Tecnológico de Monterrey

Choo Choo Train

David Benitez - A01191731
Paulina Escalante - A01191962

David Benitez

May 3, 2017

Table of Contents

Project Description

Vision

The purpose of this project is to help people get an insight into the world of programming and how code compilation and execution works. People are usually uninterested to learn how to program because they are not sure of what it actually is or what it's useful for. They think that because they don't know about computers, they will be bad at programming. The fact that programs look like huge files of unreadable text, sometimes with complicated formulas, doesn't help either. If people can relate programming to something that is familiar to them and visually pleasing, then we might be able to help them understand what programming really is about.

People will use our language to "program" blocks. Then they will be able to watch their code being executed. By mapping the elements of a language (such as loops, statements, etc.) to a specific type of train track section or element, we can help kids understand the logic and data flow of a program. This will introduce them to the world of programming while still maintaining an attractive and fun appearance that won't make them think that what they're about to do is rocket science (yet).

Objectives

The main objective of our language is to be understandable, visual, and easy to use so that people will be motivated to learn how to program. The main characteristic is that both the input and the output are visual, in a sense that the output encompasses the step by step compilation and execution of the program. With this visual representation, complicated programming terminology and structures are abstracted and represented with charts that contain relevant data. The objective is to be an educational, fun, and pleasing programming language for everyone.

Project Scope

Choo Choo Train is a simple, yet powerful compiler. It can handle basic operations, assignments, declarations, lists, four data types, recursion, conditional statements, loops statements, and calls to blocks, amongst other elements. However, it does not support matrices, cubes or more complicated data structures. it is also not an object-oriented language. It mainly consists of a chain of calls to blocks, starting with a main block.

Choo Choo Train receives input and output as text only. Files can be used to compile but must be text files. It does not require system configurations as it is a web compiler.

Requirements

Functional Requirements

- The site must allow the user to give it a text file with code or write directly the code in the site in order to compile and execute it.
- The compiler must show appropriate compilation error messages to the user.
- The virtual machine must show appropriate run time error messages to the user.
- The site must show the user different graphs that detail his/her code's execution time, number of variables, number of loops, number of executed operations, etc.
- The site should be able to send input into the program.
- The site should show the output from the execution to the user.

Non-Functional Requirements

- The site should show appropriate messages to the user in no more than one second after the compiler/virtual machine decides it should show it.
- The site should alert the user about any changes in compilation/run time status.
- The site should notify the user that some charts can't be shown since no data was collected for them.
- The site must be able to run in any browser.
- The site must be able to run in computers and tablets.
- The site must show the user a user manual so he/she can learn how to program in the Choo Choo Train language.

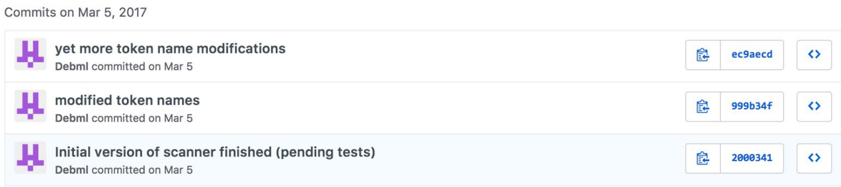
Test Cases

The following test cases were created for the language. They can be seen with more detail in the testing section further below.

- Block declaration
- Simple print statement
- Variable declaration
- Variable assignment
- Operations
- Expressions
- Loop statements
- Conditional statements
- Compilation error
- Runtime error
- Input
- List manipulation
- Block calls
- Recursion

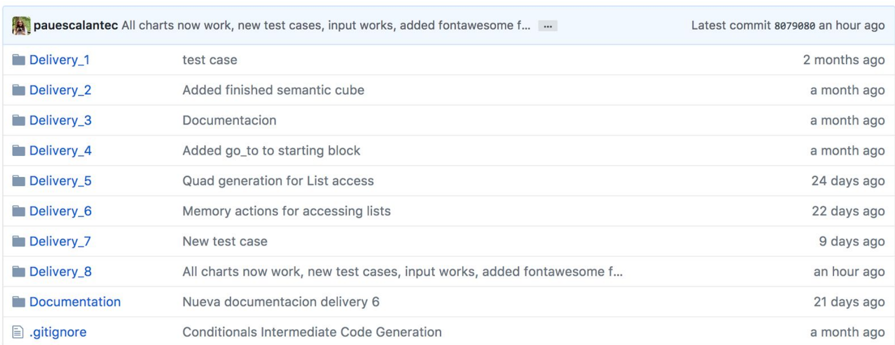
Project Development Process

The project was developed in a total of 9 weeks. The date of the first commit was done on March 5th 2017 with the first works on the lexical analysis. Version control was managed using Github. The repository can be found here: https://github.com/Debml/ChooChooTrain



A few modifications were made to the first proposal regarding our target audience and objective. It is still a visual language and easy to use but the target is now not only children but everyone new to programming. Input is still with blocks but using our own interface and not a plug-in.

Progress reports were completed weekly with a meeting every Sunday of the week. Project deliverables were made weekly and submitted on the course's Blackboard page, as well as kept in Github for evidence.



Progress Reports

Date	**Progress report**
February 27 - March 6	By this point the scanner was already generated with all of the necessary tokens, as well as the Parser with the language grammar. No ambiguities were found in the grammar, and the language was tested with a few test cases, although these were not very complex, so further testing was needed.
March 7 - March 12	In order to support intermediate actions in the grammar, empty rules were added to it, which in generated an ambiguity in the grammar that was not removed yet. The Function Directory structure was added, and we added the appropriate actions to add the variables, parameters and lists of our language into it, although there was a bug while adding lists that we decided to keep pending until the lists delivery was due.

March 13 - March 19	The Semantic Cube was done considering our data types and the possible operations we included. By this point, semantic actions to identify variables that are not declared or redeclared were added. Quadruples were also generated for arithmetic expressions were done, but logical expressions and read/write operations were left pending for the next delivery. Our quadruples were not holding real memory addresses yet, so instead we filled them with the variable names until we had an appropriate structure for memory.
March 20 - March 27	This week, quadruples were generated for conditionals (ifs) and loops, as well as the pending quadruples for logical operations and the read/write operations. Other functionality was also added, such as verbose error messages to correctly identify the compilation error, the functionality to read test cases from text files. A constant table structure was added, although its use is still pending for the memory delivery. Finally, code was refactored to reduce coupling and make it easier to modify and test in the future.
March 28 - April 3	Quadruples were generated for functions that do not return a value, although the semantic checking was done to verify that there is a match between the block signature and the returned value. Quadruples were also generated for the end of a block, as well as the first Go-To of the program that goes to the starting block. Some changes were made to the Variable table as well as the Quadruple structure. Functions that return values are pending for next week.
April 4 - April 10	Leftover work from last week was completed, mainly Quadruples for functions that return a value. Again, parts of the code were refactored to make it more maintainable and open for future changes, as well as finished commenting the code up until now. Quadruples for lists were done this week, and with that all necessary quadruple generation were finished. The structure for the memory was mostly finished, and quadruples now hold memory addresses (or pointers, for lists) instead of variable names.
April 11 - April 24	In these two weeks, the virtual machine was finished for all kinds of operations, including arithmetic and logical expressions, list access, printing, reading from console and function calls (all of our possible operations). The memory structure was finished, and the virtual machine now wrote into it. Memory limits were defined for variables and the stack segment, so by this point we can fully execute programs written in our language. The only things pending were verbose error message for execution, as well as the GUI for the program.
April 25 - April 29	This week, we added the error messages to the virtual machine, and the interface for the compiler/virtual machine was started. We branched the project on have two versions, one that would run on the command line and another that would run on the web. The code could now run on the site, although there was still some error handling pending, as well as the functionality to send input to the virtual machine from the site. Some compilation/execution data was stored in a new structure, and was now presented in the site as graphs.
April 30 - May 3	In these last few days, there was a big effort to get the final version of the GUI up and running. All of the compilation and execution errors handled in the command line interface are now shown in the site and presented to the user in a friendly way. We also finished the connection of the input from the site to the Virtual Machine, which greatly increased the amount of programs we could run in the site. New graphs were generated with run time data, and some minor formatting to the previous graphs was done. With this new functionality and data in place, the project was finally finished, and we are now ready to present a professional looking and working project.

Personal Conclusions*

The following conclusions were drawn after the project completion.

Team Member Name	**Personal Conclusion**
Paulina Escalante	"The takeaway from the project for me was being able to create something so big and complex from scratch. The way we incorporated our front-end with our back-end compiler and were able to modulate the code so easily in the end was incredibly rewarding. I felt I actually used all of my previous acquired knowledge and was able to apply it to this project. I am so proud of this and of being able to work seamlessly and efficiently with my partner. It was quite challenging for me to use a source control tool like Github so much and I learnt a lot from the experience. I feel I developed the professional skills needed to work on large scale products and systems. "
David Benitez David Benitez	"One of the biggest things I take from the experience of designing a compiler is that we are not always sure of how many operations our code actually has, and usually think only in arithmetic or logical operations. There are many more types of operations that should be considered when trying to design an efficient solution to a problem, and thinking the whole semester in terms of quads and how many each statement produces made me realize that there is much more going on behind the scenes than we think. This was also a great opportunity to test our skills in making professional, readable and modifiable code, and I think the way we structured our different classes proved that we are ready to become fully fledged Software Engineers in a world that is in need of them."

Language Description

Choo Choo Train is a friendly program that helps visualize code and run-through of the execution. It is simple and colorful to help users program, it is a block program.

Language Name

Choo Choo Train.

