Entrega 1: Analizador Léxico

Grupo 127, integrantes:

- Nicolás Cossío Miravalles - n.cossio@alumnos.upm.es - b190082
- Bárbara Rodríguez Ruiz - barbara.rodriguez.ruiz@alumnos.upm.es - b190110
- Huangjue He - h.he@alumnos.upm.es- a180022

Tabla de contenidos:

Diseño del Analizador Léxico

Tokens

Palabras reservadas

```
let : < let, - >
function : <function, - >
return: <return, - >
if: < if, - >
else : < else, - >
input : < input, - >
print : < print, - >
while : <while, - >
do-while : <doWhile, - >
true : <boolT, - >
false : <boolF, - >
int : <int, - >
boolean : <boolean, - >
string : <string, - >
```

Operadores

```
Operadores Aritméticos:
  + : <mas, - >
  * : <por, - >

Operadores Lógicos:
  && : <and, - >

Operadores relacionales:
  == : <equals, - >
  >  : <mayor, - >
```

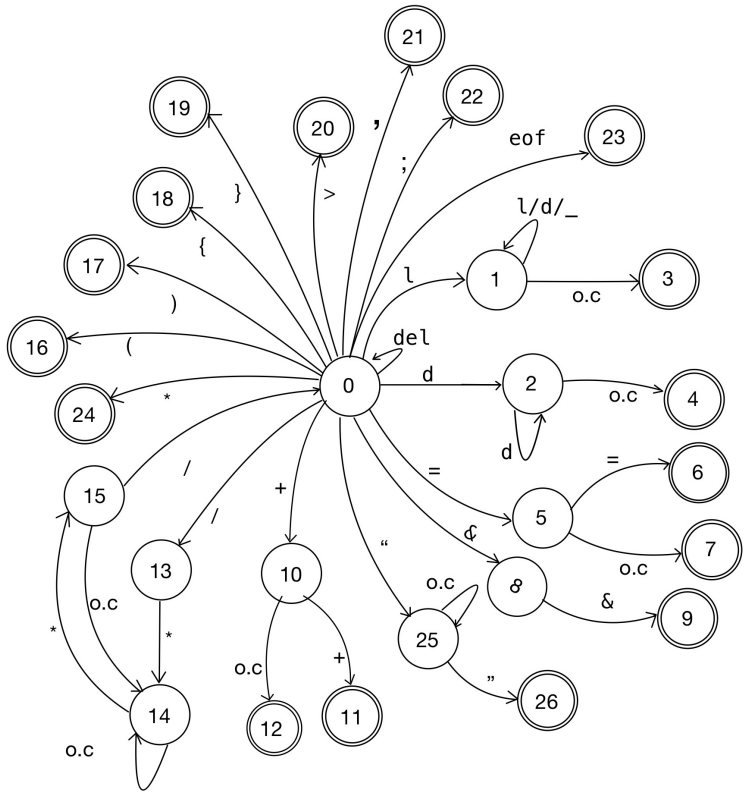
Resto de tokens

```
Identificador :  <id, ptroTS >
Asignacion = : <asig, - >
Cadena: <cadena, laCadena >
Constante Entera : <cteEnt, valor >
++ : <postIncrem, - >
, : <coma, - >
; : <puntoComa, - >
( : <parAbierto, - >
) : <parCerrado, - >
{ : <llaveAbierto, - >
} : <llaveCerrado, - >
eof : <eof, - >
```

Gramática Regular

```
S -> lA | dC | ( | ) | { | } | delS | =E | &D | > | +G | * | , | ; | eof | /B | "J
A -> lA | dA | _A | λ
B -> *H
C -> dC | λ
D -> &
E -> = | λ
G -> + | λ
H -> λH | *I
I -> /S
J -> λJ | ″K
K -> λ
otro indica cualquier otro carácter distinto de *
```

