Tecnológico de Monterrey

Choo Choo Train

David Benitez - A01191731
Paulina Escalante - A01191962

March 6, 2017

Purpose of Project

The purpose of this project is to help children get an insight into the world of programming from a small age. Children 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 children can relate programming to something that is familiar to them, then we might be able to help them understand what programming really is about, and possibly encourage them to pursue a Computer Science Degree in the future.

Children will use our visual language to "program" blocks, using a visual representation tool of code. Then they will be able to watch their code being executed, represented as a train going through its tracks. 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).

Main Objective and Area

The main objective of our language is to be understandable, visual, and easy to use so that children 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 execution of the program, as well as any output generated. With this visual representation, complicated programming terminology and structures are abstracted and represented with objects familiar to children, in this case a train track environment. Its objective is to be an educational, fun, and pleasing programming language for children.

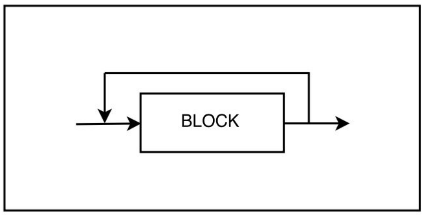
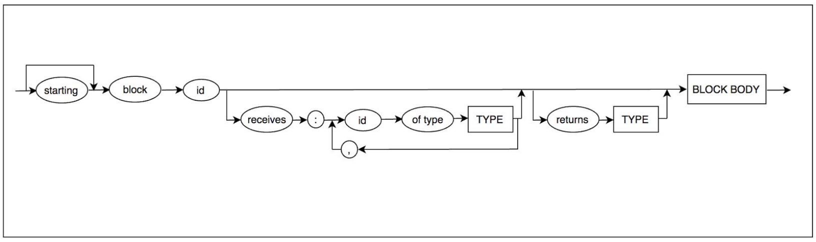
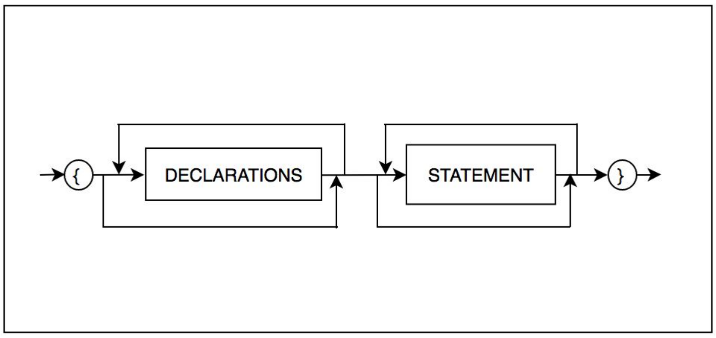
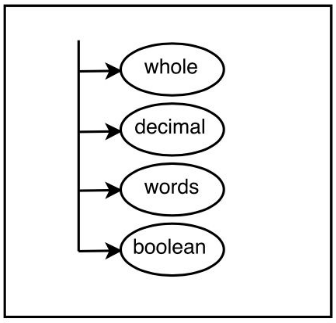
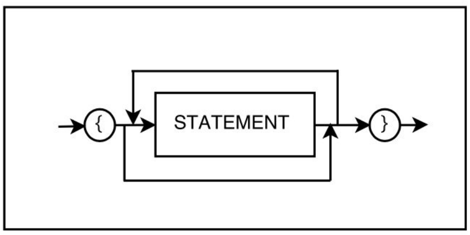
Language Requirements

