MobProl

Ángel Ulises Yahaire Salazar Mireles A01136467
Luis Carlos Flores Gallardo A01196081

Ing. Elda Quiroga
Diseño de Compiladores

Miércoles 2 de mayo de 2018

Table of Contents

PROJECT DESCRIPTION ... ***3***

Purpose ... **3**

Objective and Project Scope ... **3**

Requirements Analysis ... **3**

Project Process ... **4**

LANGUAGE DESCRIPTION ... ***6***

Language Name ... 6

Language Characteristics ... **6**

List of Possible Errors in Compilation and Execution ... **7**

COMPILER DESCRIPTION ... ***7***

Working Tools ... **7**

Lexical Analysis ... **7**

Construction Patterns ... 8

Tokens ... 8

Syntax Analysis ... **9**

Semantic Analysis ... **10**

Syntax Diagrams ... 12

Brief Explanation ... **Error! Bookmark not defined.**

Table of Semantic Consideration ... 18

Memory Management ... **19**

VIRTUAL MACHINE DESCRIPTION ... ***20***

Working Tools ... **20**

Memory Description ... **20**

LANGUAGE TESTS ... ***21***

Test 1: ... **21**

Test 3 ... **23**

... **24**

PROJECT DESCRIPTION

Purpose

The purpose of this project is to understand the different phases of how to build a compiler by developing our own programming language and compiler to test it. By the end of the development, we expect to have a functional compiler that is capable of interpreting the new language we created and do the various tasks that we told it to do in a mobile device environment.

Objective and Project Scope

The objective of the compiler is to run our own programming language that can help people interested in programming to start in an easy way. The main aspect of this compiler is its capability to be executed in a mobile phone.

The main objective of the language is to create a simple, imperative language that will be used by people with no programming experience. We will create a new programming language, a compiler, and an IDE in order to reach our goal. All of this will be contained within a mobile application running for mobile devices. The language we are creating is in the category of imperative languages, meaning that we will give instructions to our compiler in order to do some tasks.

The objectives of MobProl are the following:

1. Create the new programming language called “MobProl”, used for the specific use of our compiler. This include specifying the alphabet of the language, as well as the syntax rules that will define the way commands are read.
2. A semantic aspect to the language in order to keep a record of variable and instructions that need to be done in order for the language to do its task.
3. A virtual machine will be developed in order to take all the previous aspects of the compiler and execute the instructions it’s given and return the result to the user.

Requirements Analysis

In the requirements for the project, the main requirements were the following:

- Create a functional programming language capable of doing arithmetical computation.
- The programming language should be capable of handling conditions and loops.
- The compiler should run in a mobile device or mobile device emulator.
- The programming language should follow an iterative approach.

Project Process

1ro y 2do de mayo Se desarrolló la mayoría de semántica así como la documentación. Se implementó la memoria y la interfaz de celular, así como la manera para comunicar Python con el app. Se descubrió que había un sinfín de errores en la semántica debido a la falta de pruebas en las etapas anteriores.	Added print and more goto's support. Fixed expression support Added suport for expressions and goto's, also fixed small error i... 3 hours ago First quadruples in virtual machine Fix bug with no global variables. Merge branch 'master' of https://github.com/lucfg/compilador Modificaciones a functionCall. Updated interface and accepted quadruples; updated gitignore Modificaciones a functionCall. Cambios de sintaxis a parser. Cuadruplos para incrementos y decrementos. Modificaciones al archivo padre mobProl para la comunicación con la maqui... Changed folder for compiler and made connection base for mobile and serv... Funcionalidad de ciclos y condicionales en semantica. Funcionalidad de print. 12 hours ago Correcciones para aceptación de variable globales y diferenciado de constan... Se arreglaron probemas con la creación de cuádruplos de expresiones.
26 de abril Se agregaron cuádruplos a semántica y realizaron unas pequeñas	Cambios a semantica 6 days ago no message Merge commit '940579dc7b084807c1042b62559974c90fe05ff2' Se quito la declaración de arrays para incluirlo en variables Changes to test semantics

correcciones en el parser.	
18 de abril Se agregó una máquina virtual provisional que se espera se pudiera usar y se realizaron pequeñas correcciones en el parser y semantics.	Se agrego archivo para la maquina virtual 2 weeks ago Merge remote-tracking branch 'origin/master' Corrección de read Correcciones generales Merge remote-tracking branch 'origin/master' Agregué read a semantics Pequeñas correcciones de nombre d variables Correcciones de indentación
10 de abril Se agregaron puntos neurálgicos en parser para que fueran llamados en la semántica.	Se agregaron funciones para la parte de condiciones y funciones. 3 weeks ago Se termino de implementar las reglas neurálgicas para la sintaxis del compil... Modificaciones a los puntos neurálgicos
3-23 de marzo Se comenzó con el proyecto, desarrollando diagramas iniciales, así como los programas de parser.py y lexer.py.	Cambios a reglas de ciclos y lectura y escritura a month ago Modificaciones a la regla de functionCall. Se modifico la gramática para corregir errores en la misma. Archivo sustituto de symbolTable.py. Es una clase más clara de la tabla de va... Se agrego para su futura implementación como el encargado de la semántic... Se reviso los puntos neurálgicos de las reglas. Aun falta revisar varias de ellas. Added semantic cube (first test) no message Se amplio el parser para poder generar una tabla de símbolos. Se corrigieron errores respecto a los tokens.

Me di cuenta que, aunque estuvimos trabajando un poco cada semana, no laboramos con la debida dedicación al proyecto. De haberle sido más rigurosos con las pruebas probablemente se hubiera obtenido un mejor resultado al final. No es la primera vez que trabajo con un compilador, y me sigue pareciendo que es algo muy retador, pero ahora comprendo mucho más acerca de su desarrollo y función. Me parece poco probable utilizar los conceptos que aprendí en la clase para futuros desarrollos de lenguajes, pero sí creo que me ayudará a considerar cómo escribo mis programas para que puedan ser más eficientes.

____

Ángel Ulises Yahaire Salazar Mireles

With the semester coming to an end, I'm happy to say that I learned a lot with this project. Even if is not a complete product, all the knowledge I gained will be irreplaceable and will definitely shape my future. I regret that I didn't ask for much help before, but the project definitely help me know that some things are better to do them with someone. We could definitely work harder in the project. Because of our different schedules caused by other classes or jobs, we didn't spend too much time working until the last weeks.

____

Luis Carlos Flores Gallardo

LANGUAGE DESCRIPTION

Language Name

The name of the new language is "**MobProl**", which stands for **Mob**ile **Pro**gramming **L**anguage.

Language Characteristics

The main characteristics of the language include a basic declaration of variables, which include atomic variables and vectors of one or two dimensions. The language accepts integers, decimals, and boolean data types. The language also supports simple cycles and conditionals, as well as the declaration and use of functions. The language also the capability of performing simple arithmetic procedures, in conjunction with the other characteristics said before. The main characteristic relies in the ability of the language to be executed in a computer or in a mobile device.

List of Possible Errors in Compilation and Execution

- **The function needs to return something of type 'XX'.**
    - Occurs when the function tries to return something of a different type from the function type.
- **Condition must be bool type.**
    - Occurs whenever the result of a loop or condition is of a type different from a boolean type.
- **Type 'XX' not supported.**
    - Occurs when a different data type is given to a variable.
- **Cannot assign a value of different type to the variable 'XX'.**
    - Occurs whenever the user tries to assign a variable of certain type to another of a different type.
- **Variable 'XX' has not been declared. Cannot assign value.**
    - Occurs when the user tries to use a variable that has not been declared first.