Automata Finito Determinista



- OC: cualquier carácter distinto de los ya especificados para ese estado

Acciones Semánticas

```
- LEER: lee el siguiente carácter del fichero fuente. car := leer()
       En todas las transiciones menos en las transiciones etiquetadas con O.C, excepto en 25->25
- CONC: forma una cadena (lexema). Siempre después de leer(), lex:=lex⊕car
       En las transiciones: 13->14, 14->15 | 0->1, 1->1, x->x //por hacer, cadenas
- VALOR: convierte un carácter a su entero correspondiente, entero:=valor(carácter)
- GENTOKEN: genera un token que el A.Léxico pasa al sintáctico
0->2: num = valor(car)
2->2: num = num*10+valor(car)
2->4: if (num < 32768) genToken(cteEnt, num)
      else error()

25->26:
if (length(lex) < 65) genToken(cadena, lex)
else error()



TEstadosFinales []:=lista de códigos de los tokens de operacion y otros menos los no identificadores, cadena o constante que se gen
eran en los estados finales del autómata, permiten crear un token al usarse como argumento de genToken( )

if transicion is not 1->3, 2->4 {
  a:=TEstadosFinales[numEstadoFinal]
  if a is not null genToken( a , )
}

buscarTPR(lex): devuelve el código del token que coincide con un lexema dado, codigo:=buscarTPR(lex)
TPR []:=lista de códigos de tokens de palabras reservadas que permiten crear un token al usarse como argumento de genToken( )
1->3:
zona_decl := boolean que indica si se trata de una declaracion, global = true y local = false
p:= buscarTPR(lex)
if ( p is not null ) genToken( p , - ) //genera token de palabra reservada
else { //identificador
    p:= buscarTS(lex)
    if ( p is not null ) //ya está declarada
```

```
      genToken(id, p)
    else { //no está declarada
      p:=añadirTS_activa(lexema) //AñadirTS devuelve un ptro. al id
      genToken(id, p)
    }
}
```

Errores

```
ERROR: cualquier transición no recogida en el autómata corresponde a un caso de error.
También son errores todos los lanzados desde las acciones semánticas, que son los siguientes:
 - Entero sobrepasa valor máximo permitido
 - String sobrepasa longitud máxima permitida
```

Tabla de símbolos

```
Todas las tablas tendrán lexema y tipo, pero el resto de los Tributos de la tabla dependeran del tipo.

Para enteros, reales, cadenas, lógicos... tendremos una tabla con el siguiente formato:

TS 1#:

* LEXEMA : 'a'
Atributos:
+ tipo: 'entero'
+ despl : 0

Para un array:

TS 2#:

* LEXEMA : 'a'
Atributos:
+ tipo: 'entero'
+ despl : 0
? núm de dimensiones, límite inf y límte sup de cada dimensión,...

Para funciones la tabla seguirá el formato:

TABLA FUNCION SUMA #3:

*LEXEMA : 'suma'
Atributos:
+ tipo: 'funcion'
+ numParam: 2
+ TipoParam01: 'ent'
+ ModoParam1: 1 (es por valor)
+ TipoParam2: 'real'
+ ModoParam02: 2 (por referencia)
+ TipoRetorno: 'ent'
+ EtiqFuncion: 'Etsuma01’
```

