Programación Lógica

Programación Lógica

Lista de Figuras

Lista de Tablas

Parte I

Lenguajes Formales: Conceptos Fundamentales

Capítulo 1

Gramáticas y Lenguajes

El objetivo fundamental de este capítulo es el de introducir los conceptos básicos en el tratamiento e implementación de los lenguajes formales de programación, y ello por una razón fundamental que no es otra que la estrecha relación existente entre el desarrollo de intérpretes o compiladores lógicos por un lado, y la implementación de analizadores sintácticos por otro [7, 10, 16, 20, 21, 27]. A este respecto, el conocimiento de las técnicas clásicas de análisis sintáctico es fundamental para la comprensión efectiva de los problemas planteados en programación lógica, así como de las soluciones aportadas para subsanarlos.

1.1 Gramáticas

Todos los lenguajes, incluso los naturales utilizados en la comunicación humana, poseen una representación formal que viene dada por una estructura descriptiva denominada *gramática*, la cual fija el conjunto de reglas sintácticas que orientan la construcción de textos en el lenguaje deseado. Formalmente, una *gramática* es una 4-upla $\mathcal{G} = (N, \Sigma, P, S)$, donde:

N es un conjunto finito de símbolos *no terminales*, también

denominados *variables*. Cada uno de estos símbolos es una *categoría sintáctica* de la gramática.

Σ es un conjunto finito de símbolos *terminales*, cada uno de los cuales representa una *categoría léxica* de la gramática.

P es el conjunto de *reglas* o *producciones* de la gramática.

S es el *axioma de una gramática* o *símbolo inicial* de la gramática.

Ejemplo 1.1.1 *Un ejemplo simple de gramática puede ser la que representa el lenguaje de las expresiones aritméticas. Para simplificar en lo posible la notación, reflejaremos tan solo la suma y la multiplicación de números*[1]*. En ese caso, denominaremos a nuestra gramática* $\mathcal{G}_A = (N_A, \Sigma_A, P_A, S_A)$ *donde:*

N_A *es el conjunto* $\{S\}$.

Σ_A *es el conjunto* $\{+, *, (,), \textit{número}\}$.

P_A *es el conjunto de reglas expresadas por:*

$$\begin{array}{lcl} S & \rightarrow & S * S \\ S & \rightarrow & S + S \\ S & \rightarrow & (S) \\ S & \rightarrow & \text{número} \end{array}$$

S_A *es en este caso el símbolo* S.

La primera de las producciones indica que una expresión aritmética[2] *puede descomponerse en una multiplicación de expresiones aritméticas. La segunda regla representa la descomposición en una suma de expresiones. La tercera, la descomposición en una expresión con paréntesis y la última la descomposición en un número. Es importante observar que el símbolo* número *representa a toda una categoría léxica, que no es otra que el conjunto de los números.* □

[1] la gramática que se va a mostrar es válida tanto para números enteros como reales o complejos.

[2] denotada aquí como S.

En lo sucesivo, y para unificar criterios, utilizaremos la notación que sigue, para representar el conjunto de símbolos de una gramática:

- $V = N \cup \Sigma$, el conjunto total de símbolos.
- $a, b, c, \ldots \in \Sigma$, los símbolos terminales.
- $A, B, C, \ldots \in N$, los símbolos no terminales.
- $X, Y, Z, \ldots \in V$, símbolos arbitrarios.
- $u, v, w, \ldots \in \Sigma^*$, cadenas de terminales.
- $u_{1..n}$, una cadena $u \in \Sigma^*$ de longitud n.
- $u_{i:j}$, la subcadena de $u_{1..n} \in \Sigma^*$ que va del carácter en la posición i al carácter en la posición j.
- u_i, el carácter de $u \in \Sigma^*$ en la posición i.
- $\alpha, \beta, \gamma, \ldots \in V^*$, cadenas arbitrarias de símbolos terminales y no terminales.
- ε, la *cadena vacía*.

En este punto, estamos en condiciones de introducir el concepto de *derivación* de un símbolo no terminal. Se trata, en definitiva, de expresar la noción de descomposición de una categoría sintáctica compleja en otras más simples e incluso en categorías léxicas, aplicando el concepto de descomposición simple que de algún modo representa la noción de regla sintáctica. Ello introduce el concepto de *derivación directa*. Formalmente, decimos que $\alpha\beta\gamma$ *deriva directamente* $\alpha\delta\gamma$ si y sólo si $\beta \to \delta \in P$, y utilizaremos la notación $\alpha\beta\gamma \Rightarrow \alpha\delta\gamma$. Extendiendo ahora el concepto de derivación, consideramos el de *derivación indirecta*. Así, decimos que $\alpha\beta\gamma$ *deriva indirectamente* $\alpha\delta\gamma$ si y sólo si verifica uno de los dos casos siguientes:

- $\beta \Rightarrow \delta_1 \Rightarrow \delta_2 \Rightarrow \ldots \Rightarrow \delta_n \Rightarrow \delta$, que notaremos $\alpha\beta\gamma \overset{+}{\Rightarrow} \alpha\delta\gamma$.