COMPILER DESCRIPTION

Working Tools

The compiler was designed in both a Mac and Windows machine.

The compiler runs entirely on Python3, with the help of the PLY tool to create our tokens and shape our syntax.

The Python IDLE, as well as the command line were used for testing purposes.

Lexical Analysis

The lexical analysis was done entirely with Python using the help of Lex and Yacc for Python (PLY).

Construction Patterns

We'll be showing some of the most important tokens used for lexical analysis:

Tokens	Regular Expression
ID	[a-zA-Z_][a-zA-Z_0-9]*
Decimal	[0-9]*\.[0-9]+
Integer	\d+
Plus	\+
Times	*
Assign	=

The rest of the tokens for a similar structure, with the last ones being the most common in the creation of tokens.

The following are some reserved words, used to avoid accidental declaration of words which could contain the reserved tokens.

Tokens	Regular Expression
PROGRAM	'program'
MAIN	'main'
RETURN	'return'
VAR	'var'

The rest of the reserved words have the same structure, and, as said before, are used to avoid the use of this words as a variable.

Tokens

The following are all the tokens used, with the exception of reserved words, for the MobProl programming language.

```
# Primitives
def t_ID(t):
  r'[a-zA-Z_][a-zA-Z_0-9]*'
  if t.value in reserved:
    t.type = reserved[t.value]
  return t
```

```
# Data

def t_DECIMAL(t):
  r'[0-9]*\.[0-9]+'
  t.value = float(t.value)
  return t
def t_INTEGER(t):
```

```
    r'\d+'
    t.value = int(t.value)
    return t
t_CHARACTER = r'[a-zA-Z_]'
t_ALPHANUMERIC = r'\"[a-zA-Z_0-9\s]*\"'
# Arithmetic operators
t_INCREMENT = r'\+\+'
t_DECREMENT = r'--'
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_MOD = r'%'

#Boolean operators
t_EQUAL = r'\=='
t_NOT_EQUAL = r'\!='
t_MORE_EQUAL = r'\>='
t_LESS_EQUAL = r'\<='
t_MORE_THAN = r'\>'
t_LESS_THAN = r'\<'
t_AND = r'&&'
t_OR = r'\|\|'

#Block delimitators
t_L_KEY = r'\['
t_R_KEY = r'\]'
t_L_BRACK = r'\{'
t_R_BRACK = r'\}'
t_L_PAR = r'\('
t_R_PAR = r'\)'

#Others
t_ASSIGN = r'='
t_COMMA = r'\,'
t_DOT_COMMA = r';'
t_COMMENT_LINE = r'\#.*'
t_END_LINE = r'\n'
```

Syntax Analysis

The syntax analysis of the compiler was done using Python and PLY as well. The following is the way the syntax was written for its future use for the semantics of the language. We wrote some parameters of the rules in capital letters to differentiate the tokens from other rules in the syntax.

PROGRAM
program: PROGRAM ID L_KEY variables functions mainBody R_KEY

MAIN BODY
mainBody: MAIN L_PAR R_PAR L_KEY variables statements R_KEY

BODY
body: L_KEY statements R_KEY

VARIABLES (Declaration)
```
variables:
        | VAR type ID DOT_COMMA variables
        | VAR type ID L_BRACK INTEGER R_BRACK DOT_COMMA
variables
        | VAR type ID L_BRACK INTEGER R_BRACK L_BRACK INTEGER
        R_BRACK DOT_COMMA variables
```

FUNCTIONS (Declaration)
```
functions:
        | FUNCTION type ID L_PAR functionsHelp R_PAR L_KEY
variables statements R_KEY

functionsHelp:
        | type ID
        | type ID COMMA functionsHelp2

functionsHelp2: type ID
        | type ID COMMA functionsHelp2
```

TYPES
```
type: INT
        | DECIM
        | BOOL
        | CHAR
        | STRING
        | VOID
```

STATEMENTS
```
statements:
        | statement statements

statement:
        | assignment DOT_COMMA
        | functionCall DOT_COMMA
        | ifBlock
        | whileBlock
        | print DOT_COMMA
        | read DOT_COMMA
        | lineComment
        | return DOT_COMMA
```

RETURN
Return: RETURN megaExp

ASSIGNMENT
```
Assignment: idCall ASSIGN megaExp
        | idCall ASSIGN functionCall
        | assignIncr
        | assignDecr
```

++/--
assignIncr: idCall INCREMENT

assignDecr: idCall DECREMENT

FUNCTION (Calling)
functionCall: ID L_PAR functionCallParams R_PAR

```
functionCallParams:
        | functionCallParamsOptional

functionCallParamsOptional: megaExp COMMA
functionCallParamsOptional
        | megaExp
```

IF
ifBlock: IF L_PAR megaExp R_PAR body optionalElse

optionalElse:
| ELSE body

WHILE
whileBlock: WHILE L_PAR megaExp R_PAR body

MEGA EXPRESSIONS
megaExp: superExp
| superExp AND superExp
| superExp OR superExp

SUPER EXPRESSIONS
superExp: exp
| exp MORE_THAN exp
| exp LESS_THAN exp
| exp MORE_EQUAL exp
| exp LESS_EQUAL exp
| exp EQUAL exp
| exp NOT_EQUAL exp

EXPRESSIONS
exp: term
| term PLUS exp
| term MINUS exp

TERMS
term: factor
| factor TIMES term
| factor DIVIDE term
| factor MOD term

FACTOR
factor: INTEGER
| DECIMAL
| ALPHANUMERIC
| CHARACTER
| BOOLEAN
| VOID
| idCall
| L_PAR megaExp R_PAR
| functionCall

VARIABLES (Calling)
idCall: ID
| ID L_BRACK exp R_BRACK
| ID L_BRACK exp R_BRACK L_BRACK exp R_BRACK

PRINT
print: PRINT L_PAR print_help R_PAR

print_help:
| ALPHANUMERIC
| idCall
| functionCall
| megaExp

READ
read: READ L_PAR idCall R_PAR

Semantic Analysis

For the semantic analysis, we made use of three dictionaries for the management of virtual memory, as shown below:

globalTable = VarTable(0, 50001, 10001, 15001)
localTable = VarTable(20001, 25001, 30001, 35001)
auxTable = VarTable(40001, 45001, 50001, 55001)

According to the tables, the first parameter indicates the space in memory of variables of type 'int'; the second parameter indicates the space in memory for 'decim' variables; and the third parameter indicates the space for 'bool' variables. The fourth parameter in the dictionaries are for space of void variables, that work as extra storage if its ever needed.

Moving into the code of the semantics, the way the semantics of the syntax was implemented consisted on building an object named FuncNode that could store a big Node of type Program, which would store a list of other Nodes containing certain rule from the syntax. The parser's duty was to pass the FuncNode from an specific rule to another FuncNode, reaching at the end the FuncNode of program. The following is an example of how the parser sends a Node to the semantics.

p[0] = FuncNode('program', p[2], p[4], p[5], p[6])

The p[0] represents the resulting FuncNode of the rule, in this case, the program rule. The first parameter of the Node represents its type, which is the name of the rule in most of the cases. The rest are the arguments, that, in most cases, contain other FuncNodes from other rules.

```
class FuncNode(object):
  def __init__(self, t, *args):
    self.type = t
    self.args = args
```

The code above represents the class and its attributes, which were explained in the paragraph above.