Código

Analizador Léxico

```
import sys
import os

if len(sys.argv) > 2: sys.exit("Input file path to analyze as program argument")
#manual path specification for debugging puropses
if len(sys.argv) == 1: f = open("./casosPruebas/correcto1.txt" ,"r")
#file that the lexer will generate tokens for
elif os.path.exists(os.getcwd() + "/"+ sys.argv[1]): f = open(sys.argv[1], "r")
else: sys.exit(f"File \'{sys.argv[1]}\' does not exist")
#Lenguage definitions by class
RES_WORD = [ "let","function","rn","else","input","print","while","do","true","false","int","boolean","string"]
LETTERS = "abcdefghijlmnñopqrstuvwxyzABCDEFGHIJKLMNÑOPQRSTUVWXYZ"
SYMB_OPS ={
    "+": "mas",
    "*": "por",
```

```
    "&": "and",
    "=": "asig",
    ">": "mayor",
    ",": "coma",
    ";": "puntoComa",
    "(": "parAbierto",
    ")": "parCerrado",
    "{": "llaveAbierto",
    "}": "llaveCerrado"
}

class Error():
    def __init__(self, num:int, linea:int, attr=None):
        self.code = num
        self.line = linea
        self.att = attr

class Token:
    def __init__(self, type:str, attribute=None):
        self.code = type
        self.att = attribute

class Lexer:
    def __init__(self):
        self.num = 0 #current integer number being constructed
        self.lex = "" #current string being constructed
        self.filename = os.path.basename(f.name)
        self.car = "" #current character being read
        self.outputdir = os.getcwd()+"/" +self.filename+"Output"
        self.line = 1
        self.tokenList = [] #current list of tokens being generated and saved
        self.errorList = []

    def skipBlockComments(self):
        '''Skips block comments and detects error in its specification'''
        self.car = f.read(1)
        if self.car == "*":
            self.car = f.read(1)
            while (self.peekNextCar() != "/" and self.car != "*" and self.car != ""):
                self.car = f.read(1)
                if self.car == "" : self.error(3)
                if self.car == "/n": self.car+=1
            self.car = f.read(1)
            self.car = f.read(1)
        elif self.car == "/":
            f.readline()
            self.line+=1
            self.error(5)
        else: self.error(4, self.car)

    def skipDelimeters(self):
        '''Skips delimeters such as \\t and \\n '''
        if self.car!= "":
            while(self.car != "" and ord(self.car) < 33):
                if self.car == "\n": self.line+=1
                if self.car == "": break#Block comment processing
                self.car = f.read(1)
            if self.car == "/": self.skipBlockComments()

    def next(self):
        '''Retrieves next character recognized in the leanguage for processing'''
        # if self.peekNextCar()!="": self.car = f.read(1)
        self.car = f.read(1)
        if self.car !="":
            if self.car != "/":
               self.skipDelimeters()
            else:
                self.skipBlockComments()
                self.skipDelimeters()

    def generateNumber(self):
        '''Adds current char to number in construction after multiplying the number by 10 '''
        self.num = self.num * 10 + int(self.car)

    def concatenate(self):
        '''Concatenates current char to lexeme in contruction'''
        self.lex += self.car

    def printToken(self):
        '''Creates a directory (specified in self.ouput dir which will contain all the output of the processor.\n
        Writes all tokens with the appropiate format to the file "tokens.txt" after tokenize() has been used'''
        try: os.mkdir(self.outputdir)
        except OSError: f"Error al crear carpeta de volcado en: {self.outputdir}"
        except: FileExistsError
        print(f"Directorio de volcado del programa creado en: {self.outputdir}")
        with open(self.outputdir+"/tokens.txt", "w") as f:
            for token in self.tokenList:
```

```
                f.write(f"< {token.code} , {token.att} >\n".replace("None", " ")) #del* < código del* , del* [atributo] del* > del* RE

    def peekNextCar(self) -> str:
        '''Returns the character next to that which the file pointer is at, without advancing said file pointer'''
        pos = f.tell()
        car = f.read(1)
        f.seek(pos)
        return car

    # < codigo , atributo >
    def genToken(self, code:str, attribute=None) -> Token:
        '''Generates a token and appends it to the list of tokens:\n
        -code: specifies token code (id,string,cteEnt,etc)
        -attribute: (OPTIONAL) specifies an attribute if the token needs it i.e < cteEnt , valor >
        '''
        token = Token(code, attribute)
        self.tokenList.append(token)
        self.lex = ""
        self.num = 0
        return token

    def error(self, tipo: int, attribute=None):
        '''Generates an error and appends it to the list of error:\n
        -tipo: specifies error type. -All error types are specified in the errorHandler class
        -attribute: (OPTIONAL) specifies an attribute if the error needs it i.e symbol which was not recognized in error4
        '''
        err = Error(tipo, self.line, attribute)
        self.errorList.append(err)
        if tipo == 5: self.line+=1
        if tipo == 1: self.lex = ""

    def tokenize(self):
        ''''
        Analyzes characters, generates tokens and errors if found
        '''
        self.next()
        while self.car != "":
            #Integer being formed
            if (self.car).isdigit() and self.lex=="":
                self.generateNumber()
                if(not self.peekNextCar().isdigit()):
                    if(self.num < 32768):
                        self.genToken("cteEnt", self.num)
                    else:
                        self.error(2)

            #Identifiers or Reserved Words
            elif self.car in LETTERS or self.lex != "":
                self.concatenate()
                nextCar = self.peekNextCar()
                if(nextCar not in LETTERS or nextCar == "_" or nextCar.isdigit() or nextCar ==""):
                    if self.lex in RES_WORD:
                        self.genToken(self.lex)
                    else:
                        if(len(self.lex)<65):self.genToken("id", self.lex)
                        else: self.error(1)

            #String (cadena) processing
            elif self.car == "\"" or "":
                self.next()
                while self.car != "\"":
                    self.concatenate()
                    self.car = f.read(1)
                if(len(self.lex)<65): self.genToken("cadena",self.lex)
                else: self.error(1)

            #Operators, symbols
            elif self.car in SYMB_OPS:
                #+ or ++
                if(self.car == "+"):
                    if (self.peekNextCar() == "+") :
                        self.genToken("postIncrem")
                        self.next()
                    else: self.genToken("mas")
                #&&
                elif (self.car == "&"):
                    if (self.peekNextCar() == "&") :
                        self.genToken("and")
                        self.next()
                #= or ==
                elif(self.car == "="):
                    if (self.peekNextCar() == "="):
                        self.genToken("equals")
                        self.next()
                    else:
                        self.genToken("asig")
                else:
```

```
                    self.genToken(SYMB_OPS[self.car])
            else:
                if self.car in "\'":
                    while self.car not in "\'`":
                        self.next()
                    if self.car != "":
                        self.error(6)
                    else: self.error(7)
                else: self.error(4, self.car)
            if self.car != "": self.next()
```

