SYNTAX AND SEMANTIC ANALYZER

Authors:
Bolaños Guerrero Julián
Castillo Soto Jacqueline
Galindo Reyes Daniel Adrián
Isidro Castro Karen Cristina
Zurita Cámara Juan Pablo

A. DEPENDENCIES:

Important: To compile our program is necessary to have a C99 or higher C compiler version.

Necessary files for compilation are:

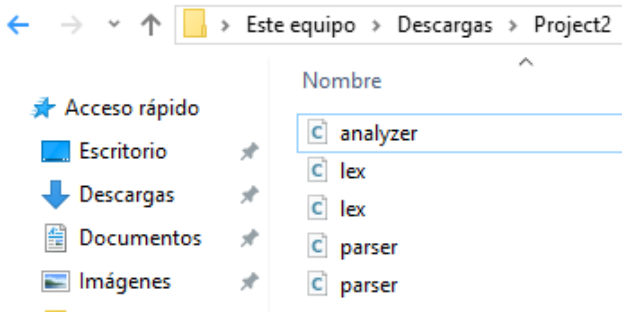
- **lex.h**: Header file that contains the declarations of functions and data structures necessary for lexical analysis.
- **lex.c:** Implements functions and initializes predefined variables from the header file *lex.h.* Is the heart of the lexical analyzer.
- **parser.h**: Header file that contains the declarations of functions and data structures necessary for syntax analysis.
- **parser.c**: Implements functions and initializes predefined variables from the header file *parser.h*. Is the heart of the syntax analyzer.
- **analyzer.c**: Integrates the lexical and syntax analysis modules. Adds the necessary functions and data structures for semantic analyzer. Is the main file of our analyzer.

These files are available online in the following GitHub repository:
https://github.com/DanielGalindoDev?tab=repositories

B. INSTALLATION GUIDE:

NOTE: It is necessary to have a C99 or higher C compiler version installed.

1. Download the necessary files for compilation (shown in section A).
2. After downloading the files, place the files in the same folder.



3. Once the files have been downloaded and placed in a folder, open a terminal and navigate to the folder containing the files.

```
PS C:\Users\juanz> cd Downloads\Project2
PS C:\Users\juanz\Downloads\Project2>
```

4. Once inside the folder, proceed to compile the three .c files: *lex.c*, *parser.c* and *analyzer.c*.

```
PS C:\Users\juanz\Downloads\Project2> gcc lex.c parser.c analyzer.c -o a.exe
PS C:\Users\juanz\Downloads\Project2>
```

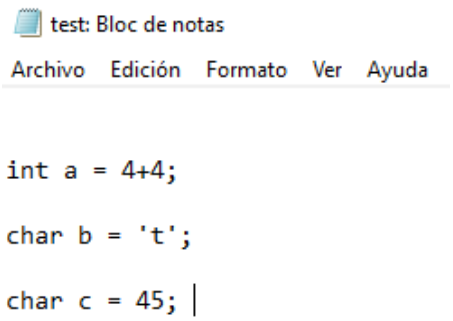
5. As a result of the compilation, we will obtain an executable file: a.exe (for Windows) and a.out (for macOs). This is the program that will do syntax and semantical analysis.

C. USAGE

Once the executable file is generated, we can use it to analyze any text file. This program will perform a syntax and semantic analysis of the file according to our grammar (see section D). When executed, it will display whether the programming type statements (either variable declaration or initialization) found in the file are correct syntactically and semantically or not.

The program can analyze any text file. To make the program analyze a certain text file we need to execute the following command in our terminal: .\a.exe fileName.<extension>. This is assuming that you are in the folder where the executable and the input file are. The most general command would be by using absolute paths for both files.

