GIS-GII

Práctica:
Procesadores de lenguajes
Pascal2HTML

PROCESADORES DE LENGUAJES

CARLOS RUIZ BALLESTEROS
HÉCTOR RUIZ-POVEDA COCA

ÍNDICE

AUTORES

Carlos Ruiz Ballesteros.

Héctor Ruiz-Poveda Coca.

Ambos autores somos del Doble Grado en Ingeniería del Software + Ingeniería Informática.

DESCRIPCIÓN DEL CÓDIGO

El código se divide en dos partes principales:

Analizador léxico

Por medio de la herramienta *"Flex"* se ha desarrollado un analizador léxico que permite identificar los distintos elementos: valores numéricos, identificadores, constantes literales, comentarios, retornos de carro...

El analizador léxico transforma estos elementos, así como palabras reservadas, en tokens y símbolos que utilizará el analizador sintáctico. Identificamos los lexemas de los distintos tokens con las siguientes expresiones regulares.

Para identificar números enteros, con la interrogación le decimos al lenguaje que puede o no tener esos dos elementos de dentro del rango:

Integer = [+-]?[0-9]+

Números hexadecimales. Que pueden empezar por "$":

HexaInteger = [$]?[A-F0-9]+

Numeros reales, que deben llevar un punto para separar la parte decimal y la parte entera:

Real = [+-]?[0-9]+.[0-9]+

Números reales en hexadecimal:

HexaReal = [$][+-]?[A-F0-9]+.[A-F0-9]+

//Regular expression to identify "identifier"

Ident = [A-Za-z_][A-Za-z_0-9]+|[A-Za-z]

Para identificar strings, cualquier palabra y espacios

Str = '[/w/s/W]+'

Y más expresiones regulares para detectar comentarios, tabuladores y que estos no afecten al analizador sintáctico.

Analizador sintáctico

La gramática que se encarga de comprobar si el código es sintácticamente correcto ha sido generada por "*javacup*".

Este analizador se encarga de comprobar que la estructura y el orden del programa son correctas.

Contiene variables y métodos para el control y detección de errores en la sintáxis del fichero a analizar. Un ejemplo son las variables booleanas ***if_expression*** y ***case_expression***, utilizadas para informar de un token incorrecto en la expresión aritmética de un *IF* o un *CASE*. Por otro lado cuenta con un método que informa la línea y la columna en la que se encuentra el error detectado (***syntax_error(Symbol s)***).

Se encuentran también en el analizador los símbolos terminales, no terminales y las reglas gramaticales necesarias para el correcto análisis gramatical.

Con motivo de una correcta recuperación de errores, se ha incluido la regla ***DEFRANG***, la cual, en caso de un error, busca un corchete cerrado y además en otras reglas si hay un error, busca el siguiente ";" o "end" para continuar con el análisis.

Cabe destacar que se han añadido las modificaciones pertinentes para añadir la gramática que permite el reconocimiento de matrices y registros, así como sentencias*"IF"*, "*WHILE"* y "*FOR"*.