Two of the main methods of the class are semantic and expression, which both have all the rule types. Both methods navigate themselves through if's and else's, until the condition of program is finished, and the quadruples are formed.

```
if self.type == "program":
     quadruples.append(gotoMain)

     for elem in self.args:
       if elem is not None:
         if isinstance(elem, str):
           print("Processing program " + elem + ".")
         else:
           elem.semantic(funcName, result)
     quadruples.append(["END","","",""])

# ------------------------------------------------------------

   elif self.type == "main":
     quadruples.append(["main","","",""])
     gotoMain[3] = len(quadruples)
     funcName = self.type
     currentTable.add(funcName, "int", "main")
     localTable.add(funcName, "funcType", self.args[0])

     for elem in self.args[1:]:
       if elem is not None:
         elem.semantic(funcName, result)

     quadruples.append(["ret","*0*","",""])
# ------------------------------------------------------------

   elif self.type == "function":
     funcName = self.args[0]
     currentTable.add(funcName, self.args[1],self.args[0])
     localTable.add(funcName, "funcType", self.args[0])
     auxFuncType = ["func", funcName, self.args[1],""]
     quadruples.append(auxFuncType)

#    auxReturn = ""

     for elem in self.args[2:]:
       if elem is not None:
         elem.semantic(funcName, result)

     quadruples.append(["ENDPROC","","",""])

# ------------------------------------------------------------

   elif self.type == "return":
      resultType, address = self.args[0].expression(funcName,
result)
      print("El result de RETURN" + str(resultType))

      print(str(globalTable[funcName].keys()))
      if resultType in globalTable[funcName].keys():
         quadruples.append(["ret", address,"",""])
      else:
         raise Exception("The function " + funcName + " needs to
return something of type " + resultType + ".")
```

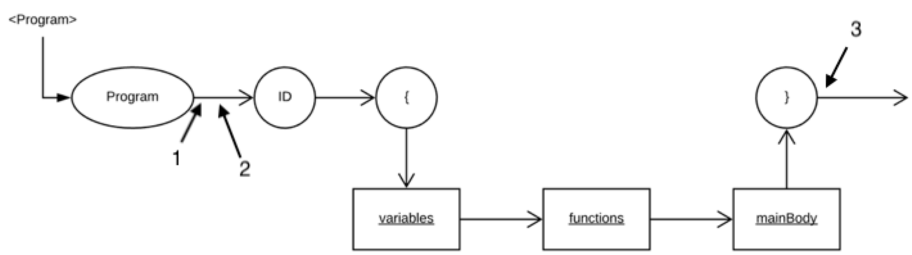
In the code above we can see some examples of how both the semantic and expression methods worked.

```
def getType(v, funcName="missingFuncName",
currentTable="error"):
   print("Getting type of " + str(v))
   if isinstance(v, str):
      if v == "true" or v == "false":
         return "bool", "value"
      elif v[0] == "\"":
         return "string", "value"
      else:
         found = False
         for key in currentTable[funcName]:
         #verifies that the variable has been declared
            if v in currentTable[funcName][key].keys():
               found = True
               return key, False
               break
```

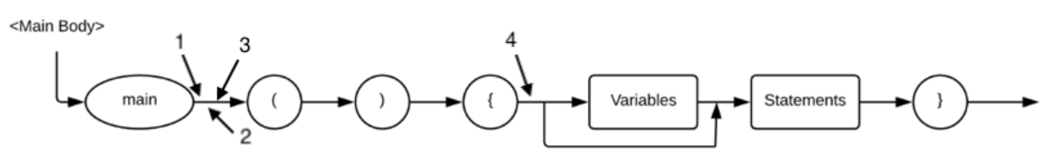
```
        for key in globalTable["global"]:
        #verifies that the variable has been declared
            if not found and (v in globalTable["global"][key].keys()):
                found = True
                return key, True
                break

    if not found:
            raise Exception("Variable '" + str(v) + "' has not been declared. Cannot assign value.")

        elif isinstance(v, int):
            return "int", "value"
        elif isinstance(v, float):
            return "decim", "value"
        else:
            return "void", "value"
```

This code shows one of the most important methods outside the FuncNode class. This method gets the data type of certain value, and it searches for the variable or constant in both the local and global tables.

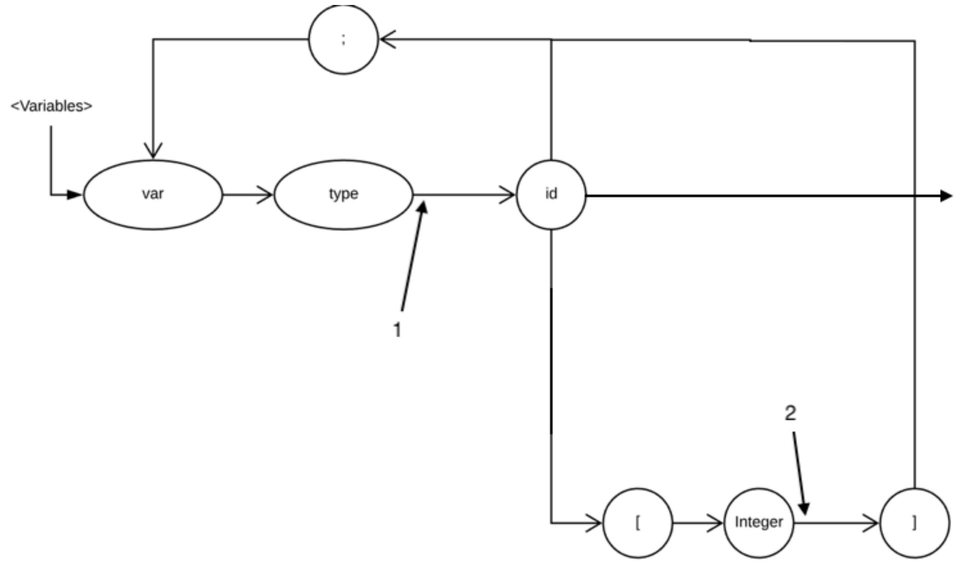
Syntax Diagrams



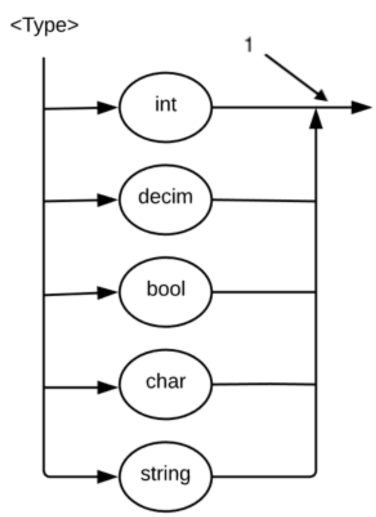
1. A "goto" quadruple is created to make a jump to the function main.
2. The semantic reads the rest of the arguments and calls the semantic of the next rule.
3. A "END" quadruple is created to tell the virtual machine to finish the execution of the program.



