

UNIVERSIDAD NACIONAL AUTÓNOMA DE HONDURAS



UNAH-CURLP
CENTRO UNIVERSITARIO REGIONAL
DEL LITORAL PACÍFICO

Cátedra: Diseño de Compiladores

Catedrático: Ing. Oscar Omar Pineda M.E.F

Integrantes

Angelo Rafael Velasquez Lara	20192330040
Dany José Valdez Escalante	20202300197
Mary Elizabeth Euceda Molina	20202300170
Omar Antonio Cruz Reyes	20202300114
Nestor Dario Aguilera Padilla	20192300030

INDICE

Introducción.....	3
Objetivos.....	3
Lenguaje.....	3
But.....	3
Objetivos principales.....	3
Objetivos secundarios.....	3
Funciones implementadas.....	3
Log.....	5
While.....	5
Operaciones abreviadas para expresiones.....	6
Condiciones.....	6
If/Else.....	8
Operador ternario.....	9
For.....	10
Switch.....	11
Do while.....	13
Var y Let.....	14
Alcance de las variables.....	14
Reglas permisivas para el punto y coma.....	15
Break y Continue.....	17
Tablas.....	18
Funciones.....	20
Return.....	23
Gestión de errores.....	24
Funciones no aplicadas (parte trasera).....	25
Guía del usuario.....	26
Requisitos.....	26
Instalación.....	26
Análisis léxico.....	26

INTRODUCTION

Nuestro proyecto, que se presenta a continuación, tiene como objetivo construir un compilador de JavaScript en Python desde cero, utilizando yacc/PLY para el lexer/parse. Este proyecto se lleva a cabo como parte de la clase de Diseño de Compiladores.

OBJETIVOS

LENGUAJE

Decidimos que para nuestro proyecto haríamos un compilador para Javascript no solo como un reto para nosotros mismos sino que también es un lenguaje el cual muchos tenemos conocimientos y hemos trabajado o estamos trabajando.

OBJETIVO

El objetivo de nuestro proyecto es tener un compilador que reconozca la mayoría de palabras reservadas y funcionalidades distintas al JavaScript orientado a objetos. Para ello, hemos establecido estas especificaciones objetivas:

OBJETIVOS PRINCIPALES

- Agregar if/else
- Agregar el para
- Agregar switch/case/default
- Agregar (para el do while)
- Agregar var/let
- Agregar reglas permisivas para el punto y coma.

OBJETIVOS SECUNDARIOS

- Agregar funciones y la palabra clave de function
- Agregar la palabra clave de return
- Añadido break/continue
- Agregar tablas con corchetes []

CARACTERÍSTICAS IMPLEMENTADAS

REGLAS

A pesar de que no está escrito explícitamente en los objetivos, hemos optado por dar la posibilidad de utilizar una sintaxis entre llaves (llamada programBlock en el analizador):

```
def p_program_block(p):  
    ''' programBlock : '{' new_scope program '}' '''  
    p[0] = p[3]  
    popscope()
```

Para esta sintaxis :

```
if(6<5){  
    console.log(2)  
} else if (1==5){  
    console.log(3)  
} else {  
    console.log(4)  
}
```

Pero también una sintaxis sin llaves para estructuras (no funciones) llamadas programStatement en el parser, ya que sin llaves, sólo se permite una declaración:

```
def p_program_statement(p):  
    '''programStatement : statement'''  
    p[0] = AST.ProgramNode([p[1]])
```

Para esta sintaxis :

```
if(6<5)  
    console.log(2)  
else if (1==5)  
    console.log(3)  
else  
    console.log(4)
```

Así como mezclas de las dos sintaxis (no recomendado porque ilegible, pero posible en javascript)

```
(2<5){  
    console.log(2)  
lse if (1==5)  
    console.log(3)  
se{  
    console.log(4)}
```

Como resultado, nuestro analizador proporciona capacidades javascript completas.

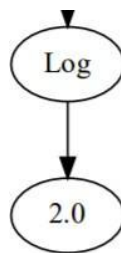
LOG

En javascript, para mostrar algo, tienes que usar `console.log()`

Cómo implementamos la parte de objetos de javascript(No todos cabe aclarar), `print` fue simplemente reemplazado por `console.log`. En el lexer, hemos añadido la palabra clave, y abajo está la implementación en el parser:

```
def p_log(p):  
    ''' logStatement : LOG '(' returnValues ')' '''  
    p[0] = AST.LogNode(p[3])
```

Aquí, los valores de retorno corresponden a lo que puede devolver una función, es decir, una expresión como una variable o un cálculo, una matriz o una función. El resultado en el árbol: para un `log(2)` :



WHILE

En primer lugar, el `while`, al igual que el `if`, contiene una condición, como se muestra a continuación. Además, hemos añadido la opción de un `while` sin llaves. El analizador sintáctico :

```
def p_while(p):  
    ''' structure : WHILE '(' condition ')' programStatement  
    | WHILE '(' condition ')' programBlock'''  
    p[0] = AST.WhileNode([p[3],p[5]])
```

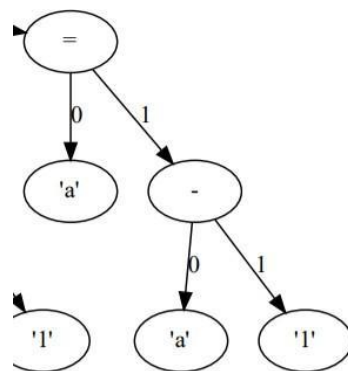
ATAJOS DE OPERACIÓN PARA EXPRESIONES

Además de las operaciones ya existentes, hemos habilitado atajos a la hora de asignar una variable, por ejemplo $a*=2$ o $a++$ que corresponden a $a = a*2$ y $a=a+1$.

```
def p_expression_op_assnation(p):  
    '''assnation : IDENTIFIER ADD_OP '=' expression  
    | IDENTIFIER MUL_OP '=' expression  
    | IDENTIFIER ADD_OP '=' functionCall  
    | IDENTIFIER MUL_OP '=' functionCall'''
```

```
def p_expression_op_assign_double(p):  
    '''assnation : IDENTIFIER ADD_OP ADD_OP'''
```

Estos atajos se representan en el árbol como si se hubieran escrito sin atajos :



