UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

"UN GENERADOR AUTOMATICO DE ANALIZADORES LEXICOS"

T E S I S

QUE PARA OBTENER EL TITULO DE:

A C T U A R I O

P R E S E N T A :

MARTIN CASTAÑON IBARRA

México, D. F. 1988

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

CONTENIDO.

INTRODUCCION.

En los sistemas de cómputo, es común clasificar objetos de acuerdo a reglas particulares del problema que se esté afrontando. Resulta ser laborioso para los programadores codificar manualmente esta clasificación, por lo que es conveniente contar con un sistema que reciba como datos de entrada las reglas de clasificación y entregue como salida un sistema que clasifique a los objetos de acuerdo a esas reglas.

Al proceso de determinar si un objeto puede ser clasificado de acuerdo a ciertas reglas se denomina *análisis léxico*, y al sistema que entrega este proceso como salida se le denomina *generador de analizadores léxicos.*

El objetivo de la presente tesis es desarrollar e implantar un generador de analizadores léxicos para una máquina PDP–11/34 con sistema operativo RSX–11M, la razón que llevó a realizar el sistema empleando esta máquina fue el auxiliar a los usuarios de esta máquina en la solución de problemas que requieren de un análisis léxico; por ejemplo a los alumnos de materias de computación como compiladores, programación de sistemas y lenguajes de programación.

El presente trabajo se encuentra estructurado de la siguiente manera.

Capítulo 1. Teoría de los Analizadores Léxicos : En este capítulo se da un bosquejo del medio ambiente bajo el cual se desarrollan los analizadores léxicos, definiendo los conceptos de *Gramáticas, Expresiones Regulares, Autómatas y Analizadores Léxicos*, y se muestra su interrelación.

Capítulo 2. Diseño del sistema : Se define el lenguaje con el cual se debe escribir la gramática para la que se desea generar un analizador léxico, y se definen los módulos en los que está constituido el generador de analizadores léxicos llamado *CIM.*

Capítulo 3. Implantación del Sistema : Se describen los módulos que componen el sistema.

Apéndice I. Manual del Usuario : Se divide en tres partes, la primera describe como hacer uso del sistema *CIM*, la segunda muestra ejemplos de gramáticas junto con sus analizadores léxicos producidos por el sistema *CIM*, y la tercera muestra una descripción detallada de los posibles mensajes que reportaría el sistema en caso de error.

Conclusiones.

Capítulo 1

TEORIA DE ANALIZADORES LEXICOS.

En este trabajo se presenta el desarrollo de *Un Generador de Analizadores Léxicos*, pero antes de tratar de describir qué es un *generador*, es apropiado explicar qué es un *analizador léxico o "scanner"* así como aquellos elementos que se emplean para su desarrollo.

En la teoría de los analizadores léxicos se utilizan los términos *AUTOMATA, GRAMATICA, y EXPRESIONES REGULARES* , los cuales guardan entre sí relaciones muy importantes. A continuación se explicará la importancia de estos elementos en la teoría.

El análisis léxico se encarga de reconocer unidades sobre una secuencia de símbolos, por lo que, para realizar este análisis es necesario haber descrito de alguna manera los elementos o símbolos que componen a la unidad. La manera formal de describir a la unidad es a través de una gramática y un vocabulario. La gramática es un conjunto finito de reglas con las cuales se indica el camino para construir toda unidad, a estas reglas se les llama reglas de producción, y el vocabulario es el conjunto finito de símbolos que al combinarlos forman una unidad, al conjunto de todas las unidades producidas por la gramática se le llama lenguaje.

Cuando se está frente a la tarea de revisar si una unidad satisface las reglas definidas en la gramática, la revisión puede resultar una tarea fácil, pero existen casos donde las producciones de la gramática son tan complejas que hacen difícil el analizar si la oración viola alguna de las reglas de producción.

En particular existen unidades donde su definición está expresada mediante expresiones regulares, a la gramática que usa este tipo de producciones se le conoce como gramática regular, donde las expresiones regulares es una notación, definida por la teoría matemática de la computación, empleada para describir a las unidades que forman el lenguaje.

Es importante describir a las unidades mediante este tipo de gramáticas, porque es posible implantar su reconocimiento a través de un autómata. Empleando un autómata se puede determinar si una secuencia de símbolos dada forma una unidad, logrando así que el análisis léxico se vuelva un proceso relativamente sencillo, además de poder representar el funcionamiento del autómata de manera gráfica.

Las formas con las cuales se puede representar a los autómatas, son las gráficas y tablas, las cuales se explican a continuación.

Supóngase el siguiente ejemplo.

Considérese la definición de la siguiente unidad, escrita mediante expresiones regulares, que define la construcción de un identificador para algún lenguaje de programación.

$ident = let(let|dig)^*$

donde *let* representa una letra del alfabeto y *dig* un dígito, por lo que *ident* se compone de una letra seguida de tantas letras o dígitos como se desee.

i) *Representación gráfica.*

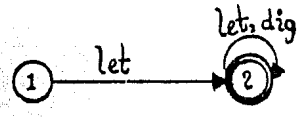


figura 1.

La gráfica que representa el autómata tiene un estado inicial, que es el estado a partir del cual se iniciará la construcción de la unidad (en el ejemplo el estado inicial es el círculo marcado con un uno) y uno o varios estados finales que al llegar a ellos indican que se ha formado una unidad, estos estados se marcan con un doble círculo (aquí el estado dos es un estado final) indicando que al llegar a éste se ha construido la unidad *ident.*

La flecha que une un estado a otro, representa la transición entre estos, además la transición se etiqueta con los símbolos mediante los cuales es posible realizar la transición.

ii) *Representación mediante una tabla.*

En la tabla los renglones representan los estados del autómata y las columnas representan a los símbolos con los cuales es posible realizar una transición de un estado a otro. El cruce de un renglón y una columna indica el estado al que se transitará , por ejemplo dado el estado uno (renglón uno) y el símbolo *let* (columna dos), la tabla indica la transición del estado uno al dos si tenemos un símbolo que pertenezca al conjunto *let.*

	let	dig
1	2	
2	2	2

tabla 1.

1.1 Gramática.

El hombre durante su vida ha escuchado acerca de reglas que hay que respetar para formar alguna unidad. Cuando se está aprendiendo un idioma, por ejemplo Español, Inglés, Francés, siempre se conjuntan la forma y el orden en que deberá escribirse el artículo, adjetívo, verbo, sustantivo, para formar una oración. De igual manera si se desea construir un elemento químico, por ejemplo el etileno, se sabe que deberá estar formado por la siguiente secuencia de elementos y/o compuestos químicos : un acetileno seguido de una presión y temperatura y un catalizador. En biología la clasificación de los seres vivos se hace mediante reglas basadas en algunas de sus características, por ejemplo los invertebrados y vertebrados. Los libros también es posible clasificarlos bajo ciertas reglas previamente definidas, por ejemplo clasificados por nombre del autor, ordenados alfabéticamente, o bien por título de la obra.

Como es posible observar en los casos expuestos, al definir la manera de formar una unidad, se habla de definir las reglas que hay que respetar para su construcción, pues bien a cada conjunto de reglas dadas para la formación de cada una de las unidades, se le denomina *GRAMATICA*, por lo que una gramática es el conjunto finito de reglas que definen el orden y forma de construcción de unidades.

La forma de expresar el orden en que se deben presentar los elementos para formar una unidad se llama regla, una unidad que no respeta las reglas se dirá que no pertenece al conjunto de unidades definidas por la gramática.

Existen unidades donde las formas de definirlas son tantas, o es infinito, que la gramática es el método que logra definir a todas estas formas con un conjunto finito de reglas. Al conjunto de todas las oraciones compuestas por los símbolos del alfabeto que han sido generadas a través de las reglas de la gramática, se le conoce como lenguaje generado por ésta.

Definición de gramática :

Una gramática consiste de un conjunto finito no vacío de reglas o producciones las cuales especifican los elementos del lenguaje . [TS85]

A las unidades empleadas para definir las reglas se les denomina símbolos no terminales, algunos de éstos son llamados símbolos iniciales, y tienen la propiedad de que a partir de éstos se puede iniciar la construcción de un elemento. Los elementos están formados por secuencias de símbolos terminales que pertenecen a un conjunto llamado alfabeto, en combinación con los símbolos no terminales.

Definición Formal :

Una gramática está definida por una cuarteta $G = (V_n, V_t, S, \phi)$ *en donde*

V_t *como el conjunto de símbolos terminales, siendo estos símbolos con los que se puede considerar a una oración formada para después analizarla si está sintácticamente correcta.*

V_n el conjunto de símbolos no terminales.

S es un elemento distinguido de V_n y además del vocabulario, es llamado símbolo inicial.

ϕ es un subconjunto finito no vacío de la forma de relación

$(V_t \cup V_n)^* V_n (V_t \cup V_n)^* \longrightarrow (V_t \cup V_n)^*$

donde un elemento (a, b) denotado como $a \longrightarrow b$ es llamado una regla de producción. [TS85]

Con el estudio de las gramáticas surge una importante rama de la ciencia de la computación llamada *teoría formal de los lenguajes*, surgida a mitad de 1950, como resultado de los estudios de *Noam Chomsky* quien dió un modelo matemático de una gramática en conexión con su estudio del lenguaje natural, llegando hasta generar una clasificación de las gramáticas en cuatro clases, basándose ésta en los diferentes tipos de restricciones impuestos sobre las producciones o reglas.

Clasificación de las gramáticas :

1) *Gramáticas libres o sin restricciones : Son aquellas donde la forma de definir las reglas que componen la gramática no presenta ninguna restricción.*

Ejemplo :

Considérese la gramática que genera todas las cuerdas que contienen igual número de *a's* y *b's* seguidas de un número arbitrario de *c's*, apareciendo en el orden siguiente, primero las *a's*, después las *b's* y por último las *c's*.

Ejemplo : *ab*, *aaabbbcc*.

$$\phi = \{$$

$S = Y$	$S = Z$
$Z = ab$	$Z = Zc$
$Y = bc$	$Y = aY$
$bc = bc$	$bc = bbcc$
$ab = ab$	$ab = aabb$

$$\}$$

$$V_t = \{a, b, c\}$$

$$V_n = \{S, Y, Z\}$$

2) *Gramática sensitivas al contexto : Son aquellas donde las reglas de producción son de la forma*

$$a \longrightarrow b \text{ con } | a | \leq | b |$$

y | a | representa la longitud de a.

Ejemplo :

Gramática que genera las cuerdas que contienen al menos una *a* seguida de al menos una *b* y seguida de al menos una *c*, con la condición de que el número de *a's*, *b's* y *c's* sea el mismo.

Por ejemplo *abc*, *aaabbbccc*.

$$\phi = \{$$

$S = aSBC$	$bC = bc$
$S = abC$	$CB = BC$
$bB = bb$	$cC = cc$

$$\}$$
$$V_t = \{a, b, c\}$$
$$V_n = \{S, B, C\}$$

3) *Gramáticas libres de contexto : Son aquellas donde las producciones son de la forma*

$$a \longrightarrow b \text{ con } | a | \leq | b | \text{ y } a \in V_n.$$

Ejemplo :

Gramática que genera todos los números naturales.

$$\phi = \{$$

$S = numero$	$numero = no$
$no = digito$	$no = digito$
$digito = 0$	$digito = 1$
$digito = 2$	$digito = 3$
$digito = 4$	$digito = 5$
$digito = 6$	$digito = 7$
$digito = 8$	$digito = 9$

$$\}$$
$$V_t = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$
$$V_n = \{S, numero, no, digito\}$$

Este tipo de gramáticas " son usadas " para especificar la sintaxis de algunos lenguajes de computación debido a que estos lenguajes son considerados como simples en estructura, dado que el lado izquierdo sólo presenta un símbolo.

4) *Gramáticas regulares : Son aquellas cuyas producciones son de la forma*

$$a \longrightarrow b \text{ con } | a | = 1$$
$$a \in V_n \text{ y } b \text{ tiene la forma } b = xB,\ b = x$$
$$\text{con } x \in V_t \text{ y } B \in V_n.$$

Ejemplo :

Gramática que genera las cuerdas que contienen al menos una *a* seguida de una *b* seguida de al menos una *a*.

