Universidad Nacional de Río Cuarto
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Departamento de Computación

Un Generador de Analizadores Léxicos Traductores

Director: Prof. Jorge Aguirre
Codirector: Lic. Marcelo Arroyo

Integrantes

Francisco Pedro Bavera

Darío Martín Nordio

Indice

Introducción

Esta tesina constituye el último escalón de los autores para finalizar sus estudios de la carrera Licenciatura en Ciencias de la Computación y obtener su título de grado.

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 y complicada, 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 *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 C*u*p – [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. Una alternativa sería contar con una herramienta que permita computar los atributos a medida que se reconocen dichas subcadenas aprovechando el mecanismo de computo garantizado por el analizador léxico. De esta manera se libra al usuario del control sobre la cadena a computar.

En el presente trabajo, motivado por lo expuesto en el párrafo anterior, se exponen los puntos principales del diseño e implementación de un generador de analizadores léxicos que, 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 Jorge Aguirre et al – Incorporando Traducción a las Expresiones Regulares [Agu99] –.

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 del *generador* es *Jav*a, como así también, el del código que genera y el que usa el usuario para definir la semántica.

Esta herramienta se integra, como un generador de analizadores léxicos alternativo al tradicional, a *japlag*e; un entorno de generación de procesadores de lenguajes – en particular de compiladores –, desarrollado en el grupo de investigación de Procesadores de

Lenguajes[♣], que permite la evaluación concurrente de cualquier Gramática de Atributos Bien Formada. Los lenguajes de especificación brindados por el generador de analizadores léxicos traductores y por el generador de analizadores sintácticos de *japlage* siguen el estilo de *Lex* y *Yacc* respectivamente – que son prácticamente un estándar –.

Estructura de esta Tesis

Esta tesis se divide en dos partes – no es una división física sino lógica – bien diferenciadas pero complementarias. La primera es netamente teórica y se refiere a los generadores de analizadores léxicos mientras que la segunda parte expone los puntos más relevantes de la implementación de un generador de analizadores léxicos traductores – basado en expresiones regulares traductoras lineales –.

La primer parte consta de dos capítulos. El primero, *Introducción a los Analizadores Léxicos*, como su título lo expresa, presenta los conceptos y nociones teóricas referentes a los generadores de analizadores léxicos basados en expresiones regulares. El segundo, *Introducción a las Expresiones Regulares Traductoras,* exhibe el nuevo formalismo en el que se basa la herramienta generada.

Los siguientes cinco capítulos componen la segunda parte. El capítulo 3 presenta el diseño de los analizadores léxicos a generar y el diseño del generador de los mismos. En el capítulo 4 se exhibe el lenguaje de especificación de los analizadores léxicos a generar. El capítulo 5 trata sobre el análisis lexicográfico y sintáctico. La generación de código se presenta en el capítulo 6. Por último, el capítulo 7 esquematiza como se incorpora el generador de analizadores léxicos traductores a la herramienta *japlage.*

[♣] Grupo perteneciente al Departamento de Computación de la Facultad de Ciencias Exactas, Físico-Químicas y Naturales de la Universidad Nacional de Río Cuarto.

Capítulo 1

Introducción a los Analizadores Léxicos

Este capítulo trata sobre los conceptos básicos para especificar e implementar analizadores léxicos. Un analizador léxico lee caracteres del archivo de entrada, donde se encuentra la cadena a analizar, reconoce lexemas y retorna tokens. Una forma sencilla de crear un analizador léxico consiste en la construcción de un diagrama que represente la estructura de los componentes léxicos del lenguaje fuente, y después hacer "a mano" la traducción del diagrama a un programa para encontrar los componentes léxicos. De esta forma, se pueden producir analizadores léxicos eficientes.

Las técnicas utilizadas para construir analizadores léxicos también se pueden aplicar a otras áreas, como por ejemplo, a lenguajes de consulta y sistemas de recuperación de información. En cada aplicación, el problema de fondo es la especificación y diseño de programas que ejecuten las acciones activadas por patrones dentro de las cadenas.

Existe una gran variedad de generadores de analizadores léxicos. Quizás la herramienta más conocida es *Lex,* un generador de analizadores léxicos para el sistema operativo UNIX basada en expresiones regulares que genera código *C*. Estas herramientas generan automáticamente analizadores léxicos, por lo general a partir de una especificación basada en expresiones regulares. La organización básica del analizador léxico resultante es en realidad un autómata finito. Es por esto, que en este capítulo se introduce el concepto de expresiones regulares como así también otros conceptos básicos como los Autómatas Finitos. Por último, se encuentran los algoritmos que permiten construir autómatas finitos determinístico (AFD) a partir de expresiones regulares (ER) entre otros.

1.1 - Función del Analizador Léxico

El analizador léxico forma parte de la primera fase de un compilador. Un compilador es un programa que lee un programa escrito en un lenguaje, el lenguaje fuente, y lo traduce a un programa equivalente en otro lenguaje, el lenguaje objeto. El proceso para construir un compilador se encuentra dividido en cuatro etapas:

- El *análisis léxico:* transforma el código fuente en tokens.
- El *análisis sintáctico:* construye un árbol sintáctico
- El *análisis semántico:* realiza el chequeo de tipos
- La *generación de código:* genera código de maquina.

Podemos representar estas sucesivas etapas con el siguiente diagrama:

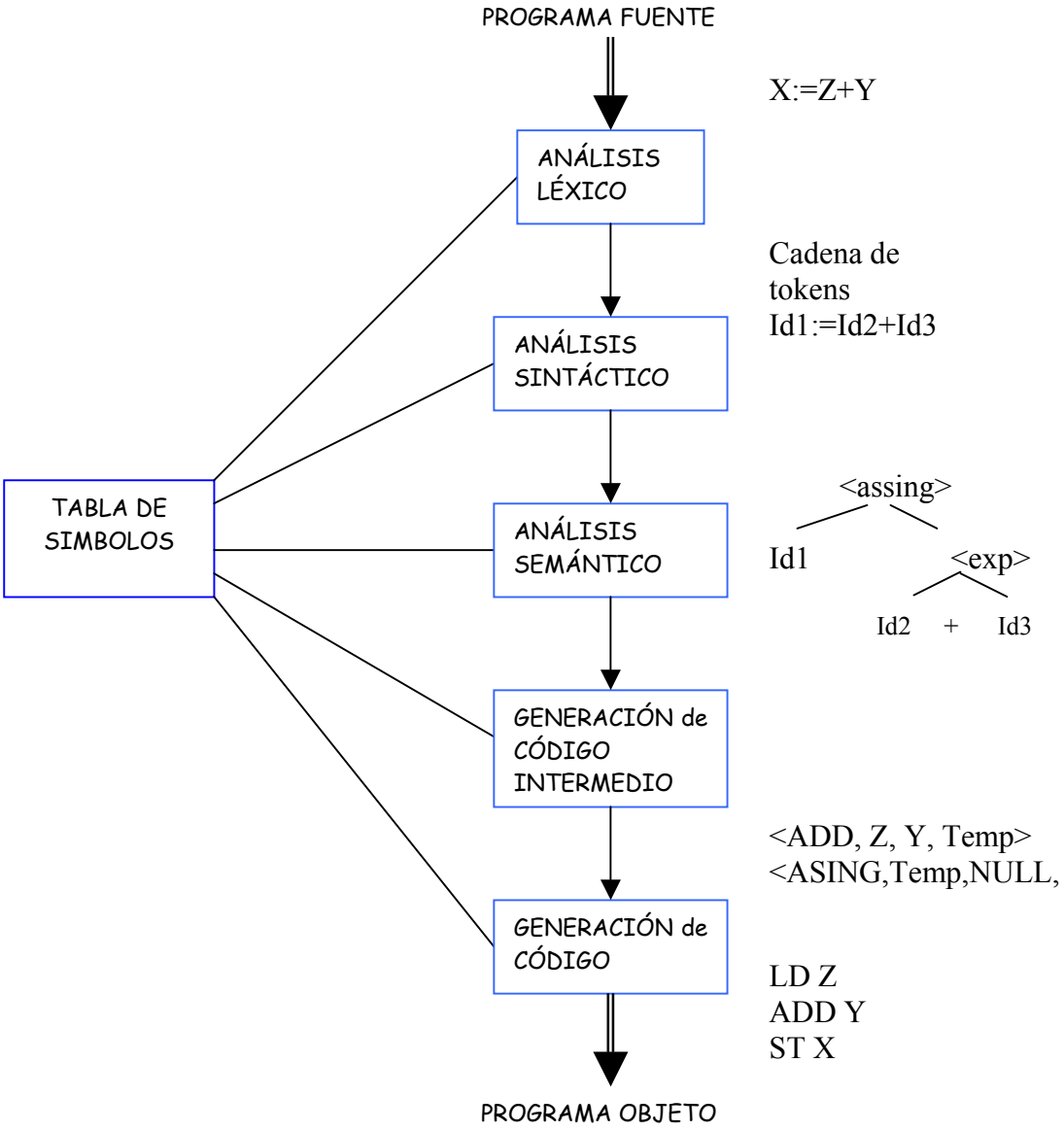


Figura 1.1: Etapas de un compilador.

La función principal de los analizadores léxicos consiste en leer la secuencia de caracteres de entrada y dar como resultado una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis. Esta interacción, esquematizada en la figura 1.2, suele aplicarse convirtiendo al analizador léxico en una subrutina del analizador sintáctico. Recibida la orden "obtener el siguiente componente léxico" del analizador sintáctico, el analizador léxico lee los caracteres de entrada hasta que pueda identificar el siguiente componente léxico.

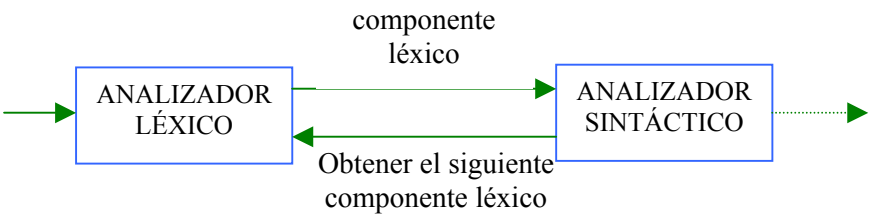


Figura 1.2: Interacción entre el analizador léxico y el analizador sintáctico.

El analizador léxico puede realizar tareas secundarias en la interfaz del usuario, como eliminar espacios en blanco, tabulaciones y caracteres de fin de línea.

En algunas ocasiones, los analizadores léxicos se dividen en una cascada de dos fases: la primera llamada "examen" y la segunda "análisis léxico". El examinador se encarga de realizar tareas sencillas, mientras que el analizador léxico es el que realiza las operaciones más complejas. Por ejemplo, un compilador de FORTRAN puede utilizar un examinador para eliminar los espacios en blanco de la entrada.

1.2 - Componentes léxicos o tokens, patrones y lexemas

Cuando se mencionan los términos "componentes léxicos" – token –, "patrón" y "lexema" se emplean con significados específicos. En general, hay un conjunto de cadenas en la entrada para el cual se produce como salida el mismo componente léxico. Este conjunto de cadenas se describe mediante una regla llamada patrón asociado al componente léxico. Se dice que el patrón *concuerda* con cada cadena del conjunto. Un lexema es una secuencia de caracteres en el programa fuente con la que concuerda el patrón para un componente léxico [Aho88]. Por ejemplo, en la preposición de Pascal

const p:i=3.1416;

la subcadena pi es un lexema para el componente léxico "identificador". En la figura 1.3 aparecen ejemplos de los usos de componentes léxicos, patrones y lexemas.

Componente Léxico	***Lexemas de Ejemplo***	***Descripción Informal del Patrón***
while	while	while
if	if	if
relación	<,<=,=,<>,>,>=	< ó <= ó = ó <> ó > ó >=
identificador	nom, dirección, c4	letra seguida de letras y dígitos
natural	3, 4, 67, 98	cualquier constante natural
real	3.45, 12.78, 37.5	cualquier constante real
literal	"ejecución de una acción"	cualquier caracter entre " ... ",excepto "

Figura 1.3: Componentes léxicos, lexemas y patrones.

Los componentes léxicos se tratan como símbolos terminales de la gramática del lenguaje fuente – en la figura 1.3 se representan con nombres en **negritas** –. Los lexemas para el componente léxico que concuerdan con el patrón representan cadenas de caracteres en el programa fuente que se pueden tratar como una unidad léxica.

En la mayoría de los lenguajes de programación, se consideran componentes léxicos a las siguientes construcciones: palabras clave, operadores, identificadores, constantes, cadenas literales y signos de puntuación como por ejemplo paréntesis, coma, punto y coma. En el ejemplo anterior, cuando la secuencia de caracteres *nom* aparece en el programa fuente, se devuelve al analizador sintáctico un componente léxico que representa un identificador. La devolución de un componente léxico a menudo se realiza mediante el paso de un número entero correspondiente al componente léxico. Este entero es al que hace referencia el *nom* en **negritas** de la figura 1.3.

Un patrón es una regla que describe el conjunto de lexemas que pueden representar a un determinado componente léxico en los programas fuentes. El patrón para el componente léxico **while** de la figura 1.3 es simplemente la cadena sencilla **while** que deletrea la palabra clave. El patrón para el componente léxico **relación** es el conjunto de los seis operadores relacionales de Pascal. Para describir con precisión los patrones para componentes léxicos más complejos, como **identificador** y **natural** – para los números naturales –, se utilizará la notación de expresiones regulares desarrollada en la siguiente sección.

1.3 - Especificación de los componentes léxicos

Las expresiones regulares son una notación sencilla y poderosa para especificar patrones. Cada patrón concuerda con una serie de cadenas, de modo que las expresiones regulares servirán como nombres para conjuntos de cadenas.

1.3.1 - Cadenas y lenguajes

El término *alfabeto* o *clase de caracter* denota cualquier conjunto finito de símbolos. Ejemplos típicos de símbolos son las letras y los caracteres. El conjunto {0,1} es el *alfabeto* binario.

Una *cadena* sobre algún alfabeto es una secuencia finita de símbolos tomados de ese alfabeto. En teoría del lenguaje, los términos *frase* y *palabra* a menudo se utilizan como sinónimos del término *cadena*. La longitud de una cadena s, denotada por |s|, es el número de apariciones de símbolos en s. Por ejemplo, *computadora* es una cadena de longitud once. La cadena *vacía*, representada por λ, es una cadena especial de longitud cero.

Si x e y son cadenas, entonces la *concatenación* de x e y que se denota xy, es la cadena que resulta de agregar y a x. Por ejemplo, si x=*caza* e y=*fortunas*, entonces xy=*cazafortunas*. La cadena vacía es el elemento identidad que se concatena. Es decir, $s\lambda = \lambda s = s$.

Cuando se considera la concatenación como un "producto", cabe definir la "exponenciación" de cadenas de la siguiente manera: se define s^0 como λ, y para i>0 se define s^i como $s^{i-1}s$. Dado que λs es s, $s^1=s$. Entonces $s^2=ss$, $s^3=sss$, etc.

A continuación se presentan algunos términos comunes asociados con las partes de una cadena:

Prefijo de s: Una cadena que se obtiene eliminando cero o más símbolos desde la derecha de la cadena **s**; por ejemplo *compu* es un prefijo de *computadora*.

Sufijo de s: Una cadena que se forma suprimiendo cero o más símbolos desde la izquierda de una cadenas; por ejemplo *dora* es un sufijo de *computadora.*

Subcadena de s: Una cadena que se obtiene suprimiendo un prefijo y un sufijo de **s**; por ejemplo *tado* es una subcadena de *computadora.* Todo prefijo y sufijo de **s** es una subcadena de **s**, pero no toda subcadena de **s** es un prefijo o un sufijo. Para toda cadena **s**, tanto **s** como λ son prefijos, sufijos y subcadenas de **s**.

Prefijo, sufijo o subcadena propios de s: Cualquier cadena no vacía x que sea, respectivamente, un prefijo, sufijo o subcadena de **s** tal que s ≠x.

Subsecuencia de s: Cualquier cadena formada mediante la eliminación de cero o más símbolos no necesariamente continuos de **s**; por ejemplo, *cora* es una subsecuencia de *computadora.*

El término *lenguaje* se refiere a cualquier conjunto de cadenas de un alfabeto fijo. Esta definición es muy amplia, y abarca lenguajes abstractos como ∅ – el lenguaje vacío –, {∅} – el conjunto vacío –, o {λ} – el conjunto que sólo contiene la cadena vacía – como así también al conjunto de todos los programas de Pascal sintácticamente bien formados. Obsérvese asimismo que esta definición no atribuye ningún significado a las cadenas del lenguaje [Aho88].

1.3.2 - Operaciones aplicadas a lenguajes

Existen varias operaciones importantes que se pueden aplicar a los lenguajes. Para el análisis léxico, interesan principalmente la unión, la concatenación y la cerradura. También se puede extender el operador de "exponenciación" a los lenguajes definiendo L^0 como {λ}, y L^i como $L^{i-1}L$. Por lo tanto, L^i es L concatenado consigo mismo i-1 veces. A continuación se definen los operadores de unión, concatenación y cerradura.

Unión de L y M denotado por L ∪M:

se define $L \cup M = \{ s \mid s \text{ está en } L \text{ o } s \text{ está en } M \}$

Concatenación L y M denotado por LM:

se define $LM = \{ st \mid s \text{ está en } L \text{ y } t \text{ está en } M \}$

Cerradura de Kleene de L denotado por L*:

se define $L^* = \bigcup_{i=0}^{\infty} L^i$, L* denota cero o mas concatenaciones de L

Cerradura positiva de L denotado por L+:

se define $L+ = \bigcup_{i=1}^{\infty} L^i$, L+ denota una o mas concatenaciones de L

Ejemplo de las operaciones de los lenguajes: Sea L el conjunto de cadenas {"A", "B", ..., "Z", "a", "b", ..., "z"} y D el conjunto de números {0, 1, ..., 9}.[♣] Los siguientes son algunos ejemplos de nuevos lenguajes creados a partir de L y D mediante la aplicación de los operadores definidos anteriormente.

[♣] Usualmente se hace abuso de la notación denotando de la misma forma un conjunto de caracteres y al conjunto de cadenas unitarias formadas por estos caracteres C_1={"A","B"} y C_2={A,B}.

1. L ∪ D es el conjunto de letras y dígitos.
2. LD es el conjunto de cadenas que consta de una letra seguida de un dígito.
3. L^5 es el conjunto de todas las cadenas de cinco letras.
4. L* es el conjunto de todas las cadenas de letras, incluyendo λ, la cadena vacía.
5. L (L ∪ D)* es el conjunto de todas las cadenas de letras y dígitos que comienzan con una letra.
6. D+ es el conjunto de todas las cadenas de uno o más dígitos.

1.3.3 - Expresiones Regulares

En Pascal, un identificador es una letra seguida de cero o más letras o dígitos; es decir, un identificador es un miembro del conjunto definido en el cuarto lugar de la figura 1.3. En esta sección, se presenta una notación, llamada expresiones regulares, que permite definir de manera precisa conjuntos como éste. Con esta notación, se pueden definir los identificadores de Pascal como

letra (letra | dígito)*

La barra vertical aquí significa "o", los paréntesis se usan para agrupar subexpresiones, el asterisco significa "cero o más casos de" la expresión entre paréntesis, y la yuxtaposición de **letra** con el resto de la expresión significa concatenación.

Una expresión regular se construye a partir de expresiones regulares más simples utilizando un conjunto de reglas de definición. Cada expresión regular r representa un lenguaje L(r). Las reglas de definición especifican cómo se forma L(r) combinando de varias maneras los lenguajes representados por las subexpresiones de r.

Las siguientes son las reglas que definen las expresiones regulares del alfabeto ∑. Asociada a cada regla hay una especificación del lenguaje representado por la expresión regular que se está definiendo.

1) Ø es una expresión regular que representa al conjunto vacío – {Ø}–.

2) λ es una expresión regular que representa al conjunto que contiene la cadena vacía –{λ}–.

3) Si "a" es un símbolo de ∑, entonces "a" es una expresión regular que representa a {a}; por ejemplo, el conjunto que contiene la cadena "a". Aunque se usa la misma notación para las tres, técnicamente, la expresión regular "a" es distinta de la cadena "a" o del símbolo "a". El contexto aclarará si se habla de "a" como expresión regular, cadena o símbolo.

4) Suponiendo que r y s sean expresiones regular representadas por los lenguajes L(r) y L(s), entonces:

   a) r | s es una expresión regular que denota a L(r) ∪ L(s).
   b) rs es una expresión regular que denota a {ab | a∈L(r), b∈L(s) }.
   c) r* es una expresión regular que denota a (L(r))*.
   d) r es una expresión regular que denota a L(r).

Se dice que un lenguaje designado por una expresión regular es un conjunto regular.

La especificación de una expresión regular es un ejemplo de definición recursiva. Las reglas 1, 2 y 3 son la base de la definición; se usa el término símbolo básico para referirse a λ o a un símbolo de ∑ que aparezcan en una expresión regular. La regla 4 proporciona el paso inductivo.

Se pueden evitar paréntesis innecesarios en las expresiones regulares si se adoptan las siguientes convenciones:

- el operador unitario * tiene mayor precedencia y es asociativo por la izquierda,
- la concatenación tiene segunda precedencia y es asociativa por la izquierda,
- | tiene la menor precedencia y es asociativo por izquierda.

Según las convenciones, (b)|((c)*(d)) es equivalente a b|c*d. Estas dos expresiones designan el conjunto de cadenas que tiene una sola b, o cero o más c seguidas de una d.

Ejemplo de expresiones regulares. Sea ∑ = {a,b}.

1) La expresión regular a | b designa el conjunto {a, b}.

2) La expresión regular (a | b) (a | b) se corresponde con {aa, ab, ba, bb}, el conjunto de todas las cadenas de a y b de longitud dos. Otra expresión regular para este mismo conjunto es aa | ab | ba | bb.

3) La expresión regular a* designa el conjunto de todas las cadenas de cero o más a, por ejemplo, {λ, a, aa, aaa, aaaa, ... }.

4) La expresión regular (a | b)* designa el conjunto de todas las cadenas que contienen cero o más casos de una a o b, es decir, el conjunto de todas las cadenas de a y b. Otra expresión regular para este conjunto es (a* b*)*.

5) La expresión regular a | a*b designa el conjunto que contiene la cadena a y todas las que se componen de cero o más a seguidas de una b.

Si dos expresiones regulares r y s representan el mismo lenguaje, se dice que r y s son equivalentes y se escribe r = s. Por ejemplo (a | b) = (b | a).

Son varias la leyes algebraicas que cumplen las expresiones regulares y pueden ser utilizadas para transformar las expresiones regulares a formas equivalentes. A continuación se presentan estas leyes que se cumplen para las expresiones regulares r, s y t.

1. **Conmutatividad del | :**
   r | s = s | r.

2. **Asociatividad del | :**
   r | (s | t) = (r | s) | t.

3. **Asociatividad de la concatenación:**
   $(r\ s)\ t = r\ (s\ t)$.

4. **Distributividad de la concatenación con respecto a | :**
   $r\ (s \mid t) = r\ s \mid r\ t$ y $(s \mid t)\ r = s\ r \mid t\ r$.

5. **Elemento identidad de la concatenación:**
   $\lambda\ r = r$ y $r\ \lambda = r$

6. **Relación entre * y λ:**
   $r^* = (r \mid \lambda)^*$.

7. **Idempotencia del *:**
   $r^{**} = r^*$.

1.3.4 - Definiciones Regulares

Por conveniencia de notación, puede ser deseable dar nombres a las expresiones regulares y definir expresiones regulares utilizando dichos nombres como si fueran símbolos. Si ∑ es un alfabeto de símbolos básicos, entonces una *definición regular* es una secuencia de definiciones de la forma:

$$d_1 \longrightarrow r_1$$
$$d_2 \longrightarrow r_2$$
$$\ldots$$
$$d_n \longrightarrow r_n$$

donde cada d_i es un nombre distinto, y cada r_i es una expresión regular sobre los símbolos de $\sum \cup \{d_1, d_2, \ldots, d_{i-1}\}$, por ejemplo los símbolos básicos y los nombres previamente definidos. Al limitar cada r_i a los símbolos de ∑ y a los nombres previamente definidos, se puede construir una expresión regular en ∑ para cualquier r_i, reemplazándo una y otra vez los nombres de las expresiones regulares por las expresiones que designan. Si r_i utilizara d_j para alguna $j \geq i$, entonces r_i se podría definir recursivamente y este proceso de sustitución podría no tener fin.

Ejemplo de definiciones regulares:

Como ya se estableció antes, el conjunto de identificadores Pascal es el conjunto de cadenas de letras y dígitos que empiezan con una letra. A continuación se da una definición regular para este conjunto.

Letra → A | B | ... | Z | a | b | ... | z
Dígito → 0 | 1 | ... | 9
Identificador → Letra (Letra | Dígito) *

Los números sin signo de Pascal son cadenas, como 5540, 2.45, 78.432E4, o 1.222E-4. La siguiente definición regular proporciona una especificación precisa para esta clase de cadenas:

Dígito → 0 | 1 | ... | 9
Dígitos → **Dígito Dígito***
Fracción_optativa → **. Dígitos** | λ
Exponente_optativo → (E (+ | - | λ) **Dígitos** | λ
Número → **Dígitos Fracción_optativa Exponente_optativo**

1.3.5 – Abreviaturas en la notación

Como ciertas construcciones aparecen con tanta frecuencia en una expresión regular es conveniente introducir algunas abreviaturas [Aho88].

1) *Uno o más casos:* El operador unitario postfijo + significa "uno o más casos de". Si r es una expresión regular que designa el lenguaje L(r), entonces (r)+ es una expresión regular que designa al lenguaje (L(r))+. Así, la expresión regular a+ representa al conjunto de todas las cadenas de una o más a. El operador + tiene la misma precedencia y asociatividad que el operador *. Las dos identidades algebraicas r* = r+ | λ y r+ = r r* relacionan los operadores * y +.

2) *Cero o un caso:* El operador unitario postfijo **?** significa "cero o un caso de". La notación de r? es una abreviatura de r | λ. Si r es una expresión regular, entonces (r)? es una expresión regular que designa el lenguaje L(r)∪{ λ}.

3) *Clases de Caracteres:* La notación [abc], donde a, b y c son símbolos del alfabeto, designa la expresión regular a | b | c. Una clase abreviada de caracter como [a-z] designa la expresión regular a | b | ... | z. Utilizando clases de caracteres, se puede definir los identificadores como cadenas generadas por la expresión regular [A-Za-z] [A-Za-z0-9]*.

Usando las abreviaturas antes mencionadas se puede rescribir la definición regular para **número** del ejemplo anterior de la siguiente forma:

Dígito → [0-9]
Dígitos → **Dígito+**
Fracción_optativa → **(. Dígitos) ?**
Exponente_optativo → (E (+ | -)? **Dígitos**)?
Número → **Dígitos Fracción_optativa Exponente_optativo**

1.4 - Autómatas Finitos

La teoría de autómatas finitos fue presentada desde sus orígenes en una gran diversidad de aspectos. Desde un punto de vista, es una rama de la matemática conectada con la teoría algebraica de semigrupos y álgebras asociativas. Desde otro punto de vista es una rama del diseño de algoritmos para manipulación de cadenas y procesamiento de secuencias.

La primer referencia histórica a autómatas finitos es una publicación de S.C. Kleene de 1954 con el teorema básico conocido como el teorema de Kleene [Kle72]. La publicación de Kleene fue un reagrupamiento de ideas matemáticas de dos investigadores

del MIT – Massachussets Institute Technology –, W. McCulloch and W. Pitts quienes presentaron este trabajo en el año 1943, un modelo lógico del comportamiento del sistema nervioso que luego fue modificado a un modelo de una maquina de estados finitos [McC43]. De echo, un autómata finito puede ser visto como un modelo matemático el cual es tan elemental como posible, en el sentido de que la maquina tiene un tamaño de memoria fijo y limitado, independientemente del tamaño de la entrada. El origen histórico del modelo de estados finitos puede ser localizado al comienzo del siglo con la noción de *cadena de Markov*. Una cadena de Makov es el modelo de un proceso estocástico en el cual la probabilidad de un evento sólo depende del evento que sucedió anteriormente en un límite de distancia aproximado.

Desde los orígenes, la teoría de autómata finito fue desarrollada teniendo en cuenta sus posibles aplicaciones y su interés como objeto matématico. En una fase primitiva, los autómatas finitos aparecieron como un desarrollo de circuitos lógicos obtenidos para introducir la secuenciabilidad de operaciones. Esto lleva a la noción de *circuitos secuenciales* el cual es aún de interés en el campo de diseño de circuitos. Pero las principales aplicaciones de los autómata finitos en la actualidad, están relacionadas con el procesamiento de texto. Por ejemplo, en la fase del proceso de compilación, conocida como análisis léxico, el código del programa es transformado de acuerdo a operaciones simples tal como remover blancos, o reconocer identificadores y palabras reservadas del lenguaje. Este proceso elemental es generalmente ejecutado por algoritmos que son autómatas finitos y usualmente puede ser diseñado y manejado por los métodos de la teoría de autómatas. De la misma manera, en procesamiento de lenguaje natural, los autómatas finitos frecuentemente llamados *redes de transición,* son utilizadas para describir alguna fase del análisis léxico. Estas aplicaciones de los autómatas para procesamiento de texto tienen una natural extensión en las áreas como compresión de texto, manipulación de archivos o más remotamente al análisis de largas secuencias de encuentro de moléculas en la biología molecular. Otra aplicación de estos autómatas, esta relacionada al estudio de procesos paralelos. De echo, la mayoría de los modelos de concurrencia y sincronización de procesos usa métodos, explícitos o a veces implícitos, los cuales son autómatas finitos [Lee98].

Un “reconocedor” de un lenguaje es un programa que toma como entrada una cadena x y responde “si”, si x es una frase del programa, y “no”, si no lo es. Para generar dicha herramienta, se compila una expresión regular en un reconocedor construyendo un diagrama de transiciones generalizado llamado autómata finito. Un autómata finito puede ser determinístico o no determinístico, en donde “no determinístico” significa que en un estado se puede dar el caso de tener más de una transición para el mismo símbolo de entrada.

Tanto los autómatas finitos determinísticos como los no determinísticos pueden reconocer con precisión a los conjuntos regulares. Por lo tanto, ambos pueden reconocer con exactitud lo que denotan las expresiones regulares. Sin embargo, hay un conflicto entre espacio y tiempo; mientras que un autómata finito determinístico puede dar reconocedores más rápidos que uno no determinístico, un autómata finito determinístico puede ser mucho mayor – en tamaño – que un autómata no determinístico equivalente. En las siguientes secciones se introducen métodos para convertir expresiones regulares en ambas clases de autómatas.

1.4.1 - Autómatas Finitos no determinístico con transiciones λ

Un autómata finito no determinístico con transiciones λ (AFNλ) es un modelo matemático:

$$AFN\lambda = < K, (\sum \cup \lambda), \delta, q_0, F >$$

donde:

1) K es el conjunto de estados o nodos.
2) $(\sum \cup \lambda)$ un conjunto de símbolos de entrada llamado *alfabeto de símbolos de entrada.*
3) δ una función de transición que transforma pares (estado, símbolo) en conjuntos de estados. δ: K x $(\sum \cup \lambda) \rightarrow P(K)$
4) q_0 estado inicial, $q_0 \in K$.
5) F conjunto de estados finales o estados de aceptación, $F \subseteq K$.

La ejecución de un *Autómata Finito no Determinístico con Transiciones λ* se puede formalizar la siguiente manera:

*Configuración (*K x $\sum$*): Una configuración es un par (q,α) siendo q el estado actual y α la parte derecha de la cadena, es decir lo que falta procesar de la cadena de entrada.

Definición de transición |- : El símbolo |- indica que se pasa de una configuración a otra, es decir, se mueve de un estado a otro:

$$(q, a\ \alpha) \vdash (r, \alpha) \text{ si } r \in \delta(q,a) \text{ con } a \in \sum \cup \{\lambda\}$$

Definición de transición |-* : El símbolo |-* significa que se pueden dar una secuencia de cero o más movimientos |- que lleven de una a otra configuración. Se puede definir recursivamente como:

1. $(q, \lambda) \vdash^* (q, \lambda)$
2. $(q, a\ \alpha) \vdash^* (r, \lambda) \Leftrightarrow (q, a\ \alpha) \vdash (q', \alpha) \vdash^* (r, \lambda)$

Definición de cadena aceptada: Dada una cadena $\alpha \in \sum^*$, se dice que la cadena es aceptada por el AFNλ o que α pertenece al lenguaje generado por el autómata si solo si existe un estado final f tal que $(q_0, \alpha) \vdash^* (f, \lambda)$.
Formalmente:

Siendo M: AFNλ, $\alpha \in L(M) \Leftrightarrow \exists f \in F$ tal que $(q_0, \alpha) \vdash^* (f, \lambda)$

Un AFNλ se puede representar gráficamente mediante un grafo dirigido etiquetado, llamado *grafo de transiciones*. Los nodos son estados y las aristas etiquetadas representan la función de transición. Este grafo se parece a un diagrama de transiciones en donde las aristas pueden etiquetarse con el símbolo especial λ y con símbolos de entrada, pero el mismo caracter puede etiquetar dos o más transiciones.

La figura 1.4 muestra el grafo de transiciones de un AFNλ que reconoce al lenguaje ((a | b)* a b b) | b*. El conjunto de estados del AFNλ es {0, 1, 2, 3, 4, 5} y el alfabeto de

símbolos de entrada es {a, b, λ}. El estado 0 se considera el estado inicial y los estados 4 y 5 estados finales. Los estados finales se representan mediante un círculo doble.

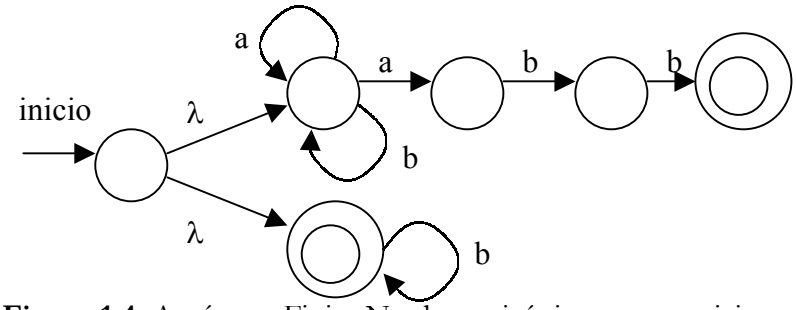


Figura 1.4: Autómata Finito No determinístico con transiciones λ.

Para describir un AFNλ, se puede utilizar un grafo de transiciones y aplicarse la función de transición de un AFNλ de varias formas. La implementación más sencilla es una *tabla de transiciones* en donde hay una fila por cada estado y una columna por cada símbolo de entrada y λ, si es necesario. La entrada para la fila i y el símbolo a en la tabla es el conjunto de estados que puede ser alcanzado por una transición del estado i con la entrada a. En la figura 1.5 se muestra la tabla de transiciones para el AFNλ de la figura 1.4.

Estado	**Símbolo de Entrada**		
	a	**B**	**λ**
0	-	-	{1,5}
1	{1,2}	{1}	-
2	-	{3}	-
3	-	{4}	-
4	-	-	-
5	-	{5}	-

Figura 1.5: Tabla de transiciones para el AFNλ de la figura 1.4.