CONDICIONES

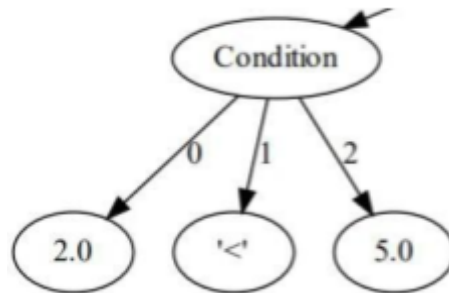
Parecía necesario introducir condiciones, en particular para mientras, si y para. En el :

```
t_LT = r'<'  
t_GT = r'>'  
t_LTE = r'<='  
t_GTE = r'>='  
t_EQUALVT = r'=== '  
t_NOTEQUALVT = r'!=='  
t_EQUALV = r'=='  
t_NOTEQUALV = r'!='  
t_AND=r'&&'  
t_OR=r'\\|\\|'
```

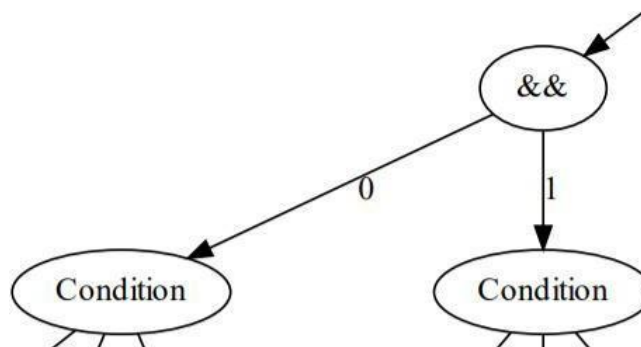
Ahora aceptamos todos los operadores de comparación y los operadores AND y OR. Además, hemos añadido! a los literales para permitir el operador NOT. En el analizador sintáctico, los operadores de comparación no son asociativos, por lo que no pueden encadenarse ($a < b < c$) y el comparador ! tiene prioridad sobre &&, que tiene prioridad sobre ||, como en javascript:

```
precedence = (  
  ('left', 'NEWLINE', 'ELSE', 'OR', 'IDENTIFIER', ',', ';'),  
  ('nonassoc', 'LT', 'GT', 'EQUALV', 'EQUALVT', 'NOTEQUALV', 'NOTEQUALVT', 'LTE', 'GTE'),  
  ('left', 'AND'),  
  ('left', 'ADD_OP'),  
  ('left', 'MUL_OP'),  
  ('right', 'UMINUS', '!')  
)
```

Una condición tiene este aspecto:



También hemos añadido los nodos AND, OR y NOT para los distintos operadores entre condiciones:

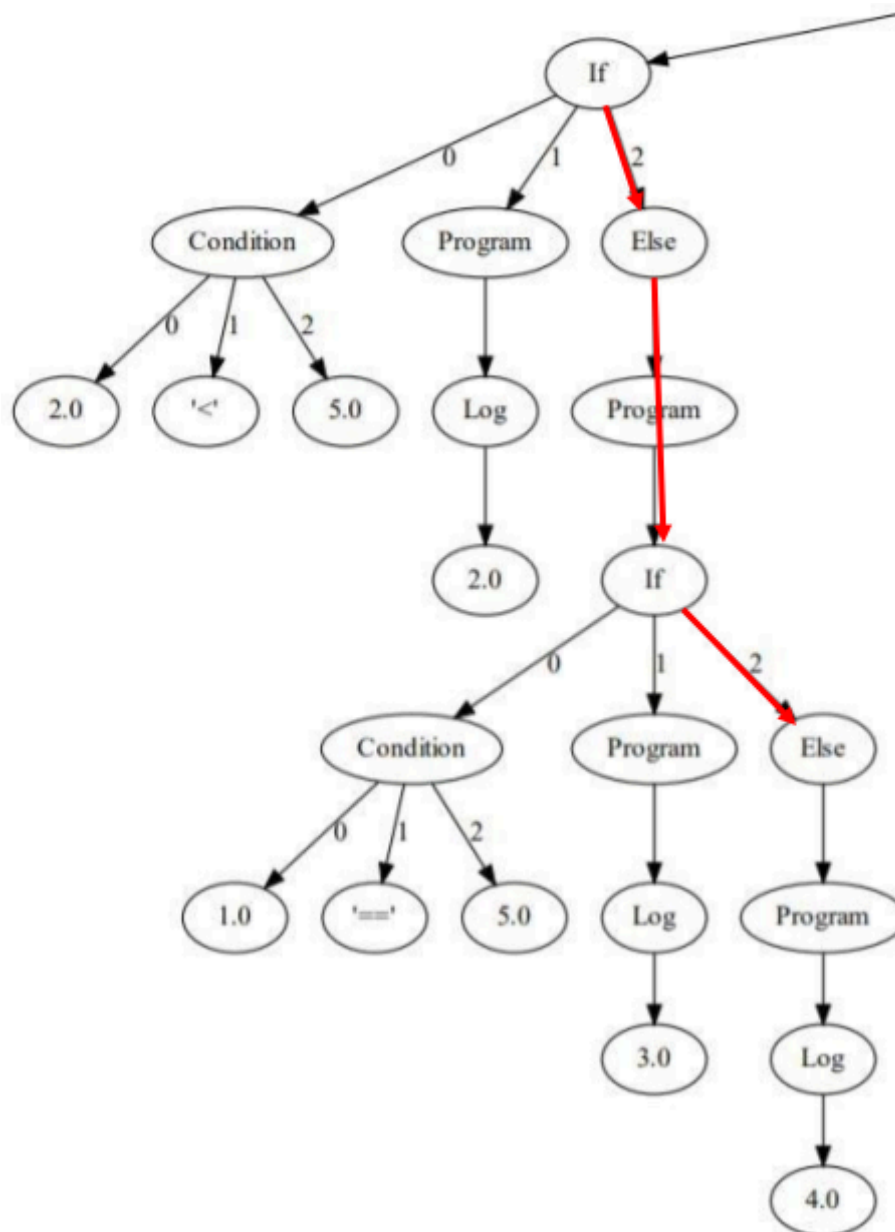


IF/ELSE

En el lexer, hemos añadido estas dos palabras clave, así como los caracteres ? y : para el operador ternario. En el analizador sintáctico, if funciona con o sin llaves. La sutileza y dificultad de esta funcionalidad reside en el hecho de que if/else son recursivos en el caso de if /else if /else por ejemplo. Hemos optado por mostrar un if con dos o tres nodos hijos. El primero es su condición. El segundo es su programa (el entonces). El tercero, que es opcional en el caso de un if solo, es el else. Su hijo es un programa. En el caso de un if else, el if está en el programa else. Como un ejemplo es más significativo, aquí hay uno con :

```
if(2<5){  
    log(2)  
} else if (1==5){  
    log(3)  
} else {  
    log(4)  
}
```

Que da :