As an example, we created a simple text file in the same folder where our program executable is and wrote down a few variable initialization statements. Then we analyze that file using our program.




```
PS C:\Users\juanz\Downloads\Project2> ls


    Directorio: C:\Users\juanz\Downloads\Project2


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        11/11/2024  09:22 p. m.          76626 a.exe
-a----        11/11/2024  07:32 p. m.           8848 analyzer.c
-a----        11/11/2024  07:35 p. m.          10403 lex.c
-a----        10/11/2024  03:44 p. m.           1106 lex.h
-a----        11/11/2024  07:44 p. m.           5657 parser.c
-a----        10/11/2024  04:11 p. m.            720 parser.h
-a----        11/11/2024  09:28 p. m.             51 test.txt


PS C:\Users\juanz\Downloads\Project2> .\a.exe test.txt
```

```
PS C:\Users\juanz\Downloads\Project2> .\a.exe test.txt

Sentencia (1):
Sintacticamente correcta.
Semanticamente correcto.
        ->Nombre de variable: a
        ->Tipo: INT
        ->Valor: 8

Sentencia (2):
Sintacticamente correcta.
Semanticamente correcto.
        ->Nombre de variable: b
        ->Tipo: CHAR
        ->Valor: t

Sentencia (3):
Sintacticamente correcta.
Semanticamente incorrecto.
        ->Tipos de datos incompatibles.
PS C:\Users\juanz\Downloads\Project2>
```

D. ARCHITECTURE AND DESING

The syntax analysis is based in the following grammar:

1. statementList → statement statementList
2. statementList → statement
3. statement → variable_creation **;**
4. variable_creation → definition
5. variable_creation → definition = expression
6. definition → **type identifier**
7. expression → expression **math1** factor
8. expression → factor
9. factor → factor **math2** term
10. factor → term
11. term → **(** expression **)**
12. term → **constant**
13. term → **math1** term

The grammar consists in variable declarations and variable initialization to constant expressions in a C and Java programming style.

According to the grammar we have:

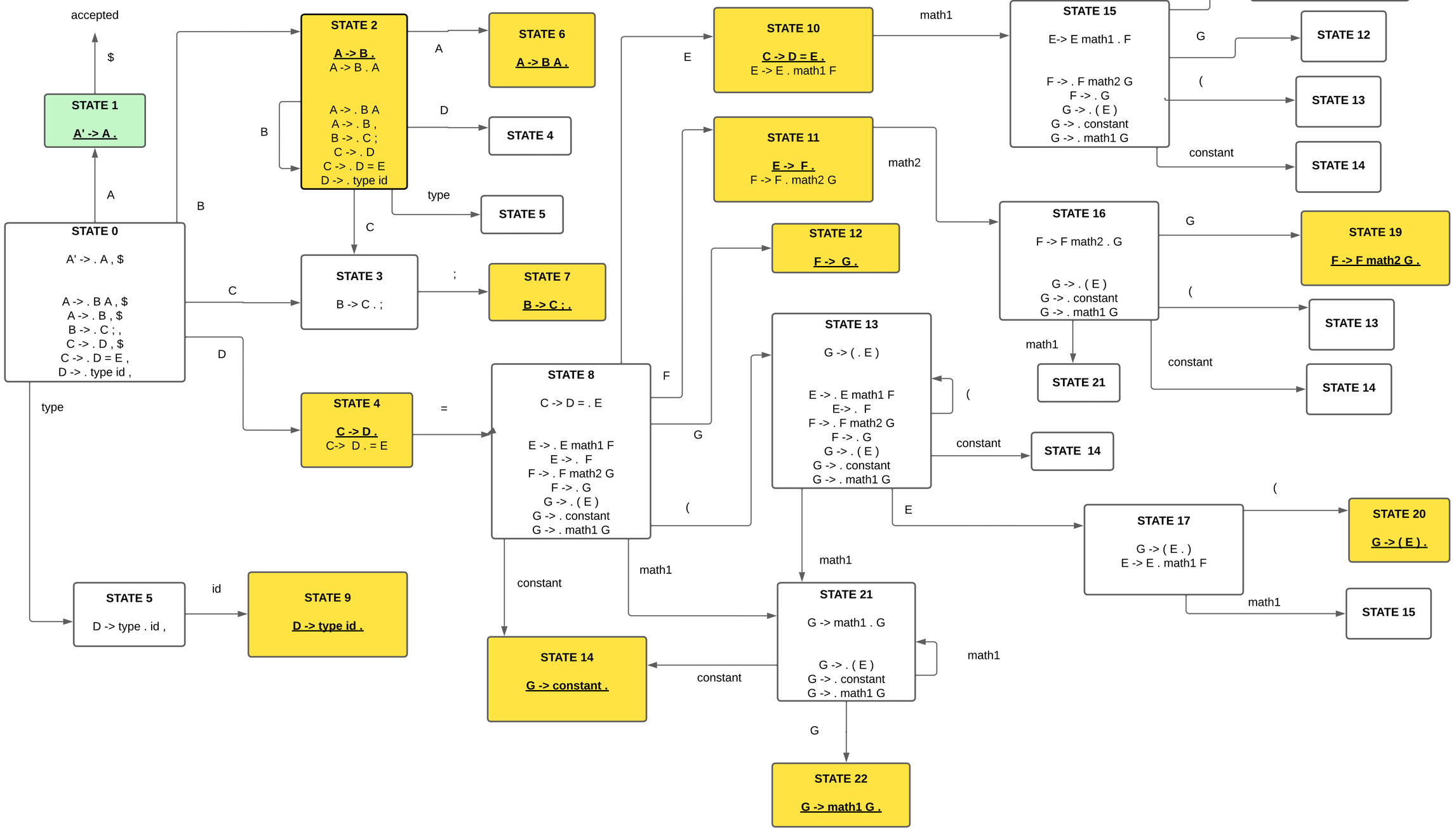
- Terminals: **;** , =, **type**, **identifier**, **math1**, **math2** and **constant**
- Non terminals: statementList (A), statement (B), variable_creation (C), definition (D), expression (E), factor (F) and term (G).
- Start symbol: statementList.

