PROYECTO PROCESADORES DE LENGUAJES

2016-2017

Grupo: 87

Celia Barahona Blanco v130248
Jorge Bodega Fernanz v130323

INDICE

DISEÑO DEL PROCESADOR

Para este proyecto de la asignatura de Procesadores de Lenguajes, cursada en el quinto semestre, del curso 2016 – 2017, se debe implementar un Analizador para la versión del lenguaje JavaScript-PL.

El lenguaje utilizado para esta implementación ha sido Python y la herramienta para visualizar el árbol del analizador sintáctico ha sido VASt.

Las características que nuestro lenguaje debe tener son las siguientes:

Comentarios

Los comentarios no generan token y el elegido es el comentario de una línea, empiezan por los caracteres // y finalizan al acabar la línea. Pueden ir colocados en cualquier parte del código.

Constantes

Tenemos tres tipos de constantes para este lenguaje:

- Enteras → Representado en decimal, pueden ser positivos o negativos y hay que reservar 2 bytes (con signo). El máximo número representable será el 32767.
- Cadenas → Van encerradas entre comillas simples (‘Hola’) y puede aparecer cualquier carácter imprimible en la cadena. Internamente se utiliza el carácter nulo (ASCII: 0) como carácter de fin de cadena.
- Lógicas → Existen dos constantes lógicas que son las palabras reservadas ***true*** y ***false***.

Operadores

Existen una variedad de operadores, donde además, se pueden utilizar los paréntesis para agrupar subexpresiones.

- Operadores aritméticos → Se ha implementado la multiplicación (*****).
- Operadores relacionales → Se ha implementado el mayor que (**>**).
- Operadores lógicos → Se ha implementado la conjunción (**&&**).
- Operadores de incremento → Se ha implementado el post-incremento (**j++**).
- Operadores de asignación → Se ha implementado la asignación simple (**=**).

Identificadores

Los nombres de identificadores están formados por cualquier cantidad de letras, dígitos y subrayados (_), siendo el primero siempre una letra. El lenguaje es dependiente de minúsculas o mayúsculas.

Declaraciones

La declaración de las variables debe ser de la forma “**var T id**”, siendo T el tipo de dicha variable (entera, cadena o lógica).

Tipos de datos

Hay un tipo de dato por cada tipo de constante: constante entera (tipo **int**), constante cadena (tipo **chars**) y constante lógica (tipo **bool**).

Instrucciones de Entrada/Salida

Las instrucciones de entrada/salida implementadas son las siguientes:

- Write (expresión) → Evalúa la expresión (cadena o lógica) e imprime el resultado por pantalla.
- Prompt (var) → Lee un número o una cadena del teclado y lo almacena en var, que tiene que ser del tipo respectivo de entero (int) o cadena (chars).

Sentencias

Las sentencias que se han debido implementar son las siguientes:

- Asignación → Se realiza mediante el (=) de la siguiente manera **[identificador = expresión]**. Como no hay conversión entre tipos, identificador y expresión deben ser del mismo tipo.
- Llamada a una función → **[nombreFunción (argumento1,…)]**. El número de argumentos tiene que coincidir con los de la declaración de la función.
- Retorno de una función → **[return (expresión)]**. Puede o no haber expresión, en el caso de no haberla, la función debe haber sido declarada sin tipo.
- Condición simple → **[if (condición) sentencia]**. Si la condición se evalúa como cierta, de ejecuta la sentencia, en caso contrario, se finaliza la ejecución.
- Selección múltiple → **[switch (expresión)**
  {
  case valor1: sentencia1
  case valor 2: sentencia 2
  …
  }]. Ejecuta una de las sentencias basándose en el resultado de la expresión.
- Break → Aborta la ejecución de un bucle switch.

Funciones

Se deben definir las funciones antes de poder utilizarlas, la definición se realiza mediante la palabra reservada ***function***. El tipo solo se pone si la función retorna algún valor.

[function [Tipo] nombre (argumento1,…)
{
sentencias
}]

Analizador Léxico

Tokens

Los tokens son los elementos más pequeños del texto y el formato que deben seguir será: **del* < código del* , del* [atributo] del* > del* RE**, donde del* será cualquier cantidad de espacios en blanco, tabuladores o vacío.

Los tokens que reconoce nuestro analizador léxico son:

< Op, * >: Multiplicación
< Op, > >: Operador MAYOR QUE
< Op, && >: Operador lógico AND
< Op, = >: Operador de asignación
< Op, ++ >: Operador especial post-incremento
< Op, (>: Inicio de paréntesis
< Op,) >: Fin de paréntesis
< Op, { >: Inicio de llaves
< Op, } >: Fin de llaves
< Op, : >: Operador necesario para switch
< Op, - >: Operador números negativos
< Op, , >: Coma
< Op, ; >: Punto y coma

< id, lexema >: Identificador
< entero, valor >: Número con un valor
< cadena, texto >: Cadena con un texto

< PalRes, True >: Palabra reservada true
< PalRes, False >: Palabra reservada false
< PalRes, var >: Palabra reservada var
< PalRes, function >: Palabra reservada function
< PalRes, int >: Palabra reservada int
< PalRes, bool >: Palabra reservada bool
< PalRes, chars >: Palabra reservada chars
< PalRes, write >: Palabra reservada write
< PalRes, prompt >: Palabra reservada prompt
< PalRes, return >: Palabra reservada return
< PalRes, if >: Palabra reservada if
< PalRes, switch >: Palabra reservada switch
< PalRes, case >: Palabra reservada case
< PalRes, break >: Palabra reservada break

< cr, - >: Salto de línea
< FinFich , eof >: Fin de fichero

Gramática (G3)

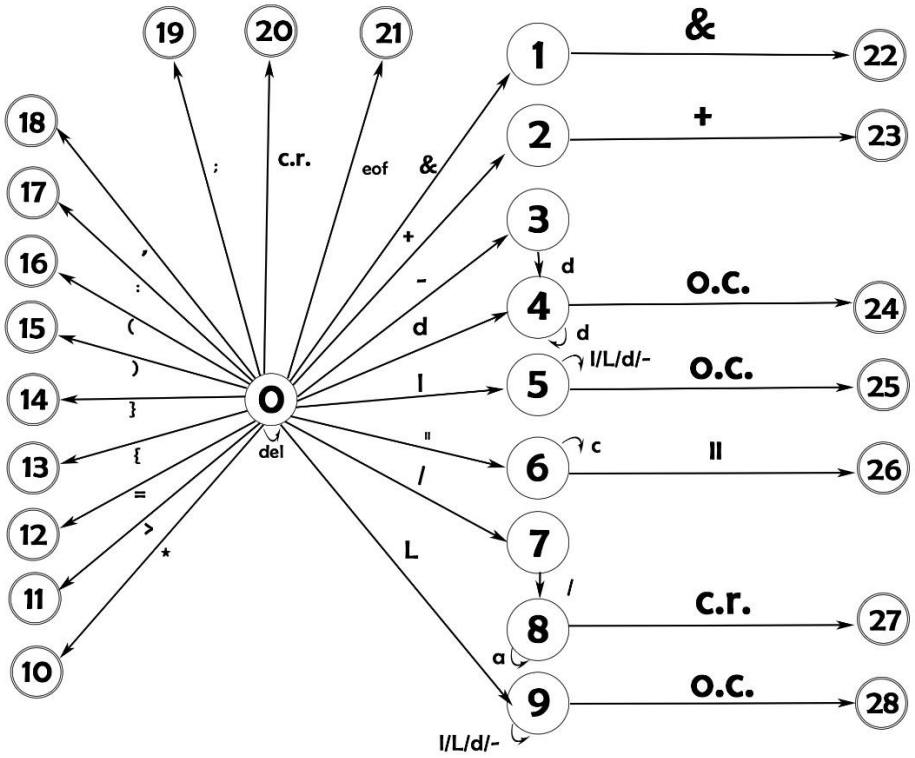
(0) S → * | > | = | (|) | { | } | : | , | ; | &A | +B | - C | dD | lE | LK | “F | /G | delS | cr | eof
(1) A → &
(2) B → +
(3) C → dD
(4) D → dD | λ
(5) E → lE | LE | dE | _E | λ
(6) F → cF | “
(7) G → /H
(8) H → aH | cr
(9) K → lK | LK | dK | _K | λ

d : {0-9}
l : {a-z}
L : {A-Z}
c : Todo – {“}
a : Todo – {cr}

Errores

Los errores que tenemos en cuenta en nuestro proyecto, es que los valores de los números enteros no pueden superar el 32767, porque estaríamos fuera de rango de los 2 bytes que se reservan para números positivos y negativos.