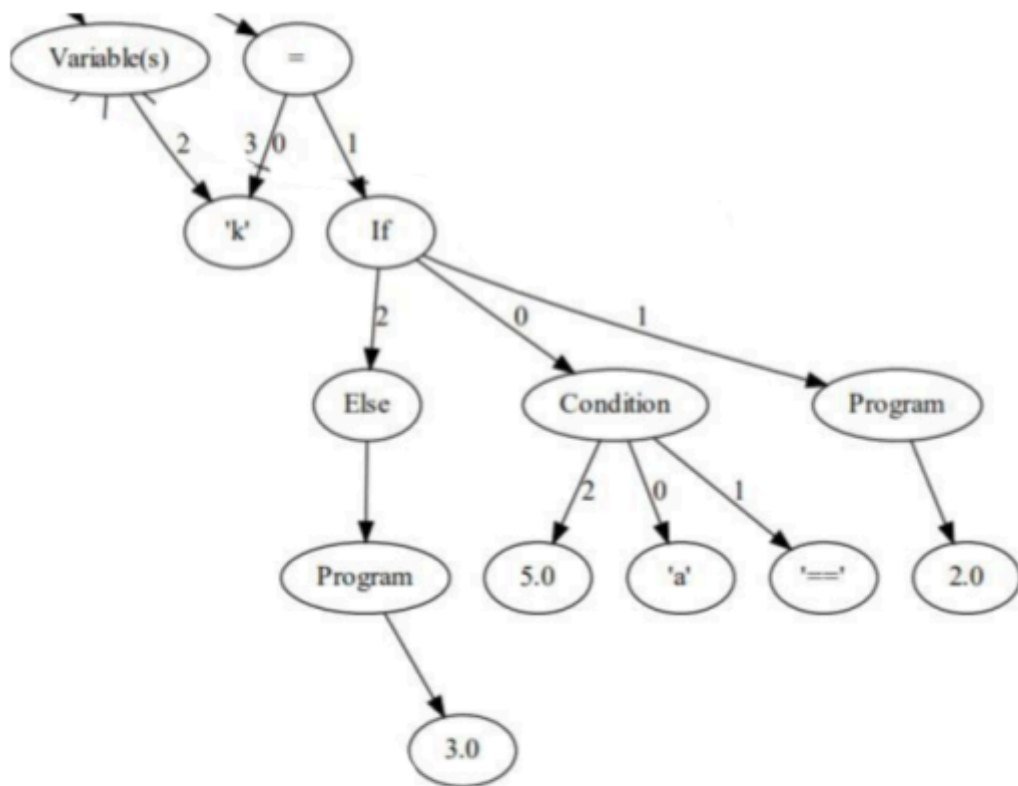
Aquí en rojo está la ruta if/else if/else

Esta recursividad era la fuente de varios errores, particularmente en la versión sin llaves. En la parte de atrás, el if funciona pero el else no, así que esta es una característica que necesita ser mejorada.

OPERADOR TERNARIO

El operador ternario, al igual que los atajos para operaciones y expresiones, es sólo un atajo. Permite utilizar esta notación, pero su estructura de árbol no difiere de la de un if/else.

`var k = (a==5)?2:3;` da por ejemplo :

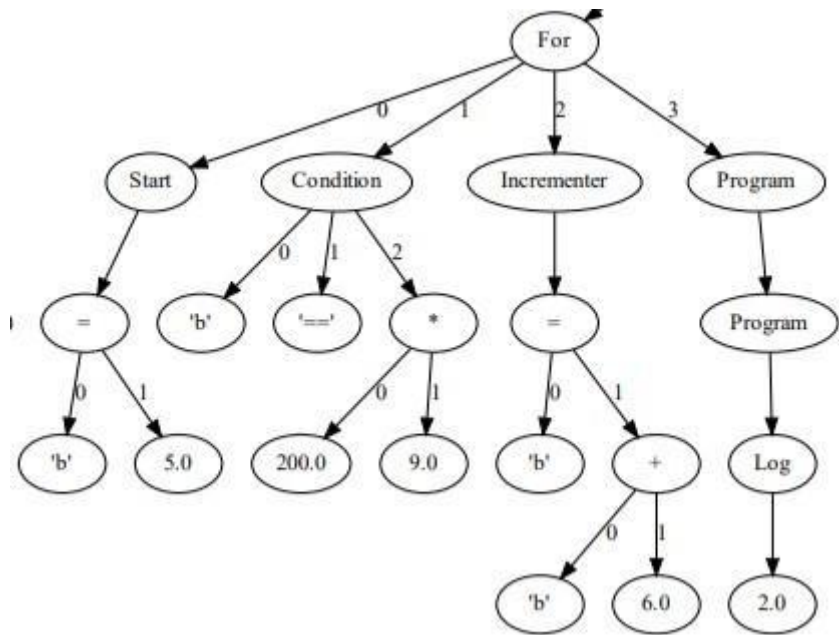


FOR

En la parte del lexer, simplemente se añade la palabra clave. Para este proyecto, hemos implementado el clásico para :

```
def p_for(p):
    '''structure : FOR new_scope '(' assignment ';' condition ';' assignment ')' programBlock
    | FOR new_scope '(' assignment ';' condition ';' assignment ')' programStatement '''
```

Consiste en una asignación de inicio (startForNode en AST), una condición final y un incrementador, y luego su programa. Los hijos del ForNode son, por tanto, estos cuatro nodos. Esto se traduce en la siguiente estructura de árbol:



Para el archivo de entrada :

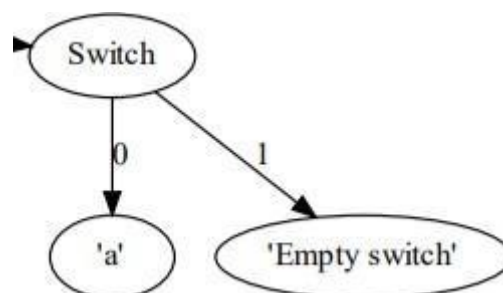
```
for(let b = 5; b==200*9; b+=6)
{
  console.log(2)
}
```

SWITCH

Para esta funcionalidad, añadimos las palabras reservadas SWITCH, CASE y DEFAULT al lexer. Luego, en el analizador sintáctico, ésta es una de las estructuras más complejas. Es fácil olvidar que un switch puede estar vacío:

```
def p_switch_void(p):
    ''' structure : SWITCH '(' IDENTIFIER ')' '{' '}' '''
```

