

**Procesadores de Lenguajes**

**Analizador Léxico**

**Docente**: Cristian Pratz

**Integrantes**:

Geronimo Ratcliffe

Javier Villagras

Santiago Russo

Fecha: 13/09/2013

Versión: 2.0

Tabla de Revisiones

|  |  |  |  |
| --- | --- | --- | --- |
| Fecha | Versión | Descripción | Autor |
| 09/09/2013 | 1.0 | Creación del Documento | Santiago Russo |
| 13/09/2013 | 1.1 | Actualización de los casos de prueba | Santiago Russo |
| 25/09/2013 | 2.0 | Modificación de los Tokens | Santiago Russo |

Contents

[I: Descripción del Trabajo PRÁCTICO 4](#_Toc366854596)

[1.1 Autómata 4](#_Toc366854597)

[1.2 Expresión Regular 5](#_Toc366854598)

[1.3 Tokens Posibles 5](#_Toc366854599)

[II: Implementación 6](#_Toc366854600)

[2.1 Restricciones 7](#_Toc366854601)

[2.2 Tabla de Simbolos 8](#_Toc366854602)

[III: Casos de Prueba 9](#_Toc366854603)

[3.1 Test 9](#_Toc366854604)

[3.2 busquedaBinaria 10](#_Toc366854605)

[3.3 fibonacciTest 11](#_Toc366854606)

[3.4 ordenVector 12](#_Toc366854607)

[3.5 palabrasReservadas 12](#_Toc366854608)

[Indice 14](#_Toc366854609)