The terminals of this grammar correspond to the tokens that a lexical analyzer should identify. Tokens are the classification of a group of characters. The tokens of this project can be represented by the following strings:

- **type**: int, char, float
- **identifier**: [a-zA-Z$_][a-zA-Z$_0-9]*
- **math1**: +, -
- **math2:** *, /
- **constant**: Either a numeric constant such as 4 or 3.2, or a character constant like ‘t’ or ‘r’.

The syntax analysis is based on a SLR parser. This made the semantic analysis part easier since it is done every time a reduction action takes place in the SLR algorithm.

LR(0) AUTOMATON

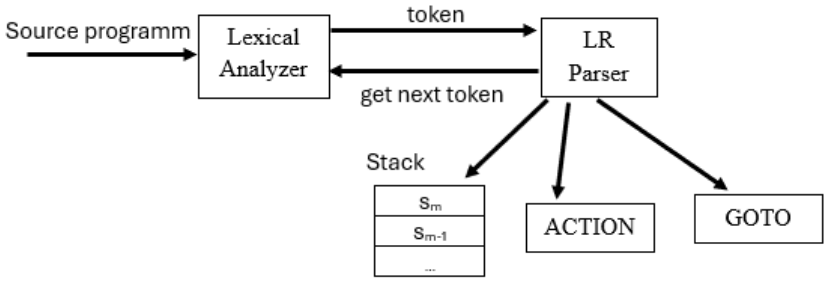


NOTATION
sX means shift to X state.
rX means reduce according to the production rule X.

PARSING TABLE

	ACTION											GOTO						
STATE	type	id	math1	math 2	CONSTANT	;	=	(	)	NON VALID	$	A	B	C	D	E	F	G
0	s5	error	error	error	error	error	error	error	error	error	error	s1	s2	s3	s4	error	error	error
1	error	error	error	error	error	error	error	error	error	error	accepted	error	error	error	error	error	error	error
2	s5	error	error	error	error	error	error	error	error	error	r2	s6	s2	s3	s4	error	error	error
3	error	error	error	error	error	s7	error	error	error	error	error	error	error	error	error	error	error	error
4	error	error	error	error	error	r4	s8	error	error	error	error	error	error	error	error	error	error	error
5	error	s9	error	error	error	error	error	error	error	error	error	error	error	error	error	error	error	error
6	error	error	error	error	error	error	error	error	error	error	r1	error	error	error	error	error	error	error
7	r3	error	error	error	error	error	error	error	error	error	r3	error	error	error	error	error	error	error
8	error	error	s21	error	s14	errorr	error	s13	error	error	error	error	error	error	error	s10	s11	s12
9	error	error	error	error	error	r6	r6	error	error	error	error	error	error	error	error	error	error	error
10	error	error	s15	error	error	r5	error	error	error	error	error	error	error	error	error	error	error	error
11	error	error	r8	s16	error	r8	error	error	r8	error	error	error	error	error	error	error	error	error
12	error	error	r10	r10	error	r10	error	ero	r10	error	error	error	error	error	error	error	error	error
13	error	error	s21	error	s14	error	error	s13	error	error	error	error	error	error	error	s17	s11	s12
14	error	error	r12	r12	error	r12	error	error	r12	error	error	error	error	error	error	error	error	error
15	error	error	s21	error	s14	error	error	s13	error	error	error	error	error	error	error	error	s18	s12
16	error	error	s21	error	s14	error	error	s13	error	error	error	error	error	error	error	error	error	s19
17	error	error	s15	error	error	error	error	error	s20	error	error	error	error	error	error	error	error	error
18	error	error	r7	s16	error	r7	error	error	r7	error	error	error	error	error	error	error	error	error
19	error	error	r9	r9	error	r9	error	error	r9	error	error	error	error	error	error	error	error	error
20	error	error	r11	r11	error	r11	error	error	r11	error	error	error	error	error	error	error	error	error
21	error	error	s21	error	s14	error	error	error	error	error	error	error	error	error	error	error	error	s22
22	error	error	r13	r13	error	r13	error	error	r13	error	error	error	error	error	error	error	error	error

