Instituto Tecnológico y de Estudios Superiores de Monterrey

Final Project:
Compiler Design

PATITO++

Enrique Villa
A01193635

Santiago Castaño
A01193760
Castaño

3 de junio de 2020

Index

Project Description & User Manual

Project Description

Purpose

The purpose of this project is to integrate the knowledge acquired in the area of Computer Science, including the subjects of programming, data structure, computer theory and programming languages by building a compiler capable of receiving a set of commands and delivering an expected result. It will make use of our newly acquired knowledge of the basic concepts of the compilation process which include: Lexical Analysis, syntax analysis, semantic analysis, the translation process and generation of intermediate code, as well as execution environments and design of virtual machines.

Objective

The objective of this project is to design and create a compiler that is capable of receiving easy-to-write code that can help new programmers develop their first skills. It will also help us as developers to understand the difficulties and complexities of our own day to day tools. By having a command line output, it keeps commands and results relatively simple.

Scope

This language contains all the basic elements of a programming language with addition of matrix operations such as:

- Variable Declaration
- Function Declaration
- Assignment Expressions

- Void Function Calls
- Function Return
- Reading of Input
- Printing of Outputs
- Decision Statements (IF)
- Cycle Statements (While & For)
- Mathematical and Boolean Expressions
- Expressions between dimensioned objects such as matrices

Requirements

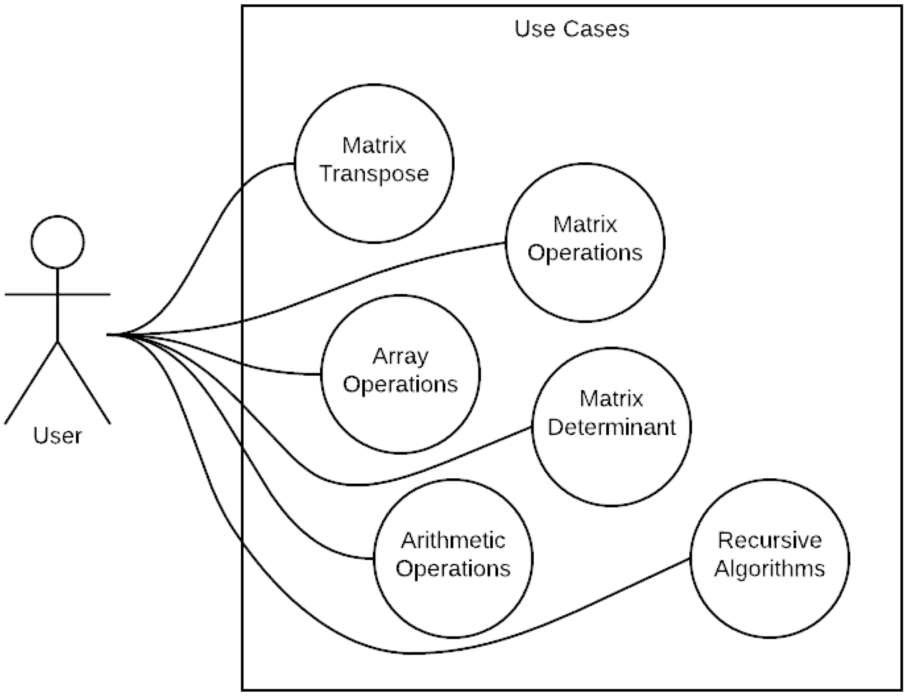
Functional Requirements

- Receive code initialized with “Program”
- Possible to generate matrices and operate with them
- Can declare functions
- Can call functions
- Can read values from command line
- Can print values to console
- Can transpose and invert a matrix with special operators.
- Errors must be displayed when necessary

Non-Functional Requirements

- Programs are read from .txt files
- Syntax must be easy to understand for a new programmer
- Project must be accessible by other coders as open source

General Use Cases



Test Cases Description

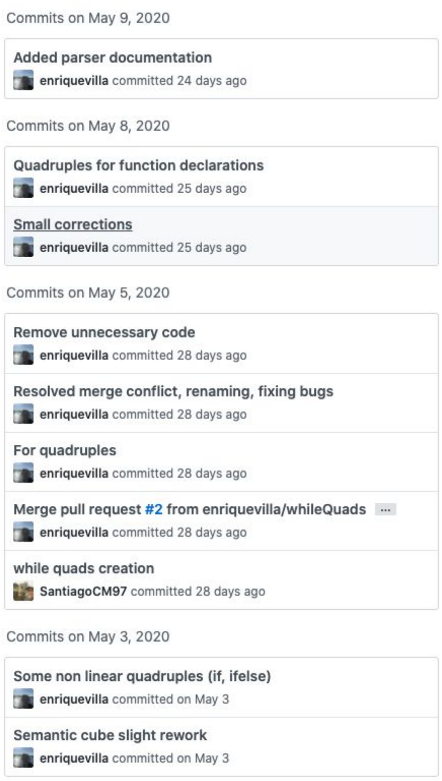
Test Name	Description
Cyclic factorial	Cyclic version of factorial calculation.
Recursive factorial	Recursive version of factorial calculation, using modules.
Cyclic Fibonacci	Cyclic version of calculating the N-th number in the Fibonacci sequence.
Recursive Fibonacci	Recursive version of calculating the N-th number in the Fibonacci sequence, using modules.
Bubble sort	Traditional bubble sort.
Array find	Finding a specific element in an array.

Matrix multiplication	Calculating the resulting matrix of a multiplication of matrices.
Matrix determinant	Calculating the determinant of a matrix.
Matrix transpose	Generating the transpose of a matrix.
Matrix inverse	Generating the inverse of a matrix.

Project Development Process

Git for version control and Github for remote contributions to the project. Pair programming done twice a week and the worklog was updated every week with an explanation of the commits made to the repository.

Commits (from Newest to Oldest):



Commits on May 9, 2020

Added parser documentation
enriquevilla committed 24 days ago

Commits on May 8, 2020

Quadruples for function declarations
enriquevilla committed 25 days ago

Small corrections
enriquevilla committed 25 days ago

Commits on May 5, 2020

Remove unnecessary code
enriquevilla committed 28 days ago

Resolved merge conflict, renaming, fixing bugs
enriquevilla committed 28 days ago

For quadruples
enriquevilla committed 28 days ago

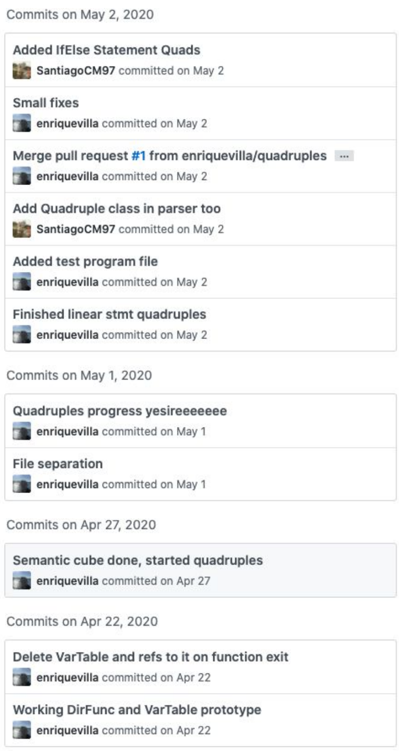
Merge pull request #2 from enriquevilla/whileQuads
enriquevilla committed 28 days ago

while quads creation
SantiagoCM97 committed 28 days ago

Commits on May 3, 2020

Some non linear quadruples (if, ifelse)
enriquevilla committed on May 3

Semantic cube slight rework
enriquevilla committed on May 3



Commits on May 2, 2020

Added IfElse Statement Quads
SantiagoCM97 committed on May 2

Small fixes
enriquevilla committed on May 2

Merge pull request #1 from enriquevilla/quadruples
enriquevilla committed on May 2

