Entrega 2: Analizador Sintáctico

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:

GCL del lenguaje

Grámatica incial dada:

```
P -> BP | F P | eof
B -> let T id ; | if ( E ) S | S | while ( E ) { C }
T -> int | boolean | string
S -> id = E ; | return X ; | id ( L ) | print ( E ) ; | input ( id ) ;
X -> E | λ
C -> B C | λ
L -> E Q | λ
Q -> , E Q | λ
F -> function id H ( A ) { C }
H -> T | λ
A -> T id K  | λ
K -> , T id K | λ
E -> E && R | R
R -> R > U | U
U -> U + V | V
V -> id | ( E ) | id (L) | entero | cadena
```

Gramática transformada

```
P -> B P | F P | eof
B -> let T id ; | if ( E ) { S } O | S | while ( E ) { C } | do { S } while ( E ) ;
O -> else { S } | λ
T -> int | boolean | string
S -> id S' | return X ; | print ( E ) ; | input ( id ) ;
S' -> = E ; | ( L ) | ++ ; | == E
X -> E | λ
C -> B C | λ |
L -> E Q | λ
Q -> , E Q | λ
F -> function id H ( A ) { C }
H -> T | λ
A -> T id K | λ
K -> , T id K | λ
E -> R E'
E'-> && E'' | > E'' | λ
E'' -> R E'
R -> U R'
R'-> > U R' | λ
U -> V U'
U’ -> + U | * U | λ
V -> id V' | ( E ) | cteEnt | cadena | true | false
V'-> ( L ) | λ
```

Justificación de que es gramática LL(1):

Como la gramática está factorizada no existe ninguna producción: $A \rightarrow \alpha \mid \beta \mid ...$ donde First(α) ∩ First(β) ≠ **Ø**

Para los consecuentes que pueden derivar a λ :

- O → else { S } | λ —> First(else { S }) ∩ Follow(O) = **Ø**
  - First(O) = { else , λ }
  - Follow(O) = { let, if, while, do, id, return, print, input, function, eof }

  { let, if, while, do } ∪ First(S) ∪ { function } ∪ { eof }⊆ First(B) ∪ First(F) ∪ { eof } ⊆ First(P) ⊆Follow(B) ⊆ Follow(O)
- X → E | λ —> First(X) ∩ Follow(X) = **Ø**
  - First(X) = { id, (, entero, cadena, true, false }
  - Follow(X) = { ; }

  First(V) ⊆ First(U) ⊆ First(R) ⊆ First(E) ⊆ First(X)
- C → B C | λ —> First(B C) ∩ Follow(C) = **Ø**
  - First(B C) = First(B) = { let, if, while, do }
  - Follow(C) = { } }
- L → E Q | λ —> First(EQ) ∩ Follow(L) = **Ø**

    - First(EQ) = { id, (, entero, cadena, true, false }
    - Follow(L) = {) }

  First(V) ⊆ First(U) ⊆ First(R) ⊆ First(E) ⊆ First(EQ)

- Q → , E Q | λ —> First(, E Q) ∩ Follow(Q) = **Ø**
    - First(, E Q) = { , }
    - Follow(Q) = Follow(L) = {) }
- H → T | λ —> First(T) ∩ Follow(H) = **Ø**
    - First(T) = { int , boolean , string }
    - Follow(H) = { (}
- A → T id K | λ —> First(T id K) ∩ Follow(A) = **Ø**
    - First(T id K) = { int, boolean, string }
    - Follow(A) = {) }

  First(T) ⊆ First(T id K)
- K → , T id K | λ —> First(, T id K) ∩ Follow(K) = **Ø**
    - First(, T id K) = { , }
    - Follow(K) = Follow (A) = {) }
- E' → && E" | > E" | λ —> First(&& E") ∩ First(> E') ∩ Follow(E') = **Ø**
    - First(&& E") = { && }
    - First(> E') = { > }
    - Follow(E') = Follow(E) = {), ; } ∪ Follow(S') = {), ; } ∪ Follow(S) = {), ; } }
- R' → > U R' | λ —> First(U R') ∩ Follow(R') = **Ø**
    - First(U R') = { id, (, entero, cadena, true, false }
    - Follow(R') = {), ;, }

  First(V) ⊆ First(U) ⊆ First(U R')

  Follow(E') ⊆ Follow(E") ⊆ Follow(R) ⊆ Follow(R')
- U' → + U" | * U" | λ —> First(+ U") ∩ First(, T id K) ∩ Follow(K) = **Ø**
    - First(+ U") = { + }
    - First(, T id K) = { , }
    - Follow (E') = Follow(E) = {), ; } ∪ Follow(S') = Follow(S) = { } }
- V'-> (L) | λ —> First((L)) ∩ Follow(V') = **Ø**
    - First((L)) = { (}
    - Follow(V') = Follow(V) = First(U') = { +, * }

