VNIVERSIDAD NACIONAL
AVFNOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

CUIDAD UNIVERSITARIA

PROGRAMAS LOGICOS
Y
GRAMATICAS LIBRES DE CONTEXTO

TESIS

QUE PARA OBTENER EL

TITULO DE MATEMATICO

PRESENTA

RAFAEL RAMIREZ MELENDEZ

Septiembre 1991

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

CONTENIDO

0. Introducción

1. Capitulo I. Un poco acerca de los lenguajes formales
    1. Cuerdas y Lenguajes
    2. Especificación finita de lenguajes
    3. Gramáticas
    4. Clasificación de gramáticas

2. Capítulo II. Otro poco acerca de programación lógica
    1. Lenguajes de primer orden
    2. Programas lógicos
    3. Sustituciones
    4. Resolución SLD

3. Capítulo III. Lenguajes formales y programación lógica
    1. Representación de listas
    2. Programas cadena

4. Capítulo IV. Análisis sintáctico y cartas
    1. Cartas
    2. La regla fundamental
    3. La regla de abajo hacia arriba
    4. Implementación

La teoría de lenguajes formales y la programación, generalmente son vistas como dos áreas inconexas. Esto se debe en parte, a que se han desarrollado independientemente. Sin embargo varios autores [6,7,8,9,10] han observado que existen puntos en común entre las dos. Por ejemplo, una sucesión de comandos en un programa puede verse como una cuerda de un lenguaje.

Nosotros vamos a centrar nuestro interés en la programación lógica y los lenguajes libres de contexto. Hay similitudes entre ellos que pueden enriquecer el uno al otro.

Es posible, como probaremos más adelante, codificar gramáticas libres de contexto en programas lógicos de manera que usemos a la programación lógica como instrumento para determinar qué cuerdas están en el lenguaje generado por la gramática. Este resultado es ya conocido pero, hasta donde sabemos, su demostración no ha sido publicada aún por otra persona.

Inversamente, hemos utilizado cierta herramienta de los lenguajes libres de contexto como procedimiento de prueba para programas lógicos orientados a estados. Hasta donde sabemos, dicha herramienta no ha sido empleada con anterioridad para ejecutar este tipo de programas.

VISTA GLOBAL DEL TRABAJO

Este trabajo está dividido en tres partes principales. Una trata de lenguajes formales (capítulos I y IV), otra es acerca de programación lógica (capítulo II), y la última (capítulos III y V)

CAPITULO I

UN POCO ACERCA DE LOS LENGUAJES FORMALES

Todo el mundo tiene una idea intuitiva bastante clara de lo que es un lenguaje. Lo que la mayoria de la gente entiende por lenguaje (a secas), los matemáticos lo llaman *Lenguaje Natural* para poder diferenciarlo de los *Lenguajes Formales.* Asi pues, el lenguaje natural es simplemente lo que las personas usamos para comunicarnos con los demás, ya sea oralmente o a través de un texto escrito. En nuestro caso es el español. Es fácil darse cuenta que para poder formar estructuras con sentido, tenemos que atenernos a ciertas reglas intrinsecas al lenguaje. Claramente "Escuela a voy la" no tiene sentido en español.

Por otra parte, existen los Lenguajes Formales. Son lenguajes creados por los matemáticos, en los que sus reglas para formar estructuras "gramaticalmente correctas" son muy precisas. Alguien podria pensar que los lenguajes formales son innecesarios dada la riqueza y poder de expresión de un lenguaje natural, pero no. Lo que pasa es que los lenguajes naturales son de alguna manera ambiguos. Dada una regla ortográfica siempre existen excepciones, algunas palabras tienen distintos significados dependiendo del contexto o del país donde se usen, y además los lenguajes van cambiando a través del tiempo, lo que obligaria a estar reformulando sus reglas. En los lenguajes formales no existe este tipo de ambigüedades debido a la manera tan precisa en que están definidos.

En este capítulo presentaremos la manera en que se definen y generan los lenguajes formales asi como algunas de sus propiedades

i) Base: $\lambda \in \Sigma^*$.

ii) Paso recursivo: Si $w \in \Sigma^*$ y $a \in \Sigma$ entonces $wa \in \Sigma^*$.

iii) Cerradura: $w \in \Sigma^*$ solo si se puede obtener de λ por un número finito de aplicaciones del paso recursivo ii).

Para cualquier alfabeto no vacío Σ, Σ^* contiene un número infinito de elementos. Si $\Sigma = \{a\}$, Σ^* contiene las cuerdas $\lambda, a, aa, aaa, \ldots$. La longitud de una cuerda w, intuitivamente es el número de elementos en esa cuerda o formalmente el número de aplicaciones del paso recursivo necesarias para construir la cuerda y se denota *long(w)*. Si Σ contiene n elementos, entonces hay n^k cuerdas de longitud k en Σ^*.

Un lenguaje está formado por cuerdas sobre un alfabeto. Normalmente se le ponen algunas restricciones a las cuerdas que componen el lenguaje. El español, por ejemplo, está formado por cuerdas de palabras que llamamos frases. No todas las cuerdas de palabras son frases en un lenguaje, sólo aquellas que satisfacen ciertas condiciones. Por lo tanto, un lenguaje es un subconjunto del conjunto de todas las cuerdas posibles sobre un alfabeto.

Definición

Un *lenguaje* sobre un alfabeto Σ es un subconjunto de Σ^*.

Puesto que los elementos de un lenguaje son cuerdas, es conveniente examinar las propiedades de las cuerdas y las operaciones entre ellas. La concatenación es la operación binaria que toma dos cuerdas y las "pega" para construir una nueva cuerda. La concatenación es la operación fundamental para generar cuerdas. Se puede definir formalmente por recursión sobre la longitud de la segunda cuerda en la concatenación.

Definición

Sea $u,v \in \Sigma^*$. La concatenación de u y v denotada generalmente por uv, es una operación binaria sobre Σ^* definida como sigue :

i) Base: Si $long(v)=0$ entonces $v=\lambda$ y $uv=u$

ii) Paso recursivo: Sea v una cuerda con $long(v)=n > 0$. Entonces $v=wa$, para alguna cuerda w con longitud $n-1$ y $a \in \Sigma$, y $uv=(uw)a$.

El resultado de la concatenación de u,v y w es independiente del orden en que se efectúen dichas concatenaciones. En matemáticas, a esta propiedad se le llama asociatividad. El siguiente teorema prueba que la concatenación es asociativa.

Teorema

Sean $u,v,w \in \Sigma^*$. Entonces $(uv)w = u(vw)$.

Demostración. La demostración es por inducción sobre la longitud de la cuerda w.

Base: $long(w)=0$. Entonces $w = \lambda$, y $(uv)w = uv$ por la definición de concatenación. Por otro lado, $u(vw)=u(v)=uv$.

Hipótesis inductiva: Supongamos que $(uv)w = u(vw)$ para toda cuerda w de longitud menor o igual que n.

Paso inductivo: Necesitamos probar que $(uv)w = u(vw)$, para toda cuerda w de longitud $n+1$. Sea w tal cuerda, entonces $w = xa$ para alguna cuerda x de longitud n y $a \in \Sigma$.

$(uv)w = (uv)(xa)$	Sustituyendo w por xa
$= ((uv)x)a$	por la definición de concatenación
$= (u(vx))a$	hipótesis inductiva
$= u((vx)a)$	por la definición de concatenación
$= u(v(xa))$	por la definición de concatenación
$= u(vw)$	sustituyendo xa por w

Como la asociatividad garantiza el mismo resultado independientemente del orden en que se realizen las operaciones, se

omitirán los paréntesis en una secuencia de concatenaciones. Se usarán exponentes para abreviar la concatenación de una cuerda con ella misma. La concatenación es claramente no conmutativa. Si $u=ab$ y $v=c$, $uv=abc$ y $vu=cab$.

Las subcuerdas se pueden definir usando concatenación. Intuitivamente, u es una subcuerda de v si u "está metida en" v. Formalmente, u es una *subcuerda* de v si existen cuerdas x y y tales que $v=xuy$. Un *prefijo* de v es una subcuerda u donde la x es la cuerda vacía en la descomposición de v. Es decir, $v=uy$. Análogamente, u es un *sufijo* de v si $v=xu$.

1.2 ESPECIFICACION FINITA DE LENGUAJES.

Un lenguaje se ha definido como un conjunto de cuerdas sobre un alfabeto. La especificación de un lenguaje requiere que la descripción de las cuerdas que lo forman no sea ambigua. Un lenguaje finito puede ser definido explícitamente enumerando sus elementos. Los lenguajes infinitos pueden ser definidos a partir de conjuntos finitos usando las operaciones sobre conjuntos.

Definición

La concatenación de lenguajes X y Y , que denotaremos por XY, es el lenguaje

$$XY = \{uv \mid u \in X \text{ y } v \in Y\}$$

La concatenación de X con él mismo n veces, se denota generalmente por X^n. X^0 se define como $\{\lambda\}$.

Ejemplo

Sea $X=\{a,b,c\}$ y $Y=\{abb,ba\}$.

$XY = \{aabb, babb, cabb, aba, bba, cba\}$.

$X^0 = \{\lambda\}$.

$X^1 = X = \{a,b,c\}$.

asegura la presencia de *bb* en cada cuerda de L. Los conjuntos $\{a,b\}^*$ permiten cualquier número de *a*'s y de *b*'s en cualquier orden, precediendo y siguiendo la ocurrencia de *bb*.

1.3 GRAMATICAS.

Hasta el momento, hemos visto cómo se pueden generar lenguajes arbitrarios a partir de conjuntos finitos usando operaciones de conjuntos como unión, concatenación, y cerradura de Kleene (que no es más que uniones de concatenaciones). Pues bien, las gramáticas son otro medio para generar lenguajes; de hecho son probablemente la clase más importante de generadores de lenguajes. Una gramática es un sistema matemático usado para definir un lenguaje, así como para darle estructura a sus cuerdas.

Una gramática para un lenguaje L usa dos conjuntos finitos y disjuntos de símbolos. Estos son el conjunto de símbolos *no terminales*, que se denota usualmente por N, y el alfabeto Σ sobre el cual está definido el lenguaje. A los símbolos de Σ se les llama generalmente símbolos *terminales*. Los símbolos no terminales son usados en la generación de las cuerdas del lenguaje.

El corazón de una gramática es un conjunto finito P de *producciones*, como generalmente se les llama, que describen la manera en la cual las cuerdas del lenguaje son generadas. Una producción es simplemente una pareja de cuerdas, o más precisamente, un elemento de $(N\cup\Sigma)^*N(N\cup\Sigma)^* \times (N\cup\Sigma)^*$. Es decir, la primera componente es cualquier cuerda conteniendo al menos un símbolo no terminal, y la segunda componente es cualquier cuerda. Nótese que aquí al referirnos a una cuerda no nos estamos

refiriendo exclusivamente a cuerdas del lenguaje, puesto que éstas pueden contener símbolos no terminales que no pertenecen al alfabeto del lenguaje.

Por ejemplo, la pareja (AB,CDE) puede ser una producción. Si una cierta cuerda α puede ser generada por la gramática, y α tiene a AB como subcuerda, entonces podemos formar una cuerda nueva β remplazando una instancia de la subcuerda AB en α por CDE. Se dice entonces que β es *generada* por la gramática. Por ejemplo, si FGABH puede ser generada, entonces FGCDEH también puede ser generada. El lenguaje definido por la gramática es el conjunto de cuerdas que constan sólo de símbolos terminales y que pueden ser generados comenzando con una cuerda particular que consta de un símbolo distinguido, generalmente denotado por S.

Si (α,β) es una producción, por convención usamos la abreviación $\alpha \rightarrow \beta$ en lugar de (α,β). Damos a continuación la definición formal de gramática.

Definición

Una *gramática* es una cuarteta $G=(N,\Sigma,P,S)$ dónde

1. N es un conjunto finito de símbolos *no terminales*.
2. Σ es un conjunto finito de símbolos *terminales*, disjunto de N.
3. P es un subconjunto finito de $(N\cup\Sigma)^*N(N\cup\Sigma)^* \times (N\cup\Sigma)^*$. Un elemento (α,β) en P se escribirá $\alpha \rightarrow \beta$ y se le llamará *producción*.
4. S es un símbolo distinguido de N llamado símbolo *inicial*.

Ejemplo

Un ejemplo de gramática es $G_1=(\{A,S\},\{a,b\},P,S)$ donde P consiste en

$$S \rightarrow aAb$$
$$aA \rightarrow aaAb$$
$$A \rightarrow \lambda$$

Los símbolos no terminales son A y S, y los terminales *a* y *b*.

Una gramática define un lenguaje de manera recursiva. Definimos recursivamente un tipo especial de cuerda llamado *forma sentencial* de una gramática $G=(N,\Sigma,P,S)$ como sigue:

i) S es forma sentencial.

ii) si $\alpha\beta\gamma$ es una forma sentencial y $\beta \longrightarrow \delta$ está en P, entonces $\alpha\delta\gamma$ también es forma sentencial.

Una forma sentencial de G que no contiene símbolos no terminales se llama *sentencia generada por G*.

El *lenguaje generado* por una gramática G, que denotaremos por $L(G)$, es el conjunto de sentencias generadas por G.