Add Quadruple class in parser too
SantiagoCM97 committed on May 2

Added test program file
enriquevilla committed on May 2

Finished linear stmt quadruples
enriquevilla committed on May 2

Commits on May 1, 2020

Quadruples progress yesireeeeeee
enriquevilla committed on May 1

File separation
enriquevilla committed on May 1

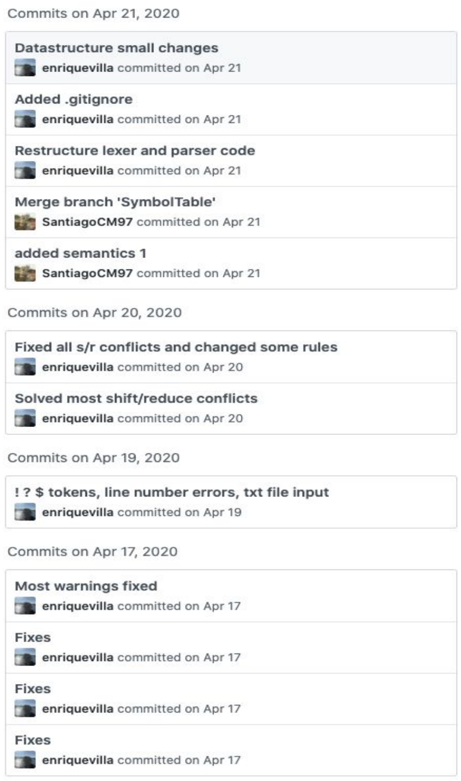
Commits on Apr 27, 2020

Semantic cube done, started quadruples
enriquevilla committed on Apr 27

Commits on Apr 22, 2020

Delete VarTable and refs to it on function exit
enriquevilla committed on Apr 22

Working DirFunc and VarTable prototype
enriquevilla committed on Apr 22



Commits on Apr 21, 2020

Datastructure small changes
enriquevilla committed on Apr 21

Added .gitignore
enriquevilla committed on Apr 21

Restructure lexer and parser code
enriquevilla committed on Apr 21

Merge branch 'SymbolTable'
SantiagoCM97 committed on Apr 21

added semantics 1
SantiagoCM97 committed on Apr 21

Commits on Apr 20, 2020

Fixed all s/r conflicts and changed some rules
enriquevilla committed on Apr 20

Solved most shift/reduce conflicts
enriquevilla committed on Apr 20

Commits on Apr 19, 2020

! ? $ tokens, line number errors, txt file input
enriquevilla committed on Apr 19

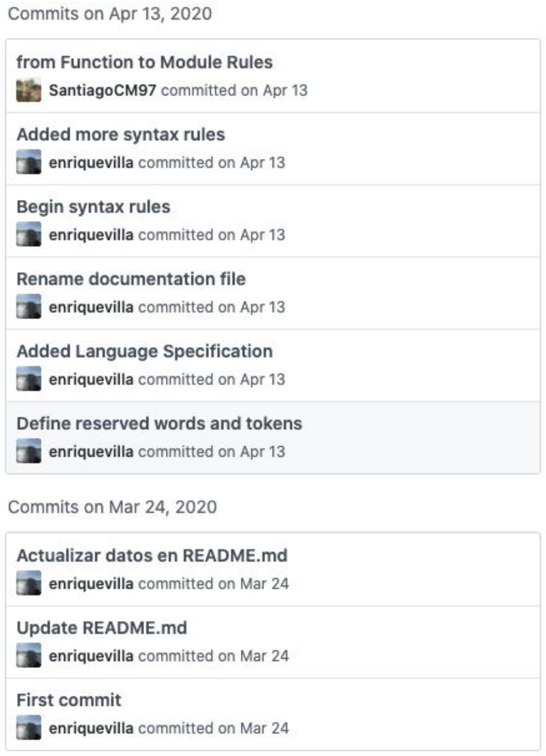
Commits on Apr 17, 2020

Most warnings fixed
enriquevilla committed on Apr 17

Fixes
enriquevilla committed on Apr 17

Fixes
enriquevilla committed on Apr 17

Fixes
enriquevilla committed on Apr 17



Commits on Apr 13, 2020

from Function to Module Rules
SantiagoCM97 committed on Apr 13

Added more syntax rules
enriquevilla committed on Apr 13

Begin syntax rules
enriquevilla committed on Apr 13

Rename documentation file
enriquevilla committed on Apr 13

Added Language Specification
enriquevilla committed on Apr 13

Define reserved words and tokens
enriquevilla committed on Apr 13

Commits on Mar 24, 2020

Actualizar datos en README.md
enriquevilla committed on Mar 24

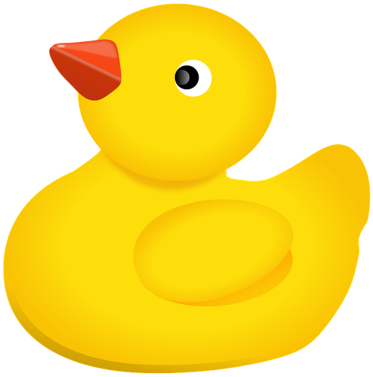
Update README.md
enriquevilla committed on Mar 24

First commit
enriquevilla committed on Mar 24

Language Description

Language Name

Patito++



Language Characteristics Description

Patito++ is a programming language that contains simple arithmetic, and boolean operations. It can be used to learn about programming with basic uses of temporal memory storage and input and output of results. Also use of arrays and two-dimensional arrays with basic arithmetic operations on them as well.

Compile-time and Execution-time Errors

Compile Time	
Syntax	Unexpected token in a specific line.
Type Mismatch	Type mismatch in assignment for a variable.
Condition Type Mismatch	Operands in a conditional operation are not the same type.
Operation Type Mismatch	Operands in arithmetic operation are incompatible

Undefined variable	Use of an undefined variable on a specific line
Redefinition of variable	An id has been declared before and cannot be defined again.
Unexpected Number of Arguments	Arguments on module use exceed those on module declaration
Type Mismatch Module	Variable assigned and Module type are not of compatible types.
Return on Void Function	A return statement appears on a void function
No Return on Type Function	A type function has no return value
Matrix accessed as array	A matrix variable only only used with one index
Type Mismatch in Index	Index used in array call is not Int
Variable not Subscriptable as Matrix	A non-matrix variable is called with two indexes
Variable not subscriptable as Array	Simple non-array variable is called with an index
Array Parameter in Module Call	Module call gets called with an array as parameter
Invalid print in array variable	A print operator gets passed an array as a parameter
Invalid operator on arrays	An array is used as an operand for an operator that doesn't accept arrays as operands.
Invalid operation in line	Any type of invalid operation
Dimensions do not match	Operation between dimensioned variables is called but their dimensions do not match
Invalid assignment to array variable	An array variable is assigned a non valid variable.
Array size must be positive	On array declaration, the array size is negative
Invalid determinant calculation	Invalid array dimensions for determinant calculation
Execution Time	

Index out of bounds	Array or matrix index access is out of the variable’s range of memory.