The way our program analyzes the input text file is as follows:



The source program refers to the input text file.

The lexical analyzer module identifies tokens from the stream of characters read from the input file. Each time it detects a token, it forwards it to the syntax analyzer module. After processing each token received from the lexer, the parser requests the next one, continuing this cycle until the lexer has processed the entire file.

The semantic analyzer works together with the parser each time a reduce action is performed.The semantic actions executed depend on the production rule used for the reduction.

The parsing algorithm implemented was:

```
let a be the first symbol fo w$;
while (1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s,a] = shift t) {
                push t onto the stack;
                let a be the next input symbol;
        } else if (ACTION[s,a] = reduce A→ β) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t,A] onto the stack;
                output the production A→β;
        } else if (ACTION[s,a] = accept ) break; /* parsing is done*/
        else call error-recovery routine;
}
```

Essentially, the algorithm consists of traversing the LR(0) automaton. The algorithm uses the parsing table to make decisions and relies on a stack to store states. By storing these states, the algorithm remembers the path it has taken. This memory is especially useful when reduction actions need to be performed.

D.1 SEMANTICAL ANALYSIS

This implementation is better explained with an example. Suppose we want to parse:

int a = 4;

The parsing algorithm explained in section D would be executed as follows:

Iteration	STACK	PARSE SYMBOLS	READ SYMBOL	ACTION
(1)	0		int	shift to 5
(2)	0 5	int	a	shift to 9
(3)	0 5 9	int a	=	reduce by D → **type id**
(4)	0 4	D	=	shift to 8
(5)	0 4 8	D =	4	shift to 14
(6)	0 4 8 14	D = 4	;	reduce by G → **constant**
(7)	0 4 8 12	D = G	;	reduce by F → G
(8)	0 4 8 11	D = F	;	reduce by E → F
(9)	0 4 8 10	D = E	;	reduce by C → D = E
(10)	0 3	C	;	shift to 7
(11)	0 3 7	C ;	$	reduce by B → C ;
(12)	0 2	B	$	reduce by A → B
(13)	0 1	A	$	accept

Although this technique does not explicitly construct a syntax tree, to understand semantic analysis, we need to think of a syntax tree. Essentially, the semantic analysis phase consists of passing the attributes of the syntax tree's leaves up through the tree until reaching the root. In the end, the root will contain the full semantic meaning of what was parsed.

We established that semantical actions take place when a reduction occurs. In this example, in iteration 3, after we reduce, we will expect that the non-terminal symbol D will have the following attributes:

char input [50]	**DATA_TYPE type**
'a'	int

These attributes were passed to D by the leaves **type** and **id** by executing what we will call semantic actions.

The semantic actions we expect to be done according to a reduction of a production rule are the following:

Production	Semantic actions
A → B A	nothing
A → B	nothing

B → C **;**	B = C print(“Parsing success”) print(“SDT verified”) // print B attributes.
C → D	C = D

C → D **=** E	if (D.type = CHAR or E.type = CHAR) then
	  if (D.type != E.type) then
	  print(“SDT error”) break
	D.STORAGE.floatValue = E.STORAGE.floatValue
	D.hasValue = 1
	