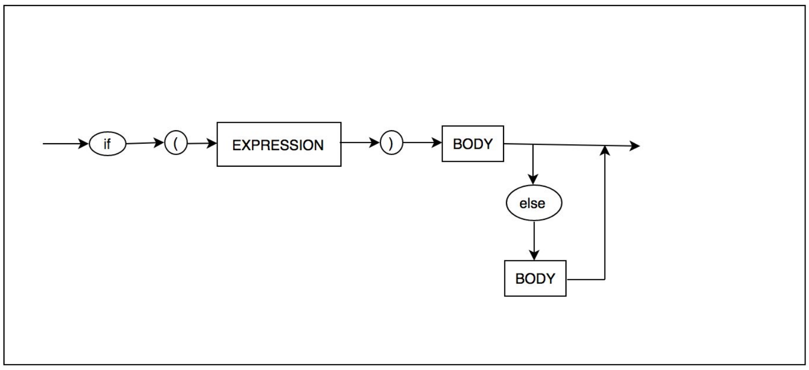
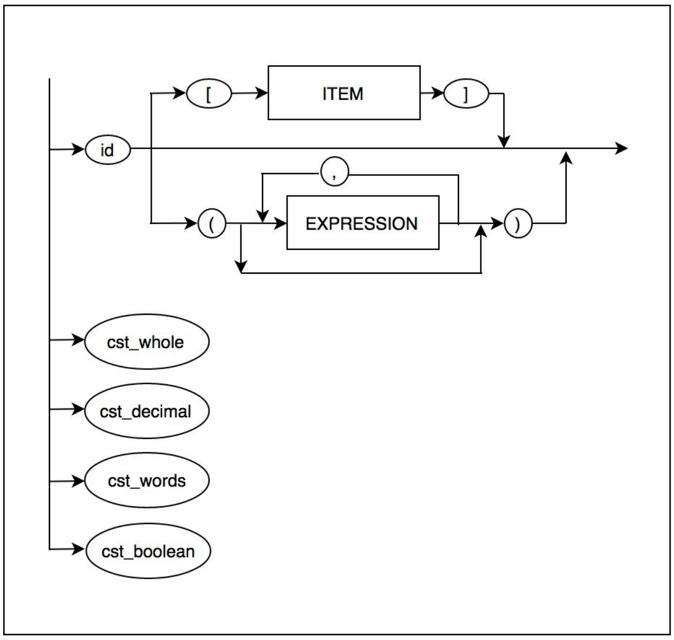
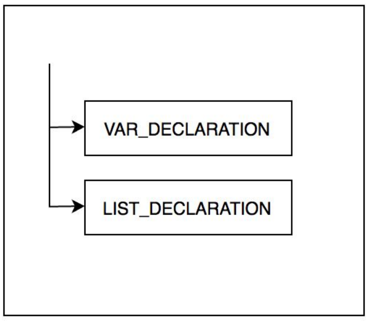
3.1 Tokens