Por ejemplo *aaaba, aabaaaaa.*

$$\phi = \{$$
$$S = aS \qquad S = aB$$
$$B = bC \qquad C = aC$$
$$C = a$$
$$\}$$
$$V_n = \{a, b\}$$
$$V_t = \{S, B, C\}$$

De la clasificación anterior se ha concluido que :

$T_3 \subset T_2 \subset T_1 \subset T_0$

donde :

T_3 = *Gramáticas regulares.*

T_2 = *Gramáticas libres de contexto.*

T_1 = *Gramáticas sensitivas al contexto.*

T_0 = *Gramáticas sin restricciones.*

1.2 Expresiones Regulares.

La teoría matemática de los lenguajes definió elementos, llamados *expresiones regulares*, para decribir a los lenguajes. Estos se describen a continuación. [UH79]

Sea V_t un alfabeto. Las expresiones regulares sobre V_t son :

i) $\emptyset$ es una expresión regular y denota el conjunto vacío.

ii) ϵ es una expresión regular y denota el conjuto $\{\epsilon\}$, es decir el lenguaje que contiene únicamente la cadena vacía.

iii) Para cada a en el alfabeto V_t, a es una expresión regular denotando $\{a\}$ el lenguaje con una única cadena, la cual consiste tan sólo del elemento a.

Definiendo también operadores sobre las expresiones regulares, haciendo posible de esta manera representar en forma concisa a las gramáticas regulares mediante estas expresiones. Estos operadores son :

Sean r y s expresiones regulares las cuales denotan a los lenguajes L_r y L_s respectivamente, entonces :

i) r | s es una expresión regular que representa la unión de los lenguajes y se denota como $L_r \cup L_s$.

Ejemplo :

Sea r = a la expresión regular que representa el lenguaje formado por la cuerda a, y sea s = b la expresión regular que representa el lenguaje formado por la cuerda

b, por lo que entonces, la expresión regular $\mathbf{r} \mid \mathbf{s} = a \mid b$, representa el lenguaje formado por la cuerda a o por la cuerda b.

ii) $\mathbf{r} \cdot \mathbf{s}$ es una expresión regular que denota $L_r \cdot L_s$ la concatenación de sus lenguajes.

Ejemplo :

Sea $\mathbf{r} = 00$ la expresión regular que representa al lenguaje formado por la cuerda 00 y sea $\mathbf{s} = (0 \mid 1)^*$ la expresión regular que representa al lenguaje formado por todas las cuerdas de 0's y 1's. Por lo que $\mathbf{s} \cdot \mathbf{r} \cdot \mathbf{s} = (0 \mid 1)^*00(0 \mid 1)^*$ es la expresión regular que representa al lenguaje formado por todas las cuerdas de 0's y 1's con al menos dos ceros consecutivos.

iii) (r^*) es una expresión regular que denota al lenguaje L_r^* que es la concatenación de ella misma, en un número arbitrario de veces.

Ejemplo :

Sea $\mathbf{r} = a \mid b$ la expresión regular que representa al lenguaje constituido de todas las cuerdas a o b. Por lo que $\mathbf{r} = (a \mid b)^*$ es la expresión regular que representa al lenguaje constituido de todas las cuerdas a's y b's, incluyendo la cuerda vacía.

1.3 Autómatas.

Por medio de gramáticas se definen las especificaciones para construir una *unidad*, es decir se dan los símbolos que la forman y el orden en que deben aparecer dichos símbolos.

Cuando estas especificaciones pertenecen a una gramática regular, es posible representarlas a través de un diagrama de transición, el cual se compone de nodos llamados estados, que están unidos mediante aristas etiquetadas con los símbolos terminales que se deben encontrar para pasar de un estado a otro. Estos símbolos pertenecen al alfabeto definido en la gramática, además este diagrama se compone de un estado inicial, donde para cada arista que se deprenda de él indicará el o los símbolos con los que es posible comenzar a formar la *unidad*, y de estados *finales* que indicarán la *unidad* que se ha formado al llegar a ellos.

Si a una secuencia de símbolos se logra asociarle un conjunto de aristas que partan del estado *inicial* y termine en un estado *final*, se dirá que esta secuencia de símbolos forma una *unidad*, en caso contrario se dirá que dicha secuencia no pertenece al lenguaje definido.

Del párrafo anterior se concluye que estos diagramas son útiles para verificar si una oración satisface la sintaxis definida en la gramática, o lo que es lo mismo, si pertenece al lenguaje definido. Estos diagramas son conocidos como *autómatas*.

Afortunadamente la teoría de la computación ha definido toda una teoría matemática para la construcción de un autómata a partir de una expresión regular, así como un autómata para cada uno de sus operadores. Dado que una gramática

regular define a sus elementos mediante expresiones regulares en combinación con sus operadores, es posible construir su autómata correspondiente.

Ejemplo :

Considérese la siguiente gramática que reconoce un número en punto flotante.

$entero = dig \cdot (dig)^*$;

$mantisa = (- \mid +) \cdot entero \cdot (E \mid e)$;

$exponente = (- \mid +) \cdot entero$;

$numptflot = mantisa \cdot exponente$;

donde *dig* es un dígito .

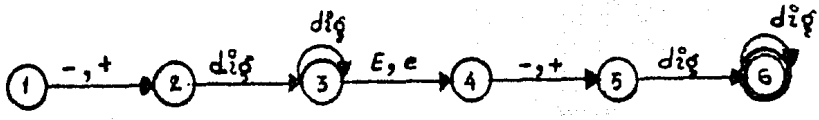


figura 2.

Para cada una de las expresiones regulares y operadores definidos en el inciso 1.2 sus autómatas correspondientes son :

i) *Expresión* $\emptyset$:

figura 3.

ii) *Expresión* ϵ :

figura 4.

iii) *Expresión* **w** :

figura 5.

Donde q_0 es el estado inicial de la expresión y q_f su estado final.
Los autómatas para los operadores son :
i) *Unión o Selección* $(r_1 \mid r_2)$:

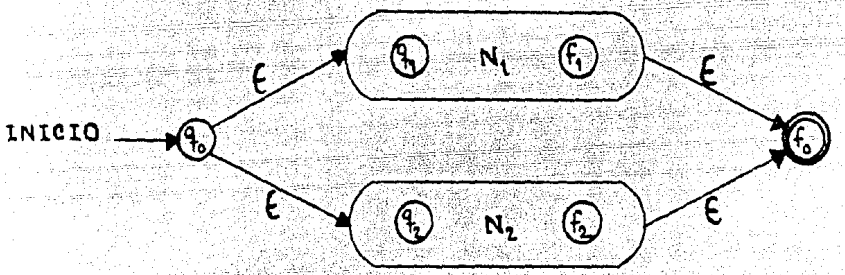


figura 6.

ii) *Concatenación* $(r_1 \cdot r_2)$:

figura 7.

iii) *Cerradura o Repetición (a^*) :*

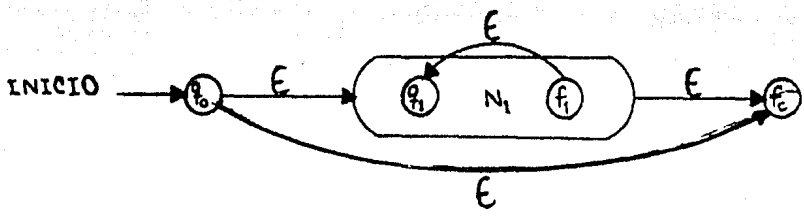


figura 8.

Por lo que a partir de estas definiciones es posible construir el autómata para cada gramática regular dada.

¿ Pero con qué finalidad se construye el autómata para una gramática dada ?

La finalidad que se tiene al representar una gramática mediante un autómata es que permite visualizar la forma en que podría crearse un algoritmo que reconoce la sintaxis especificada en la gramática.

Para cualquier *gramática regular* resulta posible asociarle un autómata, esto es fácil para algunas y complejo para otras, sin embargo puede encontrarse que el autómata presente casos como : dado un estado "i", tenga asociado dos o más aristas etiquetadas con el mismo símbolo, pero que cada una de ellas conduzca a estados diferentes, lo que crea una ambigüedad al diseñar el algoritmo para máquina computadora, debido a que cada vez que se encuentran en el estado "i" y el símbolo a analizar sea con el que se encuentran etiquetadas varias aristas, no se sabrá que estado visitar.

Este tipo de autómatas han sido definidos por la Teoría Formal de los Lenguajes como *Autómatas Finitos No Determinísticos (NFA).*

Definición : *Un autómata no determinístico de estados finitos (NFA) es una quintupla (K, V_t, M, S,Z) donde*

K es el conjunto de estados que forman el autómata.

V_t es el alfabeto de entrada.

M es un mapeo de $K \times V_t$ a un subconjunto de K.

S es el conjunto de estados iniciales.

y

Z es el conjunto de estados finales con $Z \subset K$.

Ejemplo : $r = a^*b^*$.

Expresión regular que representa la cuerda vacía y cuerdas que contienen al menos una *a* seguida de al menos una *b*.

Este diagrama presenta la ambiguëdad en los estados uno y dos sobre los símbolos a y b respectivamente ya que no se sabrá que estado visitar, una vez que se encuentre el símbolo a cuando se esté en el estado uno, y ocurre lo mismo para el estado dos con el símbolo b.

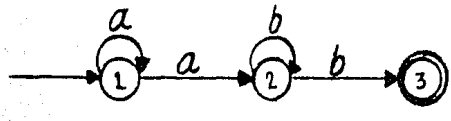


figura 9.

Pero ¿ Cómo atacar este problema ?

Dado que no es fácil transformar estos autómatas a aquellos que no presentan esta característica, la Teoría Formal de los Lenguajes, ha creado NFA tales que la ambigüedad que se presenta sólo sea sobre aquellos estados que desprendan aristas etiquetadas con el símbolo ϵ que representa al vacío. A estos autómatas se les conoce como *NFA con transiciones épsilon* (NFA-ϵ).

Definición : *Un Autómata No Determinístico de Estados Finitos con Transiciones-ϵ (NFA-ϵ) es una quintupla (K, V_t, M, S, Z) donde*

K es el conjunto de estados finales que forman el autómata.

V_t es el alfabeto de entrada.

M es un mapeo $K \times (V_t \cup \{\epsilon\})$ hacia subconjuntos de K.

S es el conjunto de estados iniciales.

y

Z es el conjunto no vacío de estados finales con $Z \subset K$.

Ejemplo : $\mathbf{r} = 01^* \mid 1$.

Expresión regular que reconoce cuerdas formadas por un 0 seguido de una cuerda de 1's, donde esta cuerda puede ser vacía, o cuerdas formadas por un 1. Por ejemplo 0, 01, 01111.

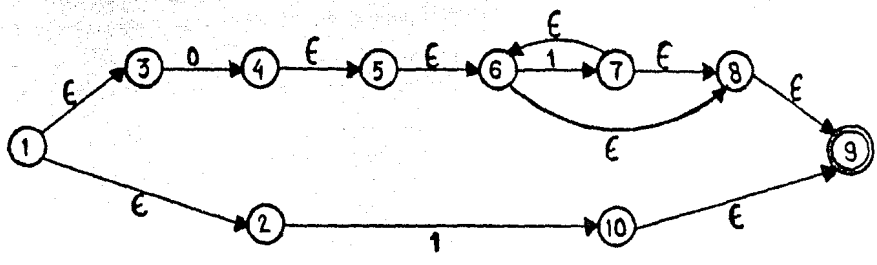


figura 10.

Una vez que fue posible crear el *NFA-ϵ* se aplica un algoritmo que transforma este diagrama a otro que no presente ningún tipo de ambigüedad. Este nuevo diagrama se conoce como *Autómata Finito Determinístico (DFA).*

Algoritmo 1.3.1.

Dado un autómata no determinístico con transiciones-ϵ obtener su autómata determinístico equivalente.

Sea q_0 = estado inicial del NFA-ϵ.

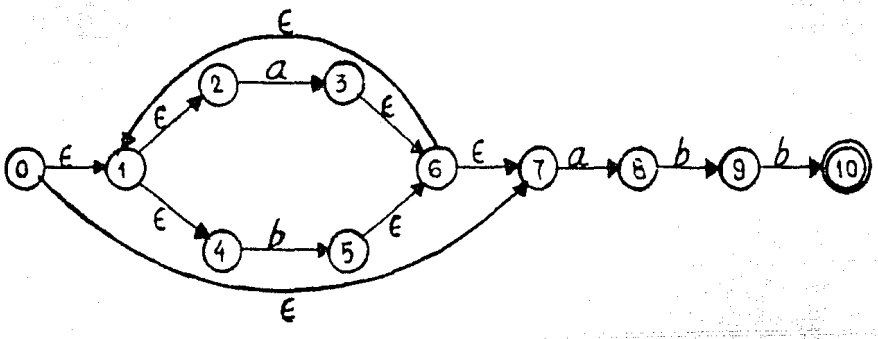
Sea cerradura-ϵ(X) = Conjunto de estados que se visitan al trazar un camino tomando como origen a un estado del conjunto X, donde las aristas que forman el camino están etiquetadas con ϵ, se traza un camino para cada estado elemento de X, incluyendo al conjunto X.