Reglas

```
1. P -> B P
2. P -> F P
3. P -> eof
4. B -> let T id puntoComa
5. B-> if parAbierto E parCerrado llaveAbierto S llaveCerrado O
6. B-> S
7. B-> while parAbierto E parCerrado llaveAbierto C llaveCerrado
8. B-> do llaveAbierto S llaveCerrado while parAbierto E parCerrado puntoComa
9. O -> else llaveAbierto S llaveCerrado
10. O -> lambda
11. T -> int
12. T -> boolean
13. T -> string
```

```
14. S -> id S'
15. S -> return X puntoComa
16. S ->  print parAbierto E parCerrado puntoComa
17. S ->  input parAbierto id parCerrado puntoComa
18. S' -> asig E puntoComa
19. S' -> parAbierto L parCerrado
20. S'-> postIncrem puntoComa
21. S'-> equals E
22. X -> E
23. X -> lambda
24. C -> B C
25. C -> lambda
26. L -> E Q
27. L -> lambda
28. Q -> coma E Q
29. Q -> lambda
30. F -> function id H parAbierto A parCerrado llaveAbierto C llaveCerrado
31. H -> T
32. H -> lambda
33. A -> T id K
34. A -> lambda
35. K -> coma T id K
36. K -> lambda
37. E -> R E'
38. E'-> and E''
39. E'-> mayor E''
40. E'-> lambda
41. E'' -> R E'
42. R -> U R'
43. R'-> mayor U R'
44. R'-> lambda
45. U -> V U'
46. U’ -> mas U
47. U’ -> por U
48. U’ -> lambda
49. V -> id V'
50. V -> parAbierto E parCerrado
51. V -> cteEnt
52. V -> cadena
53. V -> true
54. V -> false
55. V'-> parAbierto L parCerrado
56. V'-> lambda
```

Gramática para el árbol sintáctico

```
Terminales = { eof let id puntoComa if parAbierto parCerrado llaveAbierto llaveCerrado while do else function return else input print

NoTerminales = { P B O T S S' X C L Q F H A K E E' E'' R R' U U' V V' }

Axioma = P

Producciones = {
P -> B P
P -> F P
P -> eof
B -> let T id puntoComa
B-> if parAbierto E parCerrado llaveAbierto S llaveCerrado O
B-> S
B-> while parAbierto E parCerrado llaveAbierto C llaveCerrado
B-> do llaveAbierto S llaveCerrado while parAbierto E parCerrado puntoComa
O -> else llaveAbierto S llaveCerrado
O -> lambda
T -> int
T -> boolean
T -> string
S -> id S'
S -> return X puntoComa
S ->  print parAbierto E parCerrado puntoComa
S ->  input parAbierto id parCerrado puntoComa
S' -> asig E puntoComa
S' -> parAbierto L parCerrado
S'-> postIncrem puntoComa
S'-> equals E
X -> E
X -> lambda
C -> B C
C -> lambda
L -> E Q
L -> lambda
Q -> coma E Q
```

```
Q -> lambda
F -> function id H parAbierto A parCerrado llaveAbierto C llaveCerrado
H -> T
H -> lambda
A -> T id K
A -> lambda
K -> coma T id K
K -> lambda
E -> R E'
E'-> and E''
E'-> mayor E''
E'-> lambda
E'' -> R E'
R -> U R'
R'-> mayor U R'
R'-> lambda
U -> V U'
U’ -> mas U
U’ -> por U
U’ -> lambda
V -> id V'
V -> parAbierto E parCerrado
V -> cteEnt
V -> cadena
V -> boolT
V -> boolF
V' -> parAbierto L parCerrado
V' -> lambda
}
```