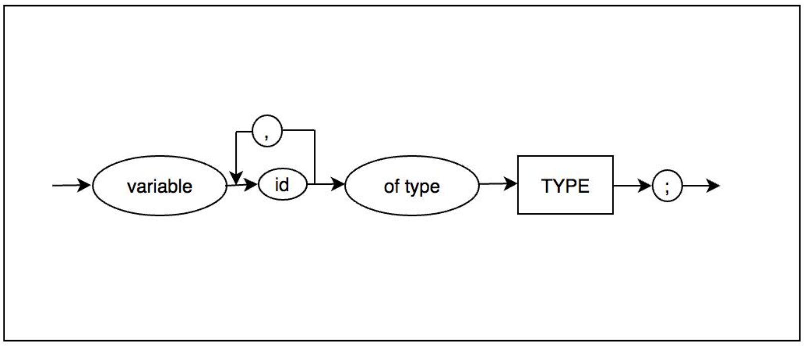
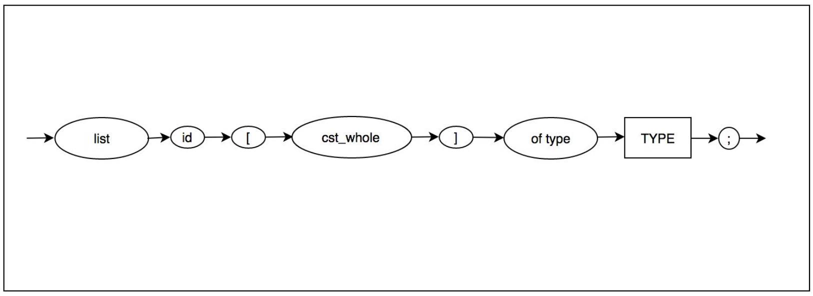
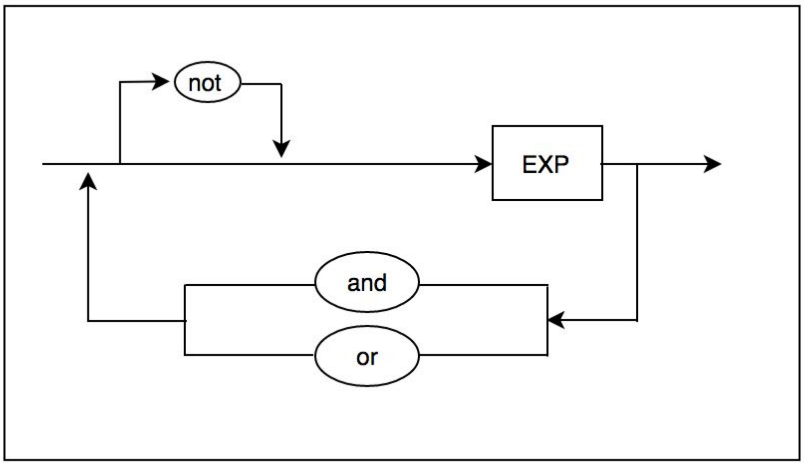
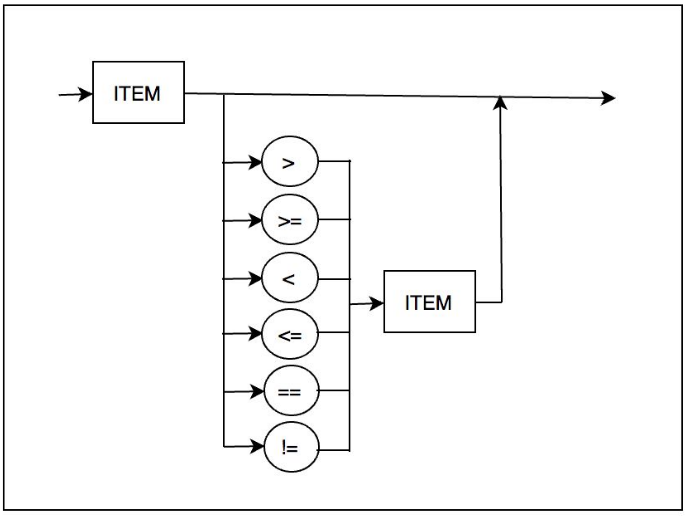
Token Name	**Regular Expression**
block	“block”
starting	“starting”
receives	“receives”
block_returns	“returns”
call	“call”
variable	“variable”
list	“list”
of	“of”
type	“type”
return_statement	“return”
do	“do”
until	“until”
if	“if”
else	“else”
colon	“:”
semicolon	“;”
comma	“,”
input	“input”
print	“print”
curlybraces_open	“{“
curlybraces_close	“}”
parenthesis_open	“(“
parenthesis_close	“)”

