C-- Syntax Analyzer

Gerardo Granados Aldaz

June 24, 2022

Table of Contents

1. Introduction

1.1 Summary

This document will detail the modifications made to the C-- grammar to comply with some of the semantic specifications, define which rules lie outside of the scope of the Syntax Analyzer (Parser), and show the design and development process of the system.

1.2 Notation

To develop the Parser, a Context-Free Grammar was provided. A Context-Free Grammar can be defined as G = (V, T, P, S) where V is the set of variables, T is the set of terminal symbols, P is the set of productions, and S is the starting symbol for the Grammar. The Grammar then, defines the structure of a language, in this case, the C-- language.

2. Analysis

2.1 Semantic Requirements

The specification document comes with several semantic requirements the C-- language MUST comply with. Since the application of semantic rules lies outside the scope of the Parser, some of these rules will not be implemented in the parser. However, other rules can be implemented by modifying the CFG. The rules implemented and their corresponding changes to the CFG are the following:

1. "The last declaration in a program MUST be a function declaration of the form void main(void)."
    a. Production "*program* → *declaration_list*" turns into "*program* → *declaration_list* **void ID (void)** *compound_stmt* | **void ID (void)** *compound_stmt*"
    b. This introduces indirect LF with *declaration*, so it will be handled in the parser
2. "In a variable declaration, only the type_specifier int can be used, void is for function declarations."
    a. In productions "var_declaration → *type_specifier* **ID ;** | *type_specifier* **ID [NUM] ;**" *type_specifier* must be changed to **int**.
    b. "var_declaration → **int ID ; | int ID [NUM] ;**"
3. "There are no parameters of type function."
    a. Productions "*param* → *type_specifier* **ID** | *type_specifier* **ID [ ]**" turns into "*param* → **int ID | int ID [ ]**"

2.2 Cleaning the Grammar

The previous modifications to the CFG introduce a new set of problems on top of the existing issues in the CFG that will make it impossible for a parser using a top-down approach to finish. These issues need to be fixed by eliminating Left-Factors (LF), Left-Recursion (LR) , and replacing the first element in a production with its productions if it is a non-terminal. Then, once the grammar has been sufficiently "cleaned" it is simplified by removing Unit-Productions (UP), Useless Symbols (US), and ε-productions (εP) wherever possible without creating new problems.

A clever and useful trick to deal with LR is that whenever α = β or β = {ε}, the non-terminal symbol in the production causing the LR can be moved to the end of the production. This was done in the following cases:

2. *declaration_list*	→ *declaration_list declaration* | *declaration*	(LR)	α=β
declaration_list	→ *declaration declaration_list* | *declaration*	(LF introduced)	
declaration_list	→ *declaration declaration_list* | ε		

11. *local_declarations* → *local_declarations var_declarations* | ε (LR) β={ ε }

local_declarations → *var_declarations local_declarations* | ε

12. *statement_ list* → *statement_list statement* | ε (LR) β={ ε }

statement_ list → *statement statement_list* | ε

The other cases of LR were solved normally with the formula A →Aα|β into A→βA' where A'→αA'|ε

8. *param_list* → *param_list* **,** *param* | *param* (LR)

param_list → *param param_list'*
param_list' → **,** *param param_list'* | ε

24. *arithmetic_expression* → *arithmetic_expression addop term* | *term* (LR)

arithmetic_expression → *term arithmetic_expression'*
arithmetic_expression' → *addop term arithmetic_expression'* | ε

26. *term* → *term mulop factor* | *factor* (LR)

term → factor term'
term' → mulop factor term' | ε

31. *args_list* → *args_list* **,** *arithmetic_expression* | *arithmetic_expression* (LR)

args_list → *arithmetic_expression args_list'*
args_list' → **,** *arithmetic_expression args_list'* | ε

With LR removed, LF was still a problem. This was removed with the formula **A → αβ1 / αβ2** into **A → αA'** where **A' → β1 / αβ2**:

4. *var_declaration* → **int ID ; | int ID [NUM] ;** (LF)

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

9. *param* → **int ID | int ID [ ]** (LF)

param → **int ID** *param'*
param' → **[ ]** | ε

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

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

18. *return_stmt* → **return ; | return** *expression* **;** (LF)

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

21. *var* → **ID | ID [** *arithmetic_expression* **]** (LF)

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

22. *expression* → *arithmetic_expression relop arithmetic_expression* | *arithmetic_expression*

expression → *arithmetic_expression expression'*
expression' → *relop arithmetic_expression expression'* | ε

Removing εP's would create an infinite loop of LF's and LR's. The only εP that can be removed is in args:

21. *var* → *args_list* | ε (εP)

call → **ID (** *args* **) | ID ( )** (LF introduced)
args → *args_list* (UP introduced)

call → **ID (** *call'*
call' → *args* **) |)**
args → *arithmetic_expression args_list'*

Now productions starting with a non-terminal symbol will be replaced with its productions so that there is at most one production per symbol that begins with a non-terminal symbol. Some Unit Productions will still be present, these will be dealt with in the next step.

3. *declaration* → *var_declaration* | *fun_declaration*

declaration → **int ID** *var_declaration'* | *fun_declaration*

13. *statement* → **ID** *var'* **=** *expression* **;** | **ID** **(** *call'* **;** | **{** *local_declarations statement_list* **}** | **if (** *expression* **)** *statement selection_stmt'* | **while (** *expression* **)** *statement* | **return** *return_stmt'* | **input** *var* **;** | **output** *expression* **;**

statement → **ID** *statement'* | **{** *local_declarations statement_list* **}** | **if (** *expression* **)** *statement selection_stmt'* | **while (** *expression* **)** *statement* | **return** *return_stmt'* | **input** *var* **;** | **output** *expression* **;**
statement' → *var'* **=** *expression* **;** | **(** *call'* **;**

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

factor → **(** *arithmetic_expression* **)** | **ID** *var'* | **ID** **(** *call'* | **NUM**

factor → **(** *arithmetic_expression* **)** | **ID** *factor'* | **NUM**
factor' → *var'* | **(** *call'*

