**Universidad CAECE**

**Procesadores de Lenguaje**

Analizador Léxico

Versión 1.4

**Docente:**Amitrano, Sergio

**Integrantes:**Dinardi, Matías (75046/3)  
Maquieira , Guillermo (70519/9)  
Palmieri, Bruno (77086/4)  
Pentreath, Matías (82598/5)  
Zarco, Nicolás (76102/8)

14 de Septiembre de 2012

Contenido

[Contenido 2](#_Toc335335848)

[Historial de Revisiones 3](#_Toc335335849)

[Descipcion del Trabajo Practico 4](#_Toc335335850)

[Autómata 4](#_Toc335335851)

[Implementación 5](#_Toc335335852)

[Restricciones 6](#_Toc335335853)

[Indice 10](#_Toc335335854)

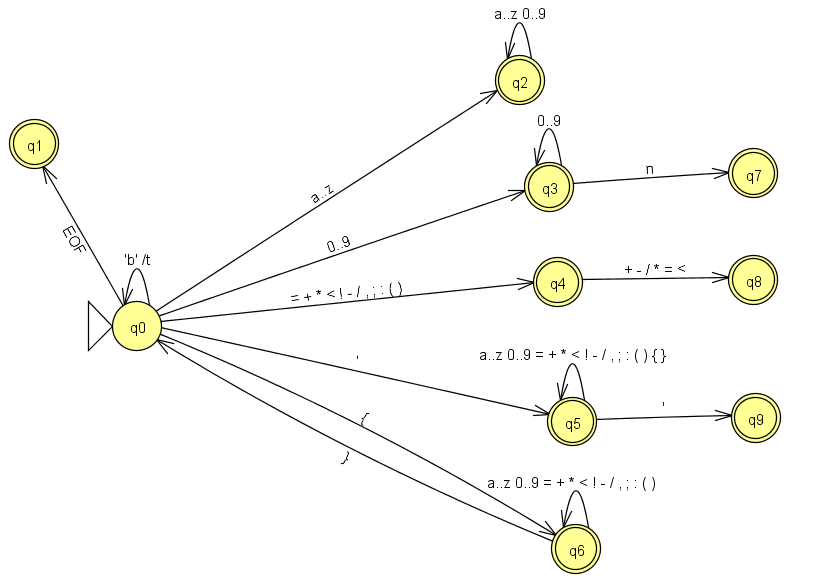
Historial de Revisiones

|  |  |  |  |
| --- | --- | --- | --- |
| **Fecha** | **Versión** | **Descripción** | **Autor** |
| **08/09/2012** | 1.0 | -Creación del documento.  -Autómata | Zarco Nicolás |
| **11/09/2012** | 1.1 | -Cambio diseño del documento  -Creación de Índice  -Creación de Tabla de Contenidos  -Implementación  -Casos de Prueba | Palmieri Bruno |
| **13/09/2012** | 1.2 | -Se agregaron los códigos de error de los *tokens* y descripción de los mismos | Palmieri Bruno |
| **13/09/2012** | 1.3 | -Creación de Restricciones | Maquieira Guillermo |
| **14/09/2012** | 1.4 | Update en descripción del uso del patrón Visitor | Matias Pentreath |

Descipcion del Trabajo Practico

Autómata

Se crea un autómata finito no 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. Los *Token* posibles son “EOF”, “Palabra”, “Entero”, “Natural”, “Operador”, “Cadena” y “Palabra reservada”.

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”.

Implementación

Para la implementación creamos una clase llamada **LexicAnalyzer** la inicializamos en su constructor con el path al archivo a analizar. Esta clase posee una función **getToken()** la cual retornara el siguiente *token* con el lexema encontrado en el archivo. Existen 7 tipos de *token* que pueden ser retornados:

* 5 para los diferentes tipos de lexemas (Entero, Natural, Operador, Palabra y Cadena)
* 1 para el EOF (*End of File* o Fin de Archivo).
* 1 de Error, el cual será enviado en caso de detectar el mismo dentro de un lexema.
* 1 de Palabra Reservada, en casa de que el lexema corresponda a una palabra reservada el lexema se enviara un *token* de dicho tipo.