Language Characteristics

Choo Choo Train has characteristics that make it a simple and fun language to use:

- Structure is made up of blocks entirely
- User Interface is generated for both input and output of compiler.
- Coding can be done by text blocks or uploading a text file
- Program output is shown as text
- Program analytics and data about compilation and runtime is displayed with charts
- Relevant data regarding the program coded can be visually analyzed.
- When user is waiting for compilation or runtime, a random patience quote is generated to bring peace to user.
- Recompilation of same code is easily made

There are 4 primitive data types in our language. We wanted to make it simple for people to understand, so while we kept the types:

- integer
- float
- string
- boolean

We decided to change their names to: whole, decimal and words, respectively; excluding the boolean type. These are more commonly used terms in the english language, so it will help people understand the values being represented by each type. The boolean type remains untouched, and is named the same, for there is no term that would semantically represent the type boolean better than its own name. To make it easier for people, values will be restricted when declaring boolean variables only to True and False.

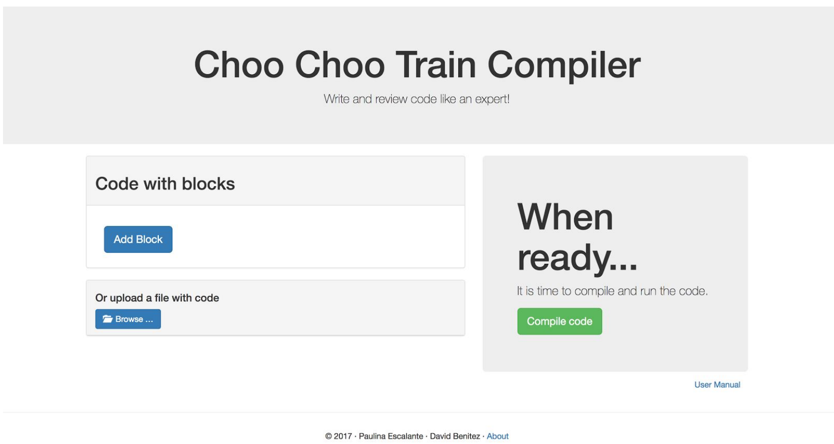
Lists are also part of the language, and can hold values for any of the available data types mentioned above. They are declared by following the same format as a variable declaration, but square brackets enclosing the size of the list follow the id name.

There are some functions that work similar to those used in C++ or Python. The logic is the same but keywords are changed to facilitate the reading of code. For example:

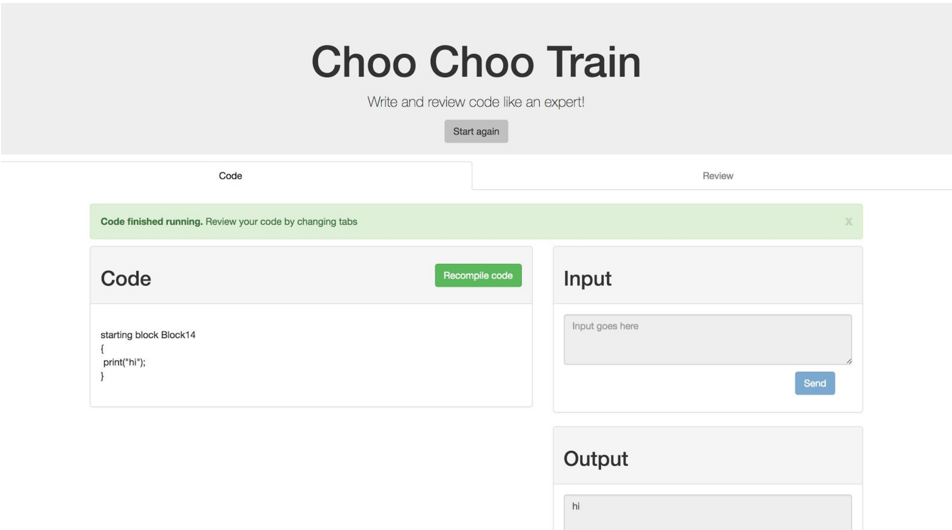
Conventional Name	**Choo Choo Train Name**
Function/Method	block
var	variable
do while	do until
&&	and
||	or
!	not

There are also three modifications for the language. Parameters in a block definition are defined with the keyword “receives” so that it is known that parameters are variables that the block receives. There is a keyword for “of type” that specifies any variable’s type, instead of the conventional way of defining variables as first specifying type and then the variable id. While loops are called “do until” loops, since this improves the clarity and readability of the code for people.

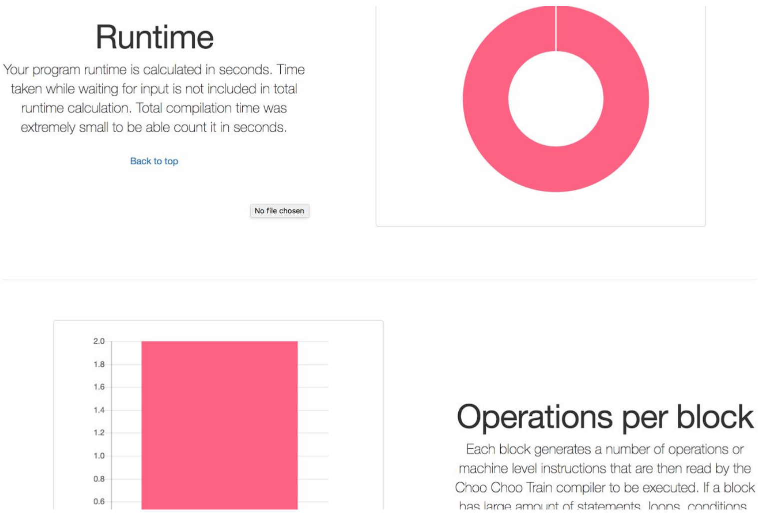
To use the compiler the user first visits the homepage of Choo Choo Train and sees the following.



The user can now upload a file or add a block. If the user is not familiar with the language, a manual is provided. The user now can click on compile code and the following prompts appear.



The user can now create inputs and see outputs while the code is running. Afterwards, an alert appears to review code on the next tab. The user can hover on the tab and review charts will appear. The charts are interactive and can show information in labels regarding the execution of the program as follows.



Errors

Errors are presented the moment they are detected and completely stop execution. Choo Choo Train can manage both compilation and runtime errors separately and handles them accordingly with a message error.

Compilation Errors

- Argument - Parameter count mismatch
- Argument - Parameter type mismatch
- Block name already defined
- Block not declared

- Block should return a value
- Invalid list index
- Invalid list size
- Invalid operator for operand(s)
- Invalid return value
- List name already defined
- List not declared
- Local memory full for data type
- Multiple starting blocks found
- No starting block found
- Number of quads exceeded the memory limit
- Parameter name already defined
- Starting block should not receive parameters
- Temporary memory full for data type
- Unexpected expression type
- Unexpected token found
- Unassigned return value
- Variable name already defined
- Variable not declared

Runtime Errors

- Block did not return a value
- Division by 0
- Index out of bounds
- Invalid input type
- Stack overflow
- Unassigned variable
- Unsupported/Unknown Operation

Compiler Description

Tools

The compiler was developed using python 2.7 for mac OS Sierra. Additionally, PLY (Python Lex-Yacc) version 3.9 was used to generate the lexer and parser based on our defined tokens and grammar, which will be outlined in the next sections.

Lexical Analysis

The lexical analysis of the compiler was done using PLY, specifically with similar functionality as Lex. The following work was completed.

Tokens

#	Token Name	Regular Expression
1	block	“block”
2	starting	“starting”
3	receives	“receives”
4	block_returns	“returns”
5	return_statement	“return”
6	call	“call”
7	variable	“variable”
8	list	“list”
9	of	“of”
10	type	“type”
11	do	“do”
12	until	“until”
13	if	“if”
14	else	“else”
15	colon	“:”
16	semicolon	“;”
17	comma	“,”
18	input	“input”
19	print	“print”

20	curlybraces_open	“{“
21	curlybraces_close	“}”
22	parenthesis_open	“(“
23	parenthesis_close	“)”
24	squarebracket_open	“[“
25	squarebracket_close	“]”
26	op_assign	“=”
27	op_less	“<”
28	op_less_equal	“<=”
29	op_greater	“>”
30	op_greater_equal	“>=”
31	op_equal	“==”
32	op_not_equal	“!=”
33	op_and	“and”
34	op_or	“or”
35	op_negation	“not”
36	op_addition	“+”
37	op_subtraction	“-”
38	op_multiplication	“*”
39	op_division	“/”
40	whole	“whole”
41	decimal	“decimal”
42	words	“words”
43	boolean	“boolean”
44	cst_whole	[0-9]+
45	cst_decimal	[0-9]+\.[0-9]+([Ee][\+-]?[0-9]+)?
46	cst_words	\”[^”]\”
47	cst_boolean	“True” | “False”
48	id	[A-Za-z]([A-Za-z0-9])*

Syntactic Analysis

Context Free Grammar is used with LR Parsing to comply with PLY standards. The following rules were created.