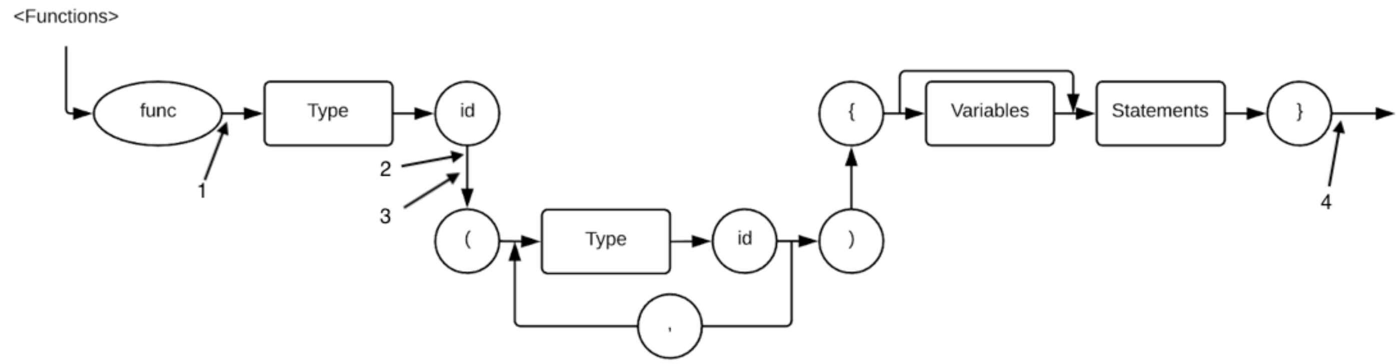
1. A "main" quadruple is created as a marker for the virtual machine.
2. The "goto" quadruples from the program rule gets assigned the current place of the "main" quadruple.
3. A new variable is added to the global table for the function use as a possible variable.
4. The semantic reads the arguments from the object and calls the semantic of the next rule.



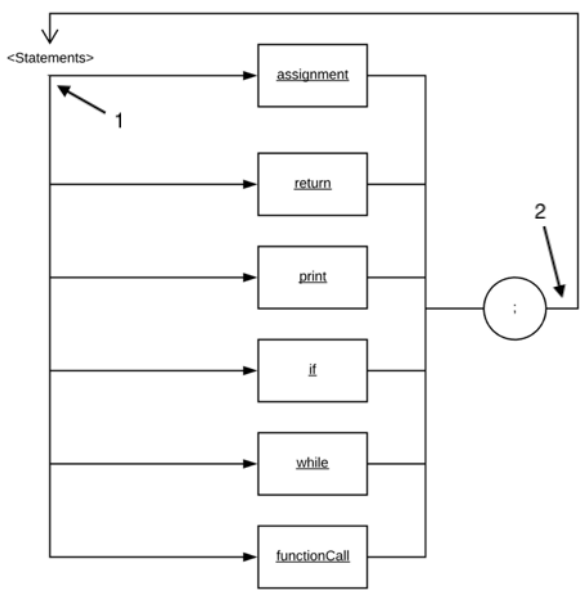
1. Adds the variable to the current table and function in use and calls the semantic method if there are more variables to be declared.
2. If an array, the semantic does the calculations necessary to get the actual size of the variable and adds it to the current table and function in use.



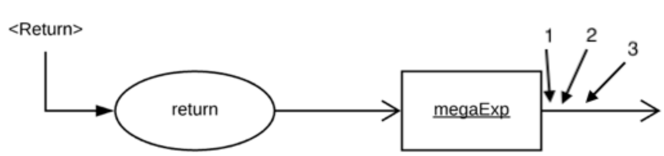
1. Returns the data type to the FuncNode that requested it.



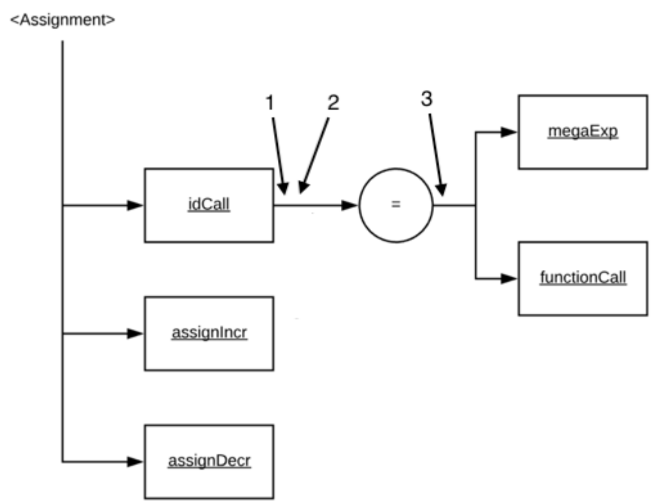
1. Updates the name of the function in use.
2. Adds the function name and type as a variable in the global and local tables.
3. The semantic reads the arguments and calls the semantics of the following rules.
4. Creates “ENDPROC” quadruples to indicate the end of the function.



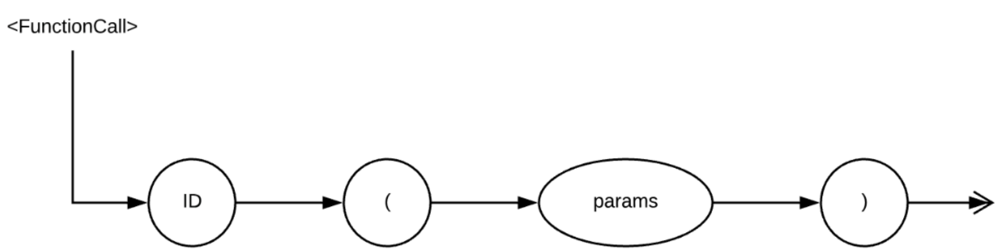
1. The semantic checks the next rule to execute and calls either the “expression” or “semantic” method.
2. If more statements, semantic calls the “semantic” method.

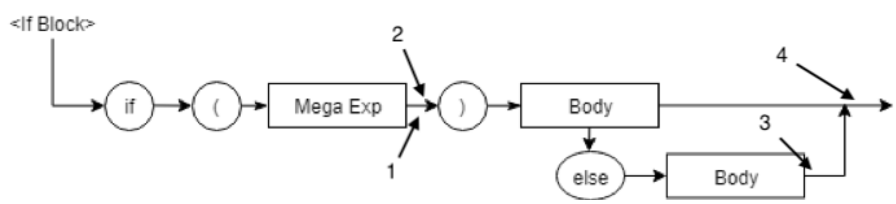


1. Solves the megaExp of the return.
2. Searches for the function type and compares it with the expression type.
3. Creates quadruple to return the value to the function variable address.

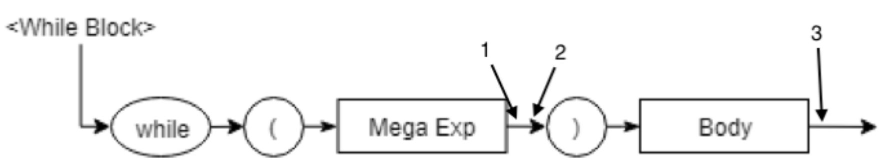


1. Check if the variable is a normal variable, an array or a matrix.
2. Search for the variable in the local and global table.
3. Assign value to the variable.

