View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by El Servicio de Difusión de la Creación Intelectual

JTLex un Generador de Analizadores Léxicos Traductores

Francisco Bavera, Darío Nordio, Marcelo Arroyo, Jorge Aguirre
Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto
{pancho,nordio,marroyo,jaguirre}@dc.exa.unrc.edu.ar♣

Resumen

En el presente trabajo se exponen los puntos principales del diseño e implementación de *JTLex*, un generador de analizadores léxicos. *JTLex*, al contrario de los generadores existentes, permite la especificación conjunta de la sintaxis y la semántica de los componentes léxicos, siguiendo el estilo de los esquemas de traducción. Para ello se basa en un nuevo formalismo, las Expresiones Regulares Traductoras, introducido por los autores. Tanto su diseño como la especificación de los procedimientos con que el usuario implementa la semántica asociada a los símbolos son Orientados a Objetos. El lenguaje de implementación de *JTLex* es *Java*, como así también, el del código que genera y el que usa el usuario para definir la semántica. *JTLex* se integra, como un generador de analizadores léxicos alternativo al tradicional, a *japlage*; un entorno de generación de procesadores de lenguajes – en particular de compiladores -, desarrollado en el grupo, que permite la evaluación concurrente de cualquier Gramática de Atributos Bien Formada. Los lenguajes de especificación brindados por *JTLex* y por el generador de analizadores sintácticos de *japlage* siguen el estilo de *Lex* y *Yacc* respectivamente – que son prácticamente un estándar -.

Palabras Clave: expresiones regulares, analizadores léxicos, expresiones regulares traductoras, autómatas traductores, análisis lexicográfico, compiladores de compiladores.

1 – Introducción

Un analizador léxico es un módulo destinado a leer caracteres del archivo de entrada, donde se encuentra la cadena a analizar, reconocer subcadenas que correspondan a símbolos del lenguaje y retornar los *tokens* correspondientes y sus atributos. Escribir analizadores léxicos eficientes a mano puede resultar una tarea tediosa, para evitarla se han creado herramientas de software – los generadores de analizadores léxicos – que generan automáticamente un analizador léxico a partir de una especificación provista por el usuario.

Puede asegurarse que la herramienta del tipo mencionado más conocida es *Lex* [Lev92]. *Lex* es un generador de analizadores léxicos, originalmente incluido dentro del ambiente de desarrollo de

♣ Este trabajo ha sido realizado en el marco de proyectos subsidiados por la SECyT de la UNRC y por la Agencia Córdoba Ciencia.

UNIX usando a *C* como lenguaje huésped y posteriormente migrado a casi todas las plataformas y lenguajes. Otra herramienta que últimamente ha tenido gran difusión es *Jlex* – que usa a *Java* como lenguaje huésped y corresponde al compilador de compiladores *Javacup*- [App98][Ber97]; mientras que algunos compiladores de compiladores actuales como *Javacc* [Javacc] y *Eli* [Com98] integran la especificación del análisis léxico sin brindar un módulo específico. Todas estas herramientas para generar analizadores léxicos permiten definir la sintaxis de los símbolos mediante expresiones regulares, mientras que sus atributos deben ser computados luego del reconocimiento de una subcadena que constituya un símbolo del lenguaje, analizándola. *JTLex* en cambio permite expresar conjuntamente sintaxis y semántica al estilo de los esquemas de traducción. A su vez el proceso de computo de atributos es implementado por *JTLex* por un autómata finito traductor con las ventajas de eficiencia que esto supone.

JTLex sigue el estilo de *Lex*, con la variante de que se basa en Expresiones Regulares Traductoras Lineales (ETL).

Una especificación *JTLex* permite no sólo asociar un procedimiento, o acción, a cada expresión regular, sino también a cada ocurrencia de un símbolo dentro de la expresión.