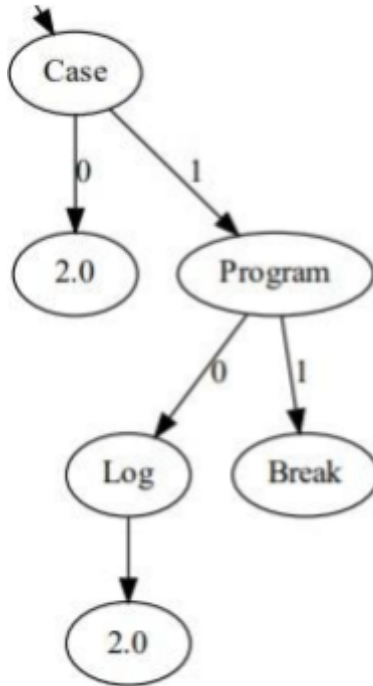
Esto da :



En caso contrario, se compone de CASOS y/o un DEFAULT :

```
def p_switch(p):
    '''structure : SWITCH '(' IDENTIFIER ')' '{' new_scope caseList '}' '''
```

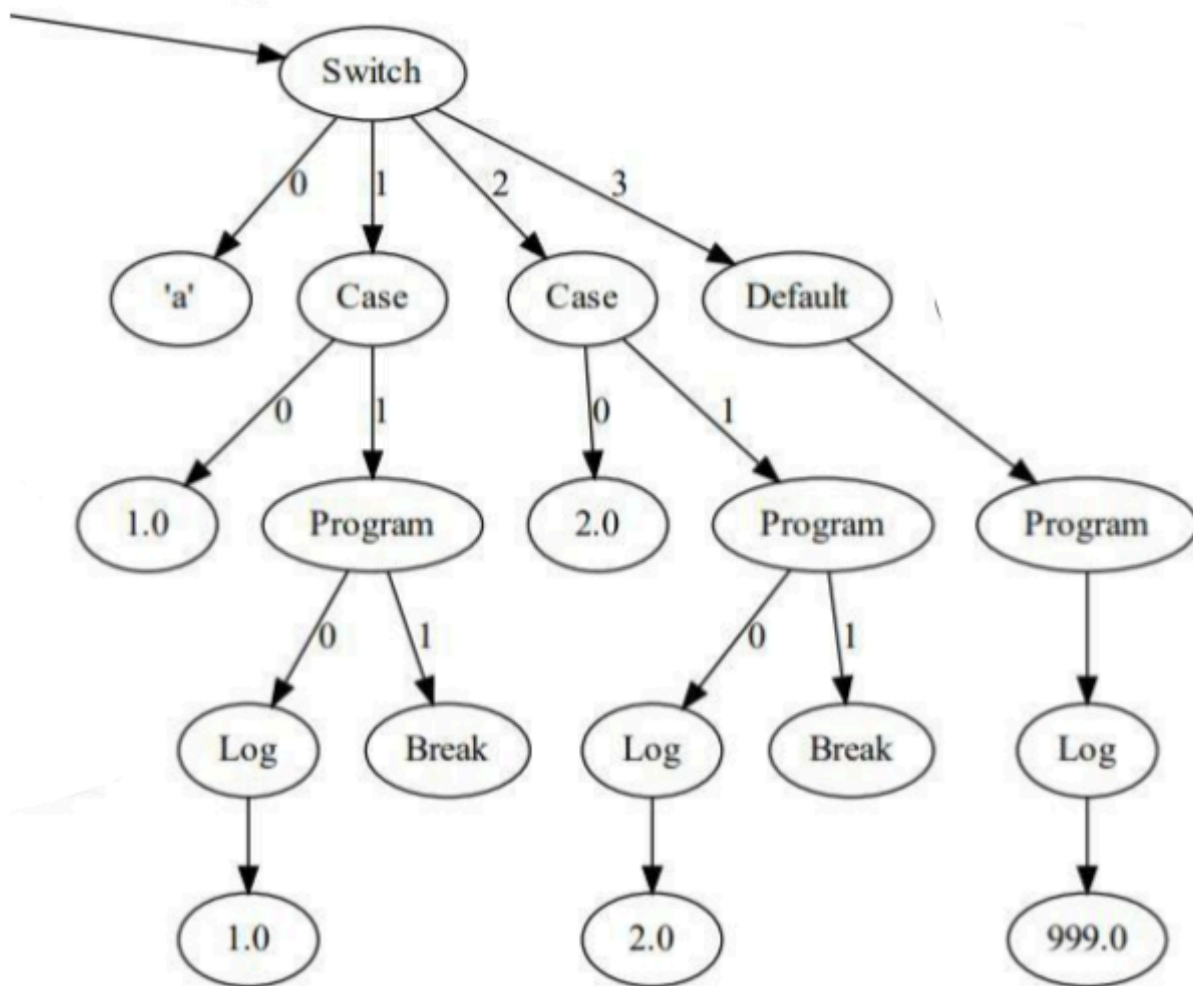
En ambos casos, comprobamos que el IDENTIFICADOR es una variable declarada. La caseList es una secuencia de CASES y DEFAULTS (sin límite de número). Cada caso tiene un número y un programa asociado:



Un valor por defecto sólo tiene un programa. Como se ha indicado anteriormente, no hay límite por defecto en nuestro analizador sintáctico. Por lo tanto, utilizamos un análisis semántico en el AST para comprobar que sólo hay uno o ningún defecto en cada SwitchNode:

```
def verifyDefault(self):
    return len([child for child in self.children if child.type == 'Default']) < 2
```

Resultado total de un cambio:



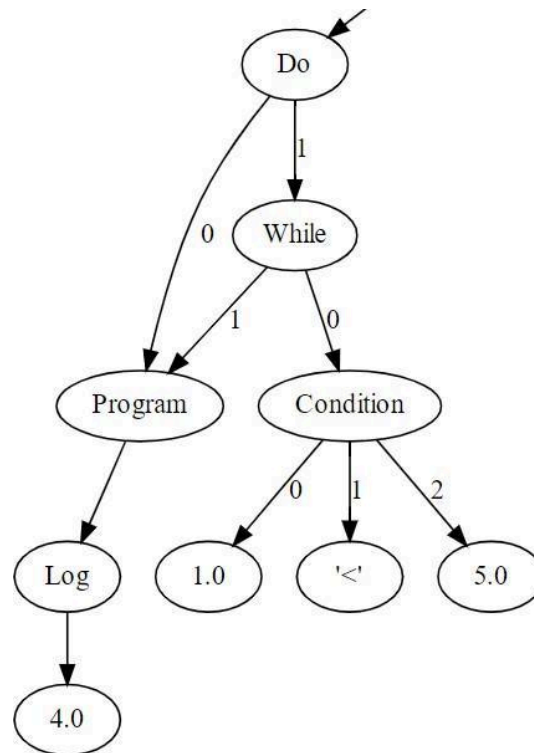
El interruptor está situado en la parte trasera.

