UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

97
2EJ

UN ANALIZADOR SINTACTICO VISTO COMO UN SISTEMA DE INFERENCIA

FALLA DE ORIGEN

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERO EN COMPUTACION

P R E S E N T A :

JESUS ROMERO MARTINEZ

DIRECTOR: DR. DAVID A. ROSENBLUETH

MEXICO, D. F. 1995

TESIS CON
FALLA DE ORIGEN

Universidad Nacional
Autónoma de México

Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Un Analizador Sintáctico Visto Como un Sistema de Inferencia

Agradecimientos:

A dios, que me permite vivir.

A mis padres, a quienes todo debo.

Al Dr. David A. Rosenblueth por dirigir este trabajo.

INDICE

INTRODUCCION

CAPITULO 1. Programación Lógica
1.1 El Cálculo Proposicional ...2
1.2 El Cálculo de Predicados de Primer Orden ...8
1.3 El Lenguaje de Programación PROLOG ..12

CAPITULO 2. Transformación de Programas en Modo Fijo a Forma Cadena
2.1 Definición de Programas en Modo Fijo ..14
2.2 Definición de Programas en Forma Cadena ..15
2.3 Algoritmo de Transformación ..15
2.4 Implementación de la Transformación ..18

CAPITULO 3. Analizador Sintáctico por Cartas
3.1 Definición de Analizador Sintáctico ..19
3.2 Analizador Sintáctico por Cartas ..19
3.3 Implementación del Analizador Sintáctico de Abajo hacia Arriba ..29

CAPITULO 4. Sistemas de Inferencia
4.1 Definición ..31
4.2 Implementación de un Analizador Sintáctico por Cartas ..32
4.3 Implementación de un Analizador Sintáctico por Cartas como un Sistema de Inferencia ..33

CAPITULO 5 Conclusiones

BIBLIOGRAFIA

APENDICES

INTRODUCCIÓN.

La razón principal para realizar el presente trabajo es mostrar cómo un analizador sintáctico puede ser visto como un sistema de inferencia. Dicho sistema presenta ventajas importantes ya que, Prolog se va a ciclos infinitos fácilmente y repite trabajo. Un analizador sintáctico por cartas convertido a un sistema de inferencia es capaz de detectan algunos ciclos infinitos y recordar algunos resultados parciales.

El presente trabajo consta de cinco capítulos. En el capítulo 1 describiremos la programación lógica, y de una manera muy especial el lenguaje de programación Prolog, ya que el sistema de inferencia está pensado para utilizar programas escritos en este lenguaje.

Una de las restricciones presentes es, que nuestro sistema de inferencia se basa en utilizar programas en Prolog, y dichos programas deben de presentar una forma especifica, llamada "forma cadena". por tal motivo, en el capítulo 2 hablaremos sobre esta forma, además definiremos el concepto de "programas en modo fijo" y un algoritmo para transformar programas en modo fijo a forma cadena.

En el capitulo 3 definiremos analizadores sintácticos, poniendo mayor énfasis en el analizador sintáctico por cartas en su forma de abajo hacia arriba.

Con ayuda de lo planteado en los tres capítulos anteriores, conjuntaremos en el capitulo 4, el concepto de analizador sintáctico y programas en forma cadena, para mostrar como puede implementarse un sistema de inferencia.

Finalmente en el capitulo 5 trataremos de presentar un panorama sobre este tipo de trabajos, y algunos aspectos relevantes del trabajo realizado.

CAPITULO 1
PROGRAMACIÓN LÓGICA

La finalidad de este capítulo, es presentar de una manera informal el concepto de programación lógica. Para ello, se plantean tres temas, los dos primeros, el cálculo proposicional, y el cálculo de predicados de primer orden, tienen como finalidad, mostrar la base teórica, de la programación lógica. Como texto de apoyo haremos mención al título "Lógica y Algoritmos" [1]. Como tercer tema, lenguaje de programación Prolog, se presenta, como un lenguaje representativo de la programación lógica. Para desarrollar este último tema, se tomó como referencia el título "Programming in Prolog" [2].

Ahora bien, el por qué hablar de programación lógica, la razón más fuerte para ello es que el sistema de inferencia a mostrar en el capítulo 4, está basado en utilizar un analizador sintáctico, como un ejecutador de programas, y este tipo de programas, son programas escritos en Prolog. Al utilizar un analizador sintáctico para ejecutar programas en Prolog se tiene ventajas, ya que Prolog se va a ciclos infinitos fácilmente, y repite trabajo, al utilizar el analizador sintáctico se pueden detectar algunos ciclos infinitos, y se recuerdan algunos resultados parciales, por ello no se repite trabajo.

1.1 EL CALCULO PROPOSICIONAL

NOTACIÓN Y CONCEPTOS BÁSICOS

El *cálculo proposicional* es un método de cálculo con proposiciones o sentencias. Nos ocuparemos aquí de proposiciones que son "ciertas" o "falsas", y con métodos de combinarlas para deducir de ellas nuevas proposiciones. Trataremos de estas proposiciones en abstracto. Es decir, si denotamos por "*p*" a una proposición, no nos preguntaremos cuál es la proposición particular que *p* denota, sino qué es lo que sucede si *p*, en un caso particular, denota una proposición verdadera o una proposición falsa.

Lo primero que realizaremos es definir los conectivos del *cálculo proposicional*, los cuales se ilustran en la siguiente tabla.

cierto	$p \vee q$	$p \subset q$	p	$p \supset q$	q	$p \equiv q$	$p \wedge q$
T	T	T	T	T	T	T	T
T	T	T	T	F	F	F	F
T	T	F	F	T	T	F	F
T	F	T	F	T	F	T	F

$p \mid q$	$p \neq q$	$\neg q$	$p \not\supset q$	$\neg p$	$p \not\subset q$	$p \downarrow q$	falso
F	F	F	F	F	F	F	F
T	T	T	T	F	F	F	F
T	T	F	F	T	T	F	F
T	F	T	F	T	F	T	F

Aqui y en el resto del presente trabajo, T abrevia "cierto" y F abrevia "falso".

FORMULAS BIEN FORMADAS

Definición 1.1.1 Una *fórmula bien formada* (fbf) del cálculo proposicional se define mediante las siguientes reglas:

1. Una variable proposicional aislada es una fbf.
2. Si A es una fbf, entonces $(\sim A)$ es una fbf.
3. Si A y B son fbf, entonces las siguientes son fbf.
   $(A \wedge B)$, $(A \vee B)$, $(A \supset B)$, $(A \equiv B)$, $(A \subset B)$,
   $(A \not\supset B)$, $(A \not\subset B)$, $(A \not\equiv B)$, $(A | B)$, y $(A \downarrow B)$.

4. Una cadena de símbolos es una fbf, si y sólo si el serlo se sigue de un número finito de aplicaciones de las reglas 1, 2 y 3.

Varias convenciones se han introducido. La primera de todas, los paréntesis en (p) aparecen redundantes, convengamos en que, si A es una variable proposicional que aparece aislada en lugar de "(A)" escribiremos simplemente "A".

Una segunda convención es el de la jerarquía u orden de precedencia. El orden de precedencia que usaremos es $\sim$, $\wedge$, $\vee$, $\subset$, $\equiv$, (podríamos incluir a los otros conectivos pero se utilizan muy raramente). Por otra parte convendremos que para dos ocurrencias de un mismo conectivo, el de más a la izquierda toma precedencia sobre el de más a la derecha.

EQUIVALENCIA LÓGICA Y CONSECUENCIA LÓGICA

Definición 1.1.2 Dos proposiciones A y B son lógicamente equivalentes (*A eq B*), si y sólo si tienen la misma tabla de verdad. Esto no implica que A y B "signifiquen" la misma cosa.

Definición 1.1.3 B es una consecuencia lógica de A $(A \models B)$, si para toda asignación de valores de verdad a las variables de A (y B), tal que A tiene el valor T, entonces B tiene también el valor de T.

FORMAS NORMALES

De las varias formas *estandar* o *canónicas* o *normales* que han sido desarrolladas, consideraremos solamente cuatro formas estrechamente relacionadas entre sí, que se pueden desarrollar fácilmente.

Definición 1.1.4 Una fórmula es *una conjunción elemental*, si es de la forma $A_1 \wedge A_2 \wedge A_3 \wedge \ldots \wedge A_n$, en donde

1. Toda A_i es una variable proposicional o la negación de una variable proposicional;
2. Ninguna variable proposicional aparece en más de una A_i;
3. Aquellas variables proposicionales que aparecen están en orden alfabético, es decir si $i < j$, entonces, la variable proposicional que aparece en A_i precede a la que está en A_j en el alfabeto.

Una fórmula está en forma *normal disyuntiva*(FND), si es de la forma $B_1 \vee B_2 \vee B_3 \vee \ldots \vee B_n$ donde

1. cada B_i es una conjunción elemental;
2. no hay dos B_i que sean iguales;
3. Si $B_i = A_{i1} \wedge A_{i2} \wedge \ldots \wedge A_{in_i}$ y

   $B_j = A_{j1} \wedge A_{j2} \wedge \ldots \wedge A_{jn_j}$ y

   $A_{i1} = A_{j1}$, $A_{i2} = A_{j2}$, ..., $A_{i,k-1} = A_{j,k-1}$ y

   $A_{ik} \neq A_{jk}$, entonces

$i < j$, si y sólo si se verifica alguna de las siguientes alternativas:

(a) A_{ik} no existe (es decir $k-1 = n_i$).

(b) $A_{ik} = p$ y $A_{jk} = \neg p$ para alguna variable proposicional p.

(c) la variable proposicional que aparece en A_{ik} precede a la que aparece en A_{jk}.

Una fbf está en forma *normal disyuntiva plena* (FNDP), si está en FND y todas las variables proposicionales que aparecen en una cualquiera de las conjunciones elementales aparece en todas las conjunciones elementales.

Los términos *disyunción elemental*, *forma normal conjuntiva* (FNC), y forma *normal conjuntiva plena*(FNCP) se definen análogamente, con la disyunción y la conjunción intercambiadas.

Esta definición especifica completamente las formas normales, hasta en el orden en que los distintos términos aparecen. Por lo tanto, una vez que dos fórmulas se han escrito en una de las formas normales plenas, una simple comparación directa de las dos formas establecerá si las fórmulas representan o no a la misma función.

LA NOTACIÓN "POLACA" Y EL ARBOL DE UNA FORMULA

En 1951, el logico polaco Jan Lukasiewicz sugirió que podía usarse una notación prefijada para todas las operaciones, y señalaba que un uso consistente de tal notación eliminaba toda necesidad de puntuación.

Denotaremos a las proposiciones por letras minúsculas: p, q, r, s; y a los operadores y a los conectivos por letras mayúsculas: N, C, A, K, E. Usaremos letras griegas minúsculas para representas a las fórmulas bien formadas, que debemos definir de nuevo, puesto que estamos usando un nuevo simbolismo.