El siguiente ejemplo muestra como puede ser definido el símbolo *constante real* y su valor mediante *ETL´s*:

(dígito** {acum_pentera})+. (**dígito** {acum_pfracc})*

donde se supone que:

- Están definidas la variables ***parte_entera*, *parte_decimal:real*, *cant_decimales:entero*** y valen 0 al comienzo del reconocimiento de cada símbolo. al finalizar se ejecuta ***fin***
- ***dígito*** representa 0..9;
- ***cc*** es el carácter corriente;
- Las acciones se ejecutan después de haber reconocido cada carácter y responden al pseudo código que se muestra continuación:

  ***acum_pentera*:** *parte_entera = parte_entera *10 + val(cc)*
  acum_pfracc: *parte_decimal =parte_decimal *10+val(cc); cant_decimales ++;*
  fin: *valor := parte_entera + parte_decimal / (10 ^ cant_decimales)*

Esto sugiere una extensión de las expresiones regulares, en las que cada átomo es un par, integrado por un carácter de entrada y una acción. Además cada símbolo reconocido debe generar una sola secuencia de acciones.

Esta herramienta se integra al entorno de generación de procesadores de lenguajes *Japlage* [Agu98] [Agu01] – herramienta desarrollada por el grupo, según se esquematiza en la figura 1.

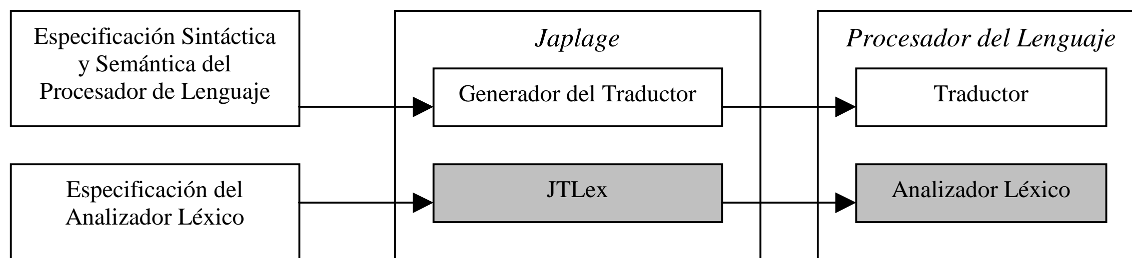

Figura 1: Incorporación de *JTLex* a *Japlage*.

2 – El Generador de Analizadores Léxicos *JTLex*

2.1 – Las Expresiones Regulares Traductoras Lineales (*ETL*) , formalismo que sustenta las especificaciones de *JTLex.*

Como se ha visto en la introducción, las expresiones de *JTLex* incluyen acciones apareadas a sus caracteres; cuando *JTLex* reconoce una subcadena va generando una de estas acciones por cada uno de sus caracteres, se puede por lo tanto pensar a la secuencia de acciones generada como una traducción de la subcadena reconocida. La traducción se produce teniendo como alfabeto de entrada al conjunto de caracteres y como alfabeto de salida al conjunto de acciones – que puede incluir a la acción nula, que se omitirá en la notación -. Por este motivo al tipo de formalismos que sustentan a *JTLex* se lo ha denominado Expresiones regulares Traductoras (*ET*)

Una expresión regular traductora (*ET*) es una expresión regular en la cual los términos están constituidos por pares carácter-acción. Dentro de las *ET* podemos caracterizar las expresiones regulares con traducción única (*ETU*), aquellas a las que a cada cadena de entrada corresponde una única cadena de salida.. Por último, se encuentran las *ETL* (Expresiones regulares Traductoras Lineales) que constituyen una subclase de las *ETU*. Las *ETL's* son definidas por razones de eficiencia. Las *ETL´s* se caracterizan porque en ellas deben ser únicas las acciones asociadas a ocurrencias de símbolos que finalicen la generación de un mismo prefijo. Para fijar ideas $0 \{A_1\}\ 1 \mid 0 \{A_2\}\ 3$ es una *ETU*, dado que las dos cadenas de entrada posibles (*01* y *03*) tienen asociada una sola secuencia de acciones – o traducción – (A_1 y A_2 respectivamente), pero no es una *ETL* o porque tanto A_1 como *A2* corresponden al último carácter de un mismo prefijo (*0*) - . Las *ETL* se corresponden con las *ETU* que pueden ser implementadas por Autómatas Finitos Traductores, cuyo tiempo de ejecución es lineal respecto de la cadena de entrada, razón por la cual han sido elegidas [Agui99].