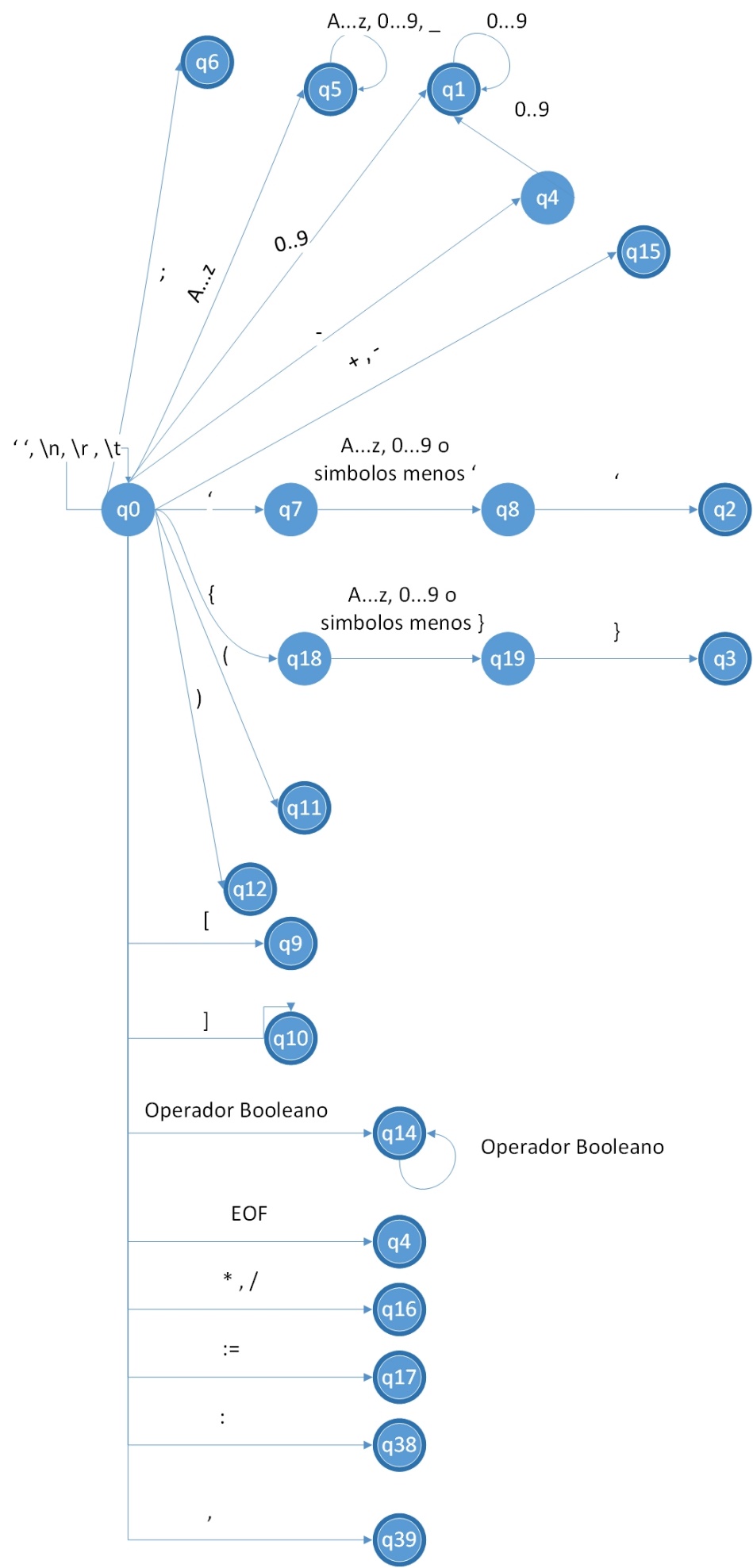
# I: Descripción del Trabajo PRÁCTICO

## 1.1 Autómata

Se crea un autómata finito determinístico para modelar los tipos de componentes léxicos a aceptar.

Se utiliza la técnica de “lookahead” (o *peeking*) de próximo carácter, para avaluar que tipo de lexema se está leyendo del archivo, y luego, categorizar el tipo de *Token* a devolver.

En cuanto a los lexemas no aceptados por el autómata diagramado, serán aceptados por los estados trampas (que se encuentran fuera de los estados diagramados, para evitar su complejidad). Serán tratados como el tipo de *Token* “Error”.



## 1.2 Expresión Regular

L (M) = (0|1|…|9)+ | -(0|1|…|9)+ |

‘Caracteres menos la comilla’ |

{Caracteres menos la llave} |

EOF |

(A|B|…|y|z) (A|B|…|z|0|1|…|9|\_)\* | (Incluye las palabras reservadas)

; |

[ |

] |

( |

) |

= | … | < | >= | < = | <> |

+ | - |

\* | / |

:= |

: |

,

## 1.3 Tokens Posibles

Los *Token* posibles son:

* *NUMERO*(1) //Enteros
* *CADENA*(2) //Strings
* *SYMBOLO*(3) //Ya no se usa. Se deja para posibles extensiones
* *EOF*(4) //Fin de Archivo
* *ID*(5)
* *ENDLINE*(6) //;
* *BOOLEANO*(7) //true o false
* *PALABRARESERVADA*(8) //ya no se usa
* *CORCHETE\_APERTURA*(9) // [
* *CORCHETE\_CIERRE*(10) // ]
* *PARENTESIS\_APERTURA*(11) // (
* *PARENTESIS\_CIERRE*(12) // )
* *TIPO*(13)
* *OPERADOR\_BOOLEANO*(14) // = <>
* *OPERADOR\_BOOLEANO\_E*(15) // > < >= <=
* *SUMA\_RESTA*(16) // + -
* *MULT\_DIV*(17) // \* /
* *ASIGNACION*(18) // :=
* //Arranco los tipos para las palabras reservadas
* *AND*(19)
* *OR*(20)//para las palabras logicas
* *BEGIN*(21)
* *BY*(22)//para el pasaje de parametros byval o byref
* *CONST*(23) //indica si es constante o variable
* *DO*(24)
* *ELSE*(25)
* *END\_FUNC*(26)
* *END\_IF*(27)
* *END\_PROC*(28)
* *END\_WHILE*(29)
* *FUNCTION*(30)
* *IF*(31)
* *CONST\_IGUAL*(32) // =
* *NOT*(33)
* *PROCEDURE*(34)
* *READ*(35)
* *SHOW*(36) //aplicapara SHOW y SHOWLN
* *THEN*(37)
* *VAR*(38)
* *WHILE*(39)
* *DEF\_TIPO*(40) //esto es para el :
* *COMMA*(41) // ,
* *ERROR*(-1);

# II: Implementación

Para la implementación creamos una clase llamada **LexicAnalyzer** la inicializamos en su constructor con el path al archivo a analizar o se podría inicializar con el objeto File ya instanciado. Esta clase posee una función **getToken()** la cual retornara el siguiente *token* con el lexema encontrado en el archivo. Existen 40 tipos de Tokens que pueden ser retornados como ya se vio en el ítem anterior.

La clase **LexicAnalyzer** al recibir una petición **getToken()** comienza a analizar el archivo, para ello comienza por omitir todos aquellos símbolos que pueden ser omitidas hasta encontrarse con el primer caracter válido para el comienzo de un *token*. Una vez encontrado dicho caracter este identifica el tipo de lexema a tratar (de los 40 indicados arriba) y se trata con la función tratar correspondiente a ese tipo (tratarPalabra, tratarCadena, etc)**.** Las funciones del helper retornarán el *token* con el **tipo** correspondiente, su lexema y sus coordenadas (fila y columnas).