DO WHILE

El "do" se añade al lexer y entonces se permiten dos sintaxis en el parser: con o sin llaves. Aquí está la sintaxis con llaves:

```
def p_do_while(p):
    '''structure : DO programBlock WHILE '(' condition ')' '''
```

El árbol de salida es un while normal con sólo un nodo Do que va directamente al nodo programa para la primera iteración, antes de que se ejecute el while:



El do while, al igual que el while, se ha implementado en el back end.

VAR Y LET

En javascript, las variables pueden declararse utilizando dos palabras clave, var y let, que se utilizan indistintamente en nuestro programa y se añaden a las palabras clave reservadas en nuestro lexer. En el analizador sintáctico, se utilizan en diferentes sintaxis. Se utilizan para crear una variable o varias, por ejemplo var a o let a,b. Así que usamos una lista de variables recursiva:

```
def p_var_creation_list_recursive(p):
    '''varList : varList ',' IDENTIFIER'''
```

A las variables se les pueden asignar valores numéricos, pero también matrices de valores y valores dados por funciones (veremos esto con más detalle en los capítulos dedicados a ello).

ÁMBITO DE LAS VARIABLES

Las variables requieren un análisis semántico. Las variables están sujetas a un ámbito. A diferencia del javascript real, nuestro lenguaje no tiene esto en cuenta:

```
Let i=2 ;
If(1<5){
    Let i=1 ;
    Console.log(i)
}
```

Aquí, en javascript, esto mostraría 1, pero para nosotros se lanza un error porque el usuario está redefiniendo i. Por lo tanto, es necesario tener un nombre de variable diferente en este caso. Aparte de eso, al igual que con javascript, hemos hecho que sea imposible llamar a una variable en un ámbito más amplio que aquel en el que fue creada. Para ello utilizamos una característica interesante que nos han sugerido desde [:https://www.dabeaz.com/ply/ply.html](https://www.dabeaz.com/ply/ply.html) (capítulo 6.11) donde utilizamos este :

```
def p_new_scope(p):  
    '''new_scope : '''  
    listScope.append(Scope())
```

En las estructuras en las que cambiamos de ámbito, ponemos este identificador "new_scope". Cada vez que se encuentra, lleva a esta función, que añade un nuevo objeto Scope a una lista de ámbitos. Este objeto scope contiene una lista de todos los nombres de variables (y nombres de funciones) de los scopes que lo engloban. Una vez completado, llamamos a la función pop Scope para eliminar el scope completado, es decir, al final de cada bloque de instrucciones entre llaves. Luego, cada vez que creamos una nueva variable, comprobamos que no exista ya, y la añadimos a la lista del scope actual. Ejemplo con un programBlock :

```
def p_program_block(p):  
    ''' programBlock : '{' new_scope program '}' '''  
    p[0] = p[3]  
    popscope()
```

Ejemplo de creación de variables :

```
def p_var_creation(p):  
    '''varCreation : VAR IDENTIFIER  
    | LET IDENTIFIER'''  
    if p[2] not in listScope[-1 if len(listScope)>1 else 0].vars:  
        listScope[-1 if len(listScope)>1 else 0].vars.append(p[2])  
        p[0] = AST.VariableNode([AST.TokenNode(p[2])])  
    else :  
        error = True  
        print(f"ERROR : {p[2]} is already declared")
```

REGLAS PERMISIVAS PARA EL PUNTO Y COMA

Otra dificultad con javascript es que él es opcional pero permite poner dos instrucciones en la misma línea. En TP4, se nos ocurrió una solución utilizando punto y coma, pero adolece de dos problemas. En primer lugar, la última instrucción de la línea

La solución más sencilla es borrarlo antes de analizar el programa:

```
if program[-1]==';': #allow to finish with a ;  
    program=program[:-1]
```

Entonces, en el mismo estilo, no se podía terminar con un punto y coma en un bloque de instrucciones como en if o while. Aquí la solución es la misma: borrar el punto y coma cada vez antes de parsear. El problema es que puede haber espacios entre el punto y coma y la llave, y nuestro tratamiento de errores, que indica el número de línea del error, significa que no podemos cambiar el número de saltos de línea en relación con el fichero original. Por lo tanto, utilizamos esta solución:

```
return yacc.parse(re.sub(r";(\n)+", lambda x : "\n"*(len(x.group())-2)+"}", program)+"\n")
```

Aquí, vamos a explicar esta línea paso a paso. En primer lugar, esta es nuestra última línea de la función parser, por lo que devuelve nuestro archivo parseado usando yacc. Sin embargo, se realizan dos operaciones sobre el fichero a parsear. La más sencilla es añadir un `\n` al final de la línea, ya que si nuestro programa termina sin una nueva línea, el parser no funcionará (como ocurría en TP4 si terminábamos con un punto y coma). A continuación, el último punto y coma de cada :

```
re.sub(r";(\n)+", lambda x : "\n"*(len(x.group())-2)+"}", program)
```

Utilizamos `re.sub()` para sustituir una expresión regular por una función lambda. Usando la expresión regular, buscamos todos los puntos y comas separados sólo por retornos de carro y una llave de cierre. Sustituimos cada una de estas expresiones por el mismo número de retornos de carro y la llave de cierre, es decir, sólo eliminamos el punto y coma. De este modo, conseguimos que el punto y coma pueda terminar en cualquier parte, sin que ello afecte a nuestro tratamiento de errores.

Ahora que el punto y coma funciona en todos los casos, sigue habiendo expresiones sin punto y coma. Para ello, hemos redefinido `NEWLINE` en el archivo :

```
def t_NEWLINE(t):  
    r'\n+'  
    t.lexer.lineno += len(t.value)  
    return t
```

Lo utilizamos en el analizador sintáctico, en particular con los archivos :


```
def p_program(p):
    ''' program : statement ';'
    | statement NEWLINE '''
    p[0] = AST.ProgramNode(p[1])

def p_program_recursive(p):
    ''' program : statement ';' program
    | statement NEWLINE program '''
    p[0] = AST.ProgramNode([p[1]]+p[3].children)
```

BREAK Y CONTINUE

Añadimos estas dos palabras clave al lexer. Luego las colocamos en su lugar en el analizador sintáctico: continue y break son palabras que se usan solas, creamos nodos AST para ellas y ya está:

```
def p_break(p):
    ''' breakStatement : BREAK '''
    p[0] = AST.BreakNode()

def p_continue(p):
    ''' continueStatement : CONTINUE '''
    p[0] = AST.ContinueNode()
```

Por desgracia, no es tan sencillo. Sintácticamente, es correcto, pero semánticamente, un break y un continue están necesariamente en bucles (o switch para el break).