Definición 1.1.5 Una *fórmula bien formada* (del cálculo proposicional y según la notación polaca) es una fórmula que puede obtenerse mediante un número finito de aplicaciones de las reglas siguientes :

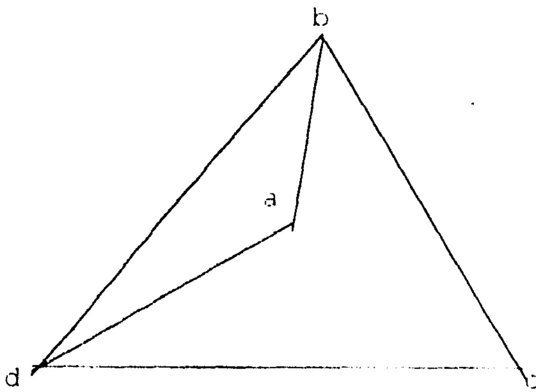
1. Una variable proposicional aislada es una fórmula bien formada (fbf).

2. Si α es una fbf, entonces $N\alpha$ es una fbf.

3. Si α y β son fbf, entonces $C\alpha\beta$, $A\alpha\beta$, $K\alpha\beta$ y $E\alpha\beta$ son fbf.

Definición 1.1.6 Una *gráfica* consiste en un conjunto P de puntos (llamados "vértices") y un conjunto L de segmentos (llamados "lados"), tales que cada segmento de L tiene asociados con él a exactamente dos puntos de P, que son sus vértices. Si a y b son vértices de una gráfica, entonces una cadena de a a b es un conjunto

$C=\{p_0,p_1,p_2,\ldots,p_n\}$ (P de vértices tales que $a=p_0$, $b=p_n$, y para cada $i=1,\ldots,n$ hay un lado $e_i \in L$ cuyos vértices son p_{i-1} y p_i. Una gráfica es *conexa* si hay una cadena de uno cualquiera de sus vértices a cualquiera otro. Un *ciclo* es una cadena $\{p_0,p_1,\ldots,p_n\}$ $n>=1$, tal que $p_0,p_1,\ldots,p_{n-1}$ son distintos y $p_n=p_0$. Una gráfica conexa que no tiene ciclos se llama un *árbol*.



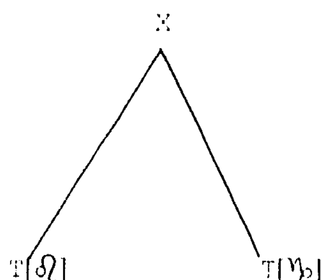
Ejemplo de una gráfica

Definición 1.1.7 Si α es una fbf (en notación polaca), entonces el arbol de α es la grafica definida por el siguiente proceso.

1. Si α es una sola variable proposicional, entonces el árbol de α es la gráfica con vértice en α y sin lado alguno.

2. Si α es de la forma Nβ, entonces el árbol de α es la gráfica cuyos vértices son {N}U{vértices de β }, y cuyos lados son los lados de β junto con un lado que une N y el conectivo principal de β (o el vértice de β, si β no tiene más de un vértice).

3. Si α es de la forma X$\beta\gamma$, donde X es uno de los conectivos C, A, K y E, entonces el árbol de α es la gráfica cuyos vértices son {X}U{vértices de β}U{vértices de γ}, y cuyos lados son los lados de β y de γ junto con un lado que une X y el conectivo principal de β, y un lado más que une X y el principal conectivo de γ.



El árbol de X$\delta\gamma$

Tres comentarios son oportunos. Primero, los arboles que estamos manejando son, en realidad, arboles rotulados. Cuando se rotulan los vértices de un arbol, se convierte en el árbol de una fórmula particular. Segundo, como se tiene el hecho de que q⊃p(Cpq) no es equivalente a q⊃p(Cqp), el orden de izquierda a derecha de los vértices es importante para nuestro propósito. Tercero, el árbol de una fórmula podria haberse desarrollado perfectamente bien, partiendo de la notación estándar de intercalaciones.

CONJUNTOS MINIMALES Y CONECTIVOS

Como cualquier función booleana puede representarse por una expresión en forma disyuntiva normal, es claro que, cuando mucho, los conectivos que necesitamos son la conjunción, la disyunción y la negación. Por otra parte, las leyes de DeMorgan nos permiten expresar la disyunción en términos de la disyunción y la negación. Luego lo que realmente necesitamos tener a nuestra disposición es la negación, y o la conjunción o la disyunción. Decimos que la disyunción y la negación forman un *conjunto mínimo de conectivos* para el cálculo proposicional.

Análogamente, la conjunción, y la negación forman también un conjunto mínimo de conectivos

Es posible seleccionar también otros conjuntos mínimos de conectivos. en realidad el trazo de Sheffer constituye, él solo, un conjunto mínimo de conectivos, lo mismo que la flecha de Pierce.

1.2 EL CALCULO DE PREDICADOS DE PRIMER ORDEN

En la sección anterior presentamos un sistema de lógica que puede usarse al tratar con proposiciones, es decir con sentencias declarativas que tienen un valor de verdad fijo "falso" o "verdadero". Deseamos ahora examinar estas proposiciones con mas detalle.

En particular, queremos poder tratar con individuos, relaciones entre individuos, y propiedades de conjuntos de individuos. por ejemplo, podemos desear describir individuos llamados "Enrique" y "Juan", y hablar acerca del hecho de que Enrique es el padre de Juan. Tenemos pues ambas funciones sobre el conjunto de individuos (por ejemplo, definiendo al padre de un individuo determinado), y predicados que describen propiedades y relaciones sobre conjuntos de individuos (por ejemplo, P puede denotar la relación binaria "es un padre de"), las funciones definen nuevos individuos en términos de los previamente conocidos, y los predicados por su valor de verdad, describen un conjunto de individuos que tiene una cierta relación o propiedad.

Al mismo tiempo, queremos que nuestra lógica sea suficientemente fuerte como para tratar con formas proposicionales estructuras que aparecen como sentencias declarativas; pero que no tienen valor definido de verdad a causa de la presencia de variables individuales. Por ejemplo, "2+3=4" es una proposición que tiene valor de verdad, pero "X+3=4" es una forma que no es ni verdadera ni falsa, ya que no sabemos qué número está representado por X. El sistema lógico que describiremos se conoce como el cálculo de predicados de primer orden.

DEFINICIONES Y PROPIEDADES BÁSICAS

La lógica que desarrollaremos será una extensión de nuestro sistema previo, y por ello, hará uso de los símbolos anteriores para las operaciones lógicas y las proposiciones. Necesitaremos, además, los siguientes símbolos.

Para constantes individuales (nombre de individuos): $a, b, c, a_1, \ldots$

Para variables individuales (pronombre): $x, y, z, x_1, y_1, \ldots$

Para *símbolos de función* (funciones): $f^i_j, g^i_j, h^i_j, \ldots$, donde i y j denotan enteros positivos

Para *símbolos de predicado* (predicados): $F^i_j, G^i_j, H^i_j, \ldots$, donde i y j denotan enteros positivos

Para cuantificadores $(\forall\alpha)$, $(\exists\alpha)$, donde α puede ser una variable individual cualquiera.

Con la introducción de estos nuevos símbolos, nuestro conceptos previos necesitan extenderse y redefinirse.

Definición 1.2.1 Un *término* está definido como sigue:

1. Las constantes individuales y las variables individuales son términos.
2. Si f^n_i es un símbolo de función y $t_1, t_2, \dots, t_n$ son términos, entonces $f^n_i(t_1, t_2, \dots, t_n)$ es un término.
3. Los únicos términos son los formados por (1) y (2).

Definición 1.2.2 Una *sucesión* es una fórmula atómica, si es o

1. una variable proposicional aislada o
2. una sucesión de la forma $F^n_i(t_1, t_2, \dots, t_n)$, donde F^n_j es un símbolo de predicado y $t_1, t_2, \dots, t_n$ son términos.

Definición 1.2.3 Una *fórmula bien formada* (fbf) se define como sigue:

1. Una formula atómica es una fbf.
2. Si A es una fbf y α es una variable individual, entonces $(\forall\alpha)A$ y $(\exists\alpha)A$ son fbf.
3. Si A y B son fbf entonces $\sim(A)$, $(A)\supset(B)$, $(A)\wedge(B)$, $(A)\vee(B)$, y $(A)\equiv(B)$ son fbf.
4. Las únicas fbf son obtenidas por un número finito de aplicaciones de (1), (2) y (3).

Haremos uso de los mismos convenios sobre paréntesis, jerarquías y puntos que se adoptaron en el tema anterior.

Los dos cuantificadores, $(\forall)$ y $(\exists)$, se llaman, respectivamente, el cuantificador *universal* y el cuantificador *existencial*, y son los equivalentes formales de las palabras "todo" y "algún". Así pues, si P (es decir P^1_i) es un símbolo de predicado y x es una variable individual, $(\forall x)P(x)$ puede leerse como "para toda x del dominio el individuo x tiene la propiedad P", y $(\exists x)P(x)$ puede leerse: "existe algún individuo x en el dominio que tiene la propiedad P". Estos dos cuantificadores no son independientes, sino que, en realidad, están relacionados por la equivalencia lógica $(\forall x)P(X)$ eq $\sim(\exists x)\sim P(x)$

Como mencionamos al principio, una forma proposicional no forzosamente tiene un valor de verdad bien definido por la posible presencia de variables no cuantificadas en ella.

VARIABLES LIBRES Y LIGADAS

Debemos distinguir entre variables que son cuantificadas y las que no lo son, y saber con precisión cuál cuantificador controla, en una expresión, a una o varias variables. Hablamos de la expresión a la que el cuantificador se aplica como del dominio del cuantificador, y decimos que una ocurrencia de una variable individual x está ligada si aparece como (x) o $(\exists x)$, o dentro del dominio de un cuantificador $(\forall x)$ o $(\exists x)$. Cualquier otro tipo de ocurrencia de una variable decimos que ocurre libre.

VALIDEZ Y SATISFACTIBILIDAD

Definición 1.2.4 Dada una fórmula bien formada F del cálculo de predicados de primer orden, una interpretación de F consiste en un dominio no vacio de D y una asignación a cada simbolo de predicado n-ario de un predicado n-ario sobre D, a cada simbolo de función n-ario de una función n-aria sobre D, y a cada constante individual de un elemento fijo de D.

Definición 1.2.5 Una fbf del cálculo de predicados de primer orden es "satisfactible" en un dominio D si existe una interpretación con dominio D, y asignaciones de elementos de D a las ocurrencias libres de variables individuales en la fórmula tal que la proposición resultante es cierta.

Una fbf es válida en un dominio D si para toda interpretación con dominio D y toda asignación de elementos de D a las ocurrencias libres de las variables individuales en la fórmula, la proposición resultante es

cierta. Una fbf es satisfactible si es satisfactible en algún dominio; es válida si es válida en todos los dominios.

Hablaremos también de fórmulas que son válidas o satisfactibles en una interpretación particular.

Se establecen fácilmente ciertas relaciones entre satisfactibilidad, validez y el dominio. Si una fbf F es válida en un dominio D, entonces es satisfactible en D. Análogamente, si F es válida, entonces es satisfactible. Si F es válida en D, entonces es válida en cualquier subconjunto no vacío de D; si F es satisfactible en D, entonces es satisfactible en cualquier dominio que contenga a D como subconjunto. F es válida, si y sólo si ~F es no satisfactible; F es satisfactible, si y sólo si ~F es no válida. Esto indica que la dualidad que se verifica en el álgebra de booleana y el cálculo proposicional se extiende al cálculo de predicados de primer orden.

1.3 EL LENGUAJE DE PROGRAMACIÓN "PROLOG"

La programación lógica es un paradigma de la programación construida por modelos abstractos de lógica de primer orden.

