Tecnológico de Monterrey

Final Project
Giga

Alejandro Mario Sánchez Pérez A01191434
Manuel Sañudo Villaseñor A01192241

Table of Contents

Project Description

Project's purpose and vision

Giga's vision is to make a programming language that can be easily used and that can also be easy to understand. This language uses Google's web-based visual programming editor, Blockly, as it's front-end and Python as it's back-end.

Language's objective and application area

Our objective is to create a language that is easy to understand, to be able to learn how to code. Giga is a language that can be used by kids who are in elementary school, so that they can learn about code and programming from an early age.

Project's scope

Because Giga's objective is to be a simple and easy to understand language, which can be useful when you want to learn about programming, the scope could appear to be limited, but we tried to make it so that it can do everything that you need to do when you're starting out. You can create local and global variables and use them to calculate expressions for 'while' cycles or 'if' conditionals, you can create your own functions and they can even call themselves recursively. You can also read values from the console into variables or print out values.

Requirements analysis

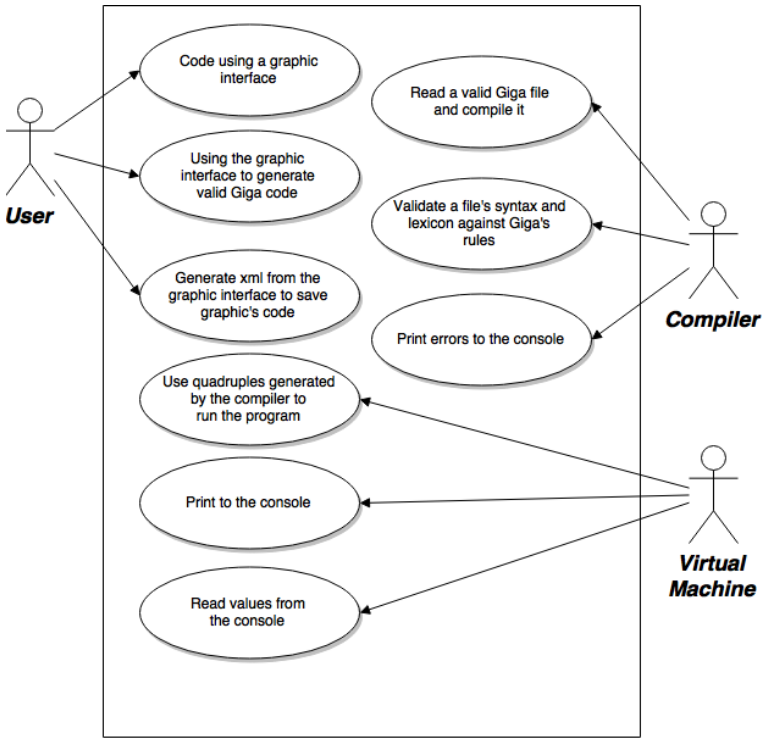
Functional requirements:

- Compiler must be able to compile valid code written with Giga's rules
- Compiler must crash if there is an error, it should never generate invalid code
- Compiler must be able to generate code for variables, arrays, expressions, if/elseif/else and while statements, and functions
- Virtual machine must be able to successfully execute all quadruples generated by the compiler
- Blockly interface must generate valid code when following Giga's rules to make a program
- Blockly must be able to create all possible code blocks

Non-functional requirements

- Giga's rules must be easy to understand
- Compiler must be able to compile from a text file
- Virtual machine must run immediately after a successful compilation

General use cases



Description of main test cases

- Tests for lexicon and syntax
- Tests for semantics
- Quadruple generation of expressions
- Quadruple generation of if/elseif/else statements
- Quadruple generation of while statements
- Quadruple generation of nested if and while statements
- Quadruple generation of function declarations
- Quadruple generation of function calls
- Virtual machine execution of expressions
- Virtual machine execution of if and while statements
- Quadruple generation of array declarations
- Quadruple generation of array indexing
- Virtual machine execution of functions
- Virtual machine execution of arrays

Logs of the process followed for Giga's development

Commit	Date	Author	Comments
1-3	Sept 21	Alejandro Sanchez	Initial commit, adds documents, adds first delivery folder
4	Sept 22	Manuel Sañudo	Adds PLY
5	Sept 24	Manuel Sañudo	Adds grammar pdf, starts work on PLY
6	Oct 1	Alejandro Sanchez	Adds grammar diagrams

Commit	Date	Author	Comments
7-10	Oct 1	Manuel Sañudo	Finished grammar, parses from file, adds test file
11-12	Oct 1	Alejandro Sanchez	Adds functions to PLY file, updates diagrams
13	Oct 4	Alejandro Sanchez	Adds variable index
14	Oct 6	Manuel Sañudo	Updates test, adds functions index, fixes variable index
15	Oct 8	Alejandro Sanchez	Adds semantic cube
16-17	Oct 8	Manuel Sañudo	Adds Giga.html, closure library, files for custom blocks
18	Oct 8	Alejandro Sanchez	Remove string from types
19	Oct 9	Alejandro Sanchez	Adds custom functions to Blockly
20	Oct 15	Alejandro Sanchez	Updates Blockly, Giga.html, adds array to blocks
24	Oct 15	Manuel Sañudo	Create fourth delivery folder, starts code generation
25	Oct 15	Alejandro Sanchez	-merge-
26-27	Oct 22	Alejandro Sanchez	Fixes negative number assignment, modifies Blockly
28	Oct 22	Manuel Sañudo	Starts work on quadruples
29	Oct 22	Alejandro Sanchez	Finished Blockly
30	Oct 22	Manuel Sañudo	Adds && and || to Cube, progress with quadruples
31	Oct 29	Alejandro Sanchez	Adds fifth delivery folder
32	Oct 29	Manuel Sañudo	Declares initial values for variable locations
33	Oct 30	Manuel Sañudo	Quadruples save with memory locations
34	Oct 31	Alejandro Sanchez	Adds Machine and Memory
35	Oct 31	Manuel Sañudo	Adds quadruples for if/elseif/else and while
36	Nov 1	Alejandro Sanchez	Adds txt button
37	Nov 1	Manuel Sañudo	Adds support for nested if's
38	Nov 1	Alejandro Sanchez	-merge-
39	Nov 1	Manuel Sañudo	Global variables must be declared before functions
40	Nov 3	Manuel Sañudo	Changes char to string
41	Nov 5	Alejandro Sanchez	Adds virtual machine and memory allocation
42	Nov 5	Manuel Sañudo	Adds support for assigning a function call
43	Nov 5	Manuel Sañudo	Adds quadruples for printing and reading
44	Nov 5	Manuel Sañudo	Changes quadruples to use variable directions directly
45-46	Nov 5	Alejandro Sanchez	Memory and machine work together, first memory test
47	Nov 7	Manuel Sañudo	Adds support for declaring arrays
48	Nov 8	Manuel Sañudo	Fixes string regex