- $\beta = \delta$ ó $\beta \stackrel{+}{\Rightarrow} \delta$, y en este caso utilizaremos la notación $\alpha\beta\gamma \stackrel{*}{\Rightarrow} \alpha\delta\gamma$. En caso de conocer el número exacto de derivaciones directas, también se usará la notación $\alpha\beta\gamma \stackrel{k}{\Rightarrow} \alpha\delta\gamma$, donde k representa el número de dichas derivaciones.

Ejemplo 1.1.2 *Partiendo de la gramática de las expresiones aritméticas indicada en el ejemplo 1.1.1 tenemos que* $S \stackrel{5}{\Rightarrow} 2 * 3 + 4$*, puesto que:*

$$S \Rightarrow S + S \Rightarrow S * S + S \Rightarrow 2 * S + S \Rightarrow 2 * 3 + S \Rightarrow 2 * 3 + 4$$

□

En la misma línea, dado un lenguaje $\mathcal{L}(\mathcal{G})$ definimos *forma sentencial* recursivamente como:

- El axioma S de $\mathcal{G}$ es una forma sentencial.
- Si $\alpha\beta\gamma \stackrel{*}{\Rightarrow} \alpha\delta\gamma$ es una derivación en $\mathcal{G}$ y $\alpha\beta\gamma$ es una forma sentencial, entonces $\alpha\delta\gamma$ también lo es.

Intuitivamente, una forma sentencial no es otra cosa que un conjunto de símbolos derivables a partir del axioma de la gramática. Un tipo particular y especialmente interesante de formas sentenciales lo forman las *sentencias*, que se definen como las formas sentenciales compuestas únicamente por símbolos terminales.

En concreto, estamos interesados en una clase particular de gramática denominada de *contexto libre* que se caracteriza por la forma peculiar de sus producciones. Las reglas de una gramática de contexto libre deben responder al patrón:

$$A \rightarrow \alpha$$

donde $A \in N$, y α es una secuencia finita de símbolos terminales y no terminales, tal y como lo hacían las reglas de la gramática de las expresiones aritméticas del ejemplo 1.1.1. Distinguiremos entre este conjunto de producciones, dos tipos de especial

interés. En primer lugar diremos que una regla es de *transmisión simple* si tiene la forma

$$A \to B$$

Intuitivamente este tipo de reglas no aumentan la longitud de la cadena derivada, tan solo expresa que los símbolos derivables de B lo son también de A. En cualquier caso, es posible eliminar este tipo de reglas de una gramática de contexto libre [1]. Por otra parte, definimos una $\varepsilon-$*regla* como aquellas de la forma

$$A \to \varepsilon$$

que como en el caso anterior, siempre pueden eliminarse de la gramática inicial obteniendo otra equivalente que genera el mismo lenguaje. En el caso de que ε sea una sentencia del lenguaje, se admite la inclusión de la regla

$$S \to \varepsilon$$

donde S es el axioma de la gramática [1].

En general, nos referiremos a las gramáticas de contexto libre por sus siglas en inglés: CFG's[3].

1.2 Lenguaje generado por una gramática

En este momento estamos ya en condiciones de introducir la noción de *lenguaje generado por una gramática* $\mathcal{G} = (N, \Sigma, P, S)$ como el conjunto:

$$\mathcal{L}(\mathcal{G}) = \{\alpha \in \Sigma^*,\ S \overset{*}{\Rightarrow} \alpha\}$$

esto es, el conjunto de cadenas terminales que pueden ser derivadas a partir del axioma. Más formalmente, el lenguaje generado por una gramática es el conjunto de sentencias para dicha gramática.

En este punto, es conveniente fijar un orden de derivación con el fin de eliminar diferencias irrelevantes en la generación de

[3] por **C**ontext-**F**ree **G**rammars.

las cadenas que constituyen nuestro lenguaje. A este respecto, dada una gramática $\mathcal{G} = (N, \Sigma, P, S)$, y una derivación indirecta $\alpha A \beta \stackrel{*}{\Rightarrow} \alpha\gamma\beta$ en $\mathcal{G}$, diremos que se trata de una *derivación por la derecha* (resp. *por la izquierda*) si y sólo si:

- $A \rightarrow \gamma \in P$
- $\beta \in \Sigma^*$ (resp. $\alpha \in \Sigma^*$)

y utilizaremos la notación $\alpha A \beta \stackrel{*}{\Rightarrow}_{rm} \alpha\gamma\beta$ (resp. $\alpha A \beta \stackrel{*}{\Rightarrow}_{lm} \alpha\gamma\beta$) para representarla. Por defecto supondremos que toda derivación es por la derecha, considerándola una *derivación canónica*.