Autómata (AFD)



Acciones semánticas

Para no crear confusión en el autómata, realizaremos un listado de las acciones semánticas de cada intervalo.

0:0 Leer
0:1 Leer
1:22 Leer, GENTOKEN (Op,'&&')
0:2 Leer
2:23 Leer, GENTOKEN (Op,'++')
0:3 Leer, num :=-1
3:4 Leer, num = num*d
0:4 Leer, num := d
4:4 Leer, num = num*10 + d
4:24 if (num>32767) then Error (Fuera de rango)
else GENTOKEN (entero, num)
0:5 Leer, CONCAT(lexema,l)
5:5 Leer, símbolo=l/L/d/_ , CONCAT (lexema,símbolo)
5:25 p := BUSCA (lexema)
if (p ϵ PalRes) then GENTOKEN (PalRes, p)
else if (p ϵ ID) then GENTOKEN (ID, p)
else p := AÑADE (lexema)
GENTOKEN (ID, p)
0:6 Leer
6:6 Leer, CONCAT (lexema, c.valor)
6:26 Leer, GENTOKEN (cadena, lexema)
0:7 Leer
7:8 Leer
8:8 Leer
8:27 Leer, GENTOKEN (cr,-)
0:9 Leer, CONCAT(lexema,L)
9:9 Leer, símbolo=l/L/d/_ , CONCAT (lexema,símbolo)
9:28 p := BUSCA (lexema)
if (p ϵ PalRes) then GENTOKEN (PalRes, p)
else if (p ϵ ID) then GENTOKEN (ID, p)
else p := AÑADE (lexema)
GENTOKEN (ID, p)
0:10 Leer, GENTOKEN (Op,'*')
0:11 Leer, GENTOKEN (Op,'>')
0:12 Leer, GENTOKEN (Op,'=')
0:13 Leer, GENTOKEN (Op,'{')
0:14 Leer, GENTOKEN (Op,'}')
0:15 Leer, GENTOKEN (Op,')')
0:16 Leer, GENTOKEN (Op,'(')
0:17 Leer, GENTOKEN (Op,':')
0:18 Leer, GENTOKEN (Op,',')
0:19 Leer, GENTOKEN (Op,';')
0:20 Leer, GENTOKEN (cr,-)
0:21 Leer, GENTOKEN (eof,-)

Analizador Sintáctico

Gramática Tipo2

P' → P
P → BZP | FZP | ZP | eof
B → var T id | if (E) S | switch (E) Z { Z W } | S
T → int | bool | chars
S → return X | id = E | id (L) | write (E) | prompt (id)
X → E | λ
L → EQ | λ
Q → , EQ | λ
E → E && R | R
R → R > U | U
U → U * V | V
V → id | entero | cadena | True | False | id (L) | (E) | id++
F → function H id (A) Z { Z C }
H → T | λ
A → T id K | λ
K → , T id K | λ
Z → cr Z'
Z' → cr Z' | λ
C → BZC | λ
W → case Y : S M Z N
Y → entero
M → ; break | cr break | λ
N → case Y : S M Z N | λ

Análisis Sintáctico Descendente Recursivo

La gramática no puede tener recursividad por la izquierda ni ser una gramática no factorizada, además ha de cumplir la condición LL (1).

1. Recursividad por la izquierda

- E → E && R | R
  E → R E'
  E' → && R E' | λ
- R → R > U | U
  R → U R'
  R' → > U R' | λ
- U → U * V | V
  U → V U'
  U' → * V U' | λ

2. Gramática factorizada

- S → return X | id = E | id (L) | write (E) | prompt (id)
  S → return X | id S' | write (E) | prompt (id)
  S' → = E | (L)

- V → id | entero | cadena | true | false | id (L) | (E) | id++
  V → id V' | entero | cadena | true | false | (E)
  V' → (L) | ++ | λ

3. Gramática correcta para A.St.Descendente Recursivo

P' → P
P → BZP | FZP | ZP | eof
B → var T id | if (E) S | switch (E) Z { Z W } | S
T → int | bool | chars
S → return X | id S' | write (E) | prompt (id)
S' → = E | (L)
X → E | λ
L → E Q | λ
Q → , E Q | λ
E → R E'
E' → && R E' | λ
R → U R'
R' → > U R' | λ
U → V U'
U' → * V U' | λ
V → id V' | entero | cadena | True | False | (E)
V' → (L) | ++ | λ
F → function H id (A) Z { Z C }
H → T | λ
A → T id K | λ
K → , T id K | λ
Z → cr Z'
Z' → cr Z' | λ
C → B Z C | λ
W → case Y : S M Z N
Y → entero
M → ; break | cr break | λ
N → case Y : S M Z N | λ

Conjuntos FIRST

FIRST (P') = { var, if, switch, return, id, write, prompt, function, cr, eof }
FIRST (P) = { var, if, switch, return, id, write, prompt, function, cr, eof }
FIRST (B) = { var, if, switch, return, id, write, prompt }
FIRST (T) = { int, bool, chars }
FIRST (S) = { return, id, write, prompt }
FIRST (S') = { =, λ }
FIRST (X) = { id, entero, cadena, True, False, (, λ }
FIRST (L) = { id, entero, cadena, True, False, (, λ }
FIRST (Q) = { ",", λ }
FIRST (E) = { id, entero, cadena, True, False, (}
FIRST (E') = { &&, λ }
FIRST (R) = { id, entero, cadena, True, False, (}
FIRST (R') = { >, λ }
FIRST (U) = { id, entero, cadena, True, False, (}
FIRST (U') = { *, λ }
FIRST (V) = { id, entero, cadena, True, False, (}
FIRST (V') = { (, ++, λ }
FIRST (F) = { function }
FIRST (H) = { int, bool, chars, λ }
FIRST (A) = { int, bool, chars, λ }
FIRST (K) = { ",", λ }
FIRST (Z) = { cr }
FIRST (Z') = { cr, λ }
FIRST (C) = { var, if, switch, return, id, write, prompt, λ }
FIRST (W) = { case }
FIRST (Y) = { entero }
FIRST (M) = { ";", cr , λ }
FIRST (N) = { case, λ }