Commit	Date	Author	Comments
49	Nov 12	Alejandro Sanchez	Memory works
50-52	Nov 12	Manuel Sañudo	Adds support for assigning/getting values of arrays
53	Nov 19	Alejandro Sanchez	Fixes arrays in memory
54-55	Nov 20	Alejandro Sanchez	Fixes arrays in memory, adds tests
56	Nov 20	Manuel Sañudo	Fixes expressions with parentheses
57	Nov 20	Alejandro Sanchez	Adds final tests
58	Nov 20	Manuel Sañudo	Updates diagrams

Full logs available here: https://github.com/sanchezz93/Giga-Compiler/commits

I think that this project has been one of the most demanding projects that I've had in my college years, but this has been one of the projects that I've liked the most. Working every Saturday in this project was a challenge, something I don't normally do for a project in school, it opened my mind and reminded me of the multiple courses that I've taken in throughout the years. It showed me that if put my mind to a project I would be able to achieve it and will finish it on time. This project showed me how a real compiler works and makes me think of all of the things that a developer has to put in mind when creating a language.

Alejandro Sanchez - A01191434

I have learned a lot while working on Giga, it's extremely interesting to be able to understand a compiler now, after years of coding but never really knowing what was going on behind the scenes. It also was the largest project I've ever worked on for school. Working on it every week to try to meet every deadline was essential, without the predefined deadlines it would've been much more difficult to achieve what we have. Working on it every week also made the semester a very long and stressful one, but it's very rewarding at the end to be able to see the finished result, one that both of us as a team are proud of.

Manuel Sañudo - A01192241

Language Description

Name of language

The name of the language we created is "Giga".

General description of the main characteristics of the language

Giga's syntax is very similar to the syntax used by C/C++, but since the scope of the language is much smaller, it is simpler and many of the complex expressions available in C are missing in Giga. But simpler is better in this case, because that makes it easier for younger people to learn and understand it. One notable difference between Giga and C is that while you can simply declare a variable on C, with Giga you must set an initial value, we made this choice because this ensures that you cannot use a variable that hasn't been initialized already.

Description of errors that may occur during compile or run time

The compiler will print an error and exit if it finds any of the following errors:

- Syntax error
- Using an operator with incompatible operands
- Using an undeclared variable or function
- Assigning an incompatible value to a variable
- Initializing an array with an array of the wrong size
- Using an array variable without specifying an index
- Assigning the return value of a void function to a variable
- Returning an incompatible type on a function
- Calling a function with an incorrect number of arguments
- Using non-matching argument types when calling a function
- Expressions not evaluating to a bool inside the conditions for 'if', 'elseif', and 'while' statements

The virtual machine will print an error and exit if it finds any of the following errors:

- Reading an incompatible type from the console to assign to a variable
- Accessing an out of bounds array position
- Accessing a variable that doesn't exist in memory

Compiler Description

Computer equipment, language, and utilities used in Giga's development

Giga was made using two MacBook Pro's running macOS 10.12. The compiler and virtual machine were built with Python 2.7, and for the front-end we used Blockly.

Description of Lexical Analysis

Regular expressions of the main elements

```
t_ASSIGN = r'='
t_PLUS = r'\+'
t_MINUS = r'\-'
t_TIMES = r'\*'
t_DIVIDE = r'\/'
t_LESSTHAN = r'\<'
t_GREATERTHAN = r'\>'
t_LESSTHANEQ = r'\<='
t_GREATERTHANEQ = r'\>='
t_EQUAL = r'=='
t_DIFFERENT = r'!='
t_OR = r'\|\|'
t_AND = r'&&'
t_LEFTBKT = r'\{'
t_RIGHTBKT = r'\}'
t_LEFTSQBKT = r'\['
t_RIGHTSQBKT = r'\]'
t_LEFTPAREN = r'\('
t_RIGHTPAREN = r'\)'
t_COMMA = r'\,'
t_SEMICOLON = r'\;'
t_NUMBERINT = r'[0-9]+'
t_NUMBERFLT = r'[0-9]+\.[0-9]+'
```

Enumeration of language tokens and associated code

```
Reserved words:
'module' = 'MODULE'  'while'  = 'WHILE'         'true'   = 'TRUE'      'float'  = 'TFLOAT'
'main'   = 'MAIN'    'if'     = 'IF'            'false'  = 'FALSE'     'string' = 'TSTRING'
'func'   = 'FUNC'    'else'   = 'ELSE'          'void'   = 'VOID'
'print'  = 'PRINT'   'elseif' = 'ELSEIF'        'bool'   = 'TBOOL'
'read'   = 'READ'    'return' = 'RETURN'        'int'    = 'TINT'
```

```
Tokens:
'ASSIGN'    = '='    'GREATERTHAN'    = '>'      'AND'         = '&&'   'RIGHTPAREN'  = ')'
'PLUS'      = '+'    'LESSTHANEQ'     = '<='     'LEFTBKT'     = '{'    'COMMA'       = ','
'MINUS'     = '-'    'GREATERTHANEQ'  = '>='     'RIGHTBKT'    = '}'    'SEMICOLON'   = ' " '
'TIMES'     = '*'    'EQUAL'          = '=='     'LEFTSQBKT'   = '['
'DIVIDE'    = '/'    'DIFFERENT'      = '!='     'RIGHTSQBKT'  = ']'
'LESSTHAN'  = '<'    'OR'             = '||'     'LEFTPAREN'   = '('
```

