Universidad Politécnica de Madrid
Escuela técnica superior de ingenieros informáticos

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software (DLSIS)

Procesadores de lenguajes

Práctica Procesadores de Lenguajes

JavaScript-PL

Grupo 81

Práctica Procesadores de Lenguajes

JavaScript-PL

Autores <u>Grupo 81</u>:
Víctor Nieves Sánchez
Alejandro Carmona Ayllón
Miguel Moreno Mardones

Resumen:
*Esta memoria resume el diseño del Procesador creado por el **grupo número 81**. Aquí se detallan nuestras correspondientes especificaciones del lenguaje **JavaScript-PL** (incluyendo características globales a todos los grupos), así como las características internas de nuestro Procesador, gramáticas empleadas y algunos casos de prueba para determinar el funcionamiento del mismo.*

Fecha:
27 de diciembre de 2018, Campus Montegancedo (Boadilla del Monte), Universidad Politécnica de Madrid.

Índice:

1. Objetivo:

La Práctica consiste en el diseño e implementación de un Procesador de Lenguajes, que realice el Análisis Léxico, Sintáctico y Semántico (incluyendo la Tabla de Símbolos y el Gestor de Errores), para un determinado lenguaje de programación denominado *JavaScript-PL*.

2. Especificación del grupo:

Siendo el **grupo** número **81**, se nos ha asignado las siguientes características:

- Sentencias: Sentencia repetitiva (**while**)
- Operadores especiales: **Pre-auto-incremento** (++ **como prefijo**)
- Técnica de Análisis Sintáctico: **Descendente recursivo**
- Comentarios: **Comentarios de línea** (//)
- Cadenas: Con **comillas dobles** (**“ “**)

Además, del resto de opciones se han elegido:

- Operador aritmético: **suma** (+)
- Operador relacional: **distinto** (**!=**)
- Operador lógico: **negación** (**!**)

3. Metodología de trabajo:

Todos los miembros del grupo han participado en todo momento en todos los aspectos de la práctica. Diseño teórico, implementación de código y realización de memorias.
El tiempo total de trabajo repartido a lo largo del curso es aproximadamente **40 horas**.

4. Introducción:

El proyecto consiste en el diseño e implementación de un *Procesador de Lenguajes*, en el que mediante los conceptos aprendidos en la asignatura ***Procesadores de Lenguajes*** y el uso de la programación aprendida a lo largo de la carrera, crearemos un *Analizador Léxico*, *Sintáctico* y *Semántico* (incluyendo las *Tablas de Símbolos* y el *Gestor de Errores*), para el lenguaje de programación pedido: *JavaScript-PL*.

Para ello trabajamos sobre una base común para todos los grupos y una parte específica de cada grupo, esto ya se ha detallado anteriormente en el punto *2.Especificación del grupo:* .

Se ha decidido usar *Java* como lenguaje en la práctica.

El diseño del paquete tiene varios scripts:

- *AnManager.java:* Es el controlador del proyecto. Es el encargado de llamar al resto de scripts y funciones.
- *AnalizadorSinSem.java:* Como su nombre indica, es la implementación del analizador sintáctico y del analizador semántico.
- *Errores.java:* Es el encargado de generar los errores obtenidos (si los hay).
- *InterfazFile.java:* Este archivo es el ejecutable, en el cual se diseña una interfaz gráfica.
- *Token.java:* Genera los tokens que va encontrando en el fichero fuente.
- *itemTS.java:* Genera las tablas de símbolos que se obtienen a partir del fichero fuente.

Para mayor facilidad de ejecución, se ha creado también un ejecutable llamado ***Analizador.jar*** en el directorio raíz.

En nuestra aplicación se deberá elegir un fichero fuente y un directorio donde crear los ficheros resultado: *lista de tokens, parse, tabla de símbolos, errores* y la *gramática* para la herramienta *Vast.*

5. Analizador léxico:

La principal tarea del analizador léxico será analizar el fichero fuente carácter a carácter y generar los tokens necesarios o errores encontrados, una vez esto se los trasmitirá al posterior analizador, el analizador sintáctico.