Ejemplo 1.2.1 *Siguiendo en la línea del ejemplo 1.1.2 y considerando la gramática de las expresiones aritméticas del ejemplo 1.1.1, tenemos:*

$$S \Rightarrow_{lm} (S) \Rightarrow_{lm} (S + S) \Rightarrow_{lm} (a + S) \Rightarrow_{lm} (a + a)$$
$$S \Rightarrow_{rm} (S) \Rightarrow_{rm} (S + S) \Rightarrow_{rm} (S + a) \Rightarrow_{rm} (a + a)$$

□

Una vez hemos introducido formalmente el concepto de lenguaje generado por una gramática, debemos establecer una primera e importante diferenciación en los lenguajes atendiendo al número de derivaciones canónicas existentes para una cadena dada. En esencia, se trata de aproximarse de una forma simple al problema de la *ambigüedad sintáctica* en los lenguajes. Así, desde un punto de vista formal diremos que $\mathcal{G} = (N, \Sigma, P, S)$ es una *gramática ambigua* si y sólo si existe $x \in \mathcal{L}(\mathcal{G})$ para la cual hay al menos dos derivaciones canónicas $S \stackrel{*}{\Rightarrow} x$. En la misma línea, decimos que el lenguaje $\mathcal{L}$ *no es ambiguo* si y sólo si existe una gramática $\mathcal{G} = (N, \Sigma, P, S)$ no ambigua, tal que $\mathcal{L} = \mathcal{L}(\mathcal{G})$. En contraposición, diremos que $\mathcal{L}$ es un *lenguaje ambiguo* cuando no existe una gramática no ambigua que lo genere.

Ejemplo 1.2.2 *La gramática de las expresiones aritméticas del ejemplo 1.1.1 es ambigua puesto que la cadena* $2 + 3 * 4$ *puede derivarse de dos formas distintas:*

$$\begin{array}{lllllllllll} S & \Rightarrow & S+S & \Rightarrow & S+S*S & \Rightarrow & S+S*4 & \Rightarrow & S+3*4 & \Rightarrow & 2+3*4 \\ S & \Rightarrow & S*S & \Rightarrow & S*4 & \Rightarrow & S+S*4 & \Rightarrow & S+3*4 & \Rightarrow & 2+3*4 \end{array}$$

Sin embargo, el lenguaje de las expresiones aritméticas no es ambiguo puesto que es posible definir una gramática no ambigua que lo genere. Dicha gramática puede ser la dada por las reglas siguientes:

$$\begin{array}{ll} E \rightarrow E \ + \ T & E \rightarrow T \\ T \rightarrow T \ * \ F & T \rightarrow F \\ F \rightarrow (\ E\ ) & F \rightarrow \textit{número} \end{array}$$

Estas nuevas reglas introducen implícitamente los conceptos de asociatividad a la izquierda y de prioridad del operador $*$ *sobre el operador* $+$*. Con ello se evitan las ambigüedades en expresiones del tipo*

$$\textit{número}_1 \ op_0 \ \textit{número}_2 \ op_0 \ \textit{número}_3$$

y

$$\textit{número}_1 \ op_1 \ \textit{número}_2 \ op_2 \ \textit{número}_3$$

respectivamente, donde $op_i \in \{+, *\}$, $i \in \{0, 1, 2\}$ *y* $op_1 \neq op_2$. □

Finalmente, es interesante hacer notar que no todas las reglas ni símbolos de una gramática son forzosamente imprescindibles para la generación del lenguaje asociado. En efecto, en al menos dos casos el razonamiento es trivial a partir de la definición misma de lenguaje:

- En el caso en el que un símbolo A no pertenezca a ninguna forma sentencial, es evidente que toda regla de la forma

  $$A \rightarrow \alpha$$

  es irrelevante para la generación del lenguaje. En este caso decimos que A es un *símbolo inaccesible*.

- Si un símbolo $A \in N$ es tal que, aún perteneciendo a una forma sentencial del lenguaje, no existe ninguna regla del tipo

  $$A \rightarrow \alpha$$

  de forma que

  $$A \Rightarrow \alpha \stackrel{*}{\Rightarrow} \omega$$

  es evidente que será imposible generar sentencias del lenguaje a partir de dichas formas sentenciales. En este caso decimos que A es un *símbolo inutil*.

En todo caso es fácilmente demostrable que es posible eliminar tanto símbolos inaccesibles como inútiles de una gramática, sin por ello afectar en absoluto a la integridad del lenguaje generado [1]. A una gramática que no contiene ni símbolos inutiles, ni inaccesibles, ni $\varepsilon-$reglas, ni reglas de transmisión simple, se le denomina *gramática reducida*.