Description of Syntax Analysis

Grammar used to represent the syntactic structures

```
Module ::= module id '{'Vars* a1 Func* Main '}'
Type ::= (bool | int | float | string) a24
Statute ::= (Call | Assignment | Vars | Condition | Read | Write | Cycle) a25
Exp ::= Term (('+' a12| '-' a12) Exp)*
Term ::= Factor (('*' a12| '/' a12) Term)*
Factor ::= ((('+' | '-')? Cte) a15 | ('(' a13 Expression ')' a14)) a15
Expression ::= Exp (( '<=' | '>=' | '<' | '>'| '==' | '!=' | '||' | '&&') a12 Exp a23)?
Cycle ::= while a16 '(' Expression ')' a17 Block a18
Call ::= id a19 '(' (Exp a20 (',' Exp a20)*)? ')' ';'
Main ::= 'main' a4 a11 Block
Block ::= '{' Statute* '}'
Read ::= 'read' '(' id ')' ';'
Write ::= 'print' '(' Exp ')' ';'
Vars ::= Type id a2 ('=' ('+' | '-')? Cte | ('[' cteN ']' '=' ConstantArray a3)) (',' id ('=' ('+' | '-')?
Cte | ('[' cteN ']' '=' ConstantArray a3)))*';'
Func ::= func a4 (void a5 | a9 Type) id a6 '(' Parameters? ')' a7 '{' Statute* (return Exp ';' a8)? '}'
a10
Parameters ::= Type id a21 (',' Type id a21)*
ConstantArray ::= '['Cte a26 (',' Cte a26)* ']'
Condition ::= 'if' '(' Expression ')' a27 Block a28 (('elseif' '(' Expression')' a29 Block a28 )* ( 'else'
Block)?) a30
Cte ::= (id |VarArr | true | false | cteN | cteS)
Assignment ::= (id | VarArr) '=' (Call | Expression) ';' a31
VarArr ::= id '[' Exp ']' a22
```

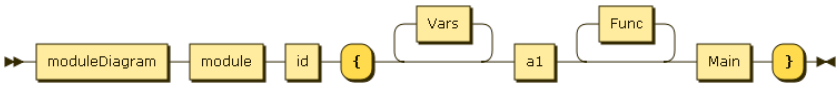
Description of Intermediate Code Generation and Semantic Analysis

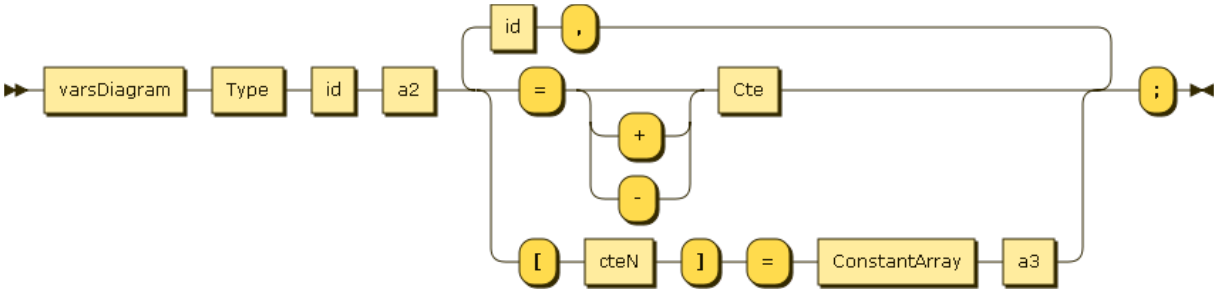
Operation code and associated virtual directions

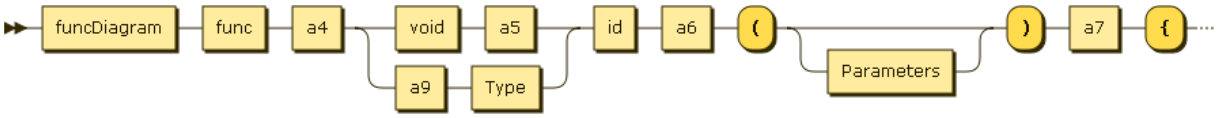
```
Values for available types:
BOOL     = 1        INT      = 2        FLOAT      = 3        STRING     = 4        VOID = 0
BOOLARRAY= 11       INTARRAY = 22       FLOATARRAY  = 33       STRINGARRAY = 44

Virtual directions for variables and constants:
```

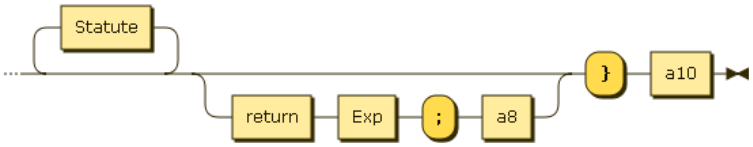
	Global variables	Local variables	Temporary	Constants
bool	10000	20000	30000	40000
int	12500	22500	32500	42500
float	15000	25000	35000	45000
string	17500	27500	37500	47500