Ahora introduciremos terminología que usaremos en lo sucesivo. Sea $G=(N,\Sigma,P,S)$ una gramática. Definimos una relación $\underset{G}{\Rightarrow}$ (*reescribe directamente*) sobre $(N\cup\Sigma)^*$ como sigue: Si $\alpha\beta\gamma$ es una cuerda en $(N\cup\Sigma)^*$ y $\beta \longrightarrow \delta$ es una producción en P, entonces $\alpha\beta\gamma \underset{G}{\Rightarrow} \alpha\delta\gamma$. Usaremos $\overset{*}{\underset{G}{\Rightarrow}}$ (*reescribe*) para denotar un número finito de reescrituras directas. Cuando sea claro a qué gramática nos estamos refiriendo, eliminaremos el subíndice G de $\Rightarrow$ y $\overset{*}{\Rightarrow}$.

Ejemplo

Consideremos la gramática G_1 del último ejemplo y la siguiente reescritura: $S \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aabb$. Esto es, que en el primer paso, S es reemplazada por aAb de acuerdo con la producción $S \longrightarrow aAb$. En el segundo paso, aA es reemplazada por $aaAb$, y en el tercero, A es reemplazada por λ. Así que $S \overset{*}{\Rightarrow} aabb$ y $aabb$ está en $L(G_1)$.

Por otra parte, usaremos la abreviación $\alpha \longrightarrow \beta_1|\beta_2|\ldots|\beta_n$ para representar las producciones $\alpha \longrightarrow \beta_1$, $\alpha \longrightarrow \beta_2, \ldots, \alpha \longrightarrow \beta_n$.

También usaremos las siguientes convenciones para cuando se

trate de representar símbolos o cuerdas relacionadas con una gramática:

1. a,b,c,d,e representan terminales o símbolos del alfabeto.

2. p,q,u,v,w,x,y,z representan cuerdas que están compuestas solo por terminales.

3. A,B,C,D y S representan no terminales; S representa al símbolo inicial.

4. U,V,W,X,Y,Z representan terminales o bien no terminales.

5. $\alpha,\beta,\gamma,\ldots$ representan cuerdas compuestas por terminales y no terminales.

Los subíndices no cambian esta convención. Respetando la convención, podemos especificar una gramática simplemente listando sus producciones. Así pues, la gramática G_1 se puede especificar sin mencionar los conjuntos de terminales y de no terminales o el símbolo inicial, como:

$$S \longrightarrow aAb$$
$$aA \longrightarrow aaAb$$
$$A \longrightarrow \lambda$$

Ejemplos

1) Sea G_1 una gramática definida por

$$S \longrightarrow aSBC \mid abC$$
$$CB \longrightarrow BC$$
$$bB \longrightarrow bb$$
$$bC \longrightarrow bc$$
$$cC \longrightarrow cc$$

Un ejemplo de reescritura en esta gramática es

$S \Rightarrow aSBC \Rightarrow aabCBC \Rightarrow aabBCC \Rightarrow aabbCC \Rightarrow aabbcC \Rightarrow aabbcc.$

2) Sea G_2 la grámatica con producciones

$$S \longrightarrow CD \qquad Ab \longrightarrow bA \qquad Aa \longrightarrow aA$$
$$C \longrightarrow aCA \qquad Ba \longrightarrow aB$$
$$C \longrightarrow bCB \qquad Bb \longrightarrow bB$$
$$AD \longrightarrow aD \qquad C \longrightarrow \lambda$$
$$BD \longrightarrow bD \qquad D \longrightarrow \lambda$$

Tenemos la siguiente reescritura en G

$S \Rightarrow CD \Rightarrow aCAD \Rightarrow abCBAD \Rightarrow abBAD \Rightarrow abBaD \Rightarrow ababBD \Rightarrow ababD \Rightarrow abab.$

1.4 CLASIFICACION DE GRAMATICAS.

Las gramáticas pueden ser clasificadas de acuerdo al formato de sus producciones. Sea $G=(N,\Sigma,P,S)$ una gramática.

Definición

A G se le llama

1. *Lineal derecha* si cada producción de P es de la forma $A \longrightarrow uB$ o $A \longrightarrow u$, dónde A y B están en N y u está en Σ^*.

2. *Libre de contexto* si cada producción de P es de la forma $A \longrightarrow \alpha$ donde A está en N y α está en $(N \cup \Sigma)^*$.

3. *Sensible al contexto* si cada producción de P es de la forma $\alpha \longrightarrow \beta$, donde $long(\alpha) \leq long(\beta)$.

A una gramática sin restricciones se le llama *irrestringida*.

Ejemplos

1) Un ejemplo de gramática lineal derecha es la gramática con producciones $S \longrightarrow aS|bS|\lambda$. Esta gramática genera el lenguaje $(a,b)^*$.

2) Un ejemplo interesante de gramática libre de contexto es $G=(\{E,T,F\},\{a,+,*,(,)\},P,E)$, donde P consiste en las producciones

$$E \longrightarrow E+T|T$$
$$T \longrightarrow T*F|F$$
$$F \longrightarrow (E)|a$$

El lenguaje generado por esta gramática es el conjunto de expresiones aritméticas que se pueden construir usando los símbolos $a,+,*,($ y $)$. Un ejemplo de reescritura en esta gramática es $E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*F \Rightarrow a+F*F \Rightarrow a+a*F \Rightarrow a+a*a$.

Dicho sea de paso, en una gramática libre de contexto, una *reescritura izquierda* es una reescritura en la que en cada reescritura directa, el simbolo no terminal que se reescribe es el simbolo de hasta la izquierda. Así que la reescritura anterior es una reescritura izquierda

3) La gramática G_1 presentada con anterioridad, es claramente una gramática sensible al contexto.

4) La gramática G_2 presentada antes es irrestringida. Obsérvese que no es ni lineal derecha, ni libre de contexto, ni sensible al contexto.

Nótese que de acuerdo con nuestra definición, toda gramática lineal derecha es también libre de contexto. La definición de gramática sensible al contexto no permite producciones de la forma $A \longrightarrow \lambda$, comúnmente llamadas producciones λ. Así que una gramática libre de contexto que tenga alguna producción λ, no es sensible al contexto.

Por convención, si un lenguaje L puede ser generado por una gramática de tipo x, entonces a L se le llama lenguaje de tipo x. Así pues, los lenguajes libres de contexto que no contienen a la cuerda vacia forman un subconjunto propio de los lenguajes sensibles al contexto. Por otra parte, hay que hacer notar que dada una gramática de cierto tipo, el lenguaje generado por esa gramática puede en algunos casos ser generado por una gramática menos poderosa. Como un ejemplo, la gramática libre de contexto

$$S \longrightarrow AS|\lambda$$
$$A \longrightarrow a|b$$

genera el lenguaje $\{a,b\}^*$, que como vimos, puede ser generada también por una gramática lineal derecha.

De la clasificación de gramáticas que dimos, las gramáticas libres de contexto son las más importantes en lo que se refiere a aplicaciones a lenguajes de programación y compilación. Una gramática libre de contexto puede ser usada para especificar la mayor parte de la estructura sintáctica de un lenguaje de programación. Por estas y otras razones, las gramáticas libres de contexto han sido ampliamente estudiadas y cualquier resultado original que se obtenga acerca de ellas resulta interesante.

Las gramáticas libres de contexto son simplemente las que en el lado izquierdo de todas sus producciones aparece un y solo un símbolo no terminal.

Definición

Dos gramáticas se dice que son *equivalentes* si generan el mismo lenguaje.

Ahora vamos a definir un tipo de gramática libre de contexto que usaremos en el capitulo 4 y que tiene la propiedad de que cualquier gramática libre de contexto es transformable en una equivalente de este tipo.

Definición

Las producciones de una gramática libre de contexto $G=(N,\Sigma,P,S)$ se dice que están *normalizadas* si cumplen lo siguiente:

1. Para cada símbolo terminal a en Σ existe una producción de la forma $A\rightarrow a$ en P, con alguna $A \in N$.
2. Sea $A \in N$. Si $A\rightarrow W$ es una producción con uno o más símbolos del lado derecho, entonces $W \in N^{*}$.

Diremos que una gramática libre de contexto es una *gramática normalizada* si todas sus producciones están normalizadas.

Supongamos por un momento que tenemos una gramática libre de

contexto $G_1=(N,\Sigma,P,S)$ y que necesitamos obtener una gramática normalizada $G_2=(N',\Sigma',P',S')$ equivalente. Lo que tenemos que hacer es modificar las producciones en P de tal manera que en P' nos queden puras producciones normalizadas y que el lenguaje generado por ambos conjuntos sea el mismo.

Bueno, empezamos por hacer una copia de N y Σ, y las ponemos en N' y Σ' respectivamente. Consideramos que P' es inicialmente vacio y hacemos S'=S. Tomamos la primera producción en P y si es la producción vacia, entonces se la agregamos a P'. Si no, entonces es de la forma $A\rightarrow xW$, donde $x \in N$ o $x \in \Sigma$, y $W \in (N\cup\Sigma)^*$. Consideramos el primer símbolo del lado derecho de la producción que en este caso es x. Aquí se presentan dos casos, o x es terminal o no lo es. Si x es terminal, entonces introducimos un nuevo símbolo no terminal X que no esté ya en N' y se lo agregamos a N', cambiamos nuestra producción original por la producción $A\rightarrow XW$ y agregamos la producción $X\rightarrow x$ a P'. Si x es no terminal, entonces no hacemos nada. Procediendo de esta manera con los demás símbolos del lado derecho de la producción $A\rightarrow XW$, es claro que al terminar, esta producción ha sido transformada en una de la forma $A\rightarrow W'$ donde $W' \in N^*$. Agregamos esta última a P'.

Haciendo lo mismo con todas las producciones restantes de P, obtenemos solamente producciones normalizadas en P' y es fácil ver que G y G' son equivalentes.

Puesto que en lo anterior consideramos una gramática libre de contexto arbitraria, podemos afirmar que, dada una gramática libre de contexto, existe siempre una gramática normalizada equivalente.

CAPITULO II

OTRO POCO ACERCA DE PROGRAMACION LOGICA

En el capítulo anterior definimos los lenguajes formales en una forma general. En éste, sin embargo, abordaremos un cierto tipo de lenguajes formales llamados *Lenguajes de Primer Orden*. Este tipo de lenguajes ha adquirido gran importancia dentro de las matemáticas debido a su simplicidad, precisión y gran poder expresivo. Con ellos es posible representar todos los enunciados que tienen sentido en matemáticas así como sus reglas de razonamiento, con solo un número reducido de símbolos y reglas para combinarlos. Pues bien, la base de la *Programación Lógica* son los lenguajes de primer orden.

La programación lógica surgió a principios de los setentas como resultado directo del trabajo realizado en inteligencia artificial. Construir sistemas automáticos de deducción es un objetivo central de la inteligencia artificial. En 1972, Kowalski y Colmerauer descubrieron que la lógica de primer orden, o al menos un subconjunto considerable de ella, podía ser usada como lenguaje de programación. Esta idea fue revolucionaria, porque hasta 1972, la lógica había sido usada en computación solo como especificación. Kowalski demostró que la lógica tiene una interpretación como procedimiento, que la hace muy efectiva como lenguaje de programación. Una cláusula de un programa puede interpretarse como la definición de un procedimiento.

Uno de los resultados prácticos más importantes que ha producido la investigación en computación hasta el momento, es el lenguaje de programación PROLOG, basado en un subconjunto de la

lógica de primer orden. La mayoría de los sistemas de programación lógica disponibles hoy en día son o intérpretes o compiladores de PROLOG. Sin embargo, la programación lógica de ninguna manera esta limitada a PROLOG. Es esencial encontrar la manera de programar en subconjuntos más grandes de la lógica de primer orden. Dichos sistemas no tendrían por qué estar basados en la resolución que usa PROLOG y podrían usar muchas reglas de inferencia.

En este capítulo consideraremos solo sistemas de programación lógica basados en resolución. Primero definiremos los lenguajes de primer orden, para luego concentrarnos en el subconjunto de cláusulas de Horn, que es la base de los programas lógicos. El objetivo final del capítulo es introducir formalmente la resolución SLD. Este capítulo está sacado del libro de programación lógica de Lloyd [2].

2.1 LENGUAJES DE PRIMER ORDEN.

La lógica de primer orden puede ser interpretada de dos maneras distintas: por su sintaxis o por su semántica. A la sintaxis le conciernen las fórmulas generadas por la gramática de un lenguaje formal, así como los aspectos de prueba teórica de dicho lenguaje. Por otra parte está la semántica, que se ocupa del significado de los símbolos que aparecen en las fórmulas.

Para poder hablar de la lógica de primer orden, lo primero que hay que hacer es definir algún lenguaje de primer orden. Así que empecemos por definir el alfabeto.

Definición

El alfabeto de un lenguaje de primer orden consiste en:

1. Variables

2. Constantes
3. Símbolos de función
4. Símbolos de predicado
5. Conectivos
6. Cuantificadores
7. Símbolos de puntuación

Las clases 5 a 7 son iguales para cualquier alfabeto, mientras que las clases 1 a 4 varían dependiendo del alfabeto. En un alfabeto, sólo las clases 2 y 3 pueden ser vacías. Normalmente, las variables se denotan por $u,v,\ldots z$, las constantes por a,b y c, las funciones por f,g y h y los predicados por p,q y r. Los conectivos son $\sim,\wedge,\vee,\rightarrow,\leftrightarrow$, mientras que los cuantificadores son $\exists$ y $\forall$. Finalmente, los símbolos de puntuación son (,) y ,. Para evitar demasiados paréntesis en un fórmula, por convención se adopta la siguiente jerarquía de precedencia: primero $\sim,\forall$ y $\exists$, luego $\vee$, luego $\wedge$, y al último $\rightarrow$ y $\leftrightarrow$.