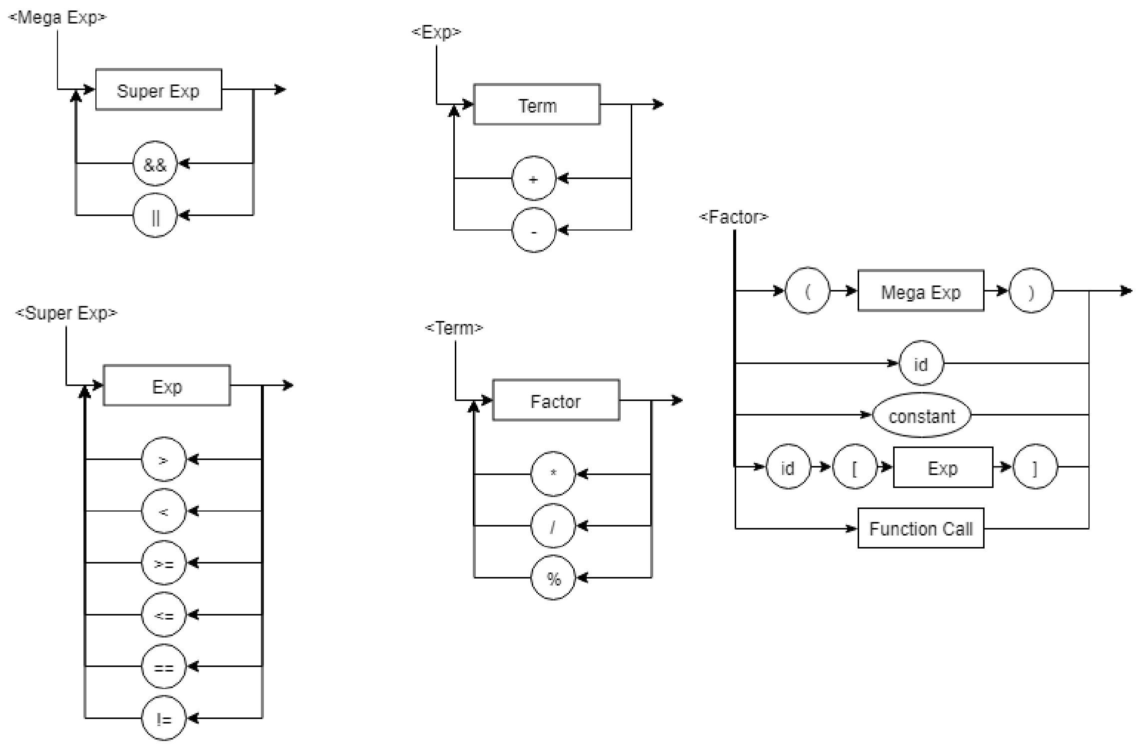


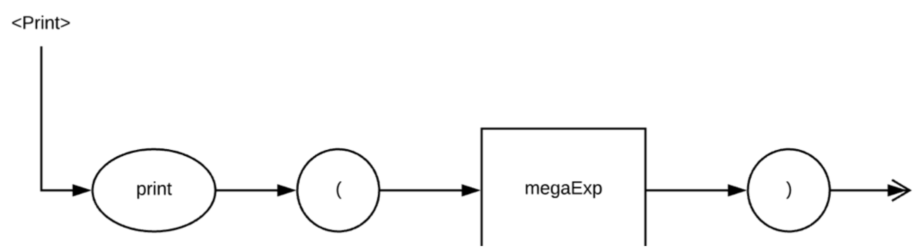


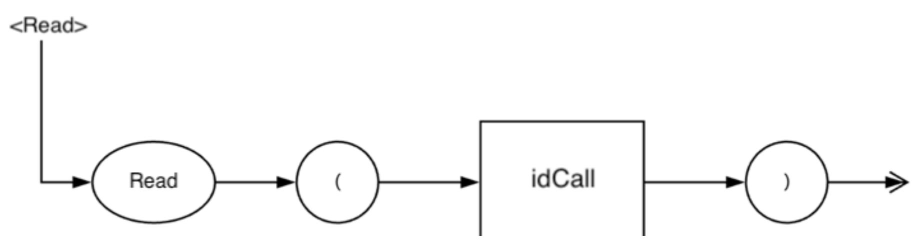
1. Solves the expression and return the type and address of the variable.
2. Creates “gotoF” quadruples to jump outside the if.
3. If an else exists, solves the expression inside the else.
4. Creates “goto” quadruple to jump outside the “if” if an else exists.



1. Solve the expression inside the condition of the while.
2. Create a “gotoF” quadruple to jump in case the condition is wrong.
3. Create a “goto” quadruple to jump to the beginning of the while.







Now we'll explain the main components of the language, those that are the most relevant to the semantics of the language.

```
# Checks if the variable is in the current table
found = False
for key in currentTable[funcName]:
#verifies that the variable has been declared
    if v in currentTable[funcName][key].keys():
        found = True
        return key, False
        break
```

This code looks for a key, which is a data type, inside the current Table, which can be the global table to look for global variables, or in the local table, to look for a variable in a function.

```
# isPrimitive
# This method checks that the input is not a variable

# Receives: numeric value, true, false, a string
# or the name of a variable
# Returns: True or False

# Commonly used in methods who may use both variables
# and numbers or other data types.
def isPrimitive(t):
    if isinstance(t, str):
        if t == "true" or t == "false":
            return True
        elif t == "void":
            return True
        elif t[0] == "\"":
            return True
    elif isinstance(t, int):
        return True
    elif isinstance(t, float):
        return True
    return False
```

The function isPrimitive checks if the parameter is a primitive data type, meaning that it will only return True if the parameter is not a variable, but a constant.

```
# Program
# This condition recieves a FuncNode object with type 'program'
# Comunicates only with the method 'semantic'
if self.type == "program":

  # Creates a 'goto' quadruple to make a jump to the main function
  quadruples.append(gotoMain)

  # For each element of the FuncNode, it calls the semantic method
  # for the element
  for elem in self.args:
    if elem is not None:

      # Checks the name of the program
      if isinstance(elem, str):
        print("Processing program " + elem + ".")
      else:
        elem.semantic(funcName, result)
  # Creates a 'END' quadruple to tell the virtual machine to stop the
  # execution
  quadruples.append(["END","","",""])
```

This if represents the majority of the semantic and expression methods. The condition grabs all the arguments from a FuncNode and calls semantic or expression again depending on the rule you are using. If doing a calculation, expression is called; anything else will call the semantic method.

Table of Semantic Consideration

For the language, static semantics will be used. The compiler will evaluate cases such as undeclared id’s, wrong number or types of parameters in an operation, incompatible types, etc. For now, cases that involve runtime checks will be omitted.

Other characteristics of the semantics are:

- Hierarchy: The hierarchy of the language is described with the following table.