Prolog es un lenguaje de programación representativo de esta clase de lenguajes con el suficiente poder funcional para facilitar el desarrollo de avanzadas aplicaciones; fue inventado por Alain Colmerauer y sus colaboradores alrededor de 1970. Fue la primera tentativa de lenguaje de programación práctico que diera a los programadores la posibilidad de especificar tareas con una base lógica. Esta motivación explica el nombre del lenguaje de programación, como "Prolog" por programation on logique (en francés).

Para relacionar Prolog y lógica, tenemos que establecer lo que entendemos por lógica. La lógica fue originalmente vista como un camino de representar la forma de argumentos, de tal forma que fuera posible verificar mediante un camino formal lo que es o no es válido. Así podemos usar la lógica para expresar proposiciones, la relación entre proposiciones y cómo uno

puede validar o inferir algunas proposiciones a partir de otras.

La forma lógica particular que utiliza Prolog, es llamada el cálculo de predicados (desarrollado en tema anterior).

CAPÍTULO 2
TRANSFORMACIÓN DE PROGRAMAS EN modo fijo A forma cadena

En este capítulo definiremos dos tipos de programas lógicos, programas en *modo fijo* y en *forma cadena*. Es importante definirlos, ya que el sistema de inferencia a presentar en el capítulo 4, está basado en utilizar programas en forma cadena. Debido a que no es fácil escribir directamente programas en forma cadena, se presentara un algoritmo para transforma programas lógicos en modo fijo a forma cadena, siendo más práctico escribir programas en modo fijo.

Asi la idea básica es escribir programas en modo fijo, transformarlos a forma cadena y ejecutarlos por medio de un analizador sintáctico.

Para desarrollar el presente capítulo, se tomó como referencia el articulo "LR parsers as inference systems for fixed-mode logic programs" [3].

2.1 PROGRAMAS EN MODO FIJO

Consideraremos que un programa lógico es de *modo fijo* cuando cada predicado es usado para efectuar una operación. Esta clase de programas es importante ya que cubre una gran cantidad de programas prácticos escritos en Prolog. Aunque muchas veces un mismo programa en Prolog puede ser usado para efectuar más de una operación, frecuentemente aparecen obstáculos. Por ejemplo, la aparición de ramas infinitas al lado izquierdo de una respuesta puede impedir a Prolog que encuentre una respuesta. Aunque la ejecución terminara, la computación de operación seria intolerablemente ineficiente. Otros obstáculos en la invertibilidad de los programas lógicos son el uso de predicados incorporados y la carencia del chequeo de ocurrencia. Como consecuencia, un programa largo en Prolog tipicamente tiene el efecto de una operación simple. Así podemos concluir que las

limitántes en los programas en modo fijo no son restricciones de gran fuerza cuando las comparamos con Prolog.

2.2 PROGRAMAS EN FORMA CADENA

Definimos un programa *cadena* como un programa lógico que consiste solo de cláusulas que tienen la forma:

$$a_0(X_0, X_n) \longleftarrow a_1(X_0, X_1),\ a_2(X_1, X_2), \ldots,\ a_n(X_{n-1}, X_n)$$

y

$$a(t, t') \longleftarrow$$

donde : $var(t') \subseteq var(t)$

Aquí y en todas partes del documento, *var(t)* denota el conjunto de variables que ocurren en t.

2.3 ALGORITMO DE TRANSFORMACIÓN

Presentaremos ahora la transformación tomando un programa de modo fijo y produciendo un programa cadena. En un programa de modo fijo , cada predicado es usado como una sola operación, ya que los argumentos del predicado son predefinidos dependiendo de su uso como de "salida" o de "entrada". Esta previa selección fija a un predicado a realizar una sola acción. Si todos los argumentos de entrada por cada operación son agrupados en un argumento y todos los de salida son agrupados en otro, podemos asumir que no perdemos generalidad y todos los predicados son binarios. Donde el primer argumento del predicado lo llamaremos la entrada, y al segundo la salida.

Un programa dirigido es un programa lógico que consiste solo en cláusulas dirigidas. Una *cláusula dirigida* es una cláusula de la forma:

$$p_0(t_0, t'_n) \longleftarrow p_1(t'_0, t_1), p_2(t'_1, t_2), \ldots, p_n(t'_{n-1}, t_n)$$
$(n>=0)$

donde

1. $var(t_i) \cap var(t_j) = \phi$, para $i,j = 0,\ldots,n$ y $i \neq j$;
2. $var(t'_i) \subseteq var(t_0) \cup \ldots \cup var(t_i)$ para $i = 0,\ldots n$;
3. Cada variable que ocurre en t'_i ocurre solo una vez en t'_i para $i = 0,\ldots,n$

La condición (1) causa que el término construido cuando una submeta tiene éxito tenga solo efecto en la entrada de otra submeta. La condición (2) causa que el argumento de entrada de todas las submetas seleccionadas esté instanciado, si el argumento de entrada de la primera submeta está instanciada y son seleccionadas de izquierda a derecha. La condición (3) es incluida solo por razones técnicas; es una restricción mínima. Intuitivamente, la transformación de programas dirigidos a cadena agrega una lista a la entrada y a la salida de cada predicado. Esta lista almacena los términos usados por subsecuentes átomos en las cláusulas que se encuentran de izquierda a derecha de la submeta seleccionada.

Considerando una cláusula dirigida:

$$p_0(t_0, t'_n) \longleftarrow p_1(t'_0, t_1), p_2(t'_1, t_2), \ldots, p_n(t'_{n-1}, t_n)$$
$(n>=0)$

se define:

$$\Pi_i = (var(t_0) \cup \ldots \cup var(t_{i-1})) \cap (var(t'_i) \cup \ldots \cup var(t'_n))$$

para $i=1,\ldots,n$ y $\Pi_1, \Pi_{n+1} = \phi$, esto es, cada Π_i, es el conjunto de variables que reciben una substitución en la salida de las submetas del lado izquierdo de la submeta

con símbolo de predicado p_i (o la entrada de la cabeza), y que ocurren en las entradas de las submetas del lado derecho de la submeta con el símbolo de predicado p_i (o la salida de la cabeza).

También se define Σ como una lista de la forma $[X_{1,i}, ..X_{di,i} | St]$, donde $\{X_{1,i}, ..., X_{di,i}\} = \Pi_i$ si $\Pi_i \neq \phi$, y Σ_i es St si $\Pi_i = \phi$, para $i=0,...,n+1$.

La cláusula producida por la transformación es una cláusula dirigida en forma cadena de la forma:

$$p'_0(X_0, X_{2n+1}) \longleftarrow k_0(X_0, X_1),\ p'_1(X_1, X_2),$$
$$k_1(X_2, X_3),\ p'_2(X_4, X_5), \ldots,$$
$$k_{n-1}(X_{2n-2}, X_{2n-1}),\ p'_n(X_{2n-1}, X_{2n}),\ k_n(X_{2n}, X_{2n+1})$$

y la definición de los predicados p'_i y k_i es:

$$p'_i(\ <St|X>, <St|Y>\ ) \longleftrightarrow p_i(X, Y) \qquad i=0,...,n$$
$$k_0(<\Sigma_0 | t_0>, <\Sigma_1, t'_0>\ ) \longleftarrow$$
$$k_1(<\Sigma_1 | t_1>, <\Sigma_2, t'_1>\ ) \longleftarrow$$

.

.

.

$$k_n(<\Sigma | t_n>, <\Sigma_{n+1} | t'_n>) \longleftarrow$$

Ejemplo: Ilustraremos la transformación con el siguiente programa. Usualmente el predicado APPEND es usado para unir listas, pero en el ejemplo se utilizar para separar listas. El predicado s(<Z>,<X,Y>) es cierto si y solo si Z es una lista que puede ser descompuesta en la lista X seguida de la lista Y

$$s(<Z>, <[\,], Z>) \longleftarrow$$
$$s(<W|Z]>, <[W|X], Y>) \longleftarrow s(<Z>, <X, Y>)$$

El programa en forma cadena para s es:

$$s'(U_0,U_1) \longleftarrow k_0(U_0,U_1)$$

$$k_0(<St,Z>,<St,[\,],Z>) \longleftarrow$$

$$s'(U_0,U_3) \longleftarrow b_1(U_0,U_1),\ s'(U_1,U_2),\ b_2(U_2,U_3)$$

$$b_1(<St,[W|Z]>,<[W|St],Z>) \longleftarrow$$

$$b_2(<[W|St],X,Y>,<St,[W|X],Y>) \longleftarrow$$

2.4 IMPLEMENTACIÓN

La implementación de la transformación se realizó en el lenguaje Prolog, y consiste en un programa que teniendo como entrada el nombre de un archivo, el cual contiene un programa en modo fijo, genera un segundo archivo el cual contiene el nuevo programa en forma cadena. El código de dicho programa se encuentra en el apéndice A.

CAPITULO 3
ANALIZADOR SINTÁCTICO POR CARTAS

En el presente capitulo, hablaremos de un analizador sintáctico para gramáticas libres de contexto, llamado *analizador por cartas*, ya que será este la base para nuestro sistema de inferencia a desarrollar en el siguiente capitulo.

Para desarrollar el primer tema: definición de analizador sintáctico, se tomó como apoyo el texto "Compiladores, Principios, Técnicas y Herramientas" [4]. Los dos siguientes temas: analizador sintáctico por cartas, e implementación de un analizador sintáctico por cartas de abajo hacia arriba, fueron tomados del texto "Natural Lenguage Processing in Prolog" [5].

3.1 Definición de analizador sintáctico

Realizar un análisis sintáctico implica agrupar los componentes léxicos de un programa fuente en frases gramaticales que el compilador utiliza para sintetizar la salida.

Para nuestro propósito definimos al *analizador sintáctico* como la herramienta que nos sirve para determinar si una cuerda pertenece o no a un lenguaje, definido por una gramática libre de contexto.

3.2 Analizador sintáctico por cartas

En este tema se presenta como un analizador sintáctico puede almacenar eficientemente resultados intermedios, haciendo frente a la redundancia existente en el espacio de búsqueda del mismo.

Primero introducimos la noción *de tabla de subcuerdas bien formadas* (TSBF). Una TSBF es un mecanismo que le

llevar un registro de las estructuras que ya ha encontrado.

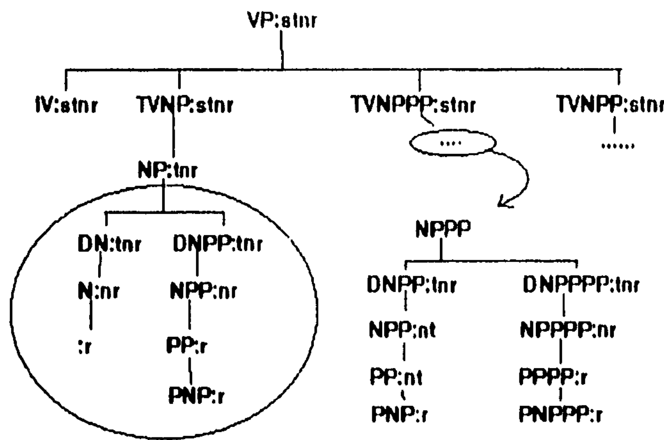
Tabla de cuerdas bien formadas (TSBF)

Para tratar este tema manejaremos un fragmento de la estructura gramatical del inglés.