Capítulo 2

Análisis Sintáctico

Una vez construida la gramática de un lenguaje, es necesario poner en marcha un mecanismo de reconocimiento tanto de las categorias léxicas como de las sintácticas, previo a la implementación del mismo en un soporte físico[1]. Al primero se le suele llamar *analizador léxico* y de algún modo constituye el diccionario de nuestro lenguaje. El segundo es el encargado de reconocer en la estructura de nuestro texto, las reglas de la gramática y se le denomina *analizador sintáctico*. Por debajo de este último, suele considerarse la noción de *reconocedor sintáctico* entendido como herramienta que verifica la adecuación de la estructura de la frase a la gramática considerada, mientras que el analizador sintáctico construye además una representación del proceso de análisis. Para facilitar la labor tanto de reconocimiento como de análisis sintáctico, se suelen considerar *gramáticas aumentadas*. Formalmente, dada una gramática $\mathcal{G} = (N, \Sigma, P, S)$, decimos que $\mathcal{G}_a = (N_a, \Sigma_a, P_a, S_a)$ es su correspondiente *gramática aumentada* si verifica las condiciones siguientes:

$$N_a = N \cup \Phi \quad \Sigma_a = \Sigma \quad P_a = P \quad S_a = \Phi$$

donde $\Phi \notin N$. Ello quiere decir que en una gramática aumentada es fácilmente reconocible el final de un proceso de

[1] esto es, en un ordenador.

análisis sintáctico. En efecto, basta con haber reconocido el no terminal Φ, puesto que este sólo aparece una vez y en una sola producción de la gramática que genera el lenguaje. En esta sección comentaremos algunos aspectos interesantes en relación a los analizadores sintácticos.

En general, podemos distinguir dos grandes tipos de analizadores en relación al modo en que abordan el reconocimiento de las reglas de la gramática: los *ascendentes* y los *descendentes*[2]. Un algoritmo de *análisis descendente* es, por definición, aquel que aplica sistemáticamente el concepto de derivación izquierda. Un *análisis ascendente*, por el contrario, es aquel que aplica sistemáticamente el concepto de derivación por la derecha[3]. Esta, aparentemente, pequeña diferencia se traduce en comportamientos fuertemente diferenciados en cuanto al rendimiento del analizador sintáctico, y ello desde dos puntos de vista distintos:

- Por un lado, la eficiencia a nivel de la complejidad tanto temporal como espacial, en lo que a la implementación práctica se refiere. Así, los analizadores descendentes requieren en general menos memoria para su programación. En contraposición, su rendimiento en cuanto a velocidad de tratamiento de las estructuras sintácticas está muy por detrás de los algoritmos de análisis ascendente. Se trata, en definitiva, del viejo enfrentamiento entre eficiencia de tratamiento y minimización de la memoria necesaria para la implementación.

- Por otra parte, el conjunto de gramáticas capaces de ser tratadas de forma efectiva por un algoritmo descendente es inferior, en general, al dominio representado por los analizadores ascendentes. De hecho, el segundo incluye en la práctica al primero.

Una vez aquí, y puesto que este no es un curso de lenguajes formales, intentaremos justificar nuestro razonamiento mediante

[2] también denominados *predictivos*.
[3] existen también algoritmos de análisis sintáctico que aplican técnicas mixtas.

ejemplos simples. Para ello, partiremos de las gramáticas anteriormente consideradas para las expresiones aritméticas: la ambigua del ejemplo 1.1.1 y la determinista introducida al final del ejemplo 1.2.2. Es importante dejar bien claro que las diferencias que vamos a exponer en la discusión que sigue, en cuanto a la aplicación de un tipo u otro de analizador sintáctico, no dependen del hecho de que una gramática sea ambigua y la otra no. El origen será la forma de las reglas de la gramática en cada caso. A este respecto, decimos que $\mathcal{G} = (N, \Sigma, P, S)$ es una *gramática recursiva por la izquierda* cuando es posible una derivación del tipo

$$A \overset{+}{\Rightarrow} A\alpha$$

aplicando reglas en P. En el caso particular en el que existe en P una regla de la forma

$$A \to A\alpha$$

hablamos de *recursividad izquierda directa*. El concepto de recursividad por la izquierda es importante por dos razones:

- En la práctica todos los lenguajes tienden a representarse de forma natural mediante gramáticas recursivas por la izquierda[4].
- Los analizadores descendentes son incapaces de abordar simplemente el análisis de gramáticas recursivas por la izquierda.

Antes de considerar algunos ejemplos ilustrativos, es necesario definir el concepto de *árbol de derivación*. Formalmente, dada una gramática de contexto libre $\mathcal{G} = (N, \Sigma, P, S)$, diremos que un árbol $\mathcal{T}$ etiquetado y orientado, es un *árbol de derivación* para $\mathcal{G}$ si verifica las dos propiedades siguientes:

[4]el paso de una gramática recursiva por la izquierda a otra no recursiva por la izquierda y que genera el mismo lenguaje, es posible en todos los casos, pero el proceso conlleva una pérdida importante de legibilidad de la gramática por parte del diseñador del lenguaje. El método se conoce como paso a la *Forma Normal de Greibach* y puede ser consultado por el lector interesado en [1].

- La raíz de $\mathcal{T}$ está etiquetada con el axioma S de la gramática $\mathcal{G}$.
- Si $\mathcal{T}_1,\ \mathcal{T}_2,\ \ldots,\ \mathcal{T}_n$ son subárboles descendientes directos de la raíz de $\mathcal{T}$, y la raíz de $\mathcal{T}_i,\ i \in \{1,\ \ldots,\ n\}$, está etiquetada con S_i, entonces:
  - $S \rightarrow S_1 S_2 \ldots S_n$ es una producción en P.
  - Cuando $S_i \notin \Sigma^*$, entonces S_i es un árbol de derivación para la gramática $\mathcal{G}_i = (N, \Sigma, P, S_i)$.