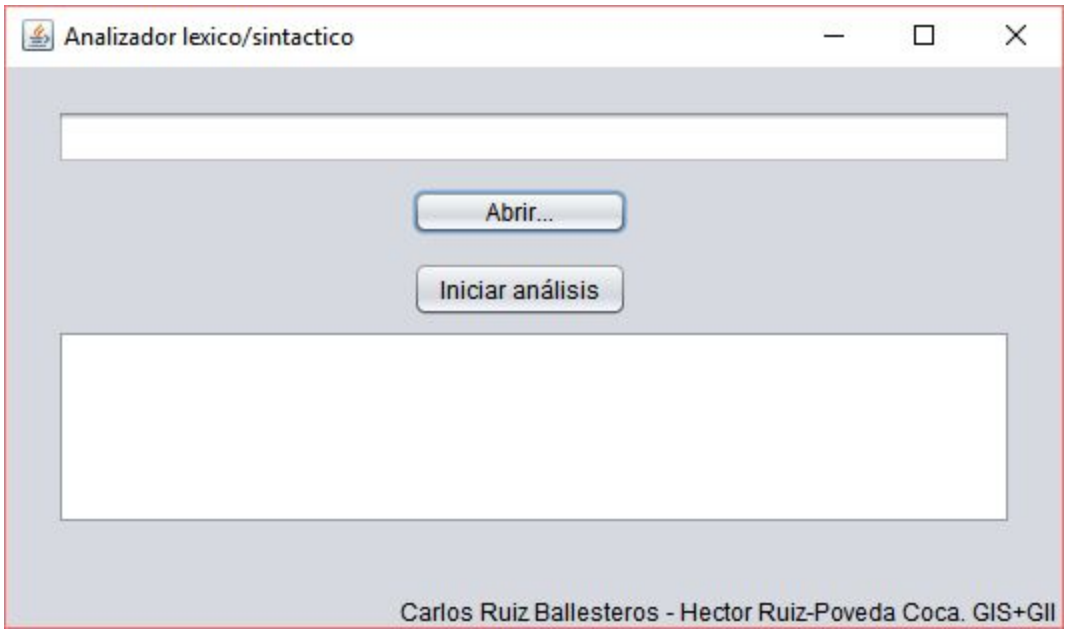
Interfaz gráfica y su uso

Con motivo de una mejor usabilidad se ha desarrollado una interfaz gráfica, la cual permite seleccionar un fichero, analizarlo, y proporciona un correcto feedback al usuario sobre el resultado de dicho análisis.

EJEMPLOS DE EJECUCIÓN

A continuación, se muestran las capturas de pantallas correspondientes a la ejecución del código:

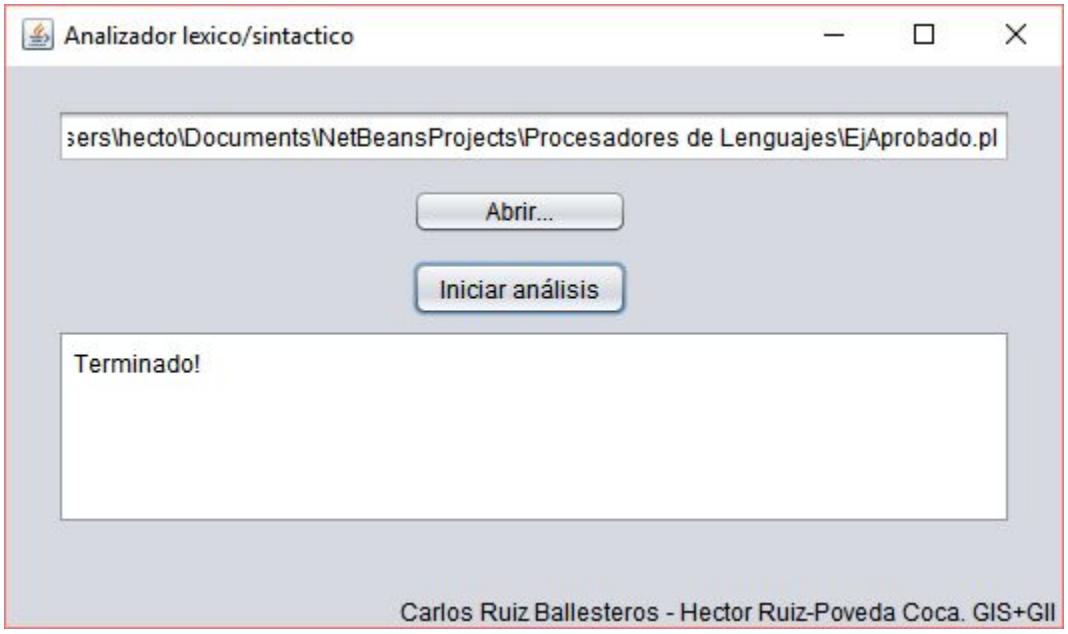
Página principal



Esta página permite la selección de un fichero, mediante un JFileChooser para mayor comodidad, el cual será analizado.