	C = D

D → **type identifier**	D.input = **identifier**.input D.type = **type**.type

E → E_1 **math1** F	if (E_1.type = CHAR or F.type = CHAR) then
	  print(“SDT error”) break
	if (**math1.input** = +) then
	  E_1.STORAGE.floatValue = E_1.STORAGE.floatValue + F.STORAGE.floatValue
	else
	  E_1.STORAGE.floatValue = E_1.STORAGE.floatValue – F.STORAGE.floatValue
	if (F.type = FLOAT) then
	  E_1.type = FLOAT
	
	E = E_1

E → F	E = F

F → F_1 **math2** G	if (F_1.type = CHAR or G.type = CHAR) then
	  print(“SDT error”) break
	if (**math2.input** = *) then
	  F_1.STORAGE.floatValue = F_1.STORAGE.floatValue * G.STORAGE.floatValue
	else
	  F_1.STORAGE.floatValue = F_1.STORAGE.floatValue / G.STORAGE.floatValue
	if (G.type = FLOAT) then
	  F_1.type = FLOAT
	
	F = F_1

F → G	F = G
G → **(** E **)**	G = E
G → **constant**	G = constant

G → **math1** G_1	if (G_1.type = CHAR) then
	  print(“SDT error”) break
	if (**math1.input** = -) then
	  G_1.STORAGE.floatValue = G_1.STORAGE.floatValue * (-1)
	G = G_1

Rules 1 and 2 of our grammar were added because that was our solution so that the parser would read multiple statements from the input, but essentially our goal is to parse statements. That is why one of the semantic actions when we reduce using rule 3 is to print that the parsing was a success.

From where are we going to get the information of each symbol involved in the semantic actions? If we look at the example table, we can see that the parse symbols column stores the symbols in a similar way to how the parsing algorithm stores states from the LR(0) automaton—in a stack structure. To implement the semantic analysis, we need to implement a stack of the user defined structure SYMBOL.

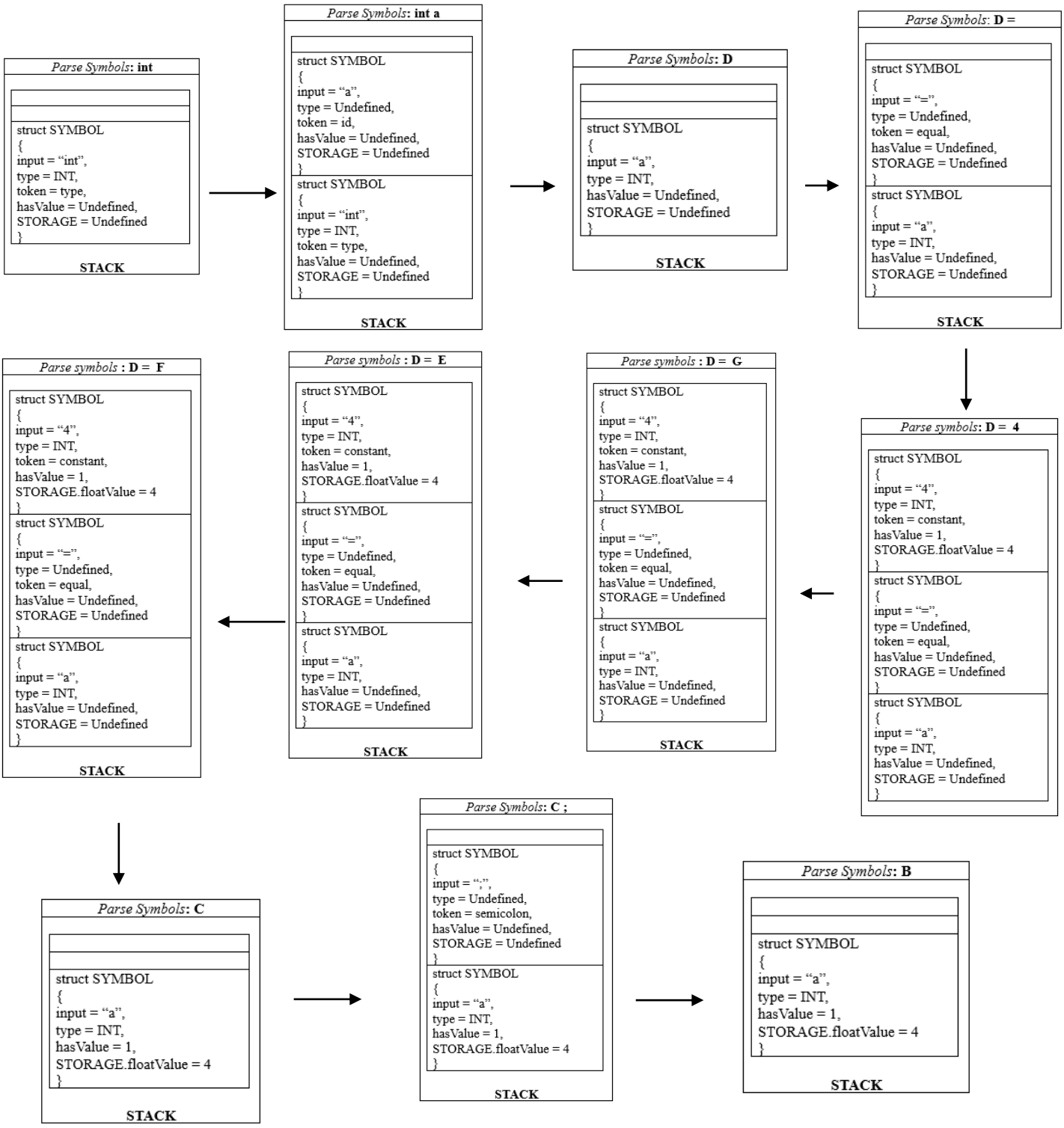
Havin in mind the parsing table, the algorithm for semantic analysis will be the following:

```
let a be the first symbol fo w$;
while (1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s,a] = shift t) {
                push s onto the symbol stack;
        } else if (ACTION[s,a] = reduce A→ β) {
                executed the semantical actions depending on the
                production rule;
                pop <length of the production rule - 1> from the
                symbol stack;
        }
}
```