Definición

Un *término* se define inductivamente como sigue:

1. Una variable es un término.

2. Una constante es un término.

3. Si f es un símbolo de función n-aria y $t_1,\ldots,t_n$ son términos, entonces $f(t_1,\ldots,t_n)$ es un término.

4. Sólo son términos los producidos por 1, 2 y 3.

Definición

Una *fórmula* se define inductivamente como sigue:

1. Si p es un símbolo de predicado n-ario y $t_1,\ldots,t_n$ son términos, entonces $p(t_1,\ldots,t_n)$ es una fórmula (normalmente llamado *átomo*).

2. Si F y G son fórmulas, entonces $(\sim F),(F\wedge G),(F\vee G),(F\rightarrow G)$ y $(F\leftrightarrow G)$ tambien son fórmulas.

3. Si F es una fórmula y x es una variable, entonces $(\forall x\ F)$ y $(\exists x\ F)$ son fórmulas.

4. Sólo son fórmulas las producidas por 1, 2 y 3.

En algunos casos será conveniente escribir la fórmula $(F \rightarrow G)$ como $(G \leftarrow F)$.

Ejemplo

$\forall x \exists y(p(x,y) \rightarrow q(x))$ y $\sim \exists x(p(x,a) \wedge q(f(x)))$ son fórmulas.

Informalmente, el significado de los conectivos y cuantificadores es el siguiente: $\sim$ es negación, $\wedge$ es conjunción (y), $\vee$ es disyunción (o), $\rightarrow$ es implicación y $\leftrightarrow$ es equivalencia. $\exists$ es el cuantificador existencial, asi que "$\exists x$" significa "existe una x", mientras que $\forall$ es el cuantificador universal por lo que "$\forall x$" significa "para toda x". Por lo tanto las fórmulas del ejemplo anterior quieren decir "para toda x, existe una y tal que si $p(x,y)$, entonces $q(x)$" y "no existe x tal que $p(x,a)$ y $q(f(x))$".

Definición

El *lenguaje de primer orden* generado por un alfabeto como el definido con anterioridad, consiste en el conjunto de todas las fórmulas construidas a partir de los símbolos del alfabeto.

2.2 PROGRAMAS LOGICOS.

Definición

Sea F una fórmula, el *alcance* de $\forall x$ en $\forall x F$ es F. El alcance de $\exists x$ se define análogamente. Una variable *ocurre acotada* en una fórmula si aparece inmediatamente después de un cuantificador, o se encuentra dentro del alcance de uno que tenga a la misma variable inmediatamente después. De otra manera, la variable se dice que

ocurre libre.

Ejemplo

En la fórmula $\exists x\ p(x,y) \wedge q(x)$,las primeras dos ocurrencias de x son acotadas mientras que la tercera es libre.

Definición

Una *literal* es un átomo o la negación de un átomo. Una *literal positiva* es simplemente un átomo. Una *literal negativa* es la negación de un átomo.

Definición

Una *cláusula* es una fórmula de la forma

$$\forall x_1 \ldots \forall x_s (L_1 \vee \ldots \vee L_m)$$

Donde cada L_i es una literal y $x_1,\ldots,x_s$ son todas las variables que ocurren en $L_1 \vee \ldots \vee L_m$.

Ejemplo

$\forall x \forall y \forall z\ (p(x,z) \vee \sim q(x,y) \vee \sim r(y,z))$ y

$\forall x \forall y\ (\sim p(x,y) \vee r(f(x,y),a))$ son cláusulas.

Como en programación lógica las cláusulas son muy comunes, es conveniente adoptar una notación especial. Así que denotaremos a la cláusula

$$\forall x_1 \ldots \forall x_s (A_1 \vee \ldots \vee A_k \vee \sim B_1 \vee \ldots \vee \sim B_n)$$

por $$A_1,\ldots,A_k \leftarrow B_1,\ldots,B_n$$

donde $A_1,\ldots,A_k,B_1,\ldots,B_n$ son átomos y $x_1,\ldots,x_s$ son todas las variables que ocurren en estos átomos.

De esta manera en la notación de cláusulas, todas las variables se consideran universalmente cuantificadas, las comas en la condición $B_1,\ldots,B_n$ denotan conjunción y las comas en la conclusión $A_1,\ldots,A_k$ denotan disyunción. Estas convenciones están justificadas porque

$$\forall x_1 \ldots \forall x_s (A_1 \vee \ldots \vee A_k \vee \sim B_1 \vee \ldots \vee \sim B_n)$$

es lógicamente equivalente a

$$\forall x_1 \ldots \forall x_s (A_1 \vee \ldots \vee A_k \leftarrow B_1 \wedge \ldots \wedge B_n)$$

Definición

Una *cláusula definida* es una cláusula de la forma

$$A \leftarrow B_1, \ldots, B_n$$

que contiene exactamente una literal positiva (que en este caso es A). A es llamada la *cabeza* y $B_1, \ldots, B_n$ el *cuerpo* de la cláusula definida.

Definición

Una *cláusula unitaria* es una cláusula de la forma

$$A \leftarrow$$

es decir, una cláusula definida con cuerpo vacío.

El significado informal de $A \leftarrow B_1, \ldots, B_n$ es "para cualquier valor de cualquier variable, si $B_1, \ldots B_n$ son todos verdaderos, entonces A es verdadera". Así que si n>0, una cláusula definida es condicional. Sin embargo, una cláusula unitaria $A \leftarrow$ es incondicional. Su significado informal es "para cualquier valor de cualquier variable, A es verdadera".

Definición

Un *programa lógico* es un conjunto finito de cláusulas definidas.

Definición

Una *meta* es una cláusula de la forma

$$\leftarrow B_1, \ldots, B_n$$

Es decir, una cláusula definida con cabeza vacía. Generalmente, a cada B_i (i=1,...,n) se le llama *submeta*.

Si $x_1, \ldots, x_s$ son las variables que occurren en la meta

$$\leftarrow B_1, \ldots, B_n$$

entonces la notación de cláusulas para la meta anterior no es más que una abreviación de

$$\forall x_1 \ldots \forall x_s (\neg B_1 \vee \ldots \vee \neg B_n)$$

que es lógicamente equivalente a

$$\neg \exists x_1 \ldots \exists x_s (B_1 \ \ldots \ B_n)$$

Definición

La *cláusula vacía*, normalmente denotada por □, es la cláusula con cabeza y cuerpo vacíos. Esta cláusula se interpreta como una contradicción.

Definición

Una *cláusula de Horn* es o bien una cláusula definida o una meta. Es decir, es una cláusula con a lo más una literal positiva.

2.3 SUSTITUCIONES.

Definición

Una *sustitución* θ es un conjunto finito de la forma $\{v_1/t_1,\ldots,v_n/t_n\}$ con $n \geq 0$, donde cada v_i es una variable, cada t_i es un término distinto de v_i y las variables $v_1,\ldots,v_n$ son distintas.

Definición

Una *expresión* es un término, o una literal, o una conjunción o disyunción de literales.

Definición

Sea $\theta=\{v_1/t_1,\ldots,v_n/t_n\}$ una substitución y sea E una expresión. Entonces $E\theta$ es la expresión que se obtiene a partir de E reemplazando simultáneamente cada ocurrencia de la variable v_i en E por el término $t_i (i=1,\ldots,n)$.

Ejemplo

Sea $E=p(x,y,f(a))$ y $\theta=\{x/b,y/x\}$, entonces $E\theta=p(b,x,f(a))$.

Si $S=\{E_1,\ldots,E_n\}$ es un conjunto finito de expresiones y θ es una sustitución, entonces $S\theta$ denota al conjunto $\{E_1\theta,\ldots,E_n\theta\}$.

Definición

Sean $\theta=\{u_1/s_1,\ldots,u_m/s_m\}$ y $\tau=\{v_1/t_1,\ldots,v_n/t_n\}$ sustituciones. la *composición* $\theta\tau$ de θ y τ es la substitución que se obtiene del conjunto

$$\{u_1/s_1\tau,\ldots,u_m/s_m\tau,v_1/t_1,\ldots,v_n/t_n\}$$

borrando las $u_i/s_i\tau$ para las cuales $u_i=s_i\tau$, y las v_j/t_j para las cuales $v_j\in\{u_1,\ldots,u_m\}$.

Definición

Sea S un conjunto finito de términos y átomos. Se dice generalmente que una substitución θ es un *unificador* para S si $S\theta$ contiene solo un elemento. Un unificador θ para S se llama *unificador más general* para S, si para cualquier unificador τ para S, existe una substitución γ tal que $\tau=\theta\gamma$.

Ejemplo

$\{p(f(x),z)\ ,\ p(y,a)\}$ es unificable porque $\tau=\{y/f(a),x/a,z/a\}$ es un unificador. Un unificador más general es $\theta=\{y/f(x),z/a\}$. Nótese que $\tau=\theta\{x/a\}$.

Definición

Sea P un programa, G una meta $\leftarrow A_1,\ldots,A_k$ y θ una substitución para variables de G. Diremos que θ es una *sustitución respuesta* para $P\cup\{G\}$ si $\forall x_1\ldots\forall x_n((A_1\ldots A_k)\theta)$ es consecuencia lógica de P. Donde $x_1,\ldots,x_n$ son todas las variables que ocurren en $A_1,\ldots,A_n$.

2.4 RESOLUCION SLD.

Definición

Una *regla de selección* es una función que va del conjunto de secuencias de metas al conjunto de átomos de un programa. El valor de la función, evaluada en una secuencia de metas es siempre un átomo, generalmente llamado el *átomo seleccionado*, en la última meta de la secuencia.

Definición

Sea G_i la meta $\leftarrow A_1,\ldots,A_m,\ldots,A_k$, C_{i+1} la cláusula $A\leftarrow B_1,\ldots,B_q$ y R una regla de selección. Entonces G_{i+1} se *deriva* de G_i y C_{i+1} usando un unificador más genera θ_{i+1} vía R si se cumplen las siguientes condiciones:

1. A_m es el átomo seleccionado por la regla de selección R.
2. $A_m\theta_{i+1}=A\theta_{i+1}$ (es decir, θ_{i+1} es un unificador más general para A_m y A)
3. G_{i+1} es la meta $\leftarrow(A_1,\ldots,A_{m-1},B_1,\ldots B_q,A_{m+1},\ldots,A_k)\theta_{i+1}$.

Se dice generalmente que G_{i+1} es el *resolvente* de G_i y C_{i+1} por θ_{i+1}.

Definición

Sea P un programa, G una meta y R una regla de selección. Una *derivación SLD* de P∪{G} vía R consiste en una secuencia posiblemente infinita $G_0=G,G_1,\ldots$ de metas, una secuencia $C_1,C_2,\ldots$ de cláusulas de P y una secuencia $\theta_1,\theta_2,\ldots$ de unificadores más generales tales que G_{i+1} se deriva de G_i y C_{i+1} usando θ_{i+1} vía R.

Definición

Una *refutación SLD* de P∪{G} vía R es una derivación SLD finita de P∪{G} vía R que tiene a la cláusula vacía □ como la última meta en

la derivación. Si $G_n=\square$, se dice que la refutación tiene longitud n.

Una derivación SLD finita puede ser exitosa o fallida. Una derivación exitosa es simplemente una refutación. Una derivación fallida es una que termina con una meta no vacía con la propiedad que el átomo seleccionado en esta meta no unifica con la cabeza de ninguna cláusula del programa.

Teorema

Sea P un programa y G una meta. supongamos que P∪{G} es insatisfacible [1]. Entonces existe una regla de selección R y una refutación SLD de P∪{G} vía R.

Demostración Ver [2,p.44].

El teorema nos dice que si P∪{G} es insatisfacible, entonces existe una refutación de P∪{G} usando alguna regla de selección R. Sin embargo, hasta el momento, no tenemos control sobre R. El siguiente resultado muestra que la regla de selección puede ser fijada a priori y después si P∪{G} es insatisfacible, siempre podemos encontrar una refutación que use la regla de selección dada. A este hecho se le llama *independencia de la regla de selección*.

Teorema (Independencia de la regla de selección)

Sea P un programa, G una meta y R una regla de selección. Supongamos que existe una refutación SLD de P∪{G} vía R. Sea R' cualquier regla de selección. Entonces existe una refutación SLD

[1] Puesto que definir INSATISFACIBLE involucra introducir la teoría de modelos y ésto no es el propósito de esta tesis, damos una referencia [3] que en caso de necesidad, el lector puede consultar.

vía R'.

Demostración Ver [2,p.47].

Ahora consideraremos las estrategias posibles que un sistema de programación lógica puede usar para encontrar una refutación. Para ello definiremos nuestro espacio de búsqueda, normalmente llamado árbol SLD.

Definición

Sea P un programa, G una meta y R una regla de selección. Entonces el *árbol SLD* para P∪{G} vía R se define como sigue:

1. Cada nodo del árbol es una meta (posiblemente vacia).

2. La raiz del árbol es G.

3. Sea $\leftarrow A_1,\ldots,A_m,\ldots,A_k (k\geq 1)$ un nodo en el árbol y supongamos que A_m es el átomo seleccionado por R. Entonces este nodo tiene un descendiente inmediato por cada cláusula $A\leftarrow B_1,\ldots,B_q$ tal que A_m y A son unificables. El descendiente inmedato es

$$\leftarrow (A_1,\ldots,A_{m-1},B_1,\ldots,B_q,A_{m+1},\ldots,A_k)\theta$$

donde θ es un unificador más general para A_m y A.