Código

Analizador Sintáctico

```
from lexer import *
from errorHandler import *

First = {
    'P': ["let", "if", "while", "do", "function", "eof"],
    'B': ["let", "if", "while", "do"],
    "O" : "else",
    "T" : ["int", "boolean", "string"],
    "S" : ["id", "return", "print", "input" ],
    "Sp": ["asig", "parAbierto", "postIncrem", "asig" ],
    "X" : [ "id", "parAbierto", "entero", "cadena", "true", "false"],
    "C" : ["let", "if", "while", "do"],
    "L" : ["id", "parAbierto", "entero", "cadena", "true", "false" ],
    "Q" : "coma" ,
    "F" : "function" ,
    "H" : [ "int", "boolean", "string" ],
    "A" : [ "int", "boolean", "string" ],
    "K" : "coma",
    "E" : ["id", "parAbierto", "entero", "cadena", "true", "false"],
    "Ep": [ "and", "mayor"],
    "Epp": [ "id", "parAbierto", "entero", "cadena", "true", "false"],
    "R" : ["id", "parAbierto", "entero", "cadena", "true", "false"],
    "Rp": "mayor",
    "U" : [ "id", "parAbierto", "entero", "cadena", "true", "false"],
    "Up": [ "mas", "por"],
    "V" : ["id", "parAbierto", "entero", "cadena", "true", "false"],
    "Vp": "parAbierto"
}

class error(Error):
    def __init__(self, num, linea):
        super().__init__(num, linea)

class Syntactic:
    def __init__(self, lexer: Lexer) -> None:
        self.lexer = lexer
        self.tokenList = lexer.tokenList
        self.index = 0 # indice que apunta al elemento actual de la lista de tokens
        self.token = self.tokenList[self.index].code
        self.reglas = []
        self.outputdir = lexer.outputdir
        self.errorList = []

    def next(self) -> Token:
```

```
        print("next: actual= " + self.token)
        self.index+=1
        self.token = self.tokenList[self.index].code
        print("siguiente " + self.token + '\n')
        return self.token

    def equipara(self, code: str, regla=None) -> bool:
        print("actual= " + self.token + " " + "a comparar= " + code)
        if regla is not None : self.reglas.append(regla)
        if (self.token == code):
            self.next()
            return True
        self.errorList.append(error(8, self.tokenList[self.index].line))

    def exportParse(self) -> None:
        '''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'''
        with open(self.outputdir+"/parse.txt", "w") as f:
            f.write("Descendente ")
            for regla in self.reglas: f.write(f"{regla} ".replace("None", ""))

    def P(self) -> None:
        # First(B)
        if (self.token) in First["B"]:
            self.reglas.append(1)
            self.B()
            self.P()
        elif (self.token) in First['F']:
            self.reglas.append(2)
            self.F()
            self.P()
        elif (self.equipara("eof")):
            self.reglas.append(3)
            return

    def B(self) -> None:
        if self.equipara("let", 4):
            self.T()
            if self.equipara("id"):
                if self.equipara("puntoComa"): return
        elif (self.equipara("if") and self.equipara("parAbierto")):
                self.E()
                if self.equipara("parCerrado") and self.equipara("llaveAbierto"):
                        self.S()
                        if(self.equipara("llaveCerrado")):
                            O()
                            return
        elif(self.token in First["S"]):
            self.reglas.append(6)
            self.S()
        elif self.equipara("while", 7):
            if self.equipara("parAbierto"):
                self.E()
                if (self.equipara("parCerrado")):
                    if self.equipara("llaveAbierto"):
                        self.C()
                        if self.equipara("llaveCerrado"): return
        elif self.equipara("do", 8):
            if self.equipara("llaveAbierto"):
                self.S()
                if self.equipara("llaveCerrado"):
                    if self.equipara("while"):
                        if self.equipara("parAbierto"):
                            self.E()
                            if self.equipara("parCerrado") and self.equipara("puntoComa"): return
        return

    def O(self) -> None:
        if self.equipara("else", 9):
            if self.equipara("parAbierto"):
                self.T()
                if self.equipara("parCerrado"): return
        else:
            self.reglas.append(10)
            return

    def T(self) -> None:
        if(self.equipara("int", 11)):
            return
        elif(self.equipara("boolean", 12)):
            return
        elif(self.equipara("string", 13)):
            return

    def S(self) -> None:
        if(self.equipara("id", 14)):
            self.Sp()
```

```
        elif(self.equipara("return", 15)):
            self.X()
            if(self.equipara("puntoComa")): return
        elif self.equipara("print", 16):
            if (self.equipara('parAbierto')):
                self.E()
                if self.equipara("parCerrado") and self.equipara("puntoComa"): return
        elif(self.equipara("input", 17) and self.equipara("parAbierto") and self.equipara("id") and self.equipara("parCerrado") and se
            return

    def Sp(self) -> None:
        if self.equipara("asig", 18):
            self.E()
            if self.equipara("puntoComa"):
                return
        elif (self.equipara("parAbierto", 19)):
            self.L()
            if self.equipara("parCerrado"): return
        elif self.equipara("postIncrem", 20): return
        elif self.equipara("equals", 21):
            self.E()


    def X(self) -> None:
        if self.token in First['E']:
            self.reglas.append(22)
            self.E()
        else: self.reglas.append(23)
        return

    def C(self) -> None:
        if self.token in First["B"]:
            self.reglas.append(24)
            self.B()
            self.C()
        elif self.token in First["T"]:
            self.T()
            if self.token in First['S']:
                self.reglas.append(9999)
                self.S()
                if (self.equipara("puntoComa")):
                    return
        else: self.reglas.append(25)
        return

    def L(self) -> None:
        if self.token in First["E"]:
            self.reglas.append(26)
            self.Q()
        else: self.reglas.append(27)
        return

    def Q(self) -> None:
        if self.equipara("coma", 28):
            self.E()
            self.Q()
        else: self.reglas.append(29)
        return

    def F(self) -> None:
        if self.equipara("function", 30) and self.equipara("id"):
            if self.token in First["H"]:
                self.H()
                if self.equipara("parAbierto"):
                    self.A()
                    if self.equipara("parCerrado") and self.equipara("llaveAbierto"):
                        self.C()
                        if self.equipara("llaveCerrado"): return

    def H(self) -> None:
        if self.token in First['T']:
            self.reglas.append(31)
            self.T()
        else: self.reglas.append(32)
        return

    def A(self) -> None:
        if self.token in First['T']:
            self.reglas.append(33)
            self.T()
            if self.equipara("id"):
                self.K()
        else: self.reglas.append(34)
        return

    def K(self) -> None:
        if self.equipara("coma", 35):
```

```
            self.T()
            if self.equipara("id"):
                self.K()
        else: self.reglas.append(36)
        return

    def E(self) -> None:
        if self.token in First["R"]:
            self.reglas.append(37)
            self.R()
            self.Ep()

    def Ep(self) -> None:
        if self.equipara("and", 38) :
            self.Epp()
        elif self.equipara(">", 39):
            self.Epp()
        else: self.reglas.append(40)
        return

    def Epp(self) -> None:
        if self.token in First["R"]:
            self.reglas.append(41)
            self.R()
            self.Ep()
        return

    def R(self) -> None:
        if self.token in First['U']:
            self.reglas.append(42)
            self.U()
            self.Rp()
        return

    def Rp(self) -> None:
        if (self .next() == ">"):
            self.reglas.append(43)
            self.U()
            self.Rp()
        else: self.reglas.append(44)
        return

    def U(self) -> None:
        if self.token in First["V"]:
            self.reglas.append(45)
            self.V()
            self.Up()
        return

    def Up(self) -> None:
        if self.equipara("mas", 46):
            self.U()
        elif self.equipara("*", 47):
            self.U()
        else: self.reglas.append(48)
        return

    def V(self) -> None:
        if self.equipara("id", 49):
            self.Vp()
        elif self.equipara("parAbierto", 50):
            self.E()
            if self.equipara("parCerrado"): return
        elif(self.equipara("cteEnt", 51)): return
        elif(self.equipara("cadena", 52)): return
        elif(self.equipara("true", 53)): return
        elif(self.equipara("false", 54)): return

    def Vp(self) -> None:
        if (self.equipara("parAbierto", 55)):
            self.L()
            if self.equipara("parCerrado"): return
        else: self.reglas.append(56)
        return
```