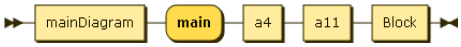
Syntax diagrams

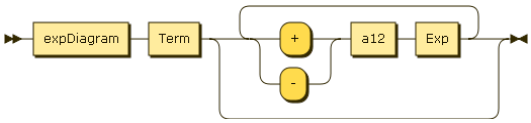


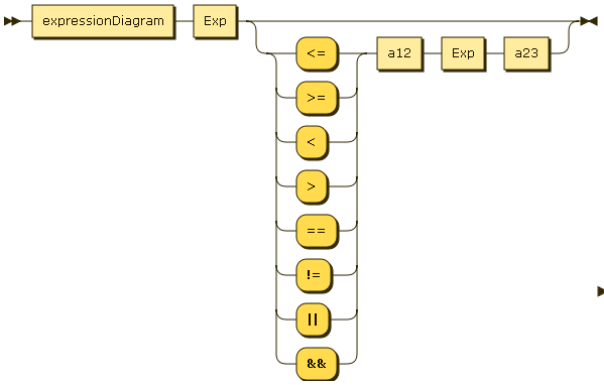


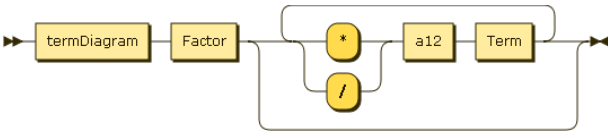


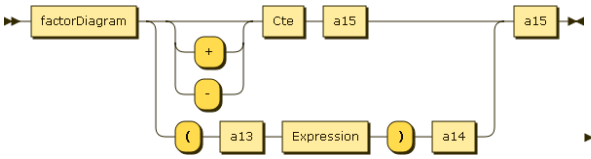


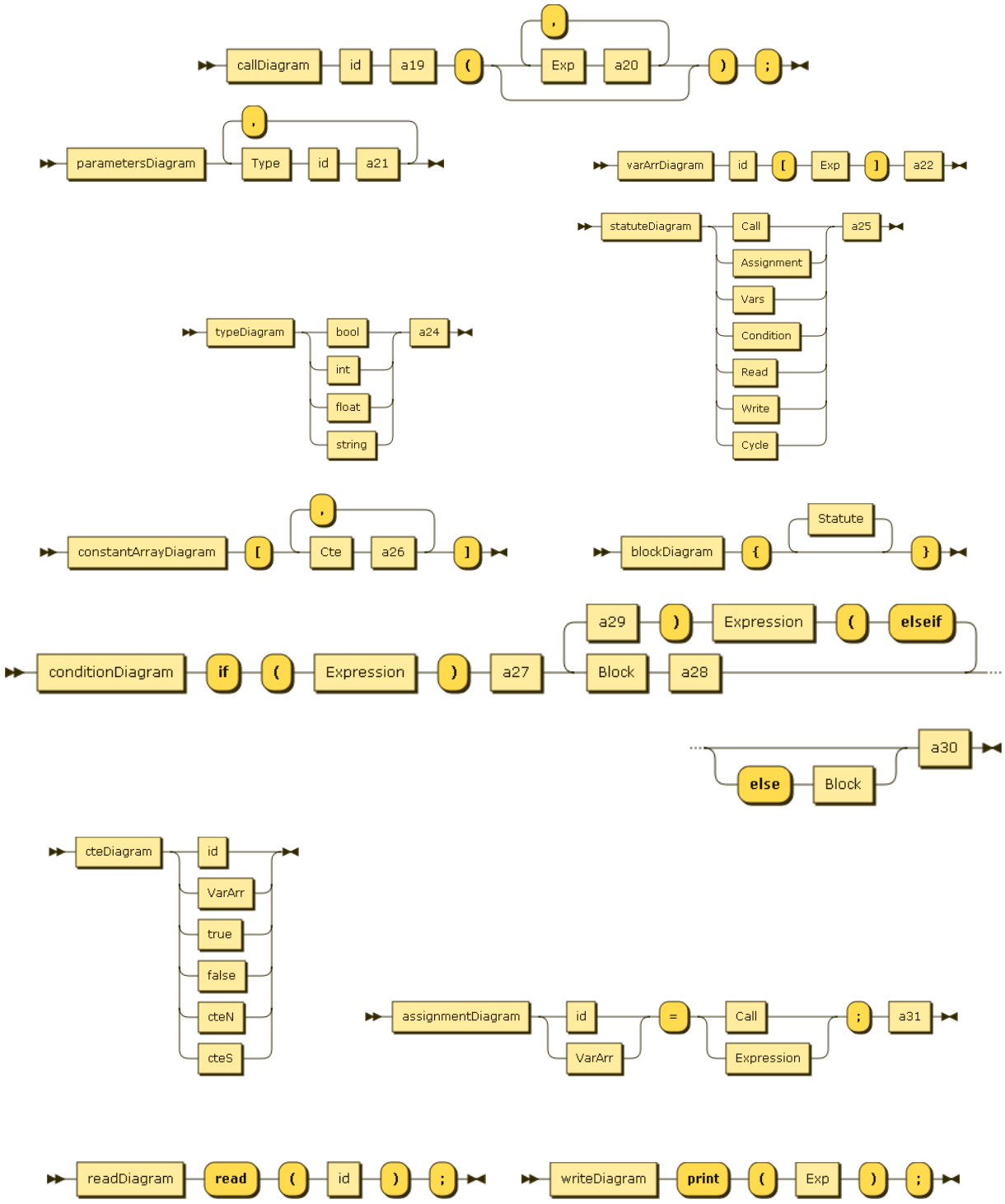












Description of each semantic action

- a1: Creates a 'GOTO' quadruple to set a jump to main
- a2: Adds the variable id to the variables dictionary
- a3: Changes the type of the variable in the dictionary to array
- a4: Changes the scope to be local
- a5: Sets the function type as void

- a6: Saves the function's id
- a7: Adds the function to the functions dictionary
- a8: Creates a 'RETURN' quadruple with the top of the operand stack
- a9: Saves the function's type
- a10: Creates an 'ENDFUNC' quadruple and saves the number of local and temporary memory spaces the function needs on the functions dictionary
- a11: Completes the 'GOTO' quadruple created in a1 with the current quadruple count
- a12: Adds the operation to the operation stack
- a13: Adds a fake bottom to the operation stack
- a14: Removes the fake bottom from the operation stack
- a15: Creates a quadruple to solve the next operation (* / &&) from the stack
- a16: Adds the current quadruple count to the jump stack
- a17: Verifies the top of the operand stack is a bool, creates a 'GOTOF' quadruple, and adds the current quadruple count to the jump stack
- a18: Creates a 'GOTO' quadruple using the second top of the jump stack and completes the 'GOTOF' quadruple on the top of the jump stack using the current quadruple count
- a19: Gets the parameters the function expects and creates a 'MEMORY' quadruple
- a20: Verifies the argument matches the parameter expected by the function, if it matches, it adds a 'PARAM' quadruple
- a21: Adds the parameter to the function in the functions dictionary
- a22: Adds 'VERIFY' quadruple and '+' quadruple to add the array's initial direction and the index
- a23: Creates a quadruple to solve the next (comparison) operation from the stack
- a24: Saves the variable's type
- a25: Discards the top of the operand stack if there's a call to a function without assigning the return value to a variable.
- a26: Saves the constant to an array to be assigned to the variable array at a later time
- a27: Adds a 'IF' separator to the jump stack and verifies the top of the operand stack is a bool, creates a 'GOTOF' quadruple, and adds the current quadruple count - 1 to the jump stack
- a28: Creates a 'GOTO' quadruple, completes the quadruple on the top of the jump stack with the current quadruple count, and adds the current quadruple count - 1 to the jump stack
- a29: Verifies the top of the operand stack is a bool, creates a 'GOTOF' quadruple, and adds the current quadruple count - 1 to the jump stack
- a30: Completes every quadruple on top of the jump stack with the current quadruple count until it finds the 'IF' marker
- a31: Verifies types are compatible and creates '=' quadruple