Se inicia la construcción del DFA obteniendo el conjunto cerradura-ϵ(X) para X = $\{q_0\}$.

Una vez obtenido el conjunto A = cerradura-ϵ($\{q_0\}$) tomar a éste como el estado inicial para el DFA a construir, para cada elemento de los símbolos terminales, obtener del conjunto A todos los estados que presentan una transición sobre este símbolo para formar así el conjunto X y obtener su cerradura-ϵ(X); determinar las nuevas transiciones sobre estos estados para cada símbolo terminal y formar así el DFA.

Si el conjunto X generado ha sido obtenido alguna vez, marcar la transición sobre el conjunto cerradura-ϵ(X) bajo el símbolo terminal que generó X y no repetir el proceso para este conjunto, finalizar este proceso hasta que los conjuntos cerradura-ϵ(X) generados para cada símbolo terminal ya hayan sido generados anteriormente, marcando al último conjunto cerradura-ϵ(X) diferente como el estado final del DFA.

Ejemplo : Dado el siguiente NFA-ϵ $(a \mid b)^* abb$, que representa el lenguaje formado por cuerdas compuestas por un número arbitrario de a's y b's en cualquier orden y seguidas de la cuerda abb, obtener su DFA correspondiente.



NFA-ϵ para $(a \mid b)^* abb$.
figura 11.

se tiene el conjunto de símbolos terminales $V_t = \{a, b\}$.

Sea $q_0 = 0$, por lo tanto A = cerradura-ϵ(q_0) = {0,1,2,4,7} y será A el estado inicial del DFA a construir.

Obtener X = { estados a los que conduce la transición de algún estado de A sobre a }

Por lo tanto X = {3,8} y como no ha sido generado entonces :

Sea B = cerradura-ϵ({3,8}) = {1,2,3,4,6,7,8} y marcar una transición de A hacia B con etiqueta a.

Obtener X = { estados que conduce la transición de algún estado de A sobre b } = {5}, y como no ha sido generado entonces :

Sea C = cerradura-ϵ({5}) = {1,2,4,5,6,7} y marcar una transición de A hacia C con etiqueta b.

Para B, obtener X = { estados que conduce la transicón de algún estado de B sobre a } = {3,8} y como ésta ya ha sido generado bajo el nombre de B, marcar una transición de B hacia el mismo con a.

Obtener X = { estados a los que conduce la transición de algún estado de B sobre b } = {5,9} y como no ha sido generado entonces :

D = cerradura-ϵ({5,9}) = {1,2,4,5,6,7,9} y marcar una transición de B hacia D con b.

Se hace lo mismo para C por lo que su conjunto X sobre a es

X = {3,8} el cual ya ha sido generado bajo el nombre de B, por lo que se marcará una transición de C hacia B con a.

Al obtener el conjunto de X con *b* se tiene que X = {5} el cual ya ha sido generado bajo el nombre de C, por lo que se marcará una transición de C hacia el mismo con etiqueta *b*.

Ahora se realiza el análisis para el conjunto D. Se tiene que el conjunto de X con *a* es X = {3,8} el cual ya ha sido generado bajo el nombre de B, por lo que se marcará una transición de D hacia B con etiqueta *a*.

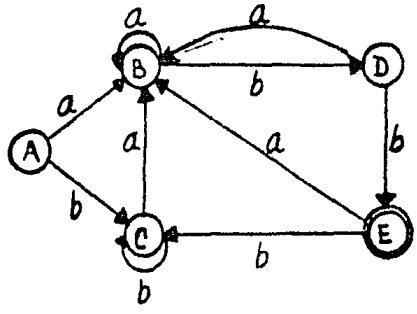
Al obtener el conjunto X con *b* se tiene que X = {5,10} el cual no ha sido generado, por lo que se tiene que E = cerradura-ϵ({5,10}) = {1,2,4,5,6,7,10}, marcando una transición de D hacia E con etiqueta *b*.

Ahora al realizar el análisis para el conjunto E se tiene que :

El conjunto X con *a* es X = {3,8} el cual ha sido generado bajo el nombre de B, entonces se marcará una transición de E hacia B con etiqueta *a*.

Al obtener X con *b* se tiene que X = {5}, ya ha sido generado, por lo que se marcará una transición del estado E hacia C con *b*, y dado que todos los conjuntos X construidos para cada símbolo elemento de V_t ya han sido construidos en procesos anteriores, se ha concluido el proceso de construcción del DFA y se marca como estado final del DFA al estado E.

De lo anterior el autómata DFA generado es :



DFA para $(a \mid b)^* abb$.

figura 12.

Cabe hacer notar que el DFA generado por este algoritmo no es el mínimo DFA con respecto al número de estados que lo forman.

Definición : *Un Autómata Determinístico de estados finitos (DFA) es una quintupla* (K, V_t, M, S, Z) *donde*

K es el conjunto de estados que forman el autómata.

V_t es el alfabeto de entrada.

M es un mapeo $M : K \times V_t \longrightarrow K$ tal que

$M(Q,T) = R$; $Q, R \in K$; $T \in V_t$.

S es el estado inicial con $S \in K$.

Z es el conjunto no vacío de estados finales con $Z \subset K$.

Ejemplo : $\mathbf{r} = (a \mid b)^* abb$.

Expresión regular que reconoce la cuerda *abb* o cuerdas compuestas por tantas *a*'s o *b*'s como se desee (desde cero) seguidas por la cuerda *abb*, se puede construir el (DFA) que queda :

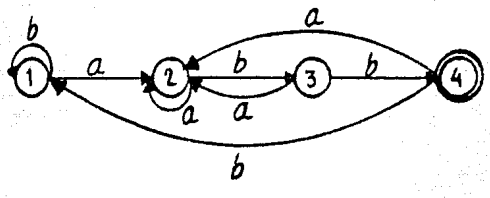


figura 13.

Una vez obtenida la gramática representada mediante un DFA, es posible construir un algoritmo basado en el DFA que sea capaz de revisar si una oración pertenece o no al lenguaje definido.

1.4 Analizadores Léxicos.

En las secciones precedentes, se discutío el medio ambiente bajo el cual se desarrolla la construcción de un analizador léxico, pero *¿Qué es un analizador léxico?.*

Un *Analizador Léxico* tiene cómo función el *"descomponer"* una secuencia de caracteres de entrada o texto de entrada en unidades denominadas átomos (token) especificados por las reglas gramaticales del lenguaje, para realizar esta función se apoya en la construcción del autómata (a través de la cual le es posible reconocer los átomos de la gramática regular especificada), siendo así más fácil la labor de reconocer los átomos.

La rutina del analizador léxico suele ser empleada por alguna otra, que hará uso de los átomos encontrados por el analizador léxico, donde a partir del átomo encontrado se ejecutará una determinada acción.

Existen aplicaciones que necesitan formar unidades para después determinar que acción ejecutar, para éstas resulta útil un analizador léxico. Ejemplos de este tipo son :

1) Revisión de sintaxis. Supóngase que se desea escribir una proposición de asignación en *Pascal* bajo la siguiente estructura :

expresion := constante | variable;

variable := variable | expresion | constante ;

Para revisar si una asignación no viola esta sintaxis habrá que revisar para cada instrucción que presente las siguientes unidades :

i) la unidad *variable.*

ii) la unidad de *asignación.*

iii) la unidad *constante, o la unidad variable, o la unidad expresion.*

Por lo que la tarea del analizador léxico es reportar las unidades que presenta la instrucción proporcionada.

2) El proceso de formatear un texto. Aquí supóngase que dado el siguiente bloque de instrucciones :

```
if c[i] < c[i + 1] then
begin
a := c[i];
c[i] := c[i + 1];
c[i + 1] := a;
end;
```

se desea transformar de la siguiente manera :

```
if c[i] < c[i+1] then
  begin
    a:= c[i] ;
    c[i] := c[i + 1] ;
```

c[i + 1] := a;
end;

es decir, cada vez que se encuentre la unidad *"if"* mover dos espacios a la derecha la instrucción que se encuentra debajo de ella , si ésta es la unidad *"begin"* mover dos espacios a la derecha todas las instrucciones que se encuentran después de ésta hasta encontrar la unidad *"end"*, colocando a ésta a la altura de la unidad *"begin"*.

Como se puede apreciar, esta aplicación requiere de formar la unidad *"if"*, *"begin"* y *"end"* por lo que resulta útil un analizador léxico para armar estas unidades.

De los ejemplos anteriores se puede observar que para resolver el problema planteado es necesario tener un proceso que realice el análisis léxico, por lo que sería conveniente contar con un sistema que recibiera las gramática que define las unidades que se desee reconocer y entregue como salida un sistema que realice el proceso de análisis léxico, a este sistema se le denomina generador de analizadores léxicos. Algunos sistemas de este tipo, ya existentes son : *Lex*, *Alex*. [LM75] y [MH86]

Referencias.

[TS85]: Tremblay, J.P. y Sorenson, P.G.; The theory and practice of compiler writing; McCraw–Hill; 1985.

[UH79]: Ullman, J.D. y Hopcroft, J.E.; Introduction to automata theory, languages, and computation; Addison Wesley; 1970.

[LM75]: Lesk, M.E.; "Lex & lexical analyzer generator"; Computer Science Technical, No 32, Bell Laboratories; Oct. 1975.

[MH86]: Mössenböck H.; "Alex"; Sigplan Notices, Vol 21, No 5; May. 1986.

Capítulo 2

DISEÑO.

El objetivo de este capítulo es mostrar a los usuarios del sistema *CIM* la forma en que habrán de construir su gramática, así como las ventajas y restricciones del sistema.

2.1 Lenguaje Para Describir el Analizador Léxico.

En esta sección se define un lenguaje que tiene como base las expresiones regulares, por lo tanto, permite al usuario describir una gran variedad de gramáticas en forma sistemática.

El usuario debe definir una gramática para poder tener su analizador léxico, los elementos que componen a ésta son :

1) Las clases de símbolos.

2) Los átomos a reconocer.

La manera de definir a cada uno de estos elementos se describe a continuación.

Las reglas que definirán los átomos que desee reconocer pueden presentarse de dos maneras.

1) Si la definición de los átomos hace uso de clases de símbolos, las reglas se definirán de la siguiente forma:

Definición de las clases de símbolos a usar.
%%%
Definición de los átomos a reconocer.

2) Si la definición de los átomos emplea únicamente cuerdas, las reglas se definen de la siguiente forma :

%%%
Definición de los átomos a reconocer.

La forma en que habrá de definirse el conjunto de clases de símbolos es :

⟨clase⟩ ⟶ ⟨nombre de la clase de símbolos⟩ = [⟨varios⟩] ;
⟨nombre de la clase de símbolos⟩ ⟶ ⟨letra⟩ | ⟨letra · mas⟩
⟨mas⟩ ⟶ ⟨letra | digito⟩ · ⟨mas⟩ | ⟨letra | digito⟩
⟨varios⟩ ⟶ ⟨simple⟩ ⟨varios⟩ | ⟨simple⟩

⟨simple⟩ ⟶ ⟨carac⟩ | ⟨carac⟩ - ⟨carac⟩
⟨letra⟩ es cualquier letra del abecedario.
⟨digito⟩ es cualquier dígito.

Ejemplo :

```
letra     = [a-zA-Z] ;
digito    = [0-9] ;
operador  = [+*/] ;
```

A cada clase de símbolos que se define se le asocia un nombre a través del cual se le hace referencia en la definición de los átomos, por lo que dos clases de símbolos no podrán ser definidas bajo el mismo nombre.

Cuando se desee usar un caracter especial, ";", "-", "", "", "(", ")", "[", "]", ".", "|", (se definirán en la sección 2.1.1) en la definición de un átomo, el caracter debe definirse como una cuerda, es decir entre comillas.

El usuario deberá tener cuidado al definir su conjunto de clases de símbolos, porque si la intersección de dos o más clases es distinta del vacío, el autómata que representa a la gramática puede presentar algún problema cuando la intersección del conjunto de las clases de símbolos sobre las que existe transición sea no vacía, y el caracter a analizar sea elemento de la intersección. Cuando éste sea el caso el sistema *gencim.c* reportará a la clase de símbolos que se haya definido en primer lugar como aquella que contiene el caracter, debido a que la revisión se realiza en el orden en que fueron definidas éstas.

Ejemplo :

1) Supongamos que se tiene la siguente gramática :

```
let    = [a-z] ;
asc    = [a-z0-9] ;
%%%
letra  = let ;
ascci  = asc ;
```

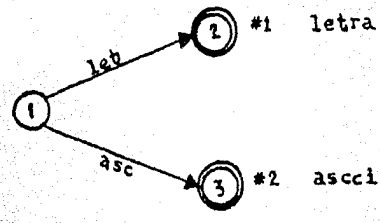


figura 1.