4. Los nodos que son la cláusula vacia no tienen descendientes.

Cada rama del árbol SLD es una derivación de P∪{G}. Las ramas que corresponden a derivaciones exitosas, derivaciones infinitas y derivaciones fallidas, generalmente se llaman *ramas exitosas*, *ramas infinitas* y *ramas fallidas* respectivamente.

Ejemplo

Consideremos el programa

$$p(x,z)\leftarrow q(x,y),\ p(y,z) \qquad (1)$$
$$p(x,x)\leftarrow \qquad (2)$$
$$q(a,b)\leftarrow \qquad (3)$$

y la meta $\leftarrow p(x,b)$. La figura 1 muestra un árbol SLD para este programa y esta meta.

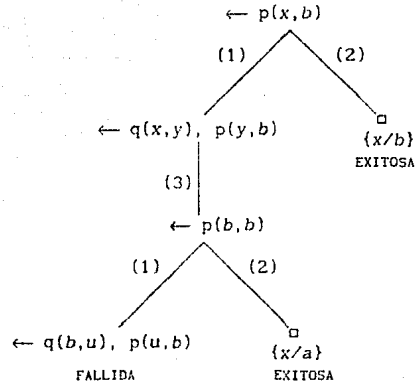


FIG 1

El árbol corresponde a la regla de selección que usa PROLOG (selecciona el átomo de hasta la izquierda). No hay ramas infinitas y las ramas exitosas y fallidas están indicadas, así como la cláusula con cuya cabeza está unificado el átomo seleccionado a cada paso de derivación.

Definición

Una *estrategia de búsqueda* es una estrategia para encontrar ramas exitosas en árboles SLD. Un *procedimiento de refutación SLD* está definido por una regla de selección y una estrategia de búsqueda.

Como nota al margen, es conveniente decir que los sistemas PROLOG estándares usan la regla de selección que selecciona siempre el átomo de hasta la izquierda conjuntamente con una estrategia de búsqueda *a lo profundo*, es decir una estrategia de búsqueda que examina primero en detalle una rama, antes de considerar una segunda. Esta estrategia está implementada por medio de una pila de metas.

Como se habrán dado cuenta, a simple vista la teoria de lenguajes formales y la programación lógica parecen ser dos áreas practicamente disconexas e independientes. De hecho, aperecieron así. Por una parte, los lenguajes formales surgieron como consecuencia de la necesidad de evitar ambigüedades dentro de los lenguajes, mientras que la programación lógica surgió como resultado directo del trabajo realizado en inteligencia artificial para construir demostradores automáticos de teoremas.

Este capítulo está encargado de mostrar lo contrario. Los lenguajes formales y la programación lógica tienen aspectos en común que no han sido estudiados en detalle. El corazón del capítulo es un teorema que relaciona estas dos áreas.

3.1 REPRESENTACION DE LISTAS.

Una *lista* es una estructura de datos muy usada en programación lógica y consiste en una secuencia finita de objetos.

La notación que usaremos para representar listas es la misma que usa PROLOG, es decir, $[a_1,a_2,a_3,\ldots]$ donde las a_i son los objetos de una secuencia.

Notación

1. $[\alpha|\beta]$ denota a el término $f(\alpha,\beta)$.
2. [] denota a la lista vacia.
3. $[\alpha|[\beta|\gamma]]$ se simplifica a $[\alpha,\beta|\gamma]$.
4. $[\alpha|[]]$ se simplifica a $[\alpha]$.

3.2 PROGRAMAS CADENA.

Los *programas cadena* son un subconjunto propio de los programas lógicos, es decir, son programas lógicos cuyas cláusulas están restringidas a un formato especial. Lo interesante de estos programas es que a cada gramática libre de contexto se le puede asociar un programa de este tipo que sirve como instrumento para determinar si una cuerda está o no en el lenguaje generado por dicha gramática.

Definición

Un programa lógico se dice que es un *programa cadena*, si todas las cláusulas que lo definen son de la forma

$$p(x_0,x_n) \leftarrow q_1(x_0,x_1), q_2(x_1,x_2), \ldots, q_n(x_{n-1},x_n) \qquad (n \geq 0)$$

o bien, son cláusulas unitarias de la forma

$$p([a|x],x) \leftarrow$$

donde p y q_i son símbolos de predicado binarios, x y x_i son variables y a es una constante.

Ahora vamos a definir cómo se le puede asociar un programa cadena a una gramática libre de contexto dada.

En lo que resta del capítulo, usaremos A,B,E,S para denotar símbolos (terminales o no) de una gramática , $\mathbb{A},\mathbb{B},\mathbb{E},\mathbb{S}$ para denotar símbolos de predicado en un programa cadena, y X para denotar variables de dicho programa. (Esto sigue siendo cierto haya o no subíndices dentro de las expresiones.)

Definición

Sea $\mathfrak{G}=(N,\Sigma,P,S)$ una gramática libre de contexto. $\mathfrak{P}$ es el *programa cadena asociado* a $\mathfrak{G}$, si está definido por las siguientes cláusulas:

1) Si la producción $E \rightarrow \lambda$ está en P, entonces la cláusula

$$\mathbb{E}(X,X) \leftarrow$$

está en $\mathfrak{P}$.

Lo primero que hay que observar es que en una gramática libre de contexto, si hay una reescritura $S \overset{*}{\Rightarrow} \alpha$, entonces hay una reescritura izquierda $S \overset{*}{\Rightarrow} \alpha$. Por lo tanto asumamos que $S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_m = \alpha$ es una reescritura izquierda.

Base:

Sea P_1: $S \rightarrow A_1 \ldots A_n$ la producción usada en $S \Rightarrow \alpha_1$ y sea

$$C_1: S(X_0, X_n) \leftarrow A_1(X_0, X_1), \ldots, A_n(X_{n-1}, X_n)$$

la cláusula en $\mathcal{P}$ que corresponde a dicha producción. Unificamos nuestra meta $\leftarrow S([\alpha' | X], X)$ con la cabeza de C_1 por la sustitución $\theta_1 = \{X_0/[\alpha' | X_n], X/X_n\}$. De esta manera nuestra nueva meta es el resultado de aplicar la sustitución θ_1 al cuerpo de C_1, es decir

$$\leftarrow A_1([\alpha' | X_n], X_1), \ldots, A_n(X_{n-1}, X_n) \qquad (1)$$

Donde, por la forma en que esta construido $\mathcal{P}$, $A_1, \ldots, A_n$ son los símbolos de predicado en $\mathcal{P}$ que corresponden respectivamente a los símbolos de $\mathcal{G}$ (terminales o no) que aparecen en α_1.

Hipótesis de inducción:

Ahora supongamos que nuestra meta después de j-1 pasos de derivación tiene la forma (1) y que los símbolos de predicado en dicha meta, corresponden respectivamente a los símbolos de $\mathcal{G}$ que aparecen en α_{j-1}.

Paso inductivo:

Fijémonos en la j-ésima producción

$$E_1 \rightarrow A_1^1 A_2^1 \ldots A_k^1$$

usada para construir la reescritura de α, es decir, la producción empleada en $\alpha_{j-1} \Rightarrow \alpha_j$. Y sea

$$E_1(X_0^1, X_k^1) \leftarrow A_1^1(X_0^1, X_1^1), \ldots, A_k^1(X_{k-1}^1, X_k^1)$$

la cláusula en $\mathcal{P}$ que le corresponde. Es claro que de esta manera, E_1 tiene que ser por lo menos alguna A_i de (1). Unificando $E_1(X_0^1, X_k^1)$ y la $A_i(X_{i-1}, X_i)$ de hasta la izquierda, por $\theta_j = \{X_0^1/X_{i-1}, X_k^1/X_i\}$, nuestra j-1 ésima meta se transforma en

$$\leftarrow A_1([\alpha'|X_n],X_1),\ldots,A_{i-1}(X_{i-2},X_{i-1}),A^1_1(X_{i-1},X^1_1),A^1_2(X^1_1,X^1_2),\ldots,$$
$$A^1_k(X^1_{k-1},X_i),A_{i+1}(X_i,X_{i+1}),\ldots,A_n(X_{n-1},X_n)$$

reetiquetando esta j-ésima meta obtenemos

$$\leftarrow B_1([\alpha'|X_n],X_1),\ldots,B_{k+n-1}(X_{k+n-2},X_{k+n-1})$$

que tiene la forma (1). Además $B_1,\ldots,B_{k+n-1}$ son los símbolos de predicado en $\mathfrak{P}$ que corresponden respectivamente a los símbolos de $\mathfrak{G}$ que aparecen en α_j.

Así que, para j=m, tenemos una meta de la forma (1) donde todos los símbolos de predicado B_i corresponden a símbolos terminales en $\mathfrak{G}$ (de hecho son los símbolos que aparecen en α).

Ahora, para obtener la cláusula α a partir de esta m-ésima meta, hay que observar lo siguiente.

Por la forma en que se construyó $\mathfrak{P}$, para cada símbolo terminal a_i en $\mathfrak{G}$ existe una cláusula unitaria $A_i([a_i|X],X)$ en $\mathfrak{P}$. Así que dada nuestra última meta

$$\leftarrow A_1([\alpha'|X_n],X_1),\ldots,A_s(X_{s-1},X_s)$$

unificamos $A_1([\alpha'|X_n],X_1)$ y $A_1([a_1|X],X)$ con la sustitución

$$\tau_1=\{X/[a_2,\ldots,a_s|X_n],\ X_1/[a_2,\ldots,a_s|X_n]\}$$

y la nueva meta es

$$\leftarrow A_2([a_2,\ldots,a_s|X_n],X_2),\ldots,A_s(X_{s-1},X_s).$$

Unificando ahora $A_2([a_2,\ldots,a_s|X_n],X_2)$ y $A_2([a_2|X],X)$ con

$$\tau_2=\{X/[a_3,\ldots,a_s|X_n],\ X_2/[a_3,\ldots,a_s|X_n]\}$$

obtenemos la meta

$$\leftarrow A_3([a_3,\ldots,a_s|X_n],X_3),\ldots,A_s(X_{s-1},X_s).$$

Procediendo de esta manera, derivamos al final la cláusula vacía, y por la correctez de la resolución SLD, $\mathfrak{P}\models S([\alpha'|X],X)$.

($\Leftarrow$) supongamos $\mathcal{P} \models S([\alpha'|X],X)$. Por inducción vamos a probar que $S \overset{*}{\Rightarrow} \alpha$.

Como $\mathcal{P} \models S([\alpha'|X],X)$, entonces, por completez de la resolución SLD, el árbol de resolución de $\mathcal{P} \cup \{\leftarrow S([\alpha'|X],X)\}$ via alguna $\mathcal{R}$, deriva $\square$.

Por el teorema de independencia de la regla de selección, sabemos que no importa qué regla de selección consideremos, en su árbol existirá al menos una rama que deriva $\square$.

Nosotros vamos a considerar la regla $\mathcal{R}$ que elige a los átomos de hasta la izquierda de la meta dada. Fijándonos en la rama $\mathcal{D}$ que deriva $\square$ en este árbol, vamos a construir la reescritura izquierda $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_m = \alpha$.

Base:

Sea $\leftarrow S([\alpha'|X],X)$ la raiz del árbol y sea

$$\mathcal{C}_1 : S(X_0,X_n) \leftarrow A_1(X_0,X_1),\ldots,A_n(X_{n-1},X_n)$$

la cláusula seleccionada en $\mathcal{D}$ para unificar con la raiz por

$$\theta_1 = \{X_0/[\alpha'|X_n],X/X_n\}.$$

Ahora nos fijamos en la producción $\mathcal{P}_1 : S \rightarrow A_1 \ldots A_n$ que corresponde a $\mathcal{C}_1$ y la tomamos como la producción para construir $S \Rightarrow \alpha_1$. Nuestra meta activa en $\mathcal{D}$ ahora es

$$\leftarrow A_1([\alpha'|X_n],X_1),\ldots,A_n(X_{n-1},X_n) \qquad (2)$$

Hipótesis de inducción:

Supongamos ahora que tenemos ya construida nuestra reescritura hasta α_{j-1}, es decir $S \overset{*}{\Rightarrow} \alpha_{j-1}$, y que además nuestra meta en $\mathcal{D}$ tiene la forma

$$\leftarrow A_1([\alpha'_h|X_n],X_1),A_{l+1}(X_l,X_{l+1}),\ldots,A_g(X_{g-1}X_g) \qquad (3)$$

donde $\alpha'_h = a_h,\ldots,a_s$ y $\alpha' = a_1,\ldots,a_h,\ldots,a_s$. Es decir que, de alguna manera hemos ya eliminado h-1 átomos cuyos símbolos de predicado corresponden respectivamente a los primeros h-1 símbolos en α.