La representación en forma de tabla de transiciones tiene la ventaja de que proporciona rápido acceso a las transiciones de un determinado estado por un caracter dado; su inconveniente es que puede ocupar gran cantidad de espacio cuando el alfabeto de entrada es grande y la mayoría de las transiciones son hacia el conjunto vacío. Las representaciones de listas de adyacencias de la función de transición proporcionan implementaciones más compactas, pero el acceso a una transición dada es más lento. Vale aclarar que se puede transformar fácilmente cualquiera de estas implementaciones de un AFNλ en otra.

Para fortalecer los conceptos antes mencionados, a continuación se presenta un ejemplo de un autómata que reconoce al lenguaje (a b+ a) | (b+ a+). La figura 1.6 muestra el grafo de transiciones del autómata, la figura 1.7 muestra la tabla de transiciones y por último la figura 1.8 muestra el conjunto de estados, conjunto de símbolos de entrada, la definición por extensión de la función de transición, el conjunto de estados finales y el estado inicial del AFNλ.

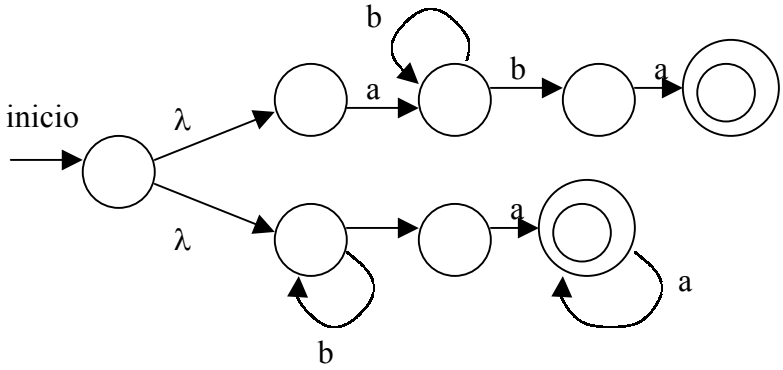


Figura 1.6: Autómata Finito No Determinístico con transiciones λ.

Estado	Símbolo de Entrada		
	A	**b**	**λ**
0	-	-	{1,5}
1	{2}	-	-
2	-	{2,3}	-
3	{4}	-	-
4	-	-	-
5	-	{5,6}	-
6	{7}	-	-
7	{7}	-	-

Figura 1.7: Tabla de transiciones para el AFNλ de la figura 1.6.

Conjunto de Estados	{0,1,2,3,4,5,6,7}
Conjunto de Símbolos de Entrada	{a, b, λ}
Definición por extensión de la Función de Transición δ	δ(0, λ) = 1 ; δ(0, λ) = 5 ; δ(1, a) = 2 δ(2, b) = 2 ; δ(2, b) = 3 ; δ(3, a) = 4 δ(5, b) = 5 ; δ(5, b) = 6 ;δ(6, a) = 7 ; δ(7, a) = 7
Conjunto de Estados Finales	{4,7}
Estado Inicial	0

Figura 1.8: Definición del AFNλ.

1.4.2 - Autómatas Finitos Deterministicos

Un autómata finito determinístico (AFD) es un caso especial de un autómata finito no determinístico en el cual:

1) ningún estado tiene una transición λ, es decir una transición con la entrada λ, y;
2) para cada estado s y cada símbolo de entrada a, hay a lo sumo una arista etiquetada a que sale de s [Aho88].

Un autómata finito determinístico se lo puede definir como una 5-upla:

$$AFD = < K, \Sigma, \delta, q_0, F >$$

donde:

1) K es el conjunto de estados o nodos.
2) $\sum$ un conjunto de símbolos de entrada.
3) δ una función de transición que transforma pares (estado, símbolo) en otro estado.
   δ: K x $\sum \rightarrow$ K
4) q_0 estado inicial, $q_0 \in$ K.
5) F conjunto de estados finales o estados de aceptación, F $\subseteq$ K.

Configuración: K x $\sum$*.

Definición de transición:

(q, a α) |- (r, α) si r = δ(q, a) con a $\in \sum$

Definición de cadena aceptada:

Siendo M: AFD, $\alpha \in$ L(M) $\Leftrightarrow \exists$ f $\in$ F tal que (q_0, α) |-*(f, λ)

En la figura 1.9 se ve el grafo de transiciones de un autómata finito determinístico que acepta el mismo lenguaje (a b+ a) | (b+ a+) aceptado por el AFNλ de la figura 1.6.

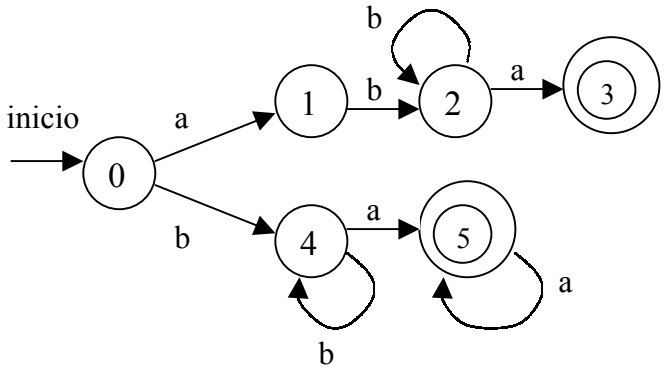


Figura 1.9: Autómata Finito Determinístico.

Estado	Símbolo de Entrada	
	a	**b**
0	1	5
1	-	2
2	3	2
3	-	-
4	5	4
5	5	-

Figura 1.10: Tabla de transiciones para el AFD de la figura 1.9.

Conjunto de Estados	{0,1,2,3,4,5}
Conjunto de Símbolos de Entrada	{a, b}
Definición por extensión de la Función de Transición δ	δ(0, a) = 1 ; δ(0, b) = 4 ; δ(1, b) = 2 δ(2, a) = 3 ; δ(2, b) = 2; δ(4, a) = 5 δ(4, b) = 4; δ(5, a) = 5
Conjunto de Estados Finales	{3,5}
Estado Inicial	0

Figura 1.11: Definición del AFD.

Un autómata finito determinístico tiene a lo sumo una transición desde cada estado por cualquier entrada. Si se está usando una tabla de transiciones para representar la función de transición de un AFD, entonces cada entrada en la tabla de transiciones es un solo estado. Como consecuencia, es muy fácil determinar con un algoritmo si un autómata finito determinístico acepta o no una cadena de entrada, puesto que hay a lo sumo un camino desde el estado de inicio etiquetado con esa cadena. Para más información del algoritmo que simula un AFD ver [Aho88].

Para el caso de un autómata finito no determinístico con transiciones λ, es más difícil crear un algoritmo que simule el autómata ya que en la función de transición δ, para una entrada "a", puede tener varios valores. Por ejemplo, el autómata de la figura 1.6 tiene dos transiciones desde el estado 2 con la entrada b; es decir puede ir al estado 2 o al 3. De la misma forma, el estado 0 con la entrada λ tiene dos transiciones: una al estado 1 y otra al estado 5. En [Aho88] se propone una forma de simular un AFNλ por medio de dos pilas cuyo mecanismo funciona leyendo la entrada de un caracter a la vez y calculando todo el conjunto de estados en los que podría estar el autómata después de haber leído todos los prefijos de la entrada.

Puesto que el algoritmo de simulación de un AFD es mucho más sencillo que el algoritmo que simula un AFNλ es conveniente obtener autómatas finitos determinísticos. Pero no siempre disponemos de AFD, es por eso que existe un algoritmo que convierte un AFNλ en un AFD equivalente. Dicho algoritmo se detalla en la siguiente sección.

1.4.3 - Conversión de un AFNλ en un AFD equivalente

Ahora se introduce un algoritmo para construir un AFD a partir de un AFNλ que reconozca el mismo lenguaje. Este algoritmo, a menudo llamado *construcción de subconjuntos*, es útil para simular un AFNλ por medio de un programa.

En la tabla de transiciones de un AFNλ, cada entrada es un conjunto de estados; en la tabla de transiciones de un AFD, cada entrada es tan sólo un estado. La idea general tras la conversión AFNλ a AFD es que cada estado de AFD corresponde a un conjunto de estados del AFNλ. El AFD utiliza un estado para localizar todos los posibles estados en los que puede estar el AFNλ después de leer cada símbolo de la entrada. Es decir, después de leer la entrada a_1, a_2, ..., a_n, el AFD se encuentra en un estado de inicio del AFNλ a lo largo de algún camino etiquetado con $a_1a_2...a_n$. El número de estados del AFD puede ser exponencial en el número de estados del AFNλ, pero en la práctica este peor caso ocurre raramente.

En este algoritmo se utilizan las operaciones *cerradura-λ (s), cerradura-λ (T) y mueve(T,a)* para localizar los conjuntos de los estados del AFNλ en donde s representa un estado del AFNλ, y T, un conjunto de estados del AFNλ. Estas operaciones se describen a continuación:

- *cerradura-λ (s):* Calcula el conjunto de estados del AFNλ alcanzables desde el estado s del AFNλ con transiciones λ solamente.
- *cerradura-λ (T):* Calcula el conjunto de estados del AFNλ alcanzables desde algún estado s en T con transiciones λ solamente.
- *mueve(T,a): Calcula el conjunto de estados del AFN*λ hacia los cuales hay una transición con el símbolo de entrada a desde algún estado s en T del AFNλ.

El algoritmo que calcula la *cerradura-λ (T)* es una típica búsqueda en un grafo de nodos alcanzables desde un conjunto dado de nodos. En este caso, los estados de T son el conjunto de nodos, y el grafo está compuesto solamente por las aristas del AFNλ etiquetadas por λ. Un algoritmo sencillo para calcular *cerradura-λ (T)* utiliza una estructura de datos tipo pila para guardar los estados en cuyas aristas no se hayan buscado transiciones etiquetadas con λ. El algoritmo es el siguiente:

Algoritmo del Cálculo de cerradura-λ:

```
Insertar todos los estados de T en la pila;
cerradura-λ (T)= T;
mientras pila no vacía hacer
        t = obtener tope de la pila;
        para cada estado u con una arista desde t a u etiquetada con λ hacer
                si u no está en cerradura-λ (T) entonces
                        cerradura-λ (T) = cerradura-λ (T) ∪ u
                        insertar u en la pila
                fin_si
        fin_para
fin_mientras
```

Algoritmo Construcción de Subconjuntos, construcción de un AFD a partir de un AFNλ*:*

Entrada: Un AFNλ.
Salida: Un AFD que acepta el mismo lenguaje.

Método: El algoritmo construye una tabla de transiciones *transicionesAFD* para el AFD. Cada estado del AFD es un conjunto de estados del AFNλ y se construye *transicionesAFD* de modo que el AFD simulará "en paralelo" todos los posibles movimientos que el AFNλ puede realizar con una determinada cadena de entrada. Antes de detectar el primer símbolo de entrada, el AFNλ se puede encontrar en cualquiera de los estados del conjunto *cerradura-λ (s_0)*, donde s_0 es el estado inicial del AFNλ. Supóngase que exactamente los estados del conjunto T son alcanzables con una secuencia dada de símbolos de entrada, y sea a el siguiente símbolo de entrada. Con una entrada a, el AFNλ puede trasladarse a cualquiera de los estados del conjunto *mueve(T,a)*. Cuando se permiten transiciones λ, el AFNλ

puede encontrarse en cualquiera de los estados de *cerradura-λ (T, a)* después de ver la entrada a.

Algoritmo
al inicio, *cerradura-λ (s_0)* es el único estado dentro de *estadosAFD* y no está marcado.
mientras haya un estado no marcado T en *estadosAFD* hacer
    Para cada símbolo de entrada a ∈ ∑ hacer
        U = *cerradura-λ (mueve(T,a));*
        Si U no está en *estadosAFD* entonces
            Insertar U como estado no marcado a *estadosAFD*
        Fin_si
        transicionesAFD (T, a) = U
    Fin_para
Fin_mientras
Fin_Algoritmo

Se construyen *estadosAFD*, el conjunto del AFD, y t*ransicionesAFD*, la tabla de transiciones del AFD, de la siguiente forma. Cada estado del AFD corresponde a un conjunto de estados del AFDλ en los que podría estar el autómata después de leer alguna secuencia de símbolos de entrada, incluidas todas las posibles transiciones-λ anteriores o posteriores a la lectura de símbolos. El estado de inicio del AFD es *cerradura-λ (s_0)*. Se añaden los estados y las transiciones al AFD generado utilizando el algoritmo antes presentado. Un estado del AFD es un estado final si es un conjunto de estados del AFNλ que contenga al menos un estado final del AFNλ.

1.4.4 – Construcción de un AFD a partir de una expresión regular

Las expresiones regulares son convenientes para especificar tokens, pero se necesita un formalismo que pueda ser implementado por un algoritmo. Es por esto que es beneficioso construir AFD's a partir de expresiones regulares.

Los pasos de la construcción del AFD son los siguientes:

1) Se aumenta la expresión regular de entrada agregándole un símbolo # que no está definido en el alfabeto de entrada.
2) Se construye un árbol sintáctico A para la expresión regular extendida (e#).
3) Se calculan las funciones: *nullable, firstpos, lastpos y followpos.* La función *followpos* se construyen a partir de las funciones: *firstpos y lastpos*.
4) Se construye el AFD a partir de *followpos.*

Paso 1: Aumentar la Expresión Regular

Este paso se realiza para poder determinar los estados finales una vez que se halla construido el autómata. Para aumentar la expresión regular es requisito utilizar un símbolo no definido en el alfabeto ya que si se utiliza otro símbolo, no existiría distinción entre el símbolo de finalización de expresión regular (#) y el símbolo del alfabeto.

Paso 2: Construcción del Arbol Sintáctico

La construcción del árbol sintáctico se realiza a partir de la expresión regular extendida e#. Las hojas del árbol, además de tener la información del caracter, tienen un atributo que es un número entero denominado posición del nodo. Todas las hojas del árbol tienen posiciones diferentes.

El árbol sintáctico está definido de la siguiente manera:

Arbol → Arbol :|: Arbol
| Arbol :.: Arbol
| :*: Arbol
| :+: Arbol
| :?: Arbol
| Hoja (∑, Posición)
| λ

El siguiente ejemplo muestra un árbol sintáctico para la expresión (0 | 1)* (0 | 1) **.** (0 | 1) #.

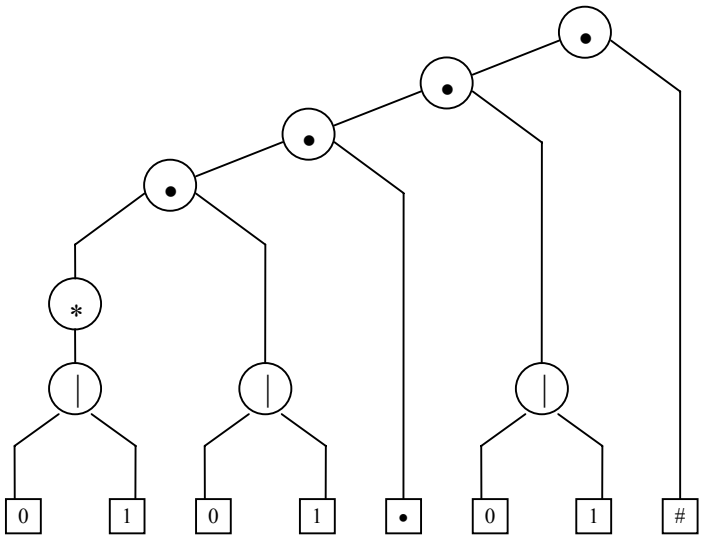


Figura 1.12: Arbol sintáctico para la expresión (0 | 1)* (0 | 1) **.** (0 | 1) #.

Paso 3: Calculo de las Funciones: nullable, firstpos, lastpos y followpos.

Dada e:ER; i, j posiciones del árbol sintáctico de e; se define:

nullable: ER → Lógico	nullable(e)=Verdadero ⇔ λ∈ L(e)
firstpos: ER → Posición	firstpos(e) es el conjunto de todas las posiciones que puedan generar un caracter que sea cabeza de alguna cadena de L(e)
lastpos: ER → Posición	lastpos(e) es el conjunto de todas las posiciones que puedan generar un caracter que sea el último de alguna cadena de *L*(e)
followpos:ER → Posición	j∈followpos(e) ⇔ existe alguna cadena generada por e de la forma **...ab...** para la cual **a** puede ser generada por la posición **i** y **b** por la posición **j.**

Las funciones *nullable, firstpos* y *lastpos* se computan mediante la implementación de funciones recursivas sobre el árbol sintáctico.

nullable (Hoja (c , p))= (c=λ)
nullable (:*: e) = Verdadero
nullable (:+: e) = Falso
nullable (:?: e) = Verdadero
nullable (e1 :|: e2) = nullable(e1) ∨ nullable(e2)
nullable (e1 :.: e2) = nullable(e1) ∧ nullable(e2)

firstpos(Hoja (λ , p)) = ∅
firstpos(Hoja (x , p)) = { p } si x≠λ
firstpos(:*: e) = firstpos(e)
firstpos(:+: e) = firstpos(e)
firstpos(:?: e) = firstpos(e)
firstpos(e1 :.: e2) = firstpos(e1) si no nullable(e1)
o firstpos(e1) U firstpos(e2) en otro caso
firstpos(e1 :|: e2) = firstpos(e1) U first pos (e2)

lastpos(Hoja (λ , p)) = ∅
lastpos(Hoja (x , p)) ={p} si x≠λ
lastpos (:*: e) = firstpos(e)
lastpos (:+: e) = firstpos(e)
lastpos (:?: e) = firstpos(e)
lastpos(e1 :.: e2) = lastpos(e2) si no nullable(e2)
o lastpos(e1) U lastpos(e2) en otro caso
lastpos(e1 :|: e2) = lastpos(e1) U lastpos (e2)

Finalmente se computa *followpos* recorriendo el árbol sintáctico de e# primero en profundidad y ejecutando en cada nodo n visitado:

Si n = e1 :.: e2
para cada i en lastpos(e1) hacer followpos(i) = followpos(i) U firstpos(e2)

Si n = :*: e
para cada i en lastpos(e) hacer followpos(i) = followpos(i) U firstpos(e)

Si n = :+: e
para cada i en lastpos(e) hacer followpos(i) = followpos(i) U firstpos(e)

Si n= (e1 :|: e2) followpos(i) = followpos(i).

Si n= :?: e followpos(i) = followpos(i).

Si n= Hoja (x , p) followpos(i) = followpos(i).

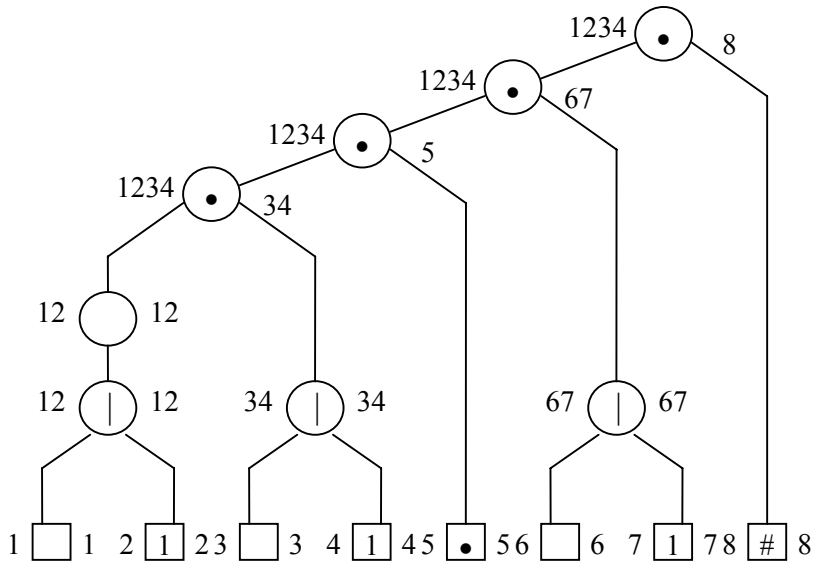


Figura 1.13:Arbol sintáctico decorado con las funciones *firstpos* y *lastpos*.

La figura anterior muestra el árbol sintáctico para la expresión regular (0 | 1)* (0 | 1) **.** (0 | 1) # decorado con *firspos y lastpos* a la izquierda y a la derecha de cada nodo respectivamente. *Followpos* se muestra en la siguiente tabla.

Posición	followpos(posición)	posición	followpos(posición)
1	{ 1, 2, 3, 4 }	5	{ 6, 7 }
2	{ 1, 2, 3, 4 }	6	{ 8 }
3	{ 5 }	7	{ 8 }
4	{ 5 }	8	-

Paso 4: Construcción del AFD a partir de un ER.

Algoritmo: Construcción de un AFD equivalente a una ER [Aho88]:

Entrada: e: ER
Salida: M: < K, Σ, δ, Q_0, F >: AFD que reconoce L(e).

4.1 Se construye el árbol sintáctico de e #.
4.2 Se computan las funciones *nullable, firstpos, lastpos y followpos*.
4.3 Se construye el correspondiente autómata **M:AFD** con el siguiente algoritmo:

K: Conjunto de conjuntos de posiciones, con la posibilidad de marcar sus elementos
Q, R: elemento de K

Q_0 = firstpos(raiz_del_arbol_sintáctico)
K:={Q_0} sin marcar
mientras haya Q$\in$K sin marcar
    marcar Q
    para cada a$\in\Sigma$

{ definir R=δ(Q,a); agregar R a K si no estaba}

R:= ∅;
para cada i∈Q tal que carac(i) = a hacer
R:= R U followpos(i);
definir δ(Q, a)=R
si R∉K : agregar R a K sin marcar
F:= { Q∈ K / ∃ i∈Q : carac(i)= # }
Si ∅∈K definir δ(∅,a) = ∅

Para el ejemplo antes mencionado, (0 | 1)* (0 | 1) **.** (0 | 1) #, el autómata generado por el algoritmo es el siguiente:

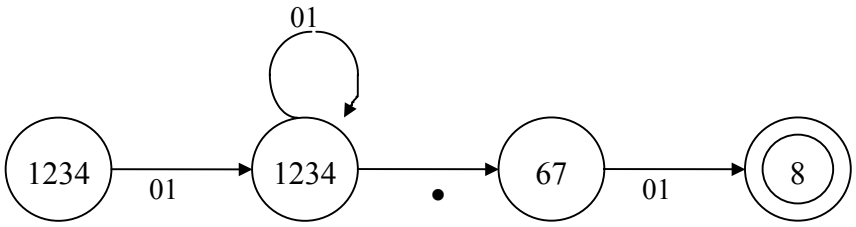


Figura 1.14: AFD de la expresión regular (0 | 1)* (0 | 1) **.** (0 | 1) #.

1.4.5 - Minimización del número de estados de un AFD

Una conclusión teórica importante es que todo conjunto regular es reconocido por un AFD con un conjunto de estados mínimo. En esta sección se presenta un algoritmo que permite construir un AFD con una cantidad mínima de estados sin afectar al lenguaje que se está reconociendo.

Se dice que la cadena *w distingue* al estado s del estado t si, empezando en el estado s del AFD y recorriéndolo con la cadena *w*, se llega a un estado final, pero para la ejecución del AFD desde el estado t con la cadena w se llega a un estado que no es final o viceversa. Por ejemplo, λ distingue cualquier estado final de cualquier estado no final.

El algoritmo para minimizar el número de estados de un AFD funciona encontrando todos los grupos de estados que pueden ser diferenciados por una cadena de entrada. Cada grupo de estados que no puede diferenciarse se fusiona en un único estado. El algoritmo opera manteniendo y refinando una partición del conjunto de estados. Cada grupo de estados dentro de la partición está formado por estados que aún no han sido distinguidos unos de otros, y todos los pares de estados seleccionados entre diferentes grupos han sido considerados distinguibles por un cadena de entrada.

Al comienzo de la ejecución del algoritmo, la partición consta de dos grupos: los estados finales y los estados no finales. El paso fundamental consiste en tomar un grupo de estados, por ejemplo A =$\{ s_1, s_2, ..., s_k\}$ y un símbolo de entrada a y comprobar qué transiciones tiene los estados $s_1, s_2, ..., s_k$ con la entrada a. Si existen transiciones hacia dos o más grupos distintos de la partición actual, entonces se debe dividir A para que las particiones desde los subconjuntos de A queden todas confinadas en un único grupo de la partición actual. Supóngase, por ejemplo, que s_1 y s_2 van a los estados t_1y t_2 con la entrada a y que t_1 y t_2 están en diferentes grupos de la partición. Entonces se debe dividir A en al menos dos subconjuntos, para que un subconjunto contenga a s_1, y el otro a s_2. Obsérvese que t_1 y t_2 son diferenciados por alguna cadena w, y s_1 y s_2 por la cadena aw. Este proceso

de dividir grupos dentro de la partición en curso se repite hasta que no sea necesario dividir ningún otro grupo.

Algoritmo: Minimización del número de estados de un AFD.

Entrada: Un AFD M con un conjunto de estados S, un conjunto de entradas Σ, transiciones definidas para todos los estados y las entradas, un estado inicial S_0 y un conjunto de estados finales F.
Salida: Un AFD M' que acepta el mismo lenguaje que M y tiene el menor número de estados posibles.

Método:

1. Constrúyase una partición inicial Π del conjunto de estados con dos grupos: los estados finales F y los no finales S-F.
2. Aplíquese el procedimiento a Π *contrucción_partitición* para construir una nueva partición Π_{nueva}.
3. $\Pi_{nueva} = \Pi$, hacer $\Pi_{final} = \Pi$ y continuar con el paso 4. Sino, repetir el paso 2 con $\Pi = \Pi_{nueva}$.
4. Escójase un estado en cada grupo de la partición Π_{final} como representante de este grupo. Los representantes serán los estados de AFD reducidos M'. Sea s un estado representante, y supóngase que con la entrada de a hay una transición de M desde s a t. Sea r el representante del grupo de t (r puede ser t). Entonces M' tiene una transición desde s a r con la entrada a. Sea el estado inicial de M' el representante del grupo que contiene al estado de inicio s0 de M, y sean los estados finales de M' los representantes que están en F. Obsérvese que cada grupo de Π_{final} consta únicamente de estados en F o no tiene ningún estado en F.
5. Si M'tiene un estado inactivo, es decir, un estado *d* que no es estado final y que tiene transiciones hacia él mismo con todos los símbolos de entrada, elimínese *d* de M'. Elimínense igualmente todos los estados que no sean alcanzables desde el estado inicial. Todas las transiciones a *d* desde otros estados se convierten en indefinidas.

Fin_Algoritmo

Algoritmo: contrucción_partitición Π_{nueva} (construye una nueva partición)

Para cada grupo G de Π hacer
Partición de G en subgrupos tales que dos estados s, t de G estan en el mismo subgrupo si, y solo si, para todos los símbolos de entrada a, los estados s y t tienen transiciones en a hacia estados del mismo grupo de Π ;

Sustituir G en Π_{nueva} por el conjunto de todos los subgrupos formados.
Fin_para
Fin_Algoritmo

Ejemplo de aplicación del algoritmo de minimización de estados:

Considérese el siguiente AFD:

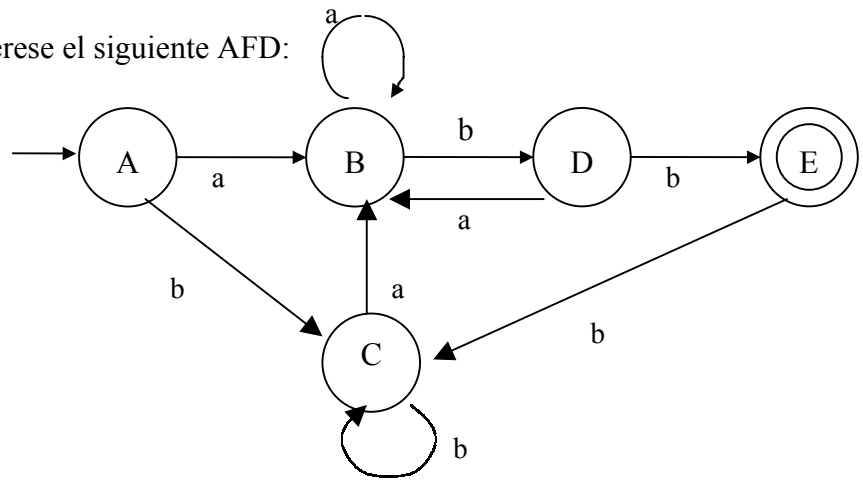


Figura 1.15: AFD que reconoce el lenguaje (a|b)* abb.

La partición inicial Π consta de dos grupos: el estado final (E) y los estados no finales (ABCD). Para construir una partición nueva, se utiliza el algoritmo *contrucción_partitición* Π_{nueva} anteriormente presentado. Considérese la partición (E), puesto que este grupo consta de un solo estado, no se puede dividir la partición así que (ET) se coloca en Π_{nueva} . Luego el algoritmo considera la partición (ABCD). Para la entrada a, cada uno de estos estados tiene una transición a B, así que todos podrían permanecer en el mismo grupo en lo que a la entrada a se refiere. Sin embargo, con la entrada b, A, B y C van a miembros del grupo (ABCD) de Π, mientras que D va a E, un miembro de otro grupo. Por lo tanto, dentro de Π_{nueva} el grupo (ABCD) se debe dividir en dos nuevos grupos, (ABC) y (D). Entonces la nueva partición Π_{nueva} esta formada por (ABC) (D) (E).

En el siguiente recorrido por el algoritmo de minimización de estados, nuevamente no hay división en la entrada a, pero (ABC) debe dividirse en dos nuevos grupos (AC) (B) debido a que para la entrada b, A y C tienen ambas una transición a C, mientras que B tiene una transición a D. Así que la nueva partición Π_{nueva} es (AC) (B) (D) (E).

Para la siguiente pasada del algoritmo, no se pueden dividir ninguno de los grupos de un solo estado. La única posibilidad es intentar dividir (AC). Sin embargo, A y C van al mismo estado B en la entrada a y al mismo estado C en la entrada b. Por lo tanto, después de este recorrido, $\Pi_{nueva} = \Pi$. Π_{final} es entonces (AC) (B) (D) (E).

El autómata minimizado es el siguiente:

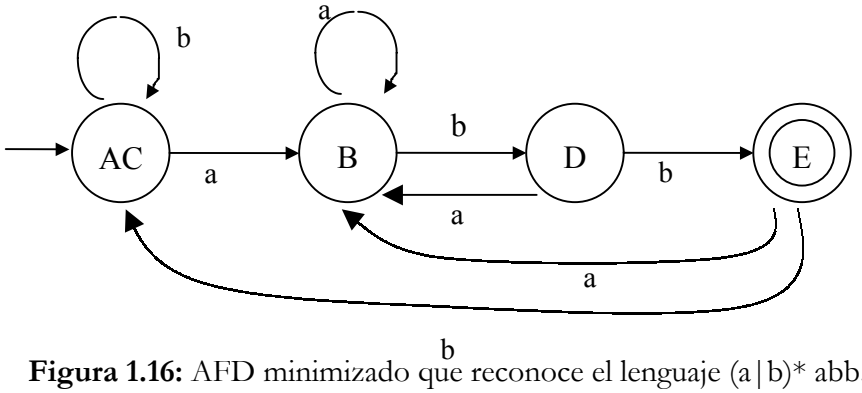


Figura 1.16: AFD minimizado que reconoce el lenguaje (a|b)* abb.

1.5 – Generación de Analizadores Léxicos

Un seudo algoritmo para construir un analizador léxico, utilizando los conceptos antes mencionados, es el siguiente:

1. Dar las expresiones regulares que definen los tokens.
2. Construir un AFD para cada expresión reglar dada en el punto anterior utilizando el algoritmo que pasa una expresión regular a un AFD – ver sección 1.4.4.
3. Construir un AFNλ uniendo los AFD, obtenidos en el punto anterior, de la siguiente forma: se define un nuevo estado inicial y se define una transición λ desde este estado inicial a cada uno de los estados iniciales de los AFD's. Los estados finales del AFNλ es la unión de los estados finales de cada uno de los AFD.

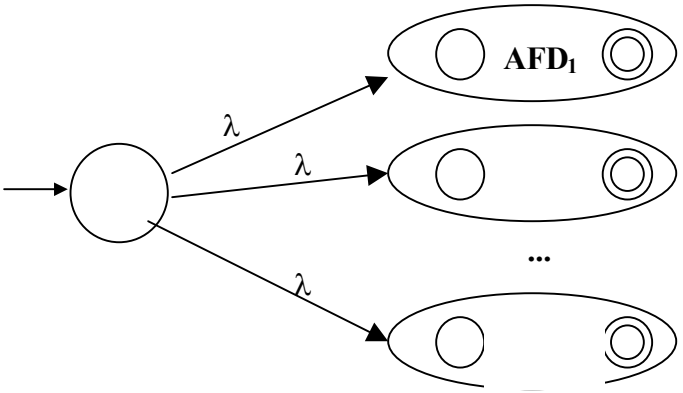


Figura 1.17: AFNλ construido a partir de n AFD's

4. Construir un AFD a partir del AFNλ utilizando el algoritmo presentado en la sección 1.4.3.
5. Implementar el algoritmo que simula un AFD.

Observaciones:

- ✓ Tanto los estados finales del AFNλ – al que se hace referencia en el punto 3 – como los del AFD – del punto 4 – se le asocian, durante el proceso de construccion de dichos AF, el número de la expresión regular con la que se corresponden.

- ✓ Si se desea mejorar la performance del analizador léxico, se puede minimizar el número de estados utilizando en algoritmo dado en la sección 1.4.5.

1.6 – Diseño de un Generador de Analizadores Léxicos

En esta sección se presentan los puntos más relevantes del diseño de una herramienta que construye automáticamente analizadores léxicos a partir de una especificación de expresiones regulares.

Los pasos para la construcción de dicha herramienta son los siguientes:

1. Definir el lenguaje de especificación de las expresiones regulares.
2. Implementar un algoritmo que parsee un archivo fuente que respete la especificación dada y construya las estructuras de datos necesarias –árbol sintáctico de las expresiones regulares, entre otras –.
3. Implementar un algoritmo que construya los AFD's de cada expresión regular – ver sección 1.4.4 –.
4. Implementar un algoritmo que construya un AFNλ a partir de los AFD's obtenidos en el paso anterior – ver paso 3 de la sección anterior –.
5. Implementar un algoritmo que construya un AFD a partir del AFNλ – ver sección 1.4.3 –.
6. Minimizar el AFD – ver sección 1.4.5 –.
7. Generar un archivo de código fuente con las estructuras y los algoritmos genéricos necesarios para simular el AFD.

Capítulo 2

Introducción a las Expresiones Regulares Traductoras

Generalmente las herramientas para generar analizadores léxicos permiten definir la sintaxis de los símbolos mediante expresiones regulares. Posteriormente sus atributos deben ser computados analizando la cadena reconocida. Así, por ejemplo, si se quisiera definir el símbolo *constante real* y dejar su valor en la variable *valor* habría que dar una definición del siguiente tipo:

(**dígito**)+ **.** (**dígito**)* { valor:= *cálculo_valor*(cadena reconocida) }

Si no se dispone de una función *cálculo_valor* el usuario deberá escribirla, realizando un trabajo muy similar al de reconocer cadenas correspondientes a este token. Este problema aparece frecuentemente.