Semantic table

`bool = 1, int = 2, float = 3, string = 4`
`Return value shown, if empty the operands' types are incompatible using that operation`

op1	op2	=	+	-	*	/	<	>	<=	>=	==	!=	&&	||
1	1	1									1	1	1	1
1	2													
1	3													
1	4													
2	1													
2	2	2	2	2	2	2	1	1	1	1	1	1		
2	3		3	3	3	3	1	1	1	1	1	1		
2	4													

op1	op2	=	+	-	*	/	<	>	<=	>=	==	!=	&&	||
3	1													
3	2	3	3	3	3	3	1	1	1	1	1	1		
3	3	3	3	3	3	3	1	1	1	1	1	1		
3	4													
4	1													
4	2													
4	3													
4	4	4									1	1		

Description of the Process of Memory Management During Compile Time

Graphic specification of each data structure used

For memory management on the compiler we used a combination of dictionaries, lists, stacks, and counters. We used four dictionaries to keep track of the current count of variables, and constants being used. For example, the dictionary globalVarCount has the counts for each type of global variable, so globalVarCount[BOOL] will return the next memory direction available for a global bool variable.

Variable and function dictionaries are used to store the necessary information of each one, using the name of the variable or function to be able to access them. We used three dictionaries, one dictionary for local variables, one for global variables, and one for functions. Dictionaries for the variables hold the variable's name, it's type, and it's location in memory, and if the variable is an array, it will also have the array's size. Dictionaries for functions hold the function's name, it's return type, it's list of parameters, the quadruple number in which it begins, the memory location where the function will save it's return value, and a count of the number of variables and temporary memory locations that it needs to run. There's also a dictionary for constant values used throughout the program's execution, this dictionary holds the type of value, it's memory location, and it's real value. Here's an example of a variable and a function dictionary:

```
Local variables dictionary:
int a[10] = [1,2,3,4,5,6,7,8,9,0];
bool b = true;

{'a': {'type': 22, 'name': 'a', 'dir': 22500, 'size': 10},
 'b': {'type': 1, 'name': 'b', 'dir': 20000}}

Functions dictionary: [code for these functions is in page 17]
{'fibonacci':
  {'floatCount': 25000, 'intTempCount': 32502, 'floatTempCount': 35000,
   'name': 'fibonacci', 'parameters': [{'type': 2, 'name': 'n', 'dir': 22500}],
   'type': 2, 'stringCount': 27500, 'intCount': 22505, 'startQuadruple': 1,
   'stringTempCount': 37500, 'boolTempCount': 30001, 'boolCount': 20000, 'dir': 12500},
 'factorialRec':
  {'floatCount': 25000, 'intTempCount': 32502, 'floatTempCount': 35000,
   'name': 'factorialRec', 'parameters': [{'type': 2, 'name': 'n', 'dir': 22500}],
   'type': 2, 'stringCount': 27500, 'intCount': 22503, 'startQuadruple': 16,
   'stringTempCount': 37500, 'boolTempCount': 30001, 'boolCount': 20000, 'dir': 12501}}
```

For the quadruples we used a list to store all of the quadruples, and each quadruple is a dictionary containing four values: var, var2, op, and result. Quadruples for expressions are self explanatory, the

other quadruples use each of those values in different ways, here's a quick explanation of each of those, to simplify even further we'll only show the used values:

```
Quadruple example: {var1:value, op:value, var2:value, result:value}

Operations GOTO, GOTOF - Jumps to the specified quadruple
  op     = GOTO / GOTOF
  result = Quadruple number to jump to

Operation PRINT / READ - prints/reads to/from the console
  op     = PRINT / READ
  result = Location in memory to print / write to

Operation VERIFY - Verifies the index is within the array's bounds
  op     = VERIFY
  var1   =  Index to verify
  result = Array's size

Operation MEMORY - Creates new memory allocation for function call
  op     = MEMORY
  result = Function's dictionary

Operation GOFUNC - Jumps to the start of the function
  op     = GOFUNC
  result = Quadruple number to jump to

Operation PARAM - Copies the value of an argument to a function's parameter
  op     = PARAM
  var1   = Memory direction of the argument sent
  result = Memory direction of the parameter

Operation RETURN - Returns a value from a function call
  op     = RETURN
  result = Memory direction of the value to return

Operation ENDFUNC - Marks the end of a function
  op     = ENDFUNC

Operation END - Marks end of program
  op     = END
```

Lastly, we used four stacks, an operand stack, an operation stack, a jump stack, and a stack of constants. The operand and operation stacks are used to be able to create the necessary quadruples to solve expressions, the jump stack is used to generate and complete GOTO and GOTOF quadruples, and the stack of constants is used to initialize an array variable.

Description of Virtual Machine

Description of the Process of Memory Management During Run Time

The virtual machine was developed using two components: the first component analyzes the quadruples, this component is called Machine, and the second component handles the variables in real memory, this is called Memory. Machine handles all of the operations that the quadruples have, this continues until the quadruples reach the 'END'. The Memory handles all of the variables that are gonna be used and stores them in real memory that has been indexed by the functions inside the Memory file.

The first step that the user has to do in order to call the virtual machine is initialize a method called 'executeVirtualMachine' that is in the Machine file in the main method of the Giga file. This method

will receive the functions, the quadruples, constants, global variable count, temporal variable count, local variable count and pointer variable count. This function will initialize the only method in the Machine that will iterate through the quadruples. Until the quadruple 'END' is executed the Machine will run the intended operation that the user wants to do. There is a counter that is used to iterate and this is increased or decreased depending of what operation wants to be done.

There is also a definition of a class object that is called active memory, this object can reference the real memory an will bring the variables used in the program, it will kill the variables when a function is finished and will wake/sleep the memory when a function is called/finished. This object will handle all of the operations that have something to do with the memory, there is multiple methods in this class .