5.1. Tokens:

Se ha agrupado el conjunto de características tanto globales, como especiales a nuestro grupo y se ha diseñado el siguiente número de tokens:

<bool, _>
<function, _>
<if, _>
<int, _>
<print, _>
<prompt, _>
<return, _>
<string, _>
<var, _>
<while, _>
<true, _>
<false, _>
<al, _> (abro llave)
<cl, _> (cierro llave)
<ID, posTS>
<entero, valor>
<Cadena, lexema>
<igual, _>
<negación, _>
<distinto, _>
<suma, _>
<Preincremento, _>
<PuntoComa, _>
<Coma, _>
<ap, _> (abro paréntesis)
<cp, _> (cierro paréntesis)
<EOF, _>

5.2. Gramática del lenguaje:

Se ha diseñado la siguiente gramática que permita implementar los tokens anteriores:

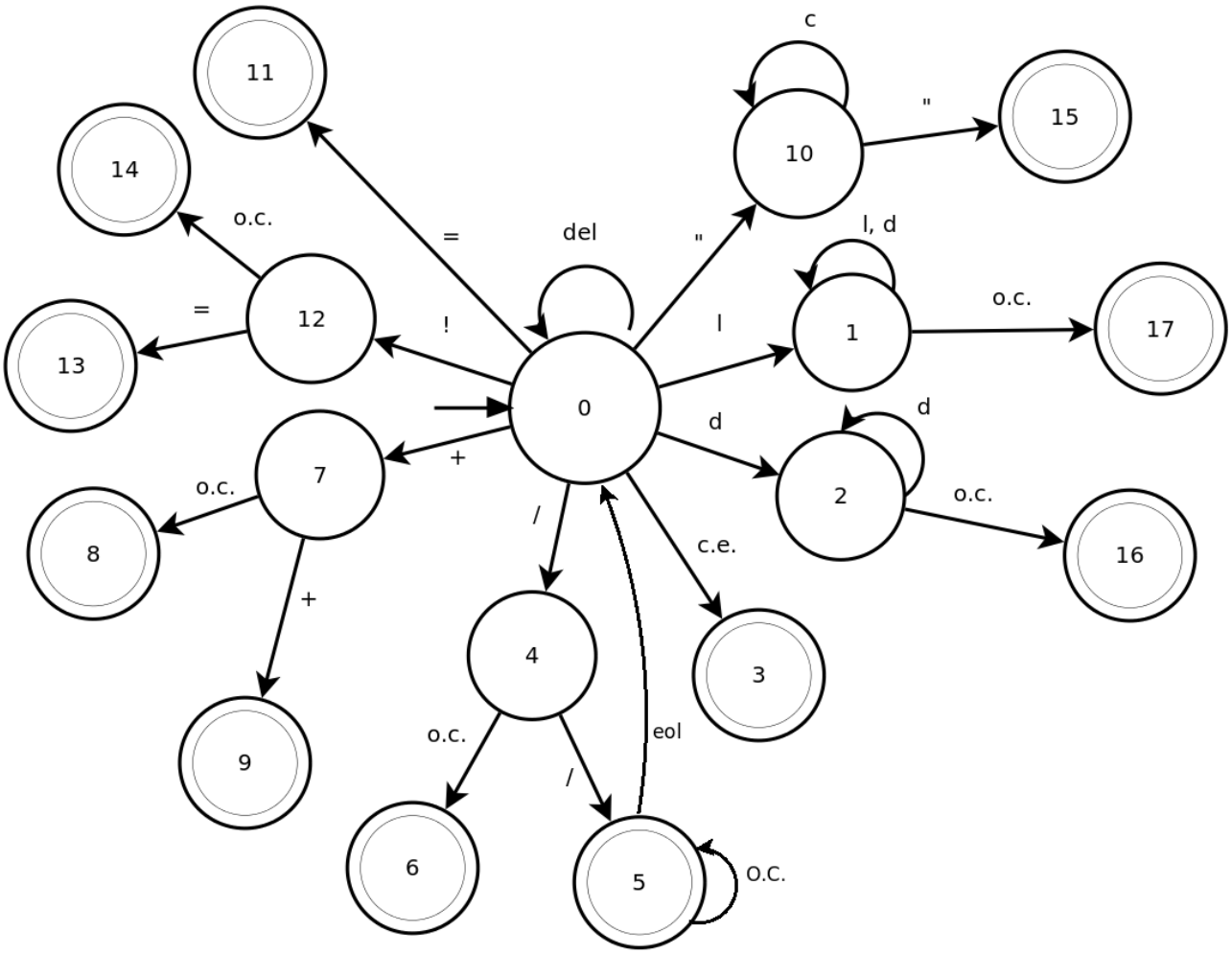
S → *d*el S | l A | d B | / C | + D | ! E | “ F | ce | =
A → l A | d A | λ
B → d B | λ
C → / | λ
D → + | λ
E → = | λ
F → c F | “

