I - INTRODUCTION

As part of the end of the first semester, we were asked to carry out a second project. This one is about making a basic calculator that can obviously calculate numbers between them but also with booleans or simple variables. This basic calculator must also be able to do other things that we will explain throughout this file.

With Lucas we manage to do this project the best way we could. In fact, our basic calculator is composed of 3 functions. Each function has a different aim.

The first analyses the operands and the operators of the input of the user.

The second evaluates it and the final one permits to quit the calculator and essentially to print the result.

II – ARCHITECTURE OF THE CALCULATOR

To be more precise we will explain each function one by one.

The parse function:

In this function we look for the integer operators, variables, Booleans, OR & AND, NOT, ==, =, <>, <=, >=, string operand, integer operand, parenthesis and if the function finds one of those in the input of the user, the function will add this to a tupple with his category (operand, operators, etc...)

The evaluate function:

This function is the most important one because it analyses the expression that it receives from the parse function, and then it identifies the different operations to do. It works recursively and considers the priorities of calculation. So, it means that it first looks for the operator with the most important precedence, when there is no operator of this precedence in the expression it looks for the operator of next precedence. In fact, we look for operator in this order: or,and,not,Boolean comparator, variable, +-,*/.

To search different operators this function goes from the end to the beginning because we work recursively, and it means that the operation will finally goes from the begin to the end.

When the function identifies an operator, it cuts the expression into 2 other one, left/right expression and with those two expressions it does the operation of the evaluation of the 2 expressions depending on the operator the function finds. So, these 2 news expressions will be cut again and again before their length became 1. Lastly, when the length is one the evaluate function returns the value of the operand without the tuple and all the operation will be done in order of precedence.

That's in this function that we identify the variable, during its analysis, if the function finds a single "=" then it will enter the value of the term next to this = in a dictionary with the key of the term before this is equal. Then when the expression has a length of 1 it checks if the only term is in the key of the dictionary and if it's the case then it returns the value associated with this key in the dictionary

The last functionality of this function is the error, if it finds that the last term of the operation is an operator it returns error, but also if it finds another type as an integer in a division or multiplication it returns error too.

The calculator function:

First, we initialise a variable named “emptycount” to 0.

Then we initialize another variable named “line” which allows the user to enter its calculation.

If the user enters nothing and skip the line, the counter of “emptycount” will increase and the second time the program will stop.

Finally, if the user has entered a calculation, this function prints the result by calling the two previous functions.

III – ACHIEVEMENTS & DIFFICULTIES

Achievements:

- Integer operations with priorities and simple parenthesis
- Negative integer
- String opetarions with simple parenthesis
- Boolean operations with priorities and simple parenthesis
- Variable
- Converting Boolean and integer operand when they are after a string
- Error of operator, unknown variable (only with simple case), and error or multiplication and division

The difficulties we encountered:

At first, it is interesting to note that we had never done anything similarly before. A calculator that allows all these calculations was a major challenge. We then asked ourselves many times on many points to make progress on more or less important issues.

The one who asked us the most work was surely on the priorities which was a real headache to solve by the recursion that we had to use and that we do not yet master perfectly. And of course, there is the problem that we have not managed to solve the parentheses in parentheses. We could not understand how it was possible to take them into account and we decided to continue the project without taking this into account.

There are also some errors that were not easy to reveal, in fact at the end of the project when we tried to do the error messages, we had a problem with the negative operand because when we evaluate them their first term is a “– “ which is a string so we could not compare the type of the right/left expression as we wanted. We had the same problem with the “not “for Boolean expression.

Finally, I think our program is not optimised because, the biggest part is in the evaluate function and sometimes we repeat same operation but doing function into a recursive function was not clear in our mind.

```
? 1
1
? (1)
1
? 4/2*3
6.0
? 4*2/3
2.6666666666666665
? 1+2*3
7
? (1+2)*3
9
? -1+-1+(-1--1)
-2
? "hello"
hello
? "hello"+""+"world"
helloworld
? "hello"+" "+("world"+"!")
hello world!
?
```

```
? true
True
? true or false and false
True
? (true or false) and false
False
? not false and not false
True
? 1<0
False
? "hi"<>"hello"
True
? "hi"<"hello"
False
? 1+2+3==2*3
True
? 1<5 and 5<10
True
? 1<5 and not "hi"=="hello"
True
?
```

```
? "1+2="+1+2
1+2=12
? "1+2="+(1+2)
1+2=3
? "1+2>3="+(1+2>3)
1+2>3=False
? "temperature:"+5+"°C,cold: "+(5<25)
temperature:5°C,cold: True
?
```

```
? x=1

? y=1+2

? message="hello"

? errorcause="type"+"mismatch"

? oneisodd=1==1

? aredistinctcolors="blue"<>"orange"

? x
1
? x+2*y
7
? "error:"+errorcause+"!"
error:typemismatch!
? oneisodd and not aredistinctcolors
False
? x=x+y

? x
4
?
```

```
? 2+
Error, missing operand
? x
Error, unknown variable
? false*false
Impossible operation
?
```

Because we don’t know how to translate certain things from python to algorithm , we wrote our code in this following way:

Algorithm: IDENTIFY VARIABLES

VAR: i,j: integer

Alphabet, string: Array of characters

Token : tupple(multidimension array)

BEGIN

i goes through the string from position 1 to the end of this one.