S ⟶ NP VP
VP ⟶ IV
VP ⟶ IVNP
VP ⟶ PP
VP ⟶ TV NP
VP ⟶ TV NP PP
VP ⟶ TV NP VP
NP ⟶ Det N
NP ⟶ det N PP
PP ⟶ P NP
N ⟶ cat
N ⟶ nurse
Det ⟶ cat
Det ⟶ her
NP ⟶ her
NP ⟶ they
NP ⟶ nurse
N ⟶ travel
N ⟶ report
TV ⟶ hear
TV ⟶ see
P ⟶ on

Consideramos ahora el problema que se presenta en el analizador sintáctico de arriba hacia abajo, al analizar la cuerda "they saw the nurse report".

El árbol que generaría es el siguiente:



Observamos que es redundante el espacio de búsqueda del analizador sintáctico, esto es, que esencialmente se repite la misma situación varias veces, así si nuestro analizador sintáctico explora de izquierda a derecha en el primer subárbol genera la cuerda "the nurse", pero este subárbol falla y tiene que buscar en el segundo, repitiendo el trabajo que ya realizó en el primer subárbol.

Para solucionar este problema la idea es que el analizador sintáctico recuerde las estructuras que de alguna forma ya encontró, en este caso debería recordar la cuerda "the nurse" como una frase nominal (NP).

Una solución al problema es utilizar TSBF la cual consiste, primero en enumerar el inicio y fin de una cuerda con los números de 0 y n; y segundo en crear arcos entre las palabras que serán numeradas del 1 al n-1 de izquierda a derecha. Ahora para cada par de puntos i,j tal que 0 =< i =< j =< n se le asocia una categoría, que represente la estructura que forman las palabras que se encuentran entre los vértices i y j.

Así podemos representar una TSBF como un conjunto de arcos donde cada arco es una estructura con los siguientes atributos:

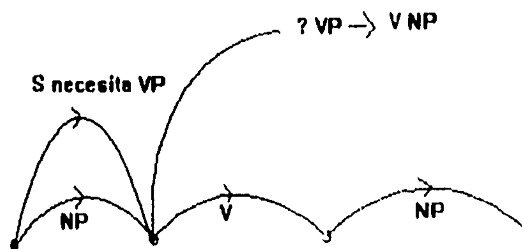
<inicio> : entero

<final>   : entero
<etiqueta>: alguna categoría

Una TSBF puede ser utilizada en cualquier analizador sintáctico ya que su estructura es completamente natural.

Las cartas activas

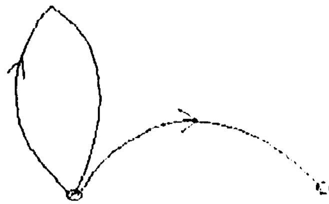
La TSBF es una buena forma de representar las estructuras, pero no podemos representar en la TSBF una hipótesis, por ejemplo para la siguiente figura:



- Se establece que la cuerda consiste en la secuencia NP V NP.
- Existe la posibilidad de que S consista de la secuencia NP VP.
- Estamos en el punto donde se debe tratar de definir si la NP ya establecida es la NP de la secuencia NP VP y
- Necesitamos establecer que la secuencia V NP representa la VP que esperamos.

Podemos ver que la TSBF puede representar un análisis parcial. Debido a que no podemos representar hipótesis en nuestra TSBF, ahora realizamos dos pequeños cambios en la estructura de datos. Primero requerimos que nuestra

gráfica dirigida pueda ser ciclica, podemos permitir arcos simples que van del nodo inicio a si mismo, ejemplo:

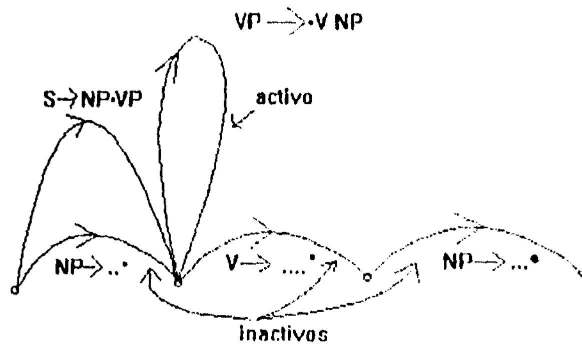


El segundo cambio que necesitamos hacer es elaborar la etiqueta para los arcos, basándonos en las reglas de la gramática. Si S ⟶ NP VP es una regla de la gramática, entonces el siguiente objeto (reglas de punto) son las etiquetas para los arcos.

S ⟶ .NP VP
S ⟶ NP.VP
S ⟶ NP VP.

Informalmente el punto en estas etiquetas indica que regla es aplicable y la hipótesis a extender.

Una TSBF que a sido modificada e incluye hipótesis es conocida como una carta activa, la cual abreviamos como *carta*. Un nodo en una carta es referido como un *vértice*, los arcos que representan una hipótesis por confirmar se conoce como *arco activo* y aquellos que ya conocemos completamente como *arcos inactivos*. Ejemplo:



Así lo que nosotros esperamos es obtener cartas que contengan arcos inactivos. Podemos representar a una carta como un conjunto de elementos que tienen la siguiente estructura:

<inicio> : entero
<final> : entero
<etiqueta> : categoria
<encontrado> : secuencia de categorias
<por_encontrar>: secuencia de categorias

Donde:

<etiqueta> es la parte del lado izquierdo (LHS) de la regla de punto.

<encontrado> es la secuencia de categorias del lado izquierdo del punto en la parte derecha de la regla (RHS).

<por_encontrar> es la secuencia de categorias del lado derecho del punto en la parte derecha de la regla (RHS).

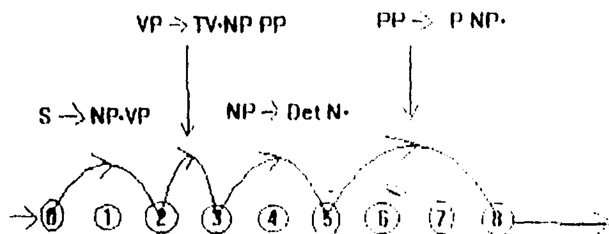
Ocasionalmente se pueden usar tuplas de la siguiente forma: <0,2,S ⟶ NP.VP>

La regla fundamental del analizador sintáctico por cartas

Esta regla solo es aplicable cuando ya se tienen arcos en la carta. Para una mayor claridad sobre esta regla supongamos que estamos a la mitad del análisis sintáctico y tenemos la siguiente carta:

{ <0,2,S ⟶ NP.VP>
<2,3,VP ⟶ TV.PN PP>
<3,5,NP ⟶ Det N.>
<5,8,PP ⟶ P NP.> }

Gráficamente tenemos:



Observamos que existen dos arcos activos, los cuales representan las hipótesis. El primero (<0,2,S ⟶ NP.VP>) representa la hipótesis acerca de la oración (S) en la cual se a encontrado una frase nominal (NP) y esta en espera de encontrar una frase verbal (VP) para generar un arco pasivo. Para el segundo (<2,3,VP ⟶ TV.NP PP>) representa la hipótesis hacerla de una frase verbal (VP) para la cual ya se a alcanzado el verbo transitivo (TV) y esta en espera de una frase nominal, seguido de una frase poposicional (PP).

Considerando el primer arco activo, para poder satisfacer nuestra hipótesis necesitamos encontrar en la carta un arco inactivo de una frase verbal que inicie en el vértice 2. Mientras no estemos seguros de que existe este arco no debemos especular con este tipo de arcos activos. Pasando al segundo arco activo tenemos una frase verbal y la hipótesis es encontrar una frase nominal que inicie en el vértice 3, podemos observar que tenemos un arco que representa una frase nominal que es un arco inactivo y podemos representar un avance agregando otro arco activo a la carta llamado <2,5,VP ⟶ TV NP.PP>, este nuevo arco contiene la hipótesis de encontrar una frase proposicional que inicie en el vértice 5, pero tenemos un arco inactivo que cumple esta hipótesis (<5,8,PP⟶P VP.>) por ello podemos agregar un arco inactivo a la carta que representa la frase verbal encontrada, dicho arco estara etiquetado con: <2,8,VP ⟶ NP PP.> . Asi obtenemos la carta:

{ <0,2,S ⟶ NP.VP>
<2,3,VP ⟶ TV.NP PP>
<2,5,VP ⟶ TV NP.PP>
<2,8,VP ⟶ TV NP PP.>
<3,5,NP ⟶ Det N.>
<5,8,PP ⟶ P NP.> }

Si ahora regresamos al primer arco activo podemos ver que su hipótesis ya puede ser confirmada, ya que existe un arco inactivo que inicia en el vertice 2. Asi que ya podemos agregar el arco inactivo <0,8,S ⟶ NP VP.>,

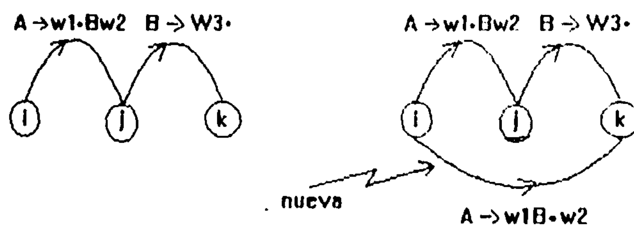
llegando en este momento al final del analizador sintáctico.

El proceso descrito representa la esencia del analizador sintáctico por cartas, y se define mediante la siguiente regla:

Regla fundamental

Si la carta contiene arcos <i,j,A ⟶ W1.B W2> y <j,k,B ⟶ W3.> donde A y B son categorías y W1,W2 y W3 son (posiblemente) una secuencia de categorías ó palabras, entonces debemos agregar el arco <i,k,A ⟶ W1 B.W2> a la carta.

Ejemplo:



Esta regla no dice nada acerca del nuevo arco, si es activo o inactivo, pero esto no es necesario ya que se determina completamente al conocer a W2 ya que esta secuencia puede ser vacía.

Ahora que conocemos la regla fundamental podemos mencionar tres ingredientes para poder tener completo el analizador sintáctico por cartas.

Inicialización

Nosotros no podemos aplicar la regla fundamental si no tenemos arcos definidos en la carta, así que requerimos de al menos un arco activo y uno inactivo para comenzar.

Es muy fácil el trabajo de inicializar la carta. Consiste en que conociendo la primera palabra de la cuerda o la cuerda completa, es fácil encontrar a qué categoría corresponde cada palabra, y agregar un arco inactivo, por cada palabra conocida. A este proceso se le conoce como inicialización.

Regla de invocación

Los arcos generados por la inicialización no son suficientes para comenzar el análisis sintáctico ya que requerimos de un arco inactivo para aplicar la regla fundamental. Un simple principio nos ayuda a lograrlo, cada vez que se agregue un arco inactivo de categoría C a la carta, debemos agregar un arco activo que inicie en el mismo vértice, para cada regla de la gramática que requiera a C como su hijo de más a la izquierda. Esta regla de invocación es la estrategia usada en el analizador sintáctico de abajo hacia arriba. La definición exacta es:

Regla de abajo hacia arriba

Si agregamos un arco <i,j,C ⟶ W1> a la carta, por cada regla en la gramática de la forma B ⟶ CW2 debemos agregar un arco de la forma <i,i,B ⟶ .CW2> a la carta.

Esta regla es aplicada cuando son adheridos arcos inactivos a la carta.

Así operando la regla de abajo hacia arriba e intercalándola con la regla fundamental podemos concluir el análisis sintáctico.