Donde los caracteres utilizados toman los valores:

l = ({A…Z}, {a…z})
d = {0...9}
c = cualquier carácter - { “ }
ce = { *{ } ( ) ;* , }
del = { ~~*b*~~, < *tab* >, < *eol* >, < *eof* >}
oc = otra cosa distinta de lo que se espera

5.3. Autómata:

El autómata obtenido tras la gramática es:



5.4. Acciones Semánticas:

0:0 *leer*
0:1 *lexema = l ; leer*
1:1 *lexema = lexema ⊕ l/d ; leer*
1:17 *if (cod = buscarPalabra_clave (lexema)) then GenToken (Palabra_clave, cod);*
Elseif (pos = buscarTS (lexema)) then GenToken (lexema, pos);
Else InsertToken (lexema, pos) GenToken (lexema, pos);
0:2 *valor = d ; leer*
2:2 *valor = valor*10 + d; leer*
2:16 *if (valor <= 32767) then GenToken (entero,valor);*
Else error (“numero fuera de rango”);
0:3 *GenToken (ce, cod);*
0:4 *lexema = / ; leer;*
4:5 *lexema = lexema ⊕ / ;*

4:6 *GenToken (op_arit, 2);*
0:7 *lexema = + ; leer;*
7:8 *lexema = lexema ⊕ + ; GenToken (op_arit, 3);*
7:9 *GenToken (op_arit, 1)*
0:12 *lexema = ! ; leer*
12:13 *lexema = lexema ⊕ = ; GenToken (distinto, -);*
12:14 *GenToken (negación, -);*
0:11 *GenToken (igual, -);*
0:10 *lexema = “ ; leer*
10:10 *lexema = lexema ⊕ c ; leer*
10:15 *lexema = lexema ⊕ “ ; GenToken(contante_cadena, lexema);*

6. Tablas de Símbolos:

Las tablas que generaría nuestro programa serian:

- *Tabla de símbolos Global*: tabla global de información.
- *Tabla de símbolos Fn*: tabla de símbolos de funciones. Tabla local.

Tabla de Símbolos Global:

Índice	Lexema	Tipo	Argumentos	Devuelto
…	…	…	…	…

Tabla de Símbolos Fn (funciones):

Índice	Lexema	Tipo	Argumentos	Devuelto
…	…	…	…	…

7. Analizador Sintáctico:

La principal tarea del analizador sintáctico será analizar programa fuente y determinar si es correcta dicha sintaxis, para ello recibe tokens del analizador léxico, los trata y una vez acabado el proceso vuelve a pedir otro token hasta completar el fichero.

La gramática empleada en el analizador sintáctico difiere de la usada anteriormente en el analizador léxico ya que en esta poseemos una gramática mucho más extensa debido a la complejidad del analizador que es mayor.