En este punto, es importante observar que el axioma de la gramática es la categoría sintáctica a partir de la que se podrán derivar todos los textos analizables según la gramática dada. De ahí el nombre de símbolo inicial de la gramática. En ocasiones, a los árboles de derivación se les denomina simplemente *árboles sintácticos*.

Ejemplo 2.0.3 *Ejemplos simples de árboles de derivación son los mostrados en la figura 2.1 para la gramática ambigua de las expresiones aritméticas considerada en el ejemplo 1.1.1, cuando la entrada viene dada por la cadena* $2 + 3 * 4$. □

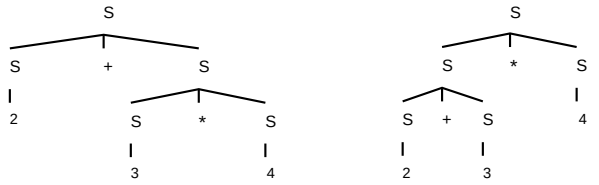


Figura 2.1: Árboles de derivación para las expresiones aritméticas ambiguas con la cadena de entrada $2 + 3 * 4$

A partir del concepto de árbol de derivación, podemos introducir la noción de *frontera de un árbol de derivación* $\mathcal{T}$ como la cadena obtenida al concatenar las hojas[5] de dicho árbol de izquierda a derecha. Evidentemente, si el análisis sintáctico

[5] esto es, los nodos terminales.

ha tenido éxito, la frontera del árbol de derivación ha de coincidir con la cadena analizada.

2.1 Técnicas descendentes

Un analizador sintáctico se denomina descendente cuando construye los árboles de derivación a partir del axioma de la gramática considerada, expandiendo el nodo correspondiente mediante predicciones. Esto es, los analizadores descendentes exploran los árboles en profundidad. De este modo, las técnicas correspondientes son predictivas en cuanto que por construcción deben predecir cuáles son las reglas de la gramática a aplicar en la derivación de un nodo del árbol sintáctico, puesto que en ese estadio de la construcción del árbol podemos no conocer todavía la frontera del nodo en cuestión. Más formalmente, las reglas aplicadas en un *análisis predictivo* se corresponden con una derivación de la cadena de caracteres que constituye la entrada, de izquierda a derecha.

Ejemplo 2.1.1 *Como ejemplo simple, tomemos la gramática de las expresiones aritméticas no ambiguas previamente introducida en el ejemplo 1.2.2. A partir de la misma, realizaremos un análisis descendente muy simple de la cadena* $2 + 3 * 4$. *Comenzaremos por predecir cuáles son las posibles alternativas para derivar el axioma* E *de dicha gramática. En principio, podemos considerar la primera producción, lo cual nos dará como resultado un árbol cuyas primeras ramas son las mostradas en la parte superior izquierda de la figura 2.2. En este punto, es importante observar que la elección de esta regla de derivación podía haber sido en principio la otra posible, esto es,* $E \rightarrow T$. *En este último caso, sólo detectaríamos el error en la elección ante la imposibilidad de derivar el símbolo* $+$.

Una vez salvado el primer escollo, y siguiendo una exploración en profundidad, el siguiente paso es predecir la regla a aplicar para la derivación de la variable E *situada más a la izquierda en el árbol que hemos comenzado a construir. En este*

punto, la elección no adecuada de la misma puede conducirnos a una situación sin salida para nuestro analizador sintáctico. Más exactamente, supongamos que sistemáticamente intentamos aplicar las derivaciones posibles de una variable siguiendo el orden indicado por las reglas de la gramática en cuestión. En ese caso, para derivar E aplicaríamos siempre la regla $E \rightarrow E + T$ en una exploración en profundidad, pero ello nos llevaría a la construcción de árboles con ramas de profundidad infinita como la representada en la figura 2.3. La elección correcta de las reglas, para el análisis descendente de la cadena de entrada que nos ocupa, es la indicada en la figura 2.2 siguiendo los dibujos de izquierda a derecha y de arriba hacia abajo. □

