Daniela Uribe 201923291
Juan Felipe Patiño 201922857

Project 0: Building a Parser

Profesor: Silvia Takahashi
Course: Lenguajes y Maquinas (Ingles)
Section: 1

Daniela Uribe 201923291
Juan Felipe Patiño 201922857

26 August 2022

Abstract**:**

This report intends to give a detailed explanation on the code we developed to solve project 0. We started by replacing commands for token and then we programmed methods to check the general syntaxis, to check an instruction, to check parameters, to check each command and to check control instructions.

Key Words: *Parser, Python, Tokens, Instructions, Code*

Daniela Uribe - 201923291
Juan Felipe Patiño - 201922857

1. Introduction:

The task is to create a parser from scratch. We chose to use python for its ease of use and debugging. To begin we used tokens to be able manipulate a single character and not a word. This allowed us to identify the purpose of a part of a text more easily and utilize similar syntax functions in the same category, even if their purposes differed. On the other hand, we created an array to keep different parts of the text and other arrays to keep important information that arose from the reading of the text. After that we have many methods, whose purpose is checking different aspects of syntaxis. We did this so we don't manipulate massive portions of code and to make the debugging process easier. At the end we have a FOR structure that checks instructions as PROG, GORP, VAR, PROC and CORP are located and used correctly and that calls the methods stated above. Finally, we checked the final state of the program and send a message indicating the user if the syntaxis was correct or wrong.

2. Materials and method

The following will be a breakdown of all the code utilized to achieve the goal. We'll look at three types of code, first the ones related to initializing the program, secondly the instruction specific codes, and finally the main section which functions as a reader, divider and text cleaner to then send to the appropriate instruction specific code.

2.1 Initializing

We begin by reading the file of the text that will be checked. Likewise, we remove spaces and tabulations as they aren't relevant

```
#Lectura del archivo
f= open("Example2.rob", "r")

text=""
for x in f:
    text+=x

#Eliminación de espacios y tabulaciones
text= text.replace(" ", "")
text= text.replace("\t", "")
```

We proceed to do several replacements which serve to identify each specific method in later stages.

```
#Reemplazo filas
text=text.replace("PROG", "P")
text=text.replace("GORP", "G")
text=text.replace("var", "A")
text=text.replace("PROC", "R")
text=text.replace("CORP", "C")
text=text.replace("canWalk", "N")
text=text.replace("walk(", "K(")
text=text.replace("jumpTo", "J")
text=text.replace("jump(", "W(")
text=text.replace("drop(", "W(")
text=text.replace("grab(", "W(")
text=text.replace("get", "W")
text=text.replace("free(", "W(")
text=text.replace("pop(", "W(")
text=text.replace("veer", "V")
text=text.replace("look", "L")
text=text.replace("isfacing", "S")
text=text.replace("isValid", "I")
text=text.replace("not", "T")
```

Then we created and array which spliced the text by a line jump. The we created 2 variables to check the state of the program and a variable to save a Boolean indicating if there was a mistake. Then we created two arrays that are going to save the variables declared at the begging of the text and the methods (with their number of parameters). Finally, we created 4 arrays to save specific groups of parameters.

```
#Arreglo por saltos de lineas
arreglo=list(map(str, text.split("\n")))

arreglo=list(filter(None, arreglo))

#Estado
prog=False
gorp=False
error=False
print(arreglo)

#Variables
variables=[]

#Metodos
metodos=[]

#Dir
dir=["north", "south", "east", "west"]
dir2=["front", "right", "left", "back"]
dir3=["right", "left", "around"]
ins=["walk", "jump", "grab", "pop", "pick", "free", "drop"]
```

2.2 Instruction specific functions

We start the specific functions for each method with the one that checks that the variables are correctly declared: that the declaration starts with VAR, that they are separated by commas, that it ends with semicolon, that the characters are alphanumeric and that they start with a letter. At the end, we save them in an array.

```
def checkA(fila):

    retorno=True
    #Espacio después de la A
    if not (fila.startswith("A")):
        return False
    fila= fila[1:]

    #Nombres de las variables separados por coma
    nombres= fila.split(",")
    for i in range(len(nombres)):
        if i== (len(nombres)-1):
            nombres.append(nombres[i][1:])
            nombres[i]=nombres[i][:1]
            #Termina por un ;
            if not (nombres[i+1]==";"):
                retorno=False
        #Son caracteres alfanuméricos
        if not(nombres[i].isalnum()):
            retorno=False
            break
        #El primer caracter es una letra
        if not(nombres[i][:1].isalpha()):
            retorno=False
            break

        variables.append(nombres[i])
    return retorno
```