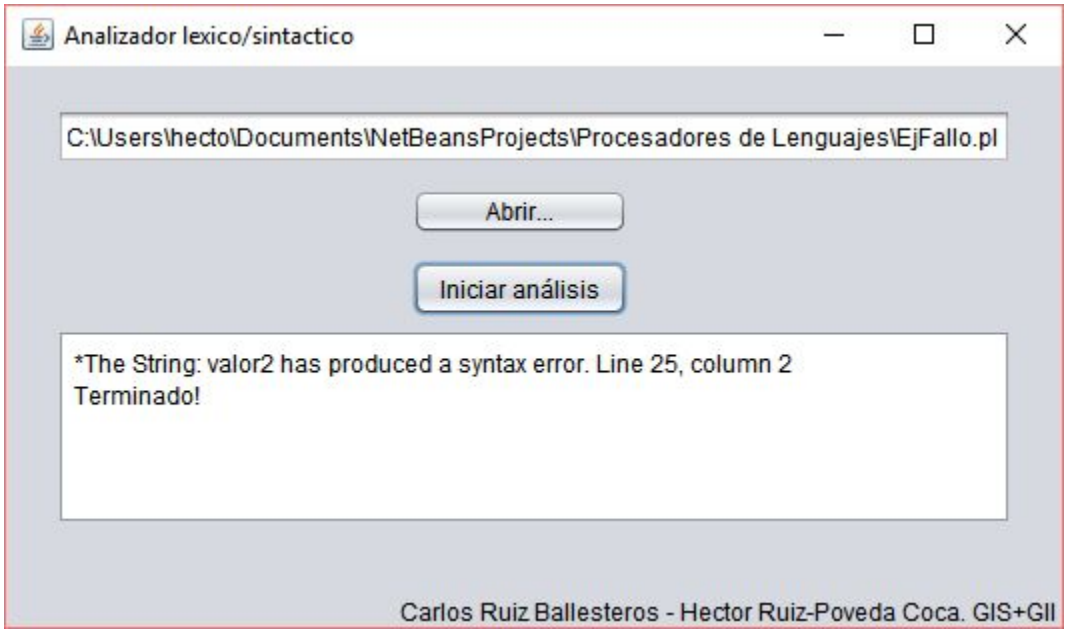
Una vez seleccionado el archivo, y tras pulsar el botón "Iniciar análisis", la retroalimentación de éste aparecerá en el cuadro de texto inferior.

Ejecución de un análisis



Aquí se muestra la ejecución del análisis sobre el fichero "EjAprobado.pl".

Ejecución de un análisis con error



Se puede apreciar cómo a pesar de detectar el error, el analizador termina su tarea analizando el archivo al completo.

ALGUNOS ERRORES EN LA GRAMÁTICA

Al ir desarrollando el ".cup" nos hemos ido encontrando con varios problemas. Uno de ellos es que a la regla CASELIST, le sobraba un ";". La regla ofrecida por el enunciado es:
CASELIST ::= EXP ":" SENT ";" | EXP ":" SENT ";" CASELIST

Bien pues el carácter ";" sobra de la regla ya que lo incluyen dentro el consecuente de SENT y no es necesario ponerlo. Por otro lado al añadir las reglas semánticas nos dio un problema de reducción, por lo que decidimos convertir la regla a lo siguiente:

CASELIST EXP two_points SENT | ;

De esta forma se podia añadir un solo "case", y el cup se pudo generar sin problemas.

ANÁLISIS SEMÁNTICO - GENERACIÓN EL HTML

Preanálisis y estructuras de datos utilizadas.

Antes de abordar la conversión decidimos dividir la generación del html en dos partes.

1º. Generamos una estructura de datos en memoria con toda la información.

2º. Recorremos toda esta información para generar el fichero html.

Como necesitamos saber los procedimientos y funciones resulta mejor tener toda la información organizada y luego generarla que tener que generar el fichero directamente.

Para ello decidimos utilizar una matriz de strings, donde cada fila se correspondia cada función/procedimiento, a excepción de las 2 primeras filas.

La primera fila contiene lo básico del html, como las cabeceras, los estilos…

La segunda columna contiene las cabeceras de las rutinas. Estas cabeceras son generadas luego por HTMLParser.java tras haber sido generada la información por el .cup.

<HTML …>	<HEAD …>	<STYLE …	…
<A href='#Prueba1'>Procedure Prueba1</A>	<A href='#Prueba2'>Procedure Prueba1</A>	<A href='#function1>Function function1</A>	…
<A name='Prueba1'> Procedure…</A>	(var1, var2: INTEGER)	begin	…

<A name='Prueba2'> Procedure...</A>	(var1 var2: INTEGER)	begin	...