This algorithm is basically the same as for parsing, so we only need to merge them.

In the table of semantic actions, we can see that the leftmost symbol on the right-hand side of the productions ends up inheriting all the attributes that give meaning to each production. This is why, in the algorithm, we only need to pop <length of the production rule - 1> because the leftmost symbol of the production, which will be the lowest in the stack with reference to the symbols used for semantic actions of the production, will contain all the necessary information. This approach prevents wasting memory on symbols that will not be used again.

Following the example of the parsing of ***int a = 4;*** , according to this new algorithm the SYMBOL stack will behave as follows:


Parse Symbols: int
struct SYMBOL
{
input = "int",
type = INT,
token = type,
hasValue = Undefined,
STORAGE = Undefined
}
STACK
Parse Symbols: int a
struct SYMBOL
{
input = "a",
type = Undefined,
token = id,
hasValue = Undefined,
STORAGE = Undefined
}
struct SYMBOL
{
input = "int",
type = INT,
token = type,
hasValue = Undefined,
STORAGE = Undefined
}
STACK
Parse Symbols: D
struct SYMBOL
{
input = "a",
type = INT,
hasValue = Undefined,
STORAGE = Undefined
}
STACK
Parse Symbols: D =
struct SYMBOL
{
input = "=",
type = Undefined,
token = equal,
hasValue = Undefined,
STORAGE = Undefined
}
struct SYMBOL
{
input = "a",
type = INT,
hasValue = Undefined,
STORAGE = Undefined
}
STACK
Parse symbols: D = 4
struct SYMBOL
{
input = "4",
type = INT,
token = constant,
hasValue = 1,
STORAGE.floatValue = 4
}
struct SYMBOL
{
input = "=",
type = Undefined,
token = equal,
hasValue = Undefined,
STORAGE = Undefined
}
struct SYMBOL
{
input = "a",
type = INT,
hasValue = Undefined,
STORAGE = Undefined
}
STACK
Parse symbols : D = G
struct SYMBOL
{
input = "4",
type = INT,
token = constant,
hasValue = 1,
STORAGE.floatValue = 4
}
struct SYMBOL
{
input = "=",
type = Undefined,
token = equal,
hasValue = Undefined,
STORAGE = Undefined
}
struct SYMBOL
{
input = "a",
type = INT,
hasValue = Undefined,
STORAGE = Undefined
}
STACK
Parse symbols : D = E
struct SYMBOL
{
input = "4",
type = INT,
token = constant,
hasValue = 1,
STORAGE.floatValue = 4
}
struct SYMBOL
{
input = "=",
type = Undefined,
token = equal,
hasValue = Undefined,
STORAGE = Undefined
}
struct SYMBOL
{
input = "a",
type = INT,
hasValue = Undefined,
STORAGE = Undefined
}
STACK
Parse symbols : D = F
struct SYMBOL
{
input = "4",
type = INT,
token = constant,
hasValue = 1,
STORAGE.floatValue = 4
}
struct SYMBOL
{
input = "=",
type = Undefined,
token = equal,
hasValue = Undefined,
STORAGE = Undefined
}
struct SYMBOL
{
input = "a",
type = INT,
hasValue = Undefined,
STORAGE = Undefined
}
STACK
Parse Symbols: C
struct SYMBOL
{
input = "a",
type = INT,
hasValue = 1,
STORAGE.floatValue = 4
}
STACK
Parse Symbols: C ;
struct SYMBOL
{
input = ";",
type = Undefined,
token = semicolon,
hasValue = Undefined,
STORAGE = Undefined
}
struct SYMBOL
{
input = "a",
type = INT,
hasValue = 1,
STORAGE.floatValue = 4
}
STACK
Parse Symbols: B
struct SYMBOL
{
input = "a",
type = INT,
hasValue = 1,
STORAGE.floatValue = 4
}
STACK