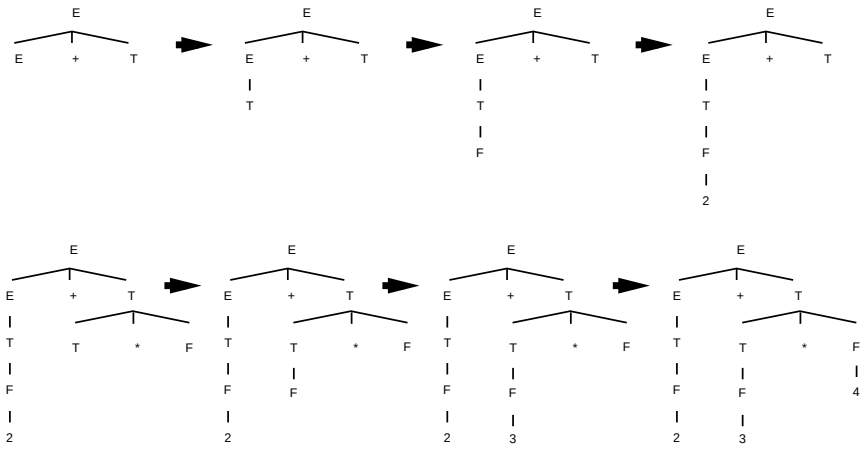


Figura 2.2: Un ejemplo de análisis descendente

El problema planteado en el último ejemplo, acerca de la generación de ramas de profundidad infinita, es consecuencia directa de la aplicación de una construcción en profundidad del árbol sintáctico cuando la gramática que define el lenguaje es recursiva por la izquierda. Ello representa una limitación fundamental para la aplicabilidad de los algoritmos de análisis descendente, por cuanto la práctica totalidad de los lenguajes incluyen la misma de forma natural. En contraposición, este tipo

de algoritmos permiten una implementación extremadamente simple, además de poco costosa desde el punto de vista del ordenador.

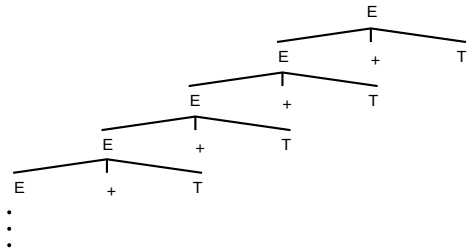


Figura 2.3: Una rama de profundidad infinita

2.2 Técnicas ascendentes

En total contraposición a los analizadores descendentes, en los analizadores ascendentes la derivación de la cadena de caracteres que constituye en cada caso la entrada del analizador, se realiza de derecha a izquierda. Esto es, sólo se considera una regla de la gramática cuando se está seguro de que la misma es idónea para la construcción del árbol en construcción. Más exactamente, el analizador procede reconociendo subárboles que servirán posteriormente para construir otros mayores hasta completar el árbol sintáctico resultante. Ello implica que el sistema debe reconocer la estructura del árbol comenzando por sus niveles inferiores, en primer término por sus hojas. A partir de este punto, el sistema busca las reglas que posibilitan la *reducción*[6] de dichas hojas, para más tarde continuar buscando producciones capaces de reducir los nuevos símbolos así obtenidos hasta la reducción final del símbolo inicial.

Una consecuencia importante de la arquitectura aquí

[6]cuando el tipo de análisis considerado es ascendente, el término *reducción* se aplica para expresar que en una regla, la parte derecha de la misma ha sido totalmente reconocida. En ese momento, el no terminal que constituye la parte izquierda de la regla puede considerarse igualmente reconocido sintácticamente.

considerada para el analizador sintáctico, es la independencia de su aplicabilidad en relación a la presencia o no de recursividad por la izquierda en la gramática que genera el lenguaje utilizado. Ello implica que, en general, todo lenguaje analizable mediante un método descendente puede ser reconocido también por un analizador ascendente. Esto es, el dominio de aplicación de estos algoritmos es notablemente superior al de los descendentes. La otra cara de la moneda es su mayor complejidad de implementación sobre un soporte físico.

Ejemplo 2.2.1 *Para mostrar la técnica general aplicada en los analizadores ascendentes, y en particular su buen comportamiento frente al fenómeno de la recursividad por la izquierda, trataremos el mismo problema planteado en el anterior ejemplo 2.1.1. Así, una vez leído el carácter* 2 *de la cadena de entrada* $2 + 3 * 4$*, el analizador reduce la regla*

$$F \rightarrow \textit{número}$$

dando lugar al subárbol representado en la parte superior izquierda de la figura 2.4. A su vez, una vez reconocido el símbolo

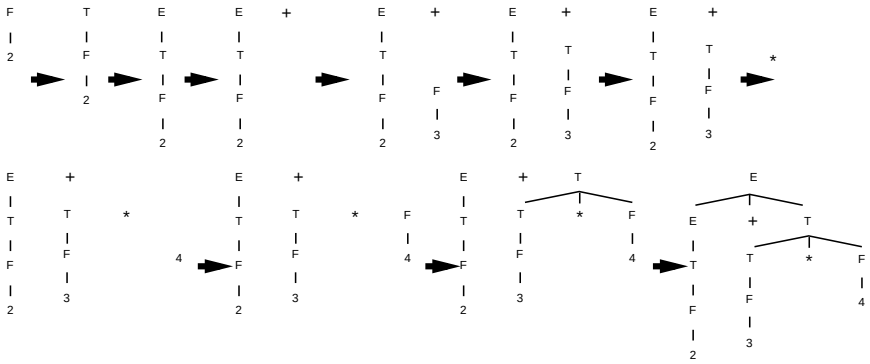


Figura 2.4: Un ejemplo de análisis ascendente

no terminal F*, el analizador puede aplicar la regla*

$$T \rightarrow F$$

para reducir la variable T a partir de la parte del árbol sintáctico ya construido en el paso anterior. De este modo obtenemos el subárbol siguiente representado en la figura 2.4. Un proceso similar se sigue para obtener el siguiente subárbol mediante la aplicación de la regla