El estado uno presenta dos aristas etiquetadas con las clases `let` y `asc` que reflejan una transición hacia los estados dos y tres respectivamente, para las cuales su intersección es no vacía, cada vez que el caracter a analizar sea una letra del abecedario, se reportará la clase `let` como aquella a la cual pertenece el caracter por ser la primera que se definió, debido a que el análisis para las clases se realiza en el orden en que fueron definidas.

Ejemplo :

2) Considérese la siguiente gramática :

```
let      = [a-z];
asc      = [a-z0-9];
@@@
ident    = { let };
coment   = "{" . {asc } . "}";
```

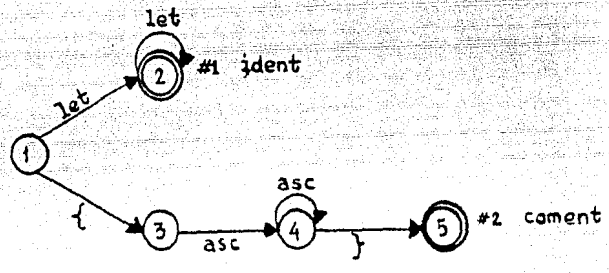


figura 2.

Este ejemplo presenta dos clases tales que su intersección no es vacía, sin embargo no existe estado alguno del autómata, tal que se desprendan de él dos aristas etiquetadas con estas clases, por lo que no habrá problemas en que esta intersección no sea vacía.

La forma en que habrán de definirse los *átomos* será :

⟨atomo⟩ ⟶ ⟨nomtok⟩ = ⟨nomsimple⟩ ;
⟨nomtok⟩ ⟶ ⟨letra⟩ | ⟨letra⟩ · ⟨mas⟩
⟨mas⟩ ⟶ ⟨letra | digito⟩ · ⟨mas⟩ | ⟨letra | digito⟩
⟨nomsimple⟩ ⟶ ⟨termino⟩{ ⟨op-punto⟩ ⟨termino⟩}
⟨termino⟩ ⟶ ⟨factor⟩{⟨op-barra⟩ ⟨factor⟩}
⟨factor⟩ ⟶ ⟨var⟩ | (⟨nomtok⟩) | { ⟨nomtok⟩ }
⟨op-punto⟩ ⟶ ·
⟨op-barra⟩ ⟶ |
⟨var⟩ ⟶ ⟨nombre de la clase de simbolos⟩ | ⟨nomtok⟩ | ⟨cuerda⟩
⟨cuerda⟩ ⟶ " ⟨carac⟩ "

Es necesario que los elementos empleados en la definición del átomo hayan sido declarados con anterioridad, se hace una excepción para las cuerdas, que se consideran como constantes.

Un caracter es cualquier caracter imprimible, para emplear el caracter " (comillas) en la definición de un átomo, habrá que definirla como una clase de símbolos.

Ejemplo : Gramática que reconoce un número real, para el cual su parte entera y decimal, se componen de dos dígitos, los dígitos pueden tomar los valores 1,2,3 o 4.

```
digito      = "1" | "2" | "3" | "4" ;
numero      = digito · digito ;
numreal     = numero · "." · numero ;
```

Para las cuerdas empleadas en la definición del átomo se deberá tener cuidado, ya que éstas presentan un problema parecido al problema anterior, si para un conjunto de cuerdas la transición de un estado a otro existe y el conjunto de cuerdas tiene la particularidad de que algunas cuerdas estén contenidas en otras y la forma de analizar a que cuerda pertenece la secuencia de caracteres es realizada en el orden en que fueron definidas las cuerdas, se puede presentar el siguiente caso.

Sea C_i y C_j dos cuerdas tales que C_i está contenida en C_j y C_i definido antes que C_j y si además la transición del estado "i" sobre la cuerda C_i y C_j existen y si $| C_j | = n$ y $| C_i | = m$, por otro lado sea C_r la cuerda de los primeros n caracteres de la secuencia a analizar, tales que $C_r = C_j$, por la forma en que se realiza el análisis, conducirá a reportar a la secuencia de caracteres como C_i, debido a que sus primeros m-caracteres son iguales a C_i, entonces si el átomo que se está analizando esperaba a la cuerda C_j, ésta no le será reportada, salvo si hubierá sido definida antes que C_i.

Ejemplo :

1) Considérese la siguiente gramática con su autómata :

```
let       = [a-z] ;
dig       = [0-9] ;
@@@
coment    = "{" · { let } · "}";
coment1   = "{*" · { dig } · "*}";
```

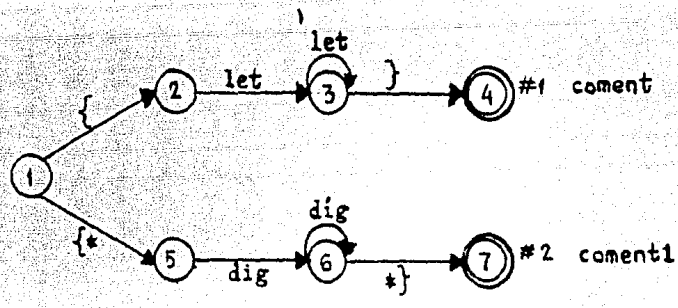


figura 3.

Sobre el estado uno existen dos aristas etiquetadas con las cuerdas "{" y "{*" partiendo de este estado hacia los estados dos y cinco respectivamente, con "{" está definida antes que "{*" y además es una subcuerda de "{*". Para formar el átomo coment y coment1 hay que partir del estado uno y para formar el átomo coment1 se necesita iniciar con la cuerda "{*", nunca se podrá iniciar la formación de este átomo, ya que al encontrar a los caracteres "{*" en la secuencia de caracteres analizada, siempre se reportará la cuerda "{" como aquella que fue encontrada, por ser la primera en definirse.

Los átomos y las clases de símbolos definidos deben estar en un archivo el cual será proporcionado por el usuario.

Un ejemplo que muestra la forma de definir una gramática, de acuerdo a las bases anteriores es :

Ejemplo :

La siguiente gramática define el lenguaje para reconocer un número en aritmética de punto flotante (notación científica).

```
dig = [0-9];
@@@
signo = "-" | "+" ;
entero = {dig} ;
mantisa = signo · entero · ("E" | "e");
exponente = signo · entero ;
numpflot = mantisa · exponente;
```

2.1.1 Operadores.

El significado de los operadores usados para la definición de los átomos y su modo de uso es :

– *Operador* · : Este se aplica a dos operandos bajo la forma $op_1 \cdot op_2$, lo cual significa que el átomo está formado por los dos operandos respetando el orden en que se definieron.

Ejemplo :

```
LETRA = "a" · "b" · "c" ;
```

El átomo LETRA queda definido por la cuerda abc.

–*Operador* | : Este es el operador binario, la forma de usarlo es $operando_1$ | $operando_2$, indicando que el átomo está formado por el $operando_1$ o el $operando_2$ pero no por ambos.

Ejemplo :

```
NOMTOK = "a" | "b" ;
```

El átomo NOMTOK queda definido por el caracter a o el caracter b.

–*Operador* { } : Este indica que los operandos que están dentro de este operador pueden repetirse un número arbitrario de veces a partir de uno.

Ejemplo :

```
NOMTOK = "c" · {"a" | "b" };
```

El átomo NOMTOK indica que el átomo está formado por el caracter c seguida de un caracter a o b o bien de tantas a's o b's como quiera uno.

Siendo las ca, cb, caabbbaaa cuerdas válidas, pero c no.

–*Operador* ( ) : Es permitido para poder definir átomos más complejos, a base de los tres operadores referidos arriba. Con la propiedad distributiva.

Ejemplo :

```
NOMTOK = "a" · ("b" | "c") ;
```

El átomo NOMTOK queda definido por la cuerda de caracteres ab o ac.

–*Operador* [ ] : Es usado para definir alguna clase de símbolos permitiéndonos definir desde algún caracter hasta un cierto rango de caracteres, mediante el operador guión (-).

Ejemplo :

```
LET = [az] ;
```

Indica que el valor que puede tomar la clase de símbolos es el caracter a o el caracter z.

```
LETDIG = [a-z0-9] ;
```

Indica que la clase de símbolos LETDIG puede tomar valores desde el caracter a hasta el caracter z o desde el caracter 0 hata el caracter 9.

2.2 Diseño del Sistema.

Como se ha venido indicando a través del desarrollo del presente trabajo, el objetivo de éste es crear un *Generador de Analizadores Léxicos*, a continuación se mencionarán las etapas por las que se pasa para llegar al objetivo deseado, así como los procesos o funciones usadas en cada una de éstas.

Se llamará *CIM* al programa que recibe como datos de entrada las especificaciones de formación de átomos (tokens) , para así generar un programa en *"C"* (llamado *GENCIM), el cual particionará una secuencia de caracteres que sean sus datos de entrada, en los átomos que fueron especificados.*

Gráficamente el flujo del sistema se verá como sigue :

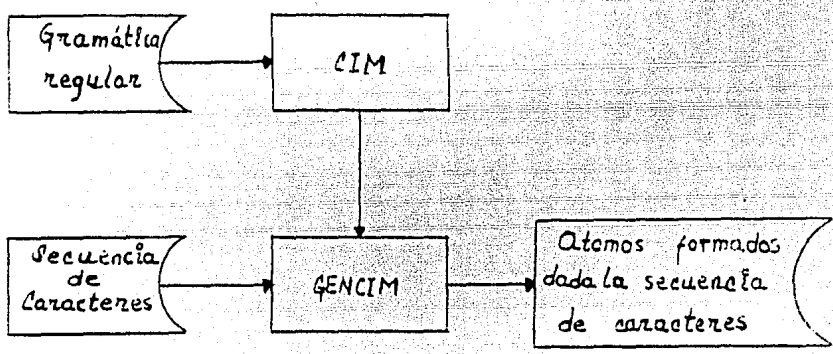


figura 4.

Los átomos aceptados por el generador CIM deberán ser escritos en forma de expresiones regulares, basándose, en las reglas descritas en el Lenguaje para decribir el analizador léxico descrito en la sección anterior, siendo una de las tareas de CIM el revisar que las especificaciones de la gramática no violen estas reglas, tarea que se realiza en la etapa denominada revisión de sintaxis.

Otra función de CIM, realizada por el proceso generación de tablas, es la de representar a los átomos mediante un autómata determinístico através de una tabla, y además crea una tabla extra, que indica el átomo que se ha formado a partir de una secuencia de caracteres dada, estas tablas son usadas por el programa generado por CIM.

En base a lo anterior CIM se dividido en las siguientes etapas :

1.- Revisión de Sintaxis.

2.- Traducción.

3.- Generación de Tablas.

4.– *Generación de Código.*

Gráficamente el proceso de las etapas a realizar por CIM así como el de las funciones empleadas en cada una de éstas estará dada de la forma siguiente :

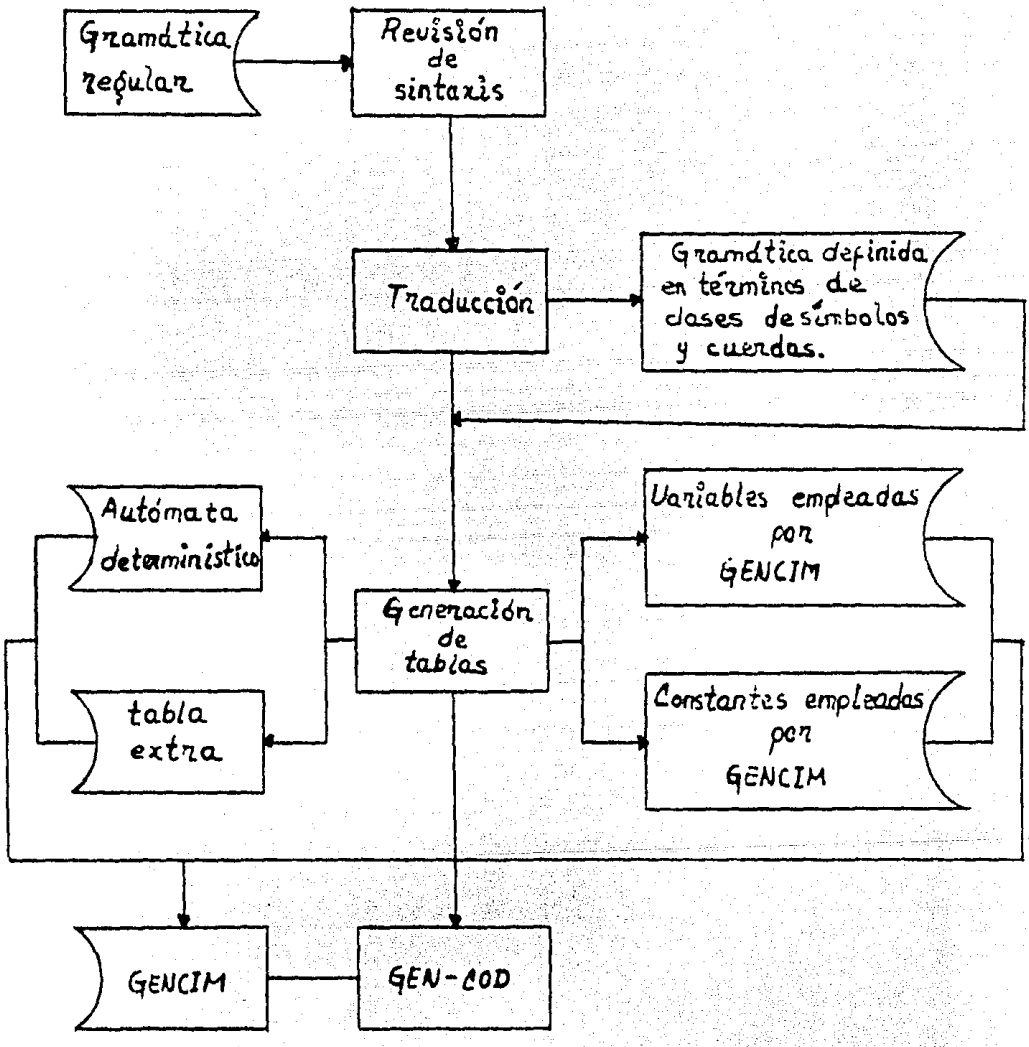


figura 5.