Regarding the Memory class there's multiple lists of variables that are defined within the initialization of the memory object. By the size of the global variables, temp variables, pointer variables and local variables you define the the amount of spaces that you're gonna need in the list of real memory variables. In case that there is a function call, the Machine executes the memoryFunc which will generate new lists of data and then when the functions asks real memory it will put that data into the real lists and store the original or the memory that was previously running. The description on how to access the real memory items and how to change the value of a variable will be described in the next sections.

Graphic specification of each data structure used

In the Machine file:

quadruples - The quadruples dictionary is a dictionary of dictionaries of all of the quadruples that are stored in the lexical and syntactical analysis. This dictionary is iterated through a while loop and will end when the while function finds the 'END' quadruple.

quadruple - In each iteration of the while loop to facilitate the iteration of the dictionary we create a single dictionary that we extract from the quadruples dictionary of dictionaries. This will help to iterate the one or two variables that we want to modify, read, or print as a virtual address then with the methods defined in the Memory file you will get the value.

quadrupleStack - The quadruple stack data structure is used to store all of the quadruples which you will have to return to when your function ends. This will store all of the positions an will pop that position when 'END FUNC' quadruple is called.

In the Memory file:

localVariableTypes/globalVariableTypes/tempBools and all of the newLocal, newTemp variables - All of this variable types are lists of data that are allocated into real memory with initial values of None, they will allocate the values of the variables when they're used within the quadruples. Every time there is a reference to a certain space of real memory. What the memory does is to find the offset of the space of memory to which it has place this data type.

globalInts example

None	None	None	None

This is the initial view of a list of 3 variables that will be allocated in the memory, but the space is not used still by the virtual variables. When a quadruple references a type of variable that is not constant, it will find the scope, type of variable, and offset of that virtual memory location. We add a +1 in order to prevent crashing from the real addresses that are sometimes referenced. After determining

all of those aspects it will find the respective offset in which it should place the variable for it to be referenced in the future.

globalInts example

1	None	None	None

Lets say that we stored the variable x with a value of 1 in the globalInts list. Every time we will want to access this memory we will have to determine which type, scope and offset the variable has, this is done to ensure that we don't have any incorrect variable referenced.

memoryStack - The memory stack will store the values of the variables that will have to sleep when a function is called. This will store the local and tempValues that are gonna be used in the future. When the wakeMemory function is called memory will be relocated into the next point in which it should continue the quadruples.

oldMemory - Is a dictionary of the variables that will have to sleep while a function or another section of code is being executed in the quadruple. This dictionary is stored in the memoryStack and will be popped when the function finishes execution.

pointer - The pointer list will contain a reference to a real memory inside the pointer list. When ever a value of an array wants to be accessed you need to recursively call the function to find the value of the address inside the pointer. And if you want to store it, the variable has already stored in the initialization of the array. But if you want to change it, you will have to change the value of the pointer variable that has been referenced to.

pointer example

12500	None	None	None

This is an example of the pointer list, it will store the value of the variable that it has referenced has inside of it, it is an address of a globalInt variable that is store in the globalInts list. If the user wants to access that value it will call the getValueAtAddress of the pointer address then internally the Memory file will reference that value of the pointer. And this will return the original value.

Association between virtual and real memory addresses

As explained in the previous point the virtual memory and the real memory have a relation given by the quadruples. The quadruples have the virtual address of the variables to which will have a reference in the lists of the real memory. When the variables are already allocated in real memory the virtual address is used as a combination of the count of the initial value of that specific type of variable to find the offset in which the variable is located.

For example the address 12500 of the quadruple has the real value of 10 referenced in the constants when stored this value in the position 0 of the list of globalInts it will now reference this position.

Language Functionality Tests

Checking Giga's functionality

Code for the test

The following Giga code tests all of the available functionality.

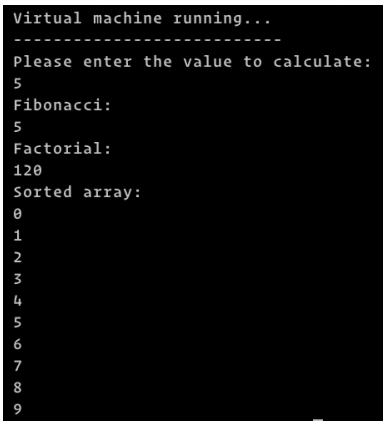
```
module gigaTest {
  func int fibonacci(int n) {
    int x = 0, y = 1, z = 1, i = 0;
    while (i < n) {
      x = y;
      y = z;
      z = x + y;
      i = i + 1;
    }
    return x;
  }

  func int factorialRec(int n) {
    int result = 1, temp = 1;
    if (n > 0) {
      temp = factorialRec(n - 1);
      result = n * temp;
    }
    return result;
  }

  main {
    int a[10] = [2, 3, 5, 9, 1, 4, 0, 6, 8, 7];
    int x = 1, i = 1, j = 0, temp = 0, size = 10, n = 0;
    print("Please enter the value to calculate:");
    read(n);
    print("Fibonacci:");
    x = fibonacci(n);
    print(x);
    print("Factorial:");
    x = factorialRec(n);
    print(x);
    while (i < size) {
      while (size - 1 > j) {
        if (a[j] > a[j+1]) {
          temp = a[j];
          a[j] = a[j+1];
          a[j+1] = temp;
        }
        j = j + 1;
      }
      i = i + 1;
      j = 0;
    }
    x = 0;
    print("Sorted array:");
    while (x < 10) {
      print(a[x]);
      x = x + 1;
    }
  }
}
```