$$E \rightarrow T$$

Una vez llegados aquí, la única posibilidad está en continuar leyendo la cadena de entrada, cuyo siguiente carácter es $+$, puesto que no podemos aplicar más reducciones. De este modo obtenemos un conjunto de dos subárboles: el primero heredado del paso anterior y el segundo formado por la hoja $+$. En este punto, todavía no podemos aplicar ninguna reducción y continuamos leyendo la cadena de entrada cuyo siguiente carácter es 3*. Ahora podemos reducir sucesivamente las reglas*

$$F \rightarrow \textit{número} \qquad\qquad T \rightarrow F$$

En principio serían ahora posibles dos alternativas:

- *Considerar una reducción mediante la aplicación de la regla $E \rightarrow T$.*
- *Continuar leyendo en la cadena de entrada. Esto es, aplicar lo que se conoce como un* desplazamiento *en la cadena de entrada.*

Cuando este tipo de situaciones se da, se dice que existe un conflicto de tipo desplazamiento/reducción[7]*. Estos han de resolverse mediante la aplicación de alguna técnica específica. Ello constituye justamente uno de los factores que suele establecer la diferencia entre los distintos tipos de analizadores ascendentes. En nuestro caso, decidirse por una de las dos posibilidades es muy sencillo, puesto que una reducción mediante $E \rightarrow T$ no sería viable al no permitir más tarde la lectura del símbolo $*$. Por tanto, la única salida posible es la lectura*

[7] también es posible la aparición de conflictos de tipo *reducción/reducción*. Por otro lado, es evidente que es innecesario plantearse la existencia de conflictos *desplazamiento/desplazamiento*.

del carácter $*$. *En este estadio, tampoco es posible aplicar nuevas reducciones y continuamos leyendo la cadena de entrada cuyo símbolo siguiente es* 4. *A partir de aquí, aplicamos sucesivamente las siguientes reducciones hasta llegar al final del análisis sintáctico con la reducción del axioma* E*:*

$$T \to T * F \qquad\qquad E \to E + T$$

□

2.3 Eliminación de ambigüedades

Los anteriores ejemplos 1.1.1 y 1.2.2 sirven de presentación para el problema de las ambigüedades en las gramáticas de contexto libre, así como para la introducción de algunas técnicas clásicas en el tratamiento de las mismas. El problema es, en general, complejo y no siempre fácil de abordar. Sin embargo, en casos como los presentados su tratamiento es estándar y servirá de justificación a la metodología aplicada en los lenguajes lógicos de programación para la introducción de nuevos operadores. Más exactamente, en una situación como la presentada en el ejemplo 1.1.1, las ambigüedades de la gramática tienen dos orígenes bien definidos:

- Por un lado, la asociatividad de los operadores no ha sido fijada. Esto es, en el caso particular que nos ocupa, no sabemos si evaluar las expresiones de derecha a izquierda, o de izquierda a derecha. Concretamente, para la expresión $2 + 3 + 4$ tendríamos los dos posibles árboles sintácticos mostrados en la figura 2.5, y ello independientemente de que el tipo de análisis considerado sea ascendente o descendente.

- Por otra parte, las reglas de la gramática no inducen ningún tipo de prioridad en la evaluación de las distintas subexpresiones. Este es el problema planteado en el análisis de $2 + 3 * 4$, tal y como se mostraba en la figura 2.1.

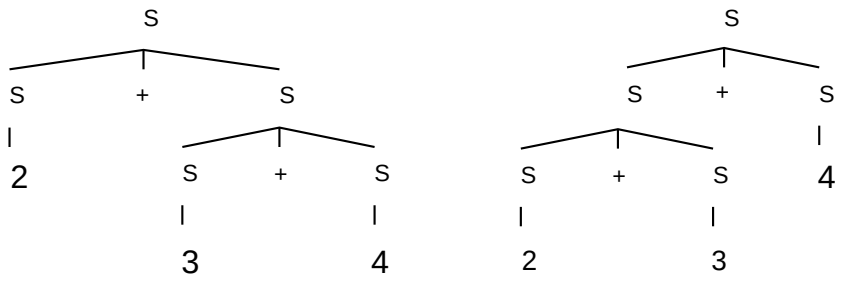


Figura 2.5: Árboles de derivación para las expresiones aritméticas ambiguas con la cadena de entrada $2 + 3 + 4$

En cualquiera de los dos casos considerados, la cuestión inicial es la definición incompleta del mecanismo de evaluación de los operadores. Situaciones de este tipo son muy corrientes en la definición de los lenguajes de programación, y desde luego serán factores a considerar en cualquier protocolo de introducción dinámica de operadores en lenguajes de programación. La solución al problema planteado pasa por la aplicación de una de las dos técnicas siguientes:

- La introducción explícita tanto del tipo de asociatividad considerada para cada operador, como de su prioridad en relación a los demás. La principal ventaja de esta aproximación es que no requiere una alteración previa de la gramática original. Ello permite fácilmente un uso dinámico de la técnica. Por el contrario, es importante constatar que el origen del problema no se ha abordado en absoluto. En efecto, las producciones de la gramática no han sido modificadas.