Then we check the sections that start with PROC and end with CORP. First, we check that it starts with and R. Then we check that the name of the procedure is alphanumeric and that the first character is a letter. After that, we check that the declaration ends with 2 parentheses, and we add the name of the method and the number of parameters to an array. At the end, we just take the "Instruction", we split it by semicolon and send it to a method to check it.

```
def checkC(bloque):
    print(bloque)

    #Espacio después de la R
    if not (bloque.startswith("R")):
        return False

    #Paréntesis apertura
    paren= bloque.find("(")
    if not(paren>0):
        return False
    else:
        nomFun=bloque[1:paren]
        #Son caracteres alfanuméricos
        if not(nomFun.isalnum()):
            return False
        #El primer caracter es una letra
        if not(nomFun[:1].isalpha()):
            return False

    #Termina con paréntesis
    paren2= bloque.find(")")
    if not(paren2>0):
        return False

    param=bloque[paren+1: paren2+1]
    #Separados por comas
    if param.startswith(")"):
        metodos.append((nomFun, 0))
    else:
        param=param[:len(param)-1]
        param= param.split(",")

        #Agregar método
        metodos.append((nomFun, len(param)))
```

```
ajustado=bloque[paren2+1:]
ajustado=ajustado[:-1]
ajustado= ajustado.replace("\n", "")

ajustado=ajustado.split(";")

return checkBloque(ajustado, param)
```

Moving on is the command walk. First, we check that it ends with parenthesis and that its parameters are separated by commas. We then check that the parameters have a correct syntaxis with the method checkn.

```
def checkK(valor, param):

    #Paréntesis
    paren= valor.find("(")
    if not(paren>0):
        return False
    if not(valor.endswith(")")):
        return False

    #Parámetros
    param2= valor[paren+1:(len(valor)-1)]
    param2=param2.split(",")

    #Casos
    if(len(param2)==1):
        param2= param2[0]
        return checkn(param2, param)
    elif len(param2)==2:
        if not((param2[0] in dir) or (param2[0] in dir2)) :
            return False
        if (not(checkn(param2[1], param))):
            return False
    else:
        return False

    return True
```

We group the commands jump, drop, grab, get, free and pop. First, we check that it ends with parenthesis and that its parameters are separated by commas. We then check that the parameters have a correct syntaxis with the method checkn.

```
def checkW(valor,param):

    #Paréntesis
    paren= valor.find("(")
    if not(paren>0):
        return False
    if not(valor.endswith(")")):
        return False

    #Parámetros
    param2= valor[paren+1:(len(valor)-1)]
    param2=param2.split(",")
    param2= param2[0]

    return checkn(param2, param)
```

Checkn is a commonly used as it is a method that checks that parameters fulfill the next conditions: either it is numeric, it is a parameter that entered in the method that is being checked or is one of the variables that were declared at the beginning of the text.

```
def checkn(valor, param):

    #Es un número, Es un parámetro de la fun original, Es una variable
    if not(valor.isnumeric() or valor in param or valor in variables):
        return False

    return True
```

Returning to functions we have jumpTo. First, we check that it ends with parenthesis and that its parameters are separated by commas. We then check that the parameters have a correct syntaxis with the method checkn and that it has 2 parameters.

```
def checkJ(valor, param):

    #Paréntesis
    paren= valor.find("(")
    if not(paren>0):
        return False
    if not(valor.endswith(")")):
        return False

    #Parámetros
    param2= valor[paren+1:(len(valor)-1)]
    param2=param2.split(",")

    if not(len(param2)==2):
        return False
    else:
        return (checkn(param2[0], param) and checkn(param2[1], param))
```

Afterwards we have the one in charge of Veer. First, we check the existence and correct placement of parenthesis, that contain the parameter of the function. Then we get the two parameters that should exist, and when we are in the execution instructions, we ensure that the first parameter is part of the acceptable list.

```
#Revisa los comandos V
def checkV(valor):
    #Paréntesis
    paren= valor.find("(")
    if not(paren>0):
        return False
    if not(valor.endswith(")")):
        return False

    #Parámetros
    param2= valor[paren+1:(len(valor)-1)]
    param2=param2.split(",")
    param2= param2[0]

    #Si se está revisando el bloque de instrucciones se debe verificar que los parámetros
    #sean válidos.
    if activado2==True:
        if not(param2 in dir3):
            return False
    return True
```