Handler de erores

```
from string import Template
from lexer import *

ERROR_MSG = {
    0:"Uso erroneo de comentario de bloque\n\tFormato:\'/* Comentario de bloque */ \'",
    1:"Lexema excede el tamaño maximo de caracteres permitido",
    2:"Digito con valor mayor al permitido (32768) en el sistema",
    3:"Comentario de bloque no cerrado",
    4:"Simbolo '$simbolo' no pertenece al lenguaje",
    5:"Comentarios de tipo: '//comentario', no estan permitidos",
    6:"Cadena se debe especificar entre \" \", no con \' \'",
    7:"cadena no cerrada"
}
class ErrorHandler():
    def __init__(self, lexer:Lexer):
        self.lexer = lexer

    def errorPrinter (self):
        with open(self.lexer.outputdir+"/errors.txt", "w") as f:
            header = f"Errors output for file '{self.lexer.filename}':\n"
            times = len(header)-1
            header+="-" * times
            header = "-" * times +"\n"+header
            f.write(header)
            for error in self.lexer.errorList:
                errorString =f"\nError code'{error.code}'': {ERROR_MSG[error.code]}"
                if error.code == 4: errorString=Template(errorString).substitute(simbolo = error.att)
                if error.code == 7:errorString+=ERROR_MSG[7]
                errorString+= f"--> linea {error.line}"
                f.write(errorString)
```

Tabla de Símbolos

```
'''
This file will contain all parts of the lenguage processor from
which its services will be launched to perform their tasks
'''
from errorHandler import ErrorHandler
from lexer import Lexer
from tablaSimbolos import TS
from errorHandler import ErrorHandler
lexer = Lexer()
lexer.tokenize()
lexer.printToken()
ts = TS(lexer)
ts.printTS()
errorHandler = ErrorHandler(lexer)
errorHandler.errorPrinter()
```

Procesador Javascript PDL

```
'''
This file will contain all parts of the lenguage processor from
which its services will be launched to perform their tasks
'''
from errorHandler import ErrorHandler
from lexer import Lexer
from tablaSimbolos import TS
from errorHandler import ErrorHandler
lexer = Lexer()
lexer.tokenize()
lexer.printToken()
ts = TS(lexer)
ts.printTS()
```

```
errorHandler = ErrorHandler(lexer)
errorHandler.errorPrinter()
```

Anexo - Casos de prueba

Casos correctos

> Los siguientes códigos son correctos según nuestro analizador léxico

Caso 1

```
function salto string (){
  return "\n";
}

function imprime (string s, string msg, int f){ /* función que recibe 3 argumentos */
  print(s);
  print(msg);
  print(f);
  print (salto());  /* imprime un salto de línea */
  return; /* finaliza la ejecución de la función (en este caso, se podría omitir) */
}
```

Fichero de tokens:

```
< function ,   >
< id , imprime >
< parAbierto ,   >
< string ,   >
< id , s >
< coma ,   >
< string ,   >
< id , msg >
< coma ,   >
< int ,   >
< id , f >
< parCerrado ,   >
< llaveAbierto ,   >
< print ,   >
< parAbierto ,   >
< id , s >
< parCerrado ,   >
< puntoComa ,   >
< print ,   >
< parAbierto ,   >
< id , msg >
< parCerrado ,   >
< puntoComa ,   >
< print ,   >
< parAbierto ,   >
< id , f >
< parCerrado ,   >
< puntoComa ,   >
< print ,   >
< parAbierto ,   >
< id , salto >
< parAbierto ,   >
< parCerrado ,   >
< parCerrado ,   >
< puntoComa ,   >
< cteEnt , 1 >
< llaveCerrado ,   >
< function ,   >
< id , salto >
< string ,   >
< parAbierto ,   >
< parCerrado ,   >
< llaveAbierto ,   >
< id , return >
< cadena , hola >
< puntoComa ,   >
< llaveCerrado ,   >
```