Paso inductivo:

Dada la $\mathcal{R}$ que elegimos, $A_1([\alpha'_h|X_n],X_1)$ es el átomo en (3) que se escoge para buscarle una unificación. Aquí se presentan dos casos:

Caso 1 A_i es un símbolo de predicado que corresponde a un símbolo terminal a_i en $\mathfrak{G}$. Como la rama $\mathfrak{J}$ que estamos considerando deriva la cláusula α, entonces el átomo

$$A_i([\alpha'_h|X_n],X_i)$$

tuvo que haber unificado con la cabeza de alguna cláusula del programa, pero dado que A_i corresponde a un símbolo terminal en $\mathfrak{G}$, dicho átomo tuvo que haber unificado con la cabeza de la cláusula unitaria

$$A_h([a_h|X],X)\leftarrow$$

en $\mathfrak{P}$, por la sustitución

$$\theta_1=\{X/[a_{h+1},\ldots,a_s|X_n],X_i/[a_{h+1},\ldots,a_s|X_n]\}$$

Por lo tanto el h-ésimo símbolo de α es a_i, por lo que $a_i=a_h$.

la meta activa en $\mathfrak{J}$ se convierte en

$$\leftarrow A_{i+1}([a_{h+1},\ldots,a_s|X_n],X_{i+1}),\ldots,A_g(X_{g-1},X_g)$$

que sigue teniendo la forma (3).

Caso 2 A_i es un símbolo de predicado que corresponde a un símbolo no terminal E_i en $\mathfrak{G}$. Por lo tanto existe en $\mathfrak{P}$

$$\mathfrak{C}_i : \quad E_i(X^1_0,X^1_k)\leftarrow A^1_1(X^1_0,X^1_1),\ldots,A^1_k(X^1_{k-1},X^1_k)$$

que fue usada para unificar con $A_i([\alpha'_h|X_n],X_i)$ por

$$\theta_1=\{X^1_0/[\alpha'_h|X_n],X^1_k/X_i\}$$

y que produce la meta

$$\leftarrow A^1_1([\alpha'_h|X_n],X^1_1),A^1_2(X^1_1,X^1_2),\ldots,A^1_k(X^1_{k-1},X^1_i),A_{i+1}(X_i,X_{i+1}),\ldots,A_n(X_{n-1},X_n).$$

que tiene la forma (3). En este caso, tomamos la producción

$$\mathfrak{P}_i: E_i\rightarrow A^1_1\ldots A^1_k$$

en $\mathfrak{G}$ que corresponde a $\mathfrak{C}_i$, como la producción usada en $\alpha_{j-1}\Rightarrow\alpha_j$.

Como la rama $\mathfrak{J}$ que estamos considerando deriva α y dado lo anterior, derivar α no es más que aplicar el caso 1 s veces, es decir eliminar átomos $A_i(X,Y)$ donde A_i es el símbolo de predicado de $\mathfrak{P}$ que corresponde al símbolo

terminal a_i de α en $\mathfrak{G}$, entonces

$$S \Rightarrow A_1 \ldots A_n \Rightarrow \ldots \Rightarrow a_1 \ldots a_s = \alpha$$

y por lo tanto $S \overset{*}{\Rightarrow} \alpha$.

Ejemplo

Sea $\mathfrak{G}=(\{S\},\{a,b,e\},P,S)$ la gramática libre de contexto con producciones

P_1: $S \longrightarrow aSb$

P_2: $S \longrightarrow e$

El programa cadena asociado a esta gramática consta de las cláusulas

$\mathfrak{C}_1$: $S(X_0,X_3) \leftarrow A(X_0,X_1), S(X_1,X_2), B(X_2,X_3)$

$\mathfrak{C}_2$: $S(X_0,X_1) \leftarrow E(X_0,X_1)$

$\mathfrak{C}_3$: $A([a|X],X) \leftarrow$

$\mathfrak{C}_4$: $B([b|X],X) \leftarrow$

$\mathfrak{C}_5$: $E([e|X],X) \leftarrow$

Sea $\alpha=aaebb$ una cuerda en $L(\mathfrak{G})$.

Tomemos la reescritura izquierda

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaebb=\alpha.$$

Vamos a ver que $\mathfrak{P} \cup \{\leftarrow S([\alpha'|X],X)\}$ deriva la cláusula $\square$.

Es claro que P_1: $S \longrightarrow aSb$ es la producción usada en $S \Rightarrow aSb$ y $\mathfrak{C}_1$ es la cláusula en $\mathfrak{P}$ que le corresponde. Unificamos la cabeza de esta cláusula con la meta $\leftarrow S([\alpha'|X],X)$ por la sustitución $\theta_1=\{X_0/[\alpha'|X_3],X/X_3\}$. De esta manera obtenemos la meta

$$\leftarrow A([\alpha'|X_3],X_1), S(X_1,X_2), B(X_2,X_3)$$

Nótese que A,S y B corresponden respectivamente a los símbolos que aparecen en $aSb=\alpha_1$. La siguiente producción usada en la reescritura es otra vez P_1. Unificando el átomo $S(X_1,X_2)$ en la última meta con la cabeza de $\mathfrak{C}_1$ por la sustitución $\theta_2=\{X_{00}/X_1, X_{33}/X_2\}$ obtenemos la meta

$$\leftarrow A([\alpha'|X_3],X_1), A(X_1,X_{11}), S(X_{11},X_{22}), B(X_{22},X_2), B(X_2,X_3)$$

Nótese que A,A,S,B y B corresponden trespectivamente a los símbolos que aparecen en $aaSbb=\alpha_2$.

La última producción usada en la reescritura es P_2 : S⟶ e. Unificando la cabeza de la cláusula C_2 que le corresponde con el átomo $S(X_{11},X_{22})$, por la sustitución $\theta_3=\{X_{000}/X_{11},X_{111}/X_{22}\}$, obtenemos la meta

$$\leftarrow A([\alpha'|X_3],X_1),A(X_1,X_{11}),E(X_{11},X_{22}),B(X_{22},X_2),B(X_2,X_3)$$

Nótese otra vez que A,A,E,B y B corresponden respectivamente a los símbolos que aparecen en $\alpha=aaebb$.

Ahora, unificamos el átomo $A([\alpha'|X_3],X_1)$ con la cláusula en P,

$$C_3\text{: } A([a|X],X)\leftarrow$$

por la sustitución

$$\tau_1=\{X/[a,e,b,b|X_3],X_1/[a,e,b,b|X_3]\}$$

y obtenemos la meta

$$\leftarrow A([a,e,b,b|X_3],X_{11}),E(X_{11},X_{22}),B(X_{22},X_2),B(X_2,X_3)$$

Unificando el átomo $A([a,e,b,b|X_3],X_{11})$ con la cláusula C_3, por la sustitución

$$\tau_2=\{X/[e,b,b|X_3],X_{11}/[e,b,b|X_3]\}$$

obtenemos la meta

$$\leftarrow E([e,b,b|X_3],X_{22}),B(X_{22},X_2),B(X_2,X_3)$$

Unificando análogamente con las cláusulas C_5,C_4 y C_4 en ese orden, obtenemos la cláusula □.

Es fácil ver que el inverso también se cumple, es decir que si tomamos una rama que derive la cláusula □, podemos construir la reescritura $S \overset{*}{\Rightarrow} \alpha$.

Como consecuencia del teorema anterior, podemos decir que una estrategia de búsqueda sobre árboles SLD, a través de los programas cadena, puede hacer el papel de analizador sintáctico para las gramáticas libres de contexto.

En este sentido, PROLOG se usa con frecuencia como analizador

sintáctico. Nosotros hemos demostrado que cualquier estrategia de búsqueda sobre árboles SLD puede servir como analizador sintáctico para grámaticas libres de contexto.

Hasta donde sabemos, la demostración de este resultado no ha sido pubicada aún por otra persona.

Los analizadores sintácticos son usados para determinar si una cuerda pertenece o no a un lenguaje. La mayoría de los analizadores, dada una gramática y una cuerda a analizar, construyen lo que normalmente se llama su árbol de análisis sintactico. Existen, como es de suponer, muchas maneras de construir estos árboles y por lo tanto hay muchas clases de analizadores sintácticos. En este capítulo trataremos en detalle sólo un tipo especial de analizador llamado *analizador por cartas*. Este algoritmo funciona para cualquier gramática normalizada y por lo tanto, para cualquier gramática libre de contexto. El capítulo está extraído de [4].

3.1 CARTAS.

Intuitivamente, las cartas son gráficas dirigidas que no contienen ciclos de longitud mayor a uno y cuyas aristas tienen nombre. De hecho, los nombres de las aristas son producciones de la gramática con un símbolo especial extra en la parte derecha de la producción. Dicho símbolo es un punto.

Definición

Sea G una gramática. Una *producción con punto* es un objeto de la forma $A \rightarrow X_1 X_2 \ldots X_k \circ X_{k+1} \ldots X_m$ si $A \rightarrow X_1 \ldots X_m$ es una producción de G. El punto entre X_k y X_{k+1} es un símbolo del metalenguaje que no es ni terminal ni no terminal. El entero k puede ser cualquier número entre 0 y m incluyendo 0 (en cuyo caso el $\circ$ es el primer símbolo) o m (en cuyo caso es el último).

Definición

Una *carta* es un conjunto de estructuras llamadas *arcos*, cada una de las

cuales es de la forma

<inicio, fin, nombre>

donde *inicio* es cualquier entero, *fin* es un entero mayor o igual a inicio y *nombre* es una producción con punto.

El nombre está dividido en tres partes: el lado izquierdo de la producción con punto, lo que está antes del ∘, y lo que está después. A las últimas dos se les identifica también como lo encontrado y lo que hay que encontrar, respectivamente.

Ejemplo

Algunos arcos en una carta pueden ser {<0,2,S⟶A∘B>,
<2,3,B⟶C∘AS>,
<3,5,A⟶a∘>,
<5,8,S⟶AB∘>,...}

que gráficamente se representa por

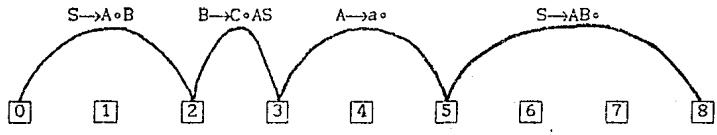


FIG 1

Definición

Un arco es un *arco inactivo* si en su nombre el ∘ aparece hasta el final. De otra manera es un *arco activo*.

3.2 LA REGLA FUNDAMENTAL.

Ahora, tengamos en cuenta que un carta es simplemente un conjunto de arcos y consideremos el problema de hacer funcionar un analizador sintáctico en la estructura que acabamos de describir. Supongamos que nos encontramos a la mitad del proceso de búsqueda que nos proporciona tal algoritmo y nuestra

carta contiene a los arcos del ejemplo anterior (figura 1), entre otros. Por claridad, hemos omitido los demás arcos que deberían estar en la carta para que éste hubiera podido desarrollarse hasta este estado.

Lo que tenemos aquí son dos arcos activos y dos inactivos. Estos últimos representan producciones completamente aplicadas. El primer arco activo representa una hipótesis sobre la producción S→AB, que para completar su aplicación ha encontrado ya un símbolo A y está buscando una B. Análogamente, el segundo arco activo representa una hipótesis sobre la producción B→CAS que ya encontró una C y está buscando una A y una S para completar su aplicación. Consideremos el primer arco activo. Para satisfacer su hipótesis, necesitamos encontrar uno inactivo con una B en el lado izquierdo del nombre y que inicie en el vértice 2. Pero supongamos que no existe tal arco. Lo que sí tenemos es una hipótesis sobre la posible existencia de tal arco, pero, hasta no confirmar la hipótesis, no podemos hacer nada para remediar la situación del primer arco activo. Ahora fijémonos en el segundo arco activo. Tenemos una hipótesis que busca una A como lado izquierdo del nombre de un arco inactivo que inicie en el vértice 3. Ahora sí tenemos tal arco. Esto quiere decir que nuestra hipótesis ha avanzado, aunque no ha sido confirmada completamente. Podemos representar esta confirmación parcial aumentando otro arco activo al carta. Tal arco es <2,5,B→CA∘S>. Esta es una hipótesis que busca un arco inactivo con una S en el lado izquierdo del nombre que inicie en el vértice 5. Y nosotros tenemos uno de esos en nuestra carta, así que esta hipótesis está completamente confirmada y podemos aumentar el arco <2,8,B→CAS∘> al carta. Volviendo a nuestro primer arco activo, podemos observar que ahora sí, su hipótesis está completamente confirmada puesto que ya existe en el carta una arco inactivo con una B en el lado izquierdo de su nombre que inicia en el vértice 2. Así que aumentamos el arco inactivo

<0,8,S→AB∘> a nuestro carta que se ve como en la fig.2.

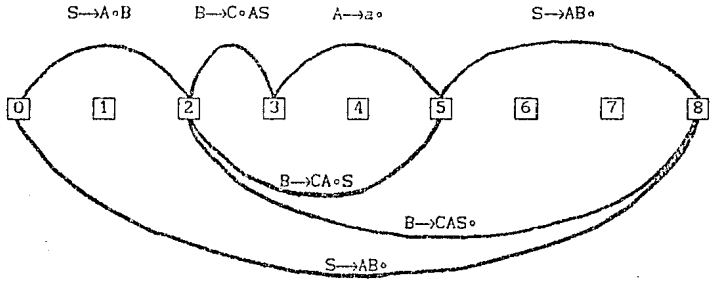


FIG 2

El hecho de que tengamos un arco inactivo con S como lado izquierdo del nombre, que cubre todo el ancho de la gráfica, nos indica que la cuerda que estábamos analizando efectivamente es generada por la gramática dada.