Next, Unit Productions are removed. This was done with a script to reduce hassle:

program→declaration_list
declaration_list->declaration declaration_list
declaration→int ID var_declaration' | type_specifier ID (params) compound_stmt
var_declaration→int ID var_declaration'
var_declaration'→; | [NUM] ;
type_specifier→int | void
fun_declaration→type_specifier ID (params) compound_stmt
params→param param_list' | void
param_list→param param_list'
param_list'→, param param_list' | ε
param→int ID param'
param'→[ ] | ε
compound_stmt→{ local_declarations statement_list }
local_declarations→var_declarations local_declarations | ε
statement_list→statement statement_list | ε
statement→ID statement' | { local_declarations statement_list } | if (expression) statement selection_stmt' | while (expression) statement | return return_stmt' | input var ; | output expression ;
statement'→var' = expression ; | (call' ;
assignment_stmt→var = expression ;
call_stmt→call ;
selection_stmt→if (expression) statement selection_stmt'
selection_stmt'→else statement | ε
iteration_stmt→while (expression) statement
return_stmt→return return_stmt'
return_stmt'→; | expression ;
input_stmt→input var ;
output_stmt→output expression ;
var→ID var'
var'→[arithmetic_expression] | ε
expression→arithmetic_expression expression'
expression'→relop arithmetic_expression expression' | ε
relop→<= | < | > | >= | == | !=
arithmetic_expression→term arithmetic_expression'

arithmetic_expression'→addop term arithmetic_expression' | ε
addop→+ | -
term→factor term'
term'→mulop factor term' | ε
mulop→* | /
factor→(arithmetic_expression) | ID factor' | NUM
factor'→var' | (call'
call→ID (call'
call'→args) |)
args→arithmetic_expression args_list'
args_list→arithmetic_expression args_list'
args_list'→, arithmetic_expression args_list' | ε

Finally, after eliminating UP's, some symbols became useless. The CFG was changed to reflect that, the following symbols were removed:

- fun_declaration
- param_list' (param_list' is now param_list)
- assignment_stmt
- call_stmt
- selection_stmt' (is now selection_stmt)
- iteration_stmt
- return_stmt' (is now return_stmt)
- input_stmt
- output_stmt
- call' (is now call)
- args_list' (is now args_list)

Leaving a clean grammar with 34 different non-terminal symbols:

program->declaration declaration_list
declaration_list->declaration declaration_list
declaration->int ID declaration' | void ID (params) compound_stmt
declaration'->; | [NUM] ; | (params) compound_stmt
var_declaration->int ID var_declaration'
var_declaration'->; | [NUM] ;
params->param param_list | void
param_list->, param param_list | ε
param->int ID param'
param'->[ ] | ε
compound_stmt->{ local_declarations statement_list }
local_declarations->var_declaration local_declarations | ε
statement_list->statement statement_list | ε
statement->ID statement' | { local_declarations statement_list } | if (expression) statement selection_stmt | while (expression) statement | return return_stmt | input var ; | output expression ;
statement'->var' = expression ; | (call ;
selection_stmt->else statement | ε
return_stmt->; | expression ;
var->ID var'
var'->[arithmetic_expression] | ε
expression->arithmetic_expression expression'
expression'->relop arithmetic_expression expression' | ε
relop-><= | < | > | >= | == | !=
arithmetic_expression->term arithmetic_expression'
arithmetic_expression'->addop term arithmetic_expression' | ε

addop->+ | -
term->factor term'
term'->mulop factor term' | ε
mulop->* | /
factor->(arithmetic_expression) | ID factor' | NUM
factor'->[arithmetic_expression] | ε | (call
call->args) |)
args->arithmetic_expression args_list
args_list->, arithmetic_expression args_list | ε

2.3 First, Follow, and First$^+$ Sets

2.3.1 First Sets

The FIRST set is the set of terminal symbols and possible ε that can appear as the first terminal in some string derived from any non-terminal symbol. If the first symbol in a production is a non-terminal, the FIRST set is the union of that non-terminal's FIRST set.

This was done with a script. The resulting sets are:

FIRST(program) = {void, int}
FIRST(declaration_list) = {void, int}
FIRST(declaration) = {void, int}
FIRST(declaration') = {[, ;, (}
FIRST(var_declaration) = {int}
FIRST(var_declaration') = {[, ;}
FIRST(params) = {void, int}
FIRST(param_list) = {,, ε}
FIRST(param) = {int}
FIRST(param') = {[, ε}
FIRST(compound_stmt) = {{}
FIRST(local_declarations) = {int, ε}
FIRST(statement_list) = {while, ID, input, return, output, if, ε, {}
FIRST(statement) = {while, ID, input, {, return, output, if}
FIRST(statement') = {[, (, ε}
FIRST(selection_stmt) = {ε, else}
FIRST(return_stmt) = {NUM, ID, ;, (}
FIRST(var) = {ID}
FIRST(var') = {[}
FIRST(expression) = {NUM, ID, (}
FIRST(expression') = {>, <=, ε, ==, >=, !=, <}
FIRST(relop) = {>, >=, !=, <, <=, ==}
FIRST(arithmetic_expression) = {NUM, ID, (}
FIRST(arithmetic_expression') = {-, +, ε}
FIRST(addop) = {-, +}
FIRST(term) = {NUM, ID, (}
FIRST(term') = {/, *, ε}
FIRST(mulop) = {/, *}
FIRST(factor) = {NUM, ID, (}

FIRST(factor') = {[, (, ε}
FIRST(call) = {)}
FIRST(args) = {NUM, ID, (}
FIRST(args_list) = {,, ε}

2.3.2 Follow Sets

The FIRST set is the set of terminal symbols and possible ε that can appear as the first terminal in some string derived from any non-terminal symbol. If the first symbol in a production is a non-terminal, the FIRST set is the union of that non-terminal's FIRST set.

This was done with a script to reduce hassle. The resulting sets are:

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

2.3.3 First+ Sets

Here the dangling else problem is dealt with by removing the **else** keyword from FIRST+(*selection_stmt* → ε).

FIRST+(program->declaration declaration_list) = {void, int}

FIRST+(declaration_list->declaration declaration_list) = {void, int}

FIRST+(declaration->int ID declaration') = {int}

FIRST+(declaration->void ID (params) compound_stmt) = {void}

FIRST+(declaration'->;) = {;}

FIRST+(declaration'->[NUM] ;) = {[}

FIRST+(declaration'->(params) compound_stmt) = {(}

FIRST+(var_declaration->int ID var_declaration') = {int}

FIRST+(var_declaration'->;) = {;}

FIRST+(var_declaration'->[NUM] ;) = {[}

FIRST+(params->param param_list) = {int}

FIRST+(params->void) = {void}

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) = {while, ID, input, }, {, return, output, if, ε}

FIRST+(statement_list->statement statement_list) = {while, ID, input, return, output, if, {}

FIRST+(statement_list->ε) = {}, ε}

FIRST+(statement->ID statement') = {ID}

FIRST+(statement->{ local_declarations statement_list }) = {{}

FIRST+(statement->if (expression) statement selection_stmt) = {if}

FIRST+(statement->while (expression) statement) = {while}

FIRST+(statement->return return_stmt) = {return}

FIRST+(statement->input var ;) = {input}

FIRST+(statement->output expression ;) = {output}

FIRST+(statement'->var' = expression ;) = {[, =}

FIRST+(statement'->(call ;) = {(}

FIRST+(selection_stmt->else statement) = {else}

FIRST+(selection_stmt->ε) = {while, ID, input, }, {, return, output, if, ε}

FIRST+(return_stmt->;) = {;}

FIRST+(return_stmt->expression ;) = {NUM, ID, (}

FIRST+(var->ID var') = {ID}

FIRST+(var'->[arithmetic_expression]) = {[}

FIRST+(var'->ε) = {;, ε, =}

FIRST+(expression->arithmetic_expression expression') = {NUM, ID, (}

FIRST+(expression'->relop arithmetic_expression expression') = {>, <=, ==, >=, !=, <}