La idea aquí desarrollada consiste en utilizar para el análisis léxico una sintaxis muy similar a la de los esquemas de traducción usados para los lenguajes independientes del contexto, que se emplean extensamente en los generadores de analizadores sintácticos. Con este estilo, la definición anterior puede escribirse como:

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

donde se supone que:

- Al comenzar el proceso las variables **pe**, **pf**, **n_fra** valen 0 y al finalizar se ejecuta ***fin***
- **dígito** es del tipo 0..9;
- ***cc*** es el caracter corriente;
- las acciones se ejecutan después de haber reconocido cada caracter y responden al pseudo código que se muestra continuación:
  ***acum_pentera*:** pe = pe*10 + val(cc)
  ***acum_pfracc*:** pf = pf *10 + val(cc); n_dfra := n_dfra+1
  ***fin*:** valor := pe + pf / (10 ^ n_dfra)

Esto sugiere una extensión de las expresiones regulares, en las que cada átomo es un par, integrado por un caracter de entrada y una acción. Generalizando, en vez de acciones pueden considerarse símbolos de un alfabeto de salida, convirtiéndose entonces el problema en el estudio de la extensión de las expresiones regulares para expresar traducciones.

Se definen las Expresiones Regulares Traductoras (**ET**) que responden a lo dicho en el párrafo anterior, luego se introduce una subclase, las Expresiones Regulares con Traducción Unica (**ETU**) y una subclase de estas últimas las Expresiones Regulares Traductoras Lineales (**ETL**). Por último se define una nueva subclase de Expresiones Regulares Traductoras Lineales: las Expresiones Traductoras Lineales Cerradas (**ETC**).

La introducción de las **ETU** responde a que interesa que una cadena tenga una sola traducción y no siempre sucede así para las **ET**, por ejemplo, la **ET**: **0*A* | 1*B* | 0*C*** genera, para la cadena **101**, cualquiera de las siguientes traducciones ***BAB*** o ***BCB***.

La introducción de las **ETL** responde a razones de eficiencia de los algoritmos de traducción, esta clase está caracterizada por el hecho de que, traduciendo la cadena de entrada de izquierda a derecha, cada traducción está determinada unívocamente por el prefijo ya reconocido. Es decir, 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 tienen asociada una sola secuencia de acciones, pero no es una ETL porque tanto A_1 como A_2 corresponden al último caracter del mismo prefijo, el prefijo "0". Por esto se afirma que la traducción de dichas acciones es lineal – con lo que se evita cualquier necesidad de back-tracking.

Finalmente se muestra como puede construirse un generador de analizadores léxicos basado en **ETL'**s, los próximos capítulos tratan sobre una implementación concreta sobre Java.

2.1 – Definiciones

2.1.1 - Traducción (»)

Dados dos alfabetos finitos $\sum$ y Γ, se llamará traducción a una relación

$$» : \sum^* \rightarrow \Gamma^*$$

2.1.2 - Expresión Regular Traductora (ET)

Dados $\sum$ un alfabeto de entrada y Γ un alfabeto de salida, se define ET a una expresión regular sobre el alfabeto $\sum \times \Gamma$. Posteriormente se definirán traducciones asociadas a cada uno de los formalismos introducidos.

Se definen ahora las siguientes funciones sobrecargadas:

2.1.3 - Proyección sobre la Primera Coordenada (Π1)

Π1 : Expresiones Regulares sobre $(\sum \text{ x } \Gamma) \rightarrow \sum^*$ tal que :

$\Pi1 (\varnothing) = \varnothing$,
$\Pi1 (\lambda) = \lambda$,
$\Pi1((a, A)) = a$,
$\Pi1 (e \mid e') = \Pi1(e) \mid \Pi1(e')$,
$\Pi1 (e . e') = \Pi1(e) . \Pi1(e')$,
$\Pi1(e^*) = (\Pi1(e))^*$

Π1 : Expresiones Regulares sobre $(\sum \times \Gamma)^* \rightarrow \sum^*$ tal que :
$\Pi1 ((a, A) \gamma) = a \Pi1(\gamma)$.

2.1.4 - Proyección sobre la Segunda Coordenada (Π2)

Π2 : Expresiones Regulares sobre $(\sum \times \Gamma) \rightarrow \sum^*$ y $(\sum \times \Gamma)^* \rightarrow \sum^*$ se definen análogamente Π1.

2.1.5 - Traducción Generada por una ET:

Dada e: ET,
$T(e) = \gg_e / \alpha \gg_e \beta \Leftrightarrow \exists \gamma \in L(e): \alpha = \Pi1(\gamma) \wedge \beta = \Pi2(\gamma)$

2.1.6 - Expresión Regular con Traducción Única (ETU)

Sea e: ET se dice que
e es una ETU $\Leftrightarrow (\alpha \gg_e \beta \wedge \alpha \gg_e \gamma \Rightarrow \beta = \gamma)$

2.1.7 - Expresión Regular Traductora Lineal (ETL)

Sea e: ET,
e es una ETL $\Leftrightarrow$ se verifica que $\forall \alpha, \alpha' \in (\sum \times \Gamma)^* \ \forall a \in \sum \ \forall A, A' \in \Gamma$
$[(\alpha (a,A) \in$ prefijos(e) $\wedge \alpha' (a,A') \in$ prefijos(e) y $\Pi1(\alpha) = \Pi1(\alpha')) \Rightarrow A=A']$ □

Ejemplos de expresiones regulares traductoras lineales:

Caso 1: e1 = 0 A 1 B | 0 A 2 C
e2 = 0 D 3 E

Caso 2: e1 = 0 A 1 B | 0 A 2 C
e2 = 0 A 3 D

Donde los números son los símbolos del lenguaje y las letras mayúsculas representan las traducciones.

En el caso 1 la expresión e1 es una ETL y la expresión e2 también es una ETL. Si se unifican las dos expresiones regulares traductoras lineales (e1,e2) para formar una nueva expresión regular que reconozca el mismo lenguaje, se obtiene la siguiente expresión:

0 A 1 B | 0 A 2 C | 0 D 3 E

Observe que la nueva expresión regular traductora ya no cumple la definición de ETL porque la acción **D** para el prefijo **0** de **0 D 3 E** es distinta a la acción **A** de **0 A 1 B** por lo tanto esta expresión traductora no es lineal.

En cambio, en el caso 2 e1: ETL y e2: ETL al unirlas a través del operador | se forma la siguiente expresión traductora:

0 A 1 B | 0 A 2 C | 0 A 3 D

En este caso la nueva expresión traductora sí cumple la definición de expresión traductora lineal.

2.1.8 - Autómata Finito Traductor (AFT)

A los autómatas finitos traductores además de rotular los arcos con caracteres del alfabeto, se los puede rotular con acciones, de modo que a medida que se reconoce una cadena se ejecutaran dichas acciones. Como convención el nombre de las acciones se escribe debajo del arco.

$M = < K, \sum, \Gamma, \delta , \tau, , q_0 , F >$
donde:

1) K es el conjunto de estados o nodos.
2) $\sum$ un conjunto de símbolos de entrada.
3) δ una función de transición que transforma pares (estado ,símbolo) en un estado.
   δ: K x $\sum \rightarrow$K
4) τ una función de transición que transforma pares (estado ,símbolo) en una acción.
   τ: K x $\sum \rightarrow \Gamma$
5) q_0 estado inicial, $q_0 \in$K.
6) F conjunto de estados finales o estados de aceptación, F $\subseteq$ K.

Configuración:
K x $\sum$* x Γ*

Transición de configuraciones:
$(q, a\ \alpha, \omega) \vdash (r, \alpha, \omega\ t)$ si $r=\delta(q,a) \wedge t \in \tau\ (q,a)$

Traducción generada por un AFT:
dado M: AFT,
se dirá que $\alpha \Rightarrow_M \beta$ si $\exists$ f $\in$ F tal que $(q_{0,} \alpha, \lambda) \vdash^*_M (f, \lambda, \beta)$

Los autómatas que aquí llamamos AFT corresponden a la Máquina de Mealy con el agregado del conjunto de estados finales para poder utilizar el concepto de aceptación.

Veamos un ejemplo de un autómata traductor que acepte el lenguaje de los números enteros. En la figura 3.1 se puede observar el AFD y en la figura 3.2 el autómata traductor.

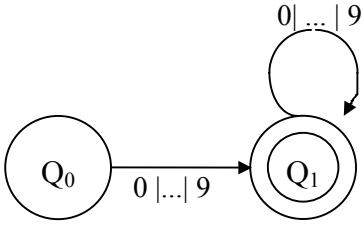


Figura 2.1: AFD que reconoce números naturales.

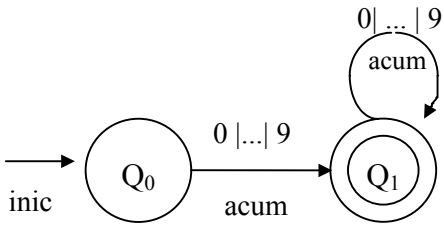


Figura 2.2: AFT que reconoce números naturales y calcula su valor.
Donde: inic: pe=0 y acum: pe = pe*10 + val(cc).

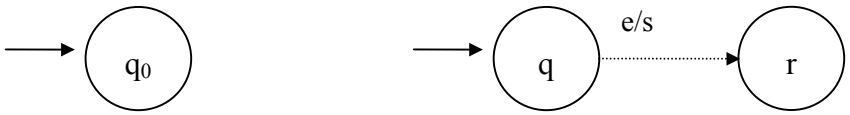
2.1.9 – Máquina de Mealy

Una máquina de Mealy [Mor00] es una estructura de la forma:

$M = < K, Ent, Sal, Tran, Resp, , q_0 >$
donde:

1) K es el conjunto de estados.
2) Ent es el alfabeto de entrada.
3) Sal es el alfabeto de salida.
4) Tran: K x Ent $\rightarrow$ K es la función de transición que transforma pares (estado ,entrada) en un estado.
5) Resp: K x Ent $\rightarrow$ Sal es la función de respuesta.
6) q_0 estado inicial, $q_0 \in K$.

La semántica procedimental de la máquina de Mealy es la siguiente: al inicio de cualquier computación la máquina se encuentra en el estado q_0 . Posteriormente, cuando la máquina se encuentra en un estado $q \in K$ y recibe un símbolo e perteneciente al alfabeto de entrada Ent, entonces emite un símbolo de salida s = Resp(q,e) y transita al nuevo estado r = Tran(q,e). Gráficamente se representa de la siguiente manera:



a) q_0 es el estado inicial

b) Si se está en el estado q y llega el símbolo e entonces se emite la salida s=Resp(q,e) y se transita al estado r =Tran(q,e)

Figura 2.3: Representación gráfica de la máquina de Mealy.

La máquina de Mealy se puede utilizar para resolver una amplia gama de problemas del mundo cotidiano. Un ejemplo sencillo de la utilización de la maquina de Mealy es el siguiente: Si $n \in N$ entonces $n^1 = 1^{(n)}$ es la representación unitaria de n. Se puede utilizar una máquina de Mealy que calcula el residuo módulo 4 de una cadena de 1's cuando se ve a esa cadena como la representación unaria de un número no negativo.
Otro problema de mayor complejidad es el problema de la máquina expendedora de gaseosas que tienen un costo de $ 4 cada una. A continuación se presentan las máquinas de Mealy para dichos ejemplos.

Ejemplo: Residuos Módulo 4

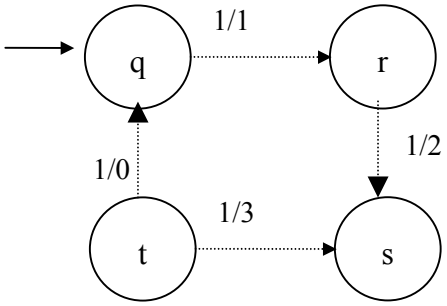


Figura 2.4:Máquina de Mealy para el ejemplo residuos módulo 4.

La máquina es M = <{q,r,s,t} {1} {0,1,2,3} Tran, Resp, q> en donde las funciones de Tran y Resp son las siguientes:

Tran	**1**		**Resp**	**1**
q	r		q	1
r	s		r	2
s	t		s	3
t	q		t	0

Ejemplo: Máquina expendedora de gaseosas

Se deben tener en cuenta las siguientes suposiciones para representar el problema de la máquina expendedora de gaseosas:

- el costo de las gaseosas debe cubrirse utilizando monedas de 1, 2, 5 y 10 pesos.
- La máquina sólo da cambio en monedas de un peso, las cuales están ubicadas en una alcancía. Si no se puede dar cambio, es decir, no hay suficientes monedas en la alcancía, se devuelve la moneda sin expender ninguna gaseosa.
- Sólo se pueden ingresar monedas en el orden inverso a su denominación.

La máquina de Mealy que modela el funcionamiento de la máquina expendedora de gaseosas tiene como alfabeto de entrada el producto cartesiano del conjunto de monedas aceptables con el conjunto que codifica a los depósitos en la alcancía. Pues hay 5 x 7 = 35 símbolos de entrada $m_i p_j$. El alfabeto de salida está dado por las cuatro posibles respuestas de la máquina expendedora. Hay 1 + 6 + 2 + 3 = 11 estados.

La máquina es M = <{q_0, a_0, a_1, a_2, a_3, a_4, a_5, b_1, b_2, c_1, c_2, c_3} { m_0, m_1, m_2, m_5, m_{10}} {s_0, s_1, s_2, s_3} Tran, Resp, q_0> en donde la codificación de los conjuntos de estados, entrada y salida es la siguiente:

- *Estados de la máquina:*
  q_0: Estado inicial
  a_i, $\forall$ i $\in$[0,5]: resta devolver i pesos.
  b_i, $\forall$ i $\in$[1,2]: falta pagar i pesos cuando se inicio el pago con \$2.
  c_i, $\forall$ i $\in$[1,3]: falta pagar i pesos cuando se inicio el pago con \$1.

- *Alfabeto de Entrada:*
  m_0 : ninguna moneda se inserta.
  m_1: moneda de un peso.
  m_2: moneda de dos pesos.
  m_5: moneda de cinco pesos.
  m_{10}: moneda de diez pesos.

- *Alfabeto de Salida (respuesta de la máquina):*
  s_0 : continúa sin hacer nada.
  s_1: entrega una gaseosa.
  s_2: entrega un peso de vuelto.
  s_3 : devuelve la moneda ingresada.

- *Depósito en la Alcancía*
  p_i, $\forall$ i $\in$[0,5]: No alcanza ha haber i pesos.
  p_7: Al menos hay 6 pesos

- *Las funciones Tran y Resp:*

Funciones Tran y Resp	**Explicación**
$\forall$ i $\leq$ 6: Tran (q_0, $m_{10}p_i$) = q_0 Resp(q_0, $m_{10}p_i$) = s_3	Si se inserta una moneda de \$10 y no hay cambio suficiente, se devuelve la moneda y se reinicia el proceso.
Tran (q_0, $m_{10}p_7$) = a_5 Resp(q_0, $m_{10}p_7$) = s_2	Ya que hay cambio precédase a dar dicho cambio.
$\forall$ i $\in$ [1-5] Tran (a_i, m_0P) = a_{k-1} Resp(a_i, m_0P) = s_2	Para P = p_j cualquiera que sea j, continúese devolviéndose un peso hasta completar el cambio. Obsérvese que aquí, en principio, puede haber combinaciones (a_i, p_j) contradictorias. Sin embargo, la interpretación que se está construyendo excluye que aparezcan estas inconsistencias.
Tran (a_0, m_0P) = q_0 Resp(a_0, m_0P) = s_1	Al terminar de dar el cambio, se entrega la gaseosa y se reinicia el proceso.
Tran (q_0, m_5p_1) = q_0 Resp(q_0, m_5p_1) = s_3	Si se inserta una moneda de \$5 y no hay cambio, se devuelve la moneda y se reinicia el proceso.
Tran (q_0, m_5P) = a_0 Resp(q_0, m_5P) = s_2	Si hay monedas en la alcancía, es decir P $\neq$ p_1 entonces se da el peso de cambio.
Tran (q_0, m_2P) = b_2 Resp(q_0, m_2P) = s_0	Se insertan \$2 y se espera a completar el importe de \$4.
Tran (b_2, m_2P) = q_0 Resp(b_2, m_2P) = s_1	Habiéndose completado el costo de la gaseosa se la entrega y se reinicia el proceso.
Tran (b_2, m_1P) = c_1 Resp(b_2, m_1P) = s_0	Se inserta un peso mas y hay que esperar que llegue el último.
Tran (b_2, MP) = b_2 Resp(b_2, MP) = s_3	Si llega una moneda con denominación mayor M = m_5 m_{10} entonces se la devuelve y se continua la espera
Tran (q_0, m_1P) = c_3 Resp(q_0, m_1P) = s_0	Si se inicia el pago con una moneda de un peso hay que esperar los otros tres pagos.
$\forall$ i = 1,2,3: Tran (c_i, m_1P) = c_{i-1} Resp(c_i, m_1P) = s_0	Se continua el pago, recibiendo un peso a la vez. Aquí c_0 = a_0 si se recibe monedas de mayor denominación se las devuelve.

	Cualquier otra posibilidad (Estado,Entrada) es inconsistente e inalcanzable en la máquina.

2.1.10 – Máquina de Moore

Una máquina de Moore [Mor00] es similar a una de Mealy, salvo que la respuesta sólo depende del estado actual de la máquina y es independiente de la entrada. Precisamente una máquina de Moore es una estructura de la forma:

$M = < K, Ent, Sal, Tran, Resp, , q_0 >$

donde:

1) K es el conjunto de estados.
2) Ent es el alfabeto de entrada.
3) Sal es el alfabeto de salida.
4) Tran: K x Ent $\rightarrow$ K es la función de transición que transforma pares (estado ,entrada) en un estado.
5) Resp: K $\rightarrow$ Sal es la función de respuesta.
6) q_0 estado inicial, $q_0 \in K$.

La semántica procedimental de la máquina de Moore es la siguiente: al inicio de cualquier computación la máquina se encuentra en el estado q_0. Posteriormente, cuando la máquina se encuentra en un estado $q \in K$ y recibe un símbolo e perteneciente al alfabeto de entrada Ent, entonces transita al nuevo estado r = Tran(q,e) y emite el símbolo de salida s = Resp(q,e).

Ejemplo: Supongamos que se da un número $n \in N$ en su representación binaria y se quiere calcular su residuo módulo 3. La figura 2.5 muestra la máquina de Moore para dicho ejemplo.

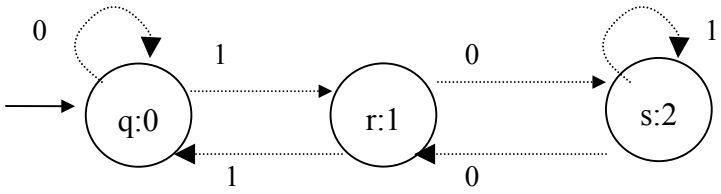


Figura 2.5: Ejemplo de una Máquina de Moore.

Las funciones de transición y de respuesta quedan definidas de la siguiente manera:

Tran	0	1		Resp	
q	q	R		Q	0
r	s	Q		r	1
s	s	S		s	2

2.2 - Propiedades de las expresiones traductoras

Para presentar las propiedades de las expresiones traductoras denotaremos :

T(ET) = { T(A) / A: ET }
T(ETU) = { T(A) / A: ETU }
T(ETL) = { T(A) / A: ETL }
T(AFT) = { T(A) / A: AFT }

El siguiente diagrama resume la propiedades de las expresiones traductoras, las cuales se demuestran a continuación.

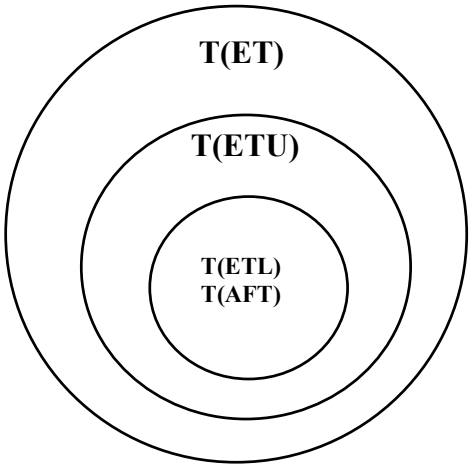


Figura 2.6: Propiedades de las Expresiones Regulares Traductoras.

Teorema 1: $\{ T(e) / e:ET \} \supset \{ T(e) / e:ETU \} \supset \{ T(e) / e:ETL \}$

Demostración: Por definición , la clase de las ETU es una subclase de las ET, por lo que $\{ T(e) / e:ET \} \supseteq \{ T(e) / e:ETU \}$. Pero como e=aA/aB es una ET que no define una traducción única, la inclusión anterior debe ser estricta.

Sea ahora e:ETL. Si e no define una traducción única:
$\exists\ \alpha= \alpha_1\alpha_2...\alpha_n \in \Sigma^*$, $\exists\ \beta= \beta_1\beta_2...\beta_n$ y $\gamma= \gamma_1\gamma_2...\ \gamma_n \in \Gamma^*$ con $\beta \neq \gamma$ tales que
$(\alpha_1\beta_1)\ (\alpha_2\beta_2)...\ (\alpha_n\beta_n) \in L(e)$ y $(\alpha_1\gamma_1)\ (\alpha_2\gamma_2)\ ...\ (\alpha_n\gamma_n) \in L(e)$.

Sean $\beta_j \neq \gamma_j$ con $\beta_i = \gamma_i\ \ \forall\ 0 \leq i < j$. Con estas hipótesis, se cumple que:

- $(\alpha_1\beta_1)...\ (\alpha_j\beta_j) \in$ prefijos(e) y $(\alpha_1\gamma_1)\ ...\ (\alpha_j\gamma_j) \in$ prefijos(e)
- $\Pi1((\alpha_1\beta_1)...\ (\alpha_{j-1}\beta_{j-1})) = \alpha_1...\alpha_{j-1} = \Pi1((\alpha_1\gamma_1)\ ...\ (\alpha_j\gamma_{j-1}))$

Y por definición de ETL, debe ser $\beta_j = \gamma_j$. El absurdo proviene de suponer que la traducción no era única, quedando demostrado que:
$\{ T(e) / e:ETU \} \supseteq \{ T(e) / e:ETL \}$.

Sea e= (d *{conv_octal}*)*.O*{NULA}* / (d *{conv_binaria}*)*.B*{NULA}*. Esta ET recibe como entrada cadenas de dígitos finalizadas en O o B, indicando si el número dado debe interpretarse o traducirse como número octal o binario, por lo que resulta claro que se trata de una ETU. Sin embargo, la expresión dada no es una ETU ya que es necesario conocer el último símbolo de la cadena para decidir qué traducción debe aplicarse sobre los

símbolos anteriores. Por lo tanto, queda demostrado que { *T(e) / e:ETU* } ⊃ { *T(e) / e:ETL* }.□

2.3 - Diseño un Generador de Analizadores Léxicos para Expresiones Traductores Lineales

En esta sección se presenta el diseño de una herramienta de software que construye automáticamente un analizador léxico a partir de una especificación escrita en un lenguaje similar al utilizado por *Lex*, con las modificaciones necesarias para implementar el nuevo formalismo propuesto en este capítulo. En cambio de utilizar expresiones regulares – como en *Lex* – se utilizan expresiones regulares traductoras lineales. La especificación tiene la siguiente forma:

E_1 { acción final $_1$ }
E_2 { acción final $_2$ }
... ...
E_n { acción final $_n$ }

donde cada patrón Ei es una expresión regular traductora lineal y cada acción *acción final i* es un fragmento de un programa que debe ejecutarse siempre que se encuentre en la entrada de un lexema de concuerde con Ei.

El problema consiste en construir un reconocedor que busque lexemas en el buffer de entrada. Si concuerda más de un patrón, el reconocedor elegirá el lexema más largo que haya encontrado. Si existen más de un patrón que concuerdan con el lexema más largo, se elige el primer patrón que haya concordado en la lista.

Para construir esta herramienta se realizan los siguientes pasos:

1- Construcción del árbol sintáctico de cada ETL.
2- Construcción del Autómata Finito Traductor (AFT) para cada ETL de la especificación.
3- Construcción de un AFNλ que acepte la unión de los lenguajes correspondientes a los autómatas obtenidos en el punto anterior.
4- Construcción de un AFD a partir del AFNλ.
5- Cálculo de los estados finales.
6- Simulación de AFD para encontrar el token correspondiente al lexema de mayor longitud.

A continuación se describe detalladamente la realización de cada paso:

Paso 1: Construcción del árbol sintáctico

Para cada patrón Ei se construye el árbol sintáctico como se indicó en el capítulo anterior. A cada hoja del árbol se le agrega un nuevo atributo que contiene la acción que se ejecuta al reconocer el símbolo asociado a esa posición del árbol. Si el símbolo no tiene acción este atributo es vacío.

Paso 2: Construcción de un AFT para cada ETL

Para construir el AFT a partir de una ETL se utiliza el algoritmo presentado en [Agu99] el cual es una extensión del algoritmo descrito en el capítulo anterior sección 1.4.4 propuesto por [Aho88]. Este algoritmo construye un AFT a partir de una ETL sin pasar previamente por la construcción de otro no determinístico. Los estados del autómata por él construido serán 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. Se construye el árbol sintáctico de *e#* y se definen sus posiciones.
2. Se computan las funciones firstpos, lastpos y followpos.
3. Se construye el correspondiente autómata **M:AFT** con el siguiente algoritmo, variante del citado [Aho88 - pág.143] (las modificaciones introducidas al algoritmo original aparecen subrayadas):

Algoritmo 1: Construcción de un AFT equivalente a una ETL

Entrada: e: ETL
Salida: M: $< K, \sum, \Gamma, \delta, \iota, Q_0, F >$: AFT
$\{ \neg error \Rightarrow (e:ETL \wedge »_e = »_M) \wedge error \Rightarrow \neg e:ETL \}$

K: Conjunto de conjuntos de posiciones, con la posibilidad de marcar sus elementos
Q, R: elemento de K — i: Posición
A: subconjunto de Γ — carac: Posición $\rightarrow \sum$
error: Bool — acción: Posición $\rightarrow \Gamma$

Q_0 = firstpos(raiz_del_arbol_sintáctico)
K:=$\{Q_0\}$ sin marcar
error:= False
mientras haya $Q \in K$ sin marcar
{ Inv es el invariante usado en la demostración de corrección que se verá }
marcar Q
para cada $a \in \sum$
{ definir $R=\delta(Q,a)$ y $\iota(Q,a)$; agregar R a K si no estaba ya y, si se viola la condición para que e sea ETL, reportarlo }
R:= ∅; A := ∅
para cada $i \in Q$ tal que carac(i) = a hacer
R:= R U followpos(i); A=A U {acción(i)}
definir $\delta(Q, a)=R$
según A = $\{A_0\}$: definir $\tau(Q,a) = A_0$
otro caso : error:= True *{ la acción no es única}*
si $R \notin K$: agregar R a K sin marcar
F:= $\{ Q \in K / \exists i \in Q : carac(i)= \# \}$
Si $\emptyset \in K$ definir $\delta(\emptyset,a) = \emptyset$ y $\tau(\emptyset,a) = NULA$

Paso3: Construcción de un AFNλ a partir de los AFT's

Para construir el AFNλ a partir de los AFT's se aplica el algoritmo presentado por en el paso 3 de la sección 1.5. El algoritmo construye un AFNλ definiendo un nuevo estado inicial y uniendo el estado inicial del autómata nuevo con los estados iniciales de cada autómata traductor.

Paso 4: Construcción del AFD a partir del AFNλ

En este paso se construye el AFD a partir del AFNλ siguiendo el algoritmo que une estados presentado en el capítulo anterior.

Paso 5: Cálculo de los estados finales

Cómputo de la función tok . Para cada estado k del último autómata.

tok(k) = 0 si k no tiene estados finales

tok(k)= mín { j / $q_j \in k \wedge q_j$ es final } en otro caso.

Paso 6: Simulación del AFD

Se implementa un procedimiento lex, que a partir de la cadena de entrada retorna el próximo token y realiza su evaluación (evaluación que resulta de la ejecución sucesiva de las acciones de traducción).

En cada invocación del procedimiento lex:

- Avanza sobre la cadena de entrada emulando al autómata M, hasta que éste se bloquee, llegando al estado ϕ.

- Retrocede hasta el último estado final visitado, ult_estado_final , computa:
  token=tok(ult_estado_final)
  y deja el puntero de la cadena de entrada en el estado en que se encontraba al visitar este último estado final.

- El lexema reconocido es la subcadena comprendida entre las posiciones inicial y final del puntero a la cadena de entrada correspondientes a la última invocación.

- Produce la evaluación del lexema emulando el AFT asociado a e_{token} y finalmente retorna token.

Conclusiones

- ✓ Los algoritmos presentados permiten la construcción de un generador de analizadores léxicos traductores que produce reconocedores-traductores que trabajan en tiempo O(|α|) - donde α es la cadena de entrada.

- ✓ El tiempo de generación de los analizadores léxicos traductores es del mismo orden que el del algoritmo presentado en [Aho88].

- ✓ Al utilizar ETL's para definir el analizador léxico, es más sencillo para el usuario, realizar la especificación de los analizadores ya que define la sintáxis y la semántica de los símbolos del lenguaje conjuntamente.

2.5 - Corrección de los Algoritmos

2.5.1 Corrección de las funciones auxiliares

Lema 1

El cómputo de las funciones auxiliares dado en el capítulo 1 sección 1.4.4 es correcto. O sea:

Dada *e* expresión regular; i, j posiciones de su árbol sintáctico:

a) i ∈ firstpos(*e)* ⇔ ∃ a.. ∈ $L(e)$ con a generada por i.
b) i ∈ lastpos(*e*) ⇔ ∃ ..a ∈ $L(e)$ con a generada por i.
c) j∈followpos(*i*) en *e* ⇔ ∃ ..ab.. ∈ $L(e)$ con a generada por i y *b* por j.

Lema 1- a)

Dada una expresión regular *e*; *i* posición*:*
i ∈ firstpos(*e)* ⇔ ∃ ..a ∈ $L(e)$ con a generada por i.

Demostración: Se utiliza inducción estructural, sobre *e*.

Casos base:

- ***e* = λ:** Como, por definición de árbol sintáctico- tipo AER -,
  ¬∃ i / carac(i) = λ ⇒ la tesis se cumple trivialmente.
- ***e* = a:** En este caso, $L(e)$ = a y, además, el árbol sintáctico de *e* tiene una única posición que genera a ⇒ se cumple la tesis.

Paso inductivo:

- ***e* = $e_1 . e_2$:**

Hipótesis inductiva: Dadas e_1, e_2: ER, i_1 posición del árbol sintáctico de e_1 , i_2 posición del árbol sintáctico de e_2:

i_1 ∈ firstpos(e_1) ⇔ ∃ a.. ∈ $L(e_1)$ en la que a es generada por i_1 ∧
i_2 ∈ firstpos(e_2) ⇔ ∃ a.. ∈ $L(e_2)$ en la que a es generada por i_2.

Tesis: Dada i posición del árbol sintáctico de $e_1.e_2$:

i ∈ firstpos($e_1 . e_2$) ⇔ ∃ a.. ∈ $L(e_1 . e_2)$ con a generada por i.

Demostración e = $e_1 . e_2$:

Primer caso: ¬ nullable(e_1)
⇒) ¬ nullable(e_1) ⇒ firstpos($e_1 . e_2$) = firstpos(e_1).
Luego, i ∈ firstpos($e_1 . e_2$) ⇒ i ∈ firstpos(e_1) ⇒ ∃ a.. ∈ $L(e_1)$ con a generada por i posición del subárbol correspondiente a e_1. Entonces, a.. ∈ ∈ $L(e_1.e_2)$ con a generada por i posición del árbol de $e_1.e_2$.

⇐) ∃ a.. ∈ $L(e_1 .e_2)$ con a generada por i posición del árbol de $e_1 .e_2$. Entonces, como ¬nullable(e_1), ∃ a.. ∈ $L(e_1)$ con a generada por i posición del árbol de e_1. Luego, por hipótesis inductiva, i ∈ firstpos(e_1), entonces i ∈ firstpos($e_1\ e_2$).

Segundo caso: nullable(e_1)
⇒) nullable(e_1) ⇒ firstpos($e_1.e_2$) = firstpos(e_1) U firstpos(e_2). (3)
Además, como λ ∈ $L(e_1)$, ∀ α = a.. ∈ $L(e_1.e_2)$:
(∃ β ∈ $L(e_1)$, β ≠ λ / α = β..) ∨ (∃ γ ∈ $L(e_2)$, γ ≠ λ / α = γ..)
Si ∃ β ∈ $L(e_1)$, β ≠ λ / α = β.. entonces, como β ∈ $L(e_1)$, sus símbolos son generados por posiciones del subárbol sintáctico de e_1 ⇒ ∃ a.. ∈ $L(e_1.e_2)$ con a generada por i posición del subárbol de e_1 . Luego, por construcción del árbol sintáctico de $e_1.e_2$, ∃ a.. ∈ $L(e_1.e_2)$ con a generada por i posición del árbol de $e_1.e_2$.
Análogamente se demuestra que si ∃ γ ∈ $L(e_2)$, γ ≠ λ / α = γ.. ⇒ ∃ a.. ∈ $L(e_1.e_2)$ con a generada por i posición del árbol de $e_1.e_2$.
Por lo tanto, siendo i posición del árbol sintáctico de $e_1.e_2$:
i ∈ firstpos($e_1.e_2$) ⇒ ∃ a.. ∈ $L(e_1.e_2)$ con a generada por i.
⇐) Si a.. ∈ ∈ $L(e_1.e_2)$ con a generada por i posición del árbol de $e_1.e_2$ ⇒ por construcción del árbol sintáctico de $e_1.e_2$:
(∃ a.. ∈ $L(e_1)$ con a generada por i posición del árbol sintáctico de e_1) ∨
(∃ a.. ∈ $L(e_2)$ con a generada por i posición del árbol sintáctico de e_2) ⇒
por hipótesis inductiva, i ∈ firstpos(e_1) ∨ i ∈ firstpos(e_2) ⇒ i ∈ (firstpos(e_1) U firstpos(e_2)) ⇒ por defínción de firstpos, i ∈ firstpos($e_1.e_2$).

- ***e* = e_1 | e_2:**

Hipótesis inductiva: Dadas e_1, e_2: ER, i_1 posición del árbol sintáctico de e_1 , i_2 posición del árbol sintáctico de e_2:

i_1 ∈ firstpos(e_1) ⇔ ∃ a.. ∈ $L(e_1)$ con a generada por i_1 ∧
i_2 ∈ firstpos(e_2) ⇔ ∃ a.. ∈ $L(e_2)$ con a generada por i_2.

Tesis: Dadas e_1, e_2: ER, i posición del árbol sintáctico de $e_1|e_2$:

i ∈ firstpos($e_1|e_2$) ⇔ ∃ a.. ∈ $L(e_1|e_2)$ con a generada por i.

Demostración e = e_1 | e_2:

i ∈ firstpos($e_1|e_2$) si y sólo si, por cómputo de firstpos, i ∈ (firstpos (e_1) U firstpos(e_2)) ⇔
i ∈ firstpos (e_1) ∨ i ∈ firstpos(e_2).
Pero, por hipótesis inductiva, lo anterior equivale a
(∃ a.. ∈ $L(e_1)$ con a generada por i posición del árbol sintáctico de e_1) ∨
(∃ a.. ∈ $L(e_2)$ con a generada por i posición del árbol sintáctico de e_2) ⇔
∃ a.. ∈ ($L(e_1)$ U $L(e_2)$) con a generada por i / i es posición del subárbol sintáctico de e_1 ∨ i es posición del subárbol sintáctico de e_2 .Y como la construcción del árbol sintáctico de $e_1|e_2$ no agrega nuevas posiciones, ∃ a.. ∈ ($L(e_1|e_2)$) con a generada por i posición del árbol sintáctico de $e_1|e_2$.