Las *ETL* pueden ser definidas formalmente de la siguiente manera [Agu99]:

Sea *e: ET*,

$$e \text{ es una } ETL \Leftrightarrow \forall \alpha, \alpha' \in (\Sigma \times \Gamma)^* \ \ \forall a \in \Sigma \ \ \forall A, A' \in \Gamma$$
$$((\alpha (a,A) \in prefijos(e) \wedge \alpha' (a,A') \in prefijos(e) \ \ y \ \Pi_1(\alpha) = \Pi_1(\alpha')) \Rightarrow A=A')$$

Donde:

$$\Pi_1 : Expresiones\ Regulares\ sobre\ (\Sigma \times \Gamma) \rightarrow \Sigma^* \ \ tal\ que:$$
$$\Pi_1 (\varnothing) = \varnothing,$$

$$\Pi_1 (\lambda) = \lambda ,$$
$$\Pi_1((a, A)) = a,$$
$$\Pi_1 (e \mid e') = \Pi_1 (e) \mid \Pi_1 (e'),$$
$$\Pi_1 (e . e') = \Pi_1 (e) . \Pi_1 (e'),$$
$$\Pi_1(e^*) = (\Pi_1 (e))^*$$
$$\Pi_1 : (\Sigma \times \Gamma)^* \rightarrow \Sigma^* \text{ tal que } \Pi_1 ((a, A) \gamma) = a\, \Pi_1 (\gamma).$$

2.2 – Esquema de Funcionamiento de *JTLex*

JTLex permite generar un analizador léxico traductor a partir de la especificación mediante *ETL´s* de los símbolos del lenguaje al que está destinado y sus correspondientes semánticas, según se muestra en la figura 2.

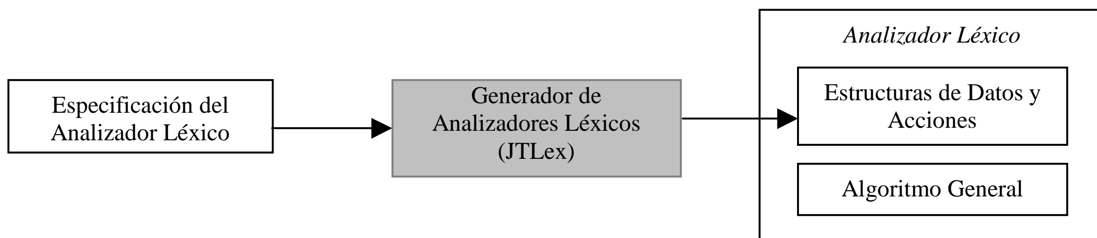

Figura 2: Entrada y salida del Generador de Analizadores Léxicos

2.3 –El Lenguaje de Especificación brindado por *JTLex*

Debido a que *Lex* es, quizás, la herramienta más conocida entre los generadores de analizadores léxicos, las especificaciones *JTLex*, siguen los lineamientos de su lenguaje de especificación con las modificaciones necesarias para implementar el nuevo formalismo. Para especificar los componentes léxicos y su semántica en vez de usarse constructores de la forma patrón-acción, se utilizan expresiones regulares cuyos términos corresponden a pares carácter-acción. *Java* es el lenguaje huésped, en él se especifican las acciones.

En esencia, la especificación *JTLex* consiste de una serie de reglas para dividir la cadena de entrada en tokens. Estas reglas son expresiones regulares traductoras lineales [Agu99].