Conjuntos FOLLOW

FOLLOW (P') = { $ }
FOLLOW (P) = { $ }
FOLLOW (B) = { cr }
FOLLOW (T) = { id }
FOLLOW (S) = { cr, }, ";"}
FOLLOW (S') = { cr, }, ";"}
FOLLOW (X) = { cr, }, ";"}
FOLLOW (L) = {) }
FOLLOW (Q) = {) }
FOLLOW (E) = { cr, }, ";",), "," }
FOLLOW (E') = { cr, }, ";",), "," }
FOLLOW (R) = { cr, }, ";",), ",", & }
FOLLOW (R') = { cr, }, ";",), ",", & }
FOLLOW (U) = { cr, }, ";",), ",", &, > }
FOLLOW (U') = { cr, }, ";",), ",", &, > }
FOLLOW (V) = { cr, }, ";",), ",", &, >, * }
FOLLOW (V') = { cr, }, ";",), ",", &, >, * }
FOLLOW (F) = { cr }
FOLLOW (H) = { id }
FOLLOW (A) = {) }
FOLLOW (K) = {) }
FOLLOW (Z) = { var, if, switch, return, id, write, prompt, function, cr, eof, case, "{", "}" }
FOLLOW (Z') = { var, if, switch, return, id, write, prompt, function, cr, eof, case, "{", "}" }
FOLLOW (C) = { "}" }
FOLLOW (W) = { "}" }
FOLLOW (Y) = { : }
FOLLOW (M) = { cr }
FOLLOW (N) = { "}" }

Condición LL (1)

La condición LL (1) se comprueba mirando si tenemos dos cadenas en un no terminal, α y β no pueden empezar por los mismos tokens. Debemos comprobar que FIRST (α) ∩ FIRST (β) = Ø, siendo A→α|β. Si β=λ, entonces FIRST (α) ∩ FOLLOW(A) = Ø.

- P → BZP | FZP | ZP | eof
  FIRST (BZP) **∩** FIRST (FZP) **∩** FIRST (ZP) **∩** FIRST (eof)
  {var, if, switch, return, id, write, prompt} **∩** {function} **∩** {cr} **∩** {eof} = **Ø**

- B → var T id | if (E) S | switch (E) Z { Z W } | S
  FIRST (var T id) **∩** FIRST (if (E) S) **∩** FIRST (switch (E) Z {Z W}) **∩** FIRST (S)
  {var} **∩** {if} **∩** {switch} **∩** {return, id, write, prompt} = **Ø**

- T → int | bool | chars
  FIRST (int) **∩** FIRST (bool) **∩** FIRST (chars)
  {int} **∩** {bool} **∩** {chars} = **Ø**

- S → return X | id S' | write (E) | prompt (id)
  FIRST (return X) **∩** FIRST (id S') **∩** FIRST (write (E)) **∩** FIRST (prompt (id))
  {return}**∩** {id} **∩** {write} **∩** {prompt} = **Ø**

- S' → = E | (L)
  FIRST (=E) **∩** FIRST ((L))
  {=} **∩** { (} = **Ø**

- X → E | λ
  FIRST (E) **∩** FIRST (λ) → { id, entero, cadena, true, false, (} **∩** { λ } = **Ø**
  FIRST (E) **∩** FOLLOW (X) → { id, entero, cadena, true, false, (} **∩** { cr, }, ";"} = **Ø**

- L → EQ | λ
  FIRST (EQ) **∩** FIRST (λ) → { id, entero, cadena, true, false, (} **∩** { λ } = **Ø**
  FIRST (EQ) **∩** FOLLOW (L) → { id, entero, cadena, true, false, (} **∩** {) } = **Ø**

- Q → , EQ | λ
  FIRST (,EQ) **∩** FIRST (λ) → { "," } **∩** { λ } = **Ø**

FIRST (,EQ) **∩** FOLLOW (Q) → { "," } **∩** {) } = **∅**

- E' → &&RE' | λ
  FIRST (&&RE') **∩** FIRST (λ) → { & } **∩** { λ } = **∅**
  FIRST (&&RE') **∩** FOLLOW (E') → { & } **∩** { cr, }, ";",), "," } = **∅**

- R' → >UR' | λ
  FIRST (>UR') **∩** FIRST (λ) → { > } **∩** { λ } = **∅**
  FIRST (>UR') **∩** FOLLOW (R') → { > } **∩** { cr, }, ";",), ",", & } = **∅**

- U' → *VU' | λ
  FIRST (*VU') **∩** FIRST (λ) → { * } **∩** { λ } = **∅**
  FIRST (*VU') **∩** FOLLOW (U') → { * } **∩** { cr, }, ";",), ",", &, > } = **∅**

- V → id V' | entero | cadena | true | false | (E)
  FIRST (id V') **∩** FIRST (entero) **∩** FIRST (cadena) **∩** FIRST (True) **∩** FIRST (False) **∩** FIRST ((E))
  {id} **∩** {entero} **∩** {cadena} **∩** {true} **∩** {false} **∩** { (} = **∅**

- V' → (L) | ++ | λ
  FIRST ((L)) **∩** FIRST (++) **∩** FIRST (λ) → { (} **∩** { + } **∩** { λ } = **∅**

  FIRST ((L)) ∩ FOLLOW (V') → { (} **∩** { cr, }, ";",), ",", &, >, * } = **∅**
  FIRST (++) ∩ FOLLOW (V') → { + } **∩** { cr, }, ";",), ",", &, >, * } = **∅**

- H → T | λ
  FIRST (T) **∩** FIRST (λ) → { int, chars, bool } **∩** { λ } = **∅**
  FIRST (T) **∩** FOLLOW (H) → { int, chars, bool } **∩** { id } = **∅**

- A → T id K | λ
  FIRST (T id K) **∩** FIRST (λ) → { int, chars, bool } **∩** { λ } = **∅**
  FIRST (T id K) ∩ FOLLOW (A) → { int, chars, bool } **∩** {) } = **∅**

- K → , T id K | λ
  FIRST (, T id K) **∩** FIRST (λ) → { "," } **∩** { λ } = **∅**
  FIRST (, T id K) ∩ FOLLOW (K) → { "," } **∩** {) } = **∅**

- Z' → cr Z' | λ
  FIRST (cr Z') **∩** FIRST (λ) → { cr } **∩** { λ } = **∅**

- C → BZC | λ
  FIRST (BZC) **∩** FIRST (λ) → { var, if, switch, return, id, write, prompt } **∩** { λ } = **∅**
  FIRST (BZC) **∩** FOLLOW (C) → { var, if, switch, return, id, write, prompt } **∩** { } } = **∅**

- M → ; break | cr break | λ
  FIRST (; break) **∩** FIRST (cr break) **∩** FIRST (λ) → { ; } **∩** { cr } **∩** { λ } = **∅**

- N → case Y : S M Z N | λ
  FIRST (case Y : S M Z N) **∩** FIRST (λ) → { case } **∩** { λ } = **∅**
  FIRST (case Y : S M Z N) **∩** FOLLOW (W) → { case } **∩** { "}" } = **∅**

Como hemos comprobado, la gramática cumple la condición LL (1)

Procedimientos A.St.D.Recursivo

PROCEDURE empareja (t)

```
{       if (siguiente_token = <t>) then
        {     siguiente_token := AL ()
        }
        else  Error_Sintáctico
}
```

PROCEDURE P' ()

```
{
        P()
}
```

PROCEDURE P ()

```
{       if (siguiente_token ϵ {<var>, <if>, <switch>, <id>, <return>, <write>, <prompt>}) then
        {       B ()
                Z ()
                P ()
        }
        else if (siguiente_token = <function>) then
        {       F ()
                Z ()
                P ()
        }
        else if (siguiente_token = <cr>) then
        {       Z ()
                P ()
        }
        else if (siguiente_token = <eof>) then
        {
                empareja (eof)
        }
        else Error (…)
}
```