Cabe hacer notar que a cada gramática se le asocia su autómata determinístico y su correspondiente programa gencim.c.

Capítulo 3

IMPLANTACION.

Los módulos mencionados en el punto 2.2 *Diseño del Sistema*, fueron implantados en Lenguaje " *C* " para una máquina *PDP–11/34*, de la forma siguiente.

3.1 Revisión de Sintaxis.

Este módulo realiza una revisión del orden que presentan los datos proporcionados por el usuario almacenados en un archivo, se le compara con el orden definido en el lenguaje para describir el analizador, a través del cual es posible darse cuenta de los elementos que define el usuario y que quedan divididos en dos conjuntos :

1) Definición de clases de símbolos.
2) Definición de los átomos a reconocer.

lo que conduce de forma natural a realizar la implantación de esta etapa mediante dos MODULOS :

1) La revisión sintáctica de las clases de símbolos.
2) La revisión sintáctica de los átomos.

3.1.1 Revisión sintáctica de las clases de símbolos :

En este módulo se revisará :

1) Que estén bien construidas sintácticamente las clases de símbolos.
2) Que no exista más de una clase definida bajo el mismo nombre.

La forma en que se trabaja en este módulo es revisando la sintaxis de cada clase de símbolos definida, parando su ejecución al terminar de revisar el conjunto de la clase de símbolos, o al encontrarse con el primer error sintáctico en alguna de estas clases. Si éste es el caso, se reportará el tipo de error en el que se ha incurrido y se mandará una indicación para que se pare de ejecutar todo el proceso.

Las funciones que se usan son :

1) LEE( ) : Función que lee un caracter del archivo proporcionado por el usuario e indica a que clase pertenece.

2) SCAN( ): Divide el archivo proporcionado por el usuario en las unidades que fueron definidas en el lenguaje para describir el analizador léxico que se desea generar.

3) VERIFICA( ) : Revisa la sintaxis de las clases de símbolos definidas, usando los datos proporcionados por la función scan( ).

2) BUSCA( ) : Usada cada vez que se encuentra con el nombre de una clase de símbolos o átomo definido. Si el nombre corresponde a un clase de símbolos, busca si ya ha sido definida, en otro caso lo inserta en la tabla donde se almacenan estos nombres. Si el nombre corresponde a un átomo, busca si ya ha sido definido, en otro caso lo inserta en la tabla donde se almacenan estos nombres.

3.1.2 Revisión sintáctica de los átomos definidos :

Este módulo actúa sobre el conjunto de átomos definidos y revisa :

1) Que no exista más de un *átomo* definido con el mismo nombre.

2) Que las clases de símbolos y los nombres de los átomos que aparecen del lado derecho del símbolo igual estén definidos anteriormente.

3) Que los átomos definidos hayan sido construidos de acuerdo a las reglas de sintaxis establecidas.

Este módulo deja de ejecutarse cuando se ha revisado el conjunto de átomos definidos, o al encontrarse con el primer error sintáctico. Si éste es el caso se reportará el tipo de error encontrado, y mandará al proceso principal una indicación para que se detenga y regrese al sistema.

Las funciones de las que hace uso además de SCAN( ) y BUSCA( ), definidas antes, son :

1) VACIA( ) : Se encarga de revisar que la pila donde se almacenan los paréntesis izquierdos y llaves izquierdas no esté vacía.

2) POP( ) : Usada cada vez que se encuentra un paéntesis derecho o llave derecha, saca un elemento de la pila.

3) PUSH( ) : Se encarga de almacenar un elemento en la pila cuando se ha encontrado un paréntesis izquierdo o llave izquierda.

4) BUSCA1( ) : Usada cada vez que encuentra en la definición de un átomo una cuerda, busca si la cuerda ya ha sido definida, en otro caso la inserta en la tabla donde se almacenan las cuerdas.

5) BUSCA2( ) : Usada cada vez que la unidad *nombre* ha sido encontrada en la declaración átomo, busca si esta unidad ya ha sido declarada.

3.2 Traducción.

Como se puede observar en el *lenguaje para describir el analizador* al definir un átomo se puede hacer referencia a algún átomo definido anteriormente, por lo que al construir el *autómata determinístico* para algún átomo es posible que en su definición haga referencia al nombre de otro átomo, lo que implicaría posicionarse en el lugar donde se encuentra definido el otro átomo y continuar con la construcción del *autómata*, además de no perder la posición en que se encontraba el generador del autómata dentro del átomo analizado, para poder continuar una vez que se ha terminado de analizar el átomo usado en la definición. Se realiza este complejo proceso cada vez que se encuentre con el nombre de un átomo en la definición del átomo analizado.

Para evitar este tedioso y engorroso procedimiento se decidió sustituir el nombre del átomo empleado en la definición, por su definición, encerrando a ésta entre paréntesis, así el átomo analizado quedará definido en términos de cuerdas y nombres de clases de símbolos.

EJEMPLO :

$num = dig \cdot \{dig\}$;

$numreal = num \cdot \text{“.”} \cdot num$;

después del proceso :

$num = dig \cdot \{dig\}$;

$numreal = (dig \cdot \{dig\}) \cdot \text{“.”} \cdot (dig \cdot \{dig\})$;

La función que realiza esta tarea es *COPIA_ARCH( )* la cual actua desde el primer *átomo* definido hasta llegar al último. Una de las restricciones que presenta el generador es, que todo lo que se emplee en la definición de un *átomo* habrá de haber sido definido anteriormente, ésto permite garantizar que para el primer átomo analizado que haga una referencia a otro, este otro esté definido en términos de clases de símbolos y cuerdas, lo cual implica que desde el primer átomo hasta el último átomo analizado, su definición esté en términos de clases de símbolos y cuerdas. En base a lo anterior, COPIA_ARCH( ) revisa si un átomo hace referencia a algún otro ya definido, de ser así, copia la definición del átomo referenciado (su definición está en términos de clases de símbolos y cuerdas) en la posición donde fue llamado, realizándo este proceso para cada referencia que se haga de algún átomo, hasta tener definido el átomo analizado en términos de clases de sìmbolos y cuerdas.

Otras funciones usadas por este módulo son :

1) COPIA( ) : Esta función para cada *cuerda* encontrada la copia en la definición del átomo.

2) BUSCA4( ) : Función que busca si el nombre leído corresponde al de algún átomo ya declarado, de ser así copia su definición (está en términos de clases de símbolos y cuerdas) en la definición del átomo analizado.

3.3 Generación de Tablas.

En este módulo se realiza la construcción del *áutomata determinístico* para cada átomo definido y la construcción de una tabla que indique que átomos definidos por el usuario se han construido, a partir de un archivo de datos que construye él.

TAB_SGT( ) : Función que para cada átomo, proporciona en una tabla el autómata determinístico que lo representa, e indica el conjunto de estados finales del autómata determinístico. Esta función es recursiva, generándose la recursividad cada vez que se encuentra con un paréntesis izquierdo o llave izquierda, terminando ésta al momento de encontrarse un paréntesis derecho o llave derecha.

-TAB_SAL( ) : Función que construye una tabla la cual indicará que átomo es el que se ha construido dada una secuencia de símbolos .[DG80]

Dentro de este módulo se crean los archivos que son usados por el programa generado (*GENCIM.C*) siendo estos : *tabb.h* el cual contiene las tablas donde se encuentra representado el autómata, y la tabla que indicará el átomo formado; *defg.h* el cual contiene los nombres de las clases y átomos asociados a un número, esto con el objetivo de que el programa generado sea más claro para el usuario, y *varsg.h* el cual contiene la definición de algunas constantes y variables globales.

3.4 Generación de Código.

Este módulo se encarga de generar un programa en " C ", el cual, haciendo uso de los archivos generados en el módulo "generación de tablas", sea capaz de reconocer los diferentes átomos definidos en la gramática del usuario más dos átomos, uno llamado *ERROOR* que es aquel que señala cuando la secuencia de caracteres no forma alguno de los átomos de los que el usuario definió y el átomo *FDA* que señala cuando se ha encontrado el fin de archivo.

Dado que el autómata determinístico y la tabla de salida varían de acuerdo a la gramática proporcionada por el usuario, es lógico pensar que el programa generado también dependerá de la gramática proporcionada, por lo que para cada gramática se obtendrá su correspondiente archivo *GENCIM.C*.

El módulo encargado de realizar esta tarea se encuentra compuesto de tres funciones :

i) *GMAIN( )* : La cual genera a la función main( ) del programa GENCIM.C.

ii) *GSCAN( )* : La cual genera a la función scan( ) del programa GENCIM.C.

iii) *GANALIZA( )* : La cual genera a la función analiza( ) correspondiente al programa GENCIM.C.

Por lo que el programa *GENCIM.C* se encuentra estructurado de la siguiente manera :

i) *MAIN( )* : Es la función encargada de leer la secuencia de caracteres proporcionados en forma directa, es decir a través del teclado, almacenándolos en una tabla, y de llamar a la función *scan( )* para así reportar el código y el nombre del átomo encontrado.

ii) *SCAN( )* : Función que indica el átomo que se ha formado dado un conjunto de datos. Esta función hace uso de las tablas generadas en el módulo "generación de tablas" y de la función *analiza( )* para determinar el átomo que se ha formado.

iii) *ANALIZA( )* : Esta función se encarga de decir a que cuerda o clase de símbolos pertenece el caracter o caracteres que proporciona el usuario, revisando primero si es fin de archivo (*EOF*), en caso contario revisa si pertenece a alguna cuerda definida en la gramática, de no ser así se procede a revisar si pertenece a alguna clase de símbolos definida, de no ser alguno de los casos anteriores se reportará el dato como error (*ERROOR*).

El proceso de revisar si el o los caracteres pertenecen a alguna cuerda se describe a continuación :

Se analiza cuerda por cuerda; el orden en que se analizarán es el orden en que fueron definidas. El criterio podría cambiarse por aquel en donde el orden de análisis se llevará de acuerdo a su orden en base a su longitud, sin embargo de esta forma al igual que sobre la que se está trabajando presentan la posibilidad de reportar alguna cuerda equivocada. Ejemplo : Supóngase que las cuerdas que se definieron son las siguientes : *"abcdef"* y *"abcd"* y supóngase que la secuencia de caracteres dadas por el usuario es *"abcdefghi"*, si se toma el criterio de revisar las cuerdas en el orden en que fueron definidos se reportaría como clase a la cuerda *"abcdef"*, cuando para la formación del átomo analizado se requiriera reportar la cuerda *"abcd"*. Ahora si las cuerdas estuviesen ordenadas de menor a mayor de acuerdo a su longitud, se reportaría la cuerda *"abcd"*, cuando pudiera ser que la cuerda a identificar debería ser *"abcdef"*.

Para determinar a que clase de símbolos pertenece el caracter leído se analiza clase por clase, iniciando el análisis a partir de la primera clase definida hasta la última, si la intersección de los símbolos contenidos en las clases es distinta del vacío y se revisa un caracter tal que el caracter sea elemento de la intersección, la clase que se reportará será aquella que contiene el caracter en la que se haya definido primero.