La idea aquí desarrollada consiste en usar para el análisis léxico una sintaxis muy similar a la de los esquemas de traducción usados para los lenguajes independientes del contexto, que se usan extensamente en los generadores de analizadores sintácticos.

El lenguaje de especificación – cuya gramática se muestra en la figura 2 – se divide en tres partes:

Declaraciones
%%
Reglas de las Expresiones Regulares Traductoras Lineales
%%
Código de Usuario

La directiva %% divide las distintas secciones del archivo de entrada y debe estar ubicada al comienzo de línea.

La sección *Declaraciones* – la primer sección de la especificación – permite la incorporación de declaraciones que se utilizan en el código del analizador léxico, también permite la definición de macros los cuales están destinados a simplificar la escritura de las expresiones regulares.

Las *reglas de las expresiones regulares* consisten de tres partes : una acción inicial opcional, una Expresión Regular Traductora Lineal y una acción final. Donde las acciones iniciales y finales son código Java. Y las expresiones regulares traductoras lineales asocian acciones a caracteres. Básicamente, en una expresión regular traductora lineal debe existir una única traducción para todos los prefijos posibles. En cada expresión, si existen prefijos iguales las acciones deben ser las mismas, pero para expresiones que definen distintos tokens pueden llegar a ser distintas.

Por último, la sección *Código de Usuario* es copiada directamente en el archivo de salida resultante. Esta provee espacio para definir e importar clases que intervengan en la traducción.

```
<gram_lex> ::=  <decl_java>  <decl_macro> “%%” <body_lex> “%%” code_java
<decl_java> ::=  ( “%{“  code_java  “%}” )? (  “%inic{“  code_java  “%inic}” )?
                 ( “%eof{“ code_java “%eof}” )? (“%error{“code_java “%error}” )?
<decl_macro> ::=  NAME_MACRO  <exp_reg_macro> <decl_macro1>
                  | NAME_MACRO  <exp_reg_macro>   | λ
<body_lex> ::=  <inic_java>  <exp_reg>  “{“  code_java  “};”  <body_lex>
                | <inic_java>  <exp_reg>  “{“  code_java  “};”
<inic_java> ::=  ^ “{“ code_java  “}”   | λ
<exp_reg_macro> ::=  <exp_reg_macro>    <exp_reg_macro>
                     | <exp_reg_macro>  “|”  <exp_reg_macro>
                     |  “(“   <exp_reg_macro> “)”   <symbol>
                     | “\”  CHAR   <symbol>
                     | CHAR   <symbol>
                     | “”” words  “””  <symbol>
                     | “[“  rango   “]”   <symbol>
                     | “{“  NAME_MACRO   “}”  <symbol>
                     | “.”   <symbol>
<exp_reg> ::=  <exp_reg>    <exp_reg>
               | <exp_reg>  “|”  <exp_reg>
               |  “(“   <exp_reg> “)”   <symbol>
               | “\”  CHAR <j_action>   <symbol>
               | CHAR   <j_action> <symbol>
               | “”” words  “””  <symbol>
               | “[“  rango   “]”  <j_action> <symbol>
               | “{“  NAME_MACRO   “}”  <symbol>
               | “.”   <j_action> <symbol>
<words> ::=  CHAR <words>    | CHAR
<symbol> ::=  “+”   | “*”    | “?”   | λ
<j_action> ::=  “{“ code_java  “}”    | λ
<rango> ::=  <rango><rango> |  <rango> “-“<rango> | <rango> “,”<rango> | CHAR | “\” CHAR | “.”
Donde, CHAR ::= conjunto de caracteres ASCII.,
       code_java = Σ*
       NAME_MACRO ::= {Letra} ({Letra}|{digito})*
```

Figura 3: Gramática del lenguaje de especificación

2.4 – Lenguaje de Implementación

El lenguaje de especificación elegido para implementar el generador; como lenguaje huésped para las traducciones y para implementar los analizadores léxicos generados es *Java* [Gos96] [Gos97]. Esta elección cumple con el requisito de que la implementación sea multi-plataforma y permita utilizar la tecnología orientada a objetos.