Syntactic Rule	Context Free Grammar (LR Parsing)
PROGRAM	**PROGRAM_AUX**
PROGRAM_AUX	**BLOCK** | **PROGRAM_AUX BLOCK**
BLOCK	**BLOCK_AUX** block id **RECEIVES_AUX RETURNS_AUX BLOCK_BODY**
BLOCK_AUX	starting | ε
RECEIVES_AUX	receives colon id of type **TYPE RECEIVES_AUX1** | ε
RECEIVES_AUX1	comma id of type **TYPE RECEIVES_AUX1** | ε
RETURNS_AUX	block_returns **TYPE** | ε
BLOCK_BODY	curlybraces_open **BLOCK_BODY_AUX** curlybraces_close
BLOCK_BODY_AUX	**DECLARATIONS BLOCK_BODY_AUX** | **BLOCK_BODY_AUX1**
BLOCK_BODY_AUX1	**STATEMENT BLOCK_BODY_AUX1** | ε
TYPE	whole | decimal | words | boolean
BODY	curlybraces_open **BODY_AUX** curlybraces_close
BODY_AUX	**STATEMENT BODY_AUX** | ε
CONDITION	if parenthesis_open **EXPRESSION** parenthesis_close **BODY CONDITION_AUX**
CONDITION_AUX	else **BODY** | ε
CONSTANT	id **CONSTANT_AUX** | CONSTANT_AUX3 cst_whole | CONSTANT_AUX3 cst_decimal | cst_words | cst_boolean
CONSTANT_AUX	squarebracket_open **ITEM** squarebracket_close | parenthesis_open **CONSTANT_AUX1** parenthesis_close | ε
CONSTANT_AUX1	**EXPRESSION CONSTANT_AUX2**

	| ε
CONSTANT_AUX2	comma **EXPRESSION CONSTANT_AUX2** | ε
CONSTANT_AUX3	op_subtraction | ε
DECLARATIONS	**VAR_DECLARATION** **| LIST_DECLARATION**
VAR_DECLARATION	variable id **VAR_DECLARATION_AUX** of type **TYPE** semicolon
VAR_DECLARATION_AUX	comma id **VAR_DECLARATION_AUX** | ε
LIST_DECLARATION	list id squarebracket_open **cst_whole** squarebracket_close of type **TYPE** semicolon
EXPRESSION	**EXPRESSION_AUX EXP EXPRESSION_AUX1**
EXPRESSION_AUX	op_negation | ε
EXPRESSION_AUX1	**op_and EXPRESSION** **| op_or EXPRESSION** **| ε**
EXP	**ITEM EXP_AUX**
EXP_AUX	op_less **ITEM** | op_less_equal **ITEM** | op_greater **ITEM** | op_greater_equal **ITEM** | op_equal **ITEM** | op_not_equal **ITEM** | ε
FACTOR	**parenthesis_open EXPRESSION parenthesis_close** **| CONSTANT**
ITEM	**TERM ITEM_AUX**
ITEM_AUX	op_addition **ITEM** | op_subtraction **ITEM** | ε
TERM	**FACTOR TERM_AUX**
TERM_AUX	op_multiplication **TERM** | op_division **TERM** | ε
STATEMENT	**ASSIGN** **| CONDITION** **| READ** **| WRITE** **| LOOP** **| RETURN** **| CALL**
CALL	call id parenthesis_open **CALL_AUX** parenthesis_close semicolon
CALL_AUX	**EXPRESSION CALL_AUX2**

	| ε
CALL_AUX2	comma **EXPRESSION CALL_AUX2** | ε
LOOP	do **BODY** until parenthesis_open **EXPRESSION** parenthesis_close
ASSIGN	id **ASSIGN_AUX** op_assign **EXPRESSION** semicolon
ASSIGN_AUX	squarebracket_open **ITEM** squarebracket_close | ε
RETURN	return_statement **EXPRESSION** semicolon
READ	input parenthesis_open id **READ_AUX** parenthesis_close semicolon
READ_AUX	squarebracket_open **ITEM** squarebracket_close | ε
WRITE	print parenthesis_open **EXPRESSION WRITE_AUX** parenthesis_close semicolon
WRITE_AUX	comma **EXPRESSION WRITE_AUX** | ε

Intermediate Code: Operation Codes and Virtual Memory

Intermediate code is generated after compiling using the quadruple format. These quadruples are stored in a List that works as a queue to list all quadruples. The quadruples have operation codes as well as virtual memory addresses.

Operation Codes

Operation name	Operation code
OP_ADDITION	1
OP_SUBTRACTION	2
OP_MULTIPLICATION	3
OP_DIVISION	4
OP_ASSIGN	5
OP_GREATER	6
OP_GREATER_EQUAL	7
OP_LESS	8
OP_LESS_EQUAL	9
OP_EQUAL	10
OP_NOT_EQUAL	11
OP_AND	12
OP_OR	13
OP_NOT	14
OP_VERIFY_INDEX	15

OP_GO_TO	16
OP_GO_TO_T	17
OP_GO_TO_F	18
OP_PRINT	19
OP_INPUT	20
OP_ERA	21
OP_PARAM	22
OP_GO_SUB	23
OP_RETURN	24
OP_END_PROC	25

Virtual Memory Addresses

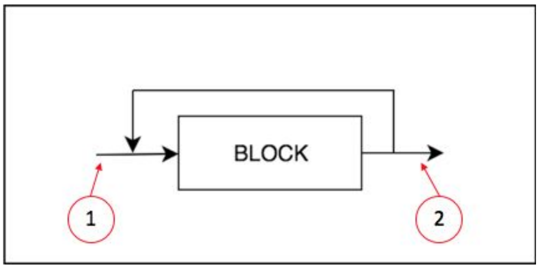
Memory ranges	Stored values
5000 - 5499	Local whole variables
5500 - 5999	Local decimal variables
6000 - 6499	Local words variables
6500 - 6999	Local boolean variables
7000 - 8499	Temporary whole variables
8500 - 9999	Temporary decimal variables
10000- 11499	Temporary words variables
11500 - 12999	Temporary boolean variables
13000 - 13999	Whole constants
14000 - 14999	Decimal constants
15000 - 15999	Words constants
16000 - 16999	Boolean constants

The intermediate code is generated with the previous codes and addresses and can

Syntax Diagrams

The following syntax diagrams were created for the compiler. These are based on the grammar rules stated below and have their corresponding actions that were also implemented in the PLY file.

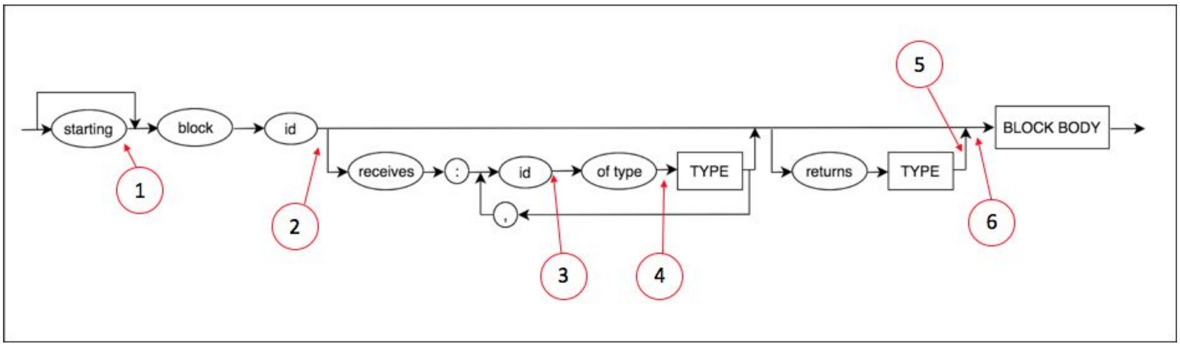
Program



Semantic Actions

1. This action generates the initial “Go-To” quad, and adds it to the pending jumps stack.
2. Validates that one of the blocks declared is the starting block, and throws an error if there isn’t.

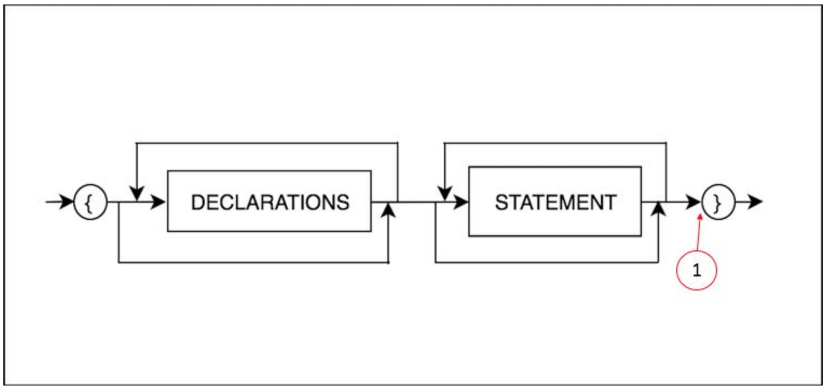
Block



Semantic Actions

1. Assigns the block as the starting block. If there was one already, it throws an error. Also fills the initial ‘Go-To’ quad to this function’s starting quad.
2. Adds the block to the Function Reference Table. Throws an error if there was one with the same name already.
3. Saves the current parameter being read in a global variable
4. Adds the parameter type into the block signature, and adds the parameter into the variable table. Verifies that the parameter name is not a duplicate.
5. Saves the block return type in the block’s row of the Function Reference Table.
6. Saves the block’s initial quad in the block’s row of the Function Reference Table. If it is the starting block, verify that it has no parameters.

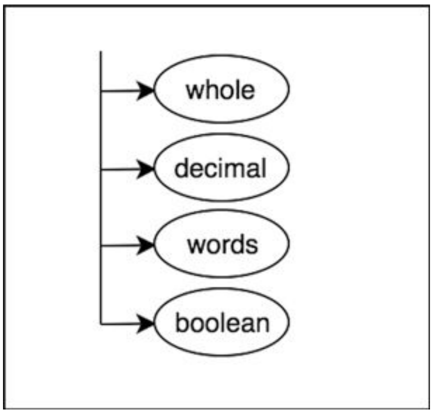
Block Body



Semantic Actions

1. Generates the ‘End-Proc’ quad, and verifies that the return type in the Function Reference Table matches the one in the block body.