Level (Higher is left for last)	Operators
1	=
2	&&, ||
3	>, <, >=, <=, ==, !=
4	+, -
5	*, /, %
6	()

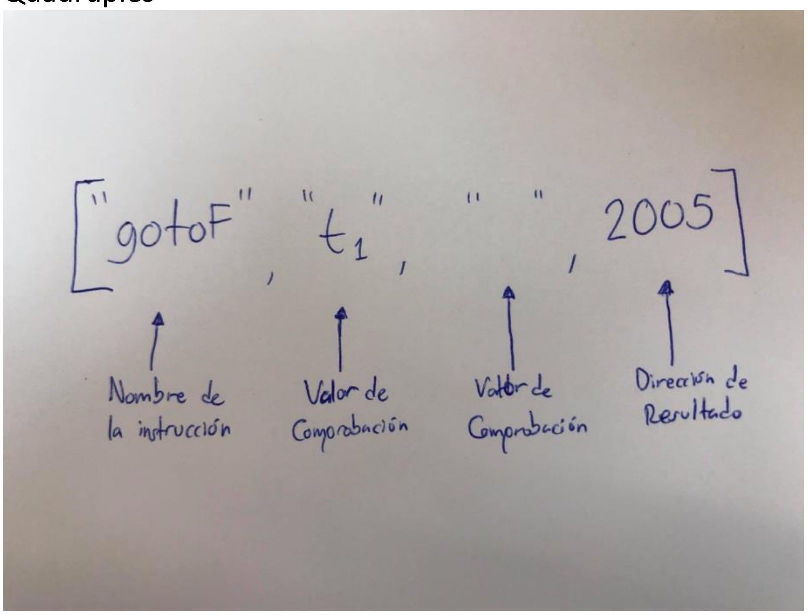
- Associativity: All expressions should follow left associativity, with the sole exception of assignment, which will use right association.
- Combination: The combination of types will be explained with the following table.

A	**B**	*	/	%	+	-	>	<	>=	<=	==	!=	&&	| |
int	int	dec	dec	dec	int	int	boo	boo	boo	boo	boo	boo	X	X
int	dec	dec	dec	dec	dec	dec	boo	boo	boo	boo	boo	boo	X	X
int	boo	X	X	X	X	X	X	X	X	X	X	X	X	X
int	cha	X	X	X	X	X	X	X	X	X	X	X	X	X
int	str	X	X	X	X	X	X	X	X	X	X	X	X	X
dec	dec	dec	dec	dec	dec	dec	boo	boo	boo	boo	boo	boo	X	X
dec	boo	X	X	X	X	X	X	X	X	X	X	X	X	X
dec	cha	X	X	X	X	X	X	X	X	X	X	X	X	X
dec	str	X	X	X	X	X	X	X	X	X	X	X	X	X
boo	boo	X	X	X	X	X	X	X	X	X	boo	boo	boo	boo

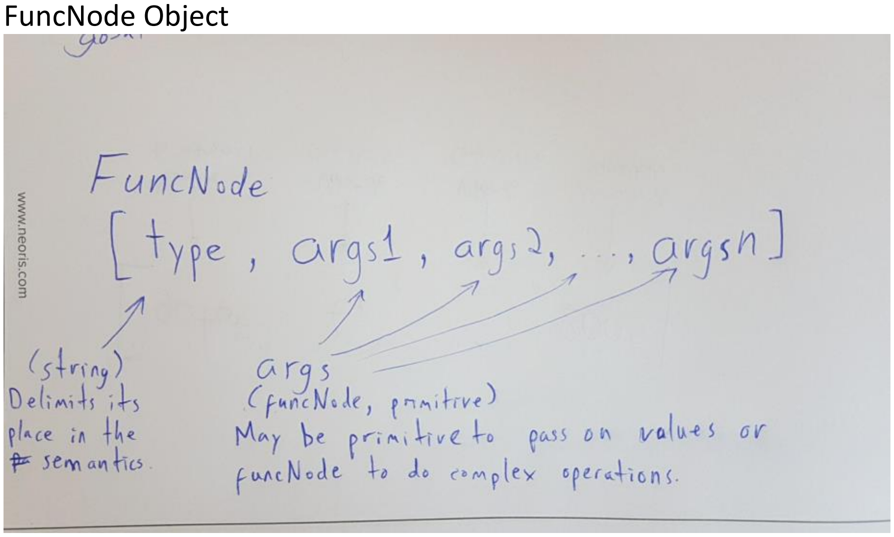
boo	cha	X	X	X	X	X	X	X	X	X	X	X	X	X
boo	str	X	X	X	X	X	X	X	X	X	X	X	X	X
cha	cha	X	X	X	str	X	X	X	X	X	boo	boo	X	X
cha	str	X	X	X	str	X	X	X	X	X	boo	boo	X	X
str	str	X	X	X	str	X	X	X	X	X	boo	boo	X	X

Memory Management

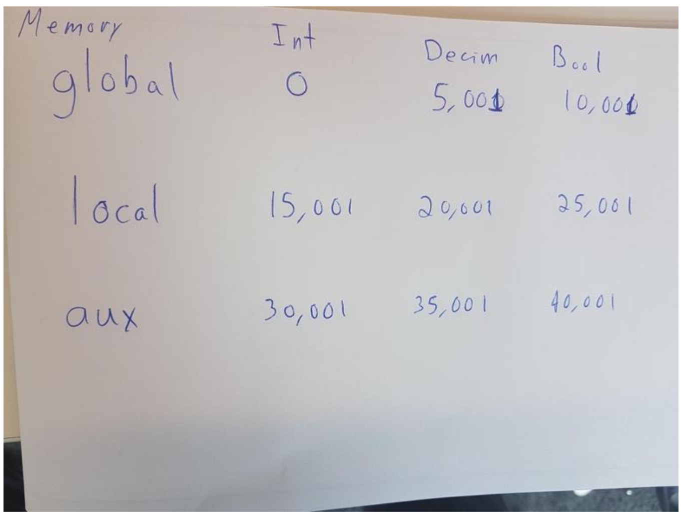
Quadruples



FuncNode Object



Variable tables



VIRTUAL MACHINE DESCRIPTION

Working Tools

The virtual machine works on the mobile device of the user. It is encased in an app developed with help of the Ionic framework, which consists mainly of angular, html, css, typescript and an adaptation of Bootstrap. The app reads code written by the user and sends it to a remote server to compile it with python. The server responds with a list of quadruples of the commands to execute.

Memory Description

The memory is handled as a vector of json objects. Each index of the vector represents a depth in the function, allowing for different contexts between many functions. When a variable is assigned, it is saved as a new json element to its respective depth and, when called, the variable is searched for in the current depth; if not initialized (noticed by the absence of the variable on the json object), the user is notified. Each level of depth in the vector is added when a function is about to be called and removed once it has returned, freeing memory on the go. There is no simple numeric limit to how many levels of depth can the vector reach, as it depends on the mobile device of the user.