E. CODE IMPLEMENTATION

E.1 USER DEFINED DATA

```
typedef enum
{
    TYPE, //value 0
    ID, // value  1

    MATH1, // value 2
    MATH2, // value 3

    CONSTANT, // value 4

    SEMICOLON, // value 5
    EQUAL, // value 6

    LP, // value 7
    RP, // value 8

    NO_VALID, // value 9
    $ // value 10
}TOKEN;
```

Categorize the different types of tokens.

```
typedef enum
{
    CHAR,
    INT,
    FLOAT
}DATA_TYPE;
```

Categorize the different data types of a variable.

```
typedef struct
{
    char input[50];
    DATA_TYPE type;

    TOKEN token;
    char hasValue;

    union
    {
        int intValue;
        float floatValue;
        char charValue;
    }ST;

}SYMBOL;
```

To perform a semantic analysis, the tokens must carry meaning. This meaning can be represented by attributes. Since our implementation is in C, we chose to store this information in a user-defined structure called SYMBOL. This structure has the following attributes:

char input [50]	TOKEN token	DATA_TYPE type	char hasValue	STORAGE

- The input attribute is used to store what has been read to categorize the token. This is very useful to store the identifiers names or resolve semantic conflicts with token types such as **math1** and **math2**. Was it (+) or (-), was it (*) or (/).
- TOKEN is a user defined enumeration. For this project, the enumeration can be: TYPE (**type**), ID (**identifier**), MATH1 (**math1**), MATH2 (**math2**), CONSTANT (**constant**), SEMICOLON (**;**), EQUAL (=), LP (**(**), RP (**)**), NO_VALID or $ (**input end marker**). These represent the terminals in the grammar designed.
- DATA_TYPE is another user defined enumeration. For this project, this enumeration can be: CHAR, FLOAT, INT. The usual data types in programming.
- The attribute hasValue works with the attribute STORAGE. The value for this attribute will be 1 when the token read is a ***constant*** and 0 if not.
- The attribute STORAGE is a user defined union that can store either a float value or a char value. We used a union for memory efficiency. When a LITERAL CONSTANT is read, this attribute will store a char value based on the literal. If a NUMERIC CONSTANT is read, this attribute will store a float value based on the numeric constant.

```
typedef struct lex
{
    FILE *pf;
    char buffer[256];

    int i;
    int peek;
    int current_state;
}LEXER;
```

This structure stores the information necessary for the lexer to identify the tokens in the file. Essentially, the lexer reads the file in blocks of 255 characters and stores them in a buffer. Once the buffer has been fully read, the lexer loads the next block from the file. This cycle continues until the entire file has been processed. The variable *i* tells the lexer how far the buffer has been analyzed. The variable *peek* is used to read ahead of *i* until a token can be identified. When a token is identified, *i* updates its value to point to the same character as *peek*.

```
//Stack Definition
typedef struct
{
    //Array to store stack elements
    int arr[100];
    // Indez of the top element in the stack
    int top;
} Stack;
```

Implementation of the stack for the parsing algorithm.

```
typedef enum
{
    A, // value 0
    B, // value 1
    C, // value 2
    D, // value 3
    E, // value 4
    F, // value 5
    G // value 6
} NONTERMINAL;
```

Non terminals of our grammar.

```
typedef struct
{
    SYMBOL stack[100];
    int top;

    char semanticError;
}SYNTAX_TREE;
```

Use for the semantic analysis implementation.

