser(self.symbol_tables)
        self.parser.parse(token_stream, self.scanner.dfa.valid_tokens)
```

Figure 32. The update to the compile method of class Compiler.

The rest of the code is in file: Parser.zip.

Testing

Test cases were developed around the requirements established during the analysis phase of development, namely:

1. Syntactic Errors that should be detected.
2. Semantic Errors that should be detected.
3. Additions to the Symbol Tables.

CMCTC-08

Description/Purpose:

To verify that the parser adequately parses a correct structure.

Test Script:

1. Provide the C Minus Language specification to the compiler.
2. Provide the CMCTC-08 test input source code.
3. Provide the expected results to the test function.
4. Run the compiler.
5. Compare the result with the expected result.

Test Input:

```
"int x[10];

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

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

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

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

```
void main(void){
        int i;
        i = 0;

        while(i<10){
        input x[i];
        i = i + 1;
        }

        sort(x,0,10);
        i = 0;
        while (i<10){
        output x[i];
        i = i + 1;
        }
} /* END of main() 戰勝利 */"
```

Expected Result:

"IDENTIFIER

| ID | VALUE | isVar|isFunc|dataType|noArgs|isGlobal|isLocal

0 | x | True | False | int | 0 | True | False

1 | miniloc | False | True | int | 3 | True | False

2 | a | True | False | int | 0 | False | True

3 | low | True | False | int | 0 | False | True

4 | high | True | False | int | 0 | False | True

5 | i | True | False | int | 0 | False | True

6 | k | True | False | int | 0 | False | True

7 | sort | False | True | void | 3 | True | False

8 | t | True | False | int | 0 | False | True

9 | main | False | True | void | 0 | True | False"

Actual Result:

IDENTIFIER

| ID | VALUE | isVar|isFunc|dataType|noArgs|isGlobal|isLocal

0 | x | True | False | int | 0 | True | False

1 | miniloc | False | True | int | 3 | True | False

2 | a | True | False | int | 0 | False | True

3 | low | True | False | int | 0 | False | True

4 | high | True | False | int | 0 | False | True

5 | i | True | False | int | 0 | False | True

6 | k | True | False | int | 0 | False | True

7 | sort | False | True | void | 3 | True | False

8 | t | True | False | int | 0 | False | True

9 | main | False | True | void | 0 | True | False

Evaluation: PASS

CMCTC-09

Description/Purpose:

To verify that the parser adequately recognizes and raises an error for: missing main declaration.

Test Script:

1. Provide the C Minus Language specification to the compiler.
2. Provide the CMCTC-09 test input source code.
3. Provide the expected results to the test function.
4. Run the compiler.
5. Compare the result with the expected result.

Test Input:

```
“void sort(void) {
   int i;
}”
```

Expected Result:

No main function was declared.

Actual Result:

Evaluation: PASS

CMCTC-10

Description/Purpose:

To verify that the parser adequately recognizes and raises an error for: main function is not the last declaration.

Test Script:

1. Provide the C Minus Language specification to the compiler.
2. Provide the CMCTC-10 test input source code.
3. Provide the expected results to the test function.
4. Run the compiler.
5. Compare the result with the expected result.

Test Input:

```
"void main(void) {
int i;
}
void sort(void) {
   int i;
}"
```

Expected Result:

main function is not last declaration.

Actual Result:

main function is not last declaration.

Evaluation: PASS

CMCTC-11

Description/Purpose:

To verify that the parser adequately recognizes and raises an error for: main function receives more than: 0 arguments.

Test Script:

1. Provide the C Minus Language specification to the compiler.
2. Provide the CMCTC-11 test input source code.

3. Provide the expected results to the test function.
4. Run the compiler.
5. Compare the result with the expected result.

Test Input:

“void main(int i, int a[]) {

int j;

}

”

Expected Result:

main function receives more than: 0 arguments.

Actual Result:

main function receives more than: 0 arguments.

Evaluation: **PASS**

CMCTC-12

Description/Purpose:

To verify that the parser adequately recognizes and raises an error for: a variable has been assigned a void type.

Test Script:

1. Provide the C Minus Language specification to the compiler.
2. Provide the CMCTC-12 test input source code.
3. Provide the expected results to the test function.
4. Run the compiler.
5. Compare the result with the expected result.

Test Input:

“void main(void) {

void i;

}

”

Expected Result:

main function receives more than: 0 arguments.

Actual Result:

main function receives more than: 0 arguments.

Evaluation: PASS

CMCTC-13

Description/Purpose:

To verify that the parser adequately recognizes and raises an error for: a variable or function has been used before declaration.

Test Script:

1. Provide the C Minus Language specification to the compiler.
2. Provide the CMCTC-13 test input source code.
3. Provide the expected results to the test function.
4. Run the compiler.
5. Compare the result with the expected result.

Test Input:

“void main(void) {
sort();
}
”

Expected Result:

Use of Identifier: sort before declaration. In line: 2

Actual Result:

Use of Identifier: sort before declaration. In line: 2

Evaluation: PASS

CMCTC-14

Description/Purpose:

To verify that the parser adequately recognizes and raises an error for: unexpected characters and tokens.

Test Script:

1. Provide the C Minus Language specification to the compiler.
2. Provide the CMCTC-14 test input source code.
3. Provide the expected results to the test function.
4. Run the compiler.
5. Compare the result with the expected result.

Test Input:

```
“void main void){
    int i;
    i = 0;


    }
”
```

Expected Result:

Expected a: (, received: void on line 1

Actual Result:

Use of Identifier: sort before declaration. In line: 2

Evaluation: **PASS**

References

1. R. Castelló, Class Lecture, Topic: “Chapter 3 - Syntax Analysis - Part 1.” TC3048, School of Engineering and Science, ITESM, Chihuahua, Chih, April, 2022.
2. R. Castelló, Class Lecture, Topic: “Chapter 4 - Syntax Analysis - Part 2.” TC3048, School of Engineering and Science, ITESM, Chihuahua, Chih, April, 2022.
3. Thomas Hamilton, “How to Write Test Cases: Sample Template with Examples”, *Guru99*, 2 April 2022 [journal on-line]; available from https://www.guru99.com/test-case.html; Internet; accessed 18 April 2022.