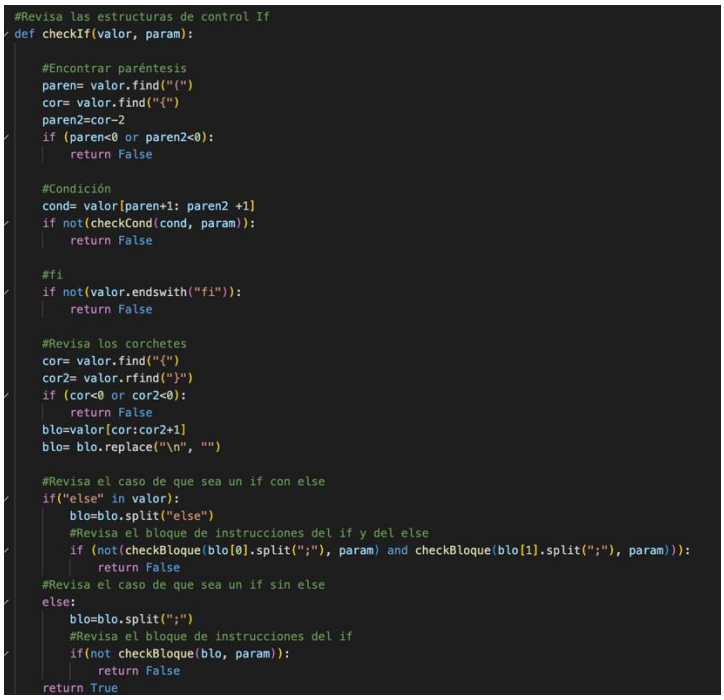
In this part we check the commands look. First, we check that it ends with parenthesis and that its parameters are separated by commas. We then check that the parameters have a correct syntaxis by looking if the parameter belongs to dir when we are in the instructions section, which has specific directions that this command allows.

```
#Revisa los comandos L
def checkL(valor):
    #Paréntesis
    paren= valor.find("(")
    if not(paren>0):
        return False
    if not(valor.endswith(")")):
        return False

    #Parámetros
    param2= valor[paren+1:(len(valor)-1)]
    param2=param2.split(",")
    param2= param2[0]

    #Si se está revisando el bloque de instrucciones se debe verificar que los parámetros
    #sean válidos.
    if activado2==True:
        if not(param2 in dir):
            return False
    return True
```

Following up we have to make sure the is valid functions work, first, we check that it ends with parenthesis and that its parameters are separated by commas. We then check that the parameters have a correct syntaxis by looking for two parameters and if it's in the instructions block, we make sure the parameters are part of the acceptable list for the method.

```
#Revisa los comandos I
def checkI(valor):

    #Paréntesis
    paren= valor.find("(")
    if not(paren>0):
        return False
    if not(valor.endswith(")")):
        return False

    #Parámetros
    param2= valor[paren+1:(len(valor)-1)]
    param2=param2.split(",")

    lista=[]
    #Casos
    #Revisa que haya 2 parámetros
    if not(len(param2))==2:
        return False
    else:
        #Si se está revisando el bloque de instrucciones se debe verificar que los parámetros
        #sean válidos.
        if activado2==True:
            if not(param2[0] in ins) :
                return False
        #Revisa los parámetros
        if not(checkn(param2[1], lista)):
            return False

    return True
```

Up next, we have the first control structure for if. First, we extract the condition by finding the parenthesis around it, and we check said condition with the function. Afterwards we make sure it follows the structure by ending with fi and after grabbing the instruction block, and making sure it exists properly, we have to cases. The first one is when the structure has an else, in which case we split it over the else and treat the two instruction codes as separate and check then, otherwise we check the single instructions block.



```
#Revisa las estructuras de control If
def checkIf(valor, param):

    #Encontrar paréntesis
    paren= valor.find("(")
    cor= valor.find("{")
    paren2=cor-2
    if (paren<0 or paren2<0):
        return False

    #Condición
    cond= valor[paren+1: paren2 +1]
    if not(checkCond(cond, param)):
        return False

    #fi
    if not(valor.endswith("fi")):
        return False

    #Revisa los corchetes
    cor= valor.find("{")
    cor2= valor.rfind("}")
    if (cor<0 or cor2<0):
        return False
    blo=valor[cor:cor2+1]
    blo= blo.replace("\n", "")

    #Revisa el caso de que sea un if con else
    if("else" in valor):
        blo=blo.split("else")
        #Revisa el bloque de instrucciones del if y del else
        if (not(checkBloque(blo[0].split(";"), param) and checkBloque(blo[1].split(";"), param))):
            return False
    #Revisa el caso de que sea un if sin else
    else:
        blo=blo.split(";")
        #Revisa el bloque de instrucciones del if
        if(not checkBloque(blo, param)):
            return False
    return True
```