When the analysis results are displayed, the variable semanticError indicates the semantic error that occurred.

E.2 HARD-CODED DATA

There are some theorical concepts that need to be hard coded in our implementation.

- Finte State machine for token identification

```
const char FA[][97] =
{
    {
    0, 7,12,12, 1,12,12, 9, 6, 6, 7, 7,12, 7, 4, 7, 3, 3, 3, 3, 3, 3, 3, 3, 3,12, 6, 7, 7, 7,12,12, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,12,12,12,12, 1,12, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 6,12, 6,12,0,0
    }, //q_0

    {
    -1,-1,-1,-1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,-1,-1,-1,-1, 1,-1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1
    }, //q_1
```

We use a bidimensional array. Each row represents a state in the finite state machine, and each column represents a character that can be read from the file. These characters include all printable ASCII characters as well as two special cases: the newline and null characters, resulting in a total of 97 columns.

Which row represents which state? Row 0 represents state 0, row 1 represents state 1, and forth.

Which columns represent which character? Index 0 through 94 represent the printable ASCII characters in order. Index 95 represents the null character, and index 96 represents the new line character.

FA[a][c] = b

If *b* is a non-negative integer, it indicates a transition from state *a* to state *b* via the character *c*.
If *b* is equal to -1, it indicates that there is no transition in state *a* with the character *c*.

- ACTION part of the parsing table

```
const char ACTION [][11] =
{
    /*  0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10  */
    {
        5,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1
    }, //q0

    /*   0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10  */
    {
        -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,30
```

We use a bidimensional array. Each row represents a state of the LR(0) automaton, and each column a token.

Which row represents which state? Row 0 represents state 0, row 1 represents state 1, and forth.
Which columns represent which character? The index corresponds to the value of each token given by the user defined enumeration TOKEN.

ACTION [a][b] = c

If *c* is less than 30 it means a shift action. *c* represents the state to shift to.
If *c* is equal to 30 it means accept.
If *c* is greater than 30 it means a reduce action. The result of *c* – 30 indicates which production rule is needed for the reduction.
If *c* is equal to -1 it means error.

When transferring the parsing table to code, we needed to implement a workaround in a specific state. When the parsing algorithm is in state 7 it must perform a reduction according to production rule 3 no matter the lookahead symbol. This adjustment was necessary because, without it, if the lookahead token was anything other than an identifier or $, the algorithm would output an error and fail to parse any subsequent statements in the file. This can be viewed as an error recovery routine.

- GOTO part of the parsing table

```
const char GOTO [][7] =
{

    /*  A, B, C, D, E, F, G*/
    {
        1, 2, 3, 4,-1,-1,-1
    }, //q0

    /*  A, B, C, D, E, F, G*/
    {
        -1,-1,-1,-1,-1,-1,-1
    }, //q1
```

We use a bidimensional array. Each row represents a state of the LR(0) automaton, and each column nonterminal.

Which row represents which state? Row 0 represents state 0, row 1 represents state 1, and forth.
Which columns represent which character? The index corresponds to the value of each nonterminal given by the user defined enumeration (NONTERMINAL).

GOTO [a][b] = c

If *c* is a non-negative integer, it represents the state to shift to.
If *c* is equal to -1 it means error.

- Production rule right hand side length

```
const char ruleSize [13] =
{
//  1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13
    2, 1, 2, 1, 3, 2, 3, 1, 3, 1, 3, 1, 2
};
```

In our grammar there are 13 rules. In this one-dimensional array each index represents a production rule. Index 0 represents rule 1, index 1 represents rule 2, and forth. The values stored in the array are integers that correspond to the size of the right-hand side length.

- Non-terminal that is reduce to by a production rule.

```
const NONTERMINAL ruleTransform [13] =
{
//  1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13
    A, A, B, C, C, D, E, E, F, F, G, G, G
};
```

In our grammar there are 13 rules. In this one-dimensional array each index represents a production rule. Index 0 represents rule 1, index 1 represents rule 2, and forth. The array stored the values of the user defined enumeration NONTERMINAL. For example: ruletransform[0] means that the production rule 1 reduces to the nonterminal A.