Para evitar el reportar alguna clase o cuerda en que la transición de algún estado sobre ésta no existe, tan sólo se realizará el análisis sobre las cuales la transición existe.

Ambos programas, el generador *CIM* como el programa generado *GENCIM*, fueron escritos en lenguaje *"C"* debido a que éste permite una gran portabilidad, haciendo posible el uso de *CIM* en diferentes máquinas sin que exista la necesidad de enfrentarse a varios problemas como el de realizar modificaciones al programa

fuente y por otro lado, este lenguaje permite manejar estructuras para hacer más eficiente el uso de espacio de memoria disponible.

Referencias.

[DG80]: Dedoureck, J.M. y Gujar, U.G.; "Scanner design"; Software Practice and Experience, vol 10 , 959–972; 1980.

Apéndice I

MANUAL DE USUARIO.

1. Operación.

En esta sección se pretende dar al usuario la manera de trabajar el sistema *CIM* empleando la máquina *PDP-11/34*.

Para trabajar con el sistema generador de analizadores léxicos *CIM*, el usuario deberá primero crear un archivo en el que se define una gramática de acuerdo a las reglas sintácticas definidas en la sección 2.1 (*lenguaje para describir el analizador léxico*).

Por ejemplo considérese el archivo *CIM.X* que contiene la gramática para definir el átomo ENTERO y el átomo IDENTIFICADOR.

1) Archivo *CIM.X* :

```
letra          = [a-z];
numero         = [0-9];
@@@
ENTERO         = {numero};
IDENTIFICADOR  = letra  {letra | numero};
```

una vez realizado esto, se procederá a ejecutar *CIM*.

2) Para realizar este paso habrá que teclear el nombre del sistema, seguido del nombre de su archivo y teclear un retorno de carro.

>RUN CIM CIM.X < ENTER >

Cuando aparezca de nuevo el prompt

TT5> < ENTER >

y aparecerá >

esto querrá decir que el proceso *CIM* ha concluído.

El proceso revisa si la definición de las clases de símbolos o de los átomos satisface las reglas definidas en el lenguaje para describir el analizador léxico, marcando un error cuando alguno de estos dos conjuntos de definición violen estas reglas parándose todo proceso de ejecución, para cada error en el que incurra el usuario éste deberá ser corregido y volver a ejecutar el sistema *CIM* (paso2).

Para corregir el error, el usuario podrá recurrir a la sección 3 de este apéndice, en el cual se da una mayor explicación de las causas por las que éste fue generado.

Una vez que en la ejecución no exista mensaje de error alguno, el archivo *GENCIM* ha sido generado, para el archivo *CIM.X* su archivo *GENCIM.C* generado es :

```
#include "stdio.h"
#include "Tabb.h"   /*  definición de tablas      */
```

```
#include "defg.h"   /*  definición de constantes */
#include "varsg.h"  /*  definición de variables  */
char nomtok[MAXIMO],arr[100];
int tran_ctes[MAXIMO],tran_clase[MAXIMO];
int codigo,pos,linea[MAXLINE],long_aux,longitud,
     lclase,lctes;
main()
{
int i = 0;
int carac;
pos = 0;
  while((linea[i] = getchar( )) != EOF && i++ < MAXLINE);
  longitud = i;
  scan( );
  while(codigo != FDA){
    printf("codigo = %d token = %s \n",codigo,tokens[codigo-1]);
  if( pos >= MAXLINE ){
    pos = 0;
    i = 0;
    while((linea[i] = getchar( )) != EOF && i++ < MAXLINE);
    longitud = i;
  }
    scan( );
  }
  printf("codigo = %d token = %s \n",codigo,tokens[codigo-1]);
}
/* Esta función se encarga de construir el átomo que
 se forma a partir de la secuencia de caracteres
 proporcionada */
scan( )
{
int sal,edo = 0,k,clase;
int tamtoken = 0;
int ji,ij;
/* obtenemos clases donde existe transición */
ji = 0;
for(ij = 0; ij < NUMCLASE ; ij++)
  if(sgte_edo[edo][ij] != EDO_ERR)
    tran_clase[ji++] = ij;
lclase = ji;
```

```
clase = analiza( );
do{
  sal = salida [edo][clase];
  edo = sgte_edo [edo][clase] - 1;
/*  obtenemos las clases donde existe transición  */
  ji = 0;
  for(ij = 0; ij < NUMCLASE ; ij++)
    if(sgte_edo[edo][ij] != EDO_ERR)
      tran_clase[ji++] = ij;
  lclase = ji;
  if(sal == 0){
    if(tamtoken < MAXIMO -1){
      if(clase >= NUMCLASE && clase != ERROR)
        for(k = 0; arr[k] != '\0';k++)
          nomtok[tamtoken++] = arr[k];
      else{
        nomtok[tamtoken++] = linea[pos-1] ;
      }
    }
    clase=analiza( );
  }
}while(sal == 0);
codigo = sal ;
nomtok[tamtoken] = '\0';
pos-=(clase >= NUMCLASE && clase != ERROR && clase != FIN)?
                                                    long_aux:1;
}
/* función que se encarga de determinar a que clase o
cuerda pertenece el caracter que se esta analizando,
incluyendo si es fin de archivo o si el caracter no
pertenece a ninguno de los anteriores reportará un error */

analiza( )
{
int m,j,aux,l,n;
  if(linea[pos] == EOF){
    pos++;
    return(FIN);
  }
  else{
```

```
    for(m = 0 ; m < lclase ; m++){
      switch(tran_clase[m]){
        case LETRA :
          if('a'<= linea[pos] && linea[pos] <= 'z'){
           pos++;
           return(LETRA);
          }
          break;
        case NUMERO :
          if('0'<= linea[pos] && linea[pos] <= '9'){
           pos++;
           return(NUMERO);
          }
          break;
     }
    }
    pos++;
    return(ERROR);
  }
}
```

CODIGO DEL ARCHIVO GENERADO.

Una vez que *GENCIM* ha sido creado el usuario tiene la posibilidad de compilar y ligar este programa, para así poder ejecutarlo de acuerdo a las especificaciones dadas en el paso tres, obteniendo como salida de *GENCIM* el código del átomo y el nombre de este.

Sin embargo es importante notar que, como el analizador léxico puede formar parte de un sistema mayor, el usuario tiene la posibilidad de modificar a *GENCIM* para que éste sea una parte del sistema, para lograr ésto tendrá que modificar la parte del *MAIN( )* de GENCIM, para así después tan sólo ligarlo con los demás procesos que componen el sistema.

Cuando GENCIM no forma parte de un sistema, después de realizar las modificaciones tan sólo habrá necesidad de realizar el paso 3.

3) Para realizar el proceso de compilación y ligado de GENCIM el usuario deberá relizar los siguientes pasos :

i)

>@C < *enter* >

aparecen los mensajes

> * DAME EL NOMBRE DEL ARCHIVO [S]: < *GENCIM* > < *enter* >

>CCX GENCIM.C

>ASM -d GENCIM.C
>TKB GENCIM/CP = GENCIM.OBJ,LB:[1,1]CCX/LB
>@<EOF>
>

Para ejecutar el programa el usuario tan sólo deberá teclear

>RUN GENCIM < *enter* >
TT5> < *enter* >

aparece el cursor en la pantalla listo para que el usuario teclee la secuencia de caracteres a analizar, finalizando éste con un < *ctrl* > Z para indicar el fin del archivo.

Después de realizar el paso tres sobre el *GENCIM* obtenido para el archivo *CIM.X*, la salida que se produce es :

Entrada.

```
3232323
variable 3443434newvar
 Z
```

Salida.

```
codigo = 1 token = ENTERO
codigo = 4 token = ERROOR
codigo = 2 token = IDENTIFICADOR
codigo = 4 token = ERROOR
codigo = 1 token = ENTERO
codigo = 2 token = IDENTIFICADOR
codigo = 4 token = ERROOR
codigo = 3 token = FDA
```

Si se modifica *GENCIM* para que al encontrarse algún átomo, definido en el archivo *CIM.X*, reporte que éste ha sido encontrado y se muestre los caracteres que lo forman, *GENCIM* estará de la siguiente forma :

GENCIM.C MODIFICADO.

```
#include "stdio.h"
#include "Tabb.h"    /*  definición de tablas      */
#include "defg.h"    /*  definición de constantes */
#include "varsg.h"   /*  definición de variables  */
char nomtok[MAXIMO],arr[100];
int tran_ctes[MAXIMO],tran_clase[MAXIMO];
int codigo,pos,linea[MAXLINE],long_aux,longitud,
     lclase,lctes;
main()
```

```
{
int i = 0;
int carac;
  pos = 0;
  while((linea[i] = getchar( )) != EOF && i++ < MAXLINE);
  longitud = i;
  scan( );
  while(codigo != FDA){
    switch(codigo){
       case ENTERO :
            printf(" se ha reconocido la unidad entero \n");
                    printf("token = %s valor = %s \n",
                            tokens[codigo-1],nomtok);
    break;
       case IDENTIFICADOR :
            printf(" se ha reconocido la unidad identificador \n");
            printf("token = %s valor = %s \n",
                            tokens[codigo-1],nomtok);
    break;
     }
    if( pos >= MAXLINE ){
      pos = 0;
      i = 0;
      while((linea[i] = getchar( )) != EOF && i++ < MAXLINE);
      longitud = i;
    }
    scan( );
  }
  printf(" se ha reconocido la unidad fin de archivo \n");
  printf("codigo = %d token = %s \n",codigo,tokens[codigo-1]);
}
/* Esta función se encarga de construir el átomo que
 se forma a partir de la secuencia de caracteres
 proporcionada */
scan( )
{
int sal,edo = 0,k,clase;
int tamtoken = 0;
int ji,ij;
/* obtenemos clases donde existe transición */
```

```
ji = 0;
for(ij = 0; ij < NUMCLASE ; ij++)
  if(sgte_edo[edo][ij] != EDO_ERR)
    tran_clase[ji++] = ij;
lclase = ji;
clase = analiza( );
do{
  sal = salida [edo][clase];
  edo = sgte_edo [edo][clase] - 1;
/*  obtenemos las clases donde existe transición  */
  ji = 0;
  for(ij = 0; ij < NUMCLASE ; ij++)
    if(sgte_edo[edo][ij] != EDO_ERR)
      tran_clase[ji++] = ij;
  lclase = ji;
  if(sal == 0){
    if(tamtoken < MAXIMO -1){
      if(clase >= NUMCLASE && clase != ERROR)
        for(k = 0; arr[k] != '\0';k++)
          nomtok[tamtoken++] = arr[k];
      else{
        nomtok[tamtoken++] = linea[pos-1] ;
      }
    }
    clase=analiza( );
  }
}while(sal == 0);
codigo = sal ;
nomtok[tamtoken] = '\0';
pos-=(clase >= NUMCLASE && clase != ERROR && clase != FIN)?
                                                  long_aux:1;
}
/* función que se encarga de determinar a que clase o
cuerda pertenece el caracter que se esta analizando,
incluyendo si es fin de archivo o si el caracter no
pertenece a ninguno de los anteriores reportará un error */

analiza( )
{
int m,j,aux,l,n;
```

```
  if(linea[pos] == EOF){
    pos++;
    return(FIN);
  }
  else{
    for(m = 0 ; m < lclase ; m++){
      switch(tran_clase[m]){
        case LETRA :
          if('a'<= linea[pos] && linea[pos] <= 'z'){
           pos++;
           return(LETRA);
          }
          break;
        case NUMERO :
          if('0'<= linea[pos] && linea[pos] <= '9'){
           pos++;
           return(NUMERO);
          }
          break;
     }
    }
    pos++;
    return(ERROR);
  }
}
```

Después de realizar el paso tres sobre el nuevo *GENCIM*, la salida que produce éste es :

CORRIDA

Entrada. 535353
identificador 3222323
numero
^Z
Salida.
se ha reconocido la unidad entero
token = ENTERO valor = 535353
se ha reconocido la unidad identificador
token = IDENTIFICADOR valor = identificador
se ha reconocido la unidad entero
token = ENTERO valor = 3222323

se ha reconocido la unidad identificador
token = IDENTIFICADOR valor = numero
se ha reconocido la unidad fin de archivo
codigo = 3 token = FDA

2. Ejemplos.

Esta sección muestra algunos ejemplos sobre los cuales trabaja el generador de analizadores léxicos *CIM*. Para cada ejemplo se muestra la gramática ,su analizador léxico *GENCIM*, y la corrida de *GENCIM*.

Ejemplo 1.