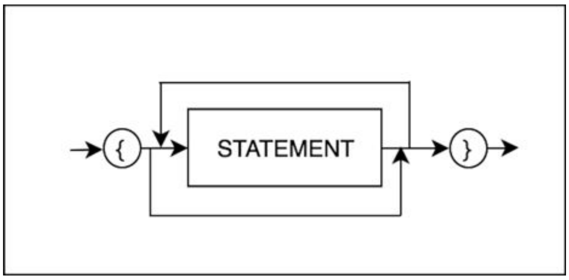
Type



Semantic Actions

N/A

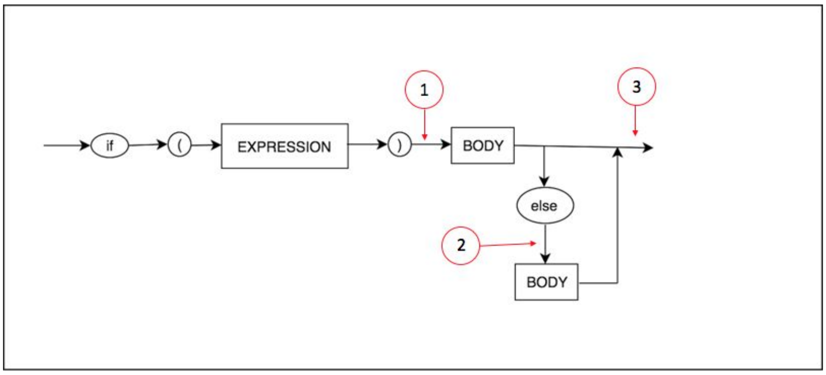
Body



Semantic Actions

N/A

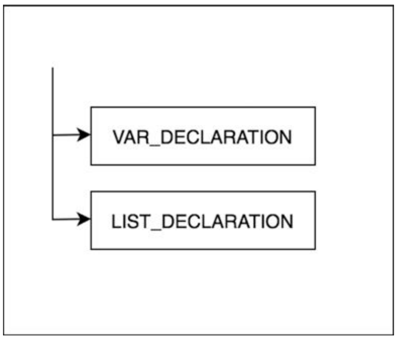
Condition



Semantic Actions

1. Checks that the expression evaluates to a boolean, generates a ‘Go-To-F’ quad and pushes it into the pending jumps stack.
2. Fills the ‘Go-To-F’ quad corresponding to the ‘if’ statement, generates a ‘Go-To’ quad for the else and pushes it into the pending jumps stack.
3. Fills the ‘Go-To-F’ quad corresponding to the ‘else’ statement if there was one, or the ‘if’ statement if there wasn’t.

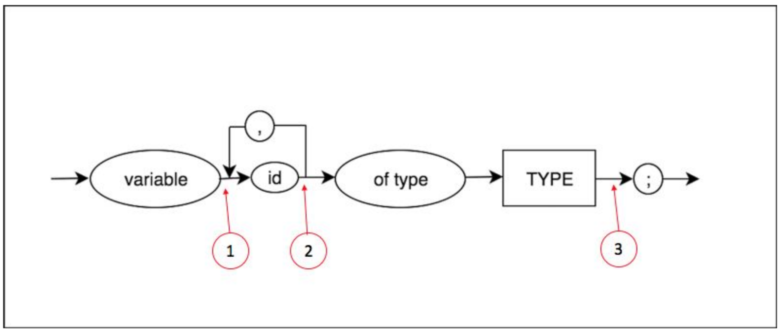
Declarations



Semantic Actions

N/A

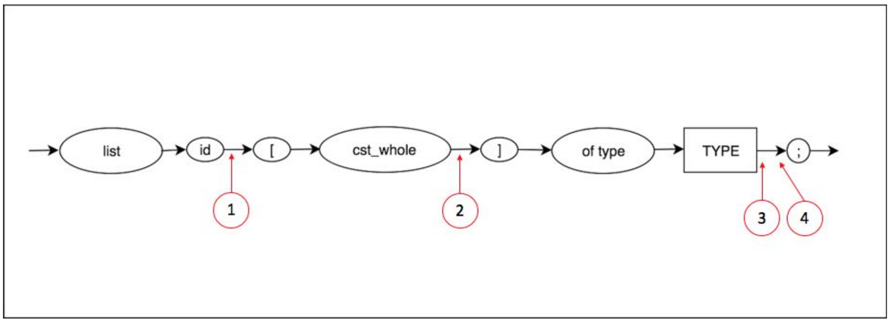
Var Declaration



Semantic Actions

1. Declares an empty array to hold the soon to be declared variables of the same type.
2. Adds the variable id to the array declared in semantic action 1.
3. Verifies that the variable id is not already declared in the current block as another variable, list, or as another block, and adds it to the block’s variable table.

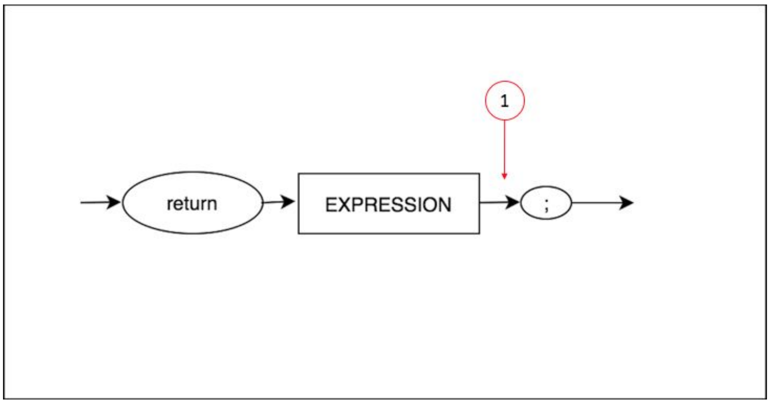
List declaration



Semantic Actions

1. Saves the current list id into a global variable.
2. Verifies that the current list size is not 0, and saves it into a global variable.
3. Saves the current list type into a global variable.
4. Verifies that the list id is not already declared in the current block as a variable, another list, or as another block, and adds it to the block's variable table.

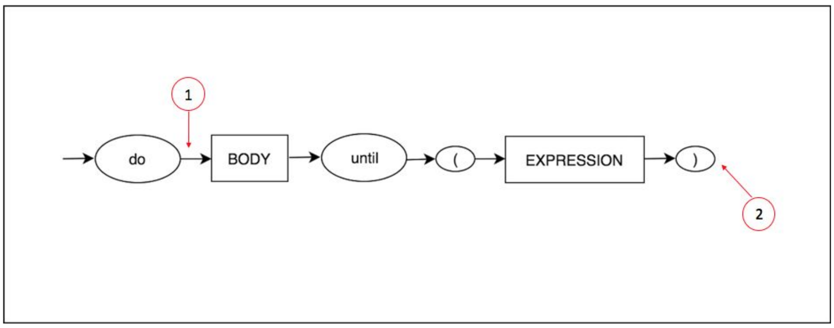
Return



Semantic Actions

1. Verifies that the data type of the expression matches the return type of the block, and generates the 'Return' operation quad.

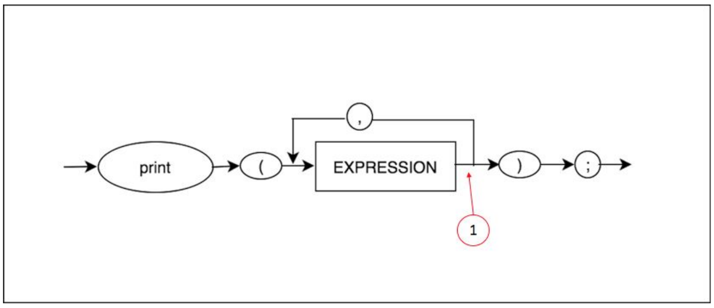
Loop



Semantic Actions

1. Pushes the starting point of the loop into the pending jumps stack
2. Verifies that the expression evaluates to a boolean value, and fills the loop's pending jump

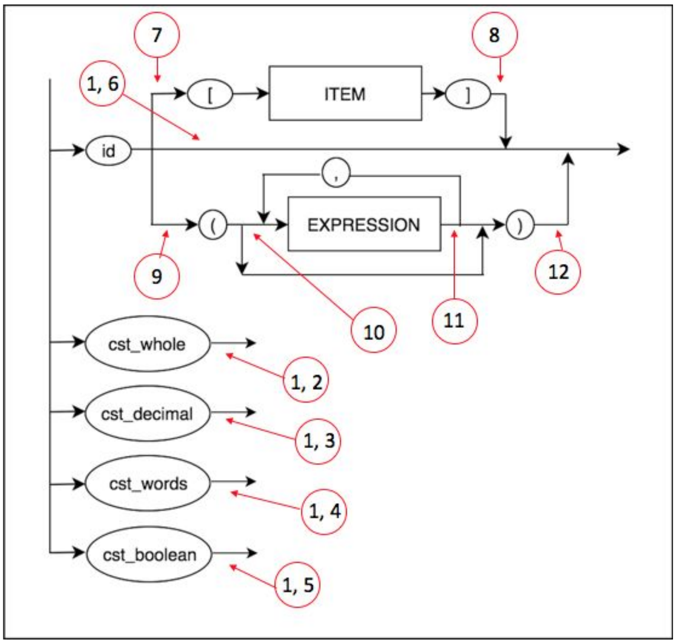
Write



Semantic Actions

1. Generates the 'Write' operation quad

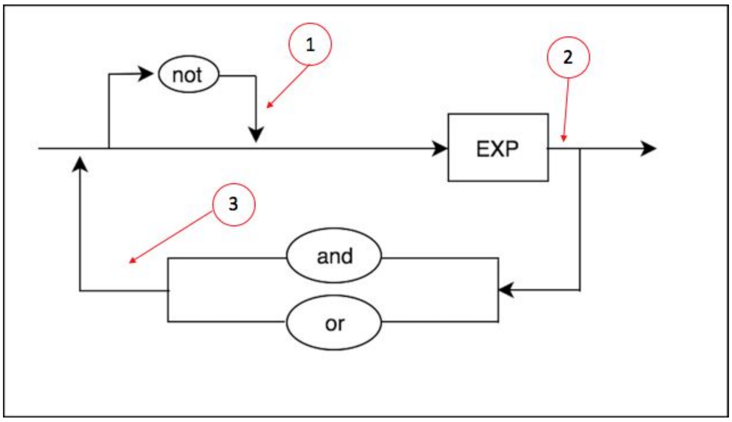
Constant



Semantic Actions

1. Adds the constant to the pending operands stack.
2. Adds the data type 'whole' to the pending operand types stack.
3. Adds the data type 'decimal' to the pending operand types stack.
4. Adds the data type 'words' to the pending operand types stack.
5. Adds the data type 'boolean' to the pending operand types stack.
6. Adds the data type of the id variable or list, or return type of the called block to the pending operand types stack.
7. Verifies that the id belongs to a list, and adds it to the pending lists stack.
8. Verifies that the index resolves to whole, and creates the quads necessary for the list index access.
9. Validates that the block returns a non-void value.
10. Generates the 'ERA' quad, and initializes the parameter count.
11. Validates that the given argument type matches the block's corresponding parameter, and generates the 'Param' quad.
12. Generates the quad for the 'Go-Sub' operation, and the 'Assign' operation for the return value.