FIRST+(expression'->ε) = {;,), ε}

FIRST+(relop-><=) = {<=}

FIRST+(relop-><) = {<}

FIRST+(relop->>) = {>}

FIRST+(relop->>=) = {>=}

FIRST+(relop->==) = {==}

FIRST+(relop->!=) = {!=}

FIRST+(arithmetic_expression->term arithmetic_expression') = {NUM, ID, (}

FIRST+(arithmetic_expression'->addop term arithmetic_expression') = {-, +}

FIRST+(arithmetic_expression'->ε) = {>,], >=, ;, !=, <, <=, ,, ==,), ε}

FIRST+(addop->+) = {+}

FIRST+(addop->-) = {-}

FIRST+(term->factor term') = {NUM, ID, (}

FIRST+(term'->mulop factor term') = {/, *}

FIRST+(term'->ε) = {>,], -, +, >=, !=, <, ;, <=, ,, ==,), ε}

FIRST+(mulop->*) = {*}
FIRST+(mulop->/) = {/}
FIRST+(factor->(arithmetic_expression)) = {(}
FIRST+(factor->ID factor') = {ID}
FIRST+(factor->NUM) = {NUM}
FIRST+(factor'->[arithmetic_expression]) = {[}
FIRST+(factor'->ε) = {>,], -, +, >=, !=, <, ;, *, <=, ,, /, ==,), ε}
FIRST+(factor'->(call) = {(}
FIRST+(call->args)) = {NUM, ID, (}
FIRST+(call->)) = {)}
FIRST+(args->arithmetic_expression args_list) = {NUM, ID, (}
FIRST+(args_list->, arithmetic_expression args_list) = {,}
FIRST+(args_list->ε) = {), ε}

2.4 Requirements

The Parser shall provide the following **outputs**:
1. Corresponding Syntactical Errors.
2. The update of the corresponding Symbol Tables.

The Parser must include the following **deliverables**:
1. The correct grammar for C Minus in order to generate a Top-Down Predictive Parser.
2. Symbol Table management:
    a. Description of semantic aspects that were updated during the syntax analysis.
3. Error messages generated by the Parser.
4. Example of the Parser Outputs

3. Design

3.1 Parser Table

With the sets made during the Analysis phasa a Parse Table is created and will be used to traverse the tokens using the LL(1) Parsing Algorithm. In the table non-terminal symbols are rows and terminal symbols are columns. The cell value represents the production number it will lead to.

This table is made with the following algorithm:

Figure 1: Parse Table creation pseudocode

```
for each non-terminal X do:
    for each terminal a do:   // initialize table.
        Table[X, a] = error;   // empty cells will be errors.
    end
    for each production p  X → β  do:
        for each terminal w ∈ First+(X → β ) - ε do:
            Table[X, w] = p;
        end
        if $ ∈ First+(X → β ) then
            Table[X, $] = p;
    end
end
```

Once executed it creates the following table:

Figure 2: Parse Table

non-terminals	!=	(	)	*	+	,	-	/	;	<	<=	=	==	>	>=	ID	NUM	[	]	else	if	input	int	output	return	void	while	{	}	$
program	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	1	ERROR	ERROR	1	ERROR	ERROR	ERROR	2
declaration_list	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	3	ERROR	ERROR	3	ERROR	ERROR	ERROR	4
declaration	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	5	ERROR	ERROR	6	ERROR	ERROR	ERROR	ERROR
declaration'	ERROR	9	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	7	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	8	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
var_declaration	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	10	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
var_declaration'	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	11	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	12	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
params	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	13	ERROR	ERROR	14	ERROR	ERROR	ERROR	ERROR
param_list	ERROR	ERROR	16	ERROR	ERROR	15	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
param	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	17	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
param'	ERROR	ERROR	19	ERROR	ERROR	19	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	18	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
compound_stmt	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	20	ERROR	ERROR
local_declarations	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	22	ERROR	ERROR	ERROR	ERROR	22	22	21	22	22	ERROR	22	22	22	ERROR
statement_list	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	23	ERROR	ERROR	ERROR	ERROR	23	23	ERROR	23	23	ERROR	23	23	24	ERROR
statement	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	25	ERROR	ERROR	ERROR	ERROR	27	30	ERROR	31	29	ERROR	28	26	ERROR	ERROR
statement'	ERROR	33	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	32	ERROR	ERROR	ERROR	ERROR	ERROR	32	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
selection_stmt	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	35	ERROR	ERROR	ERROR	34	35	35	ERROR	35	35	ERROR	35	35	35	ERROR
return_stmt	ERROR	37	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	36	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	37	37	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
var	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	38	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
var'	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	40	ERROR	ERROR	40	ERROR	ERROR	ERROR	ERROR	ERROR	39	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
expression	ERROR	41	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	41	41	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
expression'	42	ERROR	43	ERROR	ERROR	ERROR	ERROR	ERROR	43	42	42	ERROR	42	42	42	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
relop	49	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	45	44	ERROR	48	46	47	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
arithmetic_expression	ERROR	50	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	50	50	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
arithmetic_expression'	52	ERROR	52	ERROR	51	52	51	ERROR	52	52	52	ERROR	52	52	52	ERROR	ERROR	ERROR	52	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
addop	ERROR	ERROR	ERROR	ERROR	53	ERROR	54	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
term	ERROR	55	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	55	55	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
term'	57	ERROR	57	56	57	57	57	56	57	57	57	ERROR	57	57	57	ERROR	ERROR	ERROR	57	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
mulop	ERROR	ERROR	ERROR	58	ERROR	ERROR	ERROR	59	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
factor	ERROR	60	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	61	62	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
factor'	64	65	64	64	64	64	64	64	64	64	64	ERROR	64	64	64	ERROR	ERROR	63	64	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
call	ERROR	66	67	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	66	66	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
args	ERROR	68	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	68	68	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
args_list	ERROR	ERROR	70	ERROR	ERROR	69	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

This table will be used in the implementation of the LL(1) Parsing Algorithm.