- ***e* = (e_1)*:**

Hipótesis inductiva: Dada e_1: ER, i_1 posición del árbol sintáctico de e_1:

i_1 ∈ firstpos(e_1) ⇔ ∃ a.. ∈ $L(e_1)$ con a generada por i_1

Tesis: Dada i posición del árbol sintáctico de (e_1)* :

i ∈ firstpos((e_1)*) ⇔ ∃ a.. ∈ $L(($e_1$)*)$ con a generada por i.

Demostración e = (e_1):*

La construcción del árbol sintáctico de *e** no agrega hojas al árbol de *e* $\Rightarrow$ las posiciones del árbol de e_1 son las mismas que las posiciones del árbol de (e_1)*. Además, $L((e_1)^*) = L((e_1)^*).\ L(e_1) \cup \{\lambda\}$.
Por lo tanto, $\exists$ a.. $\in L((e_1)^*)$ con a generada por i posición del árbol sintáctico de $(e_1)^*$ $\Leftrightarrow$
$\exists$ a.. $\in L(e_1)$ con a generada por i posición del árbol sintáctico de e_1 . Si y sólo si, por hipótesis inductiva, i $\in$ firstpos(e_1) $\Leftrightarrow$ i $\in$ firstpos($(e_1)^*$), por definición de firstpos.
□

Lema 1-b)

Dada *e*:ER, i posición del árbol sintáctico de *e*:
i $\in$ lastpos(*e*) $\Leftrightarrow \exists$ ..a $\in L(e)$ con a generada por i.

Demostración:
Su demostración es análoga a la anterior.

Lema 1- c)

Dada una expresión regular *e*; i,j posiciones de su árbol sintáctico:
j$\in$followpos(i) en *e* $\Leftrightarrow \exists$ ..ab.. $\in L(e)$ con a generada por i y b por j.

Demostración: Se utiliza inducción estructural, sobre *e*.

Caso base:
Si *e* = a, no existe ninguna cadena de salida de longitud dos derivada de *e,* por lo que followpos (*e*) = $\varnothing$ (no existe regla de cálculo).

Paso inductivo:

- Para $e = e_1.e_2$, si ambas posiciones pertenecen a la misma subexpresión, se cumple la tesis por hipótesis inductiva. En caso contrario, i es hoja de e_1 y j es hoja de e_2. Por lo tanto, como followpos es una función cerrada sobre las posiciones de una ER, j $\notin$ followpos(i) en e_1 y j $\notin$ followpos(i) en e_2, pero j $\in$ followpos(i) en *e* $\Rightarrow$ j fue agregada a followpos(i) por la concatenación de e_1 con $e_2 \Rightarrow$ $\exists\ \alpha \in L(e_1) / \alpha \in$ lastpos(e_1) $\wedge$ $\exists\ \beta \in L(e_2) / \beta \in$ firstpos(e_2) $\Rightarrow \exists\ \alpha \in L(e_1) / \alpha =$ ..a $\wedge$ $\exists\ \beta \in L(e_2) / \beta =$ b $\Rightarrow \exists\ \alpha.\beta \in L(e_1.e_2)$ por definición de conjunto denotado por la concatenación y $\alpha.\beta$ =...ab...
- Para $e = e_1 \mid e_2$, si ambas posiciones pertenecen a la misma subexpresión, se cumple la tesis por hipótesis inductiva. En caso contrario, como i y j pertenecen a distintas subexpresiones, por un razonamiento análogo al anterior j debería haber sido agregada a followpos(i) al tratarse la disyunción de e_1 con e_2 , pero esto es absurdo pues followpos no agrega elementos en un nodo '|' (no existe regla de cálculo).
  Por otra parte, tampoco puede existir una cadena ..ab.. en la que a sea generada a partir de i y b a partir de j, siendo i y j posiciones de distintas subexpresiones , ya que toda cadena de e es generada exclusivamente por posiciones de una sola de las dos subexpresions, e_1 y e_2.

Para $e = e_1*$ se demostrarán las dos implicaciones separadamente.
$\Rightarrow$) Si j $\in$ followpos(i) en e_1, se cumple la tesis por hipótesis inductiva. Por el contrario si j $\notin$ followpos(i) en e_1 y j $\in$ followpos(i) en e es porque j fue agregado al followpos(i) por la operación de clausura cosa que, por la regla de cálculo de followpos para nodos '*', implica que i $\in$ lastpos(e_1) y j $\in$ firstpos(e_1). Entonces $\exists$ α = ...a $\in L(e_1) \wedge \exists$ β = b... $\in L(e_1)$ lo que implica que $\alpha\beta \in L(e_1*)$ quedando demostrado que $\exists$ $\alpha.\beta \in L(e_1*)$ con $\alpha.\beta$ =...ab...

$\Leftarrow$) Sea ρ=..ab.. $\in L(e_1*) \Rightarrow \exists$ ρ1,..,ρn $\in L(e_1)$ para n>=1: $\rho=\rho 1..\rho n$. Si n=1, la tesis se cumple por hipótesis inductiva.
Si n>1 y ρk=..ab.. para algún k, también se cumple la tesis por hipótesis inductiva.
Si no, $\exists$ k : ..a=ρk $\wedge$ b..=ρk+1 $\Rightarrow$ y $\in$ lastpos(e_1) Λ j $\in$ lastpos(e_1) $\Rightarrow$ j $\in$ followpos(i) en e por definición.
Para $e = (e_1)$, como no hay regla de cálculo para followpos en un nodo de este tipo y además L(e) = L(e_1), la propiedad se cumple por hipótesis inductiva.
□

2.5.2 Corrección del algoritmo 1

Teorema 2: El algoritmo 1 es correcto. O sea, termina y al finalizar se cumple:
{ $\neg$ error $\Rightarrow$ (e:ETL $\wedge$ $\gg_e$ = $\gg_M$) $\wedge$ error $\Rightarrow$ $\neg$ e:ETL }

Se demuestra la corrección del algoritmo modificado asumiendo la corrección del algoritmo original. Se denominará ALGo al algoritmo original - parte no subrayada del Algoritmo 1 - y ALGt al algoritmo completo. ALGo construye Mo: AFD y ALGt construye M:AFT , tales que:
Mo=<K, $\sum$, δ, Q_0, F> :AFD y M=< K, $\sum$, Γ,δ, τ, Q_0, F >: AFT
La corrección de Mo garantiza que $L(Mo)=L(\Pi_1(e))$ (Anexo).

Lema 2:

Inv es invariante para el ciclo principal, o sea, se cumple antes del inicio del ciclo y después de cada iteración del mismo.

Inv = (A $\wedge$ ($\neg$ error $\Rightarrow$ (B $\wedge$ C)) $\wedge$ D donde
A : $\forall$Q $\in$ K (Q es accesible)
B : $\forall$Q marcado $\in$ K ($\forall$a $\in \sum$ ($\exists$ t $\in \Gamma$ (
$\forall$i $\in$ Q ((carac(i) = a $\wedge$ acción(i) = t) $\Rightarrow$ τ(Q,a) = t)))
C : $\forall$Q marcado $\in$ K ($\forall$ a $\in$ $\sum$ ((δ(Q,a)= {j: posición/ $\exists$ i$\in$Q:carac(i) = a $\wedge$ j$\in$followpos(i)})))
D : (error $\Leftrightarrow$ $\exists$ Q marcado $\in$ K ($\exists$ i, j $\in$ Q: (carac(i) = carac(j) $\wedge$ acción(i) $\neq$ acción(j))))

Demostración:

a.1) **Inv** se cumple antes de comenzar el ciclo. En efecto, antes de comenzar el ciclo el único estado de K es el inicial, que es accesible, por lo que se satisface A. También se cumple ($\neg$ error $\Rightarrow$ (B $\wedge$ C)) pues se cumplen B y C por no haber elementos marcados, y D también se satisface ya que, además, inicialmente error es False.

a.2) Si se cumple **Inv** y la guarda del ciclo entonces también se cumple **Inv** después de la ejecución del cuerpo del ciclo. En efecto:

- Se cumple A porque si Q no fue agregado en la última iteración, debía estar antes en K y entonces es accesible, y si fue agregado en la última iteración debe verificarse que para algún Q' en K (por tanto accesible) y algún $a \in \Sigma : Q = \delta(Q', a)$, resultando que también Q es accesible.
- El cuerpo del ciclo agrega a lo sumo un estado sin marcar a los elementos de K y para él la construcción garantiza que, si $\neg$error, la traducción de cada símbolo a es única, o sea que debe cumplirse B.
- La construcción de M también garantiza C.
- Finalmente, se cumple D ya que antes de la ejecución del cuerpo del ciclo se verificaba $\neg$error y el único camino que asigna True a error se ejecuta sólo si se da el segundo miembro de D.

□

Lema 3:

El ciclo termina, y al finalizar se cumple la siguiente poscondición:

P: $I \wedge (\neg \text{error} \Rightarrow (II \wedge III)) \wedge IV \wedge V$

donde:

I: $\forall Q \in K$ (Q es accesible)

II: $\forall Q \in K (\forall a \in \Sigma (\exists t \in \Gamma : \forall i \in Q$ (carac(i) = a $\wedge$ acción(i) = t) $\Rightarrow$ $\tau(Q,a) = t)$

III: $\forall Q \in K (\forall a \in \Sigma$ ($\delta(Q,a) = \{ j / \exists i \in Q : carac(i) = a \wedge j \in followpos(i) \}$))

IV: error $\Leftrightarrow \exists Q \in K (\exists i , j \in Q$ (carac(i) = carac (j) $\wedge$ acción(i) $\neq$ acción(j))

V: $Q_0 \in K \wedge Q_0 =$ firstpos (raíz)

Demostración:

Como el control coincide con el de ALGo, la corrección del mismo (Anexo) garantiza la terminación del ciclo.

V est{a garantizado por construcción, y $I \wedge (\neg error \Rightarrow (II \wedge III)) \wedge IV$ es equivalente a la conjunción de la negación de la guarda y el invariante del ciclo principal del algoritmo (Inv).

Primero resulta conveniente establecer el siguiente lema.

□

Lema 4:

Si $\alpha(a,A), \alpha(a,A') \in pref(e) \wedge (Q_0, \Pi_1(\alpha),\lambda) \vdash^* (Q,\lambda,\_) \wedge$ (a, A) se genera por la posición i e la primera cadena y (a, A') por la posición j en la segunda $\Rightarrow$ $i,j \in Q$.

Demostración: por inducción en $|\alpha|$.

Caso base: $|\alpha| = 0$

Como (a,A) y (a,A') son la cabeza de cadenas de $L(e)$, tanto i como j deben pertenecer firstpos(raíz) por lo cual $i, j \in Q_0$, cumpliéndose la tesis.

Etapa inductiva: $|\alpha| = n+1$
Sea $\alpha = \alpha'(b,B)$ ($|\alpha'| = n$)
$(Q_0, \Pi_1(\alpha'(b,B)),\_) \vdash^* (Q,\lambda,\_) \Rightarrow (Q_0, \Pi_1(\alpha'(b,B)),\_) \vdash^* (Q', \Pi_1(b,B),\_) \rightarrow (Q,\lambda,\_)$
Como, por hipótesis, $\alpha'(b,B)(a,A)$ y $\alpha'(b,B)(a,A')$ son prefijos de e deben existir i',j' que generan (b,B) en cada caso y entonces, por hipótesis inductiva, $i',j' \in Q'$. Pero entonces, como $\delta(Q',b)=Q \wedge i \in \text{followpos}(i') \wedge j \in \text{followpos}(j')$, tanto i como j deben pertenecer a Q por III.
□

Lema 5:

Dada e:ETL y la relación entre posiciones de e: $i \rightarrow_f j \Leftrightarrow j \in \text{followpos}(i)$
$\forall i \in \text{pos}(e)\ \exists j \in \text{lastpos}(e) / i \rightarrow_f^* j$

Demostración: Por inducción estructural sobre e.

Casos base:
Si $e = \lambda$, $\text{pos}(e) = \varnothing$ y no hay nada por probar.
Si e = a, el árbol sintáctico de e tiene una única posición. Llamando i a dicha posición, se cumplirá que $i \in \text{lastpos}(e)$, por definición de lastpos, y que $i \rightarrow_f^0 i$.

Etapa inductiva:
Para $e = e_1.e_2$, será $\text{lastpos}(e_2) \subseteq \text{lastpos}(e)$.
Si $i \in \text{pos}(e_2)$, la hipótesis se cumple por lo anterior y por hipótesis inductiva.
Si $i \in \text{pos}(e_1)$ entonces, por hipótesis inductiva, existe $k \in \text{pos}(e_1)$ tal que $i \rightarrow_f^* k \in \text{lastpos}(e_1)$.
Si $\text{nullable}(e_2)$ entonces, como $\text{lastpos}(e_1) \subseteq \text{lastpos}(e)$, $k \in \text{lastpos}(e)$ que es lo que se quería probar.
Si $\neg\ \text{nullable}(e_2)$ entonces, por definición de followpos, existe un $l \in \text{firstpos}(e_2)$ tal que $\forall k \in \text{lastpos}(e_1)$ se cumple que $k \rightarrow_f l$.
Entonces $\forall i \in \text{pos}(e_1)$ será: $i \rightarrow_f^* k \rightarrow_f l \rightarrow_f^* j$.
($i \rightarrow_f^* k$: e_1; $l \rightarrow_f^* j$: e_2)
Luego $\forall i \in \text{pos}(e_1)\ \exists j \in \text{lastpos}(e_2)$ tales que $i \rightarrow_f^* j$.

Para $e = e_1 \mid e_2$, $i \in \text{pos}(e) \Leftrightarrow i \in \text{pos}(e_1) \vee i \in \text{pos}(e_2)$.
Además, por hipótesis inductiva, $\forall i_1 \in \text{pos}(e_1)\ \exists j_1 \in \text{lastpos}(e_1) / i_1 \rightarrow_f^* j_1$
y $\forall i_2 \in \text{pos}(e_2)\ \exists j_2 \in \text{lastpos}(e_2) / i_2 \rightarrow_f^* j_2$. Y como $\text{lastpos}(e) = \text{lastpos}(e_1) \cup \text{lastpos}(e_2)$, la tesis queda demostrada.

Para $e = (e_1)^*$, $\text{lastpos}(e) = \text{lastpos}(e_1)$ y $\text{pos}(e) = \text{pos}(e_1)$, por lo que la tesis se verifica a partir de la validez de la hipótesis inductiva.
□

Lema 6:

Dada e:ETL y M:AFT construido por Algoritmo 1
$\forall Q \in K: Q \neq \varnothing: \exists \alpha \in \Sigma^* / (Q, \alpha, \_) \vdash^* (Q_f, \lambda, \_)$ con $Q_f \in F$.

Demostración:

Sea m la posición de la marca - # - . Como $Q \neq \emptyset$, $\exists$ i posición de e $\in$ Q y como lastpos(e#)=m, por el lema anterior será i $\rightarrow_f$ * m. Por lo cual existe una secuencia de posiciones $i = i_1 .. i_k = m$ / $i_{j+1} \in$ followpos(i_j), $\forall j: 1 \leq j \leq k-1$.
Sea $\alpha = a_1 .. a_k$, con $a_1 .. a_k \in \Sigma$ / a_j = carac(i_j) y $E_1..E_k$ secuencia de estados de K / $E_1 = Q$ y $E_{j+1} = \delta(E_j, a_j)$.
Entonces $\forall j: 1 \leq j \leq k-1$: $i_j \in E_j$ pues $i_1 = i \in Q = E_1$ y si $i_j \in E_j \Rightarrow i_{j+1} \in E_{j+1}$ pues como $i_{j+1} \in$ follwpos(i_j) y carac(i_j)=a_j, por III del lema 4, $i_{j+i} \in \delta(E_j,a_j)=E_{j+1}$. Finalmente, como $m=i_k \in E_k$, E_k es final. □

Lema 7:

Al finalizar el algoritmo se cumple:
$\{ \neg$ error $\Rightarrow$ (e:ETL $\wedge \gg_e = \gg_M$) $\wedge$ error $\Rightarrow \neg$ e:ETL $\}$

Demostración:

d.1) $\neg$error $\Rightarrow$ e es una ETL
Demostración d.1: Si se supone que e no es una ETL $\Rightarrow \exists$ α(a,A), α' (a,A') $\in$ pref(e) / $\Pi_1(\alpha) = \Pi_1(\alpha') \wedge A \neq A'$. Sean las posiciones i y j que generan (a,A) y (a,A') respectivamente. Como se verificará que $\exists Q \in K$ / $(Q_0, \alpha, \lambda) \vdash^* (Q, \lambda, \_)$, por el lema 8 será i, j $\in$ Q y además carac(i)=carac(j) $\wedge$ acción(i) $\neq$ acción(j), lo que conduce a error = True por IV del lema 4. ♦

d. 2) $\neg$ error $\Rightarrow$ ($\gg_e = \gg_M$)
Demostración d.2: Hay que ver que siendo error = False se cumple $\forall \alpha \in \Sigma^*$ y $\forall \beta \in \Gamma^*$
1) $\alpha \gg_e \beta \Rightarrow \alpha \gg_M \beta$
2) $\alpha \gg_M \beta \Rightarrow \alpha \gg_e \beta$

1) $\alpha \gg_e \beta \Rightarrow (\exists \gamma \in L(e) / \Pi_1(\gamma)=\alpha \wedge \Pi_2(\gamma)=\beta)$
de donde $\alpha \in L(Mo)$ por la corrección de ALGo.
Sea $k = |\alpha|$ y $\alpha = a_{1..} a_k$ y $\beta = b_1..b_k$. Se demuestra, por inducción en i, que:
$\forall i: 1 <= i <= k \quad \exists Q \in K$ / $(Q_{0,} a_1 . . a_i, \lambda) \vdash^* (Q, \lambda, b_1 . . b_i)$

Caso base: i=1
$(Q_0, a, \lambda) \vdash (R, \lambda, t) \Rightarrow t = b_1$ por III del lema 4, ya que como head(α)=(a_1,b_1) la posición i que lo genera debe pertenecer a firstpos(raíz) y entonces $i \in Q_0$

Etapa inductiva: i=n+1
$(Q_{0,} a_1 . . a_n a_{n+1}, \lambda) \vdash^* (R, \lambda, \gamma) \Rightarrow \exists \gamma', t, Q$ /
$(Q_{0,} a_1 . . a_n a_{n+1}, \lambda) \vdash^* (Q, a_{n+1}, \gamma') \vdash (R, \lambda, \gamma' t) \wedge \gamma' t = \gamma$
pero por hipótesis inductiva
$(Q_{0,} a_1 . . a_n a_{n+1}, \lambda) \vdash^* (Q, a_{n+1}, b_1 . . b_n) \vdash (R, \lambda, b_1 . . b_n t) \Rightarrow$
$R = \delta(Q,a_{n+1})$ y por III $t = \tau(Q,a_{n+1}) = b_{n+1}$ ♦

2) $\alpha \gg_M \beta \Rightarrow (Q_0, \alpha, \lambda) \vdash_{Mo}^* (Q_f, \lambda, \beta)$ con $\alpha = a_1..a_k \wedge \beta = b_1..b_k$. Como α es aceptada por Mo $\Rightarrow \alpha \in L(Mo) = L(\Pi_1(e)) \Rightarrow$ por corrección de ALGo $\exists \gamma$ / $\Pi_1(\gamma) = \alpha \wedge \gamma \in L(e)$, con $\gamma = (a_1,t_1)..(a_k,t_k)$.
Sean $p_i = (a_i,t_i) \forall 1 \leq i \leq k$. Se demuestra que $b_i = t_i \forall 1 \leq i \leq k$.

i=0) $Q_0 \in K$, como $\tau(Q_0, a_1) \in K \Rightarrow$ (1) $\exists\ i \in Q_0$ / carac(i) = a_1 y acción(i) = b_1. Como $p_1 \in$ firstpos(e), $p_1 \in Q_0 \wedge$ carac(p_1) = a_1 y acción(p_1) = $t_1 \Rightarrow b_1 = t_1$.
i>0) $(Q_0, a_1..a_n a_{n+1}..a_k, \lambda) \vdash_M^* (Q_n, a_{n+1}..a_k, b_1..b_n) \vdash_M (Q_{n+1}, a_{n+2}..a_k, b_1..b_{n+1})$
$Q_0 \in K$. $\tau(Q_n, a_{n+1}) = b_{n+1} \Rightarrow \exists\ i \in Q_n$ / carac(i) = $a_{n+1} \wedge$ acción(i) = b_{n+1}. Por otro lado $p_{n+1} \in$ followpos(p_n) en cualquier cadena del lenguaje. Sea rótulo(p_n) = (a_{n+1}, t_{n+1}) $\Rightarrow p_{n+1} \in \delta(Q_n, a_{n+1}) \wedge \tau(Q_n, a_{n+1}) = t_{n+1}$. ♦

d.3) error $\Rightarrow \neg$ ETL

Demostración d.3: error $\Rightarrow \exists\ Q \in K$, $a \in \Sigma$ / $\tau(Q, a)$ no es única.
Pero como Q es accesible, $\exists\ \alpha$ / $(Q_0, \alpha, \_) \vdash^* (Q, \lambda, \_)$ y, por el lema 8, $\exists\ \alpha'$ / $(Q, \alpha', \_) \vdash^* (Q_f, \lambda, \_) \Rightarrow \alpha \in$ pref(e). Así que $\exists\ \alpha$ / $\alpha(a, A)$ y $\alpha(a, B)$ son prefijos de e y $A \neq B \Rightarrow$ e no es una ETL. ♦
□

Con lo que se completa la demostración de la corrección del algoritmo.

Capítulo 3

Diseño de un Generador de Analizadores Léxicos Traductores

En los primeros capítulos se demostró que las expresiones regulares y los autómatas finitos traductores son un mecanismo lo suficientemente expresivo para definir el análisis léxico de lenguajes. Se propuso, además, una definición teórica precisa de este mecanismo de análisis léxico. En el segundo capítulo se propuso un modelo de implementación alternativo de análisis léxico construido a partir de expresiones regulares traductoras lineales.

Si bien en teoría queda resuelto el problema de construir un analizador léxico traductor, hay que tener en cuenta que en la práctica es bastante difícil hacerlo. Ya que, pasar de la definición de un lenguaje por medio de expresiones regulares al autómata que lo reconozca – si bien es un proceso mecánico – y lo traduzca puede llegar a ser una tarea compleja. Además, una vez construido el analizador, cualquier modificación – como podría ser la inserción de nuevos símbolos en el lenguaje – puede causar la reimplementación de todo el sistema. Hay que resaltar que cada lenguaje requiere la construcción de un analizador léxico que lo reconozca y traduzca.

Por estas razones es que se propone una solución práctica al problema de construir analizadores léxicos traductores. Solución que consiste en proporcionar un generador de analizadores léxicos traductores que automatice la tarea de pasar de la definición conceptual del analizador, en términos de una especificación de expresiones regulares traductoras lineales, al autómata que reconozca y traduzca cadenas. Es decir se imita la solución dada para la generación de analizadores léxicos basado en expresiones regulares.

Los siguientes capítulos tratan sobre el diseño e implementación de un generador de analizadores léxicos traductores basado en el modelo presentado en el capítulo 2. El presente capítulo tiene dos secciones principales. Primero trata sobre el diseño e implementación de los analizadores léxicos traductores que produce el generador. Ya que se considera importante tener bien definidos los analizadores que se crearán como punto de partida del diseño del generador. La segunda parte se refiere al diseño del generador de analizadores léxicos, como así también, sobre las etapas en que fue dividida la implementación del generador. En los capítulos subsiguientes se amplían distintas cuestiones relevantes referidas a la implementación. Cuestiones referidas al modelo que se detallaron en el capítulo 2 para construir un generador de analizadores léxicos traductores.

3.1 - Decisiones de diseño

Java como lenguaje de implementación. Esta elección se basó en que se deseaba que la implementación fuera multi-plataforma, además de utilizar la tecnología orientada a objetos y por último y no menos importante se desea incorporarla al entorno de generación de procesadores de lenguajes *Japlage* – herramienta desarrollada por el grupo de investigación del Departamento de Computación de la FCEFQ de la UNRC [Agu98] [Agu01].

3.2 - Diseño de un Analizador Léxico Traductor

Como primer paso para obtener un generador de analizadores léxicos traductores se realizara el diseño de los analizadores léxicos traductores a generar, el cual, será la primer escalón hacia la implementación del modelo propuesto en el capítulo anterior.

3.2.1 - Clases y relaciones del analizador

La elección de Java como lenguaje de implementación induce a utilizar una notación de diseño orientada a objetos. La notación empleada en esta sección es tomada de [Gam95] - descripta brevemente en el anexo 3.

El generador esta conformado por tres clases y en algunos casos, también, por un grupo de clases - las clases definidas por el usuario que participan de la traducción. Es importante resaltar que el "esqueleto" de las tres primeras clases es común a todos los analizadores sólo varían las estructuras de datos que representan al AFD, AFT's y los métodos relacionados con las traducciones. A continuación se muestra el diseño del analizador léxico.

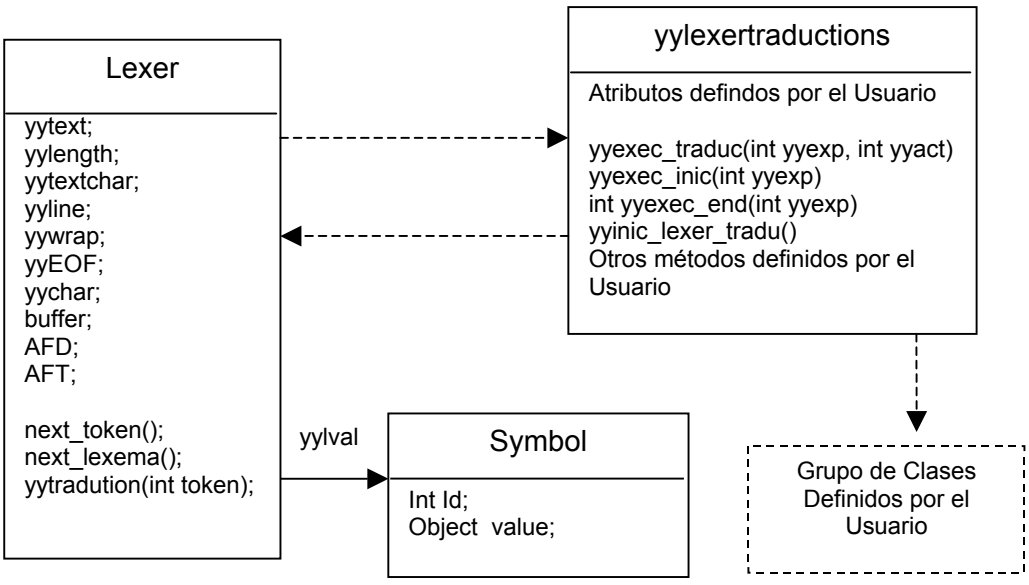


Figura 3.1: Diseño del Analizador Léxico.

Clase Lexer: su responsabilidad es leer caracteres del archivo de entrada, donde se encuentra la cadena a analizar, reconocer lexemas y retornar tokens luego de ejecutar las traducciones necesarias. Es decir, si el reconocimiento de un lexema está acompañado por

alguna acción semántica – como evaluar los atributos del token – invoca a la *clase yylexertraductions* la cual se encargará de ejecutar las traducciones correspondientes.

Clase yylexertraductions: la responsabilidad principal de esta clase consiste en ejecutar las traducciones cuando son invocadas por la *clase lexer*.

Clase Symbol: esta clase se utiliza para definir el tipo de los tokens.

Grupo de Clases del Usuario: este grupo contiene las clases definidas por el usuario las cuales generalmente participan de la traducción. Su responsabilidad es proveer las estructuras de datos y funcionalidades necesarias para llevar a cabo las traducciones. Este grupo puede existir o no según las necesidades del usuario.

3.2.2 – Clases del Analizador

Es importante recordar que las clases *lexer* y *yylexertraductions* son comunes a todos los traductores – con variaciones en la implementación de acuerdo al analizador – y que las clases de usuario deben ser creadas para cada analizador diferente por este motivo no se darán detalles de este último grupo.

a - Clase Lexer

Atributos de la clase:

- yytext: lexema que se va reconociendo.
- yylength: longitud del lexema que se va reconociendo.
- yytextchar: último caracter reconocido.
- yyline: número de línea en el que se está realizando el análisis léxico.
- yychar: número de caracter que se está reconociendo.
- yywrap: flag de error.
- yyEOF: flag de fin de archivo.
- yylval: valor del token corriente.
- buffer: contiene la cadena de entrada.
- AFD: representa el autómata finito determinístico el cual es utilizado para reconocer los lexemas.
- AFT: contiene todos los autómatas traductores los cuales son utilizados para ejecutar las acciones asociadas a la expresión regular correspondiente al lexema reconocido.

Todos los atributos de esta clase son privados por lo cual existen métodos públicos con el mismo nombre del atributo que permiten acceder a los mismos. Sólo cabe destacar el método yylval_update(Symbol s) el cual modifica el valor del token.

Métodos de la clase:

- next_token () : obtiene y retorna el próximo token ejecutando las traducciones necesarias. Para realizar esto, primero, invoca al método next_lexema y luego con el resultado obtenido invoca al método yytraduction.

- next_lexema (): simula el AFD y retorna el número de expresión regular reconocida. Este método retorna el número de expresión que machea con la cadena más larga y si existen dos expresiones que corresponden con dicha cadena, retorna el número de la primera expresión.

- yytradution (int token): invoca el método *yyexec_inic* de la clase *yylexertraductions*, el cual ejecuta, si existe, la acción inicial asociada al lexema reconocido. Luego, simula el AFT con el lexema reconocido ejecutando las acciones traductoras asociadas – invocando el método *yyexec_traduc* de la clase *yylexertraductions* . Por último ejecuta la acción final - invocando el método *yyexec_end* de la clase *yylexertraductions*.

b - Clase yylexertraductions

Esta clase tiene cuatro métodos fijos. Puede además contener algunos atributos y métodos que el usuario necesite definir para realizar las traducciones.

Los métodos fijos son los siguientes:

- yyexec_inic(int yyexp): Dada una expresión regular, ejecuta la acción inicial asociada a ella. Este método contiene todas las acciones iniciales definidas por el usuario.

- int yyexec_end(int yyexp): Dada una expresión regular, ejecuta la acción final asociada a ella. Este método contiene todas las acciones finales definidas por el usuario.

- yyexec_traduc(int yyexp, int yyact): Dada una expresión regular y una de las acciones traductoras asociada a ella, ejecuta dicha acción. Este método contiene todas las acciones traductoras definidas por el usuario.

- yyinic_lexer_tradu( ): Este método contiene el código que el usuario necesita ejecutar cuando se construye el analizador léxico. Es decir, este método será invocado por el constructor de la clase lexer.

c - Clase Symbol

Atributos de la clase:

- Id: token reconocido.

- value: valor del token reconocido.

Métodos de la clase:

- Symbol (int id, Object o): crea un symbol con un número de token y su valor.

- Symbol (int id), crea un symbol con solamente un número de token.

El usuario debería extender esta clase de acuerdo a sus necesidades.

3.2.3 - Diagrama de Interacción correspondiente a next_token

En el siguiente diagrama un objeto de tipo lexer solicita el próximo token. Como se mencionó anteriormente, para realizar esta tarea, primero se invoca al método next_lexema. El cual avanza en el buffer de entrada simulando el AFD. Cuando se reconoce un token, se invoca al método yytraduction.

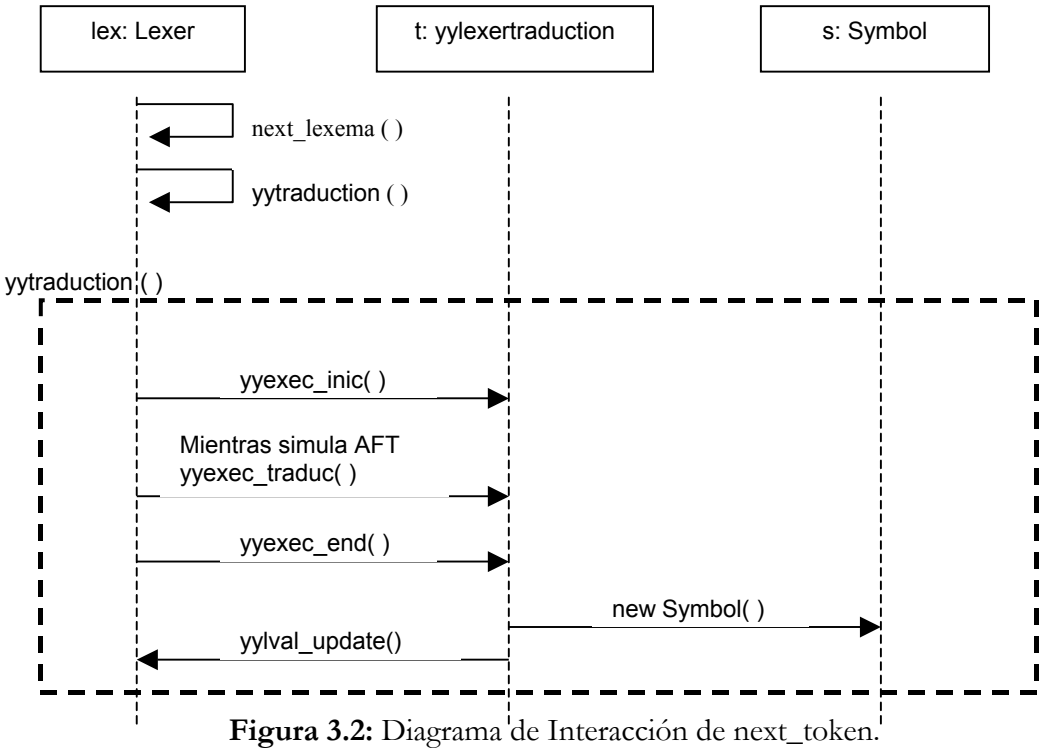


Figura 3.2: Diagrama de Interacción de next_token.

3.3 - Diseño de un Generador de Analizadores Léxicos Traductores

Así como existen herramientas que generan analizadores léxicos a partir de una especificación, a continuación se presenta el diseño de un generador de analizadores léxicos traductores. Es decir, una herramienta que dado una especificación de ETL's genere automáticamente un analizador léxico traductor. Los analizadores generados cumplen con el diseño presentado en la sección anterior.

En cuanto a la implementación del generador se darán las etapas en que se dividió la misma, desarrollando los detalles en los capítulos siguientes.

3.3.1 – Módulos del Generador

Esencialmente el generador de analizadores es un compilador que lleva una especificación de ETL's a un programa capaz de reconocer y traducir cadenas que cumplan con dicha especificación. Por lo tanto, es natural que el generador esté estructurado como muchos compiladores. De echo tiene un módulo de análisis lexicográfico, uno de análisis sintáctico y otro para generar código.

En la figura 3.3 se esbozan los módulos del generador y sus relaciones. Posteriormente se describirán las responsabilidades de cada módulo y sus relaciones con los demás.