3 – Diseño del Generador de Analizadores Léxicos Traductores

Para llevar a cabo la implementación del Generador de Analizadores Léxicos JTLex, se siguió el algoritmo presentado en [Agu99] el cual es una extensión del presentado por Aho et al [Aho88] para obtener un Autómata Finito Determinístico equivalente a una Expresión Regular, sin pasar por la construcción de uno no determinístico – como en el algoritmo de Thomson [Tho68] -.

Para cada especificación el generador construirán las estructuras de datos necesarias para que un algoritmo genérico se instancie, constituyendo el analizador específico – correspondiente a la especificación – a continuación se describe el proceso de construcción de dichas estructuras y luego el diseño del algoritmo genérico.

La construcción de las estructuras de datos sigue los siguientes pasos:

1- Construcción de Autómata Finito Traductor (AFT) para cada Expresión Regular Traductora Lineal (ETL) de la especificación
2- Construcción de un AFNλ que acepte la unión de los lenguajes correspondientes a los autómatas obtenidos en el punto anterior.
3- Construcción de un AFD a partir del AFNλ.
4- Implementación del analizador léxico a partir de las estructuras obtenidas en los pasos anteriores.

A continuación se realiza un breve comentario de cada uno de los pasos:

El primer paso se basa en el conocido algoritmo para obtener un autómata finito determinístico equivalente a una expresión regular, sin pasar previamente por la construcción de otro no determinístico citado previamente. Los estados del autómata por él construido son conjuntos de posiciones del árbol sintáctico asociado a la expresión regular. El algoritmo decide que la entrada no es una ETL cuando para una transición de estados detecta más de una traducción. Los pasos que sigue son los siguientes:

1. Para cada ET *e* de la especificación:
   a. Se construye el árbol sintáctico τ de *e#* .
   b. Se computan las funciones nullable, firstpos, lastpos y followpos sobre τ [Aho88].
   c. Se decide si *e* es una ETL y en tal caso se construye el correspondiente AFT utilizando el algoritmo desarrollado a tal efecto [Agu99].

2- Se construye un AFNλ a partir de los autómatas construidos en el paso anterior. Para la unión de los lenguajes se lo transforma en un autómata determinístico y a partir de él se definen las estructuras de datos sobre las que se basará el analizador léxico.

La figura 4 presenta el diseño del generador de analizadores léxicos traductores.

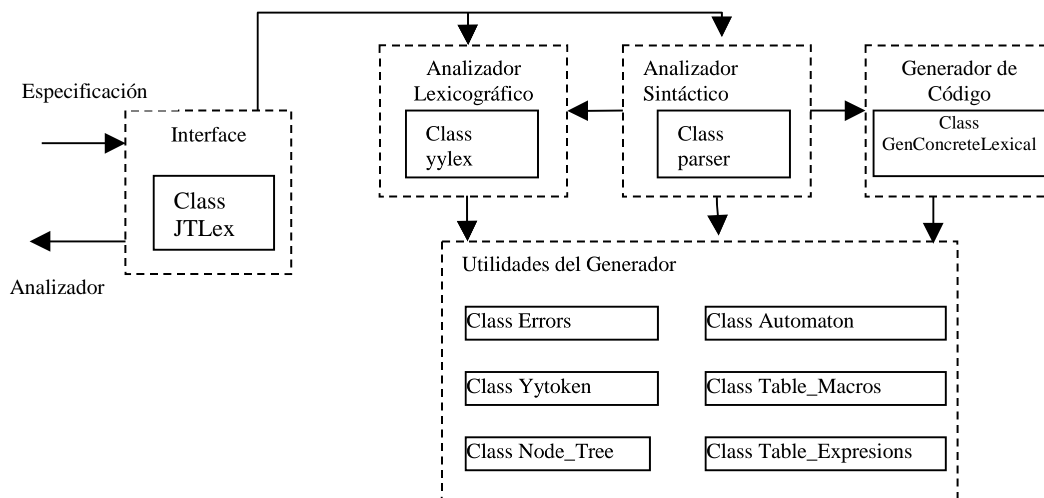

Figura 4: Diseño del Generador de Analizadores Léxicos *JTLex*.