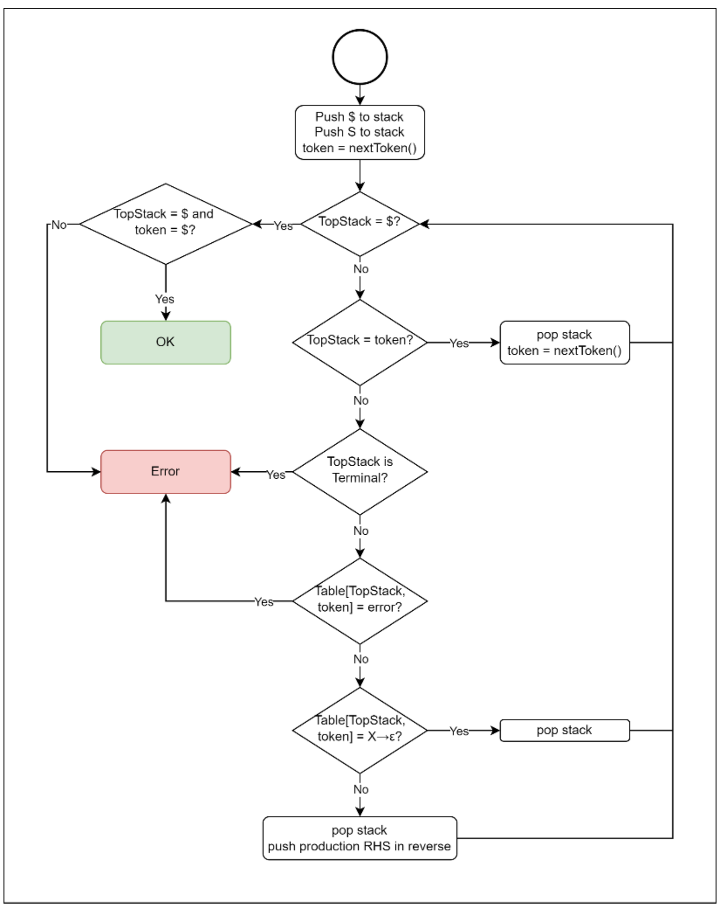
3.2 LL(1) Parsing Algorithm

The LL(1) Parsing Algorithm implements a stack initialized with the $ token and the starting symbol in the grammar. The algorithm will navigate the parse table with the input tokens until it matches a terminal symbol.

Figure 3: LL(1) Parsing Algorithm Pseudocode

```
while ( TopStack ≠ $ ){           // stack is not empty.
    if ( TopStack = token ) {     // There is a match.
        pop();                    // Take the symbol from top of stack.
        token = nextToken();      // Get following token.
    }
    else if ( TopStack ∈ T ) error ();                        // TopStack <> token.
    else if ( Table[TopStack, token] = error ) error ();      // TopStack ∈ V.
    else if ( Table[TopStack, token] = X → Y1 Y2 … Yn ){
        pop();                    // Take out the non-terminal from top of stack.
        push(Yn Yn-1 Yn-2 … Y1 ); // Push RHS of X inverse order so that.
    }                             // Y1 is at top of stack.
                                  // DO NOT push IF X → ε, only pop.
} // end while.
if ( TopStack = $ ∧ token = $ )
    Syntax_Analysis_OK
else
    error ();
```

Figure 4: LL(1) Parsing Algorithm Flow Diagram



3.3 Error Handling

The most important design decisions were made when handling errors. Since the LL(1) Parsing Algorithm does not provide an easy way to handle errors, different data structures and methods were created to handle errors.

3.4 Semantic Errors

Semantic errors were handled by evaluating tokens and adding them to the preexisting symbol table after modifying it to be a list of lists that would hold the identifier's name, type (function or variable), and scope (global or local) if variable or return type (void or int) if function. This made it possible to ensure that only one main function was declared that it was of void return type with void parameters, and that it was the last function in the program, effectively satisfying the Semantic Requirement 1.

Using the same symbol table it was possible to verify when calling a function that a parameter was not a function, satisfying Semantic Requirement 3, and that only **int** could be used to define variables, satisfying Semantic Requirement 2.

Additionally, a set of tuples was implemented to keep track of any variables in the scope of the running program increasing by 1 the level of scope each time an open bracket **{** was read, and decreasing and deleting all variables in the previous scope when a closed bracket **}** was read. This ensures that variables declared within the scope of an **if** statement, for example, would only be usable within that **if** statement.

3.5 Syntax Errors

Syntax Errors were the most tedious to implement. The approach was to find the context in which tokens could be used in the Parser Table and to specify cases for those that were used the least. This way a descriptive error message could be created for common errors. Next, for those that came from non-terminals, their context was analyzed, and they got their own descriptive error message. In the cases where the parser expected a terminal and got another, each production was analyzed to recognize their meaning giving each production their own error message.

There are in total over 100 different error messages.

4. Implementation

The breadth of the implementation was done in implementing the semantic and syntax errors and cleaning up the grammar.