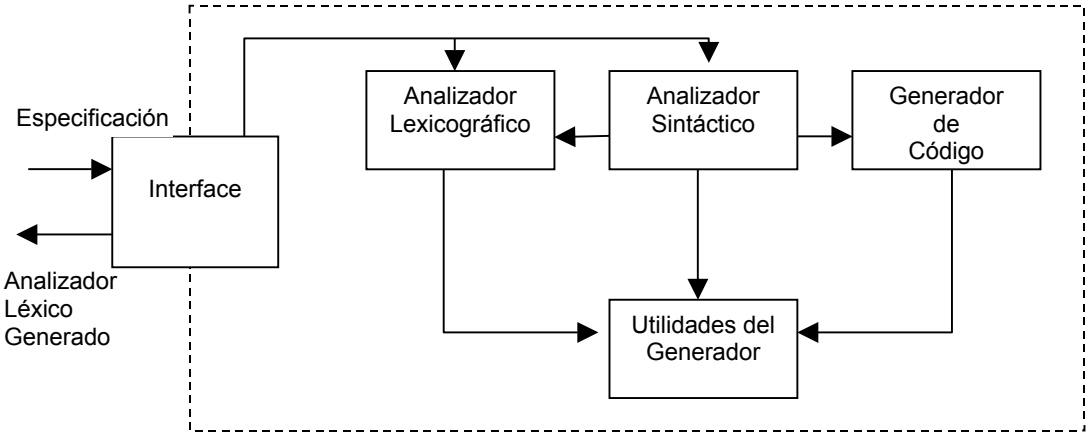


Figura 3.3: Módulos del Generador ♣.

Interface: toma la especificación – un archivo de texto - dada por el usuario para generar un analizador léxico y si no aconteció ningún error retorna el analizador generado – en un archivo de texto.

Analizador Lexicográfico: lee caracteres del archivo de texto que contiene la especificación de ETL's, reconoce lexemas y retorna tokens. Es utilizado por el módulo de interface que lo inicializa indicándole el archivo de lectura, y por el analizador sintáctico el cual hace los requerimientos de tokens. Además utiliza el módulo de utilidades, del cual obtiene las estructuras de datos y funcionalidades necesarias, como el tipo de los tokens y funciones sobre archivos.

Analizador sintáctico: el módulo de interface lo inicializa solicitándole que parsee el archivo de entrada. Utiliza el módulo de análisis lexicográfico al cual le solicita los tokens del archivo de entrada. Además, mientras realiza el análisis sintáctico construye el árbol sintáctico de las ETL's; para realizar esto utiliza tipos de datos y funcionalidades del módulo de utilidades.

Generador de Código: es el responsable de generar las estructuras de datos y el código necesario para crear los analizadores léxicos traductores. Es decir, crea el AFD y los AFT's, como así también toda la información y las funcionalidades necesarias para simularlos. Además debe crear el código que ejecute las traducciones y el código que implemente al analizador propiamente dicho. Este módulo utiliza servicios del módulo de utilidades para cumplir con sus responsabilidades.

♣ → denota relación de uso.

Utilidades del Generador: congrega todas las utilidades necesarias – tanto estructuras de datos como funcionalidades – para la generación de un analizador. Como existen distintos tipos de utilidades podría dividirse este módulo en componentes más pequeños y especificos. Con lo cual, también, se lograrían relaciones de uso más especificas.

3.3.2 - Clases y Relaciones del Generador

Como se puede apreciar el diseño presentado en el apartado anterior solo muestra una visión general del sistema sin entrar en detalles. Por lo cual, a continuación se presentará el diseño de las clases y sus inter-relaciones que conforman el generador. La notación empleada en esta sección – como en la sección anterior - es tomada de [Gam95] – y se describe brevemente en el anexo 3.

En la siguiente figura se esquematizarán los módulos del sistema – según la figura 3.3 - y las clases que los conforman.

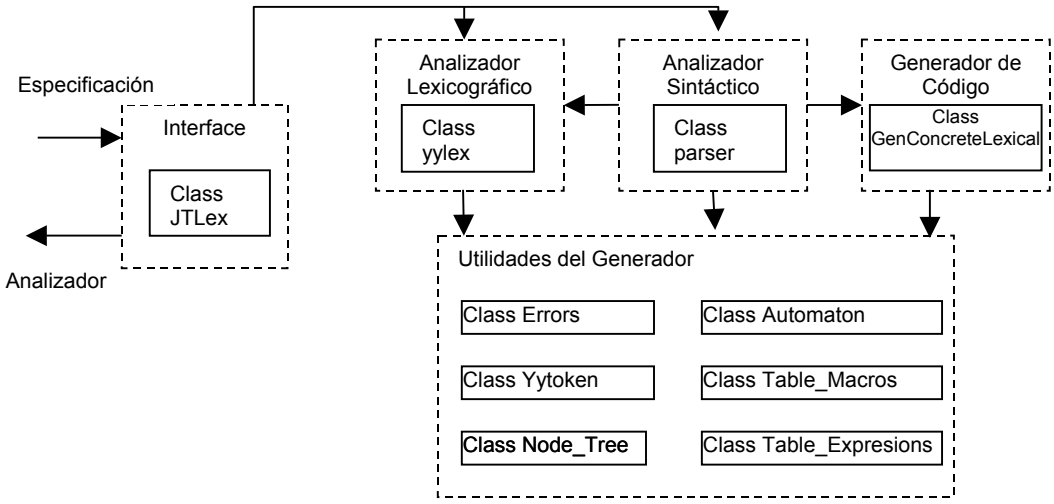


Figura 3.4: Clases del Generador ♠.

En los párrafos subsiguientes se comentara brevemente las responsabilidades y relaciones entre las clases del generador.

Clase JTLex: es la interface del generador con el usuario. Toma un archivo de entrada – un archivo de texto – con la especificación del analizador léxico traductor a generar y retorna el analizador léxico correspondiente – en el archivo *ConcreteLexical.java*.

Clase yylex: tiene por responsabilidad realizar el análisis lexicográfico de la especificación de entrada. Reconoce lexemas y retorna tokens a medida que la clase *parser* los solicita. Los tokens que retorna son de tipo *Yytoken*. Esta relacionada además con la clase *Errors* a la cual invoca cuando encuentra un error lexicográfico.

♠ □ Denota relación "Componente de"

Clase parser: su responsabilidades consiste en parsear el archivo de entrada. Solicita los token a la clase *yylex*. Mientras parsea la cadena de entrada construye el árbol sintáctico de las expresiones regulares, los nodos del árbol son de tipo *Node_Tree*. Crea y mantiene una tabla con los nombres de los macros y el árbol sintáctico de la expresión regular asociada a cada uno de estos. Dicha tabla es de tipo *Table_Macros*. Si detecta algún error invoca a la clase *Errors*.

Clase GenConcreteLexical: tiene la responsabilidad de generar todas las estructuras necesarias a partir del árbol sintáctico; estructuras tales como followpos, AFD, AFT's entre otras. Para lograr esto invoca a la clase *Table_Expresions.* Y a partir de estos datos genera el analizador léxico traductor, es decir imprime en el archivo *ConcreteLexical.java* los autómatas, demás estructuras y funcionalidades necesarias para implementar el analizador.

Clase Errors: tiene por responsabilidad informar el tipo de error que se produce. Para esto cuenta con la descripción de cada tipo de error y métodos que lo imprimen por la salida estándar. En todo momento debe ser capaz de informar si se ha producido o no un error.

Clase Yytoken: tipo de los tokens. Contiene información tal como clase del token y el lexema que fue reconocido. Además posee el número de línea en que fue reconocido.

Clase Node_Tree: tipo de los nodos del árbol sintáctico de las expresiones regulares traductoras. Cada nodo tiene como atributos una etiqueta, los firstpos y los lastpos del nodo en cuestión. Cada etiqueta es un operador de una expresión regular - |, * +, **.**, ? – o la etiqueta *caracter* – que representa cualquier caracter ASCII. Los nodos etiquetados con caracter son las hojas del árbol y tiene como atributo adicional el símbolo reconocido, la posición del símbolo en la expresión y una acción asociada. Contiene métodos para calcular los firstpos, lastpos y followpos.

Clase Table_Expresions: tiene por responsabilidad almacenar toda la información referente a las ETL's. Esta información consiste del árbol sintáctico, los followpos y el AFT para cada ETL. Además posee el AFD generado a partir de todos los AFT's. Esta clase contiene funcionalidades para generar toda la información que debe almacenar, es decir, invoca aquellos métodos necesarios para calcular los firstpos, lastpos y followpos. También, es la encargada de construir los AFT de cada ETL – invocando cuando es necesario a la clase *Automaton* – y a partir de esta invoca a *Automaton* para que construya el AFD. En el proceso de construir los AFT si se detecta que alguna de las expresiones no cumple con la definición de ETL se invoca a la clase *Errors* para informarlo.

Clase Table_Macros: almacena el nombre y el árbol sintáctico de cada macro definido en la especificación.

Clase Automaton: tiene por responsabilidad definir las estructuras de datos necesarias para representar AF. Posee además funcionalidades para definir un AF – estados y transiciones – y para construir un AFD a partir de los AFT's.

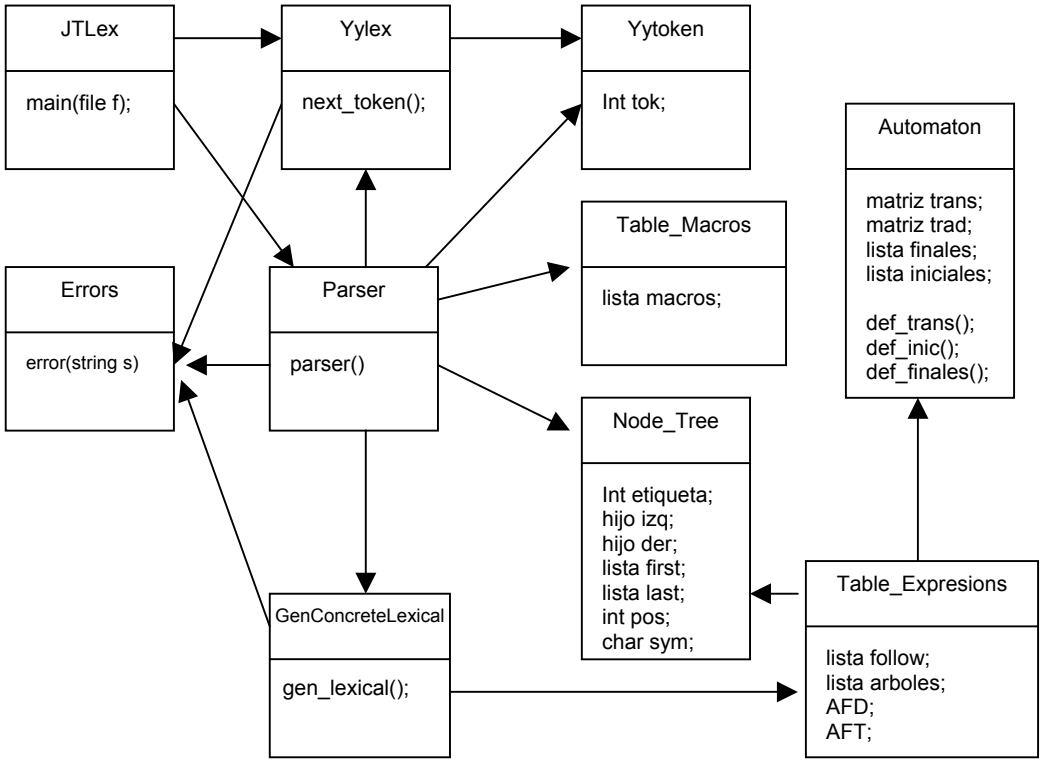


Figura 3.5: Relaciones entre las Clases del Generador.

En los siguientes capítulos se comentan los pormenores de la implementación de los módulos que conforman el generador de analizadores léxicos traductores.

Como se mencionó anteriormente el generador está estructurado como muchos compiladores. Razón por la cual resulta natural que su implementación este dividida en etapas similares en las que, por lo general, se divide el proceso de crear un compilador. Dichas etapas – enfocadas a la construcción de un generador de analizadores léxicos traductores - fueron planteadas en el capítulo 1 y más específicamente en el capítulo 2.

En el presente capítulo se presentó el diseño de los analizadores léxicos a generar y el diseño del generador de los mismos. En el capítulo 4 se exhibe el lenguaje de especificación de los analizadores léxicos a generar. El capítulo 5 trata sobre el análisis lexicográfico y sintáctico. La generación de código se presenta en el capítulo 6.

Se puede esquematizar el proceso de imlementación llevado a cabo con el siguiente gráfico:

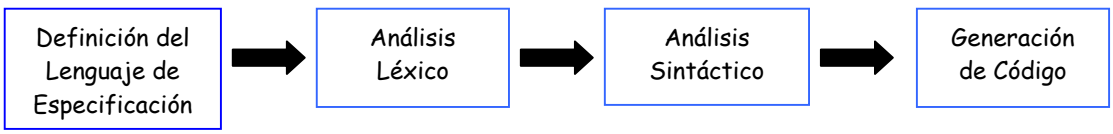


Figura 3.6: Proceso de implementación.

Capítulo 4

Definición del Lenguaje de Especificación

El primer aspecto a tener en cuenta para desarrollar un generador de analizadores léxicos es el lenguaje en que se especificarán los analizadores léxicos a generar. Dicho lenguaje es una descripción del conjunto de tokens que el analizador debe reconocer.

Generalmente estas especificaciones permiten definir la sintaxis de los símbolos mediante expresiones regulares. Quizás la herramienta más conocida es *Lex*. *Lex* es un generador de analizadores léxicos para el sistema operativo UNIX cuya especificación está basada en expresiones regulares que genera código *C*. Como *Lex* es casi un estándar el lenguaje definido es similar al de *Lex*, salvo que este lenguaje esta basado en expresiones regulares traductoras lineales.

A continuación se detallan los aspectos más relevantes de la especificación que utiliza el generador.

4.1 – Lenguaje de Especificación del Generador

Un archivo de entrada del generador está organizado en tres secciones, separadas por la directiva %%.

El formato es el siguiente:

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.

En la sección *Declaraciones* – la primer sección de la especificación – se definen macros y declaraciones propias del generador a ser utilizadas por las traducciones.

La sección *Reglas de las Expresiones Regulares Traductoras* contiene las reglas de análisis léxico, cada una de las cuales consiste de una acción inicial opcional, una expresión regular traductora lineal y una acción final.

Por último, la sección *Código de Usuario* es copiada directamente en el archivo de salida resultante. Esta provee espacio para la implementación de clases las cuales serán utilizadas por las traducciones.

4.1.1 - Declaraciones

Esta sección comienza antes del primer delimitador %%. Cada directiva debe comenzar al principio de la línea con %...{ y finaliza al principio de otra línea con %...}, a excepción de la declaración de macros. Las directivas son opcionales y deben ser especificadas en el orden que se introducen a continuación.

a - Definición de Atributos y Métodos del Usuario

La directiva %{ ... %} permite al usuario escribir código Java para ser copiado en el analizador léxico. Esta directiva es usado como se detalla a continuación:

```
%{
<Código>
%}
```

El código Java especificado en <*Código*> será copiado en la clase del analizador léxico que evalúa los atributos creada por JTLex.

```
class yylexertraductions
{
... <Código> ...
}
```

Esto permite la declaración de atributos y métodos internos para la clase que efectúa la traducción. Los nombres que comienzan con ***yy*** están reservados para ser usados por las clases del analizador léxico generadas.

b - Código de Inicialización del Código Usuario

La directiva %init{ ... %init} permite al usuario escribir código Java para ser copiado en un método de la clase yylexertraductions que es invocado por el constructor de la clase del analizador léxico (class lexer).

```
%init{
<Código>
%init}
```

El código Java especificado en <*Código*> será copiado en el método antes mencionado.

```
public void yyinic_lexer_tradu()
{
... <Código> ...
}
```

Esta directiva permite inicializar las variables de usuario en el momento de invocar el constructor del analizador léxico.

c - Código de Fin de Archivo para el Analizador Léxico

La directiva %eof{ ... %eof} permite al usuario escribir código Java para ser copiado en la clase del analizador léxico para ser ejecutado después que el fin de archivo es reconocido.

%eof{
<*Código*>
%eof}

d - Código de Error para el Analizador Léxico

La directiva %error{ ... %error} permite al usuario escribir código Java para ser copiado en la clase del analizador léxico para ser ejecutado cuando se detecta un error en el análisis léxico.

%error{
<*Código*>
%error}

e – Definiciones Regulares

Cada definición regular – macro – consiste de un nombre seguido de un espacio y una expresión regular – la cual no es traductora lineal y a la cual tampoco se le puede asociar una acción final –. El formato puede resumirse como se expresa a continuación:

<nombre del macro 1> <definición 1>
... ...
<nombre del macro N> <definición N>

Los nombres deben ser identificadores válidos, es decir secuencias de letras y dígitos comenzando con una letra. Los nombres deben estar ubicados al comienzo de una línea.

Las definiciones de macros deben ser expresiones regulares válidas tal como son descriptas en la próxima sección salvo la excepción que no se permite incluir acciones.

Estas definiciones pueden invocar a otros macros con el formato estándar {<nombre de macro>}. No es posible definir macros recursivos ni tampoco utilizar definiciones de macros no declarados previamente.

Una definición de macro termina cuando comienza otra definición de macro o cuando se encuentra el separador de secciones %%.

4.1.2 - Reglas de las Expresiones Regulares Traductoras Lineales

La segunda parte de la especificación consiste en una serie de reglas para dividir la cadena de entrada en tokens. Estas reglas son expresiones regulares traductoras lineales.

La reglas consisten de tres partes : una acción inicial opcional, una ETL y una acción final.

a - Acción Inicial

La acción Inicial es una directiva opcional de la forma INIT{ ... } que permite al usuario escribir código Java para inicializar variables. Esta le permite al usuario inicializar el entorno antes de reconocer un lexema.

INIT{ <*Código*> }

b - Expresión Regular Traductora Lineal

Los analizadores léxicos se escriben mediante expresiones regulares que permiten reconocer tokens. Para cada diferente token, se escribe una expresión regular que lo define. Esta herramienta, en vez de utilizar expresiones regulares, utiliza expresiones regulares traductoras lineales. 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 diferentes pueden llegar a ser distintas.

Por ejemplo si se dan los siguientes casos:

1. Dos expresiones regulares distintas con el mismo prefijo pero con acciones traductoras diferentes
   a ACTION{i++;} a ACTION{k++;}
   a ACTION{p++;} b
   cumple con la definición de expresión regular traductora.

2. Una expresión regular con el mismo prefijo pero con acciones traductoras diferentes
   a ACTION{i++;} a | a ACTION{k++;} b
   no cumple con la definición de expresión regular traductora.

3. Una expresión regular con el mismo prefijo pero con acciones traductoras iguales
   a ACTION{i++;} a | a ACTION{i++;} b
   cumple con la definición de expresión regular traductora.

El alfabeto es el conjunto de caracteres ASCII, los cuales los códigos de los mismos van desde el 0 al 127 inclusive.

Los siguientes caracteres son metacaracteres y tienen un significado especial para las expresiones regulares traductoras lineales:

**? * + | ( ) . [ ] { } “ **

? El signo de pregunta hace maching cuando existe cero o una ocurrencia de la expresión regular precedente.

***** El asterisco hace maching cuando existe cero o más ocurrencias de la expresión regular precedente.

+ El signo de suma hace maching cuando existe una o más ocurrencias de la expresión regular precedente, así *b*+ es equivalente a *bb**.

| Este signo se utiliza para representar la disyunción entre dos expresiones regulares. Por ejemplo si *a* y *b* son dos expresiones regulares, *a*|*b* significa que puede hace maching por *a* o por *b*.

(... **)** Los paréntesis son utilizados para agrupar expresiones regulares.

. Este signo hace maching con cualquier caracter de entrada excepto el fin de línea.

[... **]** Los corchetes se utilizan para representar conjuntos de caracteres o rangos. Existen varias formas de denotar los rangos:

> *[a,b,....,f]:* Se escribe el conjunto de caracteres a representar separados por comas. Por ejemplo si se quiere representar las letras "a", "b" y "c" se escribe [a,b,c].
>
> *[ab....f]:* Se escribe el conjunto de caracteres a representar. Por ejemplo si se quiere representar las letras "a", "b" y "c" se escribe [abc].
>
> *[a-z]:* El conjunto de caracteres representado por el rango [a-z] es el que se encuentra entre las letras "a" hasta la letra "z" de acuerdo al código ASCII.
>
> *[a*-x,z,A-Z]: Se puede utilizar una combinación de los tres casos anteriores para representar otros conjuntos de caracteres.

{nombre macro**}** Se utiliza para hacer una expansión de un macro definido previamente en la sección Declaraciones.

"abc**"** Representa a la palabra "abc".

\ Como existen caracteres reservados para el uso del generador, la barra seguido de un carácter representa el segundo caracter. De esta forma podemos escribir el caracter +, que es un caracter reservado, para el uso de la forma \+.

c - Acción Final

La acción final es una directiva *obligatoria* de la forma { ... } que permite al usuario escribir código Java que se ejecuta luego de reconocer un token. Al final del código Java y antes de escribir la segunda llave – } – el usuario debe incluir una sentencia **break** o una sentencia **return**.

{ <*Código*> }

d - Gramática de las reglas de las Expresiones Regulares Traductoras Lineales

Para clarificar los conceptos antes mencionados, a continuación se presenta la gramática de las reglas de las expresiones regulares traductoras lineales.

<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>

/* Símbolo de las expresiones regulares traductoras lineales */
<symbol> ::= + | * | ? |λ

/* Definición de las acciones */
<j_action> ::= ACTION{Código Java} | λ

en donde:

- CHAR es un carácter del código ASCII.
- words es una palabra dentro del alfabeto.
- rango es un rango.
- NAME_MACRO es el nombre de un macro definido previamente.

4.1.3 - Código de Usuario

En esta sección el usuario puede definir clases de Java que necesite utilizar para el analizador léxico creado. Este código puede comenzar con *package* <*nombre del paquete*> y también puede iniciar con *import* <nombre de la clase a importar>. El código es copiado en el archivo *Concrete_Lexical.java.*

En esta sección, por ejemplo, se puede definir una clase principal que invoque al analizador léxico. Para esto se invoca primero el constructor del analizador y luego un método llamado ***next_token*** que retorna el próximo token definido en la clase Lexer. A continuación se muestra un ejemplo del código usuario que define una clase Main que invoca al analizador léxico.

```
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();
      System.out.println("Expresion regular nro :"+i+" -
 Lexema : "+ L.yytext());
}
 System.out.println("Fin");
 }
}
```

4.1.4 – Comentarios

Los comentarios nos permiten escribir texto descriptivo junto al código, hacer anotaciones para programadores que lo puedan leer en el futuro. Comentando código se ahorra mucho esfuerzo. Además, cuando se escriben comentarios a menudo se pueden descubrir errores, porque al explicar lo que se supone que debe hacer el código es necesario pensar en dicho código. Es por esto que el lenguaje de especificación permite ingresar comentarios. Los mismos tienen la siguiente forma:

/* <Comentario> */ en donde <Comentario> es el texto que desea ingresar el usuario para comentar la especificación.

4.2 – Gramática del Lenguaje de Especificación

<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> ::= "INIT{" 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> ::= "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})*

4.3 - Un Ejemplo de una Especificación

A continuación se presenta un ejemplo de una especificación para un lenguaje algorítmico simple (subconjunto de C) denominado C--. Los símbolos básicos de C-- son los identificadores, literales y operadores.

Los identificadores tienen la siguiente estructura **"{letra} ({letra} | {digito})*"**; los literales enteros **"{digito}+"** y los denotadores reales **{digito}+ \. {digito}+**

Los delimitadores del lenguaje son los caracteres especiales y las palabras reservadas que se detallan a continuación:

+	-	*	/
%	!	?	:
=	,	>	<
(	)	{	}
||	&&	==	;
break	continue	else	float
if	int	return	while

Una posible especificación del analizador léxico traductor para C--, que en lugar de retornar tokens los imprime por pantalla, es la siguiente:

```
/* Código de inicialización */
%init{
      tokenpos=0;
      cant_coment=0;
%init}
/* Definiciones de Macros*/
letra [a-z,A-Z]

%%

/* sección declaración de las ETL's*/
(\ |\t)+ {tokenpos+=yylength();break;}
\n       {tokenpos=0;break;}
"\*"     {cant_coment++;tokenpos+=yylength();break;}
"*/"     {cant_coment--;tokenpos+=yylength();break;}
"float"{tokenpos+=yylength();System.out.println("float");break;}
"int"  {tokenpos+=yylength();System.out.println("int");break;}
"break"  {tokenpos+=yylength();System.out.println("break");break;}
"continue"
{tokenpos+=yylength();System.out.println("continue");break;}
"else"  {tokenpos+=yylength();System.out.println("else");break;}
"if"  {tokenpos+=yylength();System.out.println("if");break;}
"return"
{tokenpos+=yylength();System.out.println("return");break;}
"while"  {tokenpos+=yylength();System.out.println("while");break;}
"+"  {tokenpos+=yylength();System.out.println("+");break;}
"-"  {tokenpos+=yylength();System.out.println("-");break;}
"*"  {tokenpos+=yylength();System.out.println("*");break;}
"/"  {tokenpos+=yylength();System.out.println("/");break;}
"%"  {tokenpos+=yylength();System.out.println("%");break;}
"!"  {tokenpos+=yylength();System.out.println("!");break;}
"?"  {tokenpos+=yylength();System.out.println("?");break;}
":"  {tokenpos+=yylength();System.out.println(":");break;}
"="  {tokenpos+=yylength();System.out.println("=");break;}
","  {tokenpos+=yylength();System.out.println(",");break;}
">"  {tokenpos+=yylength();System.out.println(">");break;}
"<"  {tokenpos+=yylength();System.out.println("<");break;}
"("  {tokenpos+=yylength();System.out.println("(");break;}
")"  {tokenpos+=yylength();System.out.println(")");break;}
"{"  {tokenpos+=yylength();System.out.println("llaveO");break;}
"}"  {tokenpos+=yylength();System.out.println("llaveC");break;}
"||"  {tokenpos+=yylength();System.out.println("OR");break;}
"&&"  {tokenpos+=yylength();System.out.println("AND");break;}
"=="  {tokenpos+=yylength();System.out.println("==");break;}
";"  {tokenpos+=yylength();System.out.println("puntoYc");break;}

{letra} ({letra}|{digito})* {System.out.println("Identificador:
"+yytext());break;}

INIT{intval=0;} ([0-9]
         ACTION{tokenpos++; Integer aux=new
Integer(yytextchar()+"");
         intval=intval*10 + aux.intValue();} ) +
  {System.out.println("Natural: "+intval);break;}
```

```
INIT{realval1=0;realval2=0;cantreal=1;}
          ([0-9] ACTION{tokenpos++; Integer aux=new
Integer(yytextchar()+"");
          realval1=realval1*10 + aux.intValue();} ) +
          \. ([0-9]
          ACTION{tokenpos++; Integer aux=new
Integer(yytextchar()+"");
          cantreal=cantreal*10;
          realval2=realval2*10 + aux.intValue();}) +
  {real=realval1+(realval2/cantreal); System.out.println("Real:
"+real);break;}

. {System.out.println("Error Identificador no valido" +yytext()) ;
break;}
%%
/* sección código del 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();
     System.out.println("Expresion regular nro :"+i+" - Lexema :
"+ L.yytext());
    }
   System.out.println("Fin:  C-- ");
  }
}
```

Capítulo 5

Análisis Léxico y Sintáctico

Para generar los analizadores léxicos a partir de una especificación primero se debe entender su estructura y significado. El análisis necesario para llevar a cabo esta tarea se divide en dos etapas:

- *Análisis Léxico:* divide la entrada en palabras individuales o tokens.
- *Análisis Sintáctico:* Parsea la estructuras de las frases de la especificación.

En las siguientes subsecciones se presentan los detalles de implementación de los puntos anteriores para el generador de analizadores léxicos traductores.

5.1 – Análisis Léxico

El análisis léxico toma una cadena de caracteres y produce una cadena de nombres, palabras reservadas y signos de puntuación; este descarta espacios en blanco y comentarios entre tokens. La principal razón de separar el análisis léxico del parsing reside en simplificar la complejidad de este último.

El análisis léxico no es muy complicado, pero debe ser tratado con formalismos y herramientas poderosas porque gran parte del tiempo se consume en leer la especificación fuente y dividirla en componentes léxicos.

5.1.1 – Lexemas y Tokens del Generador

La siguiente tabla muestra los tokens, algunos lexemas de ejemplo y la expresión regular que lo describe.

Token	Lexemas de Ejemplo	Expresión Regular
OR	|	|
TIMES	*	*
PLUS	+	+
QUESTION	?	?
LPARENT	(	(
RPARENT	)	)
LBRACKET	[	[
RBRACKET	]	]

SCORE	-	-
SEMICOLON	,	,
COLON	.	“ ”
BAR_CHAR	\t, \a, \b	\.
MARKS_TEXT	“ , ”	“
JAVA_DECL	%{ t++ %}, %{int a;%}	%{ (. | \n) %}
JAVA_DECL_INIT	%init{ t=0; %init}	%init{ (. | \n) %init}
JAVA_DECL_EOF	%eof{ eof(“eof”); %eof}	%eof{ (. | \n) %eof}
JAVA_DECL_ERROR	%error{ error(“error”); %error}	%error{ (. | \n) %error}
JAVA_ACTION	ACTION{ i++;}	ACTION{ (. | \n) }
INIT_JAVA_ACTION	INIT{ i=0;}	INIT{ (. | \n) }
END_JAVA_ACTION	{ return (yytext());}	{ (. | \n) }
USE_MACRO	{signos}	{ALPHA (ALPHA | DIGIT | _)*}
DOBLE_PERCENT	%%	%%
NAME_MACRO	Signo, digito, natural, real	ALPHA (ALPHA | DIGIT | _)*
CHAR	A, b, c, e, &	.

Donde ALPHA = [A-Za-z] y DIGIT = [0-9] y **.** es cualquier caracter ASCII cuyo código esté entre 0 y 127.

Se debe recordar que en esta etapa además de reconocer los lexemas y retornar el token correspondiente, se eliminan los espacios en blanco, tabulaciones, caracteres de fin de línea – en los casos que fuera necesario – y los comentarios que se encuentran en la especificación.

Para implementar el analizador léxico del generador se utilizó un generador de analizadores léxicos llamado *JLex*. En el anexo 1 se encuentra la especificación *JLex* para el generador de analizadores léxicos traductores. Esta herramienta se describe a continuación.

5.1.2 – El Generador de Analizadores Léxicos para Java: JLex

Como se mencionó en los primeros capítulos, existe un algoritmo para la construcción de AFD’s a partir de expresiones regulares. Es por esto que existen herramientas que generan automáticamente analizadores léxicos traduciendo expresiones regulares en AFD’s.

JLex es un generador de analizadores léxicos que produce un programa *Java* a partir de una especificación léxica. Por cada tipo de token la especificación contiene una expresión regular y una acción. La acción comunica el tipo del token – y posiblemente alguna otra información – a la fase del análisis sintáctico.

La salida de *JLex* es un programa *Java* – un analizador léxico que simula un AFD y ejecuta la acción correspondiente usando los algoritmos descriptos en [App98] –. Las acciones son solamente sentencias Java que retornan valores de tokens.

Los tokens son descriptos en *JLex* de manera similar a la utilizada por *Lex*.

La primer parte de la especificación, antes del símbolo %%, contiene declaraciones de paquetes, declaraciones import y clases que pueden ser usadas por el código *Java* que se encuentra en el resto de la especificación.

La segunda parte contiene definición de macros – que luego simplifican la definición de las expresiones regulares - y declaración de estados.

La tercera, y última parte, contiene expresiones regulares y acciones. Las acciones son fragmentos de código *Java*. Cada acción puede retornar el valor de un token del tipo definido por la declaración %type. Si la acción no retorna un valor entonces el token corriente es descartado y el analizador léxico se re-invoca a el mismo para obtener el siguiente token.

Las acciones tienen disponibles algunas variables de entorno que pueden ser utilizadas para realizar las traducciones. Por ejemplo la cadena que macheó con la expresión regular, es decir el lexema reconocido, esta es almacenado por yytext( ); la línea en que se está realizando el análisis se puede obtener de yyline( ).

Estados

Las expresiones regulares son estáticas y declarativas; los autómatas son dinámicos e imperativos. Esto quiere decir que se pueden ver los componentes y estructuras de una expresión regular sin tener que simular un algoritmo, pero para entender un autómata es necesario "ejecutarlo" mentalmente. En la mayoría de los casos, las expresiones regulares proveen una notación práctica, fácil y conveniente para especificar las estructuras léxicas de los tokens.

Pero alguna veces el modelo de transición de estado de los autómatas es apropiado. *Jlex* tiene un mecanismo que mezcla estados con expresiones regulares. Uno puede declarar un conjunto de estados de inicio; cada expresión regular puede ser antepuesta por el conjunto de estados de inicio en el cual es válida. Las acciones pueden cambiar explícitamente el estado de inicio. En efecto, lo que se obtiene es un autómata finito cuyas transiciones están rotuladas, no por símbolos simples, sino por expresiones regulares. A continuación se muestra un ejemplo con identificadores simples y comentarios delimitados por /* y */:

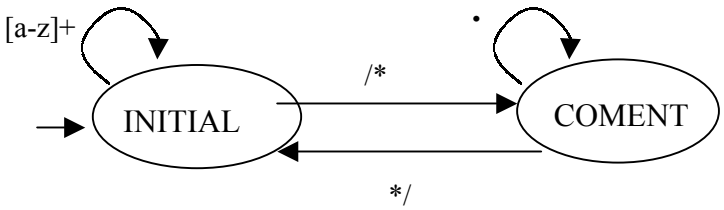


Figura 5.1: Representación de Estados en Jlex.

Para más detalles relativos a *JLex* consultar [Ber99].

5.2 – Análisis Sintáctico

El lenguaje de especificación del generador está descrito por medio de reglas que prescriben la estructura sintáctica del mismo. Esta descripción se realizó utilizando una gramática independiente del contexto. Utilizar una gramática en éste caso ofrece ventajas significativas:

- La gramática da una descripción sintáctica precisa y fácil de entender del lenguaje.

- A partir de la gramática se pudo construir automáticamente un analizador sintáctico eficiente que determina si una especificación fuente está sintácticamente bien formada. También permitió identificar ambigüedades sintácticas en el proceso de construcción del analizador sintáctico.

- Si el lenguaje de especificación del generador evoluciona, adquiriendo nuevas construcciones y realizando tareas adicionales, se pueden añadir estas nuevas construcciones con más facilidad. Es decir, cuando existe una aplicación basada en una descripción gramatical del lenguaje, se simplifican las tareas a realizar si se extiende el lenguaje especificado.

El funcionamiento del analizador sintáctico del generador es idéntico al del analizador sintáctico de un compilador. Obtiene una cadena de componentes léxicos del analizador léxico y comprueba si la cadena puede ser generada por la gramática del lenguaje fuente – ver figura 1.2.

Para implementar el analizador sintáctico del generador se utilizó un generador de analizadores sintácticos – un generador de parsers - llamado *CUP*. En el anexo 2 se encuentra la gramática *CUP* para el generador de analizadores léxicos traductores. A continuación se describe esta herramienta.