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 (ya sea del tipo Entero, Natural, Cadena, Operador, Palabra, Palabra Reservada, EOF o Error) y se trata con una función de un objeto de tipo **LexicHelper.** Las funciones del helper retornarán el *token* con el **ID** correspondiente y sus coordenadas (fila y columnas).

En caso de que se detecte algún error a nivel léxico el **LexicHelper** retornara un *token* de error junto con un número de tipo de error. Estos errores son los siguientes:

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**)

Tanto la clase **LexicAnalyzer** como la clase **LexicHelper** hacen 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 implemento un Patron de Visitor. Se crearon las interfaces de Visatble y Visitor y se creo una clase **TokenVisitor** que implementa los métodos de la interfaz para poder navegar en la estructura de los *Tokens*, accediendo a su lexema e información particular para cada caso. En esta instancia del TP el visitor solo muestra en consola el lexema del Token y en el caso de Error también muestra su código de error.

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.

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 un número Natural, como por ejemplo 22n, se modelo como un Token de tipo Natural con su valor aritmético entero, simplificando así las futuras operaciones matemáticas.
* Para las cadenas se optó por tratar a un fin de línea como un corte del mismo.
* 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 "const", "var", "ref", “val”, "natural", "entero", "procedimiento", "funcion", "comenzar", "globales", "mientras", "hacer", "si", “sino”, "entonces", "leer", “adelantado”, "mostrar" , "mostrarln", “and” y “or”
  + La palabra “fin” es considerada como reservada y sólo permitirá ser concatenada con “-“ formando "fin-si", "fin-proc", "fin-func", "fin-mientras" o "fin-globales".
* La palabra “comenzar” sólo podrá ser utilizada para indicar el inicio de un procedimiento o función.  
  **Casos de Prueba**

Los siguientes archivos fueron utilizados para realizar los casos de prueba

|  |
| --- |
| globales  const M : natural = 7n, R : entero = 90;  var N, S : entero;  var A[12] : natural;  fin-globales;  procedimiento PROC1();  var A : entero;  var B : natural;  comenzar  B := M;  while B > 0 do  si par(B) entonces  A := A + 1;  fin-si  mostrarLN 'Visualizacion', B, ' ', A  B := B - 1  fin-mientras  fin-proc;  procedimiento PROC2(ref R : entero);  const T : natural = 67n;  var W11 : entero;  var W12, Q, R : natural;  procedimiento PROC2A();  comenzar  mostrarLN 'Interno A'  fin-proc;  const S : natural = 15n;  procedimiento PROC2B();  comenzar  mostrarLN 'Interno B'  fin-proc;  comenzar  S := (S ++ M) \*\* 2n;  Q := 1n;  leer W11; {lectura de teclado}  W12 := ANATURAL(AW11 \* 2 + S);  while (W12 – 2) <= R + S do  W12 := W12 \* 2;  fin-mientras;  while W12 + M > Y / 2 do  Q := Q \* 2;  W12 := W12 / 2;  si W12 <= R entonces  R := R – W12;  Q := Q + 1;  fin-si;  fin-mientras;  A[1] := Q;  fin-proc;  var T: natural;  funcion INC(N : entero) : entero; adelantado;  funcion FUN1(T: entero, val N2 : entero) : entero;  var N: entero;  comenzar  si T > 0 entonces  N := 45;  sino  N := INC(70);  fin-si;  fin-func N \* 2;  funcion INC(N : entero) : entero;  comenzar  fin-func N + 1;  var H: entero;  procedimiento PRINCIPAL();  var X: entero;  comenzar  PROC1();  S := M + 1;  PROC2(S);  X := FUN1(5, 8);  A[5n] := X;  mostrar A[ANATURAL(S) ++ 1n];  fin-proc; |

|  |
| --- |
| 22e  ;  22nnnn  87n31  ‘dadadada  89679+  {proc  finsis}  fin\_  ++  Fin-procedimiento{  dada  }  fin-proc  Siguiente  Vardump  Char int string  Const1  “$#%&/(/)(  Pepe865juan  Lal!”#$#%”hola  Referen  Mientroasn  Finmientras  }  lalal |

Indice

A

Autómata 4

C

Casos de Prueba 7

Contenido 2

D

Descipcion del Trabajo Practico 4

H

Historial de Revisiones 3

I

Implementación 5

Indice 10

R

Restricciones 6