3.3 Implementación del analizador sintáctico por cartas de abajo hacia arriba

Es importante mencionar que un analizador sintáctico puede ser clasificado como un analizador sintáctico de *línea* ó como un analizador sintáctico *fuera de línea*. El primero es aquel que no requiere de toda la cuerda para comenzar a construir la carta, es decir, que puede ir tomando una a una las palabras e ir construyendo la carta. Por el contrario el analizador sintáctico fuera de línea requiere conocer toda la cuerda para poder construir la carta.

Debido a la forma de uso que se le dará al analizador sintáctico como un sistema de inferencia, requerimos que sea un analizador sintáctico en línea.

La implementación de dicho analizador sintáctico se realizó en el lenguaje de programación Prolog, y se utiliza la base de datos de este para almacenar la carta agregando a esta predicados de la forma edge(inicio,final,etiqueta,encontrado,por_encontrar)

Un segmento de código importante es:

```
test( String ) :-
   V0 is 1,
   start_chart( V0, Vn, String ),
   foreach( edge( V0, Vn, s, [], Parse),
      write( Parse)
   ).
start_chart( V0, V0, [] ).
start_chart( V0, Vn, [ Word|Words ] ) :-
   V1 is V0 + 1,
   foreach( word( Category, Word ),
      add_edge( V0, V1, Category, [], [Word,Category])
   ),
   start_chart( V1, Vn, Words ).
```

Donde el predicado start_chart es propiamente el analizador sintáctico y tiene como variables V0 y Vn que representan el número de vértice inicial y final respectivamente, así como String que es una lista donde se encuentra la cuerda a analizar. El predicado add_edge agrega los nuevos arcos generados por las tres reglas del

analizador sintáctico (inicialización, fundamental e invocación).

El código completo de la implementación se encuentra en el apéndice B.

CAPITULO 4
SISTEMAS DE INFERENCIA

La finalidad de este capítulo es mostrar la forma mediante la cual un analizador sintáctico por cartas puede ser utilizado como un sistema de inferencia. Para desarrollar este capítulo se utilizó como texto de apoyo el artículo "LR Parser.as inference systems for fixed-mode logic programs" [3].

4.1 Definición

Consideraremos que un *sistema de inferencia* un conjunto de reglas para puede inferir teoremas a partir de un conjunto de axiomas dado. Como ejemplo tenemos los programas probadores de teoremas o bien aquellos sistemas que basándose en un conjunto de reglas son capaces de encontrar una solución a un determinado problema.

En el presente trabajo se utiliza el sistema de inferencia como un ejecutador de programas escritos en Prolog. Así los predicados del programa a ejecutar se presentan como los axiomas del sistema, y este a su vez encuentra un teorema que representara para nosotros una solución del programa.

La idea principal es que dado un programa en Prolog que este sea ejecutado por el sistema de inferencia y no por el intérprete de Prolog.

4.2 Implementación de un analizador sintáctico por cartas

En el capitulo 3 se presentó un analizador sintáctico por cartas, el cual es capaz de dada una cuerda como entrada y una gramática G(L) determinar si la cuerda pertenece al lenguaje L.

Una característica importante de este analizador sintáctico en que es en "línea", es decir no requiere de toda la cuerda para empezar a construir la carta. Así un analizador sintáctico en linea lo podemos representar por:

```
Π(T,[Word|Words],T'') ←——— π(T,Word,T'),
                            Π(T',Words,T'').
Π(T,[],T).
```

Donde:

T	es la carta inicial
T''	es la carta final
T'	es una carta intermedia
Word	es el primer símbolo de la cuerda
Words	es la cuerda sin su primer símbolo

El predicado Π tiene como parámetros la carta T que es la carta inicial. Así el predicado π genera la nueva carta T' teniendo a T y al primer símbolo de la cuerda (Word), finalmente se llama nuevamente a Π pero con la nueva carta T' como la carta inicial y Words como el resto de la cuerda. El criterio para finalizar es cuando π no puede generar una nueva carta T' a partir de T ya que no hay más símbolos en la cuerda.

La pregunta es de la forma:

?-Π(t,[a1,a2,...,an],T)

donde:
t es el árbol de análisis sintáctico inicial
[a1,a2,...,an] es la cuerda a analizar
T es variable y es el árbol de análisis sintáctico final

La implementación se realizó en el lenguaje Prolog.
El código completo de la implementación se encuentra en el apéndice C.

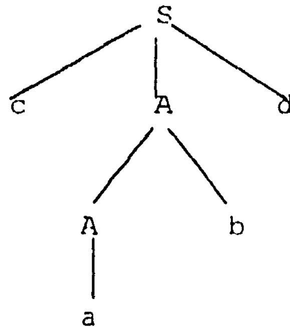
4.3 Implementación de un analizador sintáctico por cartas como un sistema de inferencia

En este momento en importante el presentar por qué podemos utilizar un analizador sintáctico como un sistema de inferencia.

Uno de los pasos que realiza el analizador sintáctico para verificar si una cuerda pertenece a un lenguaje definido por una gramática, es construir el árbol de análisis sintáctico. Por ejemplo dada la gramática:

$S \longrightarrow cAd$
$A \longrightarrow ab$
$A \longrightarrow a$

El árbol de análisis sintáctico para la cuerda "cabd" es:

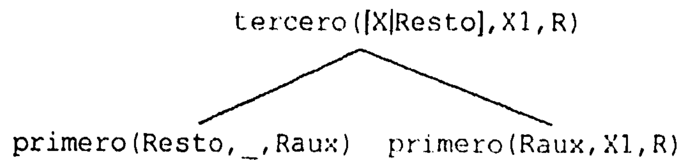


Ahora lo que realiza el intérprete de Prolog al ejecutar un programa es generar un árbol de prueba el cual recorre en busca de soluciones. Por ejemplo para el programa en Prolog:

```
1  primero([X|Resto],X,Resto).

2  tercero([X|Resto],X1,R)  ←— Primero(Resto,_,Raux),
                               primero(Raux,X1,R).
```

El árbol de prueba parcial es:



Nosotros podemos observar que la forma de las reglas de producción de la gramática es:

lado_izquierdo ⟶ lado_derecho

Además el lado izquierdo se encuentra formado por un solo símbolo no terminal y el lado derecho por uno o varios símbolos terminales o no terminales. Para las cláusulas de los programas en Prolog tenemos la forma:

cabeza ⟵ cuerpo

Donde la cabeza se encuentra formada por un solo átomo, y en el cuerpo podemos encontrar ninguno o mas átomos. Podemos concluir que de alguna forma son parecidas las cláusulas de un programa en Prolog y las reglas de producción de una gramática. Otro punto importante es la similitud entre el árbol de análisis sintáctico y el árbol de prueba. Es por ello que la idea principal es escribir programas en Prolog que tengan una forma parecida a la forma de las reglas de producción de una gramática y utilizar el analizador sintáctico para generar un árbol sintáctico, pero realmente generamos un árbol de prueba. Finalmente utilizamos el analizador sintáctico como un generador de cuerdas, y teniendo como reglas de producción un programa en Prolog. El resultado no son las cuerdas del lenguaje, sino el resultado o ejecución del programa. En importante marcar que es necesario que los programas se encuentren en forma cadena.

El sistema de inferencia lo representamos como:

$$\theta(T,X,T'',X'') \longleftarrow \text{NEXT_STATE}(A,X,X'),\ \pi(T,A,T'),\ \theta(T',X',T'',X'')$$

$$\theta(T,X,T,X) \longleftarrow \text{closed}(T).$$

donde:

- T es la carta inicial
- T'' es la carta final
- T' es una carta intermedia
- A es el simbolo inicial
- X es el resultado final

X' es un resultado parcial
A' es un símbolo sugerido para continuar la carta
X'' es el resultado final

El predicado π dada la carta T , genera la nueva carta T', además de proponer el símbolo A, el cual representa un axioma a utilizar. Es importante mencionar que A y A' son realmente predicados del programa en Prolog y que el predicado NEXT_STATE de alguna forma ejecuta o evalúa dichos predicados y transforma el resultado parcial X en X'. Una vez instanciada la variable X' se llama al predicado θ de una forma recursiva con la nueva carta T' y X' como argumentos.

El criterio para terminar es: Cuando el predicado π no puede generar una nueva carta, y el árbol que se ha generado es un árbol de prueba *cerrado,* que es aquel árbol que tiene todas sus hojas cerradas. Ahora una hoja cerrada es aquella que ya a sido resuelta con una cláusula sin cuerpo (cláusula unitaria).

La implementación del sistema de inferencia se realizó en el lenguaje Prolog y la parte del código a remarcar es la siguiente:

```
infl(X0,X1):- clsdb,!,
              inf(1,X0,X1).
inf(V,X,X):-  edge(1,V,s,[],_),
              retract(edge(1,V,s,[],_)).
inf(V,X,X1):- next_sim(V,A),
              parser(V,V1,A),
              next_state(A,X,Xp),
              inf(V1,Xp,X1).
```

Es necesario hacer notar que este sistema de inferencia no tiene la forma exacta mencionada anteriormente, pero realiza la misma función, ya que el predicado inf con ayuda del predicado next_simbol obtiene el símbolo A para continuar la carta, mediante el predicado parser construye la carta, con next_state evalúa un predicado del programa a ejecutar y se auto llama de una forma recursiva. El código completo se encuentra en el apéndice D.

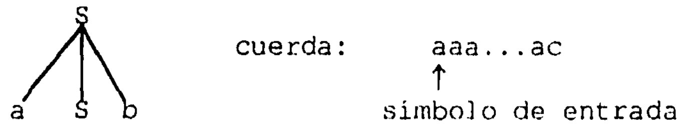
Es importante mostrar que en algunos casos, es conveniente utilizar un analizador sintáctico por cartas como un sistema de inferencia, para ello mostramos el siguiente ejemplo:

Considere la siguiente gramática libre de contexto.