Fichero de TS:

```
TS GLOBAL #1
*Lexema: 'salto'
*Lexema: 'imprime'
*Lexema: 's'
*Lexema: 'msg'
*Lexema: 'f'
```

Fichero de errores:

```
---------------------------------------
Errors output for file 'correcto1.txt':
---------------------------------------
```

Caso 2

```
function potencia (int z, int dim)  {
  let int s;  /* Oculta a la global */
  int s = 0;
  do{
    z *= z;
    print("Potencia: ");
    print(z);
    s++;
  }while(s>dim)
} /* fin de potencia: función que no devuelve nada */
```

Fichero de tokens:

```
< function ,   >
< id , potencia >
< parAbierto ,   >
< int ,   >
< id , z >
< coma ,   >
< int ,   >
< id , dim >
< parCerrado ,   >
< llaveAbierto ,   >
< let ,   >
< int ,   >
< id , s >
< puntoComa ,   >
< int ,   >
< id , s >
< asig ,   >
< cteEnt , 0 >
< puntoComa ,   >
< do ,   >
< llaveAbierto ,   >
< id , z >
< por ,   >
< asig ,   >
< id , z >
< puntoComa ,   >
< print ,   >
< parAbierto ,   >
< cadena , Potencia:  >
< parCerrado ,   >
< puntoComa ,   >
< print ,   >
< parAbierto ,   >
< id , z >
< parCerrado ,   >
< puntoComa ,   >
< id , s >
< postIncrem ,   >
< puntoComa ,   >
< llaveCerrado ,   >
< while ,   >
< parAbierto ,   >
< id , s >
< id , dim >
< parCerrado ,   >
< llaveCerrado ,   >
```

Fichero de TS:

```
TS GLOBAL #1
*Lexema: 'potencia'
*Lexema: 'z'
*Lexema: 'dim'
*Lexema: 's'
```

Fichero de errores:

```
---------------------------------------
Errors output for file 'correcto1.txt':
---------------------------------------
```

Caso 3

```
function demo ()  { /* definición de la función demo, sin argumentos y que no devuelve nada */
  let int v1; let int v2; let int v3;
  let int zv;

  print ("Escriba tres números: ");
  input (v1); input (v2); input (v3);

  if (v3 == 0) return;

  if (v1 == v2){
    print ("Escriba su nombre: ");
    let string s;
    input (s);
    if (v2 > v3){
```

```
      let int v0 = v2;
    }
    else{
      v0 = v3;
    }
    print (s);
  }
  s = "El primer valor era ";
  if (v1 == 0){
    print (s);
    print (v1);
    print (".\n");
  }
  else{
    print (s);
    print (".\n");
  }

  potencia (v0, 4);
  potencia (zv, 5);
}

demo(); /*El funcionamiento de la función demo no es correcta y el algoritmo es solo representativo*/
```

Fichero de tokens:

```
< function ,   >
< id , demo >
< parAbierto ,   >
< parCerrado ,   >
< llaveAbierto ,   >
< let ,   >
< int ,   >
< id , v >
< cteEnt , 1 >
< puntoComa ,   >
< let ,   >
< int ,   >
< id , v >
< cteEnt , 2 >
< puntoComa ,   >
< let ,   >
< int ,   >
< id , v >
< cteEnt , 3 >
< puntoComa ,   >
< let ,   >
< int ,   >
< id , zv >
< puntoComa ,   >
< print ,   >
< parAbierto ,   >
< cadena , Escriba tres números:  >
< parCerrado ,   >
< puntoComa ,   >
< input ,   >
< parAbierto ,   >
< id , v >
< cteEnt , 1 >
< parCerrado ,   >
< puntoComa ,   >
< input ,   >
< parAbierto ,   >
< id , v >
< cteEnt , 2 >
< parCerrado ,   >
< puntoComa ,   >
< input ,   >
< parAbierto ,   >
< id , v >
< cteEnt , 3 >
< parCerrado ,   >
< puntoComa ,   >
< id , if >
< parAbierto ,   >
< id , v >
< cteEnt , 3 >
< equals ,   >
< cteEnt , 0 >
< parCerrado ,   >
< return ,   >
< puntoComa ,   >
< id , if >
```