Como nuestro análisis es ***descendiente recursivo***, se debe cumplir que es necesario una ***gramática no ambigua***, que **no presente *recursividad por la izquierda*** y que esté ***factorizada por la izquierda***. Teniendo esto en cuenta, se ha desarrollado la siguiente gramática:

P → *B P* [1] | *F P* [2] | *eof* [3]
B → *var T id ;* [4] | *if (R) S* [5] | *while (R) { C }* [6] | *S* [7]
T → *int* [8] | *bool* [9] | *string* [10]
S → *id Sa* [11] | *return X;* [12] | *print (R);* [13] | *prompt (id);* [14]
Sa → *= R ;* [15] | *!= R;* [16] | *(L) ;* [17]
X → *R* [18] | *λ* [19]
C → *B C* [20] | *λ* [21]
F → *function H id (A) { C }* [22]
H → *T* [23] | *λ* [24]
A → *T id K* [25] | *λ* [26]
K → *, T id K* [27] | *λ* [28]
L → *R Q* [29] | *λ* [30]
Q → *, R Q* [31] | *λ* [32]
R → *U Ra* [33]
Ra → *= R* [34] | *!= R* [35] | *λ* [36]
U → *V Ua* [37]
Ua → *+ U* [38] | *λ* [39]
V → *id Va* [40] | *entero* [41] | *cadena* [42] | *+ + id* [43] | *true* [44] | *false* [45] | *(R)* [46] | *! Vb* [47]
Va → *(L)* [48] | *λ* [49]
Vb → *true* [50] | *false* [51] | *id* [52]

7.1. Tablas *First* y *Follow*:

Símbolos	First	Follow
P	eof var if while function id return print prompt	$
S	id return print prompt	cl eof function var if while id return print prompt
Sa	= != (	cl eof function var if while id return print prompt
X	id entero cadena + true false (! λ	;
B	var if while id return print prompt	cl eof function var if while id return print prompt
F	function	eof var if while function id return print prompt
C	var if while id return print prompt λ	}
H	int bool string λ	Id
A	int bool string λ	)
T	int bool string	Id
K	, λ	)
Q	, λ	)
Ra	= != λ	; ,)
Ua	+ λ	= != ; ,)
U	id entero cadena + true false (!	= != ; ,)
V	id entero cadena + true false (!	+ = != ; ,)
R	id entero cadena + true false (!	; ,)
Va	(λ	+ = != ; ,)
L	id entero cadena + true false (! λ	)
Vb	true false id	+ = != ; ,)

Donde *var = variable, eof = end of file* e *id = identificador*

7.2. Comprobación de la gramática:

Para comprobar que la gramática anterior es una gramática válida, se ha decidido mostrar la tabla *LL(1)*, (documento anexo *TablaLL1.xlsx*) en la cual se verá a simple vista si la gramática es válida o no. Si no fuese válida, se encontraría en alguna o varias celdas dos movimientos posibles, para un mismo símbolo.

Como no es así, podemos afirmar que esta gramática *es válida*.

8. Analizador semántico:

Este analizador carece de recuperación de errores, ya que sus errores no impiden el análisis del código.

Comprueba si los tipos de variable asignados, así como los tipos de variable utilizados tanto en las operaciones aritméticas, lógicas como relacionales o los valores devueltos en las funciones son todos correctos, en caso de que esto no ocurra se procederá a lanzar un mensaje de error.

8.1. Traducción dirigida por la Sintaxis:

P' -> *[Inicializamos TSL, TSG y TablasSimbolos]* P *[Añadimos TSG a TablasSimbolos]*
P -> B P | F P | eof

B -> var T id ; *[if TSL != NULL buscamos id en las tabla de símbolos local, si la encontramos lanzamos error: ("Variable ya declarada"). Si no: Insertamos variable id, con T.tipo en la TSL o TSG en caso de que TSL == null]*

B -> if (R *[Si R.tipo es distinto "Bool" lanzamos Error]*) S

B -> S

B -> while (R [*Si R.tipo es distinto "Bool" lanzamos Error]*) { C }

T -> int *[T.tipo = Entero]*

T -> chars *[T.tipo = chars]*

T -> bool *[T.tipo = bool]*

S -> id Sa *[Si Sa.tipo == Null, se ha llamado una función, comprobamos si la función está en la tabla de símbolos y si se ha llamado con los argumentos correspondientes : id.argumentos == Sa.argumentos. Si no se ha llamado a una función comprobamos si existe la variable, si no existe la consideramos global entera y comprobamos id.tipo != Sa.tipo then error]*