The check conditional considers four cases. First one is for isFacing, secondly isValid third is canWalk and finally for the not. With an elif we send it to the appropriately working method, but in the case of a condition of negation we remove said condition to reveal the one inside and then check that one. If it isn't part of the acceptable conditions, it is understood as an error.

```
def checkCond(valor, param):
    if(valor.startswith("S")):
        return checkK(valor, param)
    elif(valor.startswith("I")):
        return  checkL(valor)
    elif(valor.startswith("N")):
        return checkL(valor)
    elif(valor.startswith("T")):
        valor= valor[2: len(valor)-1]
        return checkCond(valor, param)
    else:
        return False
```

The next control structure is for while. First, we extract the condition by finding the parenthesis around it, and we check said condition with the function. Afterwards we make sure it follows the structure by ending with od and after grabbing the instruction block and making sure it exists properly.

```
def checkWhile(valor, param):

    #Encontrar paréntesis
    paren= valor.find("(")
    paren2=valor.find(")")
    if (paren<0 or paren2<0):
        return False

    #Condición
    cond= valor[paren+1: paren2 +1]
    if not(checkCond(cond, param)):
        return False

    #od
    if not(valor.endswith("od")):
        return False

    cor= valor.find("{")
    cor2= valor.rfind("}")
    if (cor<0 or cor2<0):
        return False
    blo=valor[cor:cor2+1]
    blo= blo.replace("\n", "")


    blo=blo.split("do")
    if (not(checkBloque(blo[0].split(";"), param))):
        return False

    return True
```

Finally, we have the control structure for repeat times. First, we extract the condition by finding the parenthesis around it, and we check said condition with the function. Afterwards we make sure it follows the structure by ending with od and after grabbing the instruction block and making sure it exists properly.

```
def checkRepeat(valor, param):

    #per
    if not(valor.endswith("per")):
        return False

    cor= valor.find("{")
    cor2= valor.rfind("}")
    if (cor<0 or cor2<0):
        return False

    n=valor[cor-1:cor]

    if(not checkn(n, param)):
        return False

    blo=valor[cor:cor2+1]
    blo= blo.replace("\n", "")
    blo=blo.split(";")

    if(not checkBloque(blo, param)):
        return False

    return True
```

Up next, we consider the methods which are user created, meaning they haven't been previously considered. We heck for appropriate parenthesis usage and grab the parameters used in the function. Then we count the amount and create a tuple of the name and number of parameters following by checking them on the list to make sure it exists in the created functions.

```
def checkMetodos(valor):

    #Paréntesis apertura
    paren= valor.find("(")
    if not(paren>0):
        return False
    #Termina con paréntesis
    paren2= valor.find(")")
    if not(paren2>0):
        return False

    param=valor[paren+1: paren2]

    cant=0
    #Separados por comas
    if not(param == ""):
        param= param.split(",")
        cant= len(param)

    nombre= valor[:paren]
    tupla=(nombre, cant)

    if not(tupla in metodos):
        return False

    return True
```

Another command is assignment, this one makes the sure the assignment syntaxis is followed and in the order expected and then makes sure the value is numeric as expected.

```
def checkAsig(valor):

    #Dos puntos, igual
    puntos= valor.find(":")
    igual=valor.find("=")

    if not(puntos==igual-1):
        return False

    if (puntos==-1 or igual ==-1):
        return False

    valorsito= valor[igual+1:]

    if not ( valorsito.isnumeric()):
        return False

    return True
```