Por lo tanto, es necesario realizar un análisis semántico. Para comprobar si hay un bucle en los nodos padre, necesitamos recorrer un árbol finito. Así que ejecutamos esta comprobación en main : `if AST.verifyNode():` que llama a la función que ejecuta `verifyBreakContinueNode()`.

Se empieza creando una lista de todos los nodos de interrupción, luego una de todos los nodos de continuación, luego una de todos los nodos de bucle y una de todos los conmutadores. A continuación se comprueba que todos los nodos continue son hijos de los bucles:

```

for continueNode in continueNodes:
    nodeVerified = False
    for loopNode in loopNodesToCheck:
        if checkInChildren(loopNode,continueNode):
            nodeVerified = True
    if not nodeVerified:
        print("ERROR : Continue Node outside of a loop")
        return False

```

Entonces, si todos los nodos de ruptura están en bucles o interruptores :

```

nodesToCheck = loopNodesToCheck+switchNodes

for breakNode in breakNodes:
    nodeVerified = False
    for nodeToCheck in nodesToCheck:
        if checkInChildren(nodeToCheck,breakNode):
            nodeVerified = True
    if not nodeVerified:
        print("ERROR : Break Node outside of a loop or switch")
        return False

```

En caso contrario, aparecerán mensajes de error y no se generará el PDF.

La función continue funciona en la parte posterior, corresponde a un JMP a la última condición pero no se ha podido establecer la ruptura.

TABLAS

Hemos creado dos tipos de matrices, matrices de números: [2,3,5,6,4] y matrices de identificadores, para imitar parcialmente una matriz de cadenas: [a,r,t,e]. Un array puede estar vacío o lleno:

```

def p_array_empty(p):
    '''arrayDeclaration : '[' ']' '''

```

```

def p_array_declaration(p):
    '''arrayDeclaration : '[' tokenList ']' '''

```

Ensuite on peut appeler un élément du tableau :

```

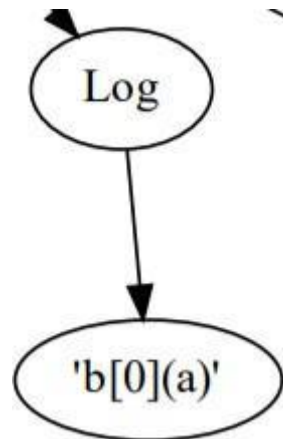
def p_array_access(p):
    ''' expression : IDENTIFIER '[' NUMBER ']' '''

```

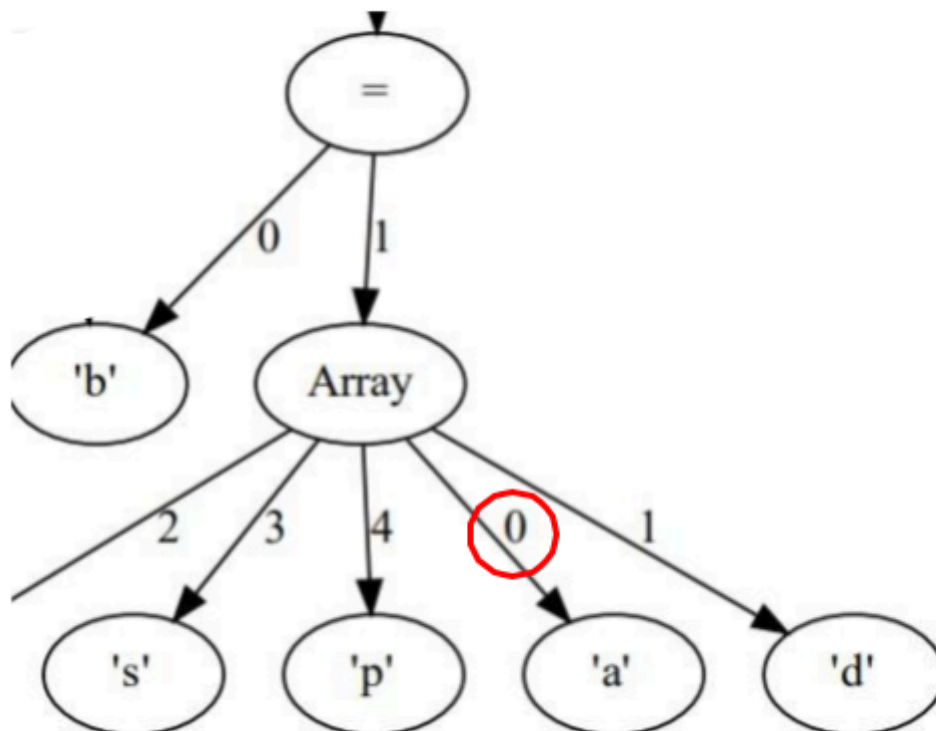
Aquí hay un análisis semántico, comprobamos que el array está declarado antes de llamarlo, que el número entre paréntesis es un entero, y que este índice no es mayor que el tamaño del array. Para ello, recuperamos el nodo correspondiente al array en AST :

```
def getArrayNodeById(id):
```

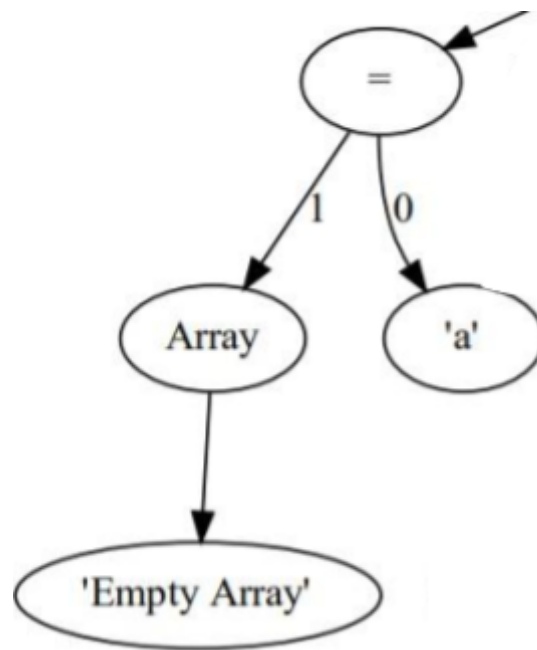
Esto también nos permite recuperar el valor de esta llamada, que da :



Aquí, b[0] es a, lo que es correcto dada la declaración de b :



En el caso de una tabla vacía :