5.2.1 –El Generador de parsers LALR para *Java: CUP*

CUP (Java Based Constructor of Useful Parser) es un sistema para generar parsers LALR de especificaciones simples. Este tiene el mismo rol que el conocido y muy usado *YACC* y, de echo, ofrece muchos de los servicios que ofrece *YACC* con el cual comparte muchas características. Sin embargo, *CUP* esta escrito en *Java*, usa especificaciones que incluyen código *Java* embebido y produce parser que estan implementados en *Java*.

Usar *CUP* consiste en crear una especificación de un lenguaje – una gramática - para el cual se necesita el parser; además es necesario contar con un escáner capaz de dividir en tokens la cadena de entrada – razón por la cual se utilizo *Jlex*..

Una especificación *CUP* se puede dividir en cuatro partes principales. La primer parte provee declaraciones preliminares para especificar como el parser será generado suministrando partes del código perteneciente al runtime del parser. La segunda parte de la especificación declara terminales y no terminales de la gramática y asocia clases de objetos con cada una. La tercer parte especifica la

precedencia y asociatividad de los terminales. La última parte de la especificación contiene la gramática. Para más detalles consultar [Hud99].

5.3 – Reconocimiento de Errores

Si el generador tuviera que procesar sólo especificaciones correctas su diseño e implementación se hubiera simplificado. Pero los programadores a menudo escriben especificaciones incorrectas, y un buen generador debería ayudar al programador a identificar y localizar errores.

Los tipos de errores que pueden aparecer son:

- Errores Léxicos.
- Errores Sintácticos.
- Errores Semánticos.

Gran parte de la detección de errores se centra en las fases de análisis léxico y sintáctico. La razón es que muchos de los errores son de naturaleza sintáctica o se manifiestan cuando la cadena de componentes léxicos que proviene del analizador léxico desobedece las reglas gramaticales que definen la especificación.

Los errores identificados por el generador en las dos fases presentadas en este capítulo son los siguientes:

- Nombre de macro ilegal: este error se produce debido a que se ingresa un caracter no permitido para definir el nombre del macro.
- Redefinición de macro: ocurre cuando se definen dos o más macros con el mismo nombre.
- Macro no definido: se produce cuando se invoca un macro y el macro no estaba previamente definido.
- Rango negativo: error que ocurre cuando se ingresa un rango negativo de acuerdo al código ASCII. Por ejemplo [z-a] es un rango negativo, no así [a-z].
- Caracter Ilegal: este error es causado cuando se ingresa un caracter que no sea un caracter ASCII entre los códigos 0 y 127.

- Llave de apertura no esperada: se produce cuando existen llaves no balanceadas.

- Llave de cierre no encontrada: este error ocurre cuando existen llaves no balanceadas.

- Apertura de comentario no esperada: se produce cuando existen comentarios no balanceados.

- Cierre de comentario no esperado: es causa de comentarios no balanceados.

Capítulo 6

Generación del Código que Implementa el Analizador Léxico Traductor Especificado

Luego de haber discutido los primeros pasos de la implementación del generador, en el presente capítulo se desarrollan los detalles más relevantes de la implementación de la última fase, la generación del código que implementa el analizador léxico traductor especificado.

Este capítulo gira entorno de dos ejes centrales. El primero consiste en la creación de las estructuras de datos necesarias y el segundo en la generación del código del analizador. Por lo tanto, trata sobre la implementación del algoritmo dado en el capítulo 2 para crear un analizador léxico traductor.

Si bien parte de lo presentado en este capítulo es realizado durante el análisis sintáctico, como la construcción de los árboles sintácticos, es comentado en este capítulo y no en el anterior debido a que estos árboles son datos indispensables para la generación del resto de las estructuras necesarias. Además, se comenta la implementación de los autómatas finitos traductores y del autómata finito determinista. Por último se realiza un breve comentario sobre la implementación del generador de código.

Para clarificar el proceso que se realiza en esta última etapa se utiliza un ejemplo basado en la siguiente especificación de un analizador léxico traductor:

```
%{ int i;
%}
%init{ i=0;
%init}
%%
   (0 |1)(0 ACTION{i++;} |1 ACTION{i++;})* \. (0|1) {System.out.println("ETL 1");}
   (0 ACTION{i++;} |1 ACTION{i++;})* X {System.out.println("ETL 2");}

%%
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();
   }
 }
}
```

Figura 6.1: Especificación del Ejemplo.

6.1 – Generación de las Estructuras de Datos

Según el algoritmo presentado en la sección 2.4 los datos necesarios para obtener un analizador léxico traductor a partir de una especificación de ETL's son los siguientes:

- Construcción del *árbol sintáctico* de cada ETL.
- Computar la función *followpos*, a partir de las funciones *firstpos* y *lastpos*, para cada árbol sintáctico.
- Construcción del AFT de cada ETL.
- Construcción del AFNλ que acepta la unión de los lenguajes correspondientes a los autómatas obtenidos en el punto anterior.
- A partir del AFNλ obtener el AFD equivalente.

6.1.1 - Construcción del Arbol Sintáctico

A medida que se parsea cada ETL se construye el árbol sintáctico de la misma. El árbol sintáctico esta definido de la siguiente manera:

Arbol → Arbol :|: Arbol
| Arbol :.: Arbol
| :*: Arbol
| :+: Arbol
| :?: Arbol
| Hoja (∑, Posición)
| λ

Los nodos del árbol están etiquetados. La etiqueta puede ser un operador de una expresión regular (|, * +, **.**, ?) o la etiqueta *caracter* (que representa cualquier caracter ASCII cuyo código esta entre el 0 y 127). Se definió la clase *Etiquette_node* para representar las etiquetas del árbol sintáctico.

Aquellos nodos que tienen la etiqueta *caracter* constituyen las hojas del árbol. Estas tienen como información adicional un símbolo – el caracter ASCII reconocido –, la acción asociada al símbolo y la posición de la hoja – un número entero. Todas las hojas tienen posiciones diferentes y se numeran de izquierda a derecha comenzando en uno.

En cambio de aumentar la ETL, una vez obtenido el árbol sintáctico de la misma, se agrega una hoja que contiene la última posición del árbol y un caracter que no pertenezca al alfabeto del lenguaje. Para esto se utilizó el caracter ASCII cuyo código es 161 (¢). Este símbolo es necesario en pasos posteriores para calcular los estados finales del autómata traductor que reconoce el lenguaje generado por la ETL.

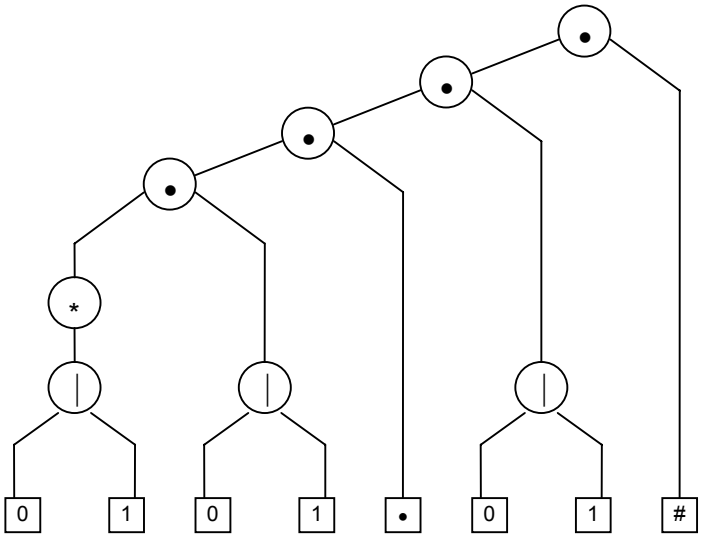


Figura 6.2a: Arbol Sintáctico de la ETL (0 | 1)* (0 | 1) **.** (0 | 1) #.

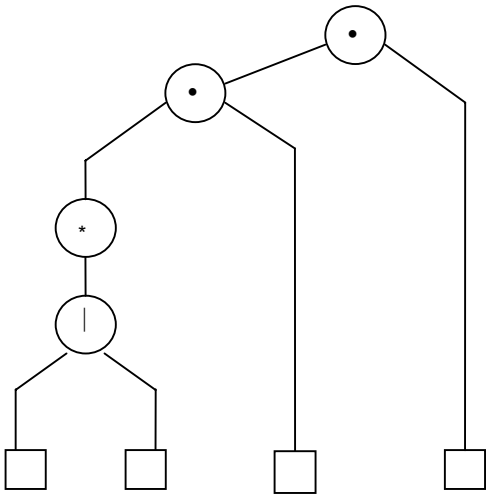


Figura 6.2b: Arbol Sintáctico de la ETL (0 | 1)* X #.

6.1.2 – Cómputo de la función *followpos*

Esta función se calcula a partir de las funciones *firstpos* y *lastpos*, las cuales, fueron definidas en el paso 3 de la sección 1.4.4. Estas funciones recursivas se computan sobre el árbol sintáctico y el resultado que arrojan, un conjunto de posiciones, se almacena en los nodos del árbol. Razón por la cual es necesario que cada nodo tenga dos atributos que almacenen estos conjuntos de posiciones. Para esto se creó la clase *Set_positions* la cual implementa un conjunto de posiciones.

Luego se calcula la función *followpos* – también definida en el paso 3 de la sección 1.4.4 – la cual por cada posición del árbol retorna un conjunto de posiciones. Se definió una clase llamada *Followpos* que posee un vector donde el elemento i-esimo contiene los *followpos* de la posición i-esima.

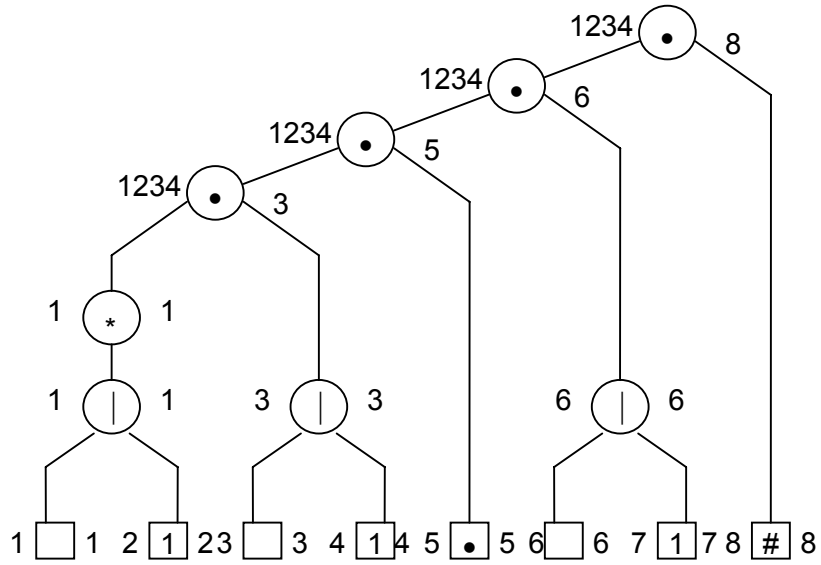


Figura 6.3a: Arbol sintácticos de la figura 6.2a decorados con las funciones *firstpos y lastpos.*

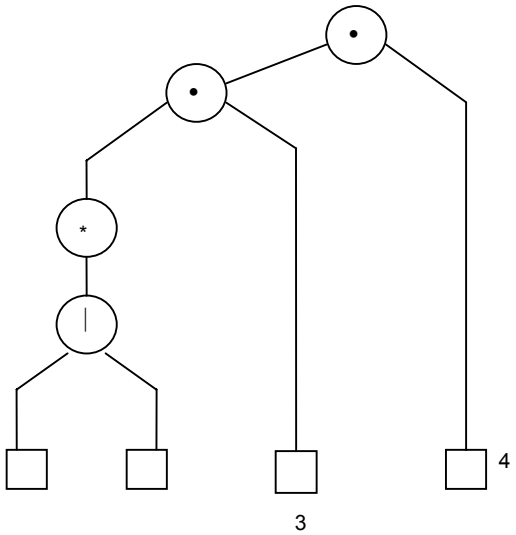


Figura 6.3b: Arbol sintácticos de la figura 6.2b decorados con las funciones *firstpos y lastpos.*

posición	followpos(posición)	posición	followpos(posición)
1	{ 1, 2, 3, 4 }	5	{ 6, 7 }
2	{ 1, 2, 3, 4 }	6	{ 8 }
3	{ 5 }	7	{ 8 }
4	{ 5 }	8	-

Figura 6.4a: Tablas con los *followpos* correspondientes al árbol de la ETL (0 | 1)* (0 | 1) **.** (0 | 1) #.

posición	followpos(posición)	posición	followpos(posición)
1	{ 1, 2, 3 }	3	{ 4 }
2	{ 1, 2, 3 }	4	-

Figura 6.4b: Tablas con los *followpos* correspondientes al árbol de la ETL (0 | 1)* X #.

.

6.1.3 – Construcción del AFT de cada ETL

Para llevar a cabo este paso, se implementó el algoritmo 1 presentado en la sección 2.3 con una leve modificación. No se construye una nueva matriz por cada AFT, sino que se unen en una misma matriz registrando los estados iniciales de cada autómata. Este proceso se realiza de la siguiente manera:

a- Inicialmente los conjuntos de estados finales e iniciales son vacíos al igual que las matrices de transiciones y de acciones.
b- Se construye el AFT de una ETL.
c- Se agrega el AFT a la matriz del AFTNλ a continuación del último AFT.
d- Se agrega el estado inicial correspondiente y los estados finales.
e- Volver al paso b hasta que no existan más AFT por construir.

Para representar los autómatas se definió la clase *Automaton* la cual tiene por atributos una matriz de transiciones, una matriz de traducciones, un conjunto de estados iniciales y un conjunto de estados finales por cada autómata.

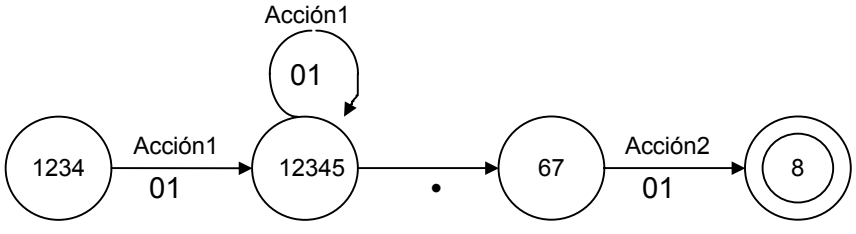


Figura 6.5a: AFT de la ETL (0 | 1)* (0 | 1) **.** (0 | 1) #.

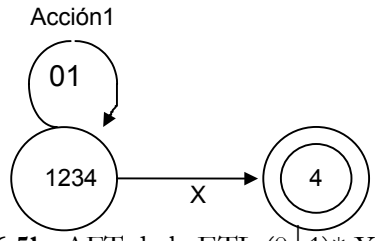


Figura 6.5b: AFT de la ETL (0 | 1)* X #.

Estados\Símbolos	0	1	.	X
Estado 1 (1234 fig 6.5a)	2	2	-	-
Estado 2 (12345 fig. 6.5a)	2	2	3	-
Estado 3 (67 fig. 6.5ª)	4	4	-	-
Estado 4 (8 fig. 6.5a)	-	-	-	-
Estado 5 (1234 fig. 6.5b)	5	5	-	6
Estado 6 (4 fig. 6.5b)	-	-	-	-

Figura 6.6a: Matriz de transiciones de los AFT's.

Estados\Símbolos	0	1	.	X
Estado 1 (1234 fig 6.5a)	Acción1.1	Acción1.1	-	-
Estado 2 (12345 fig. 6.5a)	Acción1.1	Acción1.1	-	-
Estado 3 (67 fig. 6.5ª)	Acción1.2	Acción1.2	-	-
Estado 4 (8 fig. 6.5a)	-	-	-	-
Estado 5 (1234 fig. 6.5b)	Acción2.1	Acción2.1	-	-
Estado 6 (4 fig. 6.5b)	-	-	-	-

Figura 6.6b: Matriz de traducciones de los AFT's.

Estados/Expresión	ETL (0 | 1)* (0 | 1) . (0 | 1)	ETL (0 | 1)* X
Estado Incial	1	5
Estados Finales	4	6

Figura 6.7: Estados Iniciales y Finales de los AFT's.

6.1.4 – Construcción del AFNλ

Como ya se mencionó en los primeros capítulos se necesita reconocer la unión de los lenguajes de los autómatas. Se pueden unir los autómatas creando un AFNλ a partir de los autómatas ya construidos – ver paso 3 de la sección 1.5. Dicha construcción fue simulada mediante el conjunto de estados iniciales que se obtuvo en el paso anterior. Es decir, en vez de crear un nuevo estado y unirlo por transiciones lambda a los estados iniciales de los AFT, se mantiene un registro de todos los estados que se pueden acceder por lambda desde el estado inicial del AFNλ.

6.1.5 – Construcción del AFD

A partir de la matriz de transiciones del AFNλ, obtenido en el paso anterior, se construye el AFD según el algoritmo del paso 4 presentado en la sección 2.3. La implementación del algoritmo no utiliza la función *clausura_λ*, ya que no existen transiciones λ en la matriz de transiciones. Además no es necesario calcular el estado inicial del autómata a construir ya que este es el conjunto de estados iniciales provisto por el paso anterior.

Además de la matriz de transiciones se calculan los estados finales y a que ETL corresponde cada estado final. Si un estado final corresponde a más de una ETL entonces se le asigna la primera expresión que se encuentra en la especificación – ver función *tok* definida en la sección 2.4.

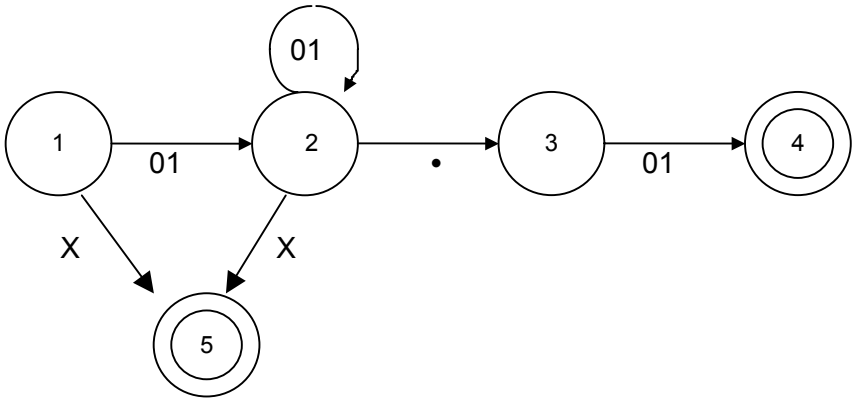


Figura 6.8: Autómata Finito Determinístico.

Estados\Símbolos	0	1	.	X
Estado 1	2	2	-	5
Estado 2	2	2	3	5
Estado 3	4	4	-	-
Estado 4	-	-	-	-
Estado 5	-	-	-	-

Figura 6.9a: Matriz de transiciones del AFD.

Estados Finales	ETL asociada
4	(0 | 1)* (0 | 1) . (0 | 1)
5	(0 | 1) * X

Figura 6.9b: Estados finales del AFD.

6.1.6 – Clase Table_expresions

Todos los datos obtenidos en los puntos anteriores se almacenan en una instancia de la clase *Table_expresions* la cual contiene:

- Un vector, donde el elemento *i* contiene el árbol sintáctico de la ETL *i-esima*.
- Un vector, donde el elemento *i* contiene los *followpos* del árbol sintáctico de la ETL *i-esima*.
- Los AFT y el AFD con la información referente a los estados iniciales y finales. Los cuales son dos instancias distintas de la clase A*utomaton*.

6.2 – Generación del Código

Para generar el código se siguió el diseño del analizador léxico que se presentó en el capítulo 3. El código generado consta de un "esqueleto fijo" que constituye el analizador léxico y una sección que varía y está constituida por las estructuras de datos necesarias para implementar el analizador léxico. Estas estructuras son el AFD y el AFT. La clase que tiene la responsabilidad de generar el archivo que contiene el analizador léxico – *Concrete_lexical.java* – es la clase *GenConcreteLexical*.

6.2.1 – Código generado para el ejemplo

En el anexo 4 se encuentra el código generado para el ejemplo presentado en este capítulo.

6.3 – Diagrama de Secuencia para Generar el Analizador

A continuación se presenta el diagrama de secuencia que esquematiza la construcción del analizador léxico:

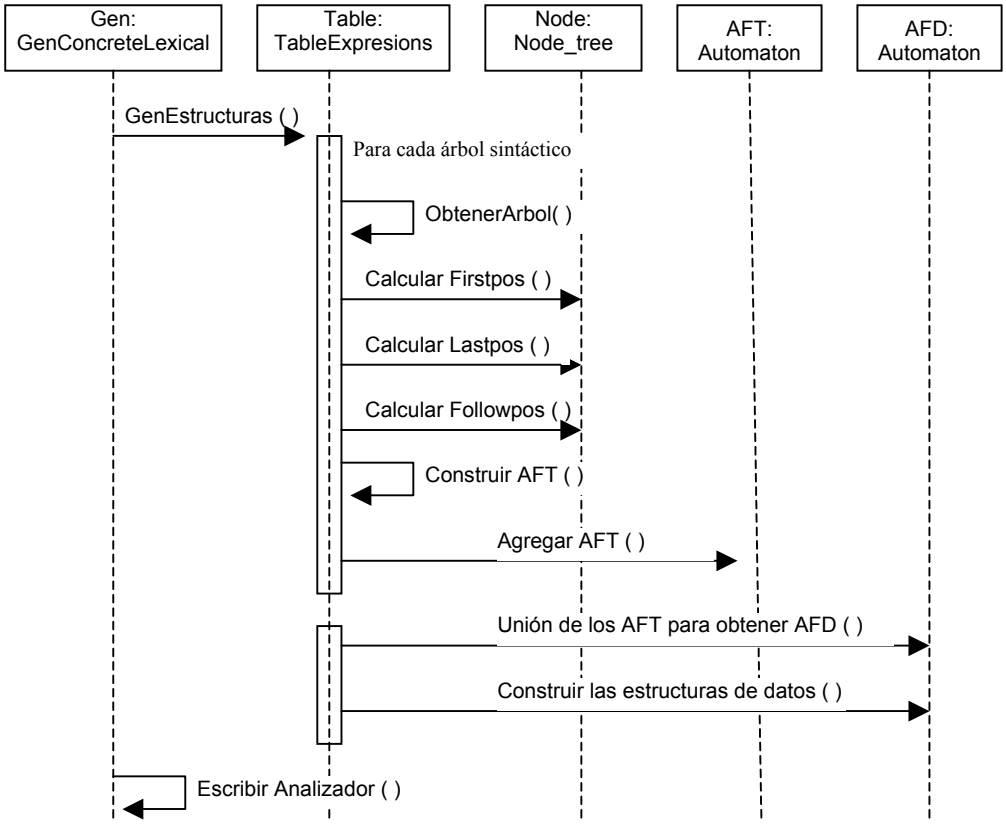


Figura 6.11: Diagrama de Secuencia para Generar el Analizador

Capítulo 7

El Generador de Analizadores Léxicos Traductores dentro del Entorno Japlage

El presente capítulo describe la incorporación del generador de analizadores léxicos traductores al entorno de generación de procesadores de lenguajes – *Japlage*, JAva Processors LAnguage Generation Environment –.

Japlage está basado en un nuevo modelo, que integra de una manera más general a los Esquemas de Traducción con las gramáticas de Atributos. El nuevo formalismo ha sido llamado Esquemas de Traducción con atributos, *Attibute Translation Schems* (ATS).

Las gramáticas de atributos (AG) permiten expresar propiedades dependientes de contexto, asociando atributos - variables tipadas - a los símbolos de una gramática y reglas de evaluación de atributos a las producciones. Las reglas de evaluación deben tener la forma $a_0 = f(a_1, a_2, a_3,..., a_n)$ - donde los a_i deben ser atributos de los símbolos de la producción, y *f* una aplicación estrictamente funcional -. Para la ejecución de estas reglas no se determina un orden explícito de evaluación sino que ellas expresan un conjunto de ecuaciones simultáneas que pueden evaluarse en cualquier orden que respete la dependencia implícita entre los valores de los atributos.

Las implementaciones de los esquemas de traducción, *Trasnslation Schems* (TS), en cambio, agregan acciones intercaladas entre los símbolos de la parte derecha de las producciones de una gramática; su semántica consiste en ejecutar tales acciones de izquierda a derecha en el orden en que aparecen, como hojas, en los árboles de derivación. En este caso el orden de ejecución es parte esencial de la gramática, ya que estas acciones pueden producir efectos colaterales, dejando, por lo tanto, de valer el principio de transparencia referencial que regía en evaluación de las AGs.

Yacc, Javacc y otros generadores de procesadores de lenguaje actuales utilizan TSs enriquecidos con atributos. Permiten hacer referencia a atributos dentro de las acciones de traducción, pero el computo de atributos se realiza mediante acciones. El orden de evaluación de los atributos, queda así determinado, por el orden en que estas deben ejecutarse. Este mecanismo pierde generalidad respecto a las AGs puesto que impone un orden de evaluación compatible sólo con las llamadas Gramáticas de atributos por izquierda, restricción que se extiende a la utilización de atributos dentro de las acciones de traducción**.**

7.1 - Esquema general de funcionamiento

Para procesar una cadena de entrada que respeta una cierta estructura – o que pertenece a un determinado lenguaje formal - se realizan las siguientes tareas: 1) dividir la cadena en unidades o componentes léxicos – elementos que despeñan el mismo papel que las palabras en una lengua natural -; 2) encontrar relaciones entre ellos que permitan construir la estructura de la cadena.

Japlage permite generar un procesador de lenguajes, traductor o compilador, a partir de su especificación por medio de un ATS y de la especificación del analizador léxico de su lenguaje de entrada, como se muestra en la figura 7.1.

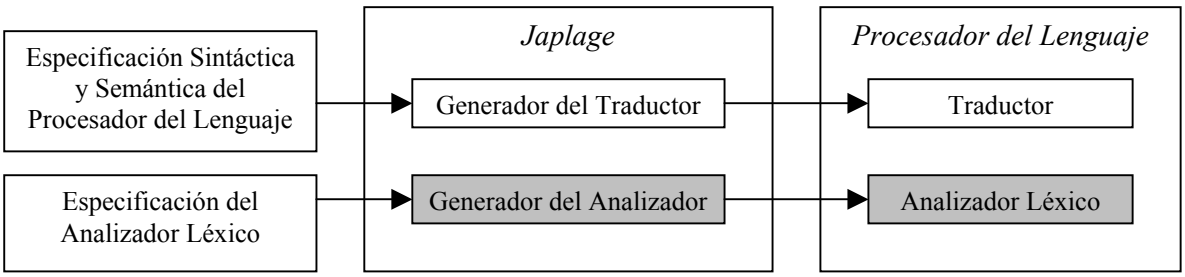


Figura 7.1: Entrada y Salida del Generador de Procesadores de Lenguajes.

La división de la cadena de entrada se denomina análisis lexicográfico y el proceso de encontrar relaciones entre los componentes léxicos se denomina análisis sintáctico. *Japlage* genera el procesador de lenguaje, traductor o compilador, a partir de la especificación del lenguaje. Esta es descripta a través de dos especificaciones: la del analizador lexicográfico y la del analizador sintáctico. La salida del generador es el código que implementa al procesador del lenguaje.

El generador de procesadores de lenguajes es, en esencia, un compilador que a partir de las especificaciones de entrada obtiene un procesador del lenguaje especificado. Este procesador de lenguaje podrá ser, por ejemplo, un compilador, que produzca programas ejecutables a partir de programas fuentes, o un interprete, una interface de usuario, que permita interactuar con un sistema de información mediante un lenguaje ad hoc. La salida del generador provee el sistema de tiempo de ejecución que ejecuta al ATS especificado en la entrada, denominado RTS *Run Time System.*

7.2 - Generador de procesadores de lenguajes *Japlage*

El generador de procesadores de lenguajes es una herramienta que dado un ATS genera automáticamente un traductor capaz de analizar sintáctica y semánticamente las cadenas que el ATS especifica.

El generador está estructurado en forma similar a la de un compilador. Sus módulos principal son: un módulo de análisis lexicográfico, un módulo de análisis sintáctico, un módulo de análisis semántico y otro de generación de código.

En la figura 7.2 se presenta el diseño de módulos del generador para mostrar esquemáticamente los módulos del sistema y sus relaciones.

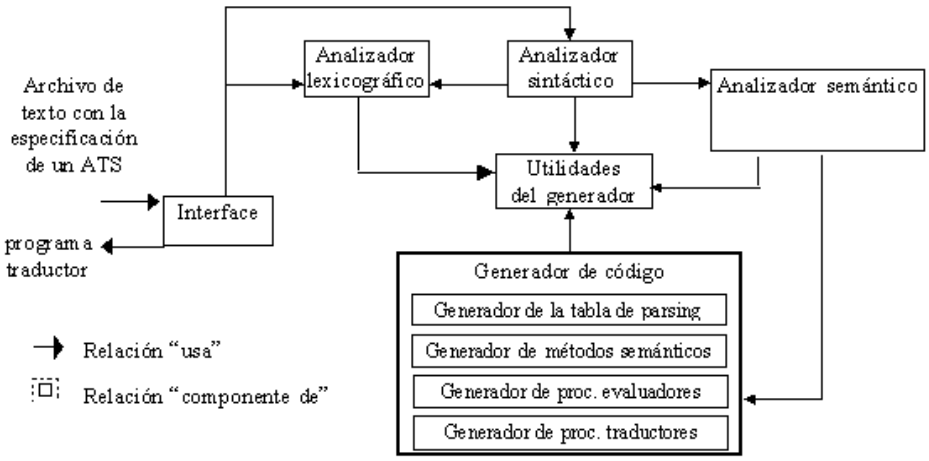


Figura 7.2: Módulos del generador de traductores.

Interface. Toma el requerimiento del usuario para la generación de un traductor y retorna el traductor generado o eventualmente el reporte de errores cometidos. Recibe un archivo de texto con la especificación de un ATS e inicializa a los analizadores lexicográfico y sintáctico. La salida producida por una compilación exitosa es un programa Java que implementa el traductor.

Analizador lexicográfico. Lee caracteres del archivo de texto que contiene la especificación de la gramática, reconoce lexemas y retorna tokens. Es utilizado por el módulo de interface que lo inicializa indicándole el archivo de lectura, y por el analizador sintáctico el cual hace los requerimientos de tokens.

Analizador sintáctico. Recibe el requerimiento del módulo de interface de analizar el archivo de entrada. Utiliza el modulo de análisis lexicográfico, del cual obtiene los tokens del archivo de entrada, y a medida que realiza el análisis de la cadena de entrada, invoca al analizador semántico.

Analizador semántico. implementa diversas verificaciones estáticas sobre la validez del ATS de entrada.

Generador de código. Este módulo es responsable de construir la tabla de parsing del traductor de salida, de generar el código Java de los procesos traductores y evaluadores, y de las otras componentes del traductor de salida que dependan del ATS de entrada. Finalmente debe integrar estos segmentos de código para conformar el traductor generado. El generador de código utiliza servicios del módulo de utilidades
Para construir la tabla de parsing LALR(1) se construye directamente el conjunto canónico de conjuntos de items LALR(1), sin la construcción previa del LR(1). Como la representación matricial de las tablas conduce a matrices ralas, se utiliza una representación que aprovecha esta característica para economizar espacio.

Utilidades del generador. En este módulo se agrupan todas las utilidades requeridas por los otros módulos.

7.2.1 - Clases del generador

El diseño de las clases del generador surge del anterior diseño de módulos. De esta manera se obtienen los grupos de clases: clases de análisis lexicográfico y sintáctico, clases de análisis semántico, clases de generación código, clases de interface y clases de utilidades.

7.2.2 - De análisis lexicográfico y sintáctico

En el diseño hay dos pares de analizadores, léxico y sintáctico. Un par de ellos actúa en tiempo de generación, ocupándose del parsing de la especificación de un ATS, mientras que el otro par está destinado a actuar en tiempo de ejecución del traductor generado. Consiguientemente se definen cuatro clases: 'LexicalATS' y 'ParserATS' que implementan el primer par de analizadores, y 'LexicalLex' y 'ParserLex' que implementan al segundo par.

7.2.3 - De análisis estático

La responsabilidad de estas clases consiste en analizar la validez del uso de atributos en las reglas de evaluación y traducción del ATS especificado y en las acciones asociadas a las expresiones regulares de la especificación del analizador lexicográfico. Por ejemplo, se controla que no se asocie a un símbolo un atributo que no posee, o que se utilice un atributo no declarado.

La representación interna de una regla del ATS es un objeto de la clase 'Rule', y una regla del analizador lexicográfico es un objeto de la clase 'RegExp', como se muestra en la figura 7.3.

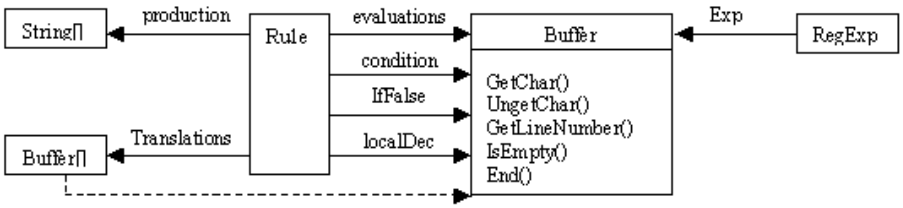


Figura 7.3: Representación interna de reglas de ATS y de un analizador lexicográfico.

Un objeto de la clase 'Rule' posee un arreglo de strings que indican el tipo de cada símbolo de la producción, una lista de buffers, cada uno de los cuales posee el código de una acción de traducción, un buffer con las declaraciones locales, un buffer con el código de las reglas evaluación, un buffer con el código de la condición y otro con el código que debe ser ejecutado cuando la condición es falsa.

Un objeto de la clase 'RegExp', simplemente posee un buffer que contiene el código asociado a la expresión regular.

Para realizar las verificaciones mencionadas se define una clase 'ScanRule' cuya función es reconocer los códigos contenidos en un objeto 'Rule', y una clase 'ScanExp'

responsable de controlar el código contenido en el buffer de un objeto 'RegExp'. Debido a que sus métodos deben ejecutarse concurrentemente, son subclases de la clase Java 'Thread'.

La comunicación entre las clases de análisis sintáctico - responsables de analizar la entrada y crear los objetos de 'Rule' y 'RegExp' - y las clases de reconocimiento, se establece a través de una clase 'Scanner' que actúa de interface entre ellas. Esta clase provee métodos de acceso a 'ScanRule' y 'ScanExp' conteniendo además los parámetros generales de reconocimiento, como una tabla de los tokens definidos en la gramática y una tabla de los tipos de símbolos de la gramática.

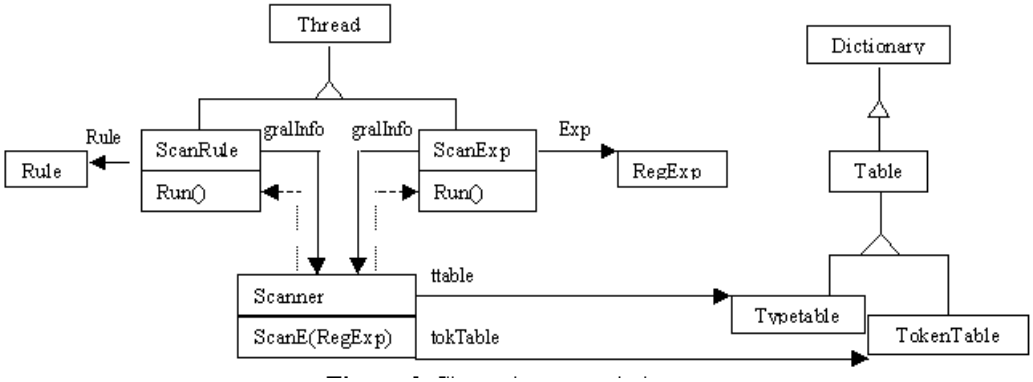


Figura 7.4: Clases de Reconocimiento.