Compiler Description

Computing Equipment, Languages and Utilities

Brand: Dell

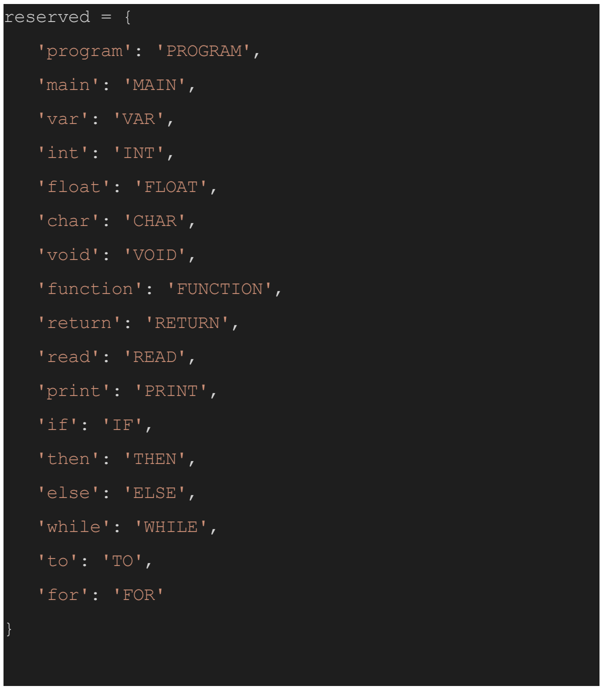
Model: G3 3579

Operating System: Windows 10

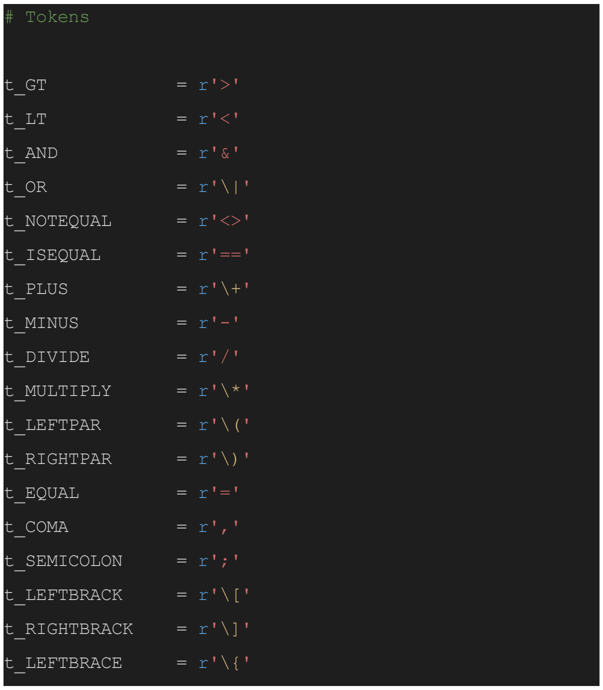
Language used: Python 3.8

Lexical and Syntax Analyzer: PLY.

Lexical Analysis Description

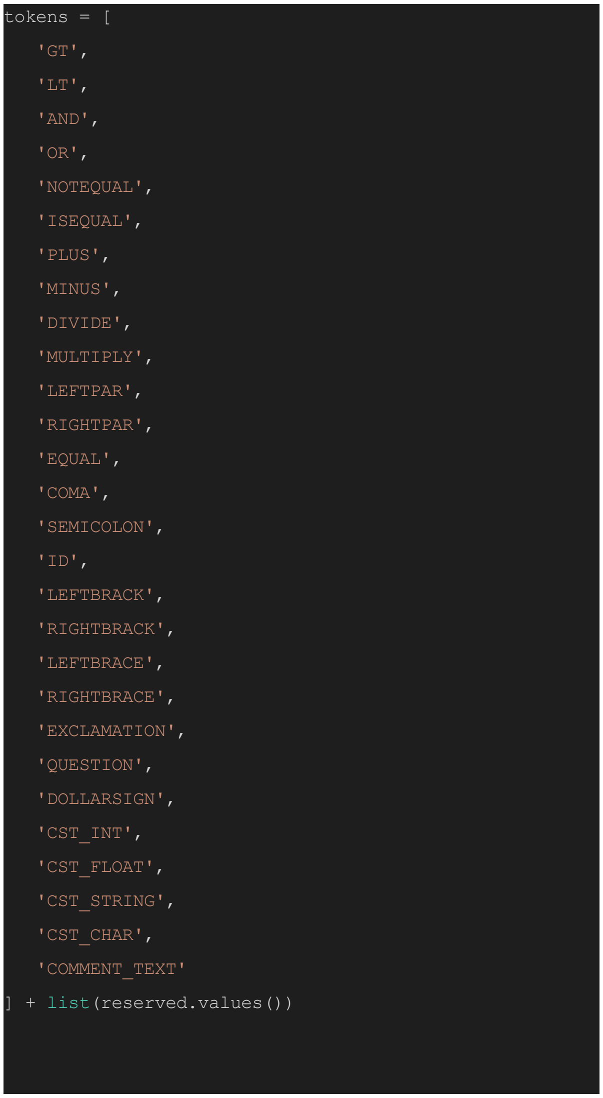


```
reserved = {
   'program': 'PROGRAM',
   'main': 'MAIN',
   'var': 'VAR',
   'int': 'INT',
   'float': 'FLOAT',
   'char': 'CHAR',
   'void': 'VOID',
   'function': 'FUNCTION',
   'return': 'RETURN',
   'read': 'READ',
   'print': 'PRINT',
   'if': 'IF',
   'then': 'THEN',
   'else': 'ELSE',
   'while': 'WHILE',
   'to': 'TO',
   'for': 'FOR'
}
```