En caso de que se detecte algún error a nivel léxico la función tratar correspondiente retornara un *token* de error junto con un número de tipo de error, en algunos casos un mensaje asociado y las coordenadas. Estos errores son los números de error:

1. Error de asignación numérico, caracter(es) invalido(s).
2. Error de asignación de operador, operador **‘!’** invalido.
3. Error cadena no finalizada apropiadamente, falta **'** (No se permiten **\n** en cadenas)
4. Error de lexema por caracteres inválidos
5. Mal uso de palabra reservada (**fin**)
6. Error al instanciar un Token por tipo invalido. (Ej quiero instanciar una Palabra Reservada con lexema = 38)

La clase **LexicAnalyzer** hace uso de un objeto de clase **FileReader**, el cual es utilizado para la lectura del archivo. Esta clase posee dos parámetros muy utilizados, los cuales son:

**read():** el cual posiciona el puntero en el siguiente carácter del archivo, almacena y devuelve el valor de este.

**peek():** el mismo retorna el siguiente carácter que se encuentra en el archivo pero no modifica la posición del puntero.

Luego, para la navegación en la Estructura de datos de los *Tokens*, se implementó un Patron de Visitor. Se crearon las interfaces de Vistable y Visitor y se creó una clase **TokenVisitor** que implementa los métodos de la interfaz para poder navegar en la estructura de los *Tokens*, accediendo a su lexema para cada caso. En esta instancia del TP el visitor solo muestra en consola el lexema del Token y en el caso de Error muestra todos sus datos.

Las clases y procedimientos también se encuentran documentadas utilizando el plugin de Eclipse para generar Javadocs, estos se pueden consultar abriendo el archivo index que se encuentra en la carpeta docs del proyecto.

## 2.1 Restricciones

Se definieron las siguientes restricciones:

* Al ser un lenguaje **case-insensitive** se optó por una solución de diseño que pase a lower-case cada uno de los caracteres del file, excepto las cadenas que se muestran por pantalla, evitando así complejidades futuras.
* Al tratar las palabras reservadas true y false se optó por un mecanismo que reconozca las mismas como tipos Booleanos en lugar de palabras reservadas en general.
* Las cadenas cortan solo cuando se lee un ’, permitiendo así meter \n dentro de las mismas.
* Para la declaración de variables no se permite valores numéricos al principio de las mismas, por ejemplo, “123pepe” será considerado como error.
* Existen palabras reservadas a las cuales no se las puede utilizar como variables. Siendo estas "and", "begin", "byref", "byval", "const", "do", "else", "end-func", "end-if", "end-proc", "end-while", "false", "function", "if", "main", "not", "or", "procedure", "read", "show", "showln", "then", "true", "var", "while". Se le suman a esto los tipos “integer” y “Boolean”.
* La palabra “end” no es considerada como reservada y permitirá ser utilizada como operador. Solo serán reservadas "end-if", "end-proc", "end-func" o "end-while".
* La palabra “begin” sólo podrá ser utilizada para indicar el inicio de un procedimiento o función.
* Los símbolos “” salvo dentro de una cadena o comentario no serán aceptados por el lenguaje tirando error. Ejemplo: “hola”

## 2.2 Tabla de Simbolos

Para la tabla de símbolos se armó una interfaz iTablaDeSimbolos la cual implementa las búsquedas como palabra reservada, como identificador o agregar un Token como identificador.

La clase TablaDeSimbolos implementa esta interfaz utilizando a su vez dos clases:

1. TablaID compuesta por un HashMap<String,Palabra> para guardar los ID’s encontrados.
2. TablaPalabrasReservadas compuesta por un List de tamaño fijo que contiene las palabras reservadas y otro List que contiene los tipos posibles.

Además, se implementó un patrón Singleton para devolver la única instancia posible de este objeto. Este patrón se implementó en la clase SingleTabla.

# **III: Conclusiones**

Debido a los cambios introducidos para realizar una separación aún más exhaustiva de los Tokens, el Analizador Léxico redujo su performance a la mitad (ya que se pasaron de 14 Tokens a 40).

# IV: Casos de Prueba

Los archivos de prueba se pueden encontrar la carpeta test junto con un jUnit que prueba el analizador léxico. Sino podra ejecutarse el .jar compilador y pasarle por línea de comando la dirección del archivo a analizar.