Fichero de TS:

```
TS GLOBAL #1
*Lexema: 'demo'
*Lexema: 'v'
*Lexema: 'zv'
*Lexema: 'if'
*Lexema: 's'
*Lexema: 'potencia'
```

Fichero de errores:

```
---------------------------------------
Errors output for file 'correcto3.txt':
---------------------------------------
```

```
< parAbierto ,   >
< id , v >
< cteEnt , 1 >
< equals ,   >
< id , v >
< cteEnt , 2 >
< parCerrado ,   >
< llaveAbierto ,   >
< print ,   >
< parAbierto ,   >
< cadena , Escriba su nombre:  >
< parCerrado ,   >
< puntoComa ,   >
< let ,   >
< string ,   >
< id , s >
< puntoComa ,   >
< input ,   >
< parAbierto ,   >
< id , s >
< parCerrado ,   >
< puntoComa ,   >
< id , if >
< parAbierto ,   >
< id , v >
< cteEnt , 2 >
< mayor ,   >
< id , v >
< cteEnt , 3 >
< parCerrado ,   >
< llaveAbierto ,   >
< let ,   >
< int ,   >
< id , v >
< cteEnt , 0 >
< asig ,   >
< id , v >
< cteEnt , 2 >
< puntoComa ,   >
< llaveCerrado ,   >
< else ,   >
< llaveAbierto ,   >
< id , v >
< cteEnt , 0 >
< asig ,   >
< id , v >
< cteEnt , 3 >
< puntoComa ,   >
< llaveCerrado ,   >
< print ,   >
< parAbierto ,   >
< id , s >
< parCerrado ,   >
< puntoComa ,   >
< llaveCerrado ,   >
< id , s >
< asig ,   >
< cadena , El primer valor era  >
< puntoComa ,   >
< id , if >
< parAbierto ,   >
< id , v >
< cteEnt , 1 >
< equals ,   >
< cteEnt , 0 >
< parCerrado ,   >
< llaveAbierto ,   >
< print ,   >
< parAbierto ,   >
< id , s >
< parCerrado ,   >
< puntoComa ,   >
< print ,   >
< parAbierto ,   >
< id , v >
< cteEnt , 1 >
< parCerrado ,   >
< puntoComa ,   >
< print ,   >
< parAbierto ,   >
< cadena , .\n >
< parCerrado ,   >
< puntoComa ,   >
< llaveCerrado ,   >
< else ,   >
< llaveAbierto ,   >
< print ,   >
```

```
< parAbierto ,   >
< id , s >
< parCerrado ,   >
< puntoComa ,   >
< print ,   >
< parAbierto ,   >
< cadena , .\n >
< parCerrado ,   >
< puntoComa ,   >
< llaveCerrado ,   >
< id , potencia >
< parAbierto ,   >
< id , v >
< cteEnt , 0 >
< coma ,   >
< cteEnt , 4 >
< parCerrado ,   >
< puntoComa ,   >
< id , potencia >
< parAbierto ,   >
< id , zv >
< coma ,   >
< cteEnt , 5 >
< parCerrado ,   >
< puntoComa ,   >
< llaveCerrado ,   >
< id , demo >
< parAbierto ,   >
< parCerrado ,   >
< puntoComa ,   >
```

Casos incorrectos

Caso 1

> Este caso es incorrecto ya que usa operadores no admitidos en nuestro sistema de tokens, por ejemplo: %, ||, !=, al igual que los comentarios con //.

```
function bisiesto boolean (int a) {
  return (a % 4 == 0 && a % 100 != 0 || a % 400 == 0);  //se tienen en cuenta la precedencia de operadores
} // fin de bisiesto: función lógica
```

Fichero de tokens:

```
< function ,   >
< id , bisiesto >
< boolean ,   >
< parAbierto ,   >
< int ,   >
< id , a >
< parCerrado ,   >
< llaveAbierto ,   >
< return ,   >
< parAbierto ,   >
< id , a >
< cteEnt , 4 >
< equals ,   >
< cteEnt , 0 >
< and ,   >
< id , a >
< cteEnt , 100 >
< asig ,   >
< cteEnt , 0 >
< id , a >
< cteEnt , 400 >
< equals ,   >
< cteEnt , 0 >
< parCerrado ,   >
< puntoComa ,   >
< llaveCerrado ,   >
```