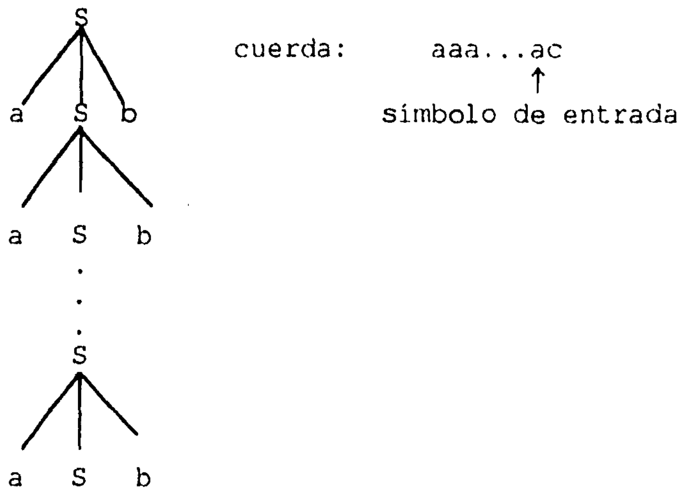
0 $S \longrightarrow a\ S\ b$
1 $S \longrightarrow a\ S$
2 $S \longrightarrow c$

Si utilizamos a Prolog para analizar la cuerda "aaaa...ac" es decir n a's seguidas de una c Prolog repite trabajo, como lo ilustramos a continuación:

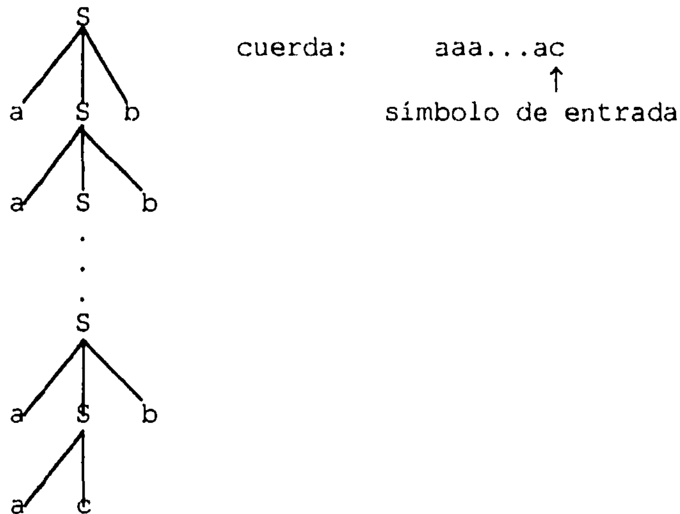
Al aplicar la regla 0 el árbol de prueba es:



Al aplicar la regla 0 n veces el árbol de prueba es:



En este momento la secuencia de reglas aplicadas es: 0000000...0, Prolog detecta que no puede aplicar la regla 0, realiza una refutación y aplica la regla 1 generando el siguiente árbol de prueba:



Podemos decir que la secuencia de reglas aplicadas es: 000000...01, así Prolog tendrá que ir aplicando las reglas en el siguiente orden:

00000...000
00000...001
00000...010
00000...011
00000...100
.
.
.
11111111111

Podemos demostrar que Prolog tarda un tiempo exponencial en la longitud de la cuerda, en realizar el

análisis sintáctico. Si utilizamos en cambio un analizador por cartas, el tiempo de análisis es cúbico en la longitud de la cuerda.

El problema de utilizar dicho analizador es que no podemos incorporar información sensible al contexto. El sistema de inferencia, basado en un analizador por cartas preserva la complejidad cúbica y además permite incorporar información sensible al contexto.

ESTA TESIS NO DEBE
SALIR DE LA BIBLIOTECA

CAPITULO VI
CONCLUSIONES

Al llegar al final del presente trabajo podemos decir que la programación lógica nos presenta un camino diferente al propuesto por los lenguajes procedurales tipo Pascal o C. Un camino el cual presenta grandes expectativas, mediante las cuales se pueden alcanzar objetivos no logrados mediante los lenguajes procedurales o resulta muy difícil de poderlos lograr dichos objetivos.

Es importante destacar que la programación lógica es una herramienta completamente diferente a la programación procedural, y es por ello que en algunos momentos es un poco difícil el tratar de comprender esta forma de programar.

Así debemos considerar la programación lógica como una alternativa más al momento de seleccionar el camino a seguir para resolver un problema mediante un sistema computacional. Existe una gran variedad de problemas que son más fáciles de resolver con herramientas no tan convencionales como: la programación lógica, redes neuronales, y otros.

Es necesario señalar que no se ha podido utilizar prácticamente toda la potencia que ofrece la programación lógica. Esto es debido a que los compiladores o intérpretes serían demasiado ineficientes e imprácticos. Por tal motivo se omiten en estos algunas características para hacerlos más prácticos y eficientes, pero al omitir dichas características se están implementando lenguajes que no se apegan por completo a la programación lógica generando algunas restricciones, las cuales deben ser completamente tomadas en cuenta al momento de escribir un programa en dichos lenguajes.

Lo ideal sería tener un lenguaje con el cual no nos preocupáramos por problemas como: si es ejecutable o no, si las oraciones deben llevar un estricto orden de precedencia, etc. Y la única labor que deberíamos

realizar sería escribir programas completos y ciertos. El desarrollar un lenguaje o la implementación de dicho lenguaje no es una tarea fácil pero es una buena alternativa para buscar algo que aún no tenemos, además pensamos que es un camino en el que existe un buen futuro. Posiblemente un buen camino alterno podría ser el tratar de crear o desarrollar arquitecturas de computadoras orientadas para ejecutar programas lógicos.

Aunque en el presente trabajo se presentó un analizador sintáctico por cartas como un sistema de inferencia, el método presentado funciona para cualquier analizador sintáctico, para gramáticas libres de contexto. Por ejemplo en el articulo "Chart parsers as proof procedures for fixed-mode logic programs" [6] (el cual es la base para el desarrollo del presente trabajo) se presenta un analizador sintáctico simple y se muestra una tabla en la cual se puede observar el tiempo de ejecución para diferentes programas ejecutados por Prolog y por el sistema de inferencia desarrollado, observándose muy buenos resultados. Es importante señalar que el presente trabajo tiene como objetivo el presentar una implementación de lo presentado en dicho articulo.

Las ventajas que ofrece el sistema de inferencia basado en el analizador por cartas con respecto al procedimiento de prueba de Prolog son:

Primero detecta algunos ciclos infinitos que Prolog no detecta.

Segundo recuerda algunos resultados parciales que solo calcula una vez y que Prolog puede calcular varias veces.

Un tema de gran importancia es el de lenguajes formales, ya que nos proporciona los conceptos necesarios para poder comprender la forma inteligente en la que se "unen" programación lógica y el analizador sintáctico para formar un sistema de inferencia.

BIBLIOGRAFIA

[1] Robert R. Korfhagan.
Lógica y algoritmos.
Addison-Wesley Iberoamericana 1970.

[2] W.F. Clocksin y C.S. Mellish.
Programming in Prolog. Springer-Verlag 1987.

[3] David A. Rosenblueth.
LR parsers as inference systems for fixed-mode logic programs.
Reporte técnico preimpreso No 14 IIMAS UNAM 1993.

[4] Alfred V. Aho, Ravi Sethi y Jeffrey D. Ulman.
Compiladores, principios, técnicas y herramientas.
Addison-Wesley Iberoamericana 1990.

[5] Gerald Gazdar y Chris Mellish.
Natural lenguage processing in Prolog.
Addison-Wesley 1989.

[6] David A. Rosenblueth.
Chart parsers as proof procedures for fixed-mode logic programs.
Notas de la conferencia internacional sobre sistemas computacionales de quinta generación, páginas 1125-1132. Tokio, Japón 1992.