Finalmente se genera el analizador léxico con un esqueleto general que se especializa por las estructuras de datos obtenidas en el paso anterior.

El analizador léxico deberá encontrar la primera subcadena maximal que corresponda a alguna de las expresiones de la especificación, si esta subcadena corresponde a más de una expresión se deberá elegir la que figure primera en la lista. Esto se logra corriendo el AFD correspondiente a la unión hasta que se bloque y seleccionando la expresión correspondiente a partir de las estructuras de datos. Una vez determinada la expresión que corresponde al lexema reconocido el analizador retorna el *token* asociado y corre el correspondiente AFT.

La figura 5 esquematiza el diseño de los analizadores léxicos generados por JTLex.

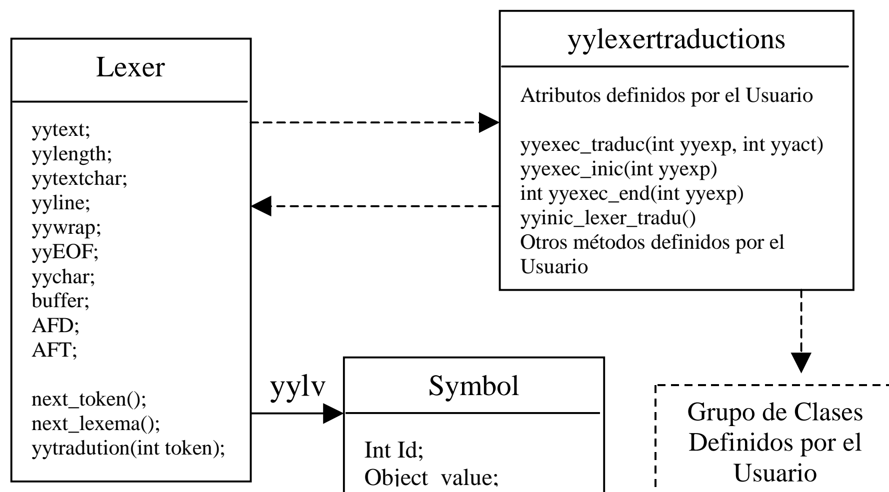

Figura 5: Diseño del Analizador Léxico.

4 – Utilización del Generador

Para generar el analizador léxico definido por una especificación *JTLex*, el usuario deberá dar el siguiente comando.

Java JTLex <nombre archivo.lex> en donde <nombre archivo.lex> es el archivo que contiene la especificación *JTLex*.

Este comando genera un archivo *Concrete_Lexical.java* que contiene el módulo analizador léxico generado, que podrá ser integrado al programa que lo utiliza

4.1 – Ejemplo de una especificación *JTLex*

El siguiente ejemplo muestra una especificación *JTLex* que reconoce cadenas que representan números decimales y ternarios; imprimiendo su valor. Los números ternarios están precedidos por 0t.

```
/*Declaración de variables globales */
%{
     int valor; /* valor del número reconocido en base 10 */
     int base; /* indica la base */

     public void inic() {valor=0;base=10;}

     public void inic3() { base=3;}

     public void acumular()
     { //acumula un dígito en la base dada
          Integer aux = new Integer(yytextchar());
          valor *= base + aux.intValue();
     }
%}
```

```
/* Tratamiento de Error */
%error{
      System.out.print(" Error, Lexema no reconocido: "+yytext());
      System.out.println(" Linea: "+ yyline());
%error}

%%  /* Declaración de la expresión regular traductora.
       Las expresiones se extienden hasta el carácter ";" */

{ inic();}

     ( [0-9] { acumular(); } ) +
    |
       0t    { inic3();}
     ( [0-3] { acumular(); } ) +

      {System.out.println("Valor: "+ valor);break;}
      ;


/* declaración de usuario */
%%

import java.io.*;
import java.lang.System;

class Main
{
  public static void main (String argv[])
  {
    Lexer L;
    try { L = new Lexer(argv[0]);
    }catch(Exception e){}
    int i = 0;
    while (i!=-1)
    {
     i = L.next_token();
    }

  } // Fin del método main
}// Fin de la clase Main
```