La siguiente gramática genera las siguientes unidades : 1) El retorno de carro, 2) Un número entero, y 3) Cuerda de caracteres de letras y dígitos iniciando con una letra.

```
carro  = [\n];
let    = [a-zA-Z];
dig    = [0-9];
%%%
rcarro = carro ;
numero = { dig };
var    = let . { let | dig };
```

A esta gramática se aplicará el sistema *CIM* para obtener su analizador léxico *GENCIM*.

Programa Generado

```
#include "stdio.h"
#include "Tabb.h"   /*  definición de tablas      */
#include "defg.h"   /*  definición de constantes */
#include "varsg.h"  /*  definición de variables  */
char nomtok[MAXIMO],arr[100];
int tran_ctes[MAXIMO],tran_clase[MAXIMO];
int codigo,pos,linea[MAXLINE],long_aux,longitud,
     lclase,lctes;
main()
{
int i = 0;
int carac;
pos = 0;
  while((linea[i] = getchar( )) != EOF && i++ < MAXLINE);
```

```
  longitud = i;
  scan( );
  while(codigo != FDA){
    printf("codigo = %d token = %s \n",codigo,tokens[codigo-1]);
  if( pos >= MAXLINE ){
    pos = 0;
    i = 0;
    while((linea[i] = getchar( )) != EOF && i++ < MAXLINE);
    longitud = i;
  }
    scan( );
  }
  printf("codigo = %d token = %s \n",codigo,tokens[codigo-1]);
}
/* Esta función se encarga de construir el átomo que
 se forma a partir de la secuencia de caracteres
 proporcionada */
scan( )
{
int sal,edo = 0,k,clase;
int tamtoken = 0;
int ji,ij;
/* obtenemos clases donde existe transición */
ji = 0;
for(ij = 0; ij < NUMCLASE ; ij++)
  if(sgte_edo[edo][ij] != EDO_ERR)
    tran_clase[ji++] = ij;
lclase = ji;
clase = analiza( );
do{
  sal = salida [edo][clase];
  edo = sgte_edo [edo][clase] - 1;
/*  obtenemos las clases donde existe transición  */
  ji = 0;
  for(ij = 0; ij < NUMCLASE ; ij++)
    if(sgte_edo[edo][ij] != EDO_ERR)
      tran_clase[ji++] = ij;
  lclase = ji;
  if(sal == 0){
    if(tamtoken < MAXIMO -1){
```

```
      if(clase >= NUMCLASE && clase != ERROR)
        for(k = 0; arr[k] != '\0';k++)
          nomtok[tamtoken++] = arr[k];
      else{
        nomtok[tamtoken++] = linea[pos-1] ;
      }
    }
    clase=analiza( );
  }
}while(sal == 0);
codigo = sal ;
nomtok[tamtoken] = '\0';
pos-=(clase >= NUMCLASE && clase != ERROR && clase != FIN)?
                                                    long_aux:1;
}
/* función que se encarga de determinar a que clase o
cuerda pertenece el caracter que se esta analizando ,
incluyendo si es fin de archivo o si el caracter no
pertenece a ninguno de los anteriores reportará un error */

analiza( )
{
int m,j,aux,l,n;
  if(linea[pos] == EOF){
    pos++;
    return(FIN);
  }
  else{
    for(m = 0 ; m < lclase ; m++){
      switch(tran_clase[m]){
        case CARRO :
          if('\n' == linea[pos]){
           pos++;
           return(CARRO);
          }
          break;
        case LET :
          if('a'<= linea[pos] && linea[pos] <= 'z'||
             'A'<= linea[pos] && linea[pos] <= 'Z'){
           pos++;
```

```
            return(LET);
           }
           break;
         case DIG :
           if('0'<= linea[pos] && linea[pos] <= '9'){
            pos++;
            return(DIG);
           }
           break;
      }
     }
     pos++;
     return(ERROR);
   }
}
```

CORRIDA.

```
Entrada
variable
212121
letra
^Z
Salida
codigo = 3 token = VAR
codigo = 1 token = RCARRO
codigo = 2 token = NUMERO
codigo = 1 token = RCARRO
codigo = 3 token = VAR
codigo = 1 token = RCARRO
codigo = 4 token = FDA
```

Ejemplo 2.

La siguiente gramática genera las siguientes unidades, las cuales son palabras reservadas en el lenguaje *Pascal* .

```
@@@
palres01 = "BEGIN";
palres02 = "END";
palres03 = "IF";
palres04 = "THEN";
palres05 = "ELSE";
```

```
palres06 = "WHILE";
palres07 = "DO";
palres08 = "PROGRAM";
palres09 = "FOR";
palres10 = "TO";
```

Para esta gramática se obtendra su analizador léxico *GENCIM*.

Programa Generado.

```
#include "stdio.h"
#include "Tabb.h"   /*  definición de tablas      */
#include "defg.h"   /*  definición de constantes */
#include "varsg.h"  /*  definición de variables  */
char nomtok[MAXIMO],arr[100];
int tran_ctes[MAXIMO],tran_clase[MAXIMO];
int codigo,pos,linea[MAXLINE],long_aux,longitud,
     lclase,lctes;
main()
{
int i = 0;
int carac;
pos = 0;
  while((linea[i] = getchar( )) != EOF && i++ < MAXLINE);
  longitud = i;
  scan( );
  while(codigo != FDA){
    printf("codigo = %d token = %s \n",codigo,tokens[codigo-1]);
  if( pos >= MAXLINE ){
    pos = 0;
    i = 0;
    while((linea[i] = getchar( )) != EOF && i++ < MAXLINE);
    longitud = i;
  }
    scan( );
  }
  printf("codigo = %d token = %s \n",codigo,tokens[codigo-1]);
}
/* Esta función se encarga de construir el átomo que
 se forma a partir de la secuencia de caracteres
 proporcionada */
scan( )
```

```
{
int sal,edo = 0,k,clase;
int tamtoken = 0;
int ji,ij;
/* obtenemos cuerdas donde existe transición */
if(pos + LLONGMIN < longitud){
  ji = 0;
  for(ij = NUMCLASE ; ij < NUMCTES + NUMCLASE ; ij++)
    if(sgte_edo[edo][ij] != EDO_ERR)
      tran_ctes[ji++] = ij - NUMCLASE;
  lctes = ji;
}
clase = analiza( );
do{
  sal = salida [edo][clase];
  edo = sgte_edo [edo][clase] - 1;
/*  obtenemos las cuerdas donde existe transición  */
  if(pos + LLONGMIN < longitud){
    ji = 0;
    for(ij = NUMCLASE ; ij < NUMCTES + NUMCLASE ; ij++)
      if(sgte_edo[edo][ij] != EDO_ERR)
        tran_ctes[ji++] = ij - NUMCLASE;
    lctes = ji;
  }
  if(sal == 0){
    if(tamtoken < MAXIMO -1){
      if(clase >= NUMCLASE && clase != ERROR)
        for(k = 0; arr[k] != '\0';k++)
          nomtok[tamtoken++] = arr[k];
      else{
        nomtok[tamtoken++] = linea[pos-1] ;
      }
    }
    clase=analiza( );
  }
}while(sal == 0);
codigo = sal ;
nomtok[tamtoken] = '\0';
pos-=(clase >= NUMCLASE && clase != ERROR && clase != FIN)?
                                                    long_aux:1;
```

```
}
/* función que se encarga de determinar a que clase o
cuerda pertenece el caracter que se esta analizando ,
incluyendo si es fin de archivo o si el caracter no
pertenece a ninguno de los anteriores reportará un error */

analiza( )
{
int m,j,aux,l,n;
  if(linea[pos] == EOF){
    pos++;
    return(FIN);
  }
  else{
    j = pos ;
    for(m = 0; m < lctes ; m++){
      n = tran_ctes[m];
      aux = arr_ctes[n].llong;
      for(l = 0 ; l < aux ; l++)
        arr[l] = linea[j++];
      arr[l] ='\0';
      if(!strcmp(arr,arr_ctes[n].cuer)){
        long_aux = arr_ctes[n].llong;
        pos += arr_ctes[n].llong;
        return(n+NUMCLASE);
      }
      j = pos;
    }
    pos++;
    return(ERROR);
  }
}
```

CORRIDA.

Entrada

```
BEGIN
IFTHENDOWHILEPROGRAMFORTO
ELSE
END
^Z
```

```
Salida
codigo = 1 token = PALRESO1
codigo = 12 token = ERROOR
codigo = 3 token = PALRESO3
codigo = 4 token = PALRESO4
codigo = 7 token = PALRESO7
codigo = 6 token = PALRESO6
codigo = 8 token = PALRESO8
codigo = 9 token = PALRESO9
codigo = 10 token = PALRES10
codigo = 12 token = ERROOR
codigo = 5 token = PALRESO5
codigo = 12 token = ERROOR
codigo = 2 token = PALRESO2
codigo = 12 token = ERROOR
codigo = 11 token = FDA
```

Ejemplo 3.

La siguiente gramática genera las siguientes unidades :

1) El retorno de carro.

2) Un número entero.

3) Un número real.

4) Una cuerda de carateres de letras y dígitos, donde el primer caracter es una letra.

5) El operador suma, resta, multiplica, y divide (+ − /*).

6) Los operadores relacionales: >,>,=,>=,<=.

7) Cuerdas de letras y dígitos encerradas por apóstrofes, llaves o un paréntesis seguido de un asterisco y un asterisco seguido de un paréntesis.

```
carro      = [\n];
let        = [a-zA-Z];
dig        = [0-9];
op         = [+/*];
oprel      = [<>=];
assc       = [ a-zA-Z0-9];
@@@
rcarro     = carro ;
numero     = { dig };
real       = numero . "." . numero ;
var        = let . { let | dig };
operador   = op | "-";
```

```
Salida
codigo = 1 token = PALRESO1
codigo = 12 token = ERROOR
codigo = 3 token = PALRESO3
codigo = 4 token = PALRESO4
codigo = 7 token = PALRESO7
codigo = 6 token = PALRESO6
codigo = 8 token = PALRESO8
codigo = 9 token = PALRESO9
codigo = 10 token = PALRES10
codigo = 12 token = ERROOR
codigo = 5 token = PALRESO5
codigo = 12 token = ERROOR
codigo = 2 token = PALRESO2
codigo = 12 token = ERROOR
codigo = 11 token = FDA
```

Ejemplo 3.

La siguiente gramática genera las siguientes unidades :

1) El retorno de carro.

2) Un número entero.

3) Un número real.

4) Una cuerda de carateres de letras y dígitos, donde el primer caracter es una letra.

5) El operador suma, resta, multiplica, y divide $(+ - /*)$.

6) Los operadores relacionales: $>,>,=,>=,<=$.

7) Cuerdas de letras y dígitos encerradas por apóstrofes, llaves o un paréntesis seguido de un asterisco y un asterisco seguido de un paréntesis.

```
carro      = [\n];
let        = [a-zA-Z];
dig        = [0-9];
op         = [+/*];
oprel      = [<>=];
assc       = [ a-zA-Z0-9];
@@@
rcarro     = carro ;
numero     = { dig };
real       = numero . "." . numero ;
var        = let . { let | dig };
operador   = op | "-";
```

```
oprelacion = oprel | "<=" | ">=";
text       = "'".{assc}."'";
coment     = "{" . {assc} ."}";
coment1    = "(*". {assc} ."*)";
```

Para esta gramática se genera su analizador léxico *GENCIM*.