Los siguientes archivos fueron utilizados para realizar los casos de prueba:

## 4.1 Test

|  |
| --- |
| const M : integer = 7, R : integer = 90;  var N, S : integer;  var A[12] : integer, C[5] : boolean;  procedure PROC1();  var A : integer;  var B : boolean;  begin  B := not C[3] and (N < A or S <> R);  C[1] := true;  while B do  if A <= 10 then  A := A + 1;  end-if;  showLN 'Visualizacion', B, ' ', A;  B := C[3] and A + N < R;  end-while;  end-proc;  procedure PROC2(byref R : integer);  const T : integer = 67;  var W11, W12, Q : integer;  begin  C[2] := false;  S := (S + M) \* 2;  Q := 1;  read W11; {lectura de teclado}  W12 := AW11 \* 2 + S;  while (W12 - 2) <= R + S do  W12 := W12 \* 2;  end-while;  while W12 + M > Y / 2 do  Q := Q \* 2;  W12 := W12 / 2;  if W12 <= R then  R := R - W12;  Q := Q + 1;  end-if;  end-while;  A[1] := Q;  end-proc;  function FUN1(T: integer, byval N2 : integer) : boolean;  var N: integer;  begin  if T > 0 then  N := -45;  else  N := 70;  end-if;  end-func N \* 2 >= 0 or not N2 = 0;  procedure MAIN();  var X: boolean;  begin  PROC1();  S := M + 1;  PROC2(S);  X := FUN1(5, 8);  C[5] := X;  show A[S + 1];  end-proc; |

## 4.2 busquedaBinaria

|  |
| --- |
| const TAM : integer = 15;  var A[TAM] : integer;  procedure PRINCIPAL();  var busca, temp, bajo, alto, central, j, i : integer;  begin  {Ordenando arreglo...}    j:=1;  while (j <= TAM) do  i:=0;  while (i < TAM - 1) do  if ( a[i+1] <= a[i] )then  temp := a[i];  a[i] := a[i+1];  a[i+1] := temp;  end-if;  i := i + 1;  end-while;  j := j + 1;  end-while;    {Implementacion de busqueda binaria}    {Seteo el valor a buscar}  busca := 12;  bajo := 0;  alto := TAM -1;  central := (bajo+alto)/2;  while(and ( bajo < alto , busca != a[central] ) ) do  if ( a[central] <= busca ) then  bajo = central+1;  then  alto = central-1;  end-if;    central=(bajo+alto)/2;    end-while;  if (busca == a[central]) then  mostrarln busca,' encontrado en posicion ', central;  else  mostrarln 'No existe ', busca;  end-if;  end-proc; |

## 4.3 fibonacciTest

|  |
| --- |
| function fibonacci (n:integer):integer;  var fib:integer;  var nAux1, nAux2:integer;  var a, b, c: integer;  begin  if n < 3 then  fib := 1;  else  nAux1 := n-2;  nAux2 := n-1;  a := fibonacci(nAux1);  b := fibonacci(nAux2);  fib := a + b;    end-if;  end-func fib;    procedure PRINCIPAL();  var nro, i, rta:integer;  begin  nro:=0;  i:=1;  while nro < 1 do  show 'Ingrese un numero Entero: ';  read nro;  showln ' ';  end-while;  while i < nro+1 do  rta := fibonacci (i);  showln i, ':', rta;  i:=i+1;  end-while;  end-proc; |

## 4.4 ordenVector

|  |
| --- |
| const TAM : integer = 10;  var A[TAM] : integer;  procedure PRINCIPAL();  var i, j : integer;  begin  {Implementacion de Ordenamiento por burbuja de mayor a menor}    j:=1;  while (j <= TAM ) do  i:=0;  while (i < TAM - 1) do  if ( a[i] < a[i+1] ) then  temp := a[i];  a[i] := a[i+1];  a[i+1] := temp;  end-if;  i := i + 1;  end-while;  j := j + 1;  end-while;  end-proc; |

## 4.5 palabrasReservadas

|  |
| --- |
| const, var, byref  boolean, integer  procedure, function, beginbegin, begin, while, do,  if, then, read, showln, end-if, endit  M : integer = 7, R : integer = 90;  var N, S : boolean;  var A[12] : integer;  procedure PROC();  var A : integer;  var B : integer;  begin  B := M;  while B > 0 do  if par(B) then  A := A + 1;  end-if  showLN 'Visualizacion', B, ' ', A  B := B - 1  end-while  end-proc; |

# Indice

A

Autómata 4

D

Descipcion del Trabajo Practico 4

I

Implementación 6

Indice 14

R

Restricciones 7