Gestor de Errores

```
from lexer import Lexer
from syntactic import Syntactic
from string import Template
```

```
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",
    8:"Error sintáctico"
}
class Error:
        def __init__(self, num:int, linea:int, attr=None):
            self.code = num
            self.line = linea
            self.att = attr

class ErrorHandler:
    def __init__(self, lexer : Lexer , syntactic : Syntactic ) -> None:
        self.lexer = lexer
        self.syntactic = syntactic

    def errorStringBuilder(self, tipo : str , f) -> None:
        if tipo == "lex": errList = self.lexer.errorList
        if tipo == "syn": errList = self.syntactic.errorList
        for error in errList:
            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)

    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 + "\nLexical errors: "
            f.write(header)
            self.errorStringBuilder("lex", f)
            header = f"\nSynactic errors':\n"
            times = len(header)-1
            header+="-" * times
            header = "-" * times +"\n"+header
            f.write(header)
            self.errorStringBuilder("syn", f)
```

Procesador JavaScript-PDL

```
from lexer import *
from syntactic import *
from tablaSimbolos import *
from errorHandler import *

'''
This file will contain all parts of the lenguage processor from
which its services will be launched to perform their tasks
'''
def main():
    lexer = Lexer()
    lexer.tokenize()
    lexer.printTokens()
    ts = TS(lexer)
    ts.printTS()
    syntactic = Syntactic(lexer)
    syntactic.P()
    syntactic.exportParse()
    errorHandler = ErrorHandler(lexer, syntactic)
    errorHandler.errorPrinter()

if __name__ == "__main__":
   main()
```

Anexo - Casos de Prueba

Casos Correctos

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) */
}
```

Caso 2

```
function potencia int (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 */
```

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*/
```

Casos Incorrectos

Caso 1

```
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
```

Caso 2

```
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
```

Caso 3

```
/*
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);
```