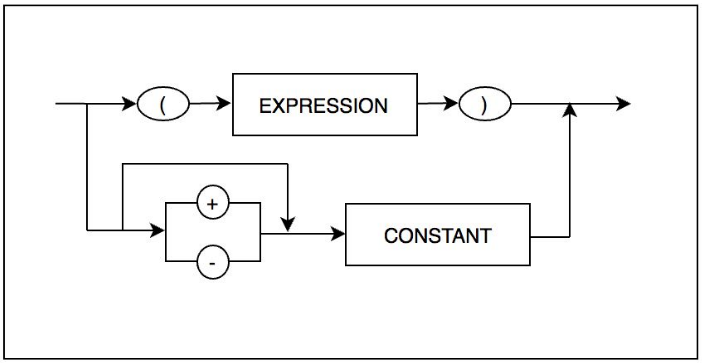
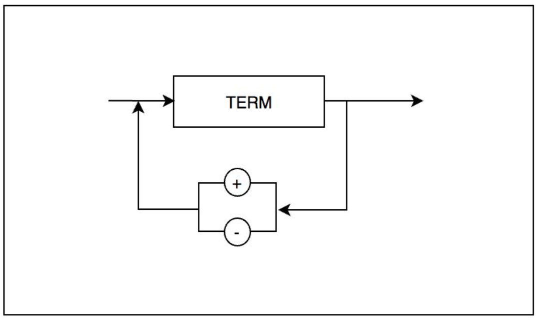
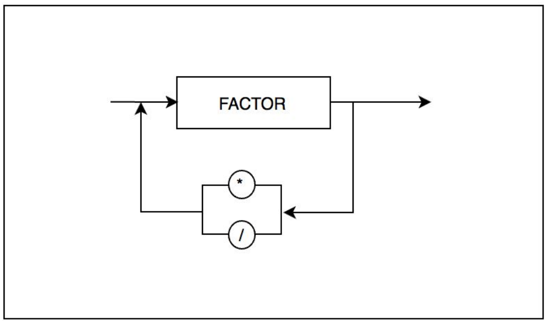
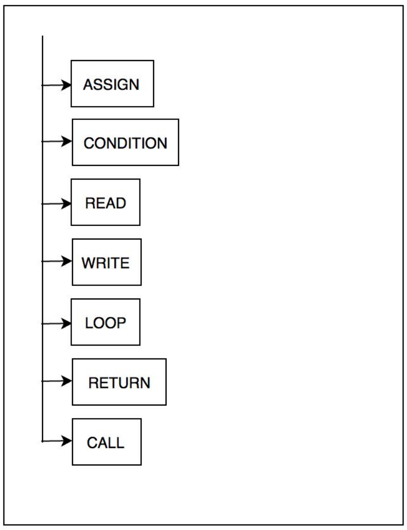
squarebracket_open	“[“
squarebracket_close	“]”
op_assign	“=”
op_less	“<”
op_less_equal	“<=”
op_greater	“>”
op_greater_equal	“>=”
op_equal	“==”
op_not_equal	“!=”
op_and	“and”
op_or	“or”
op_negation	“not”
op_addition	“+”
op_subtraction	“-”
op_multiplication	“*”
op_division	“/”
whole	“whole”
decimal	“decimal”
words	“words”
boolean	“boolean”
cst_whole	[0-9]+
cst_decimal	[0-9]+\.[0-9]+([Ee][\+-]?[0-9]+)?
cst_words	\”[^”]\”
cst_boolean	“true” | “false”
id	[A-Za-z]([A-Za-z0-9])*