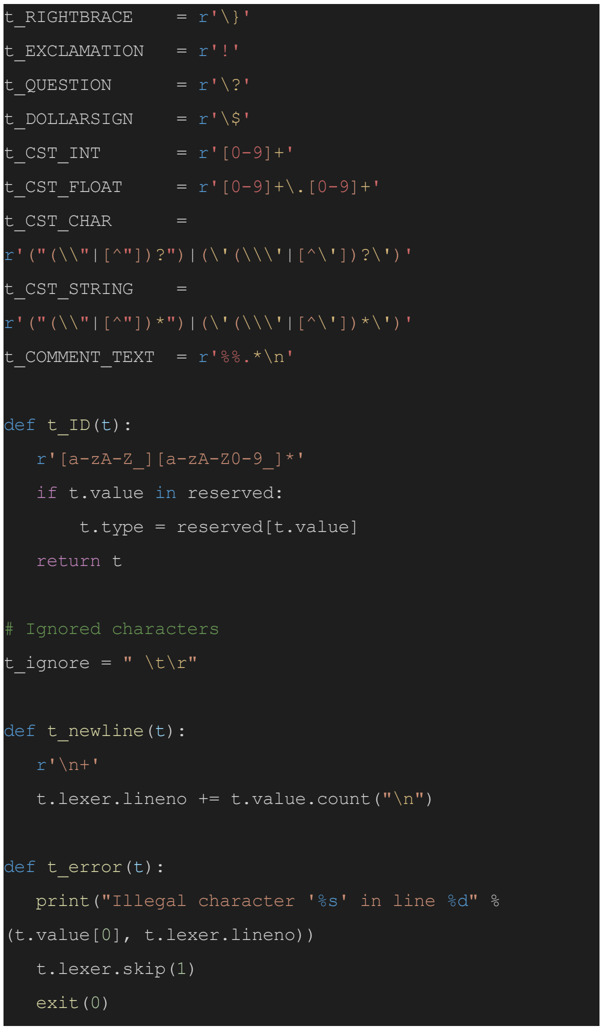


```
# Tokens

t_GT             = r'>'
t_LT             = r'<'
t_AND            = r'&'
t_OR             = r'\|'
t_NOTEQUAL       = r'<>'
t_ISEQUAL        = r'=='
t_PLUS           = r'\+'
t_MINUS          = r'-'
t_DIVIDE         = r'/'
t_MULTIPLY       = r'\*'
t_LEFTPAR        = r'\('
t_RIGHTPAR       = r'\)'
t_EQUAL          = r'='
t_COMA           = r','
t_SEMICOLON      = r';'
t_LEFTBRACK      = r'\['
t_RIGHTBRACK     = r'\]'
t_LEFTBRACE      = r'\{'
```



```
tokens = [
   'GT',
   'LT',
   'AND',
   'OR',
   'NOTEQUAL',
   'ISEQUAL',
   'PLUS',
   'MINUS',
   'DIVIDE',
   'MULTIPLY',
   'LEFTPAR',
   'RIGHTPAR',
   'EQUAL',
   'COMA',
   'SEMICOLON',
   'ID',
   'LEFTBRACK',
   'RIGHTBRACK',
   'LEFTBRACE',
   'RIGHTBRACE',
   'EXCLAMATION',
   'QUESTION',
   'DOLLARSIGN',
   'CST_INT',
   'CST_FLOAT',
   'CST_STRING',
   'CST_CHAR',
   'COMMENT_TEXT'
] + list(reserved.values())
```



```
t_RIGHTBRACE    = r'\}'
t_EXCLAMATION   = r'!'
t_QUESTION      = r'\?'
t_DOLLARSIGN    = r'\$'
t_CST_INT       = r'[0-9]+'
t_CST_FLOAT     = r'[0-9]+\.[0-9]+'
t_CST_CHAR      =
r'("(\\"|[^"])?")|(\'(\\\'|[^\'])?\')'
t_CST_STRING    =
r'("(\\"|[^"])*")|(\'(\\\'|[^\'])*\')'
t_COMMENT_TEXT  = r'%%.*\n'

def t_ID(t):
   r'[a-zA-Z_][a-zA-Z0-9_]*'
   if t.value in reserved:
       t.type = reserved[t.value]
   return t

# Ignored characters
t_ignore = " \t\r"

def t_newline(t):
   r'\n+'
   t.lexer.lineno += t.value.count("\n")

def t_error(t):
   print("Illegal character '%s' in line %d" %
(t.value[0], t.lexer.lineno))
   t.lexer.skip(1)
   exit(0)
```

Syntax Analysis Description

‘ | ’ means it’s another branch of the syntax options

```
'program : PROGRAM ID SEMICOLON declaration programFunc main'
'''programFunc : function programFunc | '''
'main : MAIN LEFTPAR RIGHTPAR LEFTBRACE declaration statement RIGHTBRACE'
'assignment : ID dimArray EQUAL hyperExpression SEMICOLON'
'''declaration : VAR declarationPrim | '''
'''declarationPrim : primitive vars SEMICOLON declarationPrim | '''
```

```
'''primitive : INT | FLOAT | CHAR '''
'return : RETURN LEFTPAR hyperExpression RIGHTPAR SEMICOLON'
'if :IF LEFTPAR hyperExpression RIGHTPAR THEN LEFTBRACE statement RIGHTBRACE ifElse'
'''ifElse : ELSE LEFTBRACE statement RIGHTBRACE | '''
'comment : COMMENT_TEXT'
'while : WHILE LEFTPAR hyperExpression RIGHTPAR LEFTBRACE statement RIGHTBRACE '
'for : FOR forAssignment TO hyperExpression LEFTBRACE statement RIGHTBRACE '
'forAssignment : ID EQUAL CST_INT '
'vars : ID varsArray varsComa'
'''varsComa : COMA vars | '''
'''varsArray : LEFTBRACK CST_INT RIGHTBRACK varsMatrix  | '''
'''varsMatrix : LEFTBRACK CST_INT RIGHTBRACK | '''
'function : functionType ID LEFTPAR param RIGHTPAR LEFTBRACE declaration statement
RIGHTBRACE'
'''functionType : FUNCTION primitive | FUNCTION VOID '''
'''param : primitive ID addFuncParams functionParam | '''
'''functionParam : COMA param | '''
'''cst_prim : CST_INT | CST_FLOAT | CST_CHAR '''
'''hyperExpression : superExpression opHyperExpression hyperExpressionNested
| superExpression opMatrix | superExpression '''
'''hyperExpressionNested : superExpression opHyperExpression hyperExpressionNested |
superExpression '''
'''opMatrix : EXCLAMATION | QUESTION | DOLLARSIGN '''
'''opHyperExpression : AND| OR '''
'''superExpression : exp opSuperExpression exp | exp '''
'''opSuperExpression : GT | LT | NOTEQUAL | ISEQUAL '''
'''exp : term expFunction | term '''
'''expFunction : PLUS exp | MINUS exp '''
'''term : factor termFunction | factor '''
'''termFunction : MULTIPLY term | DIVIDE term '''
'''factor : LEFTPAR hyperExpression RIGHTPAR | cst_prim | module | ID dimArray'''
'read : READ LEFTPAR id_list RIGHTPAR SEMICOLON'
'id_list : ID dimArray id_listFunction'
'''id_listFunction : COMA id_list | '''
'print : PRINT LEFTPAR printFunction RIGHTPAR SEMICOLON'
'''printFunction : print_param COMA printFunction2 | print_param '''
'printFunction2 : printFunction'
'''print_param : hyperExpression | CST_STRING '''
```

```
'''statement : return | if statement | comment statement | read statement | print
statement | assignment statement | module SEMICOLON statement | for statement |
while statement | '''
'module : ID LEFTPAR moduleFunction RIGHTPAR '
'''moduleFunction : hyperExpression COMA moduleFunction | hyperExpression | '''
'''dimArray : LEFTBRACK hyperExpression RIGHTBRACK dimMatrix | '''
'''dimMatrix : LEFTBRACK hyperExpression RIGHTBRACK | '''
```

Code Generation and Semantic Analysis Description

Operations are made using quadruples that are generated using the following format:

- (operator, left_operand, right_operand, result)