S -> return X ; *[Si TSL == null no estamos en una función, por lo que lanzamos error. Si no: Comprobamos si la funcionactual.tipo == X.tipo, si no, error("No devuelve el tipo correcto"). También hacemos: funcionactual.yadevuelto = true]*

S -> print (R) ;

S -> prompt (id *[Buscamos si la variable está declarada y si no es de tipo bool, si no está declarada la metemos en la tabla global como entera]*) ;

Sa -> = R ; *[Sa.tipo = R.tipo]*

Sa -> (L) ; *[Sa.Argumentos = L.argumentos y Sa.tipo = Null]*

X -> R *[X.tipo = R.tipo]*

X -> lambda *[X.tipo = void]*

C -> B *[Si TSL != Null entonces comprobamos si se ha ejecutado ya el return o no. Si se ha ejecutado ya el return de la función, lanzamos aviso de que lo que ponga no se va a ejecutar]* C

C -> lambda

F -> function H id *[Buscamos id en la TSG, si está lanzamos error: ("Función ya declarada"), si no: funcionactual = id; TSL = creamos nueva TSL; y funcionactual.tipodevuelto = H.tipo]*(A) { C } *[if funcionActual.yaDevuelto == false lanzamos error ("No se ha devuelto nada"). Añadimos TSL a TablasSimbolos]*

H -> T *[H.tipo = T.tipo]*

H -> lambda *[H.tipo = void]*

A -> T id *[Añadir id con id.tipo = T.tipo a TSL (es un argumento de funcion) y funcionActual.argumentos++]* K

A -> lambda *[funcionActual.argumentos = 0]*

K -> , T id *[Añadir id con id.tipo = T.tipo a TSL (es un argumento de funcion) y funcionActual.argumentos++]* K

K -> lambda

L -> R *[LArgumento.add (R.tipo)]* Q

L -> lambda

Q -> , R [*LArgumento.add (R.tipo)]* Q

Q -> lambda

R -> U *[R.tipo = U.tipo]* Ra *[Si RaRel y U.tipo =! "Entero" error; R.tipo = "Bool"; RaRel = false]*

Ra -> = *R [RaRel= true y si U.tipo != "Entero" error]*

Ra -> != R [RaRel= true y si U.tipo != "Entero" error]

Ra -> lambda

U -> V *[U.tipo = V.tipo]* *Ua [Si UaSuma y V.tipo == Ua.tipo == "Entero" else error ; UaSuma= false]*

Ua -> + U *[UaSuma = true y si V.tipo != "Entero" error else Ua.tipo = "Entero"]*

Ua -> lambda

V -> id Va *[Si Va.tipo == Null hemos llamado a una función por lo que comprobamos si id se encuentra en la tabla global, si no, error (“No se ha definido la función”), si se encuentra comprobamos los argumentos con VaArgumentos y con funcionActual (o TSL) (En caso de que sea una llamada recursiva o con TSG en caso de que no sea llamada recursiva. Si no se ha llamado a una función comprobamos si existe id en la tabla correspondiente]*

V -> entero *[V.tipo = Entero]*

V -> cadena *[V.tipo = chars]*

V -> ++ id *[buscamos id en la tabla de símbolos, si no está error, si está comprobamos si es de tipo entero]*

V -> true *[V.tipo = bool]*

V -> false *[V.tipo = bool]*

V -> (R *[V.tipo = R.tipo]*)

V → ! Vb *[V.tipo = Vb.tipo]*

Va -> lambda *[Va.tipo = null]*

Va -> (L) *[VaArgumentos = LArgumentos]*

Vb -> true *[Vb.tipo = bool]*

Vb -> false *[Vb.tipo = bool]*

Vb -> id *[Vb.tipo = BuscarTS(id.pos)]*

9. Errores:

Los errores se van tratando en cada parte de la ejecución en paralelo con el resto de operaciones. Esto quiere decir que, se pueden encontrar errores tanto en el léxico, sintáctico o en el semántico.