Fichero de TS:

```
TS GLOBAL #1
*Lexema: 'bisiesto'
*Lexema: 'a'
```

Fichero de errores:

```
----------------------------------------
Errors output for file 'incorrecto1.txt':
----------------------------------------
Error code'4'': Simbolo '%' no pertenece al lenguaje--> linea
2
Error code'4'': Simbolo '%' no pertenece al lenguaje--> linea
2
Error code'4'': Simbolo '!' no pertenece al lenguaje--> linea
2
Error code'4'': Simbolo '|' no pertenece al lenguaje--> linea
2
Error code'4'': Simbolo '|' no pertenece al lenguaje--> linea
2
Error code'4'': Simbolo '%' no pertenece al lenguaje--> linea
2
Error code'5'': Comentarios de tipo: '//comentario', no estan
permitidos--> linea 3
Error code'4'': Simbolo '/' no pertenece al lenguaje--> linea
4
Error code'5'': Comentarios de tipo: '//comentario', no estan
permitidos--> linea 5
Error code'4'': Simbolo '/' no pertenece al lenguaje--> linea
6
```

Caso 2

> En este segundo caso es incorrecto también por lo mencionado anteriormente y además de eso, usa operaciones que no están contempladas: la operación resta "-" y la operación relacional menor "<".

```
let int num = 1;
function factorial int (int x) {
  if (x > 1)
    return x * factorial (x - 1); // operación resta - no admitida
  return 1;
} // la función devuelve un entero
function Suma boolean (int aux, int fin){
  /* se define la función Suma que recibe dos enteros por valor */
  /* usa la variable global x */
    for(x= 1; x < fin; x= x + 2){
      aux += factorial (aux-1);
    }
    return aux > 10000;
} // la función devuelve un lógico
function Imprime (int a){
    print (a);
}
Imprime (factorial (Suma (5, 6)));  // se llama a las tres funcione
```

Fichero de tokens:

```
< let ,   >
< int ,   >
< id , num >
< asig ,   >
< cteEnt , 1 >
< puntoComa ,   >
< function ,   >
< id , factorial >
< int ,   >
< parAbierto ,   >
< int ,   >
< id , x >
< parCerrado ,   >
< llaveAbierto ,   >
< id , if >
< parAbierto ,   >
< id , x >
< mayor ,   >
< cteEnt , 1 >
< parCerrado ,   >
< return ,   >
< id , x >
< por ,   >
< id , factorial >
< parAbierto ,   >
< id , x >
< cteEnt , 1 >
< parCerrado ,   >
< puntoComa ,   >
< return ,   >
< cteEnt , 1 >
< puntoComa ,   >
< llaveCerrado ,   >
< function ,   >
< id , Suma >
< boolean ,   >
< parAbierto ,   >
< int ,   >
< id , aux >
< coma ,   >
< int ,   >
< id , fin >
< parCerrado ,   >
< llaveAbierto ,   >
< id , for >
< parAbierto ,   >
< id , x >
< asig ,   >
< cteEnt , 1 >
< puntoComa ,   >
```

Fichero de TS:

```
TS GLOBAL #1
*Lexema: 'num'
*Lexema: 'factorial'
*Lexema: 'x'
*Lexema: 'if'
*Lexema: 'Suma'
*Lexema: 'aux'
*Lexema: 'fin'
*Lexema: 'for'
*Lexema: 'Imprime'
*Lexema: 'a'
```