PROCEDURE T ()

```
{       if (siguiente_token = <int>) then
        {
                empareja (int)
        }
        else if (siguiente_token = <chars>) then
        {
                empareja (chars)
        }
        else if (siguiente_token = <bool>) then
        {
                empareja (bool)
        }
        else Error (…)
}
```

PROCEDURE S' ()

```
{       if (siguiente_token = <=>) then
        {       empareja (=)
                E ()
        }
        else if (siguiente_token = <(>) then
        {       empareja (()
                L ()
                empareja ())
        }
        else Error (…)
}
```

PROCEDURE B ()

```
{       if (siguiente_token = <var>) then
        {       empareja (var)
                T ()
                empareja (id)
        }
        else if (siguiente_token = <if>) then
        {       empareja (if)
                empareja (()
                E ()
                empareja ())
                S ()
        }
        else if (siguiente_token = <switch>) then
        {       empareja (switch)
                empareja (()
                E ()
                empareja ())
                Z ()
                empareja ({)
                Z ()
                W ()
                empareja (})
        }
        else if (siguiente_token ϵ {<id>, <return>,
                                <write>, <prompt>}) then
        {       S ()
        }
        else Error (…)
}
```

PROCEDURE S ()

```
{       if (siguiente_token = <id>) then
        {       empareja (id)
                S' ()
        }
        else if (siguiente_token = <return>) then
        {       empareja (return)
                X ()
        }
        else if (siguiente_token = <write>) then
        {       empareja (write)
                empareja (()
                E ()
                empareja ())
        }
        else if (siguiente_token = <prompt>) then
        {       empareja (prompt)
                empareja (()
                empareja (id)
                empareja ())
        }
        else Error (…)
}
```

PROCEDURE X ()

```
{       if (siguiente_token ϵ {<id>, <entero>, <cadena>,
                                <true>, <false>, <(>}) then
        {       E ()
        }
        else Ø
}
```

PROCEDURE L ()

```
{       if (siguiente_token ϵ {<id>, <entero>,
                <cadena>, <true>, <false>, <(>}) then
        {       E ()
                Q ()
        }
        else Ø
}
```

PROCEDURE Q ()

```
{       if (siguiente_token = <,>) then
        {       empareja (,)
                E ()
                Q ()
        }
        else Ø
}
```

PROCEDURE E' ()

```
{       if (siguiente_token = <&&>) then
        {       empareja (&&)
                R ()
                E' ()
        }
        else Ø
}
```

PROCEDURE E ()

```
{       R ()
        E' ()
}
```

PROCEDURE R' ()

```
{       if (siguiente_token = <>>) then
        {       empareja (>)
                U ()
                R' ()
        }
        else Ø
}
```

PROCEDURE U ()

```
{       V ()
        U' ()
}
```

PROCEDURE V ()

```
{       if (siguiente_token = <id>) then
        {       empareja (id)
                V' ()
        }
        else if (siguiente_token = <entero>) then
        {       empareja (entero)
        }
        else if (siguiente_token = <cadena>) then
        {       empareja (cadena)
        }
        else if (siguiente_token = <True>) then
        {       empareja (True)
        }
        else if (siguiente_token = <False>) then
        {       empareja (False)
        }
        else if (siguiente_token = <(>) then
        {       empareja (()
                E ()
                empareja ())
        }
        else Error (…)
}
```

PROCEDURE R ()

```
{       U ()
        R' ()
}
```

PROCEDURE U' ()

```
{       if (siguiente_token = <*>) then
        {       empareja (*)
                V ()
                U' ()
        }
        else Ø
}
```

PROCEDURE H ()

```
{       if (siguiente_token ϵ {<int>, <chars>,
<bool>}) then
        {       T ()
        }
        else Ø
}
```

PROCEDURE F ()

```
{       empareja (function)
        H ()
        empareja (id)
        empareja (()
        A ()
        empareja ())
        Z ()
        empareja ({)
        Z ()
        C ()
        empareja (})
}
```

PROCEDURE A ()

```
{       if (siguiente_token ϵ {<int>, <chars>,
<bool>}) then
        {       T ()
                empareja (id)
                K ()
        }
        else Ø
}
```

PROCEDURE K ()

```
{       if (siguiente_token = <,>) then
        {       empareja (,)
                T ()
                empareja (id)
                K ()
        }
        else Ø
}
```

PROCEDURE W ()

```
{       empareja (case)
        Y ()
        empareja (:)
        S ()
        M ()
        Z ()
        N ()
}
```

PROCEDURE N ()

```
{       if (siguiente_token = <case>) then
        {       empareja (case)
                Y ()
                empareja (:)
                S ()
                M ()
                Z ()
                N ()
        }
        else Ø
}
```

PROCEDURE Z' ()

```
{       if (siguiente_token = <cr>) then
        {       empareja (cr)
                Z' ()
        }
        else Ø
}
```

PROCEDURE M ()

```
{       if (siguiente_token = <;>) then
        {       empareja (;)
                empareja (break)
        }
        else if (siguiente_token = <cr>) then
        {       empareja (cr)
                empareja (break)
        }
        else Ø
}
```

PROCEDURE Y ()

```
{       empareja (entero)
}
```

PROCEDURE Z ()

```
{       empareja (cr)
        Z' ()
}
```

PROCEDURE C ()

```
{       if (siguiente_token ϵ {<var>, <if>, <switch>, <id>, <return>, <write>, <prompt>}) then
        {       B ()
                Z ()
                C ()
        }
        else Ø
}
```

Analizador Semántico

P' → {{
ptrTS_actual = "TS_General"
DicTS[ptrTS_actual] = [[], 0, "null"] // (TS,Desp,TS_padre)
}}
P
{{
TS_actual = "null"
}}

P → B Z P (No hace nada)
P → F Z P (No hace nada)
P → Z P (No hace nada)
P → eof (No hace nada)

B → var T id
{{
if buscarId(id.ent) ≠ null
Then Error (“Identificador ya creado”)
else
Then
insertarTS(id.ent, T.tipo, Desp_actual, ptrTS_actual)
Desp_actual += T.tamano
B.tipo := tipo_ok
}}
B → if (E)
{{
if E.tipo ≠ logico
Then Error (“condición "if" no logica”)
}}
S
B → switch (E)
{{
if E.tipo ≠ entero
Then Error ("condición \"switch\" no entera")
}}
Z {Z W}
{{
B.tipo := tipo_ok
}}
B → S {{
B.tipo := S.tipo
}}

T → int {{ T.tipo = entero , T.tamano = 2 }}
T → bool {{ T.tipo = logico , T.tamano = 1 }}
T → chars {{ T.tipo = cadena , T.tamano = 2 }}

S → id S'
{{
if buscarTipoId(id.ent) ≠ null
Then
if buscarTipoFuncion(id.ent) ≠ null // CASO: id(L)
Then
if buscarTipoFuncion(id.ent) == S'.tipo
Then S.tipo := S'.tipo
else
Then Error (“Parametros mal insertados”)
else //CASO id = E
Then
if buscarTipoId(id.ent) == S'.tipo
Then S.tipo := tipo_ok
else
Then Error (“asignacion mal formada”)
else
Then Error (“Identificador no declarado”)
}}
S → return X
{{
S.tipo := X.tipo
}}
S → write (E)
{{
S.tipo := E.tipo
}}
S → prompt (id)
{{
if buscarTipoId(id.ent) ≠ "entero" and buscarTipoId(id.ent) ≠ "cadena"
Then Error (“uso incorrecto de "prompt" ”)
else
Then S.tipo:= buscarTipoId(id.ent)
}}