El proceso que acabamos de describir informalmente representa la esencia del algoritmo por cartas. Lo único que hicimos fue aplicar tres veces la misma regla: Si un arco activo se encuentra con uno inactivo del tipo deseado, entonces agrega uno nuevo que cubra a ambos. Esto se llama la *regla fundamental* del algoritmo por cartas. Para remediar la imprecisión de esta formulación informal de la regla, definimos:

Regla Fundamental

Si la carta contiene a los arcos $\langle i,j,A\rightarrow W_1 \circ B W_2\rangle$ y $\langle j,k,B\rightarrow W_3 \circ\rangle$, donde A y B son símbolos no terminales y W_1,W_3 y W_3 son secuencias (posiblemente vacías) de símbolos terminales o no, entonces agréguese el arco $\langle i,k,W_1 B \circ W_2\rangle$ a la carta.

Esta regla no especifica si el arco nuevo es activo o inactivo, pero no es grave puesto que esto depende de si W_2 es vacío o no.

Por otra parte, no podemos aplicar la regla fundamental a una carta que

no contenga arcos. Necesitamos que exista al menos uno activo y uno inactivo para que algo suceda. Para eso está la inicialización, para asegurar que halla algunos arcos inactivos.

3.3 INICIALIZACION.

Supongamos por el momento que tenemos la gramática siguiente: $G_1=(\{S\},\{a,b,c\},P,S)$ con producciones

$$S \rightarrow aSb$$
$$S \rightarrow c$$

La transformamos en una gramática normalizada equivalente con $G_2=(\{S,A,B,C\},\{a,b,c\},P',S)$ donde P'es

$$S \rightarrow ASB$$
$$S \rightarrow C$$
$$A \rightarrow a$$
$$B \rightarrow b$$
$$C \rightarrow c$$

Y supongamos que queremos saber si $\alpha=acb$ esta en el lenguaje generado por G_1 y G_2. Podemos crear inmediatamente la carta de la fig.3.

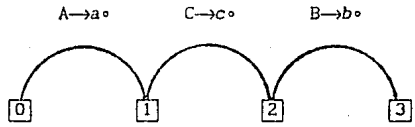


FIG 3

De esta manera hemos inicializado la carta. Normalmente a los arcos de la figura 3 se les denomina la *base* de la carta.

3.4 LA REGLA DE ABAJO HACIA ARRIBA.

Los arcos que al inicializar el carta creamos no son suficientes para que el analizador sintáctico empiece a trabajar. Necesitamos todavía agregar algunos arcos activos si es que esperamos que algo resulte al aplicar la regla fundamental. Una manera simple de asegurar que tales arcos sean agregados es la siguiente:

Regla de abajo hacia arriba

Cada vez que se agregue un arco $\langle i,j,C\rightarrow W_1\circ\rangle$ al carta, para cada producción en la gramática de la forma $B\rightarrow CW_2$, agréguese también el arco $\langle i,i,B\rightarrow\circ CW_2\rangle$.

La regla de abajo hacia arriba se aplica cuando algunos arcos inactivos son agregados a la carta. Si hubiéramos aplicado esta regla mientras inicializábamos el carta de la fig.3, habríamos obtenido la carta de la fig.4.

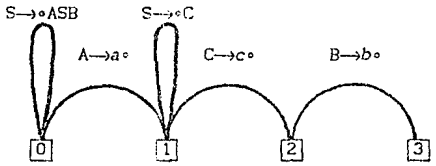


FIG 4

Ahora sí, la regla fundamental puede funcionar en esta carta y cada vez que ésta agregue un arco inactivo, la regla de abajo hacia arriba probablemente podría ser aplicada, proporcionando más arcos para que la regla fundamental funcione. De hecho, la regla de abajo hacia arriba y la regla fundamental son todo lo que necesitamos para asegurar que todos los posibles análisis sean encontrados.

3.5 IMPLEMENTACION.

Cada arco activo representa una hipótesis que necesita ser explorada. Si se confirma, aunque sea parcialmente, está en posición de generar más hipótesis mediante la regla fundamental y la regla de abajo hacia arriba. Con una computadora ordinaria, que ejecuta acciones serialmente, tenemos que decidir en qué orden deben ser consideradas las hipótesis. Al implementar un analizador por cartas, es conveniente tener una estructura de datos, una especie de agenda, para almacenar las hipótesis a explorar y los arcos por agregar a la carta. De esta manera podemos evitar investigar hipótesis idénticas.

Una posibilidad es ver a la agenda como una pila: Cada vez que produzcamos arcos nuevos, los ponemos en el tope de la pila. Después, cuando tengamos que seleccionar un arco para trabajar con él, tomamos el que esté en el tope.

Imaginemos que tenemos una pila a_1, a_2, a_3 de arcos, donde a_1 está en el tope. Al sacar a_1 de la pila podemos encontrar que al agregar este arco a la carta, son generados más arcos, digamos a_{11}, a_{12}, a_{13}. Colocándolos en el tope de la pila, tenemos una nueva pila $a_{11}, a_{12}, a_{13}, a_2, a_3$. Es claro que de esta manera, no vamos a considerar al arco a_2 hasta haber terminado con todas las hipótesis generadas directa o indirectamente por a_1. Esto es lo que se llama en ocasiones un *analizador a lo profundo.*

Otra estrategia es ver a la agenda como una cola, que es como una pila con la diferencia que los arcos nuevos se agregan en el extremo opuesto al lado de donde son retirados. Al analizador que resulta de usar esta estrategia se le denomina con frecuencia *analizador a lo ancho.*

En el listado 1 del apéndice A presentamos la implementación en PROLOG de un analizador sintáctico por cartas para la gramática normalizada G_2

descrita anteriormente. Se trata de un analizador a lo profundo, aunque es muy fácil convertirlo en un analizador a lo ancho, intercambiando simplemente el orden de dos argumentos en una cláusula.

Las *gráficas activas* [5] son una notación gráfica similar a los diagramas de flujo ordinarios, con una ventaja importante: las gráficas activas tienen la capacidad de representar problemas no deterministicos, mientras que el poder de representatividad de los diagramas de flujo está limitado a problemas deterministicos. Una gráfica activa es una gráfica dirigida con los arcos etiquetados con relaciones binarias, que especifica un comportamiento por medio de trayectorias. Aquí definiremos la noción de trayectoria de un nodo a otro por medio de cláusulas de lógica. Lo interesante es que estas cláusulas son ejecutables como programas lógicos que se pueden implementar fácilmente en una computadora.

5.1 GRAFICAS ACTIVAS.

Definición

Una *gráfica activa* está formada por un conjunto E de estados y una gráfica dirigida que tiene etiquetado cada arco con una relación binaria en E, llamada *comando*. La gráfica debe incluir dos nodos, no necesariamente distintos, llamados *inicial* y *final*.

Definición

Sea $\mathcal{G}$ una gráfica activa. Una *computación* es una sucesión no vacía y posiblemente infinita

$$\ldots,(n_i,x_i),(n_{i+1},x_{i+1}),\ldots$$

de parejas de la forma (nodo,estado), tal que x_i y x_{i+1} están relacionados por el comando C_i que etiqueta el arco que va de n_i a n_{i+1} en $\mathcal{G}$. La sucesión

$C_1, C_2, \ldots$

de comandos se llama *camino*.

Definición

La *relación binaria* asociada a un camino es la composición de todos los comandos del camino. La *relación computada* por una gráfica activa es la unión de las relaciones binarias asociadas a los caminos que van del nodo inicial al nodo final.

Dado que las gráficas activas son una notación gráfica más general comparada con los diagramas de flujo convencionales, en principio, para cualquier problema que esté representado por un diagrama de flujo, existe una gráfica activa que lo representa.

Ahora vamos a ver cómo construimos la gráfica activa correspondiente a un problema representado por un diagrama de flujo.

5.2 TRADUCCION DE DIAGRAMAS DE FLUJO A GRAFICAS ACTIVAS.

Consideremos, por ejemplo, el diagrama de flujo para calcular factoriales mostrado en la figura 1.

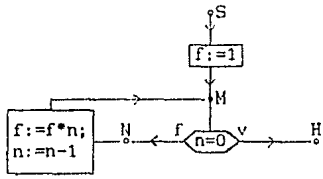


FIG 1

Este diagrama tiene dos variables, f y n, que toman valores dentro de los naturales. Así que, la gráfica activa que corresponda a este diagrama de flujo, tendrá como dominio el conjunto de parejas

de números naturales. Cada variable está asociada a una coordenada.

En una gráfica activa que emula un diagrama de flujo, los comandos corresponden a instrucciones. Los diagramas de flujo tienen dos clases de instrucciones: las asignaciones y las condiciones. En el ejemplo anterior, "f:=1" y "f:=f*n; n:=n-1" son asignaciones, mientras que "n=0" es una condición. Consideremos cada instrucción por separado.

Aunque normalmente una asignación sólo cambia una componente del estado, el comando correspondiente es una relación binaria entre tuplas con valores para todas las variables del programa, incluyendo aquellas que no cambian. El comando correspondiente a una asignación es una relación entre estados que relaciona cada estado anterior a la ejecución de la asignación con el estado obtenido al efectuarse ésta. Así, cada asignación es traducida a un comando, como se ve en la figura 2.

Una condición, por otro lado, es modelada por un comando que no cambia el estado. Nótese que las cajas condicionales en los diagramas de flujo tienen dos salidas, y conforme se realiza una computación, solo una de ellas es tomada. Esto sugiere que la traducción de una caja condicional sean dos salidas de un nodo complementarias entre sí. (Esta traducción se atribuye a Karp [10].) Entonces, al traducir diagramas de flujo en gráficas activas, las cajas condicionales dan lugar a dos comandos, complementarios entre sí, como en la figura 2.

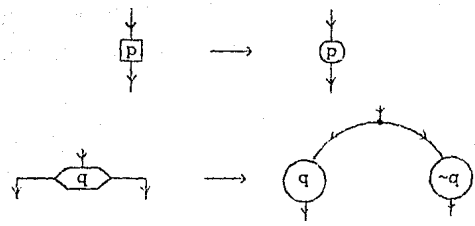


FIG 2

Ejemplo

La siguiente gráfica activa es la traducción del diagrama de flujo de la figura 1.

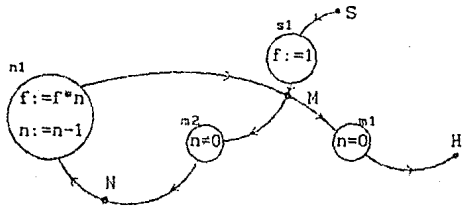


FIG 3

En este ejemplo, una computación podría ser

$$(N,\langle 1,3\rangle),(M,\langle 3,2\rangle),(N,\langle 3,2\rangle)$$

donde las parejas son de la forma (Nodo,<f,n>) y la trayectoria correspondiente n_1, m_1.

Una computación del nodo inicial S al nodo final H, podría ser

$$(S,\langle 10,3\rangle),(M,\langle 1,3\rangle),(N,\langle 1,3\rangle),(M,\langle 3,2\rangle),(N,\langle 3,2\rangle),$$
$$(M,\langle 6,1\rangle),(N,\langle 6,1\rangle),(M,\langle 6,0\rangle),(H,\langle 6,0\rangle).$$

y su trayectoria correspondiente $s_1, m_1, n_1, m_1, n_1, m_1, n_1, m_2$.

Las gráficas activas generalizan en dos aspectos a los diagramas de flujo ordinarios. Primero, las gráficas activas son no deterministicas, ya que pueden existir varias maneras de

extender las computaciones. Esto pasa cuando cuando el estado de la última pareja (nodo, estado) de una computación, ocurre como primera componente de un elemento en dos o más comandos que etiqueten arcos de salida de ese nodo. Esta situación no se presenta en los diagramas de flujo.

En segundo lugar, las gráficas activas pueden tener computaciones *fallidas* que no puedan ser extendidas. Esto pasa cuando el estado de la última pareja (nodo,estado) no está en el dominio de ninguna de las relaciones que etiquetan los arcos de salida del nodo. (Los diagramas de flujo pueden tener instrucciones con salidas indefinidas, como es el caso de una asignación que involucre una división entre cero, pero se tratan como excepciones a nivel de meta-lenguaje.)

5.3 REPRESENTACION DE TRAYECTORIAS EN LOGICA.

Tanto en una gráfica activa como en un diagrama de flujo, se obtienen computaciones a partir de trayectorias que se encuentran en ellos. Ahora daremos una definición en lógica de trayectoria.

Kowalski propuso una representación en lógica de las gráficas dirigidas usando un predicado cuyos argumentos son nodos de la gráfica. Llamaremos *arco* a dicho predicado. Un átomo sin variables *arco*(A,B) expresa que hay un arco del nodo A al nodo B. Por *trayectoria* vamos a entender una secuencia de uno o más nodos tales que hay un arco de un nodo al siguiente. Va a ser útil referirse a trayectorias cuyo último nodo es un nodo determinado. Diremos que dicho nodo H tiene la propiedad de ser nodo final y asumiremos la afirmación *final*(H)←. Definimos ahora una *trayectoria hacia atrás* como una trayectoria tal que el último

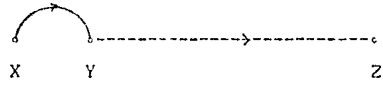
nodo es el final.

Utilizaremos el predicado *trayectoria*(X,Y) que afirma que hay una trayectoria hacia atrás de X a Y. Las cláusulas

trayectoria(X,X)← *final*(X) (2)

trayectoria(X,Z)← *arco*(X,Y), *trayectoria*(Y,Z) (3)

caracterizan las trayectorias hacia atrás. La cláusula (2) dice que una trayectoria hacia atrás puede consistir en un solo nodo. La cláusula (3), por otro lado, dice que para toda X,Y,Z, si hay una trayectoria hacia atrás de Y a Z, y un arco de X a Y, entonces hay una trayectoria hacia atrás de X a Z. Podemos ilustrar esta cláusula así:



Análogamente, podríamos definir una trayectoria hacia adelante como una trayectoria tal que el primer nodo es un nodo determinado, llamado inicial.