De esta forma, al final lo que hacemos es recorrer dicha matriz, y generar el documento html.

Algunas variables importantes son estas:

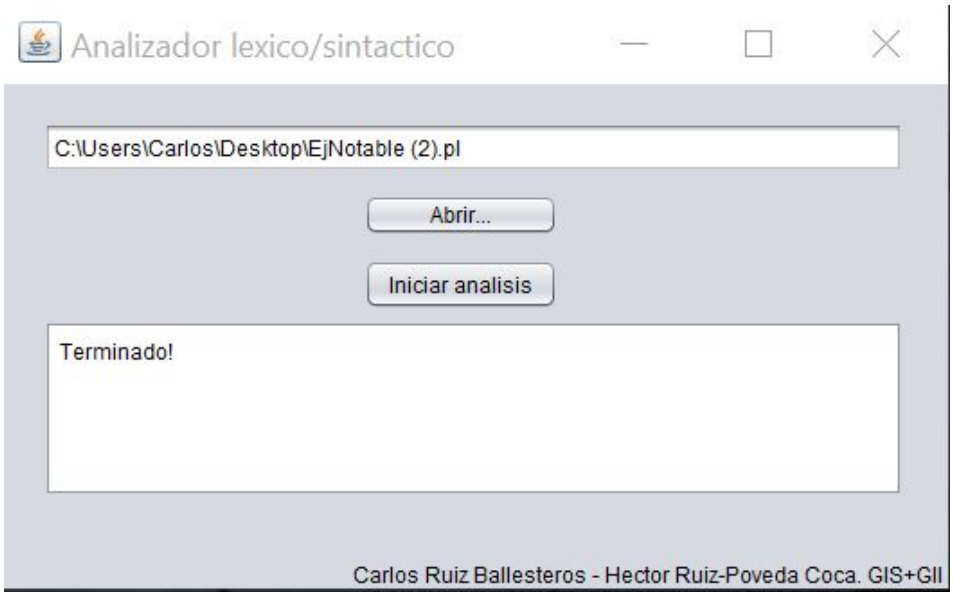
public int contador; Con el sabemos en cada momento el numero del procedimiento o función en el que estamos. Nos sirve para referenciar las variables mediante un href, sin que haya conflictos y ademas para agregar correctamente a la matriz todos las rutinas.

public double textIndent = 0;
public double indentFactor = 0.5; Los utilizamos para las estructuras de control y bucles anidados se vean correctamente con tabuladores.

EJEMPLO DE EJECUCIÓN CON ANALIZADOR SEMÁNTICO

Como sabemos con el analizador semántico vamos generando el html que finalmente podremos visualizar.

Veamos con ejemploNotable.pl



No hemos tenido ningún error, por lo que veamos el html.

Programa: EjemploNotable

Funciones y procedimientos

- function areaCuadrado (ladoREAL):REAL;
- procedure intercambio (v1, v2INTEGER);
- Programa principal

```
function areaCuadrado ( lado REAL ) : REAL ;
type miArray = array [ 0 .. 5 ] of INTEGER ;
var resultado : REAL ;
begin
   resultado := +0.0 ;
   miArray [ 1 ] := 0 ;
   resultado := lado * lado + miArray 1 + 2 ] ;
   areaCuadrado := resultado ;
end ;
Inicio de rutina
Inicio de programa
```

```
procedure intercambio ( v1, v2 INTEGER ) ;
var aux : INTEGER ;
begin
   if ( v1 <> v2 ) then
      begin
         aux := 0 ;
         aux := v1 ;
         v1 := v2 ;
         v2 := aux ;
      end
end ;
Inicio de rutina
Inicio de programa
```

Programa Principal

```
var medida : REAL ;
valor1, valor2 : INTEGER ;
begin
   medida := $4.A ;
   valor1 := -3 ;
   valor2 := $F6 ;
   while medida < valor1 do
      medida := areaCuadrado ( medida ) ;
   intercambio ( valor1 valor2 ) ;
end
Inicio de programa principal
Inicio de programa
```

Como vemos, se ha generado sin problemas, sin embargo si cogemos uno con errores, éste no se generará hasta que no estén todos corregidos.