S'→ (L)
{{
S'.tipo:= L.tipo
}}
S'→ = E
{{
S'.tipo := E.tipo
}}

X → E {{
X.tipo := E.tipo
}}
X → λ
{{
X.tipo := tipo_vacio
}}

L → E Q

{{
if E.tipo ≠ tipo_error and Q.tipo ≠ tipo_error
Then L.tipo:= f(E.tipo,Q.tipo)
else
Then Error (“contenido parentesis del identificador llamado mal formado”)
}}

L → λ

{{
L.tipo := tipo_vacio
}}

Q → , E Q

{{
if E.tipo ≠ tipo_error and Q.tipo ≠ tipo_error
Then Q.tipo:= f(E.tipo,Q.tipo)
else
Then Error (“contenido parentesis del identificador llamado mal formado”)
}}

Q → λ

{{
Q.tipo := tipo_vacio
}}

E → R E'

{{
if E'.tipo ≠ tipo_vacio and R.tipo ≠ E'.tipo
Then Error (“Comparacion de elementos con tipos distintos”)
else
Then
if E'.tipo ≠ tipo_vacio
Then E.tipo := "logico"
else
Then E.tipo := R.tipo
}}

E'→ && R E'

{{
if E'.tipo ≠ tipo_vacio and E'.tipo ≠ R.tipo ≠ "logico"
Then Error (“Comparacion de elementos de tipo no logico”)
else
Then E'.tipo := R.tipo
}}

E'→ λ

{{
E'.tipo := tipo_vacio
}}

R → U R'

```
{{
if R'.tipo ≠ tipo_vacio and U.tipo ≠ R'.tipo
        Then Error (“Comparacion de elementos con tipos distintos”)
else
        Then
                if R'.tipo ≠ tipo_vacio
                        Then R.tipo := "logico"
                else
                        Then R.tipo := U.tipo

}}
```

R'→ > U R'

```
{{
if R'.tipo ≠ tipo_vacio and R'.tipo ≠ U.tipo ≠ "entero"
        Then Error (“Comparacion de elementos de tipo no entero”)
else
        Then R'.tipo := U.tipo
}}
```

R'→ λ

```
{{
R'.tipo := tipo_vacio
}}
```

U → V U'

```
{{
if U'.tipo ≠ tipo_vacio and V.tipo ≠ U'.tipo
        Then Error (“Suma de elementos de tipo no entero”)
else
        Then U.tipo := V.tipo
}}
```

U'→ * V U'

```
{{
if V.tipo ≠ "entero"
        Then Error (“Suma de elementos de tipo no entero”)
else
        Then U'.tipo := V.tipo
}}
```

U'→ λ

```
{{
U'.tipo := tipo_vacio
}}
```

V → (E) {{ V.tipo := E.tipo }}
V → entero {{ V.tipo := entero }}
V → cadena {{ V.tipo := cadena }}
V → True {{ V.tipo := logico }}
V → False {{ V.tipo := logico }}

V → id V'

```
{{
if buscarTipoId(id.ent) ≠ null
        Then
                if  buscarTipoFuncion(id.ent) ≠ null // CASO: id(L)
                        Then
                                if buscarTipoFuncion(id.ent) == V'.tipo
                                        Then V.tipo := V'.tipo
                                else
                                        Then Error (“Parametros mal insertados”)
                else //CASO id
                        Then V.tipo := buscarTipoId(id.ent)
else
        Then Error (“Identificador no declarado”)
}}
```

V'→ (L)

```
{{
V'.tipo := L.tipo
}}
```

V'→ ++

```
{{
if buscarTipoId(TSaux[N_IDS-1], ptrTS_actual) != "entero":
    Error_Sem("uso incorrecto de \"++\"")
}}
```

V'→ λ

```
{{
V'.tipo := tipo_vacio
}}
```

F → function H id

```
{{
if buscarTS(id.ent) ≠ null
        Then Error (“Función ya declarada”)
else
        insertarTS(id.ent, "function", Desp_actual, ptrTS_actual)
        DicTS[ptrTS_actual] := (TS_actual,Desp_actual) // TS actual con el desplazamiento
        crearTS TS_nueva
        TS_actual := TS_nueva
        Desp_actual := 0
        ptrTS_anterior := ptrTS_actual
        ptrTS_actual := id.ent
}}
(A) Z {
{{
insertarFunTS(ptrTS_anterio, id.ent, A.numParam, A.tipo, id.ent)
}}
Z C }
{{
if C.tipo ≠ H.tipo or C.tipo ≠ tipo_ok
        Then Error (“Funcion mal return”)
else
```

```
                Then
                DicTS[ptrTS_actual] := (TS_actual,Desp_actual) // guarda la TS actual con el
desplazamiento
                DicTS[ptrTS_anterior] := (TS_anterior,Desp_anterior) // recuperamos la tabla de
simbolos anterior
                TS_actual       := TS_anterior
                Desp_actual := Desp_anterior
                ptrTS_actual  := ptrTS_anterio
                ptrTS_anterio := null
        }}
```

H → T {{ H.tipo = T.tipo , H.tamano = T.tamano }}
H → λ {{ H.tipo = tipo_vacio , H.tamano = 0 }}

```
A → T id
        {{
        if buscarId(id.ent) ≠ null
                Then Error (“Identificador ya creado”)
        else
                Then
                        insertarTS(id.ent, T.tipo, Desp_actual, ptrTS_actual)
                        Desp_actual += T.tamano
        }}
        K
        {{
        A.tipo := f(T.tipo, K.tipo)
        A.numParam := K.numParam + 1
        }}
A → λ
        {{
        A.tipo = tipo_vacio
        A.numParam := 0
        }}
```

```
K → , T id {{
        if buscarId(id.ent) ≠ null
                Then Error (“Identificador ya creado”)
        else
                Then
                        insertarTS(id.ent, T.tipo, Desp_actual, ptrTS_actual)
                        Desp_actual += T.tamano
        }}
        K
        {{
        K.tipo := f(T.tipo, K.tipo)
        K.numParam := K.numParam + 1
        }}
K → λ
        {{
        K.tipo = tipo_vacio
        K.numParam := 0
        }}
```

Z → cr Z' (No hace nada)
Z' → cr Z' (No hace nada)
Z' → λ (No hace nada)

C → B Z C
{{
if C.tipo ≠ tipo_error and B.tipo ≠ tipo_error
Then C.tipo := tipo_ok
else
Then Error (“contenido de corchetes mal formado”)
}}
C → λ
{{
C.tipo := tipo_vacio
}}

W → case Y : S M Z N
{{
W.tipo := entero, W.tamano := 2
}}

Y → entero
{{
Y.tipo := entero, Y.tamano := 2
}}

M → ; break (No hace nada)
M → cr break (No hace nada)
M → λ
{{
M.tipo := tipo_vacio
}}

N → case Y : S M Z N
{{
N.tipo := S.tipo
}}
N → λ
{{
N.tipo := tipo_vacio
}}

Tabla de Símbolos

Un fichero con la TS podrá tener líneas en blanco o líneas con la información de la TS. Cada línea de información debe acabar obligatoriamente con un salto de línea. Existen tres tipos de líneas obligatorias, cada una de ellas con un formato diferente:

Línea con el número de la TS

Esta línea se utiliza como encabezado para comenzar cada TS. El formato de esta línea es:

pal* **#** del* **núm** del* **:** del* **RC**

Línea del lexema

Esta línea se utiliza para indicar cada entrada de la TS. El formato de esta línea es:

del* ***** del* *[***LEXEMA** del* **:***]* del* **'nombre'** del* **RC**

Línea de atributo

Esta línea se utiliza para indicar cada atributo perteneciente a la entrada previa indicada en la 'línea del lexema' de la TS. El formato de esta línea es:

del* **+** del* **atributo** del* **:** del* **valor RC**

Ejemplo de Tabla de Símbolos con un formato correcto en un único fichero

```
TABLA PRINCIPAL #1:
* (esto es una función) LEXEMA : 'suma'
 ATRIBUTOS:
  + tipo: 'int'
   + parametros: 2
    + tipoparam1: 'int'
     + tipoparam2:'float'
      + idtabla: 2
-------------------------------------------------
TABLA DE LA FUNCIÓN suma #2:
* LEXEMA: 'a1' (tipo de entrada 'parámetro')
+ desplazamiento : 0
+ tipo : 'int'

* LEXEMA: 'b2' (tipo de entrada 'parámetro')
+ tipo : 'float'
+ desplazamiento : -2

* LEXEMA: 'c_3' (tipo de entrada 'variable')
+ tipo : 'vector'
+ elementos : 'int'
+ tam: 500
      + desplazamiento : -8
```

Ejemplos

Para la comprobación del correcto funcionamiento de nuestro proyecto, hemos usado varias pruebas, con resultados previamente conocidos, mediante las cuales obtenemos las diferentes tablas con los que observamos los datos esperados. Se componen de cinco pruebas que se ejecutan de forma correcta, y cinco pruebas que no se ejecutan de forma correcta.

PRUEBA 1:

Funcionamiento correcto:

```
var int a_
var bool b_______b____
var chars c
```

- Sus **tokens** generados serán:

```
< PalRes , var >
< PalRes , int >
< id , a_ >
< cr , - >
< PalRes , var >
< PalRes , bool >
< id , b_______b____ >
< cr , - >
< PalRes , var >
< PalRes , chars >
< id , c >
< cr , - >
< FinFich , eof >
```

- Su fichero de **parse**:

```
Des 1 2 6 10 50 52 2 6 11 50 52 2 6 12 50 52 5
```

- Su **tabla de símbolos**:

```
CONTENIDO DE LA TABLA # TS_General :
* LEXEMA : a_
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
* LEXEMA : b_______b____
  ATRIBUTOS :
  + tipo : logico
  + desplazamiento : 2
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
* LEXEMA : c
  ATRIBUTOS :
  + tipo : cadena
  + desplazamiento : 3
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
========================================
```

Funcionamiento con error:

```
var int a
var bool b
var c
```

- Sus **tokens** generados serán:

```
< PalRes , var >
< PalRes , int >
< id , a >
< cr , - >
< PalRes , var >
< PalRes , bool >
< id , b >
< cr , - >
< PalRes , var >
< id , c >
< cr , - >
< FinFich , eof >
```

- Su **tabla de símbolos**:

```
CONTENIDO DE LA TABLA # TS_General :
* LEXEMA : a
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
* LEXEMA : b
  ATRIBUTOS :
  + tipo : logico
  + desplazamiento : 2
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
========================================
```

- Su fichero de **parse**:

```
Des 1 2 6 10 50 52 2 6 11 50 52 2 6 12
```

- La **línea de error** generada por el terminal en la ejecución será:

```
ERROR Sintactico -->
Token_Actual : id en linea :2
```

PRUEBA 2:

Funcionamiento correcto:

```
// Prueba de comentario
var int a
a = 2768
var bool b
b = False
var chars c
c = "PDL"
```

- Sus **tokens** generados serán:

```
< cr , - >
< PalRes , var >
< PalRes , int >
< id , a >
< cr , - >
< id , a >
< Op , = >
< entero , 2768 >
< cr , - >
< PalRes , var >
< PalRes , bool >
< id , b >
< cr , - >
< id , b >
< Op , = >
< PalRes , False >
< cr , - >
< PalRes , var >
< PalRes , chars >
< id , c >
< cr , - >
< id , c >
< Op , = >
< cadena , "PDL" >
< cr , - >
< FinFich , eof >
```

- Su **tabla de símbolos**:

```
CONTENIDO DE LA TABLA # TS_General :
* LEXEMA : 'a'
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
--------- ----------
* LEXEMA : 'b'
  ATRIBUTOS :
  + tipo : logico
  + desplazamiento : 2
--------- ----------
* LEXEMA : 'c'
  ATRIBUTOS :
  + tipo : cadena
  + desplazamiento : 3
--------- ----------
================================
```

- Su fichero de **parse**:

```
Des 1 4 50 52 2 6 10 50 52 2 9 13 18 25 28 31 36 33
30 27 50 52 2 6 11 50 52 2 9 13 18 25 28 31 39 33 30
27 50 52 2 6 12 50 52 2 9 13 18 25 28 31 37 33 30 27
50 52 5
```

Funcionamiento con error:

```
// Prueba de comentario
var int a
a = False
var bool b
b = False
var chars c
c = "PDL"
```

- Sus **tokens** generados serán:

```
< cr , - >
< PalRes , var >
< PalRes , int >
< id , a >
< cr , - >
< id , a >
< Op , = >
< PalRes , False >
< cr , - >
< PalRes , var >
< PalRes , bool >
< id , b >
< cr , - >
< id , b >
< Op , = >
< PalRes , False >
< cr , - >
< PalRes , var >
< PalRes , chars >
< id , c >
< cr , - >
< id , c >
< Op , = >
< cadena , "PDL" >
< cr , - >
< FinFich , eof >
```

- Su **tabla de símbolos**:

```
CONTENIDO DE LA TABLA # TS_General :
* LEXEMA : 'a'
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
--------- ----------
====================================
```

- Su fichero de **parse**:

```
Des 1 4 50 52 2 6 10 50 52 2 9 13 18 25 28 31 39
33 30 27
```

- La **línea de error** generada por el terminal en la ejecución será:

```
ERROR Semantico --> asignacion mal formada,
en linea: 2
```

PRUEBA 3:

Funcionamiento correcto:

```
var int a
a = 5
var int b
b = 6
if(a>b) b = a
if(True) a = b
```

- Sus **tokens** generados serán:

```
< PalRes , var >
< PalRes , int >
< id , a >
< cr , - >
< id , a >
< Op , = >
< PalRes , var >
< PalRes , int >
< id , b >
< cr , - >
< id , b >
< Op , = >
< entero , 6 >
< cr , - >
< PalRes , if >
< Op , ( >
< id , a >
< Op , > >
< id , b >
< Op , ) >
< id , b >
< Op , = >
< id , a >
< cr , - >
< PalRes , if >
< Op , ( >
< PalRes , True >
< Op , ) >
< id , a >
< Op , = >
< id , b >
< cr , - >
< FinFich , eof >
```

- Su **tabla de símbolos**:

```
CONTENIDO DE LA TABLA # TS_General :
* LEXEMA : a
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
* LEXEMA : b
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 2
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
=======================================
```

- Su fichero de **parse**:

```
Des 1 2 6 10 50 52 2 9 13 18 25 28 31 36 33 30
27 50 52 2 6 10 50 52 2 9 13 18 25 28 31 36 33
30 27 50 52 2 7 25 28 31 34 42 33 29 31 34 42
33 30 27 13 18 25 28 31 34 42 33 30 27 50 52 2
7 25 28 31 38 33 30 27 13 18 25 28 31 34 42 33
30 27 50 52 5
```

Funcionamiento con error:

```
var int a
a = 5
var int b
b = 6
if(a) b = a
if(True) a = b
```