Consideremos al programa lógico formado por la descripción de la gráfica, junto con la afirmación que define al nodo final y las cláusulas que describen trayectorias hacia atrás. El problema de encontrar una trayectoria hacia atrás desde el nodo A, puede describirse con la meta ←*trayectoria*(A,Z). Un sistema PROLOG entonces podría encontrar tal trayectoria.

4.4 TRAYECTORIAS EN GRAFICAS ACTIVAS.

Si imaginamos a un agente recorriendo una gráfica activa, el agente cambia conforme va de un nodo a otro. Sin embargo hasta el momento, sólo hemos considerado trayectorias en gráficas en donde

el agente que las atraviesa no cambia. Afortunadamente los predicados que hemos usado para describir trayectorias no dependen del tipo de objeto que denotan sus argumentos. Una posibilidad para denotar a un agente cambiante es usar términos de la forma $f(A)$, donde f es un símbolo de función que representa a un nodo de la gráfica y A es un término que representa al estado del agente. Ahora los predicados que describen trayectorias van a tener como argumentos a términos de esta forma.

En una gráfica activa es necesario restringir el paso del agente por los arcos. Sólo vamos a permitir al agente recorrer un arco cuando su estado esté en el dominio de la relación asociada a dicho arco. Esta restricción puede incorporarse a la descripción de la gráfica original agregando una condición a cada cláusula incondicional con el símbolo de predicado *arco*. Esta condición representará a la relación asociada a cada arco.

Ejemplo

Consideremos la siguiente gráfica activa

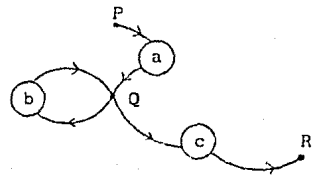


FIG. 4

A los nodos P,Q y R los vamos a denotar con los símbolos de función p,q y r respectivamente. Así, denotaremos al agente con términos de la forma $p(A)$,$q(A)$ y $r(A)$, donde A es el estado del agente. Las relaciones asociadas a los arcos de la gráfica son a,b y c, que denotaremos con los símbolos de predicado *a*, *b* y *c*

respectivamente.

La descripción de esta gráfica es

$$arco(p(X),q(Y)) \leftarrow a(X,Y) \qquad (4)$$

$$arco(q(X),q(Y)) \leftarrow b(X,Y) \qquad (5)$$

$$arco(q(X),r(Y)) \leftarrow c(X,Y) \qquad (6)$$

La cláusula (4), por ejemplo, dice que si el agente está en el nodo P, con estado X, puede pasar al nodo Q con estado Y, si la pareja (X,Y) está en la relación a. La cláusula que define el predicado *final* puede modificarse en forma similar. Si R es el nodo final, entonces la cláusula que nos queda es

$$final(r(X)) \leftarrow$$

Ya tenemos cláusulas que describen trayectorias en gráficas activas y dada una pregunta podemos por medio de resolución encontrar trayectorias en una gráfica activa. Podemos anticipar algunos pasos de resolución antes de hacer la pregunta. Podemos, por ejemplo, resolver las cláusulas (2) y (3), con las que definen los predicados *arco* y *final*. Así, cuando preguntemos, obtendremos trayectorias con menos pasos de resolución. En este caso las cláusulas resolventes son

$$trayectoria(r(X),r(X)) \leftarrow$$

$$trayectoria(p(X),Z) \leftarrow a(X,Y), trayectoria(q(Y),Z)$$

$$trayectoria(q(X),Z) \leftarrow b(X,Y), trayectoria(q(Y),Z)$$

$$trayectoria(q(X),Z) \leftarrow c(X,Y), trayectoria(r(Y),Z)$$

Algo que puede ser conveniente es tener un predicado asociado a las trayectorias hacia atrás desde cada nodo. Para ello definimos los siguientes predicados

$$\forall X \forall Z \; ptray(X,Z) \longleftrightarrow trayectoria(p(X),r(Z))$$

$$\forall X \forall Z \; qtray(X,Z) \longleftrightarrow trayectoria(q(X),r(Z))$$

$$\forall X \forall Z \; rtray(X,Z) \longleftrightarrow trayectoria(r(X),r(Z))$$

La descripción de trayectorias en nuestro ejemplo queda

$$rtray(X,X) \leftarrow$$

$$ptray(X,Z) \leftarrow a(X,Y), qtray(Y,Z)$$

$$qtray(X,Z) \leftarrow b(X,Y), qtray(Y,Z)$$

$$qtray(X,Z) \leftarrow c(X,Y), rtray(Y,Z)$$

En general, podemos describir trayectorias hacia atrás como sigue. Asociamos un símbolo de predicado *ptray* a cada nodo P y otro símbolo de predicado *c* a cada comando c. La *representación hacia atrás* de una gráfica activa tiene una cláusula de la forma

$$ptray(X,Z) \leftarrow c(X,Y), qtray(Y,Z) \qquad (7)$$

por cada arco de P a Q etiquetado con c, además de una cláusula de la forma

$$rtray(X,X) \leftarrow \qquad (8)$$

donde *rtray* es el símbolo de predicado asociado al nodo final. La cláusula (7) dice que si existe una trayectoria del nodo Q al nodo final teniendo como estado inicial Y y como estado final Z, y si (X,Y) está en el comando c, entonces existe una trayectoria del nodo P al nodo final teniendo como estado inicial X y como estado final Z. La cláusula (8) dice que existe una trayectoria consistente en un solo nodo (el nodo final), teniendo como estados final e inicial X, para cualquier X.

Una gráfica activa, además de tener una gráfica dirigida, consta de un conjunto de estados, por lo que necesitamos tener una representación para los estados. Un estado no es más que un vector y en lógica hay varias maneras de representar vectores. Nosotros usaremos una lista, con tantos elementos como componentes tenga el vector.

factoriales. El programa lógico correspondiente es

stray(X,Z)← [f:=1](X,Y),*mtray*(Y,Z)

mtray(X,Z)← [n≠0](X,Y),*ntray*(Y,Z)

mtray(X,Z)← [n=0](X,Y),*htray*(Y,Z)

ntray(X,Z)← [f:=f*n; n:=n-1](X,Y),*mtray*(Y,Z)

htray(X,X)←

Hemos encerrado en corchetes los símbolos de predicado que denotan comandos. (Dichos símbolos consisten en todos los caracteres entre e incluyendo los corchetes, aún los espacios en blanco.) Para propósitos de implementación hemos asociado a estos símbolos de predicado, otros símbolos de predicado de la siguiente manera:

A	hemos asociado
[f:=1]	s_1
[n≠0]	m_1
[n=0]	m_2
[f:=f*n; n:=n-1]	n_1

El listado 2 del apéndice A muestra cómo hicimos para implementar en PROLOG un analizador por cartas que calcula factoriales.

Ahora, consideremos un problema no determinístico.

Problema

Tenemos una cierta cantidad a pagar y para ello contamos con un número determinado de monedas de diferentes denominaciones. Queremos averiguar de qué maneras podemos pagar dicha cantidad, si es que es posible pagarla, con las monedas con que contamos.

Este problema puede tener en principio muchas soluciones. Con propósitos de ejemplificación consideraremos el caso en que tenemos 10 monedas de 5 pesos y 10 monedas de 1 peso. Los estados

de la gráfica activa serán de la forma $\langle n, n_5, n_1, p_5, p_1 \rangle$ donde n es la cantidad que aún nos queda por pagar, n_5 y n_1 son respectivamente el número de monedas de 5 pesos y de 1 peso que nos quedan, y p_5 y p_1 son respectivamente las monedas de a 5 y de a 1 que hemos utilizado hasta ese momento. Por lo tanto, el estado inicial es $\langle n,10,10,0,0 \rangle$, donde n es la cantidad a pagar.

Una gráfica activa que representa este problema es la siguiente:

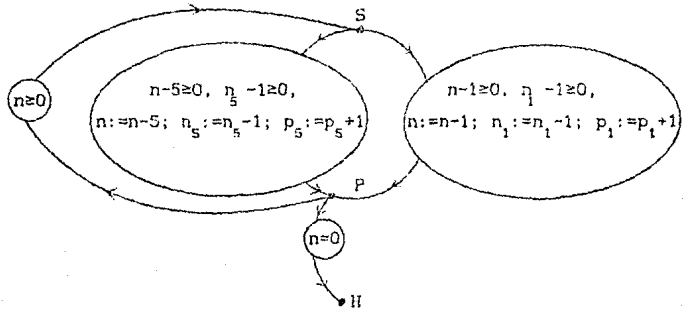


FIG 5

Si otra vez, para propósitos de implementación, asociamos de la siguiente manera

A	asociamos
[n-5≥0, n_5-1≥0, n:=n-5; n_5=n_5-1; p_5=p_5+1]	s_1
[n-1≥0, n_1-1≥0, n:=n-1; n_1=n_1-1; p_1=p_1+1]	s_2
[n=0]	p_1
[n≥0]	p_2

entonces el programa lógico que representa a esta gráfica activa

y por lo tanto a este problema es

$$stray(X,Z) \leftarrow s_1(X,Y), ptray(Y,Z)$$

$$ptray(X,Z) \leftarrow p_1(X,Y), htray(Y,Z)$$

$$ptray(X,Z) \leftarrow p_2(X,Y), stray(Y,Z)$$

$$htray(X,X) \leftarrow$$

El listado 3 del apéndice A muestra la manera en que hemos implementado en PROLOG un analizador por cartas que resuelve este problema, proporcionándonos todas las posibles soluciones.

De esta manera, hemos utilizado a los analizadores por cartas para obtener procedimientos de prueba para representaciones en lógica de gráficas activas.

Hemos observado ciertas similitudes entre la teoría de los lenguajes formales y la programación lógica. A lo largo de la tesis pudimos comprobar que, en efecto, existen puntos en común entre estas dos áreas que pueden enriquecer a ambas.

Empleamos tanto herramientas de programación lógica para resolver problemas dentro de la teoría de lenguajes formales, como herramientas de los lenguajes formales para encontrar soluciones en la programación lógica.

Probamos que cualquier estrategia de búsqueda sobre árboles SLD, a través de los programas cadena, puede servir como analizador sintáctico de lenguajes libres de contexto. Este resultado es ya conocido, puesto que PROLOG es usado con frecuencia como analizadores sintácticos para gramáticas formales. Sin embargo, hasta donde sabemos, su demostración no ha sido publicada aún por otra persona.

Inversamente, mostramos que es posible emplear a los analizadores por cartas para obtener procedimientos de prueba para programas lógicos que representen gráficas activas. Hasta donde sabemos, los analizadores por cartas no han sido usados con anterioridad para ejecutar programas orientados a estados.

De esta manera, estamos resolviendo problemas representables a través de gráficas activas por medio de analizadores por cartas. Dado que las gráficas activas nos permiten representar problemas no deterministicos, el conjunto de problemas resolvibles con analizadores por cartas no es nada despreciable. Además, resolver

problemas de esta manera tiene ventajas sobre la manera de resolverlos por medio de un lenguaje determinístico tipo Pascal, en el que el usuario tiene que preocuparse en gran medida de el control.

El trabajo realizado en esta tesis puede ser continuado, en el siguiente sentido. Sospechamos que resolver problemas (posiblemente no deterministicos) por medio de analizadores por cartas tiene también ventajas sobre los sistemas PROLOG. Esta sospecha se basa en tres hechos: El procedimiento de prueba de PROLOG se puede ver como una generalización de un analizador sintáctico de arriba a abajo con retroceso, los analizadores por cartas son una generalización del algoritmo de Earley, y los analizadores sintácticos de arriba a abajo con retroceso se van con facilidad a ciclos infinitos, mientras que el algoritmo de Earley no se va a estos ciclos. Asi que se puede pensar que los procedimientos de prueba basados en cartas se pierden en ciclos infinitos con menos frecuencia que los sistemas PROLOG.

Este apéndice consta de tres programas, que esencialmente son el mismo. Los tres son analizadores por cartas a lo profundo y están hechos en Arity PROLOG. En lo que difieren es en el uso que le damos al analizador que contienen. Como ya hemos dicho antes, es muy fácil convertirlos en analizadores a lo ancho con sólo intercambiar el orden de dos argumentos en un predicado.

Los arcos de nuestras cartas estarán representados con el predicado

```
arco(INICIO,FIN,NOM1,NOM2).
```

donde NOM1 es el lado izquierdo de la producción con punto y NOM2 es lo que se encuentra después del $\circ$ en dicha producción. Las agendas y las cartas consisten en listas de estos arcos.

Por último, los predicados **findall**, **agrega_arcos**, y **member** son aclarados al final de este apéndice.

El primer programa es un analizador sintáctico que funciona para la gramática G_2 del capitulo 3. Esta gramática es la siguiente:

$S \rightarrow ASB$
$S \rightarrow C$
$A \rightarrow a$
$B \rightarrow b$
$C \rightarrow c$

que hemos decodificado con las siguientes cláusulas

```
inicial(s).
prod(s,[a1,s,b1]).
prod(s,[c1]).
```

```
prod(a1,[a]).
prod(b1,[b]).
prod(c1,[c]).
```

El programa puede funcionar para cualquier gramática normalizada, lo único que hay que modificar son estas cláusulas. La decodificación anterior ejemplifica claramente cómo hacerlo.