2.3 General Blocks

Up next is the block checker. We'll view it in three portions to make it manageable. To begin we receive a block of instructions which must end and start with curly brackets, and after removing them we will check it line by line. Beforehand, we create some local variables

that will help us count the appearances of the command structures, and whether one is active and, in that case, add up those lines. To begin the loop, we check for appearances of command structure words such that we add to the counter when the opening words appear and rest one when the closing one ends to make sure it's at zero, or a complete block even if they are nested, to be checked. We do this for the three types of structures.

```
#Revisa los bloques de instrucciones
def checkBloque(ajustado, param):

    #Corchete apertura
    if not (ajustado[0].startswith("{")):
        return False

    #Corchete cierre
    if not (ajustado[len(ajustado)-1].endswith("}")):
        return False

    ajustado[0]=ajustado[0][1:]
    ajustado[len(ajustado)-1]=ajustado[len(ajustado)-1][:-1]

    parte=""
    control=[False, ""]
    cantidadIf=0
    cantidadWhi=0
    cantidadRe=0

    for i in range(len(ajustado)):

        #Cantidades
        if( "if" in ajustado[i]):
            ifs=ajustado[i].count("if")
            cantidadIf=cantidadIf +ifs
        if("fi" in ajustado[i]):
            ifs=ajustado[i].count("fi")
            cantidadIf=cantidadIf -ifs
        if( "while" in ajustado[i]):
            whis=ajustado[i].count("while")
            cantidadWhi=cantidadWhi +whis
        if("od" in ajustado[i]):
            whis=ajustado[i].count("od")
            cantidadWhi=cantidadWhi -whis
        if("repeatTimes" in ajustado[i]):
            res=ajustado[i].count("repeatTimes")
            cantidadRe=cantidadRe + res
        if("per" in ajustado[i]):
            res=ajustado[i].count("per")
            cantidadRe=cantidadRe - res
```

Following up we check for the appearances of the previously replaced key identifiers of the preloaded methods and then use the appropriate specific method. It is worth noting that the conditions are all placed together. Similarly, when a command structure is encountered it is activated and told as much in the control so that it can know it is active and which type of structure. If none of them are present then we check for usage of variable assignments, and if that is true, we check it with the sub method, otherwise we check if it's a user created method.

```
#Comandos
if (control[0]==False):
    """ if ( (i< len(ajustado)-3) and ( len(ajustado) >1)):
        if not(ajustado[i].endswith(";")):
            return False
        ajustado[i]=ajustado[i][:-1] """
    if(ajustado[i].startswith("K")):
        if not(checkK(ajustado[i], param)):
            return False
    elif(ajustado[i].startswith("W")):
        if not(checkW(ajustado[i], param)):
            return False
    elif(ajustado[i].startswith("J")):
        if not(checkJ(ajustado[i], param)):
            return False
    elif(ajustado[i].startswith("V")):
        if not(checkV(ajustado[i])):
            return False
    elif(ajustado[i].startswith("L")):
        if not( checkL(ajustado[i])):
            return False
    elif(ajustado[i].startswith("I") or ajustado[i].startswith("S") or ajustado[i].startswith("N") or ajustado[i].startswith("T")):
        if not( checkCond(ajustado[i], param)):
            return False
    elif (ajustado[i].startswith("if")):
        control=[True, "if"]
    elif (ajustado[i].startswith("while")):
        control=[True, "while"]
    elif (ajustado[i].startswith("repeatTimes")):
        control=[True, "repeat"]
    else:
        empieza=False
        for x in variables:
            if(ajustado[i].startswith(x + ":=")):
                empieza=True
        if(empieza==True):
            if not(checkAsig(ajustado[i])):
                return False
        elif not(checkMetodos(ajustado[i])):
            return False
```

Beyond the previous if of reserved word, we have the control structure if, which was activated in some instances of the previous if. In this case we add the parts of the text to the text saver and do so until the currently control activated structure has reached zero meaning it is closed. When this happens, the same path is followed in all cases, as the final semicolon is removed, and sent to the appropriate checker and afterwards the variables are cleaned. After the for is finished it is checked that there are no uneven instances of control structures as a final check.

```
        if control[0]==True:
            parte=parte + ajustado[i] + ";"

            if (cantidadIf==0 and control[1]=="if"):
                parte=parte[:-1]
                if not(checkIf(parte, param)):
                    return False
                control=[False, ""]
                parte=""
                cantidadWhi=0
                cantidadRe=0

            elif (cantidadWhi==0 and control[1]=="while"):
                parte=parte[:-1]
                if not(checkWhile(parte, param)):
                    return False
                control=[False, ""]
                parte=""
                cantidadIf=0
                cantidadRe=0

            elif (cantidadRe==0 and control[1]=="repeat"):
                parte=parte[:-1]
                if not(checkRepeat(parte, param)):
                    return False
                control=[False, ""]
                parte=""
                cantidadIf=0
                cantidadWhi=0

    if not(cantidadIf==0 and cantidadWhi==0 and cantidadRe==0):
        return False
    return True
```