Test results generated from code generation and execution

```
These are the quadruples generated by the compiler:
0    {var1: }      {op:GOTO }    {var2: }      {result:30 }
1    {var1:42501 } {op:= }       {var2: }      {result:22501 }
2    {var1:42502 } {op:= }       {var2: }      {result:22502 }
3    {var1:42502 } {op:= }       {var2: }      {result:22503 }
4    {var1:42501 } {op:= }       {var2: }      {result:22504 }
5    {var1:22504 } {op:< }       {var2:22500 } {result:30000 }
6    {var1:30000 } {op:GOTOF }   {var2: }      {result:14 }
7    {var1:22502 } {op:= }       {var2: }      {result:22501 }
8    {var1:22503 } {op:= }       {var2: }      {result:22502 }
9    {var1:22501 } {op:+ }       {var2:22502 } {result:32500 }
10   {var1:32500 } {op:= }       {var2: }      {result:22503 }
11   {var1:22504 } {op:+ }       {var2:42502 } {result:32501 }
12   {var1:32501 } {op:= }       {var2: }      {result:22504 }
13   {var1: }      {op:GOTO }    {var2: }      {result:5 }
14   {var1: }      {op:RETURN }  {var2: }      {result:22501 }
15   {var1: }      {op:ENDFUNC } {var2: }      {result: }
16   {var1:42502 } {op:= }       {var2: }      {result:22501 }
17   {var1:42502 } {op:= }       {var2: }      {result:22502 }
18   {var1:22500 } {op:> }       {var2:42501 } {result:30000 }
19   {var1:30000 } {op:GOTOF }   {var2: }      {result:28 }
20   {var1: }      {op:MEMORY }  {var2: }      {result:{'floatCount': 25000, 'intTempCount': 32502,
'floatTempCount': 35000, 'name': 'factorialRec', 'parameters': [{'type': 2, 'name': 'n', 'dir': 22500}],
'type': 2, 'stringCount': 27500, 'intCount': 22503, 'startQuadruple': 16, 'stringTempCount': 37500,
'boolTempCount': 30001, 'boolCount': 20000, 'dir': 12501} }
21   {var1:22500 } {op:- }       {var2:42502 } {result:32500 }
22   {var1:32500 } {op:PARAM }   {var2: }      {result:22500 }
23   {var1: }      {op:GOFUNC }  {var2: }      {result:16 }
24   {var1:12501 } {op:= }       {var2: }      {result:22502 }
25   {var1:22500 } {op:* }       {var2:22502 } {result:32501 }
26   {var1:32501 } {op:= }       {var2: }      {result:22501 }
27   {var1: }      {op:GOTO }    {var2: }      {result:28 }
28   {var1: }      {op:RETURN }  {var2: }      {result:22501 }
29   {var1: }      {op:ENDFUNC } {var2: }      {result: }
30   {var1:42504 } {op:= }       {var2: }      {result:22500 }
31   {var1:42505 } {op:= }       {var2: }      {result:22501 }
32   {var1:42506 } {op:= }       {var2: }      {result:22502 }
33   {var1:42507 } {op:= }       {var2: }      {result:22503 }
34   {var1:42502 } {op:= }       {var2: }      {result:22504 }
35   {var1:42508 } {op:= }       {var2: }      {result:22505 }
36   {var1:42501 } {op:= }       {var2: }      {result:22506 }
37   {var1:42509 } {op:= }       {var2: }      {result:22507 }
38   {var1:42510 } {op:= }       {var2: }      {result:22508 }
39   {var1:42511 } {op:= }       {var2: }      {result:22509 }
40   {var1:42502 } {op:= }       {var2: }      {result:22510 }
41   {var1:42502 } {op:= }       {var2: }      {result:22511 }
42   {var1:42501 } {op:= }       {var2: }      {result:22512 }
43   {var1:42501 } {op:= }       {var2: }      {result:22513 }
44   {var1:42503 } {op:= }       {var2: }      {result:22514 }
45   {var1:42501 } {op:= }       {var2: }      {result:22515 }
46   {var1: }      {op:PRINT }   {var2: }      {result:47500 }
47   {var1: }      {op:READ }    {var2: }      {result:22515 }
48   {var1: }      {op:PRINT }   {var2: }      {result:47501 }
49   {var1: }      {op:MEMORY }  {var2: }      {result:{'floatCount': 25000, 'intTempCount': 32502,
'floatTempCount': 35000, 'name': 'fibonacci', 'parameters': [{'type': 2, 'name': 'n', 'dir': 22500}],
'type': 2, 'stringCount': 27500, 'intCount': 22505, 'startQuadruple': 1, 'stringTempCount': 37500,
'boolTempCount': 30001, 'boolCount': 20000, 'dir': 12500} }
50   {var1:22515 } {op:PARAM }   {var2: }      {result:22500 }
51     {var1: }    {op:GOFUNC }  {var2: }      {result:1 }
52   {var1:12500 } {op:= }       {var2: }      {result:22510 }
```

```
53  {var1: }      {op:PRINT }   {var2: }      {result:22510 }
54  {var1: }      {op:PRINT }   {var2: }      {result:47502 }
55  {var1: }      {op:MEMORY }  {var2: }      {result:{'floatCount': 25000, 'intTempCount': 32502,
'floatTempCount': 35000, 'name': 'factorialRec', 'parameters': [{'type': 2, 'name': 'n', 'dir': 22500}],
'type': 2, 'stringCount': 27500, 'intCount': 22503, 'startQuadruple': 16, 'stringTempCount': 37500,
'boolTempCount': 30001, 'boolCount': 20000, 'dir': 12501} }
56  {var1:22515 } {op:PARAM }   {var2: }      {result:22500 }
57  {var1: }      {op:GOFUNC }  {var2: }      {result:16 }
58  {var1:12501 } {op:= }       {var2: }      {result:22510 }
59  {var1: }      {op:PRINT }   {var2: }      {result:22510 }
60  {var1:22511 } {op:< }       {var2:22514 } {result:30000 }
61  {var1:30000 } {op:GOTOF }   {var2: }      {result:93 }
62  {var1:22514 } {op:- }       {var2:42502 } {result:32500 }
63  {var1:32500 } {op:> }       {var2:22512 } {result:30001 }
64  {var1:30001 } {op:GOTOF }   {var2: }      {result:89 }
65  {var1:22512 } {op:VERIFY }  {var2: }      {result:10 }
66  {var1:22500 } {op:+ }       {var2:22512 } {result:50000 }
67  {var1:22512 } {op:+ }       {var2:42502 } {result:32501 }
68  {var1:32501 } {op:VERIFY }  {var2: }      {result:10 }
69  {var1:22500 } {op:+ }       {var2:32501 } {result:50001 }
70  {var1:50000 } {op:> }       {var2:50001 } {result:30002 }
71  {var1:30002 } {op:GOTOF }   {var2: }      {result:86 }
72  {var1:22512 } {op:VERIFY }  {var2: }      {result:10 }
73  {var1:22500 } {op:+ }       {var2:22512 } {result:50002 }
74  {var1:50002 } {op:= }       {var2: }      {result:22513 }
75  {var1:22512 } {op:VERIFY }    {var2: }    {result:10 }
76  {var1:22500 } {op:+ }       {var2:22512 } {result:50003 }
77  {var1:22512 } {op:+ }       {var2:42502 } {result:32502 }
78  {var1:32502 } {op:VERIFY }  {var2: }      {result:10 }
79  {var1:22500 } {op:+ }       {var2:32502 } {result:50004 }
80  {var1:50004 } {op:= }       {var2: }      {result:50003 }
81  {var1:22512 } {op:+ }       {var2:42502 } {result:32503 }
82  {var1:32503 } {op:VERIFY }  {var2: }      {result:10 }
83  {var1:22500 } {op:+ }       {var2:32503 } {result:50005 }
84  {var1:22513 } {op:= }       {var2: }      {result:50005 }
85  {var1: }      {op:GOTO }    {var2: }      {result:86 }
86  {var1:22512 } {op:+ }       {var2:42502 } {result:32504 }
87  {var1:32504 } {op:= }       {var2: }      {result:22512 }
88  {var1: }      {op:GOTO }    {var2: }      {result:62 }
89  {var1:22511 } {op:+ }       {var2:42502 } {result:32505 }
90  {var1:32505 } {op:= }       {var2: }      {result:22511 }
91  {var1:42501 } {op:= }       {var2: }      {result:22512 }
92  {var1: }      {op:GOTO }    {var2: }      {result:60 }
93  {var1:42501 } {op:= }       {var2: }      {result:22510 }
94  {var1: }      {op:PRINT }   {var2: }      {result:47503 }
95  {var1:22510 } {op:< }       {var2:42503 } {result:30003 }
96  {var1:30003 } {op:GOTOF }   {var2: }      {result:103 }
97  {var1:22510 } {op:VERIFY }  {var2: }      {result:10 }
98  {var1:22500 } {op:+ }       {var2:22510 } {result:50006 }
99  {var1: }      {op:PRINT }   {var2: }      {result:50006 }
100 {var1:22510 } {op:+ }       {var2:42502 } {result:32506 }
101 {var1:32506 } {op:= }       {var2: }      {result:22510 }
102 {var1: }      {op:GOTO }    {var2: }      {result:95 }
103 {var1: }      {op:END }     {var2: }      {result: }
```