4.1 Grammar helper Scripts

Since the LL(1) Parsing Algorithm is simple enough and the process of cleaning the grammar would be very slow to do by hand after every small change, scripts were developed to speed up the process. These are those methods.

```python
def get_grammar_from_txt(txt_name: str) -> tuple:
    """
    Turns a .txt file in the format "v1->p1 | ... | pn" into a dictionary for local
processing.

    args
        txt_name: location of the .txt file

    returns
        grammar: dict where key is V and values are P
        non_terminals: a set representing V
    """
    f = open(txt_name, "r")
    productions_str = f.read().split("\n")  # Arr with each V and its productions
    f.close()

    grammar = {}    # Empty dict for storing
    non_terminals = set()   # Set for non_terminals
    grammar_symbols = set()

    for entry in productions_str:
        if entry == "":  # Empty line edge case
            continue
        # Separates into non-terminal and productions
        prod_arr = entry.split("->")
        key = prod_arr[0]
        non_terminals.add(key)
        productions = prod_arr[1].split("|")
        values = []
        for prod in productions:
            prod = prod.strip()  # Remove leading and trailing whitespace
            RH_symbols = prod.split(" ")  # Split into array separated by " "
            values.append(RH_symbols)
            grammar_symbols.update(RH_symbols)
        grammar[key] = values   # Insert into grammar

    terminals = grammar_symbols.difference(non_terminals)
    terminals.remove("ε")
    return grammar, non_terminals, terminals
```

```python
def remove_unit_productions(grammar: dict, non_terminals: set) -> dict:
    """
    Removes all unit productions in grammar.

    args
        grammar: previously built grammar dictionary
        non_terminals: set of all non-terminals in grammar

    returns
        grammar: the grammar free of all unit productions
    """
    removed = True  # bool to confirm any change was made
    while removed:
        removed = False  # set to false to avoid infinite loop
        for key in grammar:
            # num of prod and prod to insert in place
            for i, prod in enumerate(grammar[key]):
                # len == 1 and prod in non_temrinal when only 1 symbol is in
present and it is part of V
                if len(prod) == 1 and prod[0] in non_terminals:
                    # replaces non-terminal with its productions
                    for replace in grammar[prod[0]][::-1]:
                        grammar[key].insert(i+1, replace)
                    grammar[key].pop(i)  # removes non-terminal
                    removed = True  # set to true to note a change was made
    return grammar
```

```python
def get_first_sets(grammar: dict, non_terminals: set) -> dict:
    """
    Gets first sets for each non-terminal symbol.
    Goes in reverse order from keys to ensure other symbols have their first() sets
complete.

    args
        grammar: previously built grammar dictionary
        non_terminals: set of all non-terminals in grammar

    returns
        first_sets: dictionary of sets with each symbol's first set
    """
    first_sets = dict()  # Dictionary with first sets

    pending = set()  # Set with non-terminals pending to find first sets
    # Iterate through keys in reverse order
    # move in reverse order to reduce pending nt's
```

```
    for symbol in list(grammar.keys())[::-1]:
        current_set = set()
        for production in grammar[symbol]:
            first_symbol = production[0]
            if first_symbol in non_terminals:   # if nt, first = first(nt)
                if first_symbol in first_sets.keys():   # ensure first(nt) has been
calculated before
                    # first(X) = first(X) ∪ first(Y)
                    current_set = current_set.union(first_sets[first_symbol])
                else:
                    # if first(nt) hasn't been calculated, add to pending
                    pending.add(symbol)
            else:
                current_set.add(first_symbol)   # add terminal symbol

        first_sets[symbol] = current_set    # store current set in dict

    for symbol in pending:  # repeat for all pending nt's
        current_set = set()
        for production in grammar[symbol]:
            first_symbol = production[0]
            if first_symbol in non_terminals:
                if first_symbol in first_sets.keys():
                    current_set = current_set.union(first_sets[first_symbol])
                else:
                    pending.add(symbol)
            else:
                current_set.add(first_symbol)
        first_sets[symbol] = current_set

    return first_sets
```

```
def get_follow_sets(grammar: dict, non_terminals: set, first_sets: dict) -> dict:
    """
    Gets follow sets for each non-terminal symbol.

    args
        grammar: previously built grammar dictionary
        non_terminals: set of all non-terminals in grammar
        first_sets: dictionary of sets with each symbol's first set

    returns
        follow: dictionary of sets with each symbol's follow set
    """
    follow = dict()
```

```
    # initialize empty follow sets
    for symbol in non_terminals:
        follow[symbol] = set()
    # run through grammar 5 times to ensure all follows are accounted for
    for n in range(5):
        for i, key in enumerate(list(grammar.keys())):
            if i == 0:
                # add $ follow to starting symbol in grammar
                follow[key].add("$")
            for production in grammar[key]:
                for j, symbol in enumerate(production):  # go through each symbol
                    if symbol in non_terminals:
                        if j == len(production) - 1:
                            follow[symbol] = follow[symbol].union(follow[key])
                            break
                        for follow_symbol in production[j+1:]:
                            ended_eps = False
                            if follow_symbol not in non_terminals:
                                follow[symbol].add(follow_symbol)
                                break
                            elif "ε" in first_sets[follow_symbol]:
                                follow[symbol] = follow[symbol].union(
                                    first_sets[follow_symbol])
                                ended_eps = True
                            else:
                                follow[symbol] = follow[symbol].union(
                                    first_sets[follow_symbol])
                                break
                        if ended_eps:
                            follow[symbol] = follow[symbol].union(follow[key])

    # remove epsilons
    for symbol in follow:
        if "ε" in follow[symbol]:
            follow[symbol].remove("ε")

    return follow
```

```
def get_first_plus_sets(grammar: dict, non_terminals: set, first_sets: dict,
follow_sets: dict) -> dict:
    """
    Gets first+ sets for each non-terminal symbol.

    args
```

```
        grammar: previously built grammar dictionary
        non_terminals: set of all non-terminals in grammar
        first_sets: dictionary of sets with each symbol's first set
        follow_sets: dictionary of sets with each symbol's follow set

    returns
        first_plus: dictionary of sets with each symbol's first_plus set
    """
    first_plus = dict()

    for key in list(grammar.keys()):    # get non-terminals in order
        for production in grammar[key]:  # iterate over every production
            # create key for dict
            production_key = f"{key}->{' '.join(production)}"
            first_plus[production_key] = set()  # set to empty set
            # iterate through symbols keeping track of position to recognize last
symbol
            for j, symbol in enumerate(production):
                if symbol not in non_terminals:  # if symbol is terminal, add
                    first_plus[production_key].add(
                        symbol)
                    # if epsilon FIRST+(p) = FIRST(β) ∪ FIRST(X)
                    if symbol == "ε":
                        first_plus[production_key] =
first_plus[production_key].union(
                            follow_sets[key])
                        if key == "selection_stmt":  # do not add else in
selection_stmt -> ε
                            first_plus[production_key].remove("else")

                    break
                else:
                    if "ε" not in first_sets[symbol]:   # if only terminals, add
                        first_plus[production_key] =
first_plus[production_key].union(
                            first_sets[symbol])
                        break
                    else:   # add first set without epsilon, remain in loop
                        nt_first = first_sets[symbol]
                        nt_first.remove("ε")
                        first_plus[production_key] = nt_first.union(
                            first_plus[production_key])

                        if j == len(production) - 1:    # if last symbol is nt with
ε
```

```
                            first_plus[production_key] = first_sets[symbol].union(
                                follow_sets[key])

    return first_plus
def create_parse_table(grammar: dict, terminals: set, first_plus_sets: dict,
verbose: bool = False):
    """
    Creates parse table to get transitions for parser.

    args
        grammar: previously built grammar dictionary
        terminals: set of all terminals in grammar
        first_plus_sets: dictionary of sets with each production's first plus set

    returns
        parse_table: dictionary of dictionaries representing de parse table
    """
    parse_table = {}

    non_terminals_list = list(grammar.keys())
    terminals_list = list(terminals)
    terminals_list.sort()
    terminals_list.append('$')

    n = 1

    for nt in non_terminals_list:
        parse_table[nt] = {}
        for t in terminals_list:
            parse_table[nt][t] = "ERROR"
        for production in grammar[nt]:
            production_key = f"{nt}->{' '.join(production)}"
            for t in first_plus_sets[production_key]:
                if t == 'ε':
                    continue
                if parse_table[nt][t] == "ERROR":
                    parse_table[nt][t] = n
                else:
                    parse_table[nt][t] = [parse_table[nt][t]].extend(n)
                    print(f'DOUBLE ON {nt} with {t}: {parse_table[nt][t]}')

            n += 1


    return parse_table
```