Up next, we have to Check Instructions which is used when the commands are sent at the end of the code, this is just a pathway to adapt the code to current structure and get it to the sent code. We remove the line breaks and split by semicolons.

```
def checkIns(bloque):

    ajustado= bloque.replace("\n", "")
    ajustado=ajustado.split(";")

    return checkBloque(ajustado, [])
```

For one of the most used checks, we ensure the current status is legal, meaning *prog* which is the start is active while *gorp* is not as it is the end. In other words, it ensures we are between the beginning and end of the code.

```
def checkEstado():
    return prog==True and gorp==False
```

As the main worker we have a loop over the whole introduced text. We read it line by line and to begin we have two cases. The first is when we are reading a none-final instruction block which we store together, the second case is when it is the final instruction block. We begin checking each line, first case if the P which is the start of the code and

activates it, we also check that is has not been activated previously. We also make sure the state is correct. After we check G which is the end of the code and if the state is correct, it inverts the states to the final possible one. Then we start with the A of variables, which after checking the state checks the variables.

```
#Revisión general sintaxis
for x in range(len(arreglo)):

    fila= arreglo[x]

    #Se activa si la linea pertenece a una declaración de un método
    if(activado==True):
        bloque= bloque + "\n" + fila

    #Se activa si la linea pertenece al bloque de instrucciones
    if(activado2==True):
        bloque= bloque + "\n" + fila

    #Revisa PROG
    if(fila== "P"):
        if prog==True:
            error=True
            break
        prog=True
        if not checkEstado():
            error=True
            break

    #Revisa GORP
    elif(fila=="G"):
        if not prog==True:
            error=True
            break
        gorp=True
        prog=False

    #Revisa la declaración de variables
    elif( fila.startswith("A")):
        if not checkEstado():
            error=True
            break
        if not checkA(fila):
            error=True
            break
```

In the next fragment begins R which is the beginning of instructions and activates that. If it has been activated previously it is an error or if the state is wrong. If correct it begins stacking the block to check. Then we have the finisher of code of block with the C which in the correct state checks all the code, if it shows up without a previous start it is also an error. If all is good, it cleans the variables. Finally, we have the curly bracket cases which work for the final block of code and have the same logic as previously explained. It also considers the case where a line doesn't fit into any category and isn't part of instructions, which is an error.

```
#Revisa el PROC
elif(fila.startswith("R")):
    if activado==True:
        error=True
        break
    if not checkEstado():
        error=True
        break
    activado=True
    bloque=fila

#Revisa el CORP
elif(fila.startswith("C")):
    if not checkEstado():
        error=True
        break
    #Se manda a revisar el bloque
    if(activado==True):
        if not(checkC(bloque)):
            error=True
            break
    else:
        error=True
        break
    activado=False
    bloque=""

#Se revisa si es el comienzo del bloque de instrucciones general
elif(fila.startswith("{") and activado==False):
    bloque=fila
    activado2=True

#Se revisa si es el final del bloque de instrucciones general
elif(fila.startswith("}") and activado==False):
    if not(checkIns(bloque)):
            error=True
            break

#Revisa si la linea no cumple con ninguna condición
elif(activado ==False and activado2==False):
    error=True
    break
```

Finally, after all is done it checks that the final state is valid. Then it prints the state if there is a syntax error or not.

```
#Revisión del estado final
if not (gorp==True and prog==False):
    error=True
if activado== True:
    error=True

#Imprime el mensaje final de la validación
if error:
    print("La sintaxis es incorrecta.")
else:
    print("Aprobado.")
```

3. Results

The result is an extensive but valid solution. We accomplished the goal as any non-valid results were considered within the possible mistakes and is robust in this sense. Trying it with several examples of code we have tested it takes a time ranging from 1.2 to 0.4 milliseconds to run the whole code.

4. Discussion

At the end of the day, it might not have been the most efficient method to achieve it, but it works. If this was to be done again after seeing the latest course content, it might have been an approach through regular expressions but considering time constraints and content understanding at the beginning of the process it is a serviceable code.