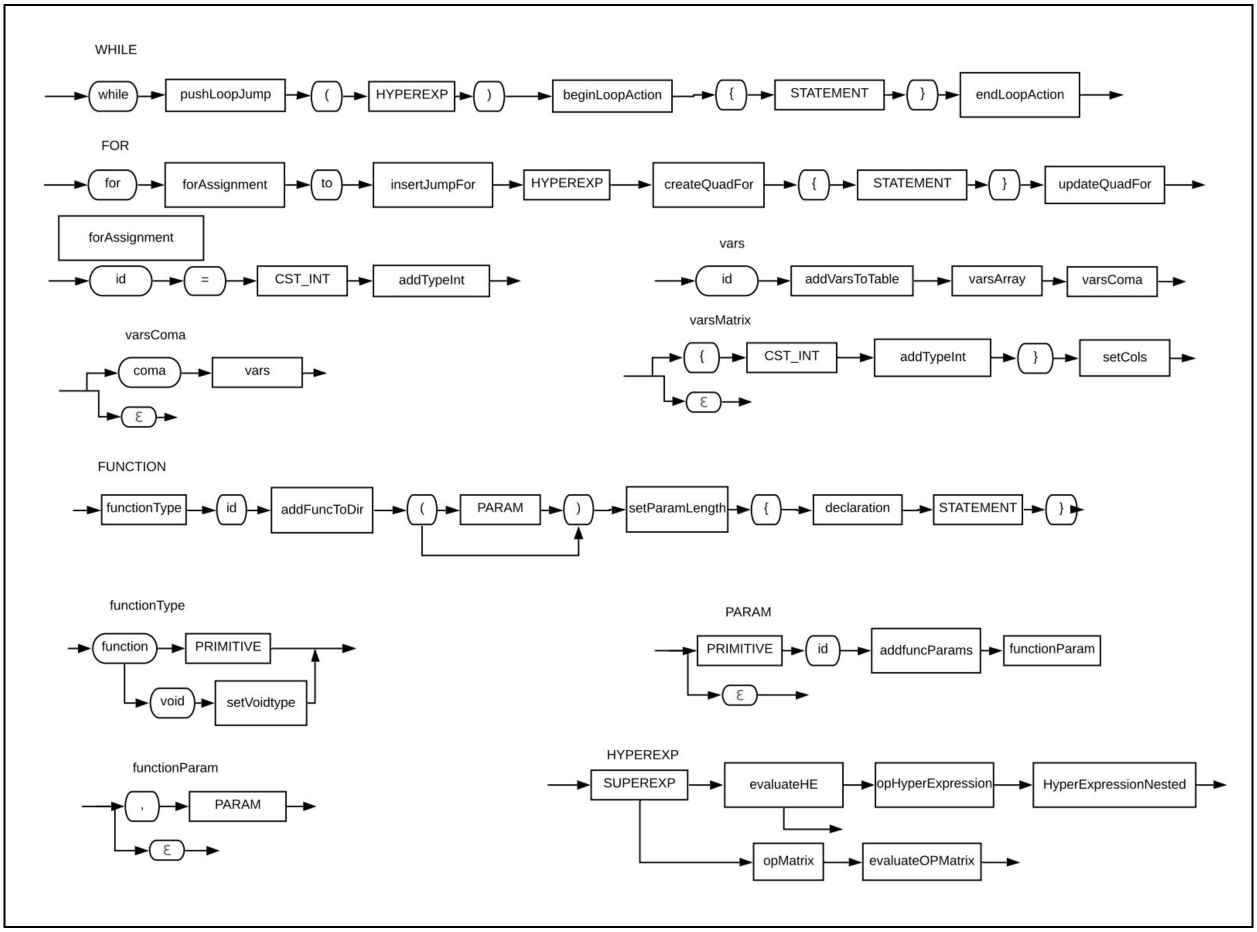
In this format, the operator can be any operator from the range of operators our language supports, which include mathematical and logical operators, or special operators like GOTO or GOTOF used in loops and conditions, GOSUB and ERA, used for handling module calls and context switching, and a few more.

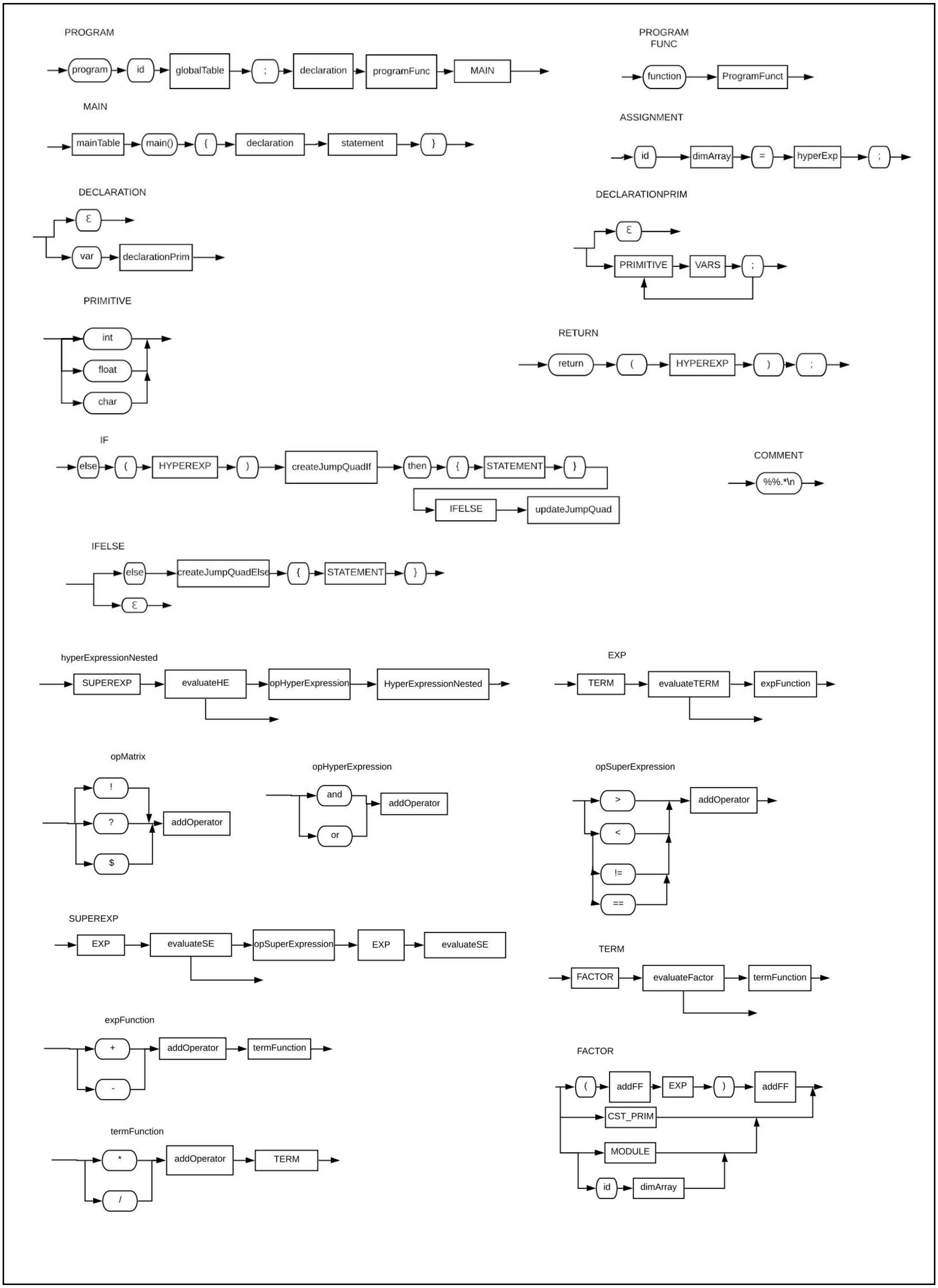
The operands and result are always memory addresses, which represent types of operands depending on their address value range. The addresses for each type of variable or constant were established as follows:

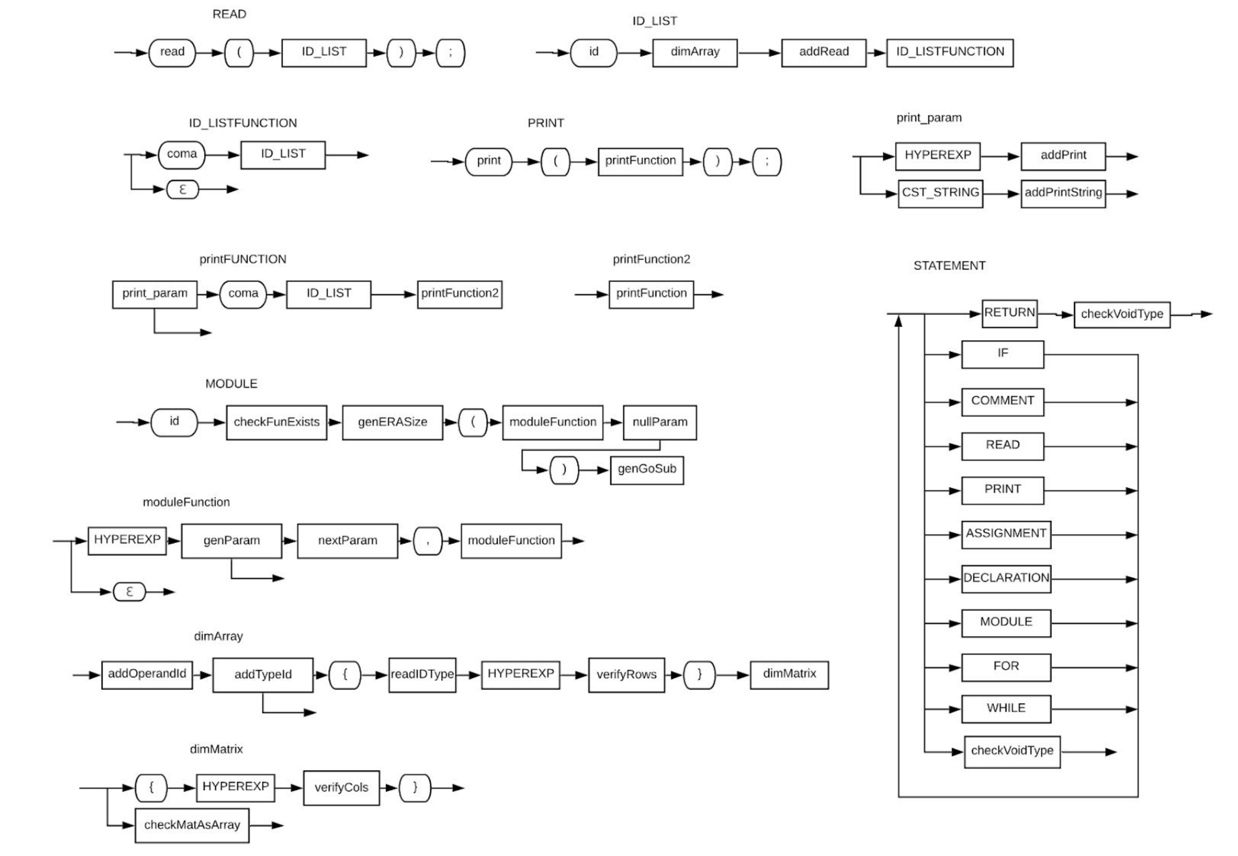
- Global int: 0-999
- Global float: 1000-1999
- Global char: 2000-2999
- Local int: 3000-3999
- Local float: 4000-4999
- Local char: 5000-5999
- Temporary int: 6000-6999
- Temporary float: 7000-7999
- Temporary char: 8000-8999
- Constant int: 9000-9999

- Constant float: 10000-10999
- Constant char: 11000-11999
- Temporary pointer: 12000-12999
- Void: 13000-13999

Syntax Diagrams and Action Description




PROGRAM
program
id
globalTable
;
declaration
programFunc
MAIN
PROGRAM FUNC
function
ProgramFunct
MAIN
mainTable
main()
{
declaration
statement
}
ASSIGNMENT
id
dimArray
=
hyperExp
;
DECLARATION
Ɛ
var
declarationPrim
DECLARATIONPRIM
Ɛ
PRIMITIVE
VARS
;
PRIMITIVE
int
float
char
RETURN
return
(
HYPEREXP
)
;
IF
else
(
HYPEREXP
)
createJumpQuadIf
then
{
STATEMENT
}
IFELSE
updateJumpQuad
COMMENT
%%.*\n
IFELSE
else
createJumpQuadElse
{
STATEMENT
}
Ɛ
hyperExpressionNested
SUPEREXP
evaluateHE
opHyperExpression
HyperExpressionNested
EXP
TERM
evaluateTERM
expFunction
opMatrix
!
?
$
addOperator
opHyperExpression
and
or
addOperator
opSuperExpression
>
<
!=
==
addOperator
SUPEREXP
EXP
evaluateSE
opSuperExpression
EXP
evaluateSE
TERM
FACTOR
evaluateFactor
termFunction
expFunction
+
-
addOperator
termFunction
FACTOR
(
addFF
EXP
)
addFF
CST_PRIM
MODULE
id
dimArray
termFunction
*
/
addOperator
TERM



Name	Definition
globalTable	Initialize program and create variable table
mainTable	Add main to varTable and initialize main function's properties. Update main quadruple to jump to start of the program.
assignment	Generate quadruple in the respective varTable
declaration	Set "start" quadruple for a function
primitive	Change the current type for a declaration
createJumpQuadIf	Check type and value for expression and generate jumping quadruple
updateJumpQuad	Update jumping quadruple with id of quad to jump to
createJumpQuadElse	Create jumping quad for else statement

pushLoopJump	Push id of quadruple to jumping stack
beginLoopAction	Check expression result type, generate quadruple and push jump id to jump stack
endLoopAction	Generate quadruple after while statement finishes and update gotof with id at the end of loop quad.
insertJumpFor	Pushes id of quadruple to jump to into jump stack
createQuadFor	Add GOTOF to to quadruples
updateQuadFor	Update GOTOF quadruple with ID of quad to jump to for FOR
forAssignment	Add iterator to constants table and create iterating variable
addVarsToTable	Add current ID to varTable with its type.
varsArray	Specific for array declaration, stores the base address in the constants of the variable table
setRows	Set amount of rows for dimensioned variable
setCols	Set amount of columns for dimensioned variable
function	Create ENDFUNC quadruple and set local variable table.
addFuncToDir	Verify function type and insert function to funcDir with type, varTable and parameters.
setVoidType	Set current type of function as Void
addFuncParams	Add a list of param types to the function’s scope
setParamLength	Set the amount of params in the function.
addTypeInt	Save int to constants table and push operand to operand stack
addTypeFloat	Save float to constants table and push operand to operand stack
addTypeChar	Save char to constants table and push operand to operand stack
evaluateOpMatrix	Evaluates operator and operands of a dimensioned variable operation.
evaluateHE	Evaluates operator and operands of boolean expressions of type AND and OR.
evaluateSE	Evaluates operator and operands of boolean expressions of type >, <, ==, and <> (not equals).