Fichero de errores:

```
----------------------------------------
Errors output for file 'incorrecto2.txt':
----------------------------------------
Error code'4'': Simbolo '-' no pertenece al lenguaje--> linea
4
Error code'5'': Comentarios de tipo: '//comentario', no estan
permitidos--> linea 5
Error code'4'': Simbolo '/' no pertenece al lenguaje--> linea
6
Error code'5'': Comentarios de tipo: '//comentario', no estan
permitidos--> linea 8
Error code'4'': Simbolo '/' no pertenece al lenguaje--> linea
9
Error code'4'': Simbolo '
' no pertenece al lenguaje--> linea 10
Error code'4'': Simbolo ' ' no pertenece al lenguaje--> linea
10
Error code'4'': Simbolo '<' no pertenece al lenguaje--> linea
11
Error code'4'': Simbolo '-' no pertenece al lenguaje--> linea
12
Error code'5'': Comentarios de tipo: '//comentario', no estan
permitidos--> linea 16
Error code'4'': Simbolo '/' no pertenece al lenguaje--> linea
17
Error code'5'': Comentarios de tipo: '//comentario', no estan
permitidos--> linea 21
Error code'4'': Simbolo '/' no pertenece al lenguaje--> linea
22
```

```
< id , x >
< id , fin >
< puntoComa ,   >
< id , x >
< asig ,   >
< id , x >
< mas ,   >
< cteEnt , 2 >
< parCerrado ,   >
< llaveAbierto ,   >
< id , aux >
< mas ,   >
< asig ,   >
< id , factorial >
< parAbierto ,   >
< id , aux >
< cteEnt , 1 >
< parCerrado ,   >
< puntoComa ,   >
< llaveCerrado ,   >
< return ,   >
< id , aux >
< mayor ,   >
< cteEnt , 10000 >
< puntoComa ,   >
< llaveCerrado ,   >
< function ,   >
< id , Imprime >
< parAbierto ,   >
< int ,   >
< id , a >
< parCerrado ,   >
< llaveAbierto ,   >
< print ,   >
< parAbierto ,   >
< id , a >
< parCerrado ,   >
< puntoComa ,   >
< llaveCerrado ,   >
< id , Imprime >
< parAbierto ,   >
< id , factorial >
< parAbierto ,   >
< id , Suma >
< parAbierto ,   >
< cteEnt , 5 >
< coma ,   >
< cteEnt , 6 >
< parCerrado ,   >
< parCerrado ,   >
< parCerrado ,   >
< puntoComa ,   >
```

Caso 3

> Este caso es similar pero no sigue las reglas del lenguaje, como es iniciar comentarios erroneamente sin "*" después del "/" y asignar valores que exceden el tamaño admitido para un int y cadenas.

```
/*
Imprimir en salida estándar con formateo de variables entre la cadena de caracteres
*/
let int five = 5;
let int ten = 10;
let int number = 32768;
print('Fifteen is ' + five '+' ten 'and not ' + 2 * five '+' ten'.');
print("An int variable is lower than " + number);
let string test = "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz";
print(test);
```

Fichero de tokens:

```
< let ,   >
< int ,   >
< id , five >
< asig ,   >
< cteEnt , 5 >
< puntoComa ,   >
```

Fichero de TS:

```
TS GLOBAL #1
*Lexema: 'five'
*Lexema: 'ten'
*Lexema: 'number'
*Lexema: 'test'
```

```
< let ,   >
< int ,   >
< id , ten >
< asig ,   >
< cteEnt , 10 >
< puntoComa ,   >
< let ,   >
< int ,   >
< id , number >
< asig ,   >
< puntoComa ,   >
< print ,   >
< parAbierto ,   >
< mas ,   >
< id , five >
< id , ten >
< mas ,   >
< cteEnt , 2 >
< por ,   >
< id , five >
< id , ten >
< parCerrado ,   >
< puntoComa ,   >
< print ,   >
< parAbierto ,   >
< cadena , An int variable is lower than  >
< mas ,   >
< id , number >
< parCerrado ,   >
< puntoComa ,   >
< let ,   >
< string ,   >
< id , test >
< asig ,   >
< puntoComa ,   >
< print ,   >
< parAbierto ,   >
< id , test >
< parCerrado ,   >
< puntoComa ,   >
```

Fichero de errores:

```
----------------------------------------
Errors output for file 'incorrecto3.txt':
----------------------------------------
Error code'2'': Digito con valor mayor al permitido (32768) e
n el sistema--> linea 4
Error code'6'': Cadena se debe especificar entre " ", no con
 ' '--> linea 5
Error code'6'': Cadena se debe especificar entre " ", no con
 ' '--> linea 5
Error code'6'': Cadena se debe especificar entre " ", no con
 ' '--> linea 5
Error code'6'': Cadena se debe especificar entre " ", no con
 ' '--> linea 5
Error code'6'': Cadena se debe especificar entre " ", no con
 ' '--> linea 5
Error code'1'': Lexema excede el tamaño maximo de caracteres
 permitido--> linea 7
```