FUNCIONES

Las funciones eran el objetivo principal de los objetivos secundarios. En el lexer, añadimos la palabra clave function. En el analizador sintáctico, en primer lugar, se implementó la declaración, con argumentos :

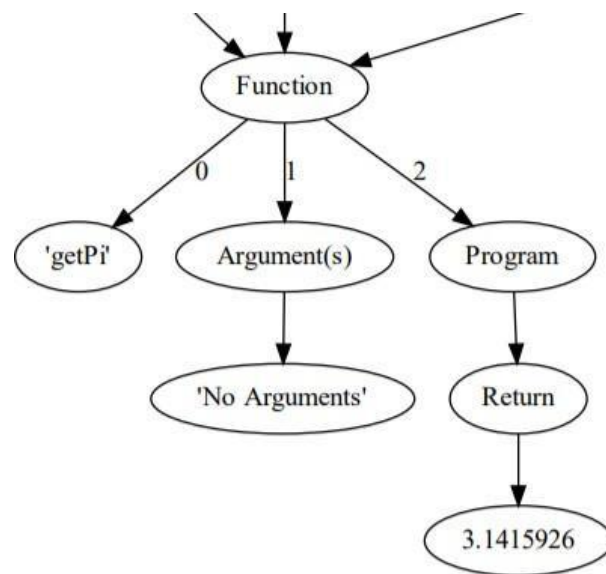
```
def p_function_creation(p):  
    '''functionDeclaration : FUNCTION IDENTIFIER '(' new_scope argList ')' programBlock'''
```

O sin argumentos :

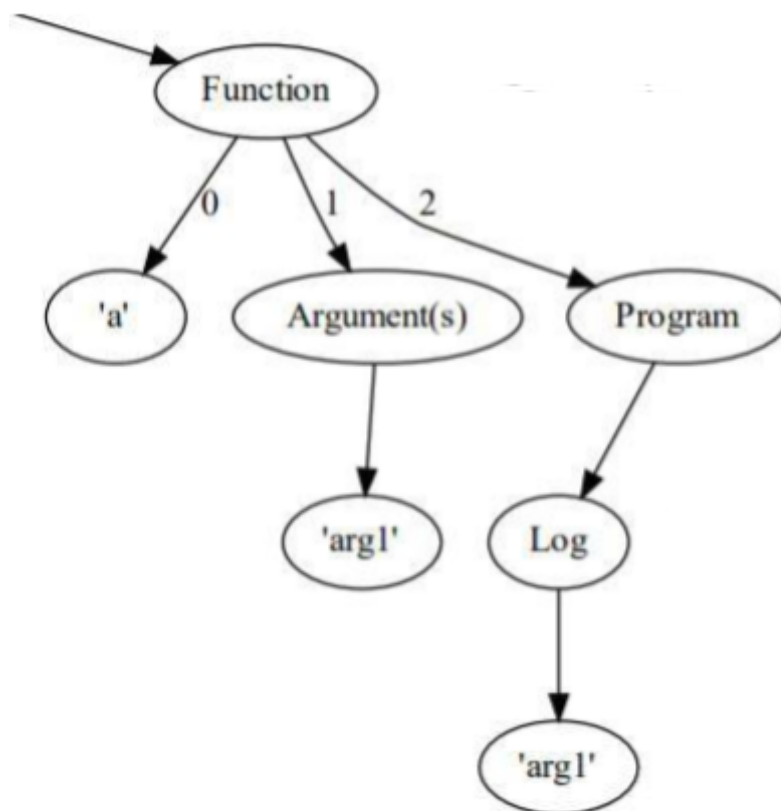
```
def p_function_creation_without_arg(p):  
    '''functionDeclaration : FUNCTION IDENTIFIER '(' ')' programBlock'''
```

En javascript, una función no tiene tipo de retorno y sus argumentos no tienen tipo. Dos funciones no pueden tener el mismo nombre, igual que las variables, pero sólo se puede acceder a ellas en su ámbito, igual que a las variables. Cada vez que declaras una función, compruebas que no exista ya. argList es una secuencia recursiva de argumentos(IDENTIFICAR) separados por comas.

Al declarar una función sin argumentos se obtiene :



Y con argumentos :



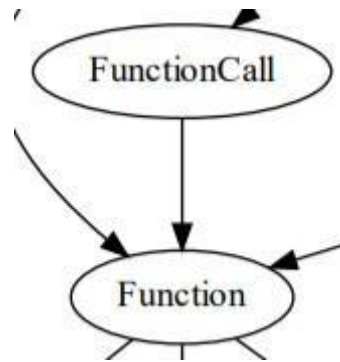
En segundo lugar, una función puede invocarse sin argumentos:

```
def p_function_call_withous_args(p):  
    '''functionCall : IDENTIFIER '(' ')' '''
```

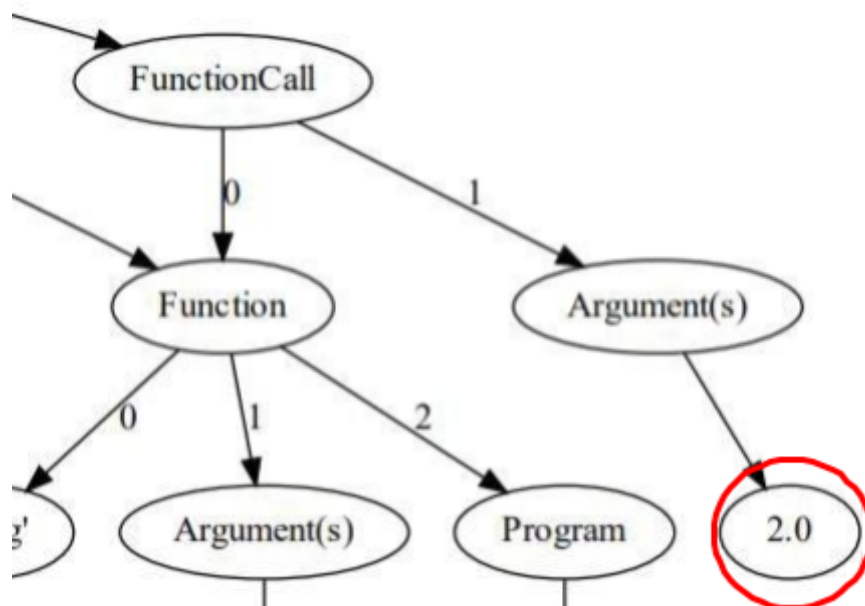
O con argumentos :

```
def p_function_call(p):  
    '''functionCall : IDENTIFIER '(' expressionList ')' '''
```

En el árbol, esto da un functionCallNode que es un padre del nodo de función creado en la declaración: sin argumentos :



O con argumentos :



En el back-end, la gestión de funciones no está totalmente implementada y, por tanto, no es funcional. Sin embargo, se han hecho algunos progresos, como el manejo de las llamadas a funciones y la gestión de los retornos en una función. El principal obstáculo para completar esta parte es el mecanismo de terminación de funciones. Se ha añadido una nueva palabra clave ("EF") al svm para marcar el final del bloque de ejecución de una función; pero la implementación de EF en el svm no es funcional.