evaluateTerm	Evaluates operator and operands of the type + and - for variables and dimensioned variables
evaluateFactor	Evaluates operator and operands of the type * and / for variables and dimensioned variables (only multiplication)
addOperator	Pushes a read operator to the operator stack
addFF	Pushes a parentheses to the operator stack as a Fake bottom
removeFF	Pops the parentheses from the operator stack
addRead	Generates a “READ” quadruple and pushes it to quads list
addPrint	Generates a “PRINT” quadruple and pushes it to quads list
addPrintString	Reads a string and stores it in the constants table to later be printed by the PRINT operator
checkVoidType	Throws an error if a “return” is present in a void function
checkNonVoidType	Throws and error if there is no “return” in a non-void function
checkFuncExists	Verifies a function exists in the funcDir and pushes the module operator to te operator stack
genERASize	Creates the ERA quadruple with the address of the function to be called.
nullParam	Throws error if there is a missing parameter in a function call
genGoSub	Creates the GoSub quadruple with the address of the function to be called and saves the result in a tmpAddress if its non-void.
genParam	Creates the PARAM quadruple with the operand that is being read
nextParam	Adds 1 to the param iterator
dimArray	Pops the id and scope of the matrix or array to use
addOperandID	Pushes the id of the array to the array ID stack and the scope to the scope stack
addTypeId	Pushes the types of the matrix to the types stack
readIDType	Checks types of operands and throws error if there is a mismatch. Also verifies operand is an array.
verifyRows	Generates the verify quad of the index being used to see if it is inside the correct range of row numbers

dimMatrix	Generates the quad to add the base address and the constant of the index being used to access the correct memory space.
verifyCols	Generates the verify quad of the second index being used to see if it is inside the correct range of row numbers
checkMatAsArray	Throws error if a matrix only has one index being used.

Semantic Characteristics Tables:

Addition, subtraction and multiplication

+, -, *	int	float	char
int	int	float	error
float	float	float	error
char	error	error	error

Division

/	int	float	char
int	float	float	error
float	float	float	error
char	error	error	error

Less than, greater than

<, >	int	float	char
int	int	int	error
float	int	int	error
char	error	error	error

Not equal, equal to

<>, ==	int	float	char
int	int	int	error
float	int	int	error
char	error	error	int

And, or

The & and | operators function identical to Python, where the value of the left operand is taken in an or operation, and the value of the right operator is taken in an and operation (e.g. The operation “a” | 1 gives “a”, whereas the operation “a” & 1 gives 1, however if you have a 0, which is false, in the left side of an or operation you will get the right side, and if you have a 0 in the right side of an and operation, you will always get 0).

Compile-time Memory Administration Description

In compile-time, we rely heavily on Python’s excellent hashtable or “dictionary” data structure for storing all necessary information about variables and functions. The reason behind using this data structure for most of the project is the fact that it has a fantastic search time of O(1), which is just what we need for efficiency. Here is a brief theoretical example of how they would look during compilation time:

```
functionDir["global"] =>
    "global": {
        "type": "void",
        "vars": variableTable["global"] => "i": {
                                                "type": "int",
                                                "address": 0
                                            }, more variables...
    }

functionDir["uno"] =>
    "uno": {
        "type": "int",
        "params": Queue[int, int, float],
        "paramsLength": len(params),
        "vars": variableTable["uno"] => "x": {
                                            "type": "int",
                                            "address": 3000
                                        }, more variables...
    }

functionDir["main"] =>
    "main": {
        "type": "void",
        "vars": variableTable["main"] => "c": {
                                              "type": "char",
                                              "address": 5000
                                         }, more variables...
    }
```

Here, we are using function names, or scopes, as the keys in the function directory hashtable. Taking “uno” as an example, we can see functionDir[“uno”] tells us that it is a function of type int, and has 3 parameters of type int, int and float. If we access functionDir[“uno”][“vars”], we would get the variable table of this function, which we can see is a reference to the variable table of “uno” and holds all the variables with their addresses and types, who are assigned during the compilation process.

In variableTable[“constants”], we store the constants identified in the parsing process. The keys are the values themselves and they store their addresses. For example, variableTable[“constants”][“a”] would contain an address of 11000, for constant chars.

As stated previously, our quadruples structure is the following:

- **(operator, left_operand, right_operand, result)**

These are constructed using the class constructor Quadruple, then stored in a Quadruples class that stores all these Quadruple objects.

Virtual Machine Description

Computing Equipment, Languages and Utilities

Brand: Dell

Model: G3 3579

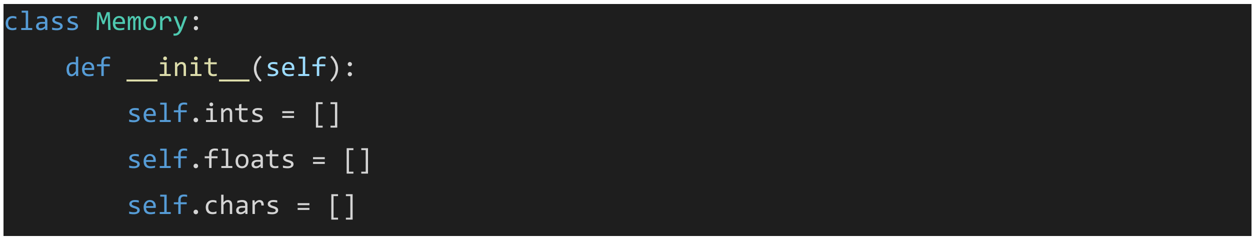
Operating System: Windows 10

Language used: Python 3.8

Lexical and Syntax Analyzer: PLY.

Execution-time Memory Administration Description

During execution, we rely on a Memory class that has a list of int, float and char type variables.



```
class Memory:
    def __init__(self):
        self.ints = []
        self.floats = []
        self.chars = []
```

In the virtual machine we initialize a global, local and temporary memory using the Memory class constructor. The result of this is a global memory, local memory and temporary memory object, each of these will have a list of int, float and char. We also make use of the constant table obtained during compilation, although we invert the keys with the addresses within them to have instead the addresses as the key and use those addresses to get the actual value of the constant, since we receive the quadruples with addresses in the virtual machine.

Language Functionality Tests

Cyclic Factorial	

Code:
program fact;
 	
main() {
    var int c, result;
    result = 1;
    for c = 1 to c < 7 {
        result = result * c;
    }
    print(result);
}
	Result
	
	Compiled successfully
	720

Recursive Factorial	

Code:
 	
program fact;
 	
function int factorial(int a) {
    if (a > 1) then {
        return(a * factorial(a - 1));
    }
    return(1);
}
 	
main() {
	Result
	
	Compiled succesfully
	120
	120

```
    var int c;
    c = factorial(5);
    print(factorial(5));
    print(c);
}
```

Cyclic Fibonacci

Code:

```
program fibonacciCyclic;

main() {
    var int nthTerm, first, second, result, i;
    first = 0;
    second = 1;

    %% nthTerm = term of the fibonacci series
    nthTerm = 10;

    %% adjust nthTerm for the cycle
    nthTerm = nthTerm + 1;

    for i = 2 to i < nthTerm {
        result = first + second;
        first = second;
        second = result;
    }

    print(result);
}
```

Result
34

Recursive Fibonacci

Code:

```
program fibonacciRecursive;

function int fibonacci(int n) {
    var int a, b;
    if (n < 2) then {
        return(n);
    }
    a = fibonacci(n - 1);
    b = fibonacci(n - 2);
    return(a + b);
}
```