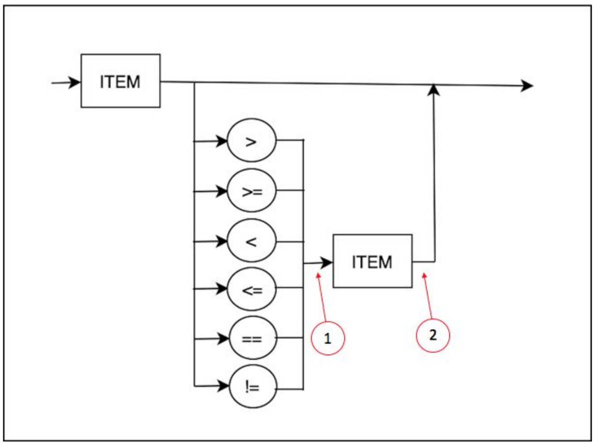
Expression



Semantic Actions

1. Adds a ‘Not’ operator into the pending operators stack.
2. Verifies that the previous value is a boolean value, and generates quads for ‘Not’, ‘And’ and ‘Or’ operators as stated by the order of operations (only if they are in the top of the pending operators stack).
3. Adds either an ‘And’ operator or an ‘Or’ operator to the operators stack.

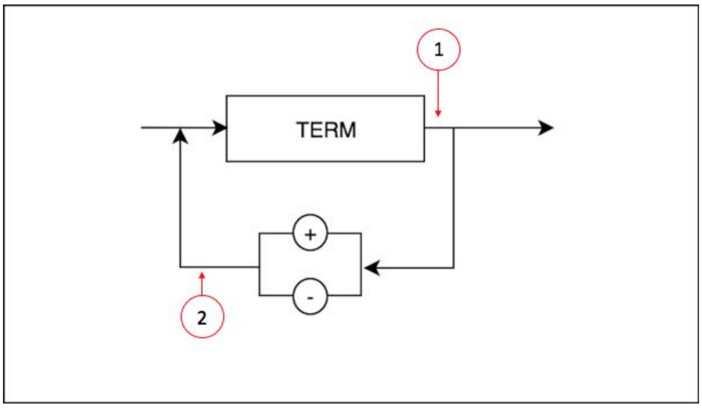
EXP



Semantic Actions

1. Pushes the appropriate relational operator into the pending operators stack.
2. If there is a relational operator in the top of the pending operators stack, verifies that the two operands can be mixed and creates a quad for the appropriate operation.

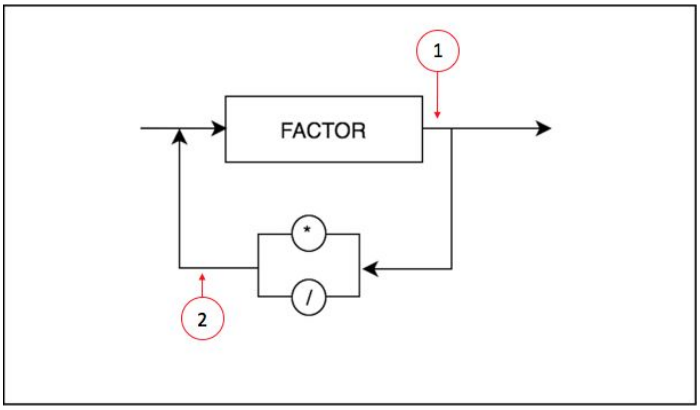
Item



Semantic Actions

1. If there is an ‘Addition’ or ‘Subtraction’ in the pending operators stack, verifies that the two operands can be mixed and generates the appropriate quad.
2. Pushes an ‘Addition’ or ‘Subtraction’ operand into the pending operands stack

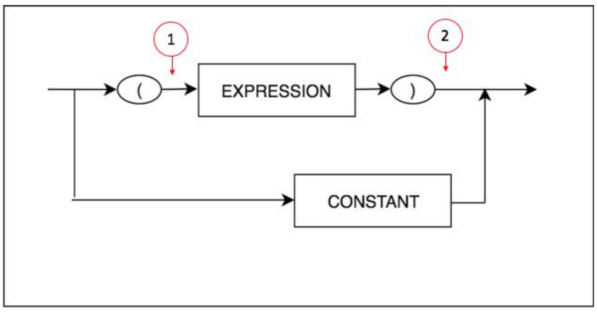
Term



Semantic Actions

1. If there is a Multiplication or ‘Division’ in the pending operators stack, verifies that the two operands can be mixed and generates the appropriate quad.
2. Pushes a ‘Multiplication’ or ‘Division’ operand into the pending operands stack

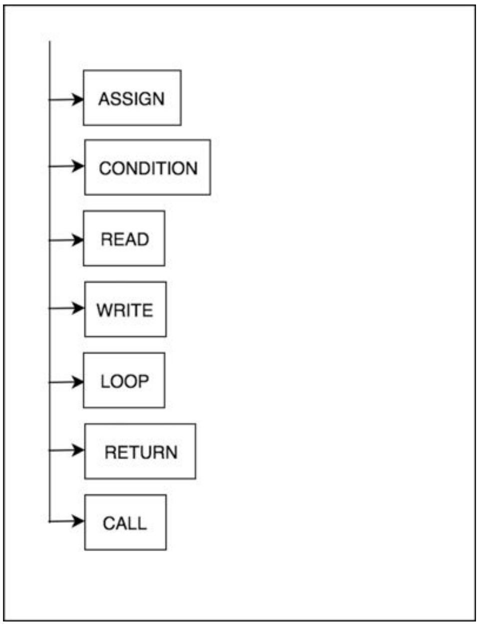
Factor



Semantic Actions

1. Pushes a ‘False Bottom Mark’ into the pending operators stack.
2. Pops the ‘False Bottom Mark’ from the pending operators stack.

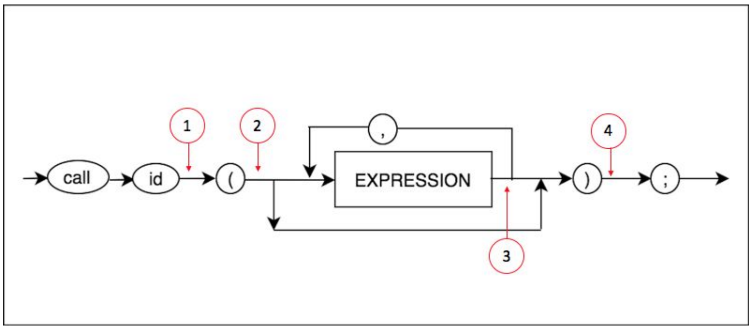
Statement



Semantic Actions

N/A

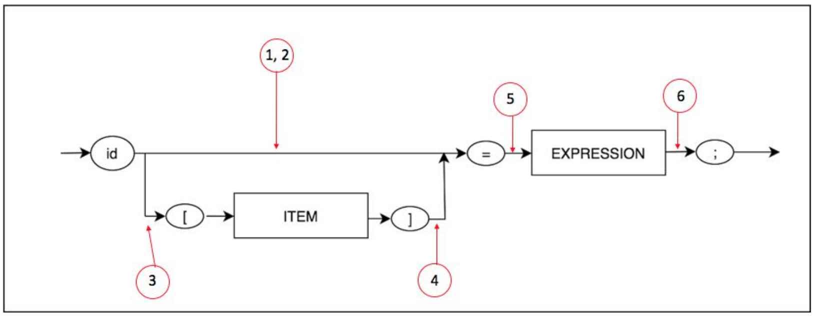
Call



Semantic Actions

1. Verifies that the block exists and has no return value,
2. Generates the 'ERA' quad, and initializes the parameter count.
3. Validates that the given argument type matches the block's corresponding parameter, and generates the 'Param' quad.
4. Validates the number of parameters, and generates the 'Go Sub' operation quad

Assign

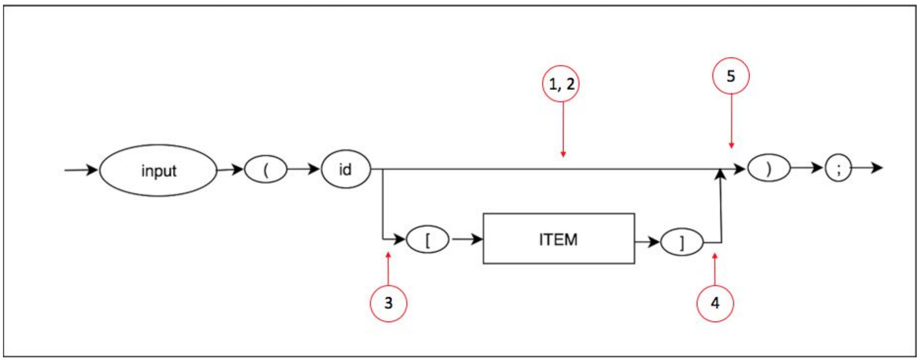


Semantic Actions

1. Adds the variable name into the pending operands stack.
2. Verifies that the variable name exists in the current scope and adds the type to the pending operands types stack.
3. Verifies that the list exists in the current scope and adds the list name into the pending lists stack.
4. Verifies that the index resolves to whole, and creates the quads necessary for the list index access.
5. Adds the 'Assign' operator into the pending operators stack.
6. If the pending operators stack is an 'Assign' operator, verify that the right operand can be stored in the left, and generate the corresponding quad.