APENDICE A

```
%
% El objetivo del presente programa es transformar un
% programa lógico en modo fijo a un programa en forma
% cadena.
%
% mem (X,L) : mem es cierto si X es elemento de L
%
mem( X,[X|Xs] ).
mem( X,[Y|Ys] ) :- mem( X,Ys ).
%
% union( l_a, l_b, l_c )  : L_c, l_a y l_b son listas ,
%                          y l_c es la union de l_a y l_b
%
union( [],X,X ).
union( [U|X],Y,    Z   ) :- mem( U,Y ),!, union( X,Y,Z ).
union( [U|X],Y,[U|Z] ) :- union( X,Y,Z ).
%
% rev2( L1,L2 )  :  L1 y L2 son listas, L2 contiene los
%                   elementos de L1 en orden inverso
%
rev2( L1,L2 ) :- revzap( L1,[],L2 ).
revzap( [],L,L ).
revzap( [X|L],L2,L3 ) :- revzap( L,[X|L2],L3 ).
%
% inter( l_a, l_b, l_c ) : l_a,l_b y l_c son Listas,
%      l_c es la intersección de l_a y l_b
%
inter( [],Y,[] ).
inter( [U|Xs], Y, [U|Zs] ) :-
   mem( U,Y ), not(mem(U,Xs)), !, inter( Xs, Y, Zs ),!.
inter( [U|Xs], Y, Z ):- inter( Xs, Y, Z ),!.
%
% intersecta( L1,L2,L3 ) : L1,L2 y L3 son Listas de
%       listas, cada lista de L3 es la intersección
%       de las listas de L1 y L2 (de la misma posicion)
%
intersecta( [], [],[] ).
intersecta( [Lu|Xs], [Lv|Ys], [Luv|Zs ] ):-
         intersecta( Xs, Ys, Zs ),
         inter( Lu,Lv,Luv ).
%
% menos_primero( L1,L2 ) : L1 es una lista, y L2 es L1 sin
```

```
% su primer elemento
%
menos_primero( [ U|Xs ], Xs ).
%
% menos_ult( L1,L2) :  L1 es una lista, y L2 es L1 sin su
% ultimo elemento
%
menos_ult( L1,L2 ) :- menos_u( L1,L2,E ).
menos_u( [],[],W ).
menos_u( [U|Xs],Ys    ,a ):- menos_u( Xs,Ys,S ), var( S).
menos_u( [U|Xs],[U|Ys],a ):- menos_u( Xs,Ys,S ).
%
% primero( L1, U )  U es el primer elemento de la lista L1
%
primero( [ U|Xs ], U ).
%
% segundo( L1, U )  U es el segundo elemento de la lista
% L1
%
segundo( [ U|Xs ], UU ) :- primero( Xs, UU ).
%
% tercero( L1, U )  U es el tercer elemento de la lista L1
%
tercero( [ U|Xs ], UU ) :- segundo( Xs, UU ).
%
% append( L1, L2, L3 ) las lista L1 y L2, unidas forman L3
%
append( [], Y, Y ).
append( [U|Xs],Y, [U|W] ) :- append( Xs,Y,W ).
%
% intercambio_ler ( L1,L2,L3,L4 ) L3 contiene el primer
% elemeto de la lista L1 y el resto de L2 , y L4 contiene
% el primer elemento de L2 y el resto de L1  el resto de
% una lista es la misma lista sin su primer elemento
%
intercambio_ler( [U|Xs], [UU|Ys], [UU|Xs],  [U|Ys] ).
%
% corrimentoC_1( L1, L2 ) los elementos de L1 se
%     encuentran recorridos circularmente una posicion en
%     L2
%
corrimientoC_1( [ U|Xs], L2 ) :- append( Xs,[U], L2 ).
%
```

```
% union_pro( L1,L2,L3 ) : L1,L2 y L3 son listas de listas,
%   el primer elemento de L3 es la union de L1 con el
%   conjunto Vacio, el segundo, es la union del segundo de
%   L1 con el primero de L3, y asi sucesivamente
%
union_pro( [], [],A ).
union_pro( [ Li|Ls ], [Lii|Lsi],A ) :-
         union( A,Li,Lii ),
         union_pro( Ls, Lsi,Lii ).
%
% union_re( L1,L2,L3 ) : L1,L2 y L3 son listas de listas,
%    el primer elemento de L3 es la union de todas las
%    listas de L1 menos la primera lista L1, el segundo,
%    es la primera lista de L3 menos la segunda de L1 y
%    asi sucesivamente
%
union_re( L1,L2, _ ) :-  rev2( L1,S ),
      union_pro( S,SS,[] ), rev2( SS,L2 ).
%
% regla1( Des, L1 ) : L1 es una lista con simbolos de
%    predicado, y genera la Cabeza y el primer atomo de la
%    primera regla. escribiendola en el archivo Des
%
regla1( Des,L1,S ) :-
         length( L1, Num ),
         primero(L1, A ),
         string_term( Sa, A ),
         Num2 is Num*2-1,
         int_text( Num2, Snum2 ),
         concat([ Sa,'(X0,X',Snum2,'):- ',S,'0(X0,X1)
         '],Ren ),
         write( Ren),  write( Des, Ren ),
         menos_ult( L1,L2 ),
         regla11( Des,L2,1,1,S ),
         write( '.' ),  write( Des, '.' ).
%
% regla11( Des,L1,S1,SS : genera los siguientes atomos de
%    la primera regla generada por la transformacion
%
regla11( Des,[], Num,Num1,S ).
regla11( Des,[A|L1], Num,Num1,S ) :-
         string_term( Sa, A ),
         Naux is Num +1,
```

```
          Num2 is Num1+1,
          Num3 is Num1+2,
          int_text( Num,  Snum  ),
          int_text( Num1, Snum1 ),
          int_text( Num2, Snum2 ),
          int_text( Num3, Snum3 ),
          concat([' ',Sa,'(X',Snum1,',X',Snum2,') ,
          ',S,Snum,'(X',Snum2,',X',Snum3,')'],Ren ),
          write(','),nl,      write(Des,','),nl(Des),
          write( Ren),        write(Des,Ren),
          regla11( Des,L1, Naux,Num3,S ).
%
% regla2 ( Des, L1,L2,Num ) : genera las reglas K(0..n)
%   primas
%
regla2( Des,[],[],Num,S ).
regla2( Des,[U|Xs],[V|Ys], Num,S ) :-
          int_text( Num, Snum ),
          concat([S,Snum,'(',U,',',V,').'],Ren),
          nl,             nl( Des),
          write( Ren ),  write( Des,Ren ),
          Naux is Num + 1,
          regla2( Des,Xs,Ys, Naux,S ).
%
% arranca ( Or, Des,Num ) : incia el proceso de
%   transformacion tomando como entrada el archivo Or y
%   salida el archivo Des
%
arranca( Or, Des,Num ) :-
          lectura( Or, L1,L2,L3,'' ),
      analizar( Or,Des, L1,L2,L3,Num).
%
% lectura( Arch, L1,L2,L3, Si ) : lee cuerdas de Arch, y
%    comieza a separar los simbolos de predicado,
%    terminos, y terminos primos el L1, L2 y L3
%    espectivamente, hasta encontara un punto '.' el en la
%    cuerda leida
%

lectura( Arch, [U|Us], [V|Vs], [W|Ws],Si ) :-
          linea( Arch, Si, Stempo ) ,
          arg_valido( Stempo,_,_ ),
          argumentos( Stempo,U,V,W,Stempo3 ),
```

```
         lectura( Arch, Us, Vs, Ws, Stempo3 ).
lectura( A,[],[],[],_ ).
%
% argumentos( Stempo,U,V,W, Stempo3 ) de la cuerda Stempo,
% obtine : un simbolo de predicado U, y dos terminos V y
% W, Stempo3  es Stempo sin el simbolo de predicado y los
% dos terminos
%
argumentos( Stempo,U,V,W, Stempo3 ) :-
         sim_p( Stempo, U, Stempo1 ),
         argumento( Stempo1, V, Stempo2 ),
         argumento( Stempo2, W, Stempo3 ).
%
% linea( Arch, Sini, Sn_ini ) : Sn_ini es una cuerda
%      formada con concatenando dos cuerdas  Sini y una
%      que es leida de arch, checando que su
%      longitud no sea muy grande y todas las cuerdas que
%      se leen perte necen a una misma regla.
%
linea( Arch, Sini, Sn_ini ) :-
         string_length( Sini, Long ),
         Long < 30, not( string_search('.',Sini,Loc)),
         read_linea( Arch,Saux ),
         blancos( Saux, Saux1 ),
         concat( Sini, Saux1, Sn_ini ).
linea( _, S1,S1 ).
%
read_linea( H, St ):-  read_line( H, St ).
read_linea( _, '.').

blancos( S1,S2) :-
         list_text( L1,S1 ),
         blancos1( L1, L2 ),
         list_text( L2,S2 ).
%
blancos1( [],[] ).
blancos1( [U|Xs],Ys ) :- es_blanco( U ), blancos1( Xs, Ys
).
blancos1( [U|Xs],[U|Ys] ):- blancos1( Xs, Ys ).
%
argumento( S1, A, S2 ) :- arg_valido( S1,A,S2).
argumento( S1, A, S2 ) :- arg_novalido( S1,A,S2).
%
```

```
arg_valido( S1, A, S2 ) :-
        string_length( S1, N_car ),
        string_search( '[',S1, Pos1 ),
        Num1 is Pos1 +1,
        Num2 is N_car - Num1,
        substring(S1,Num1,Num2,Saux),
        cierre_cor(Saux,0,Pos22,1),
        Pos2 is Pos1 + Pos22,
        Pos1aux is Pos1 + 1,
        Num_car is Pos2 - Pos1aux,
        substring( S1, Pos1aux, Num_car, A ),
        Aux1 is Pos2 + 1,
        Aux2 is N_car - Pos2-1,
        substring( S1, Aux1, Aux2, S2 ),!.
%
arg_novalido( S1,'Error_Arg', S1 ).
%
cierre_cor( S,Pos,Pos, Con ) :- Con =:= 0,!.
cierre_cor( S,Pos,Pos2,Con ) :-
        nth_char(0,S,X),
        clasificacion( X, Con,Con1),
        Pos1 is Pos + 1,
        string_length( S,N),
        N1 is N -1,
        substring( S,1,N1,Saux),
        cierre_cor(Saux,Pos1,Pos2,Con1).
%
clasificacion( X,Con,Con1):-
        es_cor_ini(X),
        Con1 is Con +1,!.
%
clasificacion( X,Con,Con1):-
        es_cor_fin(X),
        Con1 is Con -1,!.
clasificacion( X,Con,Con):- true.
%

sim_p( S1, A, S2 ) :-
        primerC( S1,Saux ),
        string_search( '(',Saux, Pos2 ),
        substring( Saux, 0, Pos2, A ),
        string_length( Saux, N_car ),
        Aux1 is N_car - Pos2,
```

```
        substring( Saux, Pos2, Aux1, S2 ).
%
sim_p( S1,'Error_Sim_p', S1 ).
%
primerC( S1, S1 ) :-
        nth_char(0,S1,X),
        es_minuscula( X ).
%
primerC( S1,S2 ) :-
         nth_char(0,S1,X),
         not( es_inicio( X )),
         string_length( S1, L1 ),
         L2 is L1 -1,
         substring(S1,1,L2,Saux),
         primerC( Saux,S2 ).
%
variables( [],[] ).
variables( [S|Xs], [L|Ys] ) :-
        variablel( S, L ),
        variables( Xs, Ys ).
%
variablel( S,[] ) :- not( primeraM(S,_)).
variablel( S, [U|L] ) :-
        variable(S,U,S1),
        variablel( S1,L ).
%
variable( S1, V, S2 ) :-
        primeraM( S1,Saux ),
        fin_var( Saux,V,S2,0).
%
vector(0,[]).
vector(Num,L):-
        Num1 is Num - 1,
        vector( Num1, LL ),
        append(LL,[Q],L).


fin_var( S1,V,S2,Ini ) :-
        nth_char(Ini,S1,X),
        es_fin_var( X ),
        string_length( S1, Long ),
        N_car is Long - Ini,
        substring(S1,0, Ini,V ),
```

```
        substring(S1,Ini,N_car, S2 ).
%
fin_var( S1,V,S2,Ini ) :-
        string_length( S1, Long ),
        Ini < Long,
        Ini1 is Ini + 1,
        fin_var( S1,V,S2,Ini1 ).
%
fin_var( S1,S1,'',0).
        substring( Saux, 0, Pos2, A ),
        string_length( Saux, N_car ),
        Aux1 is N_car - Pos2,
        substring( Saux, Pos2, Aux1, S2 ).
%
primeraM( S1, S1 ) :-
         nth_char(0,S1,X),
         es_ini_var( X ).
%
primeraM( S1,S2 ) :-
         string_length( S1, L1 ),
         L2 is L1 -1,
         substring(S1,1,L2,Saux),
         primeraM( Saux,S2 ).
%
analizar( Or,Des,[],[],[],_ ).
analizar( Or,Des, P,L2,L3,Num_v) :-
         intercambio_ler( L2,L3,Tpaux,T ),
         corrimientoC_1( Tpaux, Tp),
         variables( T  , Vtaux  ),
         variables( Tp, Vtpaux ),
         menos_primero( Vtpaux, Vtp ),
         menos_ult( Vtaux, Vt ),
         union_pro( Vt , Vart , [] ),
         union_re( Vtp, Vartp, [] ),
        intersecta(Vart, Vartp, Piaux ),
        append( [ [],[] ], Piaux, Piaux1),
        corrimientoC_1( Piaux1,Pi),
        gensig( Pi, Sigma ),
        genter( T, Tp, Sigma, ST, STp ),
        vector(Num_v,L),
        varnames([L|S]),string_term(SS,S),
        string_lower(SS,SSS),
        regla1( Des,P ,SSS),
```

```
         regla2( Des,ST, STp ,0,SSS),
         nl,nl,       nl(Des), nl(Des),
         Num1_v is Num_v + 1,
         arranca( Or,Des,Num1_v).
%
es_minuscula( X ) :- X > 96, X <123. % 'a'..'z'
es_inicio(  X )   :- X = 40.         % '<'
es_inicio(  X )   :- X = 60.         % '('
es_ini_var( X )   :- X > 64, X < 91. % 'A'..'Z'
es_ini_var( X )   :- X = 95.         % '_'
es_fin_var( X )   :- X = 124.        % '|'
es_fin_var( X )   :- X = 93.         % ']'
es_fin_var( X )   :- X = 44.         % ','
es_fin_var( X )   :- X = 32.         % ' '
es_blanco(  X )   :- X = 32.         % ' '
es_cor_ini( X )   :- X = 91.         % ']'
es_cor_fin( X )   :- X = 93.         % '['
%
gensig( [],[] ).
gensig( [U|Xs], [UU|Ys] ) :-
         gensig( Xs, Ys ),
         concat_elem( U,UU ).
%
gensig1( [],[]).
gensig1( [U|Xs], [UU|Ys] ):-
         gensig1( Xs,Ys ),
         string_term( UU, U ).
%
concat_elem( [], $St$ ).
concat_elem( L1, S  ) :-
         append( L1,[$|St$],Laux),
         string_term( S1,Laux ),
         string_search(',|',S1,Pos),
         string_length( S1, Long ),
         Pos1 is Pos + 1,
         N_C  is Long - Pos1,
         substring(S1,0,Pos,Cuerda1),
         substring(S1,Pos1,N_C, Cuerda2),
         concat( Cuerda1,Cuerda2, S).
%
genter( [],_,_,[],[] ).
genter( [U|Xs], [V|Ys], [SS|Sig], [ UU|Ws], [VV|Zs] ):-
         concat( ['[ ',SS,',',U,' ]'],UU ),
```

```
        primero( Sig,Vaux ),
        concat( ['[ ',Vaux,',',V,' ]'],VV ),
        genter( Xs,Ys,Sig,Ws,Zs).
%
tt :-   write('Anota el nombre del Archivo de entrada?'),
        read( Or ), concat( Or,'.ari',Origen ),
        write('Anota el nombre del archivo de salida?'),
        read( Des), concat( Des,'.ari',Destino),
        open( Arch_lec, Origen, r ),
        create( Arch_esc,Destino),
        arranca( Arch_lec, Arch_esc,0 ),
        close( Arch_lec),
        close( Arch_esc ),!.
```