- Sus **tokens** generados serán:

```
< PalRes , var >
< PalRes , int >
< id , a >
< cr , - >
< id , a >
< Op , = >
< entero , 5 >
< cr , - >
< PalRes , var >
< PalRes , int >
< id , b >
< cr , - >
< id , b >
< Op , = >
< entero , 6 >
< cr , - >
< PalRes , if >
< Op , ( >
< id , a >
< Op , ) >
< id , b >
< Op , = >
< id , a >
< cr , - >
< PalRes , if >
< Op , ( >
< PalRes , True >
< Op , ) >
< id , a >
< Op , = >
< id , b >
< cr , - >
< FinFich , eof >
```

- Su **tabla de símbolos**:

```
CONTENIDO DE LA TABLA # TS_General :
* LEXEMA : a
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
* LEXEMA : b
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 2
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
=======================================
```

- Su fichero de **parse**:

```
Des 1 2 6 10 50 52 2 9 13 18 25 28 31 36 33 30
27 50 52 2 6 10 50 52 2 9 13 18 25 28 31 36 33
30 27 50 52 2 7 25 28 31 34 42 33 30 27
```

- La **línea de error**:

```
ERROR Semantico --> condición "if" no logica, en linea: 4
```

PRUEBA 4:

Funcionamiento correcto:

```
var int a
a = 5
a = a++
var int b
b = 6
if(a>b) b = a
if(False) a = b
switch(a)
{
 case 1: a = 1; break
 case 2: a = 2; break
 case 3: write("Hola")
 break
 case 4: prompt(a)
}
```

- Sus **tokens** generados:

```
< PalRes , var >
< PalRes , int >
< id , a >
< cr , - >
< id , a >
< Op , = >
< entero , 5 >
< cr , - >
< id , a >
< Op , = >
< id , a >
< Op , ++ >
< cr , - >
< PalRes , var >
< PalRes , int >
< id , b >
< cr , - >
< id , b >
< Op , = >
< entero , 6 >
< cr , - >
< PalRes , if >
< Op , ( >
< id , a >
< Op , > >
< id , b >
< Op , ) >
< id , b >
< Op , = >
< id , a >
< cr , - >
< PalRes , if >
< Op , ( >
< PalRes , False >
< Op , ) >
< id , a >
< Op , = >
< id , b >
< cr , - >
< PalRes , switch >
< Op , ( >
< id , a >
< Op , ) >
< cr , - >
< Op , { >
< cr , - >
< PalRes , case >
< entero , 1 >
< Op , : >
< id , a >
< Op , = >
< entero , 1 >
< Op , ; >
< PalRes , break >
< cr , - >
< PalRes , case >
< entero , 2 >
< Op , : >
< id , a >
< Op , = >
< entero , 2 >
< Op , ; >
< PalRes , break >
< cr , - >
< PalRes , case >
< entero , 3 >
< Op , : >
< PalRes , write >
< Op , ( >
< cadena , "Hola" >
< Op , ) >
< cr , - >
< PalRes , break >
< cr , - >
< PalRes , case >
< entero , 4 >
< Op , : >
< PalRes , prompt >
< Op , ( >
< id , a >
< Op , ) >
< cr , - >
< Op , } >
< cr , - >
< FinFich , eof >
```

- Su **tabla de símbolos**:

```
CONTENIDO DE LA TABLA # TS_General :
* LEXEMA : a
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
* LEXEMA : b
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 2
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
================================
```

- Su fichero **parse**:

```
Des 1 2 6 10 50 52 2 9 13 18 25 28 31 36
33 30 27 50 52 2 9 13 18 25 28 31 34 41 33
30 27 50 52 2 6 10 50 52 2 9 13 18 25 28
31 36 33 30 27 50 52 2 7 25 28 31 34 42 33
29 31 34 42 33 30 27 13 18 25 28 31 34 42
33 30 27 50 52 2 7 25 28 31 39 33 30 27 13
18 25 28 31 34 42 33 30 27 50 52 2 8 25 28
31 34 42 33 30 27 50 52 50 52 55 56 13 18
25 28 31 36 33 30 27 57 50 52 60 56 13 18
25 28 31 36 33 30 27 57 50 52 60 56 15 25
28 31 37 33 30 27 58 50 52 60 56 16 59 50
52 61 50 52 5
```

Funcionamiento con error:

```
var int a
a = 5
var int b
b = 6
if(a>b) b = a
if(False) a = b
switch(a)
{
  case 1: a = 1; break
  case 2: a = 2; break
  case 3: write("Hola")
  break
  case 4: prompt(True)
}
```

- Sus **tokens** generados:

```
< PalRes , var >
< PalRes , int >
< id , a >
< cr , - >
< id , a >
< Op , = >
< entero , 5 >
< cr , - >
< PalRes , var >
< PalRes , int >
< id , b >
< cr , - >
< id , b >
< Op , = >
< entero , 6 >
< cr , - >
< PalRes , if >
< Op , ( >
< id , a >
< Op , > >
< id , b >
< Op , ) >
< id , b >
< Op , = >
< id , a >
< cr , - >
< PalRes , if >
< Op , ( >
< PalRes , False >
< Op , ) >
< id , a >
< Op , = >
< id , b >
< cr , - >
< PalRes , switch >
< Op , ( >
< id , a >
< Op , ) >
< cr , - >
< Op , { >
< cr , - >
< PalRes , case >
< entero , 1 >
< Op , : >
< id , a >
< Op , = >
< entero , 1 >
< Op , ; >
< PalRes , break >
< cr , - >
< PalRes , case >
< entero , 2 >
< Op , : >
< id , a >
< Op , = >
< entero , 2 >
< Op , ; >
< PalRes , break >
< cr , - >
< PalRes , case >
< entero , 3 >
< Op , : >
< PalRes , write >
< Op , ( >
< cadena , "Hola" >
< Op , ) >
< cr , - >
< PalRes , break >
< cr , - >
< PalRes , case >
< entero , 4 >
< Op , : >
< PalRes , prompt >
< Op , ( >
< PalRes , True >
< Op , ) >
< cr , - >
< Op , } >
< cr , - >
< FinFich , eof >
```

- Su **tabla de símbolos**:

```
CONTENIDO DE LA TABLA # TS_General :
* LEXEMA : a
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
* LEXEMA : b
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 2
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
================================
```

- Su fichero **parse**:

```
Des 1 2 6 10 50 52 2 9 13 18 25 28 31 36
33 30 27 50 52 2 6 10 50 52 2 9 13 18 25
28 31 36 33 30 27 50 52 2 7 25 28 31 34 42
33 29 31 34 42 33 30 27 13 18 25 28 31 34
42 33 30 27 50 52 2 7 25 28 31 39 33 30 27
13 18 25 28 31 34 42 33 30 27 50 52 2 8 25
28 31 34 42 33 30 27 50 52 50 52 55 56 13
18 25 28 31 36 33 30 27 57 50 52 60 56 13
18 25 28 31 36 33 30 27 57 50 52 60 56 15
25 28 31 37 33 30 27 58 50 52 60 56 16
```

- **Línea de error**:

```
ERROR Sintactico -->
Token_Actual : True en
línea: 11
```

PRUEBA 5:

Funcionamiento correcto:

```
//Comentario de linea

var int s   // comentario final de linea
s = 555 //safafasf
var chars z

function int imprime (int x)
{
        var int h
        return z
}
function int FactorialRecursivo (int n, bool p)
{
        if (p && True) return FactorialRecursivo ( 5 , True )
}
var bool b
b = s > (s * 2)
```