2.

Read



Semantic Actions

1. Adds the variable name into the pending operands stack.
2. Verifies that the variable name exists in the current scope and adds the id to the pending operands stack and id type to the pending operands types stack.
3. Verifies that the list exists in the current scope and adds the list name into the pending lists stack.
4. Verifies that the index resolves to whole, and creates the quads necessary for the list index access.
5. Generates the quad for the 'Read' operation

Semantic Characteristics

- The keyword “starting” denotes the initial function.
- Variables have to be declared before being used in any way.
- Id names can’t be repeated.
- A square bracket should only be used after an id that represents an array.
- Variables have a local scope.
- Only the following operations between types are allowed (note that the order of the types are irrelevant and every other combination is considered an error)

<table>
<tr><th colspan="4">Valid Operations</th></tr>
<tr><th>Operand 1</th><th>Operator</th><th>Operand 2</th><th>Result</th></tr>
<tr><td>whole</td><td>+ | - | * | / | =</td><td>whole</td><td>whole</td></tr>
<tr><td>whole</td><td>< | <= | > | >= | == | !=</td><td>whole</td><td>boolean</td></tr>
<tr><td>whole</td><td>+ | - | * | /</td><td>decimal</td><td>decimal</td></tr>
<tr><td>whole</td><td>< | <= | > | >= | == | !=</td><td>decimal</td><td>boolean</td></tr>
<tr><td>decimal</td><td>+ | - | * | / | =</td><td>whole</td><td>decimal</td></tr>
<tr><td>decimal</td><td>+ | - | * | / | =</td><td>decimal</td><td>decimal</td></tr>
<tr><td>decimal</td><td>< | <= | > | >= | == | !=</td><td>decimal</td><td>boolean</td></tr>
<tr><td>words</td><td>+ | =</td><td>words</td><td>words</td></tr>
<tr><td>words</td><td>== | !=</td><td>words</td><td>boolean</td></tr>
<tr><td>boolean</td><td>and | or | not | == | != | =</td><td>boolean</td><td>boolean</td></tr>
</table>

Memory Handling

Memory handling and data structures in compilation are defined by the following tables.
*Dictionaries/Custom classes in bold are shown in their own table

Function Directory (Class)			
starting block key	**function reference table**	**constant table**	memory handler
int	Dictionary	Dictionary	**Memory_Handler**

Function Reference Table (Dictionary)						
Key	**Value**					
block name	return type	**variables**	parameters	quad position	local type counter	temporary type counter
string	string	Dictionary list	string list	int	int list	int list

Variables (Dictionary List)	
primitives	**lists**
Dictionary	Dictionary

Primitives (Dictionary)		
Key	**Value**	
variable name	variable type	memory address
string	string	int

Lists (Dictionary)			
Key	**Value**		
list name	list type	memory address	list size
string	string	int	int

Constant Table (Dictionary)		
Key	**Value**	
constant value	constant type	memory address
int/float/string/boolean	string	int

Memory Handler (Class)								
Local Counter	Local Ranges	Local Size	Temp Counter	Temp Ranges	Temp Size	Const Counter	Const Ranges	Const Size
int list	int list	int	int list	int list	int	int list	int list	int

Quad (Class)			
operator	left operand	right operand	result
int	int/string	int	int/string

*Also used in execution

Code_Review_Data (Class)						
total var counter	total loop counter	total if counter	num ar on call	max num ar	**block data**	total run time
int	int	int	float list	int	Dictionary	int

*Also used in execution

Block Data (Dictionary)								
Key	**Value**							
block name	compiled quad counter	variable counter	compiled loop counter	if counter	executed quad counter	num calls	**executed loop counter**	run time
string	int	int	int	int	int	int	Dictionary	int

Executed Loop Counter (Dictionary)	
Key	**Value**
quad number	counter
int	int

Virtual Machine Description

The virtual machine works together with the REST API implementation of the compiler to generate corresponding results during Compilation and Runtime. The virtual machine in itself is described separately from the REST API.

Tools

To host the compiler for front-end use, Flask and Flask-CORS were imported and used to encompass the compiler into a REST API. The API is then hosted using Heroku.

REST API Implementation

The compiler is encapsulated by a REST API implementation. The API contains 4 get/post functions that call on the compiler handler that then calls the compiler. The handler exists in order to control the flow of data between the server and the functions it can access from the compiler.

REST method	Function	Parameter	Return Value
post_code (POST)	Sends code to compiler	Code: String	Result code: int
get_code (GET)	Receives code from compiler	N/A	Code: String
compile (GET)	Triggers compilation and execution	N/A	Compilation result and data: json data
post_input (POST)	Sends input to compiler	User_input: String	Compilation result and data: json data

Memory in Runtime Architecture

Memory is handled with structures and an offset for virtual addresses as defined previously. Each data type and scope is contained in its own structure and therefore the offset is calculated differently for each.

Memory Data Structures

Memory Handling and structures in execution diagrams are as follows.

Program Memory (Class)						
Quad Memory	Local Ranges	Local Size	Temp Ranges	Temp Size	Stack Segment	Stack Segment Limit
Quad list	int list	int	int list	int	**Activation Record** Stack	int

Program Memory (Class) - Continued						
constant counter	constant ranges	constant size	const whole memory	const decimal memory	const words memory	const boolean memory
int list	int list	int	int list	int list	int list	int list

Activation Record (Class)			
block name	block memory	return value	return address
string	**Block_Memory**	int/float/string/boolean	int

Block_Memory (Class)					
local counter	local ranges	local size	temp counter	temp ranges	temp size
int list	int list	int	int list	int list	int

Block_Memory (Class) - Continued							
local whole memory	local decimal memory	local words memory	local boolean memory	temp whole memory	temp decimal memory	temp words memory	temp boolean memory
int list	int list	int list	int list	int list	int list	int list	int list

Virtual vs. Real Memory Addresses

As mentioned earlier, there are virtual addresses for memory. The way data is structured is as follows.

Program Memory

Quad List	read-only, real address 0-4999
Stack Segment	real address 0-499
Constants	real address 0-999 (for each type)

Stack Segment

Local	read-only, real address 0-499
Stack Segment	real address 0-1499

Testing

Testing was crucial to ensure the correct compilation, runtime, and front-end behavior of the compiler. There were several tests created to facilitate this process.

Tests

1. **Test title:** Simple print test
   Description: Tests a simple print statement, block definition, and semantics.

Code

```
starting block Block14
{
 print("hi");
}
```

Results (intermediate code):

Quadruples

```
0        16      -1       -1       1
1        19      15000    -1       -1
2        25      -1       -1       -1
```

Execution results:

Output

```
hi
```

2. **Test title:** Simple Hello Test and Arithmetics
   Description: Tests a simple print statement, block definition, variable declaration, variable assignation, conditional statement, and operators for operands that are using different data types.

Code

```
starting block Block14
{
        variable h, i of type words;
        variable a, b of type whole;
        h = "Hello";
        i = " World";

        a = 1;
        b = 3;

        if (a > 0)
        {
         print(a);
        }
```

```
        print(h + i);
        print(a);
        print(b);
        print(a+b);
}
```

Results (intermediate code):

Quadruples									
0	16	-1	-1	1	7	19	5000	-1	-1
1	5	15000	-1	6000	8	1	6000	6001	10000
2	5	15001	-1	6001	9	19	10000	-1	-1
3	5	13000	-1	5000	10	19	5000	-1	-1
4	5	13001	-1	5001	11	19	5001	-1	-1
5	6	5000	13002	11500	12	1	5000	5001	7000
6	18	11500	-1	8	13	19	7000	-1	-1
					14	25	-1	-1	-1

Execution results:

Output

```
1
Hello World
1
3
4
```

3. **Test title:** Compilation Error
Description: Tests a compilation error triggered by assigning a words value to a whole data type.