APENDICE B

```
%
%
% El objetivo del presente programa es implementar un
% analizador sintáctico por cartas para la gramática
%
%    S ——→ N V Det N
%    S ——→ NP VP
%    NP——→ Det N
%    VP——→ V NP
%    VP——→ V
%    N ——→ maria
%    N ——→ man
%    N ——→ apple
%    N ——→ house
%    V ——→ eats
%    V ——→ sees
%    Det——→ a
%    Det——→ the
%
% La forma de ejecutar el programa es:
% ?- test([a1,a2,a3,..,an]).
%
word(  n, maria ).
word(  n, man   ).
word(  n, apple ).
word(  n, house ).
%
word(  v, eats  ).
word(  v, sees  ).
%
word(det, a     ).
word(det, the   ).
%
rule( s , [n,v,det,n] ).
rule( s , [np,vp] ).
rule( np, [det,n]).
rule( vp, [v,np]).
rule( vp, [v]).
```

```
%
foreach( X,Y ) :-
   X,
   do( Y ),
   fail.
foreach( X,Y ) :- true.
%
do( Y ) :- Y,!.
%
test( String ) :-
   lineas,
   V0 is 1,
   start_chart( V0, Vn, String ),
   foreach( edge( V0, Vn, s, [], Parse),
      write( Parse)
   ),
   lineas,
   clsdb.
%
retractall( Clause ):-
   retract( Clause ),
   fail.
retractall( _ ).
%
clsdb :- retractall( edge(_,_,_,_,_ ) ).
%
start_chart( V0, V0, [] ).
start_chart( V0, Vn, [ Word|Words ] ) :-
   V1 is V0 + 1,
   foreach( word( Category, Word ),
      add_edge( V0, V1, Category, [], [Word,Category])
   ),
   start_chart( V1, Vn, Words ).
%
add_edge( V0, V1, Category, Categories, Parse ) :-
   edge( V0, V1, Category, Categories, Parse ) , !.
%
add_edge( V1, V2, Category1,[], Parse ) :-
   ass( edge( V1, V2, Category1,[], Parse )),


      foreach( rule( Category2, [ Category1|Categories ]),
```

```
      add_edge(
V1,V1,Category2,[Category1|Categories],[Category2])
   ),
   foreach( edge(
V0,V1,Category2,[Category1|Categories],Parses),
      add_edge( V0,V2, Category2,
         Categories,[Parse|Parses] )
      ).
%
add_edge( V0,V1,Category1,[Category2|Categories],Parses )
:- ass( edge(
V0,V1,Category1,[Category2|Categories],Parses)),
   foreach( edge( V1,V2, Category2,[],Parse),
      add_edge( V0,V2, Category1, Categories,
        [Parse|Parses])
   ).
%
ass( X ) :- asserta( X ), write(X),nl.
lineas :- nl,nl.
```

APENDICE C

```
%    El objetivo del siguiente programa es que dada una
% gramatica genere todas las cuerdas que forman el lemguaje
% definido por la gramatica.
% la gramatica predefinida es:
%
%       S   ———→ N V Det N
%       S   ———→ NP VP
%       NP  ———→ Det N
%       VP  ———→ V NP
%       VP  ———→ V
%       N   ———→ maria
%       N   ———→ man
%       N   ———→ apple
%       N   ———→ house
%       V   ———→ eats
%       V   ———→ sees
%       Det———→ a
%       Det———→ the
%
%
word(  n, maria ).
word(  n, man   ).
word(  n, apple ).
word(  n, house ).
%
word(  v, eats  ).
word(  v, sees  ).
%
word(det, a     ).
word(det, the   ).
%
inicial(s).
%
rule( s , [np,vp] ).
rule( np, [det,n]).
rule( vp, [v,np]).
rule( vp, [v]).
%
```

```
foreach( X,Y ) :-
   X,
   do( Y ),
   fail.
foreach( X,Y ) :- true.
%
do( Y ) :- Y,!.
%
test( String ) :-
   lineas,
   V0 is 1,
   parserl( V0,Vn,String),
   foreach( edge( V0, Vn, s, [], Parse),
      write( Parse)
   ),
   lineas,
   retractall( edge( _,_,_,_,_ ) ).
%
parserl( V0,Vn,[Word|Words] ):-
   parser( V0,V1,Word ),
   parserl(V1,Vn,Words).
%
parserl( V0, V0, [] ).
parser( V0,V1,Word ):-
   V1 is V0 + 1,
   foreach( word( Category, Word ),
      add_edge( V0, V1, Category, [], [Word,Category])
   ).
retractall( Clause ):-
   retract( Clause ),
   fail.
retractall( _ ).
%
clsdb :- retractall( edge(_,_,_,_,_ ) ).
%
add_edge( V0, V1, Category, Categories, Parse ) :-
   edge( V0, V1, Category, Categories, _ ) , !.
add_edge( V1, V2, Category1,[], Parse ) :-
   ass( edge( V1, V2, Category1,[], Parse )),


    foreach( rule( Category2, [ Category1|Categories ]),
add_edge(,V1,Category2,[Category1|Categories],[Category2])
```

```
    ),
foreach(edge(V0,V1,Category2,[Category1|Categories],Parses),
     add_edge( V0,V2, Category2, Categories,[Parse|Parses] )
).
%
add_edge( V0,V1,Category1,[Category2|Categories],Parses ):-
   ass(edge(V0,V1,Category1,[Category2|Categories],Parses)),
   foreach( edge( V1,V2, Category2,[],Parse),
      add_edge(V0,V2,Category1,Categories, [Parse|Parses])
   ).
%
ass( X ) :- asserta( X ), write(X),nl.
lineas :- nl,nl.
%
next_sim(V,A):-
   edge(V0,V,N_t,[Category|R],_),
   V0 =\= V,
   re( V0,N_t,Category,A),
   recorrido([Category],A).
%
next_sim(V,'.'):-
   edge(1,V,s,[],_).
%
re(V,N,C,A):-regreso(V,N),!.
re(V,N,C,A):-word(C,A).
%
regreso(1,s).
regreso(V,N_t):-
   edge(V0,V,N_t1,L,_),
   mem(N_t,L),
   regreso(V0,N_t1).
%
recorrido([Category|Resto],A):-
   word(Category,A).
recorrido([Category|Resto],A):-
   rule(Category, L_aux ),
   recorrido( L_aux, A).
%
recorrido(['.'|A],'.').

%
first_sim( A ):- recorrido( [s], A).
%
```

```
generador1( [Word|Words]) :-
   first_sim( Word ),
   generador(1,Vn,Word,Words).
%
generador(V0,V0,'.',[]):-!.
generador(V0,Vn,Word,[Word1|Resto] ):-
   parser(V0,V1,Word) ,
   next_sim(V1,Word1),
   generador(V1,Vn,Word1,Resto).
%
% mem (X,L) : mem es cierto si X es elemento de L
%
mem( X,[X|Xs] ).
mem( X,[Y|Ys] ) :- mem( X,Ys ).
```

APENDICE D

El objetivo del siguiente programa es implementar un sistema de inferencia, para poder ejecutar el programa APPEND como separador de listas. Usualmente la relación APPEND es usada para unir listas, pero en el ejemplo se utilizar para separar listas. el predicado s(<Z>,<X,Y>) es cierto si y solo si Z es una lista que puede ser descompuesta en la lista X seguida de la lista Y

```
s(<Z>,<[],Z>) ←———
s(<W|Z]>,<[W|X],Y>)←——  s(<Z>,<X,Y>)
```

El programa en forma cadena para s es:

$s'(U_0,U_1) \longleftarrow k_0(U_0,U_1)$

$k_0(<St,Z>,<St,[\,],Z>) \longleftarrow$

$s'(U_0,U_3) \longleftarrow b_1(U_0,U_1),\ s'(U_1,U_2),\ b_2(U_2,U_3)$

$\quad b_1(<St,[W|Z]>,<[W|St],Z>) \longleftarrow$

$\quad b_2(<[W|St],X,Y>,<St,[W|X],Y>) \longleftarrow$

```
%
word(  b0a, b0 ).
word(  k0a, k0 ).
word(  kla, kl ).
word(  k2a, k2 ).
%
inicial(s).
rule( s , [a] ).
rule( a , [b0a] ).
rule( a , [k0a,a,kla]).
%
next_state(b0,[St,Y],[St,[],Y]).
next_state(k0,[St,[W|Z]],[[W|St],Z]).
next_state(kl,[[W|St],X,Y],[St,[W|X],Y]).
%
foreach( X,Y ) :-
   X,
   do( Y ),
   fail.
foreach( X,Y ) :- true.
%
```

```
do( Y ) :- Y,!.
%
parser( V0,V1,Word ):-
   V1 is V0 + 1,
   foreach( word( Category, Word ),
      add_edge( V0, V1, Category, [], [Word,Category])
   ).
%
retractall( Clause ):-
   retract( Clause ),
   fail.
retractall( _ ).
%
clsdb :- retractall( edge(_,_,_,_,_ ) ).
%
add_edge( V0, V1, Category, Categories, Parse ) :-
   edge( V0, V1, Category, Categories, _ ) , !.
%
add_edge( V1, V2, Category1,[], Parse ) :-
   ass( edge( V1, V2, Category1,[], Parse )),
   foreach( rule( Category2, [ Category1|Categories ]),
add_edge(V1,V1,Category2,[Category1|Categories],[Category2])
   ),

foreach(edge(V0,V1,Category2,[Category1|Categories],Parses),
      add_edge(V0,V2,Category2,Categories,[Parse|Parses])
).
%
add_edge( V0,V1,Category1,[Category2|Categories],Parses):-
   ass(edge(V0,V1,Category1,[Category2|Categories],Parses)),
   foreach( edge( V1,V2, Category2,[],Parse),
   add_edge(V0,V2,Category1,Categories,[Parse|Parses])).
%
ass( X ) :- asserta( X ), write(X),nl.
%
next_sim(1,A):-
   first_sim(A).
%
next_sim(V,A):-
   edge(V0,V,N_t,[Category|R],_),
   V0 =\= V,
   re( V0,N_t,Category,A),
   recorrido([Category],A) .
```

```
%
next_sim(V,'.'):-
   edge(1,V,s,[],_).
%
re(V,N,C,A):-regreso(V,N),!.
re(V,N,C,A):-word(C,A).
%
regreso(1,a).
regreso(V,N_t):-
   edge(V0,V,N_t1,L,_),
   V0 =\= V,
   mem(N_t,L),
   regreso(V0,N_t1).
%
recorrido([Category|Resto],A):-
   word(Category,A).
%
recorrido([Category|Resto],A):-
   rule(Category, L_aux ) ,  %[!!]
   recorrido( L_aux, A).
%
recorrido(['.'|A],'.').
%
first_sim( A ):- recorrido( [s], A).
%
generadorl( [Word|Words]) :-
   first_sim( Word ),
   generador(1,Vn,Word,Words).
%
% mem (X,L) : mem es cierto si X es elemento de L
%
mem( X,[X|Xs] ).
mem( X,[Y|Ys] ) :- mem( X,Ys ).
%
infl(X0,X1):-
   clsdb,!,
   inf(1,X0,X1).
%
inf(V,X,X):-
   edge(1,V,s,[],_), retract(edge(1,V,s,[],_) ).

inf(V,X,X1):-
   next_sim(V,A),
```

```
parser(V,V1,A),
next_state(A,X,Xp),
inf(V1,Xp,X1).
```