These are the results shown in the console after entering a value of 5 when prompted:



Documented Listings of the Project

Description of each module, it's parameters, and it's return value

Some important modules in compile time:

Function	Description
`def p_moduleg(p):`	Initial grammar rule recognizer
`def p_vars(p):`	Variable declaration grammar rule, saves the variable if the identifier is available
`def p_funcg(p):`	Function declaration grammar rule, saves the function if the identifier is available
`def p_expression(p):`	Expressions grammar rule, creates the appropriate quadruples to solve the expression
`def p_factor(p):`	Factor grammar rule, creates the appropriate quadruples to solve the factor
`def p_call(p):`	Verifies the function called exists and the call has the correct arguments, adds the quadruples needed to call the function
`def p_constantArray(p):`	Adds the constants in the array to the constants stack to then initialize the array variable
`def p_cteS(p):`	Identifies and saves string constants
`def p_assignement(p):`	Generates the quadruples to assign a value to a variable
`def p_varArr(p):`	Grammar rule for getting a value out of an array
`def verifyVar(var):`	Verifies the variable you're trying to use has been declared
`def p_addConstant(p):`	Adds a constant to the constants dictionary
`def p_addFakeBottom(p):`	Adds a fake bottom to the operation stack
`def p_removeFakeBottom(p):`	Removes the fake bottom from the operation stack

Function	Description
`def p_ifStart(p):`	Marks the start of an if statement by adding a marker to the jump stack
`def p_ifEnd(p):`	Completes all GOTO quadruples of the if statement by looking for the marker on the jump stack
`def p_whileStart(p):`	Saves current quadruple count on the jump stack
`def p_whileCheck(p):`	Adds the GOTOF quadruple to exit the while loop
`def p_whileEnd(p):`	Completes all GOTOF quadruple of the while loop and adds a GOTO quadruple to return to the beginning of the loop
`def p_jumpToMain(p):`	Adds an empty GOTO quadruple to jump to the main function
`def p_completeJumpToMain(p):`	Completes the GOTO quadruple with the quadruple number at the start of the main function
`def p_funcReturn(p):`	Adds the RETURN quadruple of the function if the return value matches the definition of the function
`def p_funcEnd(p):`	Saves the number of variables used on the function's dictionary and adds the ENDFUNC quadruple
`def addVariable(variable, varType):`	Adds the variable to the variables dictionary
`def addFunction(name, funType, startQuadruple):`	Adds the function to the functions dictionary
`def addQuadruple(operation, var1, var2, result):`	Creates a quadruple with the specified values
`def completeQuadruple(index, newValue):`	Completes the quadruple at index by adding newValue as the result value
`def resetLocalCounters():`	Resets the counters for variables and temporary values used

Some important modules in run time:

Function	Description
`def executeVirtualMachine(functions, quadruples, constants, globalVarCount, tempVarCount, localVarCount, pointerCount):`	Starts the execution of the virtual machine
`def __init__(self, name, localVariables, globalVariables, tempVariables, pointerVariables):`	Creates the necessary lists the virtual machine will use as its memory
`def memoryFunc(self, result):`	Creates new memory lists for a function call
`def sleepMemory(self):`	Adds the current active memory to the stack of memories to change context
`def wakeMemory(self):`	After a function is over, this will wake up the previous memory to return to the previous context
`def getNewArrayOfType(self, variableType):`	Returns the memory list for the specified scope and type from the newly created function memory

Function	Description
`def getOffset(self, address, scope, varType):`	Returns the real index position on the memory list given a virtual address, scope, and type
`def storeParam(self, address, value):`	Using the current memory and the newly created memory for a function it moves the values of the arguments from the previous context into the function's context