Un error en el analizador léxico sería, por ejemplo un tipo de token desconocido. Pero hay que tener en cuenta que, las palabras clave no identificadas (por ejemplo, en nuestro caso, un *for*) serán tokens *id* (identificadores).

Los errores del analizador sintáctico serán del tipo: *“te falta un ‘;’ ”*, *“te falta un ‘(‘ “*, etc

Mientras que los errores del analizador semántico serán errores del tipo “*una variable de tipo entero no puede tener valor bool”*, etc.

10. Pruebas:

10.1. Pruebas de éxito:

10.1.1. Prueba 1:

Código fuente:

```
var int c;
c = 100;
function int b (){
	var int c;
	var string s;
	s = "$en una funcion$";
	// este comentario no deberia salir
	c = 3;
	if (true)
		n = 4;
	return c;
}
```

Tokens generados:

```
<var,>
<int,>
<ID,c>
<PuntoComa, >
<ID,c>
<igual,>
<Entero,100>
<PuntoComa, >
<function,>
<int,>
<ID,b>
<ap, >
<cp, >
<al, >
<var,>
<int,>
<ID,c>
<PuntoComa, >
<var,>
<string,>
<ID,s>
<PuntoComa, >
<ID,s>
<igual,>
<Cadena,"$en una funcion$">
<PuntoComa, >
<ID,c>
<igual,>
<Entero,3>
<PuntoComa, >
<if,>
<ap, >
<true,>
<cp, >
<ID,n>
<igual,>
<Entero,4>
<PuntoComa, >
<return,>
<ID,c>
<PuntoComa, >
<cl, >
<EOF, >
```

Tabla de símbolos:

```
TABLA GLOBAL #1 :

  * LEXEMA: 'c'
        + Tipo: 'Entero'
        + Desplazamiento: '0'

  * LEXEMA: 'n'
        + Tipo: 'Entero'
        + Desplazamiento: '2'

  * LEXEMA: 'b' (Funcion)
        + TipoDevuelto: 'Entero'
        + NoParametros: '0'
```

```
TABLA DE LA Funcion b #2 :

  * LEXEMA: 'c'
        + Tipo: 'Entero'
        + Desplazamiento: '0'

  * LEXEMA: 's'
        + Tipo: 'string'
        + Desplazamiento: '2'
```

Archivo de errores:

```
*No se han detectado errores en el codigo*
```

Gramática para el *Vast:*

```
Terminales = { var id ; if =  {  } (  ) ! != +  ++ , function int string bool
true false return print prompt entero cadena while eof }
NoTerminales = { P B T S Sa X C F H A K L Q R Ra U Ua V Va Vb }

Axioma = P

Producciones = {
P -> B P
P -> F P
P -> eof
B -> var T id ;
B -> if ( R ) S
B -> while ( R ) { C }
B -> S
T -> int
T -> bool
T -> string
S -> id Sa
S -> return X ;
S -> print ( R ) ;
S -> prompt ( id ) ;
Sa -> = R ;
Sa -> != R ;
Sa -> ( L ) ;
X -> R
X -> lambda
C -> B C
C -> lambda
F -> function H id ( A ) { C }
H -> T
H -> lambda
A -> T id K
A -> lambda
K -> , T id K
K -> lambda
L -> R Q
L -> lambda
Q -> , R Q
Q -> lambda
R -> U Ra
Ra -> = R
Ra -> != R
Ra -> lambda
U -> V Ua
Ua -> + U
Ua -> lambda
V -> id Va
V -> entero
V -> cadena
V -> ++ id
V -> true
V -> false
V -> ( R )
V -> ! Vb
Va -> ( L )
Va -> lambda
Vb -> true
Vb -> false
Vb -> id
}
```

Parse obtenido:

```
Descendente 1 4 8 1 7 11 15 33 37 41 39 36 2 22 23 8 26 20 4 8 20
 4 8 20 4 10 20 7 11 15 33 37 42 39 36 20 7 11 15 33
37 41 39 36 20 5 33 37 44 39 36 11 15 33 37 41 39 36 20 7
 12 18 33 37 40 49 39 36 21 3
```

Árbol obtenido de *Vast:*

Como solo se nos pedía toda la información (tokens, tabla de símbolos, parse, gramática, árbol y errores) de una sola prueba, el resto de pruebas solo se enseñará el

código fuente. Todas las pruebas aquí mostradas han sido testeadas por nosotros, y no se generan errores y se muestra el árbol correctamente.