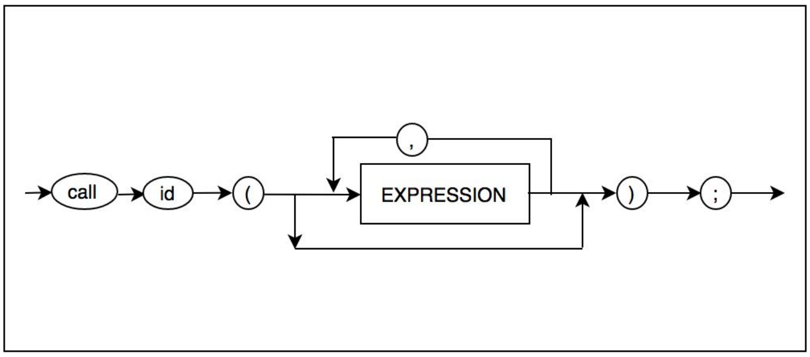
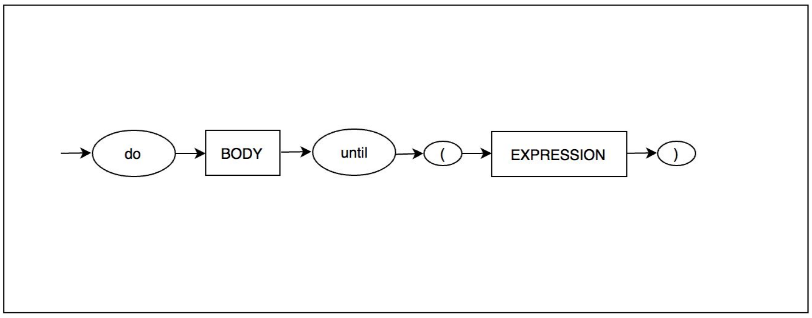
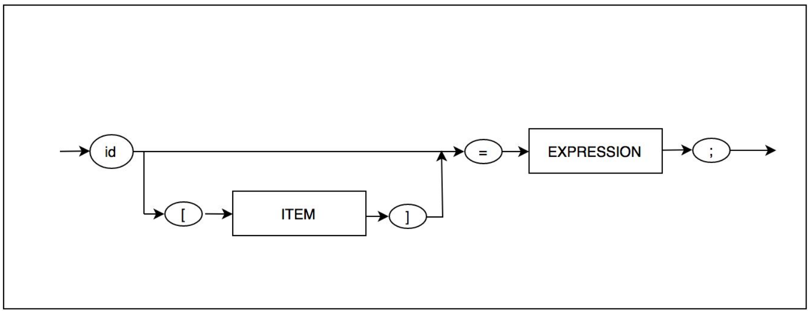
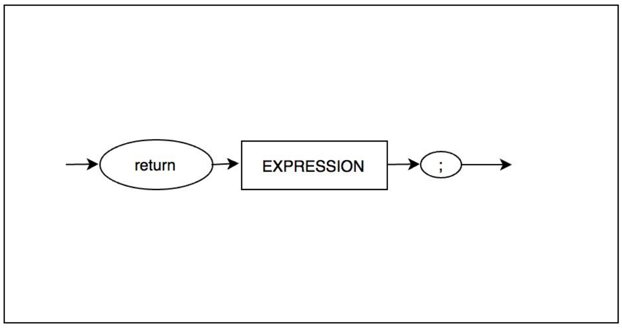
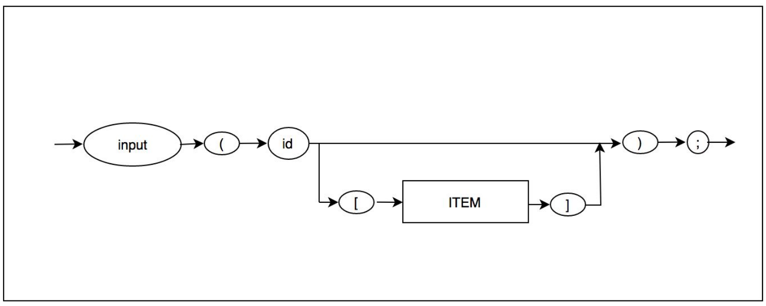
3.2 Syntax Diagrams