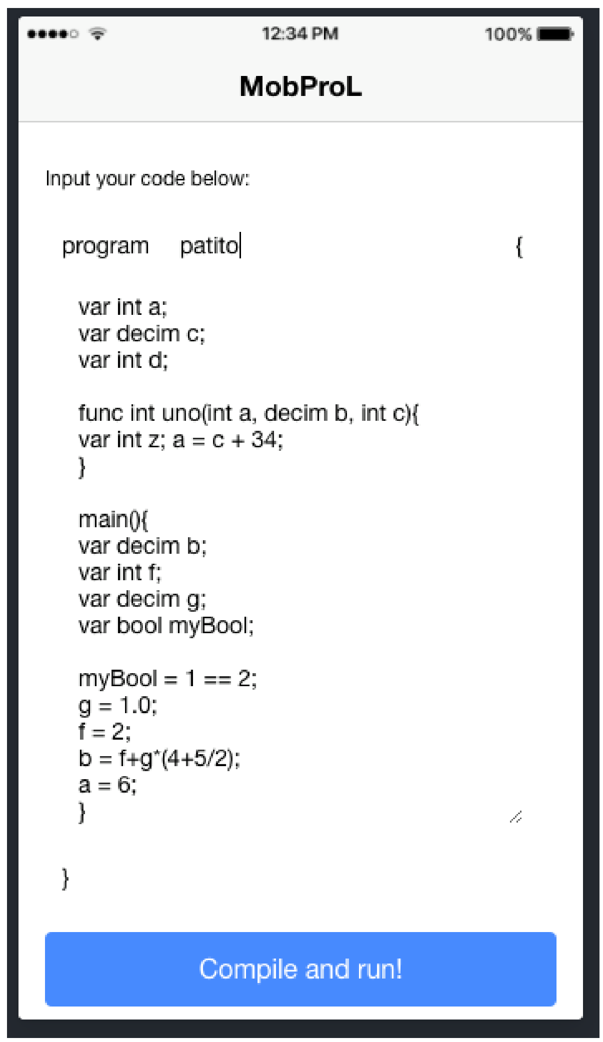
LANGUAGE TESTS

Test 1:

Code

```
var int a;
var decim c;
var int d;

func int uno(int a, decim b, int c){
var int z; a = c + 34;
}

main(){
var decim b;
var int f;
var decim g;
var bool myBool;

myBool = 1 == 2;
g = 1.0;
f = 2;
b = f+g*(4+5/2);
a = 6;
}
```

Code as seen on the app



Resulting quadruples

Resulting memory

```
▶0: (4) ["GOTO", "", "", "6"]
▶1: (4) ["func", "uno", "int", ""]
▶2: (4) ["+", "15002", "*34*", "30001"]
▶3: (4) ["=", "30001", "", "15001"]
▶4: (4) ["ENDPROC", "", "", ""]
▶5: (4) ["main", "", "", ""]
▶6: (4) ["==", "*1*", "*2*", "40001"]
▶7: (4) ["=", "40001", "", "25001"]
▶8: (4) ["=", "*1.0*", "", "20003"]
▶9: (4) ["=", "*2*", "", "15004"]
▶10: (4) ["/", "*5*", "*2*", "35001"]
▶11: (4) ["+", "*4*", "35001", "35002"]
▶12: (4) ["*", "20003", "35002", "35003"]
▶13: (4) ["+", "15004", "35003", "35004"]
▶14: (4) ["=", "35004", "", "20002"]
▶15: (4) ["=", "*6*", "", "0"]
▶16: (4) ["ret", "*0*", "", ""]
▶17: (4) ["END", "", "", ""]
```

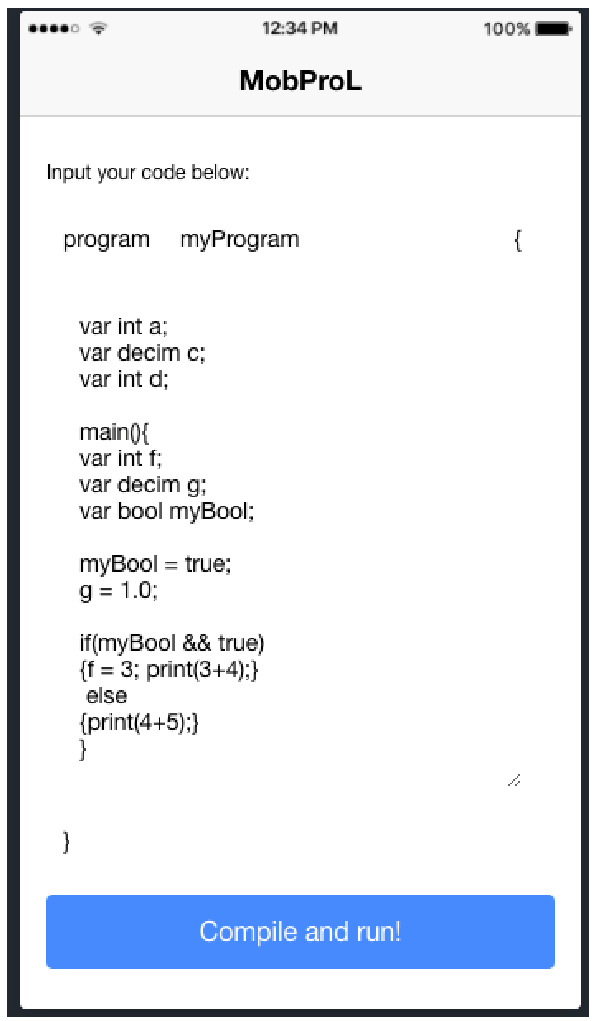
```
▶0: varTuple {value: 6}
▶15004: varTuple {value: 2}
▶20002: varTuple {value: 8.5}
▶20003: varTuple {value: 1}
▶25001: varTuple {value: false}
▶35001: varTuple {value: 2.5}
▶35002: varTuple {value: 6.5}
▶35003: varTuple {value: 6.5}
▶35004: varTuple {value: 8.5}
▶40001: varTuple {value: false}
```

Test 2:

Code

```
program patito {
var int a;
var decim c;
var int d;

main(){
var int f;
var decim g;
var bool myBool;

myBool = true;
g = 1.0;

if(myBool && true)
{f = 3; print(3+4);}
 else
{print(4+5);}
}
}
```

Code as seen on the app



Resulting quadruples

Resulting memory

```
0['GOTO', '', '', 2]
1['main', '', '', '']
2['=', 'true', '', 25001]
3['=', '*1.0*', '', 20001]
4['&&', 25001, 'true', 40001]
5['gotof', 40001, ' ', 10]
6['=', '*3*', '', 15001]
7['+', '*3*', '*4*', 30001]
8['print', 30001, '', '']
9['goto', ' ', ' ', 12]
10['+', '*4*', '*5*', 30002]
11['print', 30002, '', '']
12['ret', '*0*', '', '']
13['END', '', '', '']
```