10.1.2. Prueba 2:

Código fuente:

```
var string a;

//distintos tipos de if's

if (!true)
        a = "dentro true 1";

if ( ! false )
        a = "dentro false 1";

var string b;
if (true!=true)
        b = "dentro true 2";

if (false != false )
        b = "dentro false 2";
```

10.1.3. Prueba 3:

Código fuente:

```
var string s;
s = "esto se va a imprimir";
var int n;
n = 1;
print(n + s);
var string s2;
s = "esto es el final";
n = 2;
print(n + s2);
```

10.1.4. Prueba 4:

Código fuente:

```
var int n;
var bool t;
t = false;
while (!t){
        t = !t;
        if (t)
                n = ++n;
}
prompt(n);
```

10.1.5. Prueba 5:

Código fuente:

```
//un while
var bool a;
var bool b;
var int d;
var int c;
c = 1;
a = true;
b = false;
while ( a!=b ){
        d = ++c;
        if (c = d)
                b = true;
}
```

10.1.6. Notas:

Todos los casos de prueba vistos anteriormente se encuentran en el código entregado, en la carpeta *Pruebas/Pruebas Correctas/* .

Como se ha mencionado anteriormente, solo se pedía mostrar los resultados de un solo caso. Nosotros hemos elegido el primer caso por ser el más “completo” que se nos ha ocurrido. Si se desea probar alguno de los códigos anteriores, solo se tiene que ejecutar la aplicación y ver los resultados en la carpeta *Resultados/* .

La gramática que se usa en el parse se generará también por cada código fuente que se pruebe, pero esta no cambiará nunca, ya que siempre es la misma gramática.

10.2. Pruebas de errores:

10.2.1. Prueba 1:

Código fuente:

```
var int n;
n = true;
var bool b;
b = "string";
```

Fichero errores:

```
->Error en Analizador semantico: No se puede asignar un valor de tipo Bool a la variable n de tipo Entero
->Error en Analizador semantico: No se puede asignar un valor de tipo string a la variable b de tipo Bool
```

En esta prueba se ha intentado asignar valores incorrectos a variables de tipo, por ejemplo a la variable *n entera* se le intenta asignar el valor *true booleano*, lo cual nos devuelve un error.

10.2.2. Prueba 2:

Código fuente:

```
var int a;
var int b;
//prueba for
a = 2;
var string s;
s = "estoy dentro del for";
for (a; a != 3; ++a){
        print(s);
}
```

Fichero errores:

```
->Error en Analizador sintactico: Se esperaba ')' y se ha recibido el token <PuntoComa, > aqui
```

Este error muestra que el analizador sintáctico estaba esperando un ‘ *)* ‘, esto es debido a que el *for* no está implementado en nuestro código. Como no es una *palabra clave* para nosotros, el *for* se toma como un *id* (*identificador*), por lo que genera el token < *ID, for*>. El error que se muestra es por que al leer “*for(a”* está esperando cerrar el paréntesis, ya que no se entiende el que estamos dentro de la condición de un *for*.

10.2.3. Prueba 3:

Código fuente:

```
var int s;
function  s(int a, int s){ //no tiene tipo
        a = 4 //falta un ;
```

```
        var string s;
        s = "salida";
        return a;
}
```

Fichero errores:

```
->Error en Analizador semantico: La Funcion s ya se ha declarado anteriormente
->Error en Analizador sintactico: Se esperaba ';'. Se ha recibido el token <var,> aqui
->Error en Analizador sintactico: No se permite el token <EOF, > aqui
->Error en Analizador sintactico: Se esperaba '}'. Se ha recibido el token <EOF, > aqui
```

Como la variable *s* ya estaba antes, vemos que nos salta el error pertinente de que ya se ha declarado antes. También vemos que falta un ‘ ; ‘ , el cual también se muestra, indicando el siguiente token que se ha encontrado, en este caso <*var,* >. El resto de errores, vienen arrastrados por el segundo error, ya que hace que el analizador sintáctico se “pierda”.