4.2 Parser Code

The LL(1) Parsing Algorithm was translated to Python and implemented directly. Then it was modified to make accommodations for semantic and syntax error recognition. On a successful run, it will display SUCCESS and the symbol table.

```
def LL1(grammar: dict, parse_table: dict, input: list, symbol_table: list):
    """
    Runs LL(1) Parsing Algorithm.

    args
        grammar: dict representing grammar derived from .txt
        parse_table: dict of dicts representing LL(1) parsing table
        input: list of lists from scanner output. Must not be empty.

    returns
        bool indicating success.
    """
    # Validate input
    if len(input) == 1 and input[0][1] == 30:
        raise Exception("INPUT: code file cannot be empty")

    non_terminals = list(grammar.keys())
    productions = gram.enumerate_productions(grammar)

    production_number = 0

    stack = ['$', non_terminals[0]]  # stack with symbols to match
    input_pointer = 0   # pointer to traverse input
    symbol_table = initialize_symbol_table(symbol_table)

    current_nt = ''

    current_scope = set()  # set of accessible variables in scope
    scope_lvl = 0   # lvl of scope for vars

    while stack[-1] != '$':
        top = stack[-1]  # assign top to variable for legibility
        # current token from input
        token = id_to_token(input[input_pointer][1])

        print(f'stack: {stack}\ttoken: {token}')

        if top == token:    # if match
            print(f'matched {token}. nt: {current_nt}')
```

```
            if token == 'ID':   # matched ID
                identifier = input[input_pointer][2] - 1  # identifier position
                identifier_name = symbol_table[identifier][0]
                print(identifier_name)
                if current_nt == 'declaration':  # matched global fun or var

                    next_token = id_to_token(input[input_pointer + 1][1])

                    if next_token == '(':   # matched fun
                        current_scope = get_global_variables(symbol_table)
                        print(f'current_scope: {current_scope}')
                        fun_type = id_to_token(input[input_pointer - 1][1])
                        if identifier_name == 'main':   # matched main function
                            # if not equal, neither is None. Overwriting main
                            if symbol_table[identifier][1] != None:
                                raise Exception(
                                    f'SEMANTIC ERROR in line
{input[input_pointer][0]}: Function main can only be declared once')
                            if fun_type != 'void' or
id_to_token(input[input_pointer + 2][1]) != 'void' or
id_to_token(input[input_pointer + 3][1]) != ')':
                                raise Exception(
                                    f'SEMANTIC ERROR in line
{input[input_pointer][0]}: Function main must be type void with single parameter
void')

                        symbol_table[identifier][1] = 'function'
                        symbol_table[identifier][2] = fun_type

                    else:   # matched global var
                        if identifier_name == 'main':
                            raise Exception(
                                f'SEMANTIC ERROR in line {input[input_pointer][0]}:
Variable cannot be named main')

                        symbol_table[identifier][1] = 'var'
                        symbol_table[identifier][2] = 'global'

                        current_scope = current_scope.union(
                            get_global_variables(symbol_table))
                        print(f'current_scope: {current_scope}')

                if current_nt == 'var_declaration':   # matched local var
                    if identifier_name == 'main':
```

```
                        raise Exception(
                            f'SEMANTIC ERROR in line {input[input_pointer][0]}:
Variable cannot be named main')
                    symbol_table[identifier][1] = 'var'
                    symbol_table[identifier][2] = 'local'
                    current_scope.add((identifier_name, scope_lvl))
                    print(f'current_scope: {current_scope}')

                if current_nt == 'statement':   # assigning var or calling function
                    next_token = id_to_token(input[input_pointer + 1][1])
                    if next_token == '(':  # calling function
                        # if equal, function has not been declared.
                        if symbol_table[identifier][1] == None:
                            raise Exception(
                                f'SEMANTIC ERROR in line {input[input_pointer][0]}:
Function {identifier_name} has not been declared')
                        elif symbol_table[identifier][1] != 'function':
                            raise Exception(
                                f'SEMANTIC ERROR in line {input[input_pointer][0]}:
{identifier_name} is not a function and cannot be called')
                        elif identifier_name == 'main':
                            raise Exception(
                                f'SEMANTIC ERROR in line {input[input_pointer][0]}:
main function cannot be called')
                    elif next_token == '=':  # assigning variable
                        if symbol_table[identifier][1] == None:
                            raise Exception(
                                f'SEMANTIC ERROR in line {input[input_pointer][0]}:
Var {identifier_name} has not been declared')
                        elif symbol_table[identifier][1] == 'function':
                            raise Exception(
                                f'SEMANTIC ERROR in line {input[input_pointer][0]}:
Cannot assign value to function {identifier_name}')
                        elif not in_scope(identifier_name, current_scope):
                            raise Exception(
                                f'SEMANTIC ERROR in line {input[input_pointer][0]}:
{identifier_name} not in scope of statement')

                if current_nt == 'param':  # parameters in function declaration
                    # only exists in params
                    if symbol_table[identifier][1] == symbol_table[identifier][2]:
                        # both equal to allow overwriting in global or local
variables
                        symbol_table[identifier][1] = 'param'
                        symbol_table[identifier][2] = 'param'
```

```
                        current_scope.add((identifier_name, scope_lvl))
                        print(f'current_scope: {current_scope}')

                if current_nt == 'factor':  # doing math
                    # if function does not return value
                    if symbol_table[identifier][2] == 'void':
                        raise Exception(
                            f'SEMANTIC ERROR in line {input[input_pointer][0]}:
{identifier_name} does not return a value. Cannot be factor')
                    elif symbol_table[identifier][1] != 'function' and not
in_scope(identifier_name, current_scope):
                        raise Exception(
                            f'SEMANTIC ERROR in line {input[input_pointer][0]}:
{identifier_name} not in scope of statement')

                # if current_nt == 'var':  # var is only accessed in input

            elif token == '{':
                scope_lvl += 1
            elif token == '}':
                current_scope = remove_level_of_scope(current_scope, scope_lvl)
                scope_lvl -= 1
            stack.pop()  # remove from stack
            input_pointer += 1  # traverse input

        # if TopStack is terminal without match
        elif top not in non_terminals:
            handle_error_stack(token, input[input_pointer]
                               [0], productions, production_number)
        elif parse_table[top][token] == "ERROR":
            handle_error_table(top, token, input[input_pointer][0])
        else:   # traverse Parse Table to new production
            production_number = parse_table[top][token]  # production to go to
            # symbols in RHS of production
            production_symbols = productions[production_number]
            print(f'{production_symbols[0]} -> {production_symbols[1:]}')

            current_nt = production_symbols[0]

            stack.pop()   # pop before inserting new symbols
            if "ε" not in production_symbols:   # do not push epsilon
                # insert symbols in reverse
                stack.extend(production_symbols[::-1])

    if stack[-1] == '$' and token == '$':  # program ended correctly
```

```
        print('-----------SUCCESS-----------')
        show_symbol_table(symbol_table)
        if last_fun_main(symbol_table):
            return True
        else:
            raise Exception(
                "SEMANTIC: Last function declaration must be void main(void){}")
    elif input_pointer >= len(input):    # input incomplete
        print(f'stack: {stack}\ttoken: {token}')
        raise Exception(
            f'INPUT: Input ended prematurely, top of stack: {stack[-1]}')
    else:
        print(f'stack: {stack}\ttoken: {token}')     # did not end correctly
        raise Exception(f'TOP: Did not end on $, got {token}')
```