Code

```
starting block Block14
{
        variable h, i of type words;
        variable a, b, c of type whole;
        h = "Hello";
        a = " World";

        a = 1;
        b = 3;
        c = 0;

        if (a > 0)
        {
         print(a);
        }

        a = a/c;

        print(h + i);
        print(a);
        print(b);
        print(a+b);
}
```

Results (intermediate code):

Quadruples	
0 16 -1 -1 1 1 5 15000 -1 6000	

Execution results:

Output
Compilation error in line 6: Expression of type 'words' cannot be assigned to ID of type 'whole'

4. **Test title:** Lexical Error
Description: Tests a lexical error triggered by assigning an invalid id

Code

```
starting block Block14
{
        variable h, i of type words;
        variable a, b, c_start of type whole;
        h = "Hello";
        i = " World";

        a = 1;
        b = 3;
        c_start = 0;

        if {} (a > 0)
        {
         print(a);
        }

        a = a/c;

        print(h + i);
        print(a);
        print(b);
        print(a+b);
}
```

Results (intermediate code):

Quadruples	
0 16 -1 -1 1	

Execution results:

Output
Compilation error in line 9: Unexpected token 'start' found

5. **Test title:** Runtime Error
Description: Tests a runtime error triggered by dividing by 0

Code

```
starting block Block14
{
        variable h, i of type words;
        variable a, b, c of type whole;
        h = "Hello";
        i = " World";

        a = 1;
        b = 3;
        c = 0;

        if (a > 0)
        {
         print(a);
        }

        a = a/c;

        print(h + i);
        print(a);
        print(b);
        print(a+b);
}
```

Results (intermediate code):

Quadruples

0	16	-1	-1	1	9	4	5000	5002	7000
1	5	15000	-1	6000	10	5	7000	-1	5000
2	5	15001	-1	6001	11	1	6000	6001	10000
3	5	13000	-1	5000	12	19	10000	-1	-1
4	5	13001	-1	5001	13	19	5000	-1	-1
5	5	13002	-1	5002	14	19	5001	-1	-1
6	6	5000	13002	11500	15	1	5000	5001	7001
7	18	11500	-1	9	16	19	7001	-1	-1
8	19	5000	-1	-1	17	25	-1	-1	-1

Execution results:

Output

```
1
Runtime error: Cannot divide by 0
```

6. **Test title:** Binary Search
Description: Tests a loop statement that can meet expressions and do assignments, operations, and other statements.

Code

```
starting block main
{
    list a[300] of type whole;
    variable i, size, value, val of type whole;
    variable found of type boolean;

    size = 299;
    value = 60;
```

```
    val = 0;
    i = 0;
    do {
        a[i] = val;
        val = val + 1;
        i = i + 1;
    } until(i > size)

    i = 0;
    do {
        i = i + 1;
    } until(i > size)

    found = False;
    i = 0;
    do {
        if(a[i] == value){
            found = True;
        }

        i = i + 1;
    } until(i > size or found == True)

    if(found == True){
        print("found a", value, "at index", i);
    }
    else{
        print(value, "not found");
    }
}
```

Results (intermediate code):

Quadruples									
0	16	-1	-1	1	21	15	5300	300	-1
1	5	13000	-1	5301	22	1	5300	&5000	*7004
2	5	13001	-1	5302	23	10	*7004	5302	11502
3	5	13002	-1	5303	24	18	11502	-1	26
4	5	13002	-1	5300	25	5	16001	-1	6500
5	15	5300	300	-1	26	1	5300	13003	7005
6	1	5300	&5000	*7000	27	5	7005	-1	5300
7	5	5303	-1	*7000	28	6	5300	5301	11503
8	1	5303	13003	7001	29	10	6500	16001	11504
9	5	7001	-1	5303	30	13	11503	11504	11505
10	1	5300	13003	7002	31	18	11505	-1	21
11	5	7002	-1	5300	32	10	6500	16001	11506
12	6	5300	5301	11500	33	18	11506	-1	39
13	18	11500	-1	5	34	19	15000	-1	-1
14	5	13002	-1	5300	35	19	5302	-1	-1
15	1	5300	13003	7003	36	19	15001	-1	-1
16	5	7003	-1	5300	37	19	5300	-1	-1
17	6	5300	5301	11501	38	16	-1	-1	41
18	18	11501	-1	15	39	19	5302	-1	-1
19	5	16000	-1	6500	40	19	15002	-1	-1
20	5	13002	-1	5300	41	25	-1	-1	-1

Execution results:

Output
found a 60 at index 61

7. **Test title:** Binary Sort

Description: Tests a loop statement that can meet expressions and do assignments, operations, and other statements. Most importantly, it can test list manipulation.

Code

```
starting block main
{
    variable i, j, val, temp, count of type whole;
    list a[10] of type whole;
    val = 9;
    count = 0;

    i = 0;
    do {
        a[i] = val;
        i = i + 1;
        val = val - 1;
    } until(i > 9)

    i = 0;
    do {
        print(a[i]);
        i = i + 1;
    } until(i > 9)

    i = 1;
    do {
        j = 0;
        do {
             count = count + 1;
            if (a[j] > a[j + 1]) {
                temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
            j = j + 1;
        } until (j > (9 - i))
        i = i + 1;
    } until (i > 9)

    i = 0;
    do {
        print(a[i]);
        i = i + 1;
    } until(i > 9)

    print(count);
}
```

Results (intermediate code):

Quadruples									
0	16	-1	-1	1	32	15	5001	10	-1
1	5	13000	-1	5002	33	1	5001	&5005	*7009
2	5	13001	-1	5004	34	5	*7009	-1	5003
3	5	13001	-1	5000	35	15	5001	10	-1
4	15	5000	10	-1	36	1	5001	&5005	*7010
5	1	5000	&5005	*7000	37	1	5001	13002	7011
6	5	5002	-1	*7000	38	15	7011	10	-1
7	1	5000	13002	7001	39	1	7011	&5005	*7012
8	5	7001	-1	5000	40	5	*7012	-1	*7010
9	2	5002	13002	7002	41	1	5001	13002	7013
10	5	7002	-1	5002	42	15	7013	10	-1
11	6	5000	13000	11500	43	1	7013	&5005	*7014
12	18	11500	-1	4	44	5	5003	-1	*7014
13	5	13001	-1	5000	45	1	5001	13002	7015
14	15	5000	10	-1	46	5	7015	-1	5001
15	1	5000	&5005	*7003	47	2	13000	5000	7016
16	19	*7003	-1	-1	48	6	5001	7016	11503
17	1	5000	13002	7004	49	18	11503	-1	23
18	5	7004	-1	5000	50	1	5000	13002	7017
19	6	5000	13000	11501	51	5	7017	-1	5000

20	18	11501	-1	14	52	6	5000	13000	11504
21	5	13002	-1	5000	53	18	11504	-1	22
22	5	13001	-1	5001	54	5	13001	-1	5000
23	1	5004	13002	7005	55	15	5000	10	-1
24	5	7005	-1	5004	56	1	5000	&5005	*7018
25	15	5001	10	-1	57	19	*7018	-1	-1
26	1	5001	&5005	*7006	58	1	5000	13002	7019
27	1	5001	13002	7007	59	5	7019	-1	5000
28	15	7007	10	-1	60	6	5000	13000	11505
29	1	7007	&5005	*7008	61	18	11505	-1	55
30	6	*7006	*7008	11502	62	19	5004	-1	-1
31	18	11502	-1	45	63	25	-1	-1	-1

Execution results:

Output	
9 8 7 6 5 4 3 2 1 0 0	1 2 3 4 5 6 7 8 9 45

8. **Test title:** Recursion
Description: Tests a recursive call to a function to calculate the nth number of the fibonacci series.

Code

```
block recursiveFibonacci
receives:
n of type whole
returns whole
{
        if(n == 1 or n == 2){
                return 1;
        }
        else{
                return recursiveFibonacci(n-1) + recursiveFibonacci(n-2);
        }
}

starting block main
{
        variable n of type whole;
        n = 21;

        print(recursiveFibonacci(n));
}
```

Results (intermediate code):

Quadruples				
0	16	-1	-1	20
1	10	5000	13000	11500
2	10	5000	13001	11501
3	13	11500	11501	11502
4	18	11502	-1	7
5	24	13000	-1	-1

```
6         16      -1       -1       19
7         21      recursiveFibonacci         -1       -1
8         2       5000     13000    7000
9         22      7000     -1       5000
10        23      recursiveFibonacci         -1       1
11        5       recursiveFibonacci         -1       7001
12        21      recursiveFibonacci         -1       -1
13        2       5000     13001    7002
14        22      7002     -1       5000
15        23      recursiveFibonacci         -1       1
16        5       recursiveFibonacci         -1       7003
17        1       7001     7003     7004
18        24      7004     -1       -1
19        25      -1       -1       -1
20        5       13002    -1       5000
21        21      recursiveFibonacci         -1       -1
22        22      5000     -1       5000
23        23      recursiveFibonacci         -1       1
24        5       recursiveFibonacci         -1       7000
25        19      7000     -1       -1
26        25      -1       -1       -1
```

Execution results:

Output
5

9. **Test title:** Expressions test
Description: Tests a large expression being assigned to a list element, tests order of operations.

Code

```
starting block Block14
{
        variable a, b of type decimal;
        list c[6] of type decimal;
        variable d, e of type decimal;

        b = 1;
        c[3] = 5;

        c[1] = b + 1.3 * 1.5 + 1 + c[3];

        print(c[1]);
}
```

Results (intermediate code):

Quadruples

```
0         16      -1       -1       1
1         5       13000    -1       5501
2         15      13001    6        -1
3         1       13001    &5502    *7000
4         5       13002    -1       *7000
5         15      13000    6        -1
6         1       13000    &5502    *7001
7         3       14000    14001    8500
8         1       5501     8500     8501
9         1       8501     13000    8502
10        15      13001    6        -1
11        1       13001    &5502    *7002
12        1       8502     *7002    8503
13        5       8503     -1       *7001
14        15      13000    6        -1
15        1       13000    &5502    *7003
16        19      *7003    -1       -1
17        25      -1       -1       -1
```

Execution results:

Output
8.95

10. **Test title:** Input and call to block
Description: Tests an input action assignment.