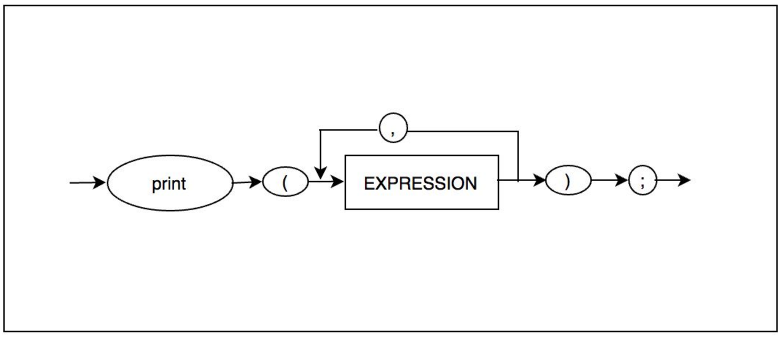
Syntactic Rule	**Syntax Diagram**
PROGRAM	
BLOCK	
BLOCK_BODY	
TYPE	
BODY	

<table>
<tr><td>CONDITION</td><td>
</td></tr>
<tr><td>CONSTANT</td><td>
</td></tr>
<tr><td>DECLARATIONS</td><td>
</td></tr>
</table>

VAR_DECLARATION	
LIST_DECLARATION	
EXPRESSION	
EXP	

FACTOR	
ITEM	
TERM	
STATEMENT	

<table>
<tr><td>CALL</td><td>
</td></tr>
<tr><td>LOOP</td><td>
</td></tr>
<tr><td>ASSIGN</td><td>
</td></tr>
<tr><td>RETURN</td><td>
</td></tr>
<tr><td>READ</td><td>
</td></tr>
</table>

WRITE	

3.3 Context Free Grammar

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** | cst_whole | 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
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** **| FACTOR_AUX**
FACTOR_AUX	**op_addition CONSTANT** **| op_subtraction CONSTANT** **| 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

3.3 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>result of type whole number: operation is mathematically evaluated</td></tr>
<tr><td>whole</td><td>+ | - | * | /</td><td>decimal</td><td>result of type decimal number: operation is mathematically evaluated</td></tr>
<tr><td>decimal</td><td>+ | - | * | /</td><td>decimal</td><td>result of type decimal number: operation is mathematically evaluated</td></tr>
<tr><td>words</td><td>+</td><td>words</td><td>result of type words: Operand 2 is added to Operand 1 (string concatenation)</td></tr>
<tr><td>whole</td><td>< | <= | > | >= | == | !=</td><td>whole</td><td>result of type boolean: evaluates the relational operation</td></tr>
<tr><td>whole</td><td>< | <= | > | >= | == | !=</td><td>decimal</td><td>result of type boolean: evaluates the relational operation</td></tr>
<tr><td>decimal</td><td>< | <= | > | >= | == | !=</td><td>decimal</td><td>result of type boolean: evaluates the relational operation</td></tr>
<tr><td>words</td><td>== | !=</td><td>words</td><td>result of type boolean: evaluates the relational operation</td></tr>
<tr><td>boolean</td><td>and</td><td>boolean</td><td>result of type boolean: evaluates the logical operation</td></tr>
<tr><td>boolean</td><td>or</td><td>boolean</td><td>result of type boolean: evaluates the logical operation</td></tr>
<tr><td>-</td><td>not</td><td>boolean</td><td>result of type boolean: evaluates the unary operation</td></tr>
</table>

3.4 Special Functions

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 children. For example:

Conventional Name	**Choo Choo Train Name**
Function/Method	block
var	variable
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 little children.

3.5 Primitive Data Types

There are 4 primitive data types in our language. We wanted to make it simple for children 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 children 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 children, 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.

Language and Computers

The project will be developed using Python, Google's Blockly Library, and using two Macbook Pros.

Bibliography

- Programiz. "Python Variables and Data types". *Programiz*. Web. <https://www.programiz.com/python-programming/variables-datatypes>

- Esthier T., Guyot J. "The BNF Web Club Language". Web. 1998. <http://cuiwww.unige.ch/db-research/Enseignement/analyseinfo/BNFweb.html>

- MIT. "Scratch - Imagine, Program, Share." *Scratch - Imagine, Program, Share*. <https://scratch.mit.edu/>

- Google for Education. "Blockly". *Google.* Web. <https://developers.google.com/blockly/>