10.2.4. Prueba 4:

Código fuente:

```
/* este comentario no esta */
var s; //variable sin tipo
var int d //falta el ;
a = 5; //esto funciona pq cualquier variable no definida se inicializa bien
return ++a;
```

Fichero errores:

```
->Error en Analizador lexico: No se reconoce el caracter ( * )
->Error en Analizador lexico: No se reconoce el caracter ( * )
->Error en Analizador sintactico: No se permite el token <div, > aqui
```

Nosotros no hemos implementado los comentarios de bloque (‘/* */‘), por eso se muestran los dos primeros errores, ya que el símbolo ‘*‘ no existe en nuestra gramática, mientras que el símbolo ‘ / ‘ si hemos hecho que aparezca como token aunque no lo contemplamos en en analizador sintáctico ni semántico. El motivo de que exista el token <*div,* > (*división*) es por que al tener el comentario de línea (‘ // ‘), en nuestro autómata era fácil el implementarlo, el problema es que a la hora de hacer la gramática del analizador sintáctico se nos olvidó contemplarlo, y nos dimos cuenta demasiado tarde, ya que añadirlo sería cambiar demasiadas cosas. Como ya tenemos la operación aritmética *suma* (+), el resto es opcional, por lo que no supone un problema.

10.2.5. Prueba 5:

Código fuente:

```
var string s;
s = "cadena uno";
```

var string s;
s = "cadena dos";

Fichero errores:

```
|->Error en Analizador semantico: La variable s ya se ha declarado anteriormente
```

Vemos que tanto léxicamente como sintácticamente el código no tiene errores. El problema es que el analizador semántico ha visito que la variable *s* se ha declarado dos veces en el mismo bloque de función (en este caso, la función es el programa principal), por eso ha lanzado el error de que ya se ha declarado anteriormente.

10.2.6. Notas:

Todos los casos de prueba mostrados en esta sección se encuentran junto al código fuente, en la carpeta *Pruebas/Pruebas Erroneas/* .

Estas pruebas imprimen todo, es decir, se muestran errores (los que ya se han enseñado anteriormente), tokens, tabla de símbolos, gramática (que ya hemos dicho que no cambia nunca, ni si quiera cuando son casos de fallo) y parse. Todos los ficheros de resultado demuestran lo que ha podido o no hacer nuestro Procesador de Lenguajes, es decir, en el caso del *for* se muestra en los tokens que se ha creado un token < *ID*, *for*>, pero este no es una palabra clave, ya que no ha sido contemplado. También se muestra el parse, pero este o no es correcto, o tiene algún fallo (devuelve un token en lugar de un número, significando que ese token no se “ha entendido”).

11. Resultados:

Con esta practica se ha conseguido implementar el procesador deseado, pero no completo (todo tipo de operaciones y palabras clave) del lenguaje *JavaScript-PL*.

Se ha aprendido el funcionamiento de cada una de las partes de un Procesador de Lenguajes, la manera de crear gramáticas válidas para las diferentes partes del mismo y, también se han mejorado en relación a nuestra manera de programar y forma de diseñar algoritmos en en lenguaje de programación (en este caso Java).

12. Conclusiones:

Esta práctica ha servido para fortalecer los conocimientos adquiridos en la asignatura *Procesadores de Lenguajes*. Gracias a lo aprendido en clase y los conocimientos adquiridos en programación a lo largo de la carrera, se ha implementado sin problema un programa que muestre si es correcto o no cierto código fuente en el lenguaje de programación *JavaScript-PL*.

La mayor dificultad encontrada fue el diseño del algoritmo base en el analizador sintáctico y semántico, y la realización de una interfaz gráfica para el programa.