Fill a list containing all the letters of the alphabet

Check if the element at the index i is in this list.

If it is, our variable “i” will stack to the index of the beginning of the expression

and the variable “j” will increase to the end of the expression while there is a letter.

the expression between “i” and “j” will be assimilated to a variable named “variable”.

If variable is Boolean operand (true or false), we add this variable to a list and we precise in the second term of the list that it’s an operand

If variable=or or and or not, we do the same and precise its an operator in the second term of the list

Else, the variable is a simple variable and we add it also to a tupple and we precise it’s a simple one

Finally we add the list to a tupple named tokens and i take the value of j

we continue

END

This algorithm is needed to look for variables. We search in the input of the user if he has entered a letter. Then we delimit the word. We check if it’s; a Boolean variable or a simple one. Finally, we add it to a tupple and we initialize I to j to search after this word if there is something else.

```
Algorithm= VARIABLE
Variable: length,j : integer
          Expression : tupple(multidimensional array)
          Dictionary : dictionnary
          Is_variable : boolean
BEGIN
Is_variable=false
If length of the expression=1:
For j in the key of the dictionary
  If j = expression:
          Is_variable = true
If is variable= true:
  Return dictionary [expression [0][0]]
If expression [0][1] = “variable”:
Return (“error, unknown variable”)
If expression [0][1] =” operator”:
 Return (“error missing operand”)
End if end if end if end if
Else:
return (expression [0][0])
END
```

This algorithm is needed when the length of the expression of the user is 1. First, we search in our dictionary of variable if this variable was attribute to a key. If it exists we return the value of the dictionary at this key. If it doesn’t exist, we search if this expression is a variable or an operator. And if it’s not it’s obviously just an operand.

```
Algorithm : PARENTHESIS
Variable : length,i,j,save,parenthesis2,w : integer
           Expression,newExpression,parenthesis : tupple(multidimensionnal array
BEGIN
Length <- length(expression)
i <- 0
If length >1
        NewExpression = []
        If expression[length-1][1] == 'operator' then
                Return "error, missing operand"
        While I < length do
                If expression[i][0] == '('
                        j = i + 1
                        Save = i
                        While j < length and expression[j][0]=/=')' do
                                j<-j+1
                        end while
                        parenthesis  = expression from i to j
                        parenthesis2 = evaluate(parenthesis)
                        w <-0
                end if
                while w < length do
                        if w =/= save then
                                newExpression[w] <- [(expression[w],expression[w][1]
                                w<-w+1
                        end if
                        if w=save then
                                newExpression[w]<-[(parenthesis2, «parenthesis'»)]
                                w<-w+(j-w)
                        end if
                end while
                return evaluate(newExpression,variable)
        i <- i+1
        end while
end if
END
```

This algorithm is needed if there are parenthesis in the expression of length superior to 1. We browse the expression from the position 0 to length-1. If we find an open parenthesis we use a second variable j to search the end of this parenthesis then we cut the expression between the parenthesis and we evaluate this expression. Then, we create a new tupple using a while loop, this tupple will have the same value as previous one except for the expression between parenthesis which will be only one operand. Finally, we evaluate the new expression.

```
Algorithm: ASSOCIATE VARIABLE
VAR : i, length:integer
Left expression, right expression, expression : tupple
Variable : dictionary
BEGIN
i<-len(expression)
while i>0 do
        If expression[i][0]= « = » then
            left expression<- expression before "="
            right expression<- expression after "="
            variable (left expression [0][0]<-evaluate(right expression, variable)
            return (write new line )
          end if
      i<-i-1
end while
END
```

This algorithm is needed to create and save variables. If we find one "=", we cut our expression and we get two expressions. One at the left of the "=" and the other at the right. Then we associate the dictionary key to the value of the left expression and the value of the dictionary at this key to the evaluation of the value of the right expression. Finally, we return a blank so that the user can use his variable or to input something else. At the end of the while loop, we reinitialize i to length -1 because we think recursively.

```
Algorithm : EVALUATION OF EXPRESSIONS WITH DIFFERENTS OPERATORS
VAR: i: integer
        Left expression, right expression, expression: tupple
BEGIN
 I<- length(expression)
While i>0 do
    If expression[i][0]=”*” then
       left expression<- expression before “*”
       right expression<- expression after “*”
       if left expression and right expression have different type then
       return (“error”)
       return (evaluate(left expression, variable) * evaluate(right expression, variable))
       end if
   end if
end while
END
```

This algorithm is needed to evaluate an expression with operators, in this example,we have illustrate with “*”, but it works the same way for all the operators. We cut the expression where the operator is. Then we have two expressions. One at the left of the operator and one at the right. We evaluate at each side and we make the calculation between the evaluation of the expressions.

THESE DIFFERENTS ALGORITHMS BUILD THE EVALUATE FUNCTION WHICH WORKS RECURSIVELY