LISTADO 1

```
ini_agenda([],V0,[]).                                          (1)
ini_agenda([Simbolo|Simbolos],V0,Agenda) :-
        V1 is V0+1,
        findall(arco(V0,V1,Noterminal,[]),
                prod(Noterminal,[Simbolo]),
                Agenda),
        ini_agenda(Simbolos,V1,Agenda2),
        append(Agenda1,Agenda2,Agenda).
extiende_arcos([],Chart,Chart).                                (2)
extiende_arcos([Arco|Agenda1],Chart1,Chart2) :-
        member(Arco,Chart1), !,
        extiende_arcos(Agenda1,Chart1,Chart2).
extiende_arcos([Arco|Agenda1],Chart1,Chart3) :-
        Chart2=[Arco|Chart1],
        arcos_nuevos(Arco,Chart2,Arcos),
        agrega_arcos(Arcos,Agenda1,Agenda2),                   (3)
        extiende_arcos(Agenda2,Chart2,Chart3).
arcos_nuevos(arco(V1,V2,Simbolo,[]),Chart,Arcos) :-            (4)
        findall(arco(V1,V1,Noterminal,[Simbolo|Simbolos]),
                prod(Noterminal,[Simbolo|Simbolos]),
                Arcos1),
        findall(arco(V0,V2,Noterminal2,Simbolos2),
                member(arco(V0,V1,Noterminal2,[Simbolo|Simbolos2]),
                                                        Chart),
                Arcos2),
        agrega_arcos(Arcos1,Arcos2,Arcos).
```

```
arcos_nuevos(arco(V1,V2,Noterminal,[Simbolo|Simbolos]),Chart,
                                                    Arcos) :-
     findall(arco(V1,V3,Noterminal,Simbolos),
             member(arco(V2,V3,Simbolo,[]),Chart),
             Arcos).
prueba(Cuerda) :-                                          (5)
     ini_agenda(Cuerda,0,Agenda),
     extiende_arcos(Agenda,[],Chart),
     inicial(Simbolo),
     member(arco(0,M,Simbolo,[]),Chart),
     N is M+1,
     not(member(arco(_,N,_,_),Chart)).
inicial(s).                                                (6)
prod(s,[a1,s,b1]).
prod(s,[c1]).
prod(a1,[a]).
prod(b1,[b]).
prod(c1,[c]).
```

El segundo programa es un analizador por cartas que calcula factoriales. A diferencia del programa anterior, los arcos que intervienen en este, tienen como INICIO y FIN listas de la forma [N:E], donde N es un natural y E es el estado en que se encuentran las variables del programa. (En el programa anterior INICIO y FIN eran simplemente naturales.)

Otra diferencia es que este analizador por cartas inicializa la carta creando un solo arco como base y la va extendiendo conforme la va necesitando. Esto es porque en un principio no sabemos cómo será nuestra base completa.

```
ini_agenda(Com,[V0|L1],Agenda) :-                              (7)
     V1 is V0+1,
     findall(arco([V0|L1],[V1|L2],Com,[]),
             comando(Com,L1,L2),
             Agenda).
extiende_arcos([],Chart,Chart).                                (2)
extiende_arcos([Arco|Agenda1],Chart1,Chart2) :-
     member(Arco,Chart1), !,
     extiende_arcos(Agenda1,Chart1,Chart2).
extiende_arcos([Arco|Agenda1],Chart1,Chart3) :-
     Chart2=[Arco|Chart1],
     arcos_nuevos(Arco,Chart2,Arcos),
     agrega_arcos(Arcos,Agenda1,Agenda2),                      (3)
     extiende_arcos(Agenda2,Chart2,Chart3).

arcos_nuevos(arco([V1|L1],[V2|L2],Simbolo,[]),Chart,Arcos) :-  (4)
     findall(arco([V1|L1],[V1|L1],Noterminal,[Simbolo|Simbolos]),
             prod(Noterminal,[Simbolo|Simbolos]),
             Arcos1),
     findall(arco([V0|L0],[V2|L2],Noterminal2,Simbolos2),
             member(arco([V0|L0],[V1|L1],Noterminal2,
                         [Simbolo|Simbolos2]),Chart),
             Arcos2),
     agrega_arcos(Arcos1,Arcos2,Arcos).
arcos_nuevos(arco([V1|L1],[V2|L2],Noterminal,[Simbolo|Simbolos]),Chart,
                                                    Arcos) :-
     findall(arco([V1|L1],[V3|L3],Noterminal,Simbolos),
             member(arco([V2|L2],[V3|L3],Simbolo,[]),Chart),
             Arcos2),
     extiende_base(Arcos2,[V2|L2],Simbolo,Arcos3),
     agrega_arcos(Arcos2,Arcos3,Arcos).
extiende_base([Arco|Arcos],[V1|L1],Noterminal,[]).             (8)
extiende_base([],[V1|L1],Noterminal,Arcos3) :-
     findall(Arco([V1|L1],[V2|L2],Com,[]),
             prodcomando(Noterminal,[Com|Simbolo],[V1|L1],[V2|L2]),
             Arcos3).
```

```
prodcomando(Simbolo,[Com|Simbolo1],[V1|L1],[V2|L2]) :-
        prod(Simbolo,[Com|Simbolo1]),
        V2 is V1+1,
        comando(Com,L1,L2).
prueba(N,R) :-                                          (5)
        ini_agenda(s(X),[0,N,F],Agenda),
        extiende_arcos(Agenda,[],Chart),
        inicial(Simbolo),
        member(arco([0,N,F],[M,1,R],Simbolo,[]),Chart),
        L is M+1,
        not(member(arco(_,[L|P],_,_),Chart)).
inicial(stray).                                         (9)
prod(stray,[s(1),mtray]).
prod(mtray,[m(1),ntray]).
prod(mtray,[m(2),htray]).
prod(ntray,[n(1),mtray]).
prod(htray,[htray]).
comando(htray,X,X).
comando(s(1),[N,F],[N,F1]) :- F1 is 1.
comando(m(1),[N,F],[N,F]) :- N\=1.
comando(m(2),[N,F],[N,F]) :- N is 0.
comando(n(1),[N,F],[N1,F1]) :- F1 is F*N, N1 is N-1.
```

Este programa es muy parecido al anterior, con la exepción que INICIO y FIN son solo estados. Hemos quitado el contador que aparecia en la cabeza de las listas correspondientes a INICIO y FIN en el programa anterior, por razones de eficiencia.

LISTADO 3

```
ini_agenda(Com,L1,Agenda) :-                            (7)
        findall(arco(L1,L2,Com,[]),
                comando(Com,L1,L2),
                Agenda).
extiende_arcos([],Chart,Chart).                         (2)
extiende_arcos([Arco|Agenda1],Chart1,Chart2) :-
        member(Arco,Chart1), !,
```

```
        extiende_arcos(Agenda1,Chart1,Chart2).
extiende_arcos([Arco|Agenda1],Chart1,Chart3) :-
        Chart2=[Arco|Chart1],
        arcos_nuevos(Arco,Chart2,Arcos),
        agrega_arcos(Arcos,Agenda1,Agenda2),                    (3)
        extiende_arcos(Agenda2,Chart2,Chart3).
arcos_nuevos(arco(L1,L2,Simbolo,[]),Chart,Arcos) :-             (4)
        findall(arco(L1,L1,Noterminal,[Simbolo|Simbolos]),
                prod(Noterminal,[Simbolo|Simbolos]),
                Arcos1),
        findall(arco(L0,L2,Noterminal2,Simbolos2),
                member(arco(L0,L1,Noterminal2,
                                [Simbolo|Simbolos2]),Chart),
                Arcos2),
        agrega_arcos(Arcos1,Arcos2,Arcos).
arcos_nuevos(arco(L1,L2,Noterminal,[Simbolo|Simbolos]),
                                        Chart,Arcos) :-
        findall(arco(L1,L3,Noterminal,Simbolos),
                member(arco(L2,L3,Simbolo,[]),Chart),
                Arcos2),
        extiende_base(Arcos2,L2,Simbolo,Arcos3),
        agrega_arcos(Arcos2,Arcos3,Arcos).
extiende_base([Arco|Arcos],L1,Noterminal,[]).                   (8)
extiende_base([],L1,Noterminal,Arcos3) :-
        findall(Arco(L1,L2,Com,[]),
                prodcomando(Noterminal,[Com|Simbolo],L1,L2),
                Arcos3).
prodcomando(Simbolo,[Com|Simbolo1],L1,L2) :-
        prod(Simbolo,[Com|Simbolo1]),
        comando(Com,L1,L2).
prueba(N,R5,R1) :-                                              (5)
        ini_agenda(s(X),[N,10,10,0,0],Agenda),
        extiende_arcos(Agenda,[],Chart),
        inicial(Simbolo),
        member(arco([N,10,10,0,0],[0,_,_,R5,R1],Simbolo,[]),Chart).
inicial(stray).                                                 (10)
prod(stray,[s(1),ptray]).
```

```
prod(ptray,[p(1),htray]).
prod(ptray,[p(2),stray]).
prod(htray,[htray]).
comando(htray,X,X).
comando(s(1),[N,N5,N1,P5,P1],[Ns,Ns5,N1,Ps5,P1]) :-
     N5>=1, N>=5,
     Ns is N-5,
     Ns5 is N5-1,
     Ps5 is P5+1.
comando(s(1),[N,N5,N1,P5,P1],[Ns,N5,Ns1,P5,Ps1]) :-
     N1>=1, N>=1,
     Ns is N-1,
     Ns1 is N1-1,
     Ps1 is P1+1.
comando(p(1),[N|X],[N|X]) :- N is 0.
comando(p(2),[N|X],[N|X]) :- N>0.
```

COMENTARIOS A LOS PROGRAMAS.

(1) ini_agenda(Cuerda,Vertice,Agenda) es usado para crear la base de la carta dada una Cuerda que comienza en Vertice y la guarda en una agenda inicial.

(2) extiende_arcos(Agenda,Chart1,Chart2) sirve para agregar los arcos en Agenda a una carta inicial Chart1 y tomando en cuenta la regla fundamental y la regla de abajo para arriba, genera más arcos, produciendo una carta nuevo Chart2.

(3) agrega_arcos(Arcos,Agenda1,Agenda2) es el predicado responsable de que este analizador sintáctico sea un analizador a lo profundo, puesto que estamos agregando los arcos nuevos del mismo lado del que los estamos retirando. Sustituyendo este predicado por el predicado agrega_arcos(Agenda1,Arcos,Agenda2), nos produce un analizador a lo ancho.

(4) arcos_nuevos(Arco,Chart,Arcos), dados Arco y Chart, es usado para producir un conjunto de arcos nuevos Arcos, tomando en cuenta la regla fundamental y la regla de abajo hacia arriba.

(5) El predicado prueba se emplea para hacer funcionar el analizador y para verificar si el arco solución esta en nuestra carta.

(6) Decodificación de una gramática.

(7) Inicializa la Agenda con una base que abarca sólo dos nodos.

(8) Extiende la base agregándole un arco más.

(9) Decodificación del programa lógico que representa a la gráfica activa que computa factoriales.

(10) Decodificación del programa lógico que representa a la gráfica activa que resuelve el problema de las monedas.

ACLARACION DE ALGUNOS PREDICADOS.

El predicado findall es usado para coleccionar, en su tercer argumento, una lista que contiene para cada solución del segundo argumento la instanciación apropiada del primer argumento.

El predicado agrega_arcos(Arcos1,Arcos2,Arcos3) se usa para unir los arcos en Arcos1 y los arcos en Arcos2, y el resultado lo deja en Arcos3. Las cláusulas que lo definen son:

```
agrega_arcos([],Arcos,Arcos).
agrega_arcos([Arco|Arcos],Arcos1,Arcos3) :-
     agrega_arco(Arco,Arcos1,Arcos2),
     agrega_arcos(Arcos,Arcos2,Arcos3).
agrega_arco(Arco,Arcos,Arcos) :-
     member(Arco,Arcos), !.
agrega_arco(Arco,Arcos,[Arco|Arcos]).
```

El predicado member(X,Y) se emplea para verificar si X es

miembro de Y. Las cláusulas que lo definen son

```
member(X,[X|Y]).
member(X,[Y|Z]) :- member(X,Z).
```

[1] Thomas A.Sudkamp. Languages and Machines. Addison-Wesley, 1988.

[2] J.W. Lloyd. Logic Programming. Springer-Verlag, 1984.

[3] Elliott Mendelson. Introduction to Mathematical Logic. Van Nostrand, 1964.

[4] Gerald Gazdar, Chris Mellish. Natural language processing in Prolog. Addison-Wesley, 1989.

[5] David Rosenblueth y Carlos Velarde. Programas, Autómatas, y Gráficas Activas. Articulo sin terminar.

[6] J.D. Rutledge. On Ianov's program schemata. Journal of the ACM, 1964.

[7] Donald M.Kaplan. Regular expressions and the equivalence of programs. Journal of Computer and System Sciences, 1969.

[8] Andrzej Blikle. Equational languages. Information and control. 1972.

[9] Edward Ashcroft, Zohar Manna, and Amir Pnuell. Decidable properties of monadic functional schemas. Journal of the ACM, 1973.

[10] Joost Engelfriet. Simple Program Schemes and Formal Languages. Springer-Verlag, 1974. Lecture Notes in Computer Science No.20.