7.2.4 - De generación de código

Estas clases - figura 7.5 - son responsables de generar código especifico o de integrar código ya generado en la definición de una clase. La salida de estas clases es almacenada en archivos de nombre estándar de manera tal que la comunicación entre una clase que genera un determinado código y otra que lo utiliza se establece a través de los archivos creados.

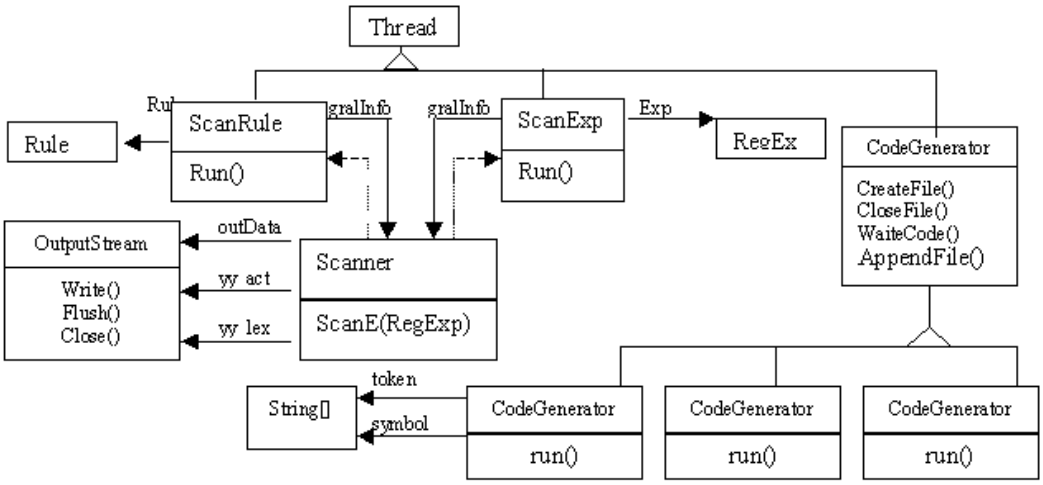


Figura 7.5: Clases de Generación de Código.

Por otra parte, la generación de código podría iniciarse cuando aun otras acciones de análisis sintáctico o semántico están pendientes, por lo tanto, es conveniente que se permita su ejecución concurrente con otras acciones del generador. En consecuencia las clases de generación de código son subclases de la clase de Java 'Thread'.

Los elementos comunes a la generación de código son encapsulados en la clase 'CodeGenerator', y las demás clases son extensiones de ella. Esta clase provee métodos que permite el manejo de archivos.

7.2.5 - De interface

La interface de nivel superior del generador es provista por una clase denominada 'Japlage' (por Java compiler of compilers). Esta clase define una interface uniforme para la funcionalidad del generador y actúa como fachada del subsistema, ya que ofrece a los usuarios una interface simple y fácil de utilizar.

Los usuarios de la herramienta no tienen acceso a los objetos del subsistema directamente, sino que se comunican con él enviando requerimientos a 'Japlage', quien dirige los mismos a los objetos correspondientes del generador.

La clase 'Japlage' es responsable de inicializar la generación de un traductor: debe recibir los archivos en donde se encuentran la especificación del ATS y la especificación del analizador lexicográfico. Además, la clase 'Japlage' es responsable de crear objetos de la clase 'Parser' y 'Lexical' e iniciar la generación del traductor invocando al método parse().

En la siguiente figura se muestra el esquema de todas las clases del generador.

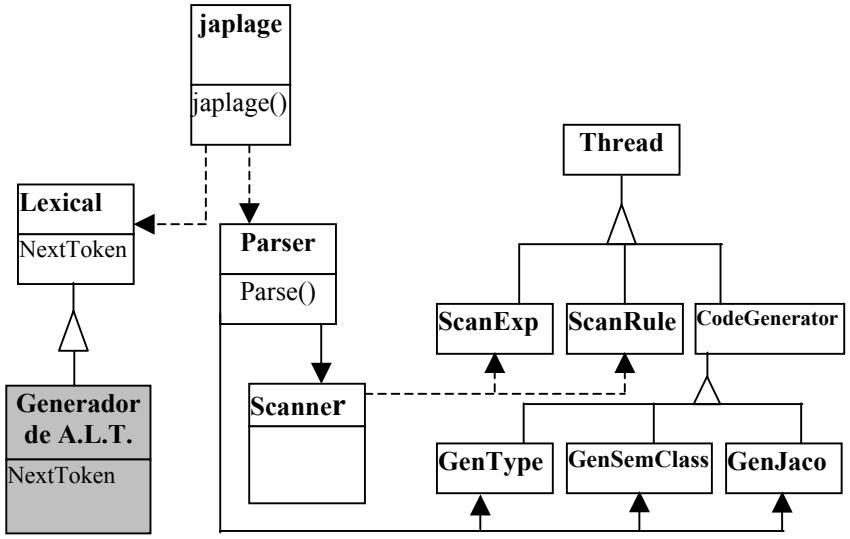


Figura 7.6: Incorporación del Generador de Analizadores Léxicos Traductores a Japlage[♣].

7.3 – Interacción de las Herramientas Generadas

El procesador de lenguajes generado por *japlage* consta de un parser y de un analizador léxico traductor. Estos interactúan de manera similar a par *Lex-Yacc*. Es decir, el parser genera al analizador léxico y hace el requerimiento de tokens invocando al método Next_token( ). Puede acceder, el parser, a las variables de entorno del analizador – para más detalles ver capítulo 3 –.

[♣] En sombreado se encuentra el generador de analizadores léxicos traductores.

Conclusiones

Se ha obtenido una primer versión de un Generador de Analizadores Léxicos basado en ETL's. El lenguaje del generador 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's, 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 procesan una cadena a en tiempo O(|a |).

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

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

No hay que recalcar solamente como conclusiones relevantes de este trabajo de grado aquellas que se desprenden de la implementación de la herramienta, sino también, es muy importante resaltar el aporte, tanto en lo académico como en lo profesional, que significó interactuar en el ámbito de un grupo de investigación.

Otra conclusión significativa es el papel desempeñado como nexo entre una investigación teórica y los usuarios – programadores, investigadores, entre otros –. Es decir, con el producto obtenido se logró aplicando la teoría desarrollada una herramienta que simplifica y mejora la productividad de los usuarios. Consideramos que este nexo esta en vías de desarrollo y que el ámbito científico tendría que promocionar su desarrollo. Con este aporte, nuestro pequeño granito de arena, creemos que contribuimos en la construcción de este escalón de vital importancia para el desarrollo de cualquier ciencia.[♣]

Trabajos Futuros

Sería interesante encontrar e implementar algoritmos eficientes para traducir cualquier ETU. Como así también, extender el formalismo para permitir, asociar no solo traducciones a los símbolos del lenguaje, sino también, asociar traducciones a subexpresiones de las ETL's.

Extender el lenguaje de especificación para así abarcar todos los operadores utilizados por *Lex* – por ejemplo ^ y / –.

[♣] Nótese que esta es una opinión personal de los autores.

Extender el lenguaje de especificación – y realizar las implementaciones necesarias – para incorporar utilización conjunta de estados y expresiones regulares para especificar los símbolos del lenguaje siguiendo el estilo de las especificaciones *Jlex* – ver sección 5.1.2 –.

Anexo 1

Especificación JLex para el Generador de Analizadores Léxicos Traductores

```
/*************************************************************************/
/*       Especificacion JLex para un generador de analizadores lexicos       */
/*                        con expresiones regulares traductoras                        */
/*************************************************************************/
import java_cup.runtime.Symbol;
import java.lang.System;
import java.io.*;
import utils.Yytoken;
import utils.Errors;

%%

%{
   //Flags que indican  si están balanceados { y /*. (Si 0, No otro)
  private int comment_count = 0;
  private int jaction_count = 0;
  private int branch_count = 0;
  private int jdecl_end = 0;

   //Flag que indica si esta analizando la sección de declaración de macros:(Si 1, No 0)
  private int decl_macro = 1;

   //Buffer que contiene el string que se debe insertar en el token corriente.
   //El cual en ciertas ocasiones no coincide con el lexema reconocido (yytext).
  private String buffer;
%}

%eofval{
   if (comment_count > 0) Errors.error(Errors.E_ENDCOMMENT);
   if (jaction_count > 0) Errors.error(Errors.E_ENDCODE);
   if (branch_count > 0)  Errors.error(Errors.E_ENDCODE);
   return new Symbol(sym.EOF);
%eofval}

%cup
%line
%char
%full
%notunix
%yyeof
```

```
%state COMMENT   JDECL   JDECL1   JACTION          IJACTION        JDECL1_EOF
%state BRANCH    JDECL_END       DECL_MACRO   MARKS
%state JDECL_INIC JDECL1_INIC    JDECL_ERROR  JDECL1_ERROR  JDECL_EOF

ALPHA = [A-Za-z]
DIGIT = [0-9]
COMMENT_TEXT = ([ ^/*\n]|[ ^*\n]"/"[ ^*\n]|[ ^/\n]"*"[ ^/\n]|"*"[ ^/\n]|"/"[ ^*\n])*
JACTION_SOURCE = ([ ^\{\}])*
BRANCH_SOURCE = ([ ^\{\}])*

%%

<YYINITIAL,COMMENT> (\ |\t|\n|\b)+ { }

<YYINITIAL> "|" { return new Symbol(sym.OR,new Yytoken("OR",yytext(),yyline)); }
<YYINITIAL> "*" { return new Symbol(sym.TIMES,new Yytoken("TIMES",yytext(),yyline)); }
<YYINITIAL> "+" { return new Symbol(sym.PLUS,new Yytoken("PLUS",yytext(),yyline)); }
<YYINITIAL> "?" { return new Symbol(sym.QUESTION,
                    new Yytoken("QUESTION",yytext(),yyline)); }
<YYINITIAL> "(" { return new Symbol(sym.LPARENT,
                    new Yytoken("LPARENT",yytext(),yyline)); }
<YYINITIAL> ")" { return new Symbol(sym.RPARENT,
                    new Yytoken("RPARENT",yytext(),yyline)); }
<YYINITIAL> "[" { return new Symbol(sym.LBRACKET,
                    new Yytoken("LBRACKET",yytext(),yyline)); }
<YYINITIAL> "]" { return new Symbol(sym.RBRACKET,
                     new Yytoken("RBRACKET",yytext(),yyline)); }
<YYINITIAL> "-" { return new Symbol(sym.SCORE,new Yytoken("SCORE",yytext(),yyline)); }
<YYINITIAL> "," { return new Symbol(sym.SEMICOLON,
                    new Yytoken("SEMICOLON",yytext(),yyline)); }
<YYINITIAL> "." { return new Symbol(sym.COLON,new Yytoken("COLON",yytext(),yyline)); }

<YYINITIAL> \\. { if (Errors.is_valid_symbol(yytext()))
                      return new Symbol(sym.BAR_CHAR,
                      new Yytoken("BAR_CHAR",yytext(),yyline));
                    else Errors.error1(Errors.E_UNMATCHED_SYMBOL,yytext(),yyline); }

<YYINITIAL> "}" {Errors.error1(Errors.E_STARTCODE,"",yyline);}


<YYINITIAL> \"  {yybegin(MARKS);return new Symbol(sym.MARKS_TEXT,
                     new Yytoken("COMILLA",yytext(),yyline));}
<MARKS> [^\"] {return new Symbol(sym.CHAR,new Yytoken("CHAR",yytext(),yyline));}
<MARKS> \" {yybegin(YYINITIAL);return new Symbol(sym.MARKS_TEXT,
                new Yytoken("COMILLA",yytext(),yyline));}


<YYINITIAL> ^"%{"   {decl_macro=0;buffer=new String();yybegin(JDECL);}
<JDECL> (.*\n|\n)   {buffer=buffer+yytext();yybegin(JDECL1);}
<JDECL> ^"%}"  {decl_macro=1;yybegin(YYINITIAL);
        return new Symbol(sym.JAVA_DECL,new Yytoken("JAVA_DECL",buffer,yyline)); }

<JDECL1> ^"%}"  {decl_macro=1;yybegin(YYINITIAL);return new
Symbol(sym.JAVA_DECL,new Yytoken("JAVA_DECL",buffer,yyline)); }
<JDECL1> (.|\n)   {buffer=buffer+yytext();yybegin(JDECL);}
```

```
<YYINITIAL> ^"%init{"   {decl_macro=0;buffer=new String();yybegin(JDECL_INIC);}
<JDECL_INIC> (.*\n|\n)   {buffer=buffer+yytext();yybegin(JDECL1_INIC);}
<JDECL_INIC> ^"%init}"  {decl_macro=1;yybegin(YYINITIAL);
return new Symbol(sym.JAVA_DECL_INIT,new Yytoken("JAVA_DECL_INIC",buffer,yyline));}

<JDECL1_INIC> ^"%init}"  {decl_macro=1;yybegin(YYINITIAL);return new
Symbol(sym.JAVA_DECL_INIT,new Yytoken("JAVA_DECL_INIC",buffer,yyline)); }
<JDECL1_INIC> (.|\n)   {buffer=buffer+yytext();yybegin(JDECL_INIC);}



<YYINITIAL> ^"%eof{"   {decl_macro=0;buffer=new String();yybegin(JDECL_EOF);}
<JDECL_EOF> (.*\n|\n)   {buffer=buffer+yytext();yybegin(JDECL1_EOF);}
<JDECL_EOF> ^"%eof}"  {decl_macro=1;yybegin(YYINITIAL);
return new Symbol(sym.JAVA_DECL_EOF,new Yytoken("JAVA_DECL_EOF",buffer,yyline)); }

<JDECL1_EOF> ^"%eof}"  {decl_macro=1;yybegin(YYINITIAL);return new
Symbol(sym.JAVA_DECL_EOF,new Yytoken("JAVA_DECL_EOF",buffer,yyline)); }
<JDECL1_EOF> (.|\n)   {buffer=buffer+yytext();yybegin(JDECL_EOF);}


<YYINITIAL> ^"%error{"   {decl_macro=0;buffer=new String();yybegin(JDECL_ERROR);}
<JDECL_ERROR> (.*\n|\n)   {buffer=buffer+yytext();yybegin(JDECL1_ERROR);}
<JDECL_ERROR> ^"%error}"  {decl_macro=1;yybegin(YYINITIAL);
                           return new Symbol(sym.JAVA_DECL_ERROR,
                           new Yytoken("JAVA_DECL_ERROR",buffer,yyline)); }

<JDECL1_ERROR> ^"%error}"  {decl_macro=1;yybegin(YYINITIAL);return new
Symbol(sym.JAVA_DECL_ERROR,new Yytoken("JAVA_DECL_ERROR",buffer,yyline)); }
<JDECL1_ERROR> (.|\n)   {buffer=buffer+yytext();yybegin(JDECL_ERROR);}


<YYINITIAL> "ACTION{"({JACTION_SOURCE}|\n)*
        { buffer=new String(yytext());yybegin(JACTION); jaction_count= jaction_count+1;}

<YYINITIAL> ^"ACTION{"({JACTION_SOURCE}|\n)*
        { buffer=new String(yytext());yybegin(JACTION); jaction_count= jaction_count+1;}
<JACTION> "{" { buffer=buffer+yytext();jaction_count = jaction_count + 1; }
<JACTION> "}" { buffer=buffer+yytext();
        jaction_count = jaction_count - 1;
        Errors.assert(jaction_count >= 0);
        if (jaction_count == 0) {
                yybegin(YYINITIAL);
                buffer=buffer.substring(6,buffer.length());
                return new Symbol(sym.JAVA_ACTION,
                new Yytoken("JAVA_ACTION",buffer,yyline));
                }
        }
<JACTION> ({JACTION_SOURCE}|\n)*    {buffer=buffer+yytext(); }


<YYINITIAL> ^"INIT{"({JACTION_SOURCE}|\n)*  { buffer=new
String(yytext());yybegin(IJACTION); jaction_count= jaction_count+1;}
<IJACTION> "{" { buffer=buffer+yytext();jaction_count = jaction_count + 1; }
<IJACTION> "}" { buffer=buffer+yytext();
        jaction_count = jaction_count - 1;
        Errors.assert(jaction_count >= 0);
        if (jaction_count == 0) {
```

```
                yybegin(YYINITIAL);
                buffer=buffer.substring(4,buffer.length());
                return new Symbol(sym.INIT_JAVA_ACTION,
                new Yytoken("INIT_JAVA_ACTION",buffer,yyline));
                }
        }
<IJACTION> ({JACTION_SOURCE}|\n)*    {buffer=buffer+yytext(); }


<YYINITIAL> "{" {buffer=new
String(yytext());yybegin(BRANCH);branch_count=branch_count+1;}

<BRANCH> {ALPHA}({ALPHA}|{DIGIT}|_)*"}" { buffer=buffer+yytext();
                                          buffer=buffer.substring(1,buffer.length()-1);
                                          yybegin(YYINITIAL);branch_count=0;
                                          return new Symbol(sym.USE_MACRO,
                                         new Yytoken("USE_MACRO",buffer,yyline)); }
<BRANCH> "{" { buffer=buffer+yytext();branch_count = branch_count + 1; }
<BRANCH> "}" { buffer=buffer+yytext();
        branch_count = branch_count - 1;
        Errors.assert(branch_count >= 0);
        if (branch_count == 0) {
                yybegin(YYINITIAL);
                return new Symbol(sym.JSOURCE_FINAL_EXP_REG,
                new Yytoken("JSOURCE_FINAL_EXP_REG",buffer,yyline));
                }
        }
<BRANCH> ({BRANCH_SOURCE}|\n)*    {buffer=buffer+yytext(); }


<YYINITIAL> ^"%%"  {if (jdecl_end==0) {decl_macro=0;jdecl_end=1;return new
Symbol(sym.DOUBLE_PERCENT,new Yytoken("DOUBLE_PERCENT",yytext(),yyline)); }
                        else {buffer=new String();yybegin(JDECL_END);
                                }
                    }
<JDECL_END> (.|\n)* {return new Symbol(sym.JDECL_END,
                        new Yytoken("JDECL_END",yytext(),yyline));}

<YYINITIAL> ^{ALPHA} {if (decl_macro==1) {buffer=new
String(yytext());yybegin(DECL_MACRO);}
                        else {return new Symbol(sym.CHAR,
                                new Yytoken("CHAR",yytext(),yyline));}      }

<DECL_MACRO> ({ALPHA}|{DIGIT}|_)*(\ |\t) {buffer=buffer+yytext();
                        buffer=buffer.substring(0,buffer.length()-1);
                       yybegin(YYINITIAL);
return new Symbol(sym.NAME_MACRO,new Yytoken("NAME_MACRO",buffer,yyline));}
<DECL_MACRO> .  {yybegin(YYINITIAL); Errors.error(Errors.E_UNMATCHED_NAME);}


<YYINITIAL> .  {if (Errors.is_valid_symbol(yytext()))
                return new Symbol(sym.CHAR,new Yytoken("CHAR",yytext(),yyline));
                else Errors.error1(Errors.E_UNMATCHED_SYMBOL,yytext(),yyline);    }

<YYINITIAL> "/*" { yybegin(COMMENT); comment_count = comment_count + 1; }
```

```
<COMMENT> "/*" { comment_count = comment_count + 1; }
<COMMENT> "*/" {
        comment_count = comment_count - 1;
        Errors.assert(comment_count >= 0);
        if (comment_count == 0) {
                yybegin(YYINITIAL);    } }
<COMMENT> {COMMENT_TEXT} { }


<YYINITIAL> "*/" {Errors.error1(Errors.E_STARTCOMMENT,"",yyline);}
```

Anexo 2

Especificación Cup para el Generador de Analizadores Léxicos Traductores

```
/*************************************************************************/
/*          Especificación Cup para un generador de analizadores         */
/*                 léxicos con expresiones regulares traductoras         */
/*************************************************************************/
import java_cup.runtime.*;
import utils.*;

action code {:
//Variables globales de usuario.

   Table_macros table_macros = new Table_macros(); //Contiene los arboles
                                                   //sintactico de los macros.
   Utility_actions file_actions = new Utility_actions();//Construye la clase
                                   //yylexertraductions con las traducciones
                                   //asociadas a las expresiones regulares.
   Table_expresions table_expresions = new Table_expresions();//Contiene
                                   //los arboles sintacticos, los first, last y
                                   //followpos, las posiciones y los automatas
                                   //asociados a las expresiones regulares.
   String code_action; //Auxiliar que contiene la traduccion asociada a un simb.
   String javadeclerror;//Auxiliar que contiene el codigo java del usuario en
                        //caso de error.
   String javadecleof; //Auxiliar que contiene el codigo java del usuario para
                        //cuando se retorna eof.
   GenConcreteLexical Concrete_lex; //genera el archivo con el lexer especificado.
            :};

parser code {:
   public void syntax_error(Symbol cur_token)
          {
            System.out.println("Syntax error : (line : "+ (((Yylex)getScanner()).yyline+1)+") –
                                                  Input string : "+((Yylex)getScanner()).yytext());
          }

   public void Unrecovered_syntax_error(Symbol cur_token)
          {
            System.out.println("Couldn't continue parser");
          }
```

```
    public void report_error(String message, Object info)
        {
          System.out.println("Syntax error : (line : "+ (((Yylex)getScanner()).yyline+1)+") –
                                                    Input string : "+((Yylex)getScanner()).yytext());
        }
    public void report_fatal_error(String message, Object info)
        {
          System.out.println("Couldn't continue parser");
        }
:};
/* Terminales (tokens retornados por el analizador lexico). */



terminal            DOUBLE_PERCENT, COLON, SEMICOLON, PLUS, TIMES;
terminal            QUESTION, LPARENT, RPARENT, MARKS_TEXT, LBRACKET,
RBRACKET;
terminal Yytoken  OR, SCORE, CHAR, BAR_CHAR;
terminal Yytoken  JAVA_DECL, JAVA_ACTION, INIT_JAVA_ACTION, USE_MACRO;
terminal Yytoken  JSOURCE_FINAL_EXP_REG, JDECL_END, NAME_MACRO;
terminal Yytoken  JAVA_DECL_INIT, JAVA_DECL_EOF, JAVA_DECL_ERROR;

/* No terminales */

non terminal          gram_lex, decl_macro, decl_macro1,body_lex;
non terminal          decl_java, init_java;
non terminal Node_tree exp_reg, exp_reg1, exp_reg2;
non terminal Node_tree exp_reg_macro, exp_reg1_macro, exp_reg2_macro;
non terminal Node_tree rango, rango1, words, symbol;
non terminal          decl_java1, decl_java2, decl_java3, decl_java4;
non terminal String    bchar, j_action;

/* Gramatica */

gram_lex ::=    {: System.out.println("Processing first section -- user code."); :}
          decl_java
              {: System.out.println("Processing second section -- lexical rules."); :}
            decl_macro DOUBLE_PERCENT body_lex
              {: System.out.println("Processing third section -- user classes."); :}
            JDECL_END:j
           {:
                    if (Errors.get_error()==false)
                    {
                      System.out.println("Check lexical, sintactic and semantic sucessfull.");
                      table_expresions.const_automaton();
                      if (Errors.get_error()==false)
                      {  String aut = table_expresions.toArray_automaton();
                         Automaton aut_det = table_expresions.get_aut_det();
                           Concrete_lex = new GenConcreteLexical(j.get_text(),
                                    file_actions.get_file_action(),aut, javadecleof, javadeclerror, aut_det);
                            System.out.println("Code generate sucessfull.");
                      }
                      else  System.out.println("No code generate.");
                    }
                    else   System.out.println("No code generate.");
                     :}
           ;
```

```
/* Declaraciones en codigo Java */
decl_java ::=   decl_java1 decl_java2 decl_java3 decl_java4
            ;

/* Declaraciones de Variables en Java*/
decl_java1 ::=  JAVA_DECL:j  {: file_actions.print_decl_java(j.get_text());:}
            |
            ;

/* Inicializaciones de Variables en Java*/
decl_java2::=  JAVA_DECL_INIT:j {: file_actions.print_constructor(j.get_text());:}
            |
            ;

decl_java3 ::=  JAVA_DECL_EOF:j {: javadecleof = j.get_text(); :}
            |  {: javadecleof = " "; :}
            ;

decl_java4 ::=  JAVA_DECL_ERROR:j {: javadeclerror = j.get_text(); :}
            | {: javadeclerror = " "; :}
            ;

/* Declaraciones de Macros en el Analizador*/
decl_macro ::=  decl_macro1                 {:Node_tree.reset_position();:}
             |
             ;

/* Los Macros en el Analizador*/
decl_macro1 ::=  NAME_MACRO:m   exp_reg_macro:e
                    {:table_macros.insert(m.get_text(),e,m.get_line());:}
          decl_macro1
              | NAME_MACRO:m   exp_reg_macro:e
                    {:table_macros.insert(m.get_text(),e,m.get_line());:}
             ;

/* Cuerpo del Analizador (inicializaciones de variables, seguido de expresiones
regulares, seguido codigo Java para ejecutarse en cada expresion regular*/

body_lex ::=
            init_java
            exp_reg:e JSOURCE_FINAL_EXP_REG:j
                                    {: Utility_actions.reset_num_act();
                                       file_actions.print_final_action(j.get_text());
                                       Utility_actions.add_num_exp();
                                       file_actions.print_end_switch();
                                       table_expresions.insert(new Node_tree(
Etiquette_node.CONCAT,e,new Node_tree(Etiquette_node.CHAR,'\242')));
                                       Node_tree.reset_position();
                                    :}
            body_lex

           |        init_java exp_reg:e JSOURCE_FINAL_EXP_REG:j
                                    {:
                                      table_expresions.insert(new Node_tree
(Etiquette_node.CONCAT,e,new Node_tree(Etiquette_node.CHAR,'\242')));
                                       Node_tree.reset_position();
                                   file_actions.print_end_switch();
```

```
                                    file_actions.print_final_action(j.get_text());
                                    file_actions.print_final();
                                   :}
            ;

/* Acciones que se ejecutan para inicializar variables */
init_java ::=  INIT_JAVA_ACTION:j {:file_actions.print_case_exp();
                                    file_actions.print_init_action(j.get_text());
                                    table_expresions.have_init(1);
                                   :}
            |                     {:file_actions.print_case_exp();
                                   table_expresions.have_init(0);
                                  :}
            ;

/* Definicion de expresiones regulares para los macros */
/* Expresiones regulares: Composicion */
exp_reg_macro ::=  exp_reg1_macro:e1 exp_reg_macro:e2
                                    {:RESULT = new Node_tree(Etiquette_node.CONCAT,e1,e2);:}
              | exp_reg1_macro:e
                                    {:RESULT = e;:}
                ;

/* Expresiones regulares: Disyuncion */
exp_reg1_macro ::=  exp_reg2_macro:e1 OR exp_reg1_macro:e2
                                    {:RESULT = new Node_tree(Etiquette_node.OR,e1,e2);:}
              | exp_reg2_macro:e
                                    {:RESULT = e;:}
              ;

/* Expresiones regulares: Rangos, caracteres, cadenas, uso de macros, etc. */
exp_reg2_macro ::=  LPARENT exp_reg_macro:e RPARENT symbol:s
                                          {: RESULT=Node_tree.add_tree(s,e);:}
              | BAR_CHAR:t symbol:s
                                          {: RESULT=Node_tree.add_tree(s,new Node_tree(
                                    Etiquette_node.CHAR,Utility.toASCII(t.get_text())));:}
              | CHAR:t symbol:s
                                          {: RESULT=Node_tree.add_tree(s,new Node_tree
                                    (Etiquette_node.CHAR,t.get_text().toCharArray()[0]));:}
              | MARKS_TEXT words:e MARKS_TEXT symbol:s
                                          {: RESULT=Node_tree.add_tree(s,e);:}
              | LBRACKET rango:e RBRACKET symbol:s
                                          {: e.add_action("","",0,0);
                                           RESULT= Node_tree.add_tree(s,e);:}
              | USE_MACRO:m  symbol:s
                                          {:RESULT = Node_tree.add_tree
                                    (s,table_macros.get_tree(m.get_text(),m.get_line()));:}
               | COLON symbol:s
                                          {: RESULT=Node_tree.add_tree
                                    (s,Node_tree.create_tree_colon("","",0,0));:}
              ;

/* Definicion de expresiones regulares para el cuerpo del analizador */
/* Expresiones regulares: Composicion */

exp_reg ::=  exp_reg1:e1  exp_reg:e2
                    {:RESULT = new Node_tree(Etiquette_node.CONCAT,e1,e2);:}
```

```
            | exp_reg1:e
                    {:RESULT = e;:}
            ;

/* Expresiones regulares: Disyuncion */
exp_reg1 ::=  exp_reg2:e1 OR exp_reg1:e2
                    {:RESULT = new Node_tree(Etiquette_node.OR,e1,e2);:}
            | exp_reg2:e
                    {:RESULT = e;:}
            ;

/* Expresiones regulares: Rangos, caracteres, cadenas, uso de macros, etc. */
exp_reg2 ::=  LPARENT exp_reg:e RPARENT symbol:s
                                {: RESULT=Node_tree.add_tree(s,e);:}
            | BAR_CHAR:t j_action:j symbol:s
                                {: RESULT=Node_tree.add_tree(s,new Node_tree
                                (Etiquette_node.CHAR,Utility.toASCII(t.get_text()),j,code_action,
                                Utility_actions.get_num_exp(),Utility_actions.get_num_act()-1));:}
            | CHAR:t j_action:j symbol:s
                                {: RESULT=Node_tree.add_tree(s,new Node_tree
                                (Etiquette_node.CHAR,t.get_text().toCharArray()[0],j,code_action,
                                Utility_actions.get_num_exp(),Utility_actions.get_num_act()-1));:}
            | MARKS_TEXT words:e MARKS_TEXT symbol:s
                                {: RESULT=Node_tree.add_tree(s,e);:}
            | LBRACKET rango:e RBRACKET j_action:j symbol:s
                                {: e.add_action(j,code_action,Utility_actions.get_num_exp(),
                            Utility_actions.get_num_act()-1);RESULT=Node_tree.add_tree(s,e);:}
            | USE_MACRO:m symbol:s
                                {:RESULT = Node_tree.add_tree
                            (s,table_macros.get_tree(m.get_text(),m.get_line()).reasing_pos());:}
            | COLON j_action:j symbol:s
                                {: RESULT=Node_tree.add_tree(s,
                            Node_tree.create_tree_colon(j,code_action,Utility_actions.get_num_exp(),
                            Utility_actions.get_num_act()-1));:}
            ;

/* Definicion de cadenas */
words ::=  words:w CHAR:t
                    {: RESULT=Node_tree.add_tree_word(w, new Node_tree
                    (Etiquette_node.CHAR,t.get_text().toCharArray()[0]));:}
          | CHAR:t
                    {:RESULT= new Node_tree(Etiquette_node.CHAR,t.get_text().toCharArray()[0]);:}
          ;

/* Definicion de los simbolos de las expresiones regulares: +,*,? */
symbol ::=  PLUS  {: RESULT= new Node_tree(Etiquette_node.PLUS); :}
          | TIMES {: RESULT= new Node_tree(Etiquette_node.TIMES); :}
          | QUESTION {: RESULT= new Node_tree(Etiquette_node.QUESTION,null); :}
          | {:RESULT=null; :}
          ;

/* Definicion de las acciones */
j_action ::=  JAVA_ACTION:j
                    {: RESULT=Utility_actions.get_name_action();
                      file_actions.print_name_action(j.get_text());
                      Utility_actions.add_num_act();
                      code_action=j.get_text();
                    :}
```

```
            |       {:RESULT=" ";code_action=" ";:}
            ;

/* Definicion de los rangos */
rango ::=  bchar:t
            {:  RESULT= new Node_tree(Etiquette_node.CHAR,t.toCharArray()[0]);:}
          rango1:r
                    {:  if (r!=null) RESULT=new Node_tree(Etiquette_node.OR,RESULT,r);:}

        | bchar:t SCORE:s bchar:tt
                    {: RESULT= Node_tree.create_tree_rango
                    (t.toCharArray()[0],tt.toCharArray()[0],s.get_line());:}
             rango1:r
                 {: if (r!=null) RESULT=new Node_tree(Etiquette_node.OR,RESULT,r);:}

             | SEMICOLON rango1:r
                  {: RESULT=r;:}

            | COLON
                  {:   RESULT=Node_tree.create_tree_colon(" "," ",0,0);  :}
             rango1:r
                       {: if (r!=null) RESULT=new Node_tree(Etiquette_node.OR,RESULT,r);:}
           ;

bchar::=   CHAR:t
             {: RESULT=t.get_text(); :}
        | BAR_CHAR:t
              {: RESULT = String.valueOf(Utility.toASCII(t.get_text())); :}
       ;

rango1 ::=  rango:r
                    {:RESULT=r;:}
             |
                {:RESULT=null;:}
            ;
```

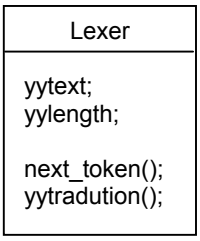
Anexo 3

Notación utilizada en el diseño

La notación aquí descripta es tomada del libro Design Patterns [Gam95] y se escogida por la expresividad y claridad de los esquemas con ella expresados.

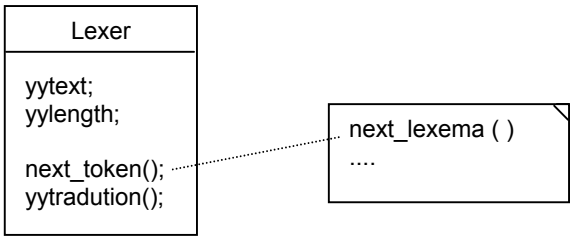
3.1 - Descripción de clases

Las clases se denotan con un dibujo rectangular en donde se indica el nombre de la clase y separados por una línea, los atributos y metodos mas importantes de una clase. Por ejemplo:



es una clase llamada Lexer, con atributos yytext y yylength y métodos next_token y yytraduction.

Cuando se desea describir algún aspecto de la implementación de un método de una clase, se utiliza el siguiente dibujo:

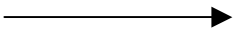


que expresa que el método next_token( ) invoca al método next_lexema( )

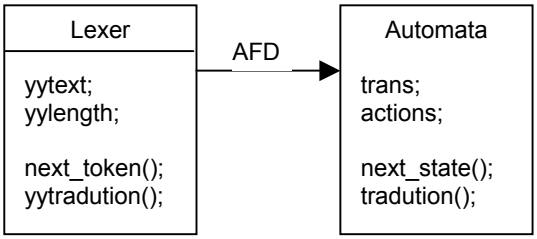
3.1 - Descripción de relaciones entre clases

Las diferentes relaciones entre clases se expresan con distintas flechas. Se presentan aquí tres tipos de relaciones.