ANÁLISIS SEMÁNTICO DEL NÚMERO DE ARGUMENTOS

Como vimos anteriormente, cualquier función puede ser llamada con o sin argumentos, pero en javascript esto no es así. Por eso hemos establecido un análisis semántico en el que se comprueba el número de argumentos necesarios para llamar a la función:

```
functionCallNode = AST.getFunction(p[1])
if functionCallNode.children[0].verifyArgumentsNumber(len(p[3].children)):
```

Aquí p[1] es el nombre de la función y p[3] es la lista de argumentos. Llama a esta función en el FunctionNode :

```
class FunctionNode(Node):
    type = 'Function'
    def verifyArgumentsNumber(self,nb):
        if nb == 0:
            return self.children[1].children[0].tok == 'No Arguments'
        return len(self.children[1].children) == nb and self.children[1].children[0].tok != 'No Arguments'
```

Si p[3] está vacío, comprobamos que la función llamada no tenía argumentos. Si p[3] es un número, comprobamos que la función llamada tiene el mismo número de argumentos y que esta función sí tenía argumentos. De hecho, si p[3] = 1, y la función no tiene argumentos, el nodo 'Sin argumentos' contaría como uno y el resultado estaría distorsionado.

RETURN

return es una palabra reservada (añadida en el lexer) que puede ir seguida o no de una expresión. Puede utilizarse simplemente para detener una función sin devolver nada:

```
def p_return(p):
    '''returnStatement : RETURN '''
    p[0] = AST.ReturnNode()
```

Si no, en nuestro caso, podemos devolver un array, una expresión, otra función o una condición (dando un booleano):

```
def p_return_expression(p):
    '''returnStatement : RETURN returnValues
    | RETURN condition'''
```

Avec :

```
def p_return_values(p):
    '''returnValues : expression
    | arrayDeclaration
    | functionCall'''
```

El return requiere un análisis semántico, ya que nunca debe situarse fuera de una función. De la misma forma que para break y continue, una vez completado el árbol comprobamos si todos los nodos return están debajo de un nodo de función con

```
def verifyReturnNode():
```

que se coloca en el archivo AST y se ejecuta al mismo tiempo que las comprobaciones break y continue. Si uno de los nodos de retorno está mal colocado, se lanza un error y se muestra un mensaje.

Como se ha visto anteriormente, nuestro lenguaje permite hacer un `var a = getPi()` por ejemplo. En el caso de que `getPi()` no tenga valor de retorno, javascript simplemente devuelve "Undefined" pero no genera ningún error. Por eso, al igual que javascript, no comprobamos si la función a la que llamamos devuelve algo.

GESTIÓN DE ERRORES

En nuestro proyecto, los errores se muestran en la consola y se establece una variable de error a True para evitar que el árbol se muestre en la consola y así evitar sobrecargar al usuario con errores en cascada. Utilizamos el sistema de errores configurado para TP4 para los errores de sintaxis con el número de línea correspondiente:

```
def p_error(p):
    error = True
    if p:
        print (f"Syntax error in line {p.lineno} with {p}")
        parser.errok()
    else:
        print ("Sytax error: unexpected end of file!")
```

Pero además, como nuestro análisis semántico se realiza en el mismo archivo, los errores semánticos también se muestran en la consola, por ejemplo, un error de declaración de variable.

REQUERIMIENTOS

GUÍA DEL USUARIO

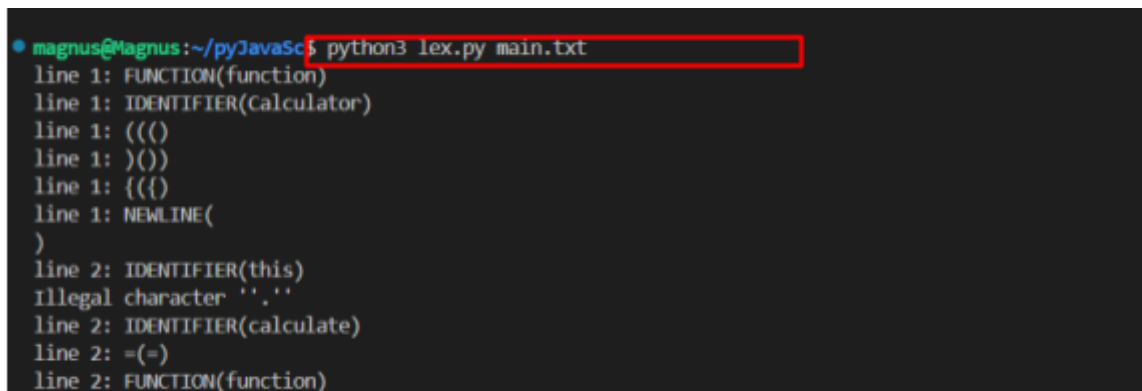
Para lanzar nuestro proyecto, primero debe instalar los módulos PLY y pydot utilizando el comando pip, e instalar Graphviz

INSTALACIÓN

Una vez instalados los módulos y Graphviz, basta con extraer el archivo proporcionado y ejecutarlo desde un terminal. Te recomendamos que utilices powershell, que facilita el acceso a las pruebas gracias a las pestañas que permiten ver lo que hay en una carpeta.

ANALYSE LEXICALE

Exactamente igual que para el trabajo práctico en el que se basa nuestro proyecto, para acceder únicamente al análisis léxico, ejecute el archivo lex.py con el archivo de texto de su elección como argumento, en este caso una de nuestras pruebas preparadas:



```
magnus@Magnus:~/pyJavaSc$ python3 lex.py main.txt
line 1: FUNCTION(function)
line 1: IDENTIFIER(Calculator)
line 1: (((
line 1: ))
line 1: {{{
line 1: NEWLINE(
)
line 2: IDENTIFIER(this)
Illegal character ','
line 2: IDENTIFIER(calculate)
line 2: =(=)
line 2: FUNCTION(function)
```

El resultado se escribe en la consola con su grupo léxico para cada grupo de caracteres si se reconoce, de lo contrario se mostrará un error en su lugar.