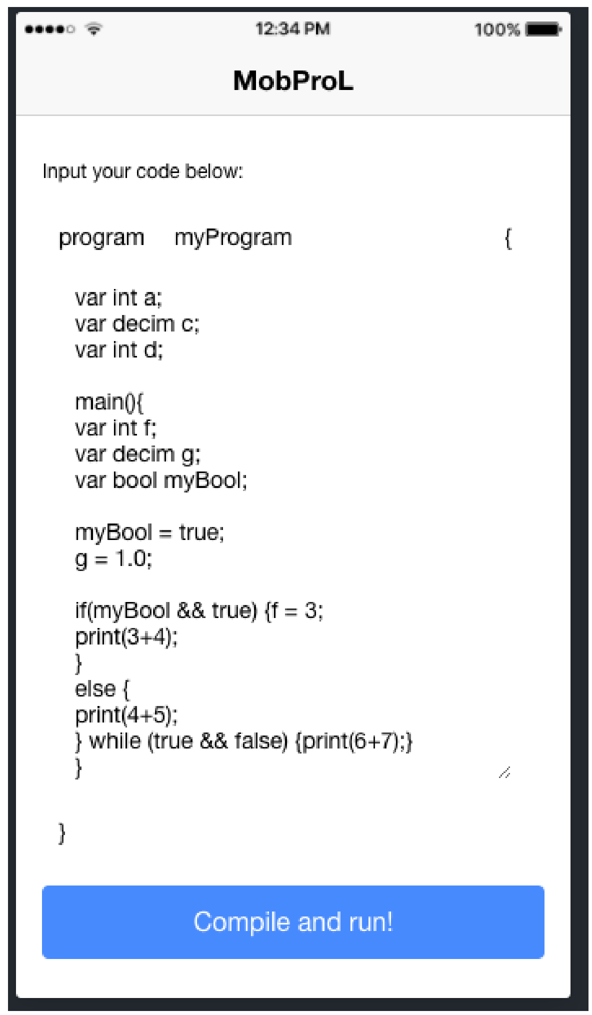
```
▶ 0: varTuple {value: 6}
▶ 15004: varTuple {value: 2}
▶ 20002: varTuple {value: 8.5}
▶ 20003: varTuple {value: 1}
▶ 25001: varTuple {value: false}
▶ 35001: varTuple {value: 2.5}
▶ 35002: varTuple {value: 6.5}
▶ 35003: varTuple {value: 6.5}
▶ 35004: varTuple {value: 8.5}
▶ 40001: varTuple {value: false}
▶ __proto__: Object
```

Test 3:

Code

```
program patito {
var int a;
var decim c;
var int d;

main(){
var int f;
var decim g;
var bool myBool;

myBool = true;
g = 1.0;

if(myBool && true) {f = 3;
print(3+4);
}
else {
print(4+5);
} while (true && false) {print(6+7);}
}
}
```

Code as seen on the app



Resulting quadruples

```
0['GOTO', '', '', 2]
1['main', '', '', '']
2['=', 'true', '', 25001]
3['=', '*1.0*', '', 20001]
4['&&', 'true', 'false', 40001]
5['gotof', 40001, ' ', 9]
6['+', '*6*', '*7*', 30001]
7['print', 30001, '', '']
8['goto', ' ', ' ', 4]
9['ret', '*0*', '', '']
10['END', '', '', '']
```

Resulting memory

```
program patito {
	var int a;
	var decim c;
	var int d;

	main(){
		var int f;
		var decim g;
		var bool myBool;

		myBool = true;
		g = 1.0;

		if(myBool && true) {f = 3;
			print(3+4);
		}
		else {
			print(4+5);
		} while (true && false) {print(6+7);}
	}
}
```

```
0['GOTO', '', '', 2]
1['main', '', '', '']
2['=', 'true', '', 25001]
3['=', '*1.0*', '', 20001]
4['&&', 'true', 'false', 40001]
5['gotof', 40001, ' ', 9]
6['+', '*6*', '*7*', 30001]
7['print', 30001, '', '']
8['goto', ' ', ' ', 4]
9['ret', '*0*', '', '']
10['END', '', '', '']
Global: dict_items([('global', {'int': {'a': 0, 'd': 1}, 'decim': {'c': 5001}}),
 ('main', {'int': {'main': 2}})])
Local: dict_items([('main', {'funcType': {'return': 'void'}, 'int': {'f': 15001}
, 'decim': {'g': 20001}, 'bool': {'myBool': 25001}})])
Aux: dict_items([('Aux', {'bool': {'t0': 40001}, 'int': {'t1': 30001}})])
```

MobProl

User Manual

Flores, L.;Salazar, Y.

Compiler Design

5/7/18

Index

Lexis and syntax *27*

Example programs *29*

1. **Iterative Fibonacci series** 29
2. **Iterative factorial function** 30
3. **Recursive factorial function** 30
4. **Array find** 31

Lexis and syntax

MobProl is a language aimed at young audiences who would like to start learning about programming with simple programs with basic concepts or want to try programming ideas on the go.

On the lexis, MobProl supports four different types for variables and functions: int (Integer numbers), decim (Decimal values), bool (Boolean operators), void (Aimed to be used in functions). You will find that, aside from the name "decim" for our floating-point values, most other reserved words in the syntax are congruent with most popular programming languages. This language is meant to serve as a starting point for a programmer.

The syntax of a common program requires to write a main function as well as the name of the program. Thus, the simplest program one could write is as follows:

```
program     myProgram                  {

  main() {}

}
```

MobProl's syntax supports the following:

- Primitive, array and matrix variables declaration
  - var int myInt;
  - var bool myBoolArr[10];
  - var decim myDecimMat[10][2];
- Variable assignation
  - myInt = 5;
  - myDecimMat[8][2] = 9.5;
- Functions and recursive calls with pass by value parameters
  - func int myFunc(int myParam) {}
  - myVar = myFunc(myParamToSend);
- Prints
  - print (myVar);

    - print (1);
    - print (“Hello World”);
- Read for user input
    - read (myVar);
- Expressions
    - myInt = myOtherInt + 4;
- Conditionals
    - if (myCondition) {}
    - if (myCondition) {} else {}
- Cycles
    - while (myCounter <= myArrLength) {}

Example programs

MobProl is aimed to write simple programs with simple user interaction. Following, some example programs that one could write:

Iterative Fibonacci series

```
program Fibonacci {
    main() {
            var int n;
            var int t1;
            var int t2;
            var int nextTerm;
            var int i;

            t1 = 0;
            t2 = 1;
            nextTerm = 0;

            print("Terms to print for Fibonacci");
            read (n);

            i = 0;
            while (i <= n) {
                    if (i == 1) {
                      print (t1);
                    } else {
                    if (i == 2) {
                      print (t2);
                    }
                    else {
                      nextTerm = t1 + t2;
                      t1 = t2;
                      t2 = nextTerm;
                      print (nextTerm);
                    }}

                    i++;
            }

    }
}
```

Iterative factorial function

```
program FactorialIterative {
    func int factorial(int num) {
        var int result;
        result = 1;

        while (num > 0) {
            result = result * num;
            num--;
        }

        return result;
    }

    main() {
        var int factStart;

        print("Pick a number to get its factorial");
        read(factStart);

        print(factorial(factStart));
    }
}
```

Recursive factorial function

```
program FactorialRecursive {
        func int factorial(int num) {
            if (num > 1) {
                return (num * factorial(num-1));
            }
            else {
                return num;
            }
        }

        main() {
            var int factStart;

            print("Pick a number to get its factorial");
            read(factStart);

            print(factorial(factStart));
        }
}
```

Array find

```
program ArrayFind {
  main() {
    var int a[10];
    var int aLength;
    var int i;
    var int numToSearch;
    var int numIndex;

    aLength = 10;

    print("Please input 10 numbers for your array");
    i = 1;
    while (i <= aLength) {
        read(a[i]);
        i++;
    }

    i = 1;
    while (i <= aLength) {
        print(a[i]);
        i++;
    }

    print("Please tell me the number you would like to find");
    read(numToSearch);

    numIndex = 0;
    i = 1;
    while (i <= aLength && numIndex < 1) {
        if (a[i] == numToSearch) {
            numIndex = i;
        }
        i++;
    }

    if (numIndex > 0) {
        print("Found your number on index");
        print(numIndex);
    }
    else {
        print("Sorry I could not find your number on the array provided");
    }
  }
}
```