- Sus **tokens** generados:

```
< cr , - >
< cr , - >
< PalRes , var >
< PalRes , int >
< id , s >
< cr , - >
< id , s >
< Op , = >
< entero , 555 >
< cr , - >
< PalRes , var >
< PalRes , chars >
< id , z >
< cr , - >
< cr , - >
< PalRes , function >
< PalRes , int >
< id , imprime >
< Op , ( >
< PalRes , int >
< id , x >
< Op , ) >
< cr , - >
< Op , { >
< cr , - >
< PalRes , var >
< PalRes , int >
< id , h >
< cr , - >
< PalRes , return >
< id , z >
< cr , - >
< Op , } >
< cr , - >
< PalRes , function >
< PalRes , int >
< id , FactorialRecursivo >
< Op , ( >
< PalRes , int >
< id , n >
< Op , , >
< PalRes , bool >
< id , p >
< Op , ) >
< cr , - >
< Op , { >
< cr , - >
< PalRes , if >
< Op , ( >
< id , p >
< Op , && >
< PalRes , True >
< Op , ) >
< PalRes , return >
< id , FactorialRecursivo >
< Op , ( >
< entero , 5 >
< Op , , >
< PalRes , True >
< Op , ) >
< cr , - >
< Op , } >
< cr , - >
< PalRes , var >
< PalRes , bool >
< id , b >
< cr , - >
< id , b >
< Op , = >
< id , s >
< Op , > >
< Op , ( >
< id , s >
< Op , * >
< entero , 2 >
< Op , ) >
< cr , - >
< FinFich , eof >
```

- Su **tabla de símbolos**:

```
CONTENIDO DE LA TABLA # TS_General :
* LEXEMA : 's'
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
--------- ----------
* LEXEMA : 'z'
  ATRIBUTOS :
  + tipo : cadena
  + desplazamiento : 2
--------- ----------
* LEXEMA : 'imprime'
  ATRIBUTOS :
  + tipo : function
  + desplazamiento : 4
  + Tam_parametros : 1
  + Tipo_parametro : entero
  + Retorno_funcion : entero
  + ptrTS_padre : TS_General
--------- ----------
* LEXEMA : 'FactorialRecursivo'
  ATRIBUTOS :
  + tipo : function
  + desplazamiento : 8
  + Tam_parametros : 2
  + Tipo_parametro : entero,logico
  + Retorno_funcion : entero
  + ptrTS_padre : TS_General
--------- ----------
* LEXEMA : 'b'
  ATRIBUTOS :
  + tipo : logico
  + desplazamiento : 12
--------- ----------
====================================
CONTENIDO DE LA TABLA # imprime :
* LEXEMA : 'x'
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
--------- ----------
* LEXEMA : 'h'
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 2
--------- ----------
====================================
CONTENIDO DE LA TABLA # FactorialRecursivo :
* LEXEMA : 'n'
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
--------- ----------
* LEXEMA : 'p'
  ATRIBUTOS :
  + tipo : logico
  + desplazamiento : 2
--------- ----------
====================================
```

- Su fichero **parse**:

```
Des 1 4 50 51 52 2 6 10 50 52 2 9 13 18 25 28 31
36 33 30 27 50 52 2 6 12 50 51 52 3 43 44 10 46
10 49 50 52 50 52 53 6 10 50 52 53 9 14 19 25 28
31 34 42 33 30 27 50 52 54 50 52 3 43 44 10 46
10 48 11 49 50 52 50 52 53 7 25 28 31 34 42 33
30 26 28 31 38 33 30 27 14 19 25 28 31 34 40 21
25 28 31 36 33 30 27 23 25 28 31 38 33 30 27 24
33 30 27 50 52 54 50 52 2 6 11 50 52 2 9 13 18
25 28 31 34 42 33 29 31 35 25 28 31 34 42 32 36
33 30 27 33 30 27 50 52 5
```

Funcionamiento con errores:

```
//Comentario de linea

var int s   // comentario final de linea
s = 555 //safafasf
var chars z

function int imprime (int x)
{
        var int h
        return z
}
function int FactorialRecursivo (int n, bool p)
{
        if (p && True) return FactorialRecursivo ( 5 , 5 )
}
var bool b
b = s > (s * 2)
```

- **Línea de error**:

ERROR Semantico --> Parametros mal insertados, son: entero,entero deberia ser : entero,logico, en linea: 13

- Sus **tokens**:

```
< cr , - >
< cr , - >
< PalRes , var >
< PalRes , int >
< id , s >
< cr , - >
< id , s >
< Op , = >
< entero , 555 >
< cr , - >
< PalRes , var >
< PalRes , chars >
< id , z >
< cr , - >
< cr , - >
< PalRes , function >
< PalRes , int >
< id , imprime >
< Op , ( >
< PalRes , int >
< id , x >
< Op , ) >
< cr , - >
< Op , { >
< cr , - >
< PalRes , var >
< PalRes , int >
< id , h >
< cr , - >
< PalRes , return >
< id , z >
< cr , - >
< Op , } >
< cr , - >
< PalRes , function >
< PalRes , int >
< id , FactorialRecursivo >
< Op , ( >
< PalRes , int >
< id , n >
< Op , , >
< PalRes , bool >
< id , p >
< Op , ) >
< cr , - >
< Op , { >
< cr , - >
< PalRes , if >
< Op , ( >
< id , p >
< Op , && >
< PalRes , True >
< Op , ) >
< PalRes , return >
< id , FactorialRecursivo >
< Op , ( >
< entero , 5 >
< Op , , >
< entero , 5 >
< Op , ) >
< cr , - >
< Op , } >
< cr , - >
< PalRes , var >
< PalRes , bool >
< id , b >
< cr , - >
< id , b >
< Op , = >
< id , s >
< Op , > >
< Op , ( >
< id , s >
< Op , * >
< entero , 2 >
< Op , ) >
< cr , - >
< FinFich , eof >
```

- Su fichero **parse**:

```
Des 1 4 50 51 52 2 6 10 50 52 2 9 13 18 25 28 31
36 33 30 27 50 52 2 6 12 50 51 52 3 43 44 10 46
10 49 50 52 50 52 53 6 10 50 52 53 9 14 19 25 28
31 34 42 33 30 27 50 52 54 50 52 3 43 44 10 46
10 48 11 49 50 52 50 52 53 7 25 28 31 34 42 33
30 26 28 31 38 33 30 27 14 19 25 28 31 34 40 21
25 28 31 36 33 30 27 23 25 28 31 36 33 30 27 24
```

- Su **tabla de símbolos**:

```
CONTENIDO DE LA TABLA # TS_General :
* LEXEMA : 's'
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
--------- ----------
* LEXEMA : 'z'
  ATRIBUTOS :
  + tipo : cadena
  + desplazamiento : 2
--------- ----------
* LEXEMA : 'imprime'
  ATRIBUTOS :
  + tipo : function
  + desplazamiento : 4
  + Tam_parametros : 1
  + Tipo_parametro : entero
  + Retorno_funcion : entero
  + ptrTS_padre : TS_General
--------- ----------
* LEXEMA : 'FactorialRecursivo'
  ATRIBUTOS :
  + tipo : function
  + desplazamiento : 8
  + Tam_parametros : 2
  + Tipo_parametro : entero,logico
  + Retorno_funcion : entero
  + ptrTS_padre : TS_General
--------- ----------
==================================
CONTENIDO DE LA TABLA # imprime :
* LEXEMA : 'x'
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
--------- ----------
* LEXEMA : 'h'
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 2
--------- ----------
==================================
CONTENIDO DE LA TABLA # FactorialRecursivo :
* LEXEMA : 'n'
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
--------- ----------
* LEXEMA : 'p'
  ATRIBUTOS :
  + tipo : logico
  + desplazamiento : 2
--------- ----------
========================================
```