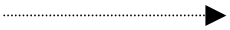
La relación de **composición** entre clases, es decir, la relación que se establece cuando una clase tiene como campo o componente un objeto de otra clase, se denota con la siguiente flecha:



La clase que se encuentra en el origen de la flecha posee un objeto de la clase que se encuentra en el extremo de la flecha. Ocasionalmente puede escribirse sobre la flecha el nombre del atributo que da lugar a la relación. Por ejemplo, en el siguiente esquema se utiliza una relación de composición entre clases:

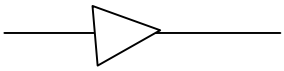


La relación de **uso** entre clases, que se establece cuando dentro de una clase se accede a métodos y campos de otra clase, se denota mediante la flecha:



en donde, la clase del origen accede a los elementos de la clase en el extremo.

Finalmente, la relación de herencia se expresa utilizando la siguiente flecha:



en cuyo origen se coloca la relación que hereda.

Anexo 4

Código del Analizador Léxico Generado para el Ejemplo del Capítulo 6

```
/*************************************************************/
/*       Analizador Lexico generado por JTLex                */
/*************************************************************/

//Comienza seccion de declaraciones de usuario
import java.io.*;


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();
    }
  }
}

//Clase que contine los metodos para ejecutar las traducciones
public class yylexertraductions
{
//metodo que ejecuta las traducciones
public void yyexec_traduc(int yyexp, int yyact)
{          switch (yyexp)
           {
            case 1:
                    switch (yyact)
                              {
                              case -1: break;
                              case 1:{i++;}
                                        break;
                              case 2:{i++;}
                                        break;
                              }
```

```
                    break;
                case 2:
                    switch (yyact)
                            {
                            case -1: break;
                            case 1:{i++;}
                                    break;
                            case 2:{i++;}
                                    break;
                            }
                    break;
            }
}

//metodo que ejecuta las acciones iniciales
public void yyexec_inic(int yyexp)
{           switch (yyexp)
            {
            case -1: break;

            }
}

//metodo que ejecuta las acciones finales
public int yyexec_end(int yyexp)
{           switch (yyexp)
            {
             case 1:{System.out.println("ETL 1");}
             case 2:{System.out.println("ETL 2");}
            }
return yyexp;
}

//Atributos definidos por el usuario

   int i;

//metodo que iniciliza los  atributos de usuario
 public void yyinic_lexer_tradu()
{
    i=0;
}


//Acceso a los atributos de entorno desde la clase yylexertraductions
   private Lexer L = new Lexer();
   public String yytext()
   { return L.yytext();
   }

   public char yytextchar()
   { return L.yytextchar();
   }

   public boolean yyEOF()
   { return L.yyEOF();
   }
```

```
    public int yylength()
    { return L.yylength();
    }

    public boolean yywrap()
    { return L.yywrap();
    }

    public int yyline()
    { return L.yyline();
    }

    public int yychar()
    { return L.yychar();
    }
    public Symbol yylval()
    { return L.yylval();
    }
    public void yylval_update(Symbol s)
    { L.yylval = s;
    }

}//Fin de la clase yylexertraductions

//yylval es de tipo Symbol. Utilizada para retornar el valor de los token.
public class Symbol {

  public Symbol(int id, Object o)
  {//constructor de la clase.
    sym = id;
    value = o;
  }

  public Symbol(int id)
  {//constructor de la clase
    sym = id;
  }

  public int sym; // Numero de token
  public Object value; // Valor del token

  public String toString() { return "#"+sym; }
}

//Analizador Lexico Traductor
public class Lexer
{
//definicion de variables de entorno

        private char [] inputString;
        private yylexertraductions yylext;
        private int index_end;
        static private boolean yyEOF;
        static private boolean yywrap;
        static private boolean yylambda;
        static private String yytext;
        static private char yytextchar;
        static private int yylength;
```

```
        static private int yyline;
        static private int yychar;
        static public Symbol yylval;

//Constructor
public Lexer(){}


//Constructor
public Lexer(String FileIn)
{
        try{
//              Leer la entrada
                FileReader m_instream = new FileReader(FileIn);
                BufferedReader m_input = new BufferedReader(m_instream);
                String inputStr = new String();
                String aux = new String();
                aux = m_input.readLine();
                while (aux!=null)
                {
                        inputStr+=aux+"\n";
                        aux = m_input.readLine();
                }
                inputString = inputStr.toCharArray();
        }catch(Exception e){}
        //Inicializar variables de entorno
        index_end = -1;
        yyEOF = false;
        yywrap = false;
        yylambda = false;
        yytext = new String();
        yytextchar = ' ';
        yylength= 0;
        yyline = 1;
        yychar = 0;
        yylval = null;
        yylext = new yylexertraductions();
        //Inicializar variables de usuario
        yylext.yyinic_lexer_tradu();
}//Fin constructor


//Para encontrar la ubicacion de los simbolos en el automata
        private int [] ubic_symbol = {
        -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
        -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
        -1,-1,-1,-1,-1,-1,0,-1,1,2,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
        -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
        -1,-1,-1,-1,-1,-1,-1,-1,3,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
        -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
        -1,-1,-1,-1,-1,-1,-1,-1,};

//Estados finales
        private int [] finals = { 2, 4, };

//final_of_expreg i contiene la expresion regular asociada al estado final finals i
        private int [] final_of_expreg = { 1, 0, };
```

```
//Automata
        private int [][] automaton =
        {/*          .   0   1   X  */
         {/*Estado 0:*/ -2 ,  1 ,  1 ,  2 , },
         {/*Estado 1:*/  3 ,  1 ,  1 ,  2 , },
         {/*Estado 2:*/ -2 , -2 , -2 , -2 , },
         {/*Estado 3:*/ -2 ,  4 ,  4 , -2 , },
         {/*Estado 4:*/ -2 , -2 , -2 , -2 , },
        };
//AFT's
private int [][] tradu_automatons =
        {/*          .   0   1   X  */

         {/*Estado 0:*/ -2 ,  1 ,  1 , -2 ,  },
         {/*Estado 1:*/  2 ,  1 ,  1 , -2 ,  },
         {/*Estado 2:*/ -2 ,  3 ,  3 , -2 ,  },
         {/*Estado 3:*/ -2 , -2 , -2 , -2 ,  },
         {/*Estado 4:*/ -2 ,  4 ,  4 ,  5 ,  },
         {/*Estado 5:*/ -2 , -2 , -2 , -2 ,  },        };
//Estados iniciales
        private int [] iniciales = { 0, 4, };

//Acciones traductoras asociadas a u aft
private int [][] tradu_actions =
        {/*          .   0   1   X  */

         {/*Estado 0:*/  0 ,  0 ,  0 ,  0 ,  },
         {/*Estado 1:*/  0 ,  1 ,  2 ,  0 ,  },
         {/*Estado 2:*/  0 ,  0 ,  0 ,  0 ,  },
         {/*Estado 3:*/  0 ,  0 ,  0 ,  0 ,  },
         {/*Estado 4:*/  0 ,  1 ,  2 ,  0 ,  },
         {/*Estado 5:*/  0 ,  0 ,  0 ,  0 ,  },};

        private int [] have_tradu = {1, 1, };

//Retorna el numero de la exp reg asociada a un estado final, -1 si no es final
        private int num_final(int estado)
        {
                for(int i=0;i<finals.length;i++)
                {if (finals[i]==estado) return final_of_expreg[i];}
                return -1;
        }

//Ejecuta una transicion retornando el proximo estado
        private int next_state(int state, char symbol)
        {
           int sym = (int) symbol;
           if ((sym>=0) && (sym<=127))
                {
                   if (-1!=ubic_symbol[sym])
                           return automaton[state][ubic_symbol[sym]];
                   else { /*erroooooooooooooooooorrr simbolo no valido*/
                           yywrap = true;
                           return -1;
                         }
                }
           else {/*errorrrrrrrrrrrrrrrrrr*/
                   yywrap = true;
```

```
                    return -1;
                }
            }

//yytext y demas metodos para acceder a las variables de entorno
    public String yytext()
    { return yytext;
    }

    public char yytextchar()
    { return yytextchar;
    }

    public boolean yyEOF()
    { return yyEOF;
    }

    public int yylength()
    { return yylength;
    }

    public boolean yywrap()
    { return yywrap;
    }

    public int yyline()
    { return yyline;
    }

    public int yychar()
    { return yychar;
    }
    public Symbol yylval()
    { return yylval;
    }

    //Next_lexema, usada por next token
    public int next_lexema()
    {  if (!yyEOF){
            boolean exist_final = false;
            int actual_state = 0;
            int actual_state_final = -1;
            int exp_reg = -1;
            index_end++;
            int index_end_aux = -1;
            String yytext_aux = new String();
            int yylength_aux = 0;
            int yyline_aux = yyline;
            int yychar_aux = yychar;

            int exp_reg_aux =  num_final(actual_state);
            if (exp_reg_aux != -1)
            {
                exist_final = true;
                actual_state_final = actual_state;
                index_end_aux = index_end-1;
                yytext = yytext_aux;
              yylength = yylength_aux;
```

```
                exp_reg = exp_reg_aux;
                   }

            while (true)
            {
                if (index_end==inputString.length-1)
                {
                   if ((exist_final)&&(!yylambda))
                      {
                         if (index_end_aux==inputString.length-2)                                    {
yyEOF = true; }                                  if (yylength==0) yylambda=true;
index_end = index_end_aux;
                         return exp_reg;
                      }
                   else
                      {          /*Error*/
                                 yywrap = true;
                                 index_end=-1;
                                 yyEOF=true;
                                 return   -1;
                      }
                }
                else
                {

                   char actual_sym = inputString[index_end];
                   if (actual_sym=='\n') { yychar_aux=0;yyline_aux++;}
                      else yychar_aux++;
                   int state_aux = next_state(actual_state,actual_sym);
                   if (state_aux==-1) {System.out.println("Error, it isn't a valid symbol :
"+actual_sym);
                                              yywrap = true;
                                 return -1;}
                   if (state_aux!=-2)
                   {
                      index_end++;
                      actual_state = state_aux;
                      yytext_aux = (String) yytext_aux + actual_sym;
                     yylength_aux++;

                         exp_reg_aux =  num_final(actual_state);
                         if (exp_reg_aux != -1)
                         {
                            exist_final = true;
                            actual_state_final = actual_state;
                            index_end_aux = index_end-1;
                            yytext = yytext_aux;
                            yylength = yylength_aux;
                            exp_reg = exp_reg_aux;
                                  }
                   }
                   else
                   {
                      if ((exist_final)&&(!yylambda))
                         {
                                        index_end = index_end_aux;
                                        if (yylength==0) yylambda=true;
                                        return exp_reg;
```

```
                        }
                    else
                        {       /*Error*/
                                yywrap = true;
                            index_end=-1;
                            return   -1;
                        }
                    }
                }
            }
        }
        else {          index_end=-1;
             return -1; }
    }

//Next token
    public int next_token()
    {
        yylval = null;      int token = next_lexema();
        if (token!=-1)
        {
           return yytraduction(token);
        }
        else
        {
           yytext = new String();
           if (yywrap)
                        System.out.println("Lexical error...");
           else
                    System.out.println("Lexical analizer successfull.");
         }
         return token;
    }

    //yytraduction (ejecuta las traducciones de un lexema reconocido)
    private int yytraduction(int token)
    {
        yylext.yyexec_inic(token+1);
        if (have_tradu[token]==1)
        {
                int actual_state = iniciales[token];
                char [] yytext_aux = yytext.toCharArray();
                int index_end_aux = 0;
                yytext = new String();
                yylength = 0;

                while (index_end_aux<yytext_aux.length)
                {
                   char sym_actual = yytext_aux[index_end_aux];
                   yylength++;
                   yytext = (String) yytext + sym_actual;
                   yytextchar = sym_actual;
                   if (sym_actual=='\n') { yychar=0;yyline++;}
                      else yychar++;
                   int sym = (int) sym_actual;
                   int action = tradu_actions[actual_state][ubic_symbol[sym]];
                   if (action!=0)
                   {
```

```
                        yylext.yyexec_traduc(token+1, action);
                    }
                    actual_state = tradu_automatons[actual_state][ubic_symbol[sym]];
                    index_end_aux++;
                }
            }
            return yylext.yyexec_end(token+1);
        }

}//Fin class Lexer
```

Anexo 5

Manual de Usuario

Universidad Nacional de Río Cuarto
Facultad de Ciencias Exactas, Físico-Quimicas y Naturales
Departamento de Computación

JTLex: Un Generador de Analizadores Léxicos

Manual de Usuario

Versión 1.0 – Agosto 2002

Indice

1 – Introducción

Un analizador léxico lee caracteres del archivo de entrada, donde se encuentra la cadena a analizar, reconoce lexemas y retorna tokens. Escribir analizadores léxicos a mano puede resultar se una tarea tediosa y es por esto que existen herramientas de software – los generadores de analizadores léxicos – que han sido desarrolladas para facilitar esta tarea. Los generadores de analizadores léxicos generan automáticamente analizadores léxicos.

Generalmente estas herramientas para generar analizadores léxicos permiten definir la sintaxis de los símbolos mediante una serie de reglas bien definidas. Posteriormente sus atributos deben ser computados analizando la cadena reconocida.

Quizás la herramienta más conocida es *Lex*. *Lex* es un generador de analizadores léxicos que genera código *C*. Básicamente una especificación *Lex* contiene una secuencia de reglas patrón-acción.

JTLex está basado en el modelo de generadores de analizadores léxicos de *Lex*. Con la variante de que JTLex es un generador de analizadores léxicos que acepta una especificación similar a la aceptada por *Lex* y crea código Java para el correspondiente analizador. La especificación es similar porque, no solo permite asociar una acción a cada patrón, sino que, también permite asociar acciones a cada ocurrencia de un símbolo en las reglas que determinan los símbolos del lenguaje. Con lo que se evita, en los casos en que fuera necesario, tener que analizar la cadena reconocida para computar el valor de los atributos.

2 – Instalación y Ejecución de JTLex

Para instalar el generador de analizadores léxicos JTLex realice los siguientes pasos:

1) Compile con Java las clases ubicadas en el subdirectorio Utils.
2) Compile las clases del subdirectorio Java_Cup/runtime.
3) Compile el resto de las clases ubicadas en el directorio a instalar JTLex.

Para compilar con JTLex la especificación creada por el usuario ingrese el siguiente comando:

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

Para utilizar el analizador léxico creado realice los siguientes pasos:

1) Compile con Java el archivo Concrete_Lexical.java.
2) El analizador léxico puede ser usado como un módulo de cualquier programa *Java* – ver sección 4 en donde encontrará más información sobre la interface de los analizadores generados –.

3 - Especificación JTLex

Un archivo de entrada JTLex está organizado en tres secciones, separadas por la directiva %%.

El formato es el siguiente:

Declaraciones
%%
Reglas para Definir los Símbolos del Lenguaje
%%
Código de Usuario

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

En la sección *Declaraciones* – la primer sección de la especificación – se definen macros y directivas de JTLex.

La sección *Reglas para Definir los Símbolos del Lenguaje* contiene las reglas de análisis léxico, cada una de las cuales consiste de una acción inicial opcional, una regla – ver sección 3.2 – y una acción final.

Por último, la sección *Código de Usuario* es copiada directamente en el archivo de salida resultante. Esta provee espacio para la implementación de clases.

3.1 – Declaraciones

Esta sección comienza antes del primer delimitador %%. Cada directiva debe comenzar al principio de la línea con %...{ y debe finalizar al principio de otra línea con %...}, a excepción de la declaración de macros. Las directivas son opcionales y deben ser especificadas en el orden que se introducen a continuación.

3.1.1 – Definición de Atributos y Métodos del Usuario

La directiva %{ ... %} permite al usuario escribir código Java para ser copiado en el analizador léxico. Esta directiva es usado como se detalla a continuación:

```
%{
<Código>
%}
```

El código Java especificado en <*Código*> será copiado en la clase del analizador léxico que evalúa los atributos creada por JTLex.

```
class yylexertraductions
{
... <Código> ...
}
```

Esto permite la declaración de atributos y métodos internos para la clase que efectúa la traducción. Los nombres que comienzan con ***yy*** están reservados para ser usados por las clases del analizador léxico generadas.

3.1.2 – Código de Inicialización del Código Usuario

La directiva %init{ ... %init} permite al usuario escribir código Java para ser copiado en un método de la clase yylexertraductions que es invocado por el constructor de la clase del analizador léxico (*class lexer*).

```
%init{
<Código>
%init}
```

El código Java especificado en <*Código*> será copiado en el método antes mencionado.

```
public void yyinic_lexer_tradu()
{
... <Código> ...
}
```

Esta directiva permite inicializar las variables de usuario en el momento de invocar el constructor del analizador léxico.

3.1.3 – Código de Fin de Archivo para el Analizador Léxico

La directiva %eof{ ... %eof} permite al usuario escribir código Java para ser copiado en la clase del analizador léxico para ser ejecutado después que el fin de archivo es reconocido.

```
%eof{
<Código>
%eof}
```

3.1.4 – Código de Error para el Analizador Léxico

La directiva %error{ ... %error} permite al usuario escribir código Java para ser copiado en la clase del analizador léxico para ser ejecutado cuando se detecta un error en el análisis léxico.

```
%error{
<Código>
%error}
```

3.1.5 - Definición de Macros

Cada definición de macro consiste de un nombre del macro seguido de un espacio y una expresión regular - la cual no es traductora lineal y a la cual tampoco se le puede asociar una acción final. El formato puede resumirse como se expresa a continuación:

```
<nombre del macro 1>   <definición 1>
         ...                                  ...
<nombre del macro N>   <definición N>
```

Los nombres deben ser identificadores válidos, es decir secuencias de letras y dígitos comenzando con una letra. Los nombres deben estar ubicados al comienzo de una línea.

Las definiciones de macros deben ser reglas válidas tal como son descriptas en la próxima sección salvo la excepción que no se permite incluir acciones.

Estas definiciones pueden invocar a otros macros con el formato estándar {<nombre de macro>}. No es posible definir macros recursivos ni tampoco utilizar definiciones de macros no declarados previamente.

Una definición de macro termina cuando comienza otra definición de macro o cuando se encuentra el separador de secciones %%.

3.2 - Reglas para Definir los Símbolos del Lenguaje

La segunda parte de la especificación JTLex consiste de una serie de reglas para dividir la cadena de entrada en tokens. Por ejemplo una regla que reconozca un número real positivo (uno o más números naturales, seguidos por un punto que puede, o no, estar seguido por una serie de números naturales) podría especificarse de la siguiente manera:

```
INIT{ pe=0;  pf=0; n_dfra=0; }
(dígito ACTION{ pe=pe*10 + val( cc ); } ) +  \.
(dígito ACTION{ pf=pf*10 + val( cc ); n_dfra = n_dfra+1; } )*
{ valor = pe + pf / (10 ^ n_dfra); }
```

donde se supone que:

- **dígito** es del tipo 0..9;
- ***cc*** es el carácter corriente;
- las acciones se ejecutan después de haber reconocido cada carácter.

Si existen mas de una regla que machea con el string de entrada, este generador de analizadores léxicos resuelve este conflicto retornando el token asociado al lexema de mayor longitud que se puede reconocer. Si existe mas de una regla con la que machea la cadena reconocida, JTLex devolverá la regla que aparece primero en la especificación JTLex. Si el lexema reconocido no tiene asociado ningún token – la regla no retorna ningún token – el analizador se re-invoca a él mismo.

La reglas consisten de tres partes : una acción inicial opcional, una regla y una acción final.

3.2.1 - Acción Inicial

La acción Inicial es una directiva opcional de la forma INIT{ ... } que permite al usuario escribir código Java para inicializar variables. Esta le permite al usuario inicializar el entorno antes de reconocer un lexema.

```
INIT{  <Código> }
```

3.2.2 – Regla

Los analizadores léxicos se especifican mediante reglas que permiten reconocer tokens. Para cada diferente token, se escribe una regla que lo define. Esta herramienta permite especificar conjuntamente la sintaxis y la semántica de los símbolos del lenguaje. Permite asociar acciones a cada ocurrencia de los caracteres dentro de las reglas. Por razones de eficiencia dentro de cada regla debe existir una única traducción – secuencia de acciones asociadas a los caracteres – para todos los prefijos posibles. En cada regla, si existen prefijos iguales las acciones deben ser las mismas, pero para reglas diferentes pueden llegar a ser distintas.

Por ejemplo si se dan los siguientes casos:

1. Dos expresiones regulares distintas con el mismo prefijo pero con acciones traductoras diferentes
   ```
   a ACTION{i++;} a ACTION{k++;}
   a ACTION{p++;} b
   ```
   cumple con la definición de expresión regular traductora.

2. Una expresión regular con el mismo prefijo pero con acciones traductoras diferentes
   ```
   a ACTION{i++;} a | a ACTION{k++;} b
   ```
   no cumple con la definición de expresión regular traductora.

3. Una expresión regular con el mismo prefijo pero con acciones traductoras diferentes
   ```
   a ACTION{i++;} a | a ACTION{i++;} b
   ```
   cumple con la definición de expresión regular traductora.

El alfabeto para JTLex es el conjunto de caracteres ASCII, los cuales los códigos de los mismos van desde el 0 al 127 inclusive.

Los siguientes caracteres son metacaracteres y tienen un significado especial para las reglas de JTLex:

**? * + | ( ) . [ ] { } “ **

? El signo de pregunta hace maching cuando existe cero o una ocurrencia de la expresión regular precedente.

***** El asterisco hace maching cuando existe cero o más ocurrencias de la expresión regular precedente.

+ El signo de suma hace maching cuando existe una o más ocurrencias de la expresión regular precedente, así *b*+ es equivalente a *bb**.

| Este signo se utiliza para representar la disyunción entre dos expresiones regulares. Por ejemplo si *a* y *b* son dos expresiones regulares, *a*|*b* significa que puede hace maching por *a* o por *b*.

(... **)** Los paréntesis son utilizados para agrupar expresiones regulares.

. Este signo hace maching con cualquier caracter de entrada excepto el fin de línea.

[... **]** Los corchetes se utilizan para representar conjuntos de caracteres o rangos. Existen varias formas de denotar los rangos:

[a,b,....,f]: Se escribe el conjunto de caracteres a representar separados por comas. Por ejemplo si se quiere representar las letras "a", "b" y "c" se escribe [a,b,c].

[ab....f]: Se escribe el conjunto de caracteres a representar. Por ejemplo si se quiere representar las letras "a", "b" y "c" se escribe [abc].

[a-z]: El conjunto de caracteres representado por el rango [a-z] es el que se encuentra entre las letras "a" hasta la letra "z" de acuerdo al código ASCII.

[a-x,z,A-Z]: Se puede utilizar una combinación de los tres casos anteriores para representar otros conjuntos de caracteres.

{nombre macro**}** Se utiliza para hacer una expansión de un macro definido previamente en la sección Declaraciones.

"abc**"** Representa a la palabra abc.

\ Como existen caracteres reservados para el uso de JTLex, la barra seguido de un carácter representa el segundo carácter. De esta forma podemos escribir el carácter + que esta reservado para el uso de JTLex de la forma \+.

3.2.3 - Acción Final

La acción final es una directiva *obligatoria* de la forma { ... } que permite al usuario escribir código Java que se ejecuta luego de que un lexema machee con una regla. Al final del código Java y antes de escribir la segunda llave - } – el usuario debe incluir una sentencia **break** – si no se desea retornar un token – o una sentencia **return** – que retorna un entero que codifique el token asociado –.

```
{  <Código> }
```

3.2.4 – Gramática de las reglas

Para clarificar los conceptos antes mencionados, a continuación se presenta la gramática de las reglas de las expresiones regulares traductoras lineales.

```
<regla> ::=   <regla>     <regla>
            | <regla>  "|"  <regla>
            |  "("   <regla> ")"   <operador>
            |  "\"  CARACTER <acción>   <operador>
```

```
              |  CARACTER <acción> <operador>
              |  “”” PALABRA  “””  <operador>
              |  “[“  RANGO  “]”  <acción> <operador>
              |  “{“  NOMBRE_MACRO   “}”  <operador>
              |  “.”   <acción> <operador>

<operador> ::=  +     |   *      |   ?      |λ

 <acción> ::=  ACTION {Código Java}     | λ
```

en donde:

- CARÁCTER es un carácter del código ASCII.
- PALABRA es una palabra dentro del alfabeto.
- RANGO es un rango.
- NOMBRE MACRO es el nombre de un macro definido previamente.

3.3 - Código de Usuario

En esta sección el usuario puede definir clases Java que necesite utilizar en el análisis léxico. Este código puede comenzar con la declaración *package* <*nombre del paquete*> y/o puede iniciar con *import* <nombre de la clase a importar>. El código de esta sección es copiado en el archivo ***Concrete_Lexical.java.***

En esta sección, por ejemplo, se puede definir una clase principal que invoque al analizador léxico. Para esto se invoca primero el constructor del analizador y luego un método llamado ***next_token*** que retorna el próximo token definido en la clase Lexer – ver sección 4 para más detalles –. A continuación se muestra un ejemplo del código usuario que define una clase Main que invoca al analizador léxico.

```
 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();
      System.out.println("Expresion regular nro :"+i+" -
 Lexema : "+ L.yytext());
}
 System.out.println("Fin”);
 }
}
```

3.4 – Comentarios en JTLex

Los comentarios nos permiten escribir texto descriptivo junto al código, hacer anotaciones para programadores que lo puedan leer en el futuro. Comentando nuestro código nos ahorraremos mucho esfuerzo. Además, cuando escribimos comentarios a menudo descubrimos errores, porque al explicar lo que se supone que debe hacer el código nos obligamos a pensar en ellos. Es por esto que JTLex nos permite ingresar comentarios en la especificación. Los mismos se realizan de la siguiente manera:

/* <Comentario> */ en donde <Comentario> es el texto que desea ingresar el usuario para comentar la especificación.

4 – Analizadores Léxicos Generados

JTLex toma una especificación y la transforma en un archivo Java para el correspondiente analizador léxico.

El analizador léxico generado reside en el archivo Concrete_lexical.java. El mismo se destacan cuatro diferentes tipos de clases: las clases del usuario, la clase yylexertraductions, la clase Lexer y la clase Symbol. A continuación se detalla en contenido de cada clase y los métodos que el usuario puede invocar para la utilización del analizador.

Al principio del archivo Concrete_lexical.java se encuentran las **clases definidas por el usuario**.

Luego se encuentra la clase **yylexertraductions**. Esta contiene las acciones iniciales, las acciones de las expresiones regulares traductoras lineales y las acciones finales. Esta clase contiene tres métodos importantes invocados por la clase Lexer.

- *public void yyexec_inic(int yyregla):* dada una regla ejecuta la acción inicial de la misma. Este método contiene todas las acciones iniciales asociadas a las reglas.

- *public void yyexec_traduc(int yyregla, int yyact):* dada una regla y un número de acción ejecuta ésta acción traductora de dicha regla. Este método contiene todas las acciones traductoras asociadas a las reglas.

- *public int yyexec_end(int yyregla):* dada una regla ejecuta la acción final asociada a la misma. Este método contiene todas las acciones finales asociadas a las reglas.

Inmediatamente se encuentra la **clase Symbol**. Esta clase se utiliza para retornar el valor de los atributos de los tokens. El atributo *yylval* de la clase *Lexer* es de tipo *Symbol* y en él se retornan los atributos de los tokens. Esta clase tiene dos diferentes constructores: *public Symbol(int id, Object o),* crea un *Symbol* con un número de identificador y su valor; *public Symbol(int id),* crea un *Symbol* con solamente un número de identificador.

Al final del archivo se encuentra la clase más importante, la **clase Lexer**. Esta clase tiene varios métodos que el usuario puede utilizar en las acciones finales, iniciales o en las acciones traductoras para obtener distintos resultados. Los métodos son los siguientes:

- *public String yytext():* retorna el lexema reconocido dentro de un string.
- *public char yytextchar():* retorna el carácter actual.
- *public boolean yyEOF():* retorna true si esta posicionado al final del archivo de entrada sino retorna false.
- *public int yylength():*retorna la longitud del lexema reconocido.
- *public boolean yywrap():* retorna true si existe algún error sino retorna false.
- *public int yyline():* retorna la línea en la que se encuentra el analizador.
- *public int yychar():* retorna el número de carácter dentro de la línea corriente.
- *public Symbol yylval():* Retorna el symbol del token reconocido.
- *public void yylval_update (Symbol s):* Modifica el symbol del token reconocido.
- *public int next_token():* Reconoce el próximo token y lo retorna.

5 - Un ejemplo de una especificación JTLex

A continuación se presenta un ejemplo de una especificación JTLex para un lenguaje algorítmico simple (subconjunto de C) denominado C--. Los símbolos básicos de C-- son los identificadores, literales y operadores.

Los identificadores tienen la siguiente estructura **"{letra} ({letra} | {digito})*"**; los literales enteros **"{digito}+"** y los denotadores reales **{digito}+ \. {digito}+**

Los delimitadores del lenguaje son los caracteres especiales y las palabras reservadas que se detallan a continuación:

+	-	*	/
%	!	?	:
=	,	>	<
(	)	{	}
||	&&	==	;
break	continue	else	float
if	int	return	while

La especificación JTLex es la siguiente:

```
%init{ tokenpos=0; cant_coment=0;
%init}
letra [a-z,A-Z]
digito [0-9]
%%

(\ |\t)+ {tokenpos+=yylength();break;}
```

```
\n        {tokenpos=0;break;}
"\*"      {cant_coment++;tokenpos+=yylength();break;}
"*/"      {cant_coment--;tokenpos+=yylength();break;}
"float"{tokenpos+=yylength();System.out.println("float");break;}
"int"  {tokenpos+=yylength();System.out.println("int");break;}
"break"  {tokenpos+=yylength();System.out.println("break");break;}
"continue"
{tokenpos+=yylength();System.out.println("continue");break;}
"else"  {tokenpos+=yylength();System.out.println("else");break;}
"if"  {tokenpos+=yylength();System.out.println("if");break;}
"return"
{tokenpos+=yylength();System.out.println("return");break;}
"while"  {tokenpos+=yylength();System.out.println("while");break;}
"+"  {tokenpos+=yylength();System.out.println("+");break;}
"-"  {tokenpos+=yylength();System.out.println("-");break;}
"*"  {tokenpos+=yylength();System.out.println("*");break;}
"/"  {tokenpos+=yylength();System.out.println("/");break;}
"%"  {tokenpos+=yylength();System.out.println("%");break;}
"!"  {tokenpos+=yylength();System.out.println("!");break;}
"?"  {tokenpos+=yylength();System.out.println("?");break;}
":"  {tokenpos+=yylength();System.out.println(":");break;}
"="  {tokenpos+=yylength();System.out.println("=");break;}
","  {tokenpos+=yylength();System.out.println(",");break;}
">"  {tokenpos+=yylength();System.out.println(">");break;}
"<"  {tokenpos+=yylength();System.out.println("<");break;}
"("  {tokenpos+=yylength();System.out.println("(");break;}
")"  {tokenpos+=yylength();System.out.println(")");break;}
"{"  {tokenpos+=yylength();System.out.println("llaveO");break;}
"}"  {tokenpos+=yylength();System.out.println("llaveC");break;}
"||"  {tokenpos+=yylength();System.out.println("OR");break;}
"&&"  {tokenpos+=yylength();System.out.println("AND");break;}
"=="  {tokenpos+=yylength();System.out.println("==");break;}
";"  {tokenpos+=yylength();System.out.println("puntoYc");break;}

{letra} ({letra}|{digito})* {System.out.println("Identificador:
"+yytext());break;}

INIT{intval=0;} ([0-9]
         ACTION{tokenpos++; Integer aux=new
Integer(yytextchar()+"");
         intval=intval*10 + aux.intValue();} ) +
  {System.out.println("Natural: "+intval);break;}

INIT{realval1=0;realval2=0;cantreal=1;}
         ([0-9] ACTION{tokenpos++; Integer aux=new
Integer(yytextchar()+"");
         realval1=realval1*10 + aux.intValue();} ) +
         \. ([0-9]
         ACTION{tokenpos++; Integer aux=new
Integer(yytextchar()+"");
         cantreal=cantreal*10;
         realval2=realval2*10 + aux.intValue();}) +
  {real=realval1+(realval2/cantreal); System.out.println("Real:
"+real);break;}

. {System.out.println("Error Identificador no valido" +yytext()) ;
break;}
```

```
%%
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();
     System.out.println("Expresion regular nro :"+i+" - Lexema :
"+ L.yytext());
    }
   System.out.println("Fin:  C-- (version 3)");
  }
}
```

6 – Gramática de JTLex

La gramática de una especificación JTLex es la siguiente:

<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 <regla_macro> <decl_macro1>
| NAME_MACRO <regla_macro>
| λ

<body_lex> ::= <inic_java> <regla> "{" code_java "}" <body_lex>
| <inic_java> <regla> "{" code_java "}"

<inic_java> ::= "INIT{" code_java "}" | λ

<regla_macro> ::= <regla_macro> <regla_macro>
| <regla_macro> "|" <regla_macro>
| "(" <regla_macro> ")" <operador>
| "\" CHAR <operador>
| CHAR <operador>
| """ words """ <operador>
| "[" rango "]" <operador>
| "{" NAME_MACRO "}" <operador>
| "." <operador>

<regla> ::= <regla> <regla>
| <regla> "|" <regla>
| "(" <regla> ")" <operador>

| “\” CHAR <j_action> <operador>
| CHAR <j_action> <operador>
| “””” words “”” <operador>
| “[“ rango “]” <j_action> <operador>
| “{“ NAME_MACRO “}” <operador>
| “.” <j_action> <operador>

<words> ::= CHAR <words> | CHAR

<operador> ::= “+” | “*” | “?” | λ

<j_action> ::= “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})*

Bibliografía

[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 Traduccion 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.

[Aho72] A.V. Aho, J.D. Ullman, “The Theory of Parsing, Translation and Compiling”, Prentice Hall, INC., Englewood Cliffs, New Jersey .1972.

[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.

[Fra95] C. Fraser, “Retargetable C Compiler: Desing and Implementation”, Benjamín Cummings Publishing CompaNY. 1995.

[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.

[Gri81] D. Gries, “The Science of Programming”, Springer-Verlag. 1981.

[Hol90] A. Holub, “Compiler Desing in C”, Prentice Hall. 1990.

[Hop69] J.E.Hopcroft, “Formal Languages and their Relation to Automata “, Addison Wesley.1969.

[Hop79] J.E Hopcroft., J.D. Ullman, “Introduction to Automata Theory, Languajes and Computation“, Addison Wesley.1979.

[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

[Kle72] S. C. Kleene, “Representation of events in nerve nets and finite automata”, C. Shannon and J. McCarthy, eds. Automata Studies (Princeton Univ. Press, Princeton, NJ. 1956.

[Lee98] “Handbook of Theoretical Computer Science”, J. Van Leeuwen, Managin Editor. 1998.

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

[McC43] W. S McCulloch, y W.Pitts, “A logical calculus of ideas immanent in nervous activity”, Bull. Math. Biophys. 1943.

[Mor00] G. Morales-Luna, “Principios de Autómatas Finitos”, Sección de Computación CINVESTAV-IPN. 2000.

[Tra85] J. Trambley, P. Sorenson, “The Theory and Practice of Compilers Writing”, Mc Graw Hill. 1985.

[Wai84] Waite, Goos, “Compiler Construction”, Springer-Verlag. 1984.

[Wai92] Waite, Carter, “An Introduction Compiler Construction”, HarperCollins College Publishers. ISBN: 0-673-39822-6. 1992.