Code

```
block Block15
{
    variable a of type decimal;
    a = 10;

    print("en block 15, a vale ", a);
}

starting block Block14
{
         variable a, b, c, d, e of type decimal;
         variable g of type whole;
         variable h, i of type words;
         a = 1;
         b = 2;
         c = 5;
         d = 0;
         g = 1;

         if(not g != g and not g < g or a > b){
                  g = 5;
                  call Block15();
        print("en block 14 pero en el if, a vale ", a);
         }

    print("en block 14 fuera del if, a vale ", a);

         h = ", como estas?";
         print(h);

         input(i);
         print(i);

         print("custom greeting: " + i + h);
}
```

Results (intermediate code):

Quadruples

0	16	-1	-1	5	17	18	11506	-1	23
1	5	13000	-1	5500	18	5	13003	-1	5000
2	19	15000	-1	-1	19	21	Block15	-1	-1
3	19	5500	-1	-1	20	23	Block15	-1	1
4	25	-1	-1	-1	21	19	15001	-1	-1
5	5	13001	-1	5500	22	19	5500	-1	-1
6	5	13002	-1	5501	23	19	15002	-1	-1
7	5	13003	-1	5502	24	19	5500	-1	-1
8	5	13004	-1	5503	25	5	15003	-1	6000
9	5	13001	-1	5000	26	19	6000	-1	-1
10	11	5000	5000	11500	27	20	words	-1	6001
11	14	11500	-1	11501	28	19	6001	-1	-1
12	8	5000	5000	11502	29	1	15004	6001	10000
13	14	11502	-1	11503	30	1	10000	6000	10001
14	12	11501	11503	11504	31	19	10001	-1	-1
15	6	5500	5501	11505	32	25	-1	-1	-1
16	13	11504	11505	11506					

Execution results:

Output	
en block 15, a vale 10 en block 14 pero en el if, a vale 1	en block 14 fuera del if, a vale 1 , como estas? Hola custom greeting: Hola, como estas?

11. **Test title:** Parameters
Description: Tests sending parameter and calling a block several times.

Code

```
block Block15
receives:
a of type whole
{
    print(a);
}

starting block Block14
{
    variable a of type whole;
    a = 5;
    print(a);

    call Block15(1);
    call Block15(2);
    call Block15(3);
}
```

Results (intermediate code):

Quadruples

0	16	-1	-1	3	7	23	Block15	-1	1
1	19	5000	-1	-1	8	21	Block15	-1	-1
2	25	-1	-1	-1	9	22	13002	-1	5000
3	5	13000	-1	5000	10	23	Block15	-1	1
4	19	5000	-1	-1	11	21	Block15	-1	-1
5	21	Block15	-1	-1	12	22	13003	-1	5000
6	22	13001	-1	5000	13	23	Block15	-1	1
					14	25	-1	-1	-1

Execution results:

Output	
5 1	2 3

Code

The code is completely documented and is currently in Github. The final product is on a separate section titled Final Product, as previous deliveries are documented as well.

Code comments

Comments for code are present in python as ”#<comment>” or as text inside three quotation marks. Comment for Javascript are present as “//<comment>”

Important Functions

The most relevant functions are shown in the following descriptions.

Function name: read_from_memory
Location: Memory.py -> class Program_Memory
Description: The reason we decided that this function was important is because it shows that the client is 100% independent and unaware of the memory limits and ranges. The client program only calls the appropriate Program Memory and it will be in charge of deciding which value to return, even if it receiving a pointer.

Code

```python
#Reads a value from  memory
def read_from_memory(self, address = None):
    #address must not be empty
    if address is not None:
        #address is special character for memory location
        if str(address)[0] == '&':
            #return addres
            num = int(address[1:])
            return num

        #address is special character for memory location
        elif str(address)[0] == '*':
            num = int(address[1:])
            return self.read_from_memory(self.read_from_memory(num))

        #address is regular address
        else:
            #read from current (top of stack) activation record's local memory
            if address >= self._local_ranges[0] and address < (self._local_ranges[3] + self._local_size):
                current_activation_record = self._stack_segment.peek()
                value = current_activation_record.get_block_memory().read_from_local_memory(address)
                return value

            #read from current (top of stack) activation record's temporary memory
            elif address >= self._temporary_ranges[0] and address < (self._temporary_ranges[3] + self._temporary_size):
                current_activation_record = self._stack_segment.peek()
                value = current_activation_record.get_block_memory().read_from_temporary_memory(address)
                return value

            #read from constant table
            elif address >= self._constant_ranges[0] and address < (self._constant_ranges[3] + self._constant_size):
                #add memory to constant
                value = self._read_from_constant_memory(address)
                return value

            else:
                #error
                return None
```

Function name: input_operation
Location: virtual_machine.py
Description: Input stops execution so there is a custom error being thrown when asking for input. The error is handled and the appropriate steps are made to ensure input return. After input is returned, execution continues and this method is the first to be called. With a current value for user_input, the value is assigned and no error is thrown. Right after, input is cleared so that it can be used as a flag for further input.

Code

```python
#Reads from console
def input_operation(current_instruction):
    input_address = current_instruction.get_result()
    input_type = current_instruction.get_left_operand()

    #If there is no input from the user
    if (global_scope.user_input == ""):
        increase_run_time()
        #raise input error to stop execution until there is an input
        raise constants.ChooChooInput()

    #if the function did not raise an error, the user sent an input
    input_value = global_scope.user_input

    #reset input value for future input operations
    global_scope.user_input = ""

    validated_input = validate_input(input_value, input_type)

    #validate that the input is the expected data type
    if validated_input is not None:
        global_scope.program_memory.write_to_memory(validated_input, input_address)
        global_scope.instruction_pointer += 1
    else:
        return stop_exec("Input value '%s' is not of type '%s'" % (input_value, input_type))
```

Function name: era_operation
Location: virtual_machine.py
Description: Whenever a new function is called, we generate a new Activation Record. This is done by sending the amount of memory needed (local variables and temporary variables) and the name of the block, and our Program Memory can easily create the needed memory in an easy to use structure.

Code

```python
#Generates an activation record
def era_operation(current_instruction):
    block_name = current_instruction.get_left_operand()
    local_counter_for_block = global_scope.function_directory.get_local_type_counter(block_name)
    temporary_counter_for_block = global_scope.function_directory.get_temporary_type_counter(block_name)

    #Creates a new AR, leaving the return address pending for when the go_to_sub operation is called
    global_scope.temp_activation_record = Activation_Record(block_name, local_counter_for_block, temporary_counter_for_block, -1)

    global_scope.instruction_pointer += 1
```

Function name: initialize_memory
Location: virtual_machine.py
Description: This is how we generate our Program Memory; We port the quad list into it, generate an activation record for the starting block and push it into the stack segment, and port the constant table. We could now erase all information from the Function Reference Table safely.

Code

```python
#Initializes memory for Run-Time
def initialize_memory():
    #Reads all of the necessary info from the FRT
    global_scope.starting_block = global_scope.function_directory.starting_block_key
    starting_local_type_counter = global_scope.function_directory.get_local_type_counter(global_scope.starting_block)
    starting_temporary_type_counter = global_scope.function_directory.get_temporary_type_counter(global_scope.starting_block)
    starting_activation_record = Activation_Record(global_scope.starting_block, starting_local_type_counter, starting_temporary_type_counter, 0)
    cc = global_scope.function_directory.memory_handler.get_constant_counter()

    #Creates the program memory
    aux_program_memory = Program_Memory(global_scope.quad_list, starting_activation_record, cc, global_scope.function_directory.constant_table)
    global_scope.program_memory = aux_program_memory
```

Function name: p_EC_SEEN_BLOCK_ID
Location: parser.py
Description: This is how we add new blocks into our Function Reference Table, a key step in the compilation process. We have to check that there is no other block with the same name, and set it as the starting block if needed.

Code

```python
#BLOCK action 2 - Creates a new row in the FRT for a new block
def p_EC_SEEN_BLOCK_ID(p):
    "EC_SEEN_BLOCK_ID : "
    global_scope.block_returns = False
    global_scope.current_block_id = p[-1]

    #Block name should not be a duplicate
    if not global_scope.function_directory.block_id_exists(global_scope.current_block_id):
        if global_scope.is_starting_block:
            global_scope.function_directory.starting_block_key = global_scope.current_block_id

        global_scope.function_directory.add_block_name(global_scope.current_block_id)
        global_scope.code_review.add_entry(global_scope.current_block_id)
    else:
        stop_exec("Block named '%s' is already defined" % global_scope.current_block_id)
```

Function name: p_EC_SEEN_CONST_LIST
Location: parser.py
Description: This function is in charge of creating the quads necessary to access a list index. The way we handle the special variable that holds an address is by using an '*', the universal symbol for pointers in programming languages. Our memory classes know how to handle arrays for both reading and writing.

Code

```python
#Value in result is an address
result = constants.Misc.POINTER + str(result)

if not global_scope.quad_list.append_quad(constants.Operators.OP_ADDITION, list_index, list_address, result):
    stop_exec("Number of operations permitted has surpassed the limit (%i)" % constants.Memory_Limits.QUAD_SIZE)
```

Function name: continue_execution
Location: choo_choo_train.py
Description: This function is called after input is posted to the REST API. This function sets the value for user_input to global_scope which will then allow execution to continue. execute_code is called and execution continues. This can return several status codes that are handled 1-success, 0-runtime error, 3-input needed. These status codes are handled by the front-end.

Code

```python
def continue_execution():
    #send input to virtual machine
    #virtual_machine.set_input(attributes.user_input)
    global_scope.user_input = attributes.user_input

    try:
        #RERUN CODE - r_status returns 0 if there is a runtime error
        r_status = virtual_machine.execute_code()
    except constants.ChooChooInput as error:
        r_status = 3

    #set attributes for index 3
    attributes.output = global_scope.output_builder

    #set attributes for index 1 from compiler
    attributes.runtime = global_scope.code_review.total_run_time

    #set attributes for index 4, check for runtime error or input error
    if r_status == 0:
        #some runtime error occurred, return default values and error code
        attributes.compilationstatus = 0

    elif r_status == 3:
        #get input
        attributes.compilationstatus = 3

    else:
        #runtime complete
        attributes.compilationstatus = 1

    #set attributes for index 9
    attributes.last_output = global_scope.last_output
```