Output:

```
-----------SUCCESS-----------
global  var     global
sort    function        int
x       param   param
n       param   param
another param   param
main    function        void
i       var     local
arr     var     local
whilevar        var     local
```

4.3 Semantic Error Handler functions

```
def last_fun_main(symbol_table: list) -> bool:
    """
    Traverses symbol table in reverse until a function is found. If the function is
void and called main, accept.
    """
    for entry in symbol_table[::-1]:
        if entry[1] == 'function':  # last of type function
            # last function is void main()
            if entry[0] == 'main' and entry[2] == 'void':
                return True
            else:
                return False


def show_symbol_table(symbol_table: list):
    """
    Show symbol table in a readable format

    args
```

```
        symbol_table: list of identifiers in format {identifier: [type(fun/var), 
return type or var scope]}
    """

    for entry in symbol_table:
        entry = map(lambda x: 'None' if x is None else x, entry)
        print('\t'.join(entry))
    return


def initialize_symbol_table(symbol_table: list) -> list:
    """
    Initialize symbol table to be in parser format. [identifier, type(fun/var), 
return type or var scope]

    args
        symbol_table: list of identifiers from scanner output.

    returns
        symbol_table: list of lists in parser format. [identifier, type(fun/var), 
return type or var scope]
    """

    for i, identifier in enumerate(symbol_table):
        symbol_table[i] = [identifier, None, None]

    return symbol_table


def get_global_variables(symbol_table: list) -> set:
    """
    Gets set of global variables in format {('name', scope_lvl)}
    """
    globals = set()
    for entry in symbol_table:
        if entry[2] == 'global':
            globals.add((entry[0], -1))
    return globals


def remove_level_of_scope(vars: set, scope_lvl: int) -> set:
    """
    Removes all vars lower that the specified level of scope, making them 
inaccessible
    """
```

```
    scoped_vars = set()
    for var in vars:
        if var[1] < scope_lvl:
            scoped_vars.add(var)

    return scoped_vars


def in_scope(var_name: str, scoped_vars: set) -> bool:
    for var in scoped_vars:
        if var_name == var[0]:
            return True
    return False
```