Programa Generado .

```
#include "stdio.h"
#include "Tabb.h"   /*  definición de tablas       */
#include "defg.h"   /*  definición de constantes */
#include "varsg.h"  /*  definición de variables  */
char nomtok[MAXIMO],arr[100];
int tran_ctes[MAXIMO],tran_clase[MAXIMO];
int codigo,pos,linea[MAXLINE],long_aux,longitud,
     lclase,lctes;
main()
{
int i = 0;
int carac;
pos = 0;
  while((linea[i] = getchar( )) != EOF && i++ < MAXLINE);
  longitud = i;
  scan( );
  while(codigo != FDA){
    printf("codigo = %d token = %s \n",codigo,tokens[codigo-1]);
  if( pos >= MAXLINE ){
    pos = 0;
    i = 0;
    while((linea[i] = getchar( )) != EOF && i++ < MAXLINE);
    longitud = i;
  }
    scan( );
  }
  printf("codigo = %d token = %s \n",codigo,tokens[codigo-1]);
}
/* Esta función se encarga de construir el átomo que
 se forma a partir de la secuencia de caracteres
 proporcionada */
scan( )
{
```

```
int sal,edo = 0,k,clase;
int tamtoken = 0;
int ji,ij;
/* obtenemos cuerdas donde existe transición */
if(pos + LLONGMIN < longitud){
  ji = 0;
  for(ij = NUMCLASE ; ij < NUMCTES + NUMCLASE ; ij++)
    if(sgte_edo[edo][ij] != EDO_ERR)
      tran_ctes[ji++] = ij - NUMCLASE;
  lctes = ji;
}
/* obtenemos clases donde existe transición */
ji = 0;
for(ij = 0; ij < NUMCLASE ; ij++)
  if(sgte_edo[edo][ij] != EDO_ERR)
    tran_clase[ji++] = ij;
lclase = ji;
clase = analiza( );
do{
  sal = salida [edo][clase];
  edo = sgte_edo [edo][clase] - 1;
/*  obtenemos las cuerdas donde existe transición  */
  if(pos + LLONGMIN < longitud){
    ji = 0;
    for(ij = NUMCLASE ; ij < NUMCTES + NUMCLASE ; ij++)
      if(sgte_edo[edo][ij] != EDO_ERR)
        tran_ctes[ji++] = ij - NUMCLASE;
    lctes = ji;
  }
/*  obtenemos las clases donde existe transición  */
  ji = 0;
  for(ij = 0; ij < NUMCLASE ; ij++)
    if(sgte_edo[edo][ij] != EDO_ERR)
      tran_clase[ji++] = ij;
  lclase = ji;
  if(sal == 0){
    if(tamtoken < MAXIMO -1){
      if(clase >= NUMCLASE && clase != ERROR)
        for(k = 0; arr[k] != '\0';k++)
          nomtok[tamtoken++] = arr[k];
```

```
      else{
        nomtok[tamtoken++] = linea[pos-1] ;
      }
    }
    clase=analiza( );
  }
}while(sal == 0);
codigo = sal ;
nomtok[tamtoken] = '\0';
pos-=(clase >= NUMCLASE && clase != ERROR && clase != FIN)?
                                                    long_aux:1;
}
/* función que se encarga de determinar a que clase o
cuerda pertenece el caracter que se esta analizando ,
incluyendo si es fin de archivo o si el caracter no
pertenece a ninguno de los anteriores reportará un error */

analiza( )
{
int m,j,aux,l,n;
  if(linea[pos] == EOF){
    pos++;
    return(FIN);
  }
  else{
    j = pos ;
    for(m = 0; m < lctes ; m++){
      n = tran_ctes[m];
      aux = arr_ctes[n].llong;
      for(l = 0 ; l < aux ; l++)
        arr[l] = linea[j++];
      arr[l] ='\0';
      if(!strcmp(arr,arr_ctes[n].cuer)){
        long_aux = arr_ctes[n].llong;
        pos += arr_ctes[n].llong;
        return(n+NUMCLASE);
      }
      j = pos;
    }
    for(m = 0 ; m < lclase ; m++){
```

```
switch(tran_clase[m]){
 case CARRO :
   if('\n' == linea[pos]){
    pos++;
    return(CARRO);
   }
   break;
 case LET :
   if('a'<= linea[pos] && linea[pos] <= 'z'||
      'A'<= linea[pos] && linea[pos] <= 'Z'){
    pos++;
    return(LET);
   }
   break;
 case DIG :
   if('0'<= linea[pos] && linea[pos] <= '9'){
    pos++;
    return(DIG);
   }
   break;
 case OP :
   if('+' == linea[pos] ||
      '/' == linea[pos] ||
      '*' == linea[pos]){
    pos++;
    return(OP);
   }
   break;
 case OPREL :
   if('<' == linea[pos] ||
      '>' == linea[pos] ||
      '=' == linea[pos]){
    pos++;
    return(OPREL);
   }
   break;
 case ASSC :
   if(' ' == linea[pos] ||
      'a'<= linea[pos] && linea[pos] <= 'z'||
      'A'<= linea[pos] && linea[pos] <= 'Z'||
```

```
            '0'<= linea[pos] && linea[pos] <= '9'){
          pos++;
          return(ASSC);
          }
          break;
      }
    }
    pos++;
    return(ERROR);
  }
}
```

CORRIDA.

Entrada

```
4242variable>232.3223<+-*=
{comentario} >='texto'/(* otro comentario *)
^Z
```

Salida

```
codigo = 2 token = NUMERO
codigo = 4 token = VAR
codigo = 6 token = OPRELACION
codigo = 3 token = REAL
codigo = 6 token = OPRELACION
codigo = 5 token = OPERADOR
codigo = 5 token = OPERADOR
codigo = 5 token = OPERADOR
codigo = 6 token = OPRELACION
codigo = 1 token = RCARRO
codigo = 8 token = COMENT
codigo = 6 token = OPRELACION
codigo = 7 token = TEXT
codigo = 5 token = OPERADOR
codigo = 9 token = COMENT1
codigo = 1 token = RCARRO
codigo = 10 token = FDA
```

3. Errores.

A continuación se describen los diferentes tipos de errores en los que se puede incurrir al definir la gramática.

Este apéndice se divide en dos partes, la primera describe los errores en que se puede incurrir al definir el conjunto de las clases de símbolos.

La segunda se refiere a los errores reportados al definir el conjunto de átomos.

PARTE I.

CERROR 0 : Existió un error en el nombre dado a la clase de símbolos o faltó nombre.

CERROR 1 : Se olvidó dar un nombre a la clase de símbolos definidas.

CERROR 2 : Faltó símbolo de asignación entre el nombre de la clase de símbolos y la definición de la clase de símbolos.

CERROR 3 : Existe un error en la defrinición de los elementos que componen la clase de símbolos.

CERROR 4 : Faltó finalizar la declaración de la clase de símbolos con un punto y coma.

CERROR 5 : Se definieron bajo el mismo nombre dos clases de símbolos.

PARTE II.

ERROR 0 : Existe algún error al inicio de la declaración del átomo.

ERROR 1 : El nombre de la definición del átomo está mal construida o falta darle un nombre a la definición del átomo.

ERROR 2 : Indica que el nombre dado a la definición del átomo ya ha sido empleado para definir una clase de símbolos o bien a otro átomo definido anteriormente.

ERROR 3 : Indica que falta símbolo de asignación entre el nombre del átomo y la definición de el mismo.

ERROR 4 : Indica que falta definir el átomo.

ERROR 5 : Se inició la definición del átomo con un operador de selección o concatenación.

ERROR 6 : En la definición del átomo existe algún paréntesis izquierdo que faltó ser cerrado.

ERROR 7 : Dentro del operador paréntesis existe una llave izquierda que faltó ser cerrada.

ERROR 8 : Indica que en la definición del átomo faltó cerrar una llave izquierda.

ERROR 9 : Indica que dentro del operador llaves existe un paréntesis izquierdo que no fue cerrado.

ERROR 10 : En la definición existen dos operandos juntos o bien un operador seguido de un operador llave o paréntesis, por lo que faltó separarlos por un operador de selección o concatenación.

ERROR 11 : En la definición existen dos operadores paréntesis o llaves o alguna combinación de ellos tales que ninguno de los operadores de selección o concatenación se encuentra entre ellos.

ERROR 12 : Este es marcado cada vez que dentro de la definición del átomo se hace referencia a algún nombre de átomo o clase de símbolos el cual no ha sido definido antes.

ERROR 13 : Es marcado cada vez que en la definición del átomo existe una llamada a el mismo.

ERROR 14 : Es reportado cada vez que hay dos operadores juntos.

ERROR 15 : Es reportado cada vez que un operador de selección o concatenación aparece seguido de un punto y coma.

ERROR 16 : Es reportado cada vez que hay un operando seguido de un operador llave o paréntesis faltando entre ellos un operador de selección o concatenación.

ERROR 17 : Es reportado cada vez que hay dos operandos juntos faltando entre ellos un operador de selección o concatenación.

ERROR 18 : Es reportado cuando dentro de la definición del átomo aparece un operador llave dentro de otro operador llave.

ERROR 19 : Es reportado cuando dentro de un operador llave o paréntesis no se inicia con algúno de éstos o con alguna clase de símbolos o cuerda o átomo declarado antes.

CONCLUSIONES.

El sistema *CIM* presenta varias características importantes, desde la manera de especificación de los datos hasta las estructuras de datos empleadas para su desarrollo, que hacen ser a *CIM* un sistema *confiable, seguro, eficiente y "amigable "*. Todas estas características quedan respaldadas bajo las siguientes razones.

Al contar con pocos operadores para describir a la gramática, se hace posible describirla de manera fácil, logrando que el usuario tenga facilidad para especificar los datos de entrada, así mismo *CIM* cuenta con una rutina la cual reporta algún error, si éste existe en la gramática especificada por el usuario, lo que representa una ventaja para éste al momento de corregir los datos de entrada, lográndose de esta manera que el sistema *CIM* sea un siatema "amigable", en caso de existir un error, el sistema *CIM* deja de ejecutarse, ésta es una deficiencia que presenta *CIM*, pues sería conveniente reportar todos los errores de sintaxis que tenga la gramática. Por otro lado el sistema *CIM* puede ser empleado por personas que no posean conocimiento alguno de computación, así como de aquellos que posean un alto conocimiento de la computación, logrando de esta forma que *CIM* sea un sistema poderoso y sencillo en su uso.

Por otra parte, la finalidad de haber escrito a *CIM* para una minicomputadora como la *PDP-11/34* en vez de una microcomputadora, se debe a la falta de este tipo de software para estas máquinas, así mismo la decisión de escribir el sistema en lenguaje *C* se debió a factores como la portabilidad que nos ofrece este lenguaje, por ejemplo ya se cuenta con éste sistema para una microcomputadora con sistema operativo *msdos*, además de que el lenguaje *C* cuenta con estructuras que permiten realizar una buena administración de la memoria,también cuenta con funciones de biblioteca y operandos como son el operando de autodecremento y autoincremento, los cuales hacen que la ejecución del sistema sea mas rápida.

De igual forma el código generado *GENCIM* presenta estas características, también se encuentra escrito modularmente, donde los módulos son independientes entre sí, para que el usuario no tenga problema en el número de parámetros cuando decida hacer mejoras a *GENCIM* en el momento de integrarlo a un sistema mayor.

Haciendo una comparación entre *CIM* y *LEX*, se puede observar lo siguiente :

- El lenguaje para describir los átomos en CIM es pequeño y claro, mientras que el de LEX es más grande y complejo debido a su "potencialidad" teniendo como consecuencia que sea más difícil de aprender y manejar.

- Para obtener una mayor provecho de LEX se requiere tener conocimiento del lenguaje "C".

- CIM, al igual que LEX, representa el autómata determinístico de los átomos definidos a través de una tabla, sin embargo, la tabla generada por LEX generalmente es grande y ralas.

Con respecto a ALEX, las diferencias que se pueden mencionar, debido a la escasa información que se tiene de éste, son :

– El análizador léxico generado por ALEX está codificado en MODULA–2, lo que implica un restricción para ser usado por los alumnos de la facultad de ciencias, ya que éstos no estan familiarizados con este lenguaje, mientras que el analizador léxico generador por CIM se encuentra codificado en "C", lenguaje que cada dia es más popular entre la comunidad de la facultad de ciencias, lo que facilita su uso.

– Alex crea un autómata no determinístico para los átomos definidos, para después, generar el autómata determinístico correspondiente, método que podría repercutir en la ejecución haciendolo más lento.

En base a lo anterior el objetivo queda totalmente cubierto, sin embargo otro aspecto importante que se desprende de la tesis, y con lo cual no es posible cerrar las conclusiones sin hacerlo presente en éstas, es el siguiente. Como se puede apreciar la tesis presenta un capítulo el cual muestra en forma breve pero clara y sencilla de entender todo el medio ambiente bajo el cual se desarrollan los analizadores léxicos, por lo que el usuario puede emplear la tesis como notas de consulta para introducirse al mundo de los analizadores léxicos.

BIBLIOGRAFIA.

[LG80]: Legarreta, G.L.;*Compiladores*; Comunicaciones Internas, Fac. Ciencias; 1980.

[TS85]: Tremblay, J.P. y Sorenson, P.G.;*The theory and practice of compiler writing*; McCraw-Hill; 1985.

[UH79]: Ullman, J.D. y Hopcroft, J.E.;*Intoduction to automata theory, languages, and computation*; Addison Wesley; 1979.

[AU79]: Aho, A.V. y Ullman, J.D.;*Principles of compiler design*; Addison Wesley; 1979.

[DG80]: Dedoureck, J.M. y Gujar, U.G.;*"Scanner disgn"*; Software Practice and Experience, vol 10 , 959–972; 1980.

[LM75]: Lesk, M.E.; *"Lex & lexical analyzer generator"*; Computer Science Thechnical, No 32, Bell Laboratories; Oct. 1975.

[MH86]: Mössenböck H.; "Alex"; Sigplan Notices, Vol 21, No 5; May. 1986.