Result
55

```
main() {
    print(fibonacci(10));
}
```

Bubble Sort

Code:

```
program bubblesort;
var int array1[5];

main() {
    var int sorted, i, changed, aux;
    sorted = 0;
    i = 0;
    changed = 0;

    %% assign array
    array1[0] = 2;
    array1[1] = 8;
    array1[2] = 5;
    array1[3] = 33;
    array1[4] = 25;

    while (sorted == 0) {
        if (array1[i] > array1[i + 1]) then {
            aux = array1[i];
            array1[i] = array1[i + 1];
            array1[i + 1] = aux;
            changed = 1;
        }
        if (i == 3) then {
            if (changed == 1) then {
                i = 0;
                changed = 0;
            } else {
                sorted = 1;
            }
        }
        i = i + 1;
    }

    for i = 0 to i < 5 {
        print(array1[i]);
    }
}
```

Result

```
2
5
8
25
33
```

Array Find

Code:

```
program arrayfind;
var int array1[6];

function int find(int a, int j) {
    if (j < 0) then {
        return(0 - 1);
    }

    if (array1[j] == a) then {
        return(j);
    }

    return(find(a, j - 1));
}

main() {
    var int result;

    %% assign array
    array1[0] = 2;
    array1[1] = 8;
    array1[2] = 5;
    array1[3] = 33;
    array1[4] = 25;
    array1[5] = 9;

    result = find(25, 5);

    print(result);
}
```

Result

```
4
```

Matrix Multiplication

Code:

```
program matrixmultiplication;

main() {
    var int matrix1[3][4], matrix2[4][6], result[3][6], i, j;
```

Result

```
Compiled successfully
14
20
26
20
```

```
    %% matrix1 assigning
    for j = 0 to j < 4 {
        for i = 0 to i < 3 {
            matrix1[i][j] = i + j;
        }
    }


    %% matrix2 assigning
    for j = 0 to j < 6 {
        for i = 0 to i < 4 {
            matrix2[i][j] = i + j;
        }
    }

    result = matrix1 * matrix2;

    for j = 0 to j < 6 {
        for i = 0 to i < 3 {
            print(result[i][j]);
        }
    }
}
```

```
30
54
32
50
68
38
60
82
44
70
96
```

Matrix Determinant

Code:

```
program matrixdeterminant;

main() {
    var float result;
        int i, j, matrix[3][3];

    %% assign matrix
    matrix[0][0] = 2;
    matrix[1][0] = 2;
    matrix[2][0] = 1;
    matrix[0][1] = 0 - 3;
    matrix[1][1] = 0;
    matrix[2][1] = 4;
    matrix[0][2] = 1;
    matrix[1][2] = 0 - 1;
    matrix[2][2] = 5;

    result = matrix$;

    print(result);
}
```

Result

```
49.000000000000014
```

Matrix Transpose

Code:

```
program matrixtranspose;

main() {
    var int i, j, matrix[2][3], result[3][2];

    %% assign matrix
    matrix[0][0] = 1;
    matrix[1][0] = 2;
    matrix[0][1] = 3;
    matrix[1][1] = 4;
    matrix[0][2] = 5;
    matrix[1][2] = 6;

    print("Matrix assigned:");
    for j = 0 to j < 3 {
        for i = 0 to i < 2 {
            print(matrix[i][j]);
        }
    }

    result = matrix!;

    print("Result matrix:");
    for j = 0 to j < 2 {
        for i = 0 to i < 3 {
            print(result[i][j]);
        }
    }
}
```

Result

```
Compiled successfully
Matrix assigned:
4
5
6
Result matrix:
1
3
5
2
4
6
```

Matrix Inverse

Code:

```
program matrixinverse;

main() {
    var int i, j, matrix[3][3];
        float result[3][3];

    %% assign matrix
    matrix[0][0] = 0 - 1;
    matrix[1][0] = 2;
    matrix[2][0] = 3;
    matrix[0][1] = 0 - 2;
```

Result

```
Result matrix:
0.043478260869565 26
-0.3043478260869566
0.2173913043478261
0.7826086956521741
-0.47826086956521746
-0.08695652173913046
```

matrix[1][1] = 1;
matrix[2][1] = 4;
matrix[0][2] = 2;
matrix[1][2] = 1;
matrix[2][2] = 5;
 	
result = matrix?;
 	
print("Result matrix:");
for j = 0 to j < 3 {
for i = 0 to i < 3 {
print(result[i][j]);
}
}
}
	-0.1739130434782609
	0.21739130434782608
	0.13043478260869568

Project Files Documentation

Module Name	Details
datastructures.py	**Purpose**: Declares and initializes the main structures that will be used throughout the project such as: ● Function Directory ● Variable Table ● Semantic Cube (Filled in this same module) ● Operators Stack ● Operands Stack ● Types Stack ● Array or Matrix Operands Stack ● Type to Address Mapping dictionary ● Types IDs Map ● Operators List Also creates the Stack() and Queue() Python objects to be used in multiple modules. **Used in:** ● parser.py: Imports initialized objects ● quadruples.py: Imports the Stack data structure ● virtualmachine.py: Imports the variableTable
error.py	**Purpose**: Declares and exports an Error() class which centralizes error displays. All errors in compile-time have the line number where the error happens passed as an argument to display where the error occurred, whereas execution-time errors display only the type of error.

	Used in: ● parser.py: Imports Error() class and uses it in the syntax/grammar functions
lexer.py	**Purpose**: Makes use of the Lex module of PLY. Declares the reserved words of the language in a dictionary. Also lists all the tokens to symbolize all the native operators of the language. Finally declares the regular expressions for each token. This lexer is then passed to the parser, which uses this lexical analysis to perform its syntax analysis. **Used in:** ● parser.py: Imports lexer to make use of the tokens and the line numbers to report them to the Error class.
memory.py	**Purpose:** Declares a Memory() class that instantiates a block of memory with a single array of each of the types ints, floats, and chars. Also implements the getter and setter methods to manage memory inserts and extensions for arrays of each type. **Used in:** ● virtualmachine.py: virtual machine declares three instances of Memory() class: ○ globalMem which stores all global variables of the used code ○ localMem which stores the variables declared inside a function ○ tempMem which stores all the results from expressions to be used later in the code.
parser.py	**Purpose**: The parser's main purpose is to transform the literal code written in Patito++ into intermediate code in the form of quadruples. It makes use of the Yacc module of PLY to achieve this. **Parser.py is the file that must be run to actually run the Patito++ code and compile it.** **Imports**: ● lexer.py ● Yacc module ● datastructures.py ● quadruples.py ● error.py ● virtualmachine.py
quadruples.txt	**Purpose**: Declares the Quadruple() and Quadruples() classes.

	The **Quadruple** class is capable of building an object with an operator, a left operand, a right operand and a result. The **Quadruples** class holds the quadruples list, the jumps stack and can manipulate the list to then pass it to the virtual machine. **Used in:** ● parser.py: imports quadruples to create them on their respective actions ● virtualmachine.py: imports the quadruples list to iterate it and execute the code.
virtualmachine.py	**Purpose**: Declares the runner_duckie() method which is in charge of iterating through the entire quadruples list. With each quadruple it reads, it then executes the instruction related to its operator. The virtual machine is also responsible for the creation of the Memory() class instances, the local memory stack, the pointer stack and the constants memory map. **Used in:** ● parser.py: imports the runner_duckie() method and runs it after all code has been analysed and parsed to execute it.