4.4 Syntax Error Handler functions

```
def handle_error_stack(token: str, line: int, productions: dict, n: int):
    print(
        f'\n\nTOKEN: {token}\tProduction {n}: {productions[n][0]} ->
{productions[n][1:]}')

    error_msgs = [
        f"Program must begin with a function or variable declaration.",
        f"Program must begin with a function or variable declaration.",
        f"Invalid declaration. Variables can only be int. Functions can only be
Void.",
        f"Invalid declaration. Variables can only be int. Functions can only be
Void.",
        f"Invalid declaration. 'int' must be followed by a valid identifier.",
        f"Invalid void function declaration. Must be of format 'void ID(params)
{{}}'.",
        f"Invalid variable declaration. Must be of format 'int ID;'.",
        f"Invalid array declaration. Must be of format 'int ID[NUM];'.",
        f"Invalid int function declaration. Must be of format 'int ID(params)
{{}}'.",
        f"Invalid variable declaration. 'int' must be followed by a valid
identifier.",
        f"Invalid variable declaration. Must be of format 'int ID;'.",
        f"Invalid array declaration. Must be of format 'int ID[NUM];'.",
        f"Invalid parameters. If multiple parameters, they must be of type int.",
        f"Invalid parameters. Only one parameter allowed for 'void' parameters.",
        f"Invalid parameters. Parameters must be separated by commas ','.",
        f"Invalid parameters. Parameter list must be closed by a parenthesis ')'.",
        f"Invalid parameter. int params must be valid identifiers.",
```

```
        f"Invalid parameter. Array parameters must not have a size in format 'int
ID[]'.",
        f"Invalid parameters. List of parameters must close on a ')' and be
separated by commas ','.",
        f"Invalid function body. The body of a function must be enclosed by
brackets '{{}}'.",
        f"Invalid local declaration. Variables can only be ints.",
        f"Invalid statement. Cannot do operations outside of an assignment,
comparison, function call, or array call.",
        f"Invalid statement. Cannot do operations outside of an assignment,
comparison, function call, or array call.",
        f"Invalid statement. No declarations may be after the first statement in a
scope. Statements must begin with an ID or keyword.",
        f"Invalid statement. Cannot call or assign to an invalid ID.",
        f"Invalid statement. Must be encompased by brackets '{{}}'.",
        f"Invalid if statement. Must be of format 'if (expression) statement' with
an optional 'else statement'.",
        f"Invalid while statement. Must be of format 'while (expression)
statement'.",
        f"Invalid return statement.",
        f"Invalid input statement. Must be followed by an existing variable in
format 'input var;'.",
        f"Invalid output statement. Must be followed by an existing variable in
format 'output var;'.",
        f"Invalid assignment statement. Can only assign to a valid var or array
element in format 'ID = expression;'",
        f"Invalid function call. Must be called with args between parentheses in
format 'ID(args)'.",
        f"Invalid else statement. Must begin with 'else' keyword and be followed by
another statement.",
        f"Invalid else statement. Must begin with 'else' keyword and be followed by
another statement.",
        f"Invalid void return statement.",
        f"Invalid int return statement. Must return an expression or variable.",
        f"Invalid input statement. Must be followed by an existing variable in
format 'input var;'.",
        f"Invalid input statement. Must be followed by an existing variable in
format 'input var[position];'.",
        f"Invalid var declaration or assignment.",
        f"Invalid expression.",
        f"Invalid relational expression.",
        f"Invalid expression. Must begin with an ID or keyword.",
        f"Invalid relational operator. The only relational operators are '<, >, <=,
>=, ==, !='.",
```

```
        f"Invalid relational operator. The only relational operators are '<, >, <=, 
>=, ==, !='.",
        f"Invalid relational operator. The only relational operators are '<, >, <=, 
>=, ==, !='.",
        f"Invalid relational operator. The only relational operators are '<, >, <=, 
>=, ==, !='.",
        f"Invalid relational operator. The only relational operators are '<, >, <=, 
>=, ==, !='.",
        f"Invalid relational operator. The only relational operators are '<, >, <=, 
>=, ==, !='.",
        f"Invalid arithmetic expression. Expressions can only be done with NUMs 
IDs.",
        f"Invalid arithmetic expression. May only have one operator. Operators may 
only be '-, +, *, /'.",
        f"Invalid arithmetic expression. May only have one operator. Operators may 
only be '-, +, *, /'.",
        f"Incomplete arithmetic expression. Missing second term.",
        f"Incomplete arithmetic expression. Missing second term.",
        f"Invalid arithmetic expression. Expressions can only be done with NUMs 
IDs.",
        f"Invalid arithmetic expression. May only have one operator. Operators may 
only be '-, +, *, /'.",
        f"Invalid arithmetic expression. May only have one operator. Operators may 
only be '-, +, *, /'.",
        f"Incomplete arithmetic expression. Missing second term.",
        f"Incomplete arithmetic expression. Missing second term.",
        f"Invalid arithmetic expression. May only have one operator. Operators may 
only be '-, +, *, /'.",
        f"Invalid arithmetic expression. Expressions can only be done with NUMs 
IDs.",
        f"Invalid arithmetic expression. Expressions can only be done with NUMs 
IDs.",
        f"Invalid array call. Array elements must be called with a NUM or an 
expression that resolves to an int.",
        f"Invalid arithmetic expression. May only have one operator. Operators may 
only be '-, +, *, /'.",
        f"Invalid function call. Function call must be 'fun_name(params)'.",
        f"Invalid function call. Arguments must resolve to an int with a NUM, ID or 
expression.",
        f"Invalid function call. Unclosed arguments list. Must end in ')'.",
        f"Invalid function call. Arguments must resolve to an int with a NUM, ID or 
expression.",
        f"Invalid function call. Arguments must be separated by commas ','.",
        f"Invalid function call. Unclosed arguments list. Must end in ')'."]
```

```
    raise Exception(
        f'SYNTAX ERROR in line {line}: {error_msgs[n - 1]} Got {token}')


def handle_error_table(top: str, token: str, line: int):

    if top == "program":
        error = f"Program must begin with a declaration."
    elif top == "declaration_list":
        error = f"Declaration must begin with int or void."
    elif top == "declaration":
        error = f"Declaration must begin with int or void."
    elif top == "declaration'":
        error = f"A function declaration must include '(params)'. Variable name
must be followed by ';' or '[SIZE]'."
    elif top == "var_declaration":
        error = f"Variables can only be int."
    elif top == "var_declaration'":
        error = f"Variable name must be followed by ';' or '[SIZE]'."
    elif top == "params":
        error = f"Parameter definition must be list of ints or a single void."
    elif top == "param_list":
        error = f"List of parameters must be separated by commas ',' and closed by
a parenthesis ')'."
    elif top == "param":
        error = f"Params can only be of type int once another int has been declared
as a param."
    elif top == "param'":
        error = f"List of parameters must be separated by commas ',' and closed by
a parenthesis ')'."
    elif top == "compound_stmt":
        error = f"The body of a statement must begin with brackets '{{'."
    elif token in ['!=', '==', '<', '<=', '>', '>=']:
        error = f"Relational operators may only exist in a relational expression."
    elif token == 'NUM':
        error = f"Numbers may only be used in relational and arithmetic
expressions, to access arrays or as params."
    elif token in ['+', '/', '+', '-']:
        error = f"Arithmetic operators may only exist in an arithmetic expression."
    elif token == '=':
        error = f"Assignments can only be done after declarations and to
variables."
    elif token == ';':
        error = f"Unexpected ';'. May only be used after a complete statement."
    elif token in '[]':
```

```
        error = f"Square brackets may only be used after an ID to access an array's
element."
    elif token in '{}':
        error = f"Brackets may only be used to open or close statement bodies."
    elif token == 'void':
        error = f"'void' may only be used to define functions or to indicate null
parameters."
    elif token == ',':
        error = f"Commas may only be used to separate parameters or arguments."
    elif token == 'else':
        error = f"Else statements may only be declared after closing an if
statement."
    elif token in '()':
        error = f"Parentheses may only be used in function declarations, calls, or
logical or arithmetic expressions."
    elif token == "if" or token == "while":
        error = f"If and While statements may only be used inside a function body."
    elif token == 'return':
        error = f"Return statements may only be used inside function declarations."
    elif top == "local_declarations":
        error = f""
    elif top == "statement_list":
        error = f"After the first statement in a function body, no more
declarations may be made."
    elif top == "statement":
        error = f"After the first statement in a function body, no more
declarations may be made."
    elif top == "statement'":
        error = f"Did not end statement correctly."
    elif top == "selection_stmt":
        error = f"After the first statement in a function body, no more
declarations may be made."
    elif top == "return_stmt":
        error = f"No other statements or declarations may be made to return."
    elif top == "var":
        error = f"vars must be a valid ID"
    elif top == "var'":
        error = f"Invalid var."
    elif top == "expression":
        error = f"Expressions must begin with an ID."
    elif top == "expression'":
        error = f"Did not end expression correctly"
    elif top == "relop":
        error = f"Relational operators are '!=, ==, >, >=, <, <=='"
    elif top == "arithmetic_expression":
```

```
        error = f"An arithmetic expression must begin with an identifier."
    elif top == "arithmetic_expression'":
        error = f"Did not end arithmetic expression correctly."
    elif top == "addop":
        error = f"Can only add or substract with '+' or '-'."
    elif top == "term":
        error = f"An arithmetic expression must begin with an identifier."
    elif top == "term'":
        error = f"Did not end arithmetic expression correctly."
    elif top == "mulop":
        error = f"Can only multiply or divide with '*' or '/'."
    elif top == "factor":
        error = f"An arithmetic expression must begin with an identifier."
    elif top == "factor'":
        error = f"Did not end arithmetic expression correctly."
    elif top == "call":
        error = f""
    elif top == "args":
        error = f""
    elif top == "args_list":
        error = f""

    raise Exception(
        f'SYNTAX ERROR in line {line}: {error} Got {token}')
```

5. Verification and Validation

5.1 Validation

5.1.1 Test Cases

Test ID	Purpose	Expected	Received	Result
#SCAN_01	Correct code	Full output	Full output	PASS
#SCAN_02	Correct longer code	Full output	Full output	PASS
#SCAN_03	Test Single 'void'	out = [SYNTAX ERROR] error = Invalid void function declaration. Must be of format 'void ID(params) {}'.	out = [SYNTAX ERROR] error = Invalid void function declaration. Must be of format 'void ID(params) {}'.	PASS
#SCAN_04	Test Empty File	out = [INPUT ERROR] error = code file cannot be empty	out = [INPUT ERROR] error = code file cannot be empty	PASS
#SCAN_05	Out of scope var	out = [SEMANTIC ERROR] error = var out of scope	out = [SEMANTIC ERROR] error = var out of scope	PASS

5.2 Verification

This document provides the required documentation as well as explaining the source code sufficiently. All semantical requirements that can be solved in the syntax analysis phase are also complied.

6. References

7. Bibliography

Alfred V, A. M. (2014). *compilers: principles, techniques, and tools.* Harlow, Essex: Pearson.