5 – Conclusiones y Trabajos Futuros

Se ha obtenido una primer versión de JTLex. El lenguaje de JTLex integra el formalismo usado al paradigma OO y resulta totalmente familiar para los usuarios de Lex. El lenguaje de especificación definido, basado en las ETL, sigue el mismo estilo que los lenguajes usados para la especificación de esquemas de traducción en las herramientas más usadas como yacc [Joh75] y CUP [Hud99].

La implementación sobre Java garantiza la característica multi-plataforma del generador.

Los algoritmos implementados permitieron la construcción de un generador de analizadores léxicos que produzca reconocedores-traductores que reconocen una cadena α y la traducen en acciones en tiempo $O(|\alpha|)$.

El tiempo de generación es del mismo orden que el requerido por los algoritmos tradicionalmente usados.

Las ventajas reales que represente esta modificación para los usuarios sólo se podrán conocer después de que se haya practicado su uso.

Sería interesante encontrar algoritmos eficientes para traducir cualquier ETU, tema que piensan abordar los autores en el futuro inmediato. Como así también, extender el lenguaje de especificación para permitir asociar traducciones a subexpresiones, en lugar de a caracteres.

Referencias bibliográficas

[Agu98] J. Aguirre, V. Grinspan, M.. Arroyo, "Diseño e Implementación de un Entorno de Ejecución, Concurrente y Orientado a Objetos, generado por un Compilador de Compiladores", Anales ASOO 98 JAIIO, pp. 187-195, Buenos Aires 1998.

[Agu99] J. Aguirre, G. Maidana, M. Arroyo, "Incorcorando Traducción a las Expresiones Regulares", Anales del WAIT 99 JAIIO, pp 139-154, 1999.

[Agu01] J. Aguirre, V. Grinspan, M. Arroyo, J. Felippa, G. Gomez, "JACC un entorno de generación de procesadores de lenguajes" Anales del CACIQ 01, 2001.

[Aho88] A.V. Aho, R. Sethi, J.D. Ullman, "Compilers: Principles, Thechniques, and Tools", Addison Wesley 1988.

[App98] A. W. Appel, "Modern Compiler in Java", Cambridge University Press, ISBN: 0-521-58388-8.1998.

[Ber97] E. Berk, "JLex: A lexical analyzer generator for Java", Department of Computer Science, Princeton University. 1997.

[Com98] Compilers Tools Group, "ELI", Department of Electrical and Computer Engineering, Colorado University, C.O., USA. 1998.

[Gos96] J. Gosling, B. Joy, Steele, "The Java language specification", Addison Wesley. 1996.

[Gos97] J. Gosling, K. Arnold, "El Lenguaje de Programación Java", Addison Wesley. 1997.

[Hud99] S. Hudson, "CUP User's Manual", Graphics Visualization and Usability Center Georgia Institute of Technology. 1999.

[Javacc] http://falconet.inria.fr/~java/tools/JavaCC_0.6/doc/DOC/index.html

[Joh75] Johnson, S.C., “Yacc – yet another compiler compilers”, Computing Science Technical Report. AT&T. 1975.

[Lev92] J. Levine, T. Mason, D. Brown, “Lex & Yacc”, O’Reilly & Associates, Inc. 1992.

[Tho68] Thompson K., “Regular Expression Search Algorithm, Communications ACM 11:6. 1968.