- La introducción implícita de la asociatividad y de las prioridades entre los operadores, aprovechando el formalismo descriptivo del lenguaje. Este es el acercamiento que permitió eliminar las ambigüedades en el ejemplo 1.2.2, donde la posición de los operadores en la nueva gramática se diferencia claramente en lo que se refiere

a su accesibilidad desde el axioma. Así, la multiplicación aparece representada por un operador situado a mayor profundidad en la estructura de la gramática con respecto a la suma. Ello garantiza para la multiplicación una prioridad de evaluación más elevada que para la adición.

En relación a la introducción de asociatividades para los operadores, la recursividad izquierda en las reglas de la gramática propuesta garantiza una asociatividad por la izquierda. Esto es, la evaluación se realizará de izquierda a derecha.

Con respecto a la solución propuesta anteriormente, esta última es sin duda más elegante y completa. En efecto, hemos abordado el origen mismo del problema, la forma inapropiada de las producciones. Sin embargo, una solución de este tipo no puede ser considerada como dinámica por cuanto su aplicación conlleva un rediseño completo de la gramática que sirve de sustento a nuestro lenguaje.

Parte II

Lógica: Conceptos Fundamentales

Capítulo 3

Cálculo de Proposiciones

Este capítulo constituye una introducción a los conceptos fundamentales del cálculo de proposiciones. El poder expresivo de las lógicas basadas en la noción de proposición es limitado debido sobre todo a la ausencia del concepto genérico de variable[1], pero su conocimiento es imprescindible para comprender posteriormente otras lógicas más avanzadas, como las basadas en el concepto de predicado.

Las lógicas basadas en proposiciones surgen como un intento de formalizar el conocimiento que se posee del mundo, a través de un lenguaje formal cuyas sentencias sean fácilmentes manipulables mediante la aplicación de unas reglas sencillas. Los elementos individuales de las sentencias son identificadores que expresan hechos ciertos o falsos acerca del mundo. Por ejemplo, *casa_roja* es un identificador, cuyo valor puede ser $\mathcal{T}$ (verdadero) o $\mathcal{F}$ (falso). Todos los identificadores son de tipo *booleano*. Este es precisamente el origen de las limitaciones en su uso.

Inicialmente los identificadores pueden construirse tomando enunciados del mundo real que representan conceptos atómicos en el sentido de que no se expresan a partir de una relación entre otros conceptos más simples. De este modo podemos representar el concepto "la casa es roja" mediante la expresión $casa\_roja \Leftarrow \mathcal{T}$.

[1] en lógica de proposiciones sólo existe el concepto de variable boolena.

Para poder expresar un mayor conocimiento acerca del mundo que nos rodea es necesario poder combinar los identificadores para formar expresiones complejas. Los operadores lógicos son los encargados de realizar esta tarea. En la tabla 3.1 se muestran los cinco operadores lógicos fundamentales: la *negación*, la *conjunción*, la *disyunción*, la *implicación* y la *igualdad*. Los cuatro últimos son binarios, mientras que la negación es unaria.

Operador	**Nombre**	**Resultado**
$\wedge$	conjunción	$\mathcal{T}$ si los dos operandos son $\mathcal{T}$.
$\vee$	disyunción	$\mathcal{T}$ si alguno de los dos operandos es $\mathcal{T}$.
$\Rightarrow$	implicación	$\mathcal{T}$ si el segundo operando es una consecuencia del primero.
$=$	equivalencia	$\mathcal{T}$ si los dos operandos tienen el mismo valor.
$\neg$	negación	$\mathcal{T}$ si el operando es $\mathcal{F}$.

Tabla 3.1: Tabla de operadores lógicos

Estos operadores no sólo combinan identificadores para formar sentencias, sino que pueden ser aplicados a las propias sentencias. De este modo, es posible construir sentencias complejas mediante la combinación de otras más simples. Como en todo lenguaje, existe una gramática que expresa la forma en que se deben combinar los distintos elementos para construir frases o sentencias sintácticamente correctas. A continuación se muestra una gramática de contexto libre para la lógica de

proposiciones:

$$
\begin{array}{rcl}
\mathit{Proposición} & \rightarrow & \mathcal{T} \\
& | & \mathcal{F} \\
& | & (\mathit{Identificador}) \\
& | & (\mathit{Proposición} \wedge \mathit{Proposición}) \\
& | & (\mathit{Proposición} \vee \mathit{Proposición}) \\
& | & (\mathit{Proposición} \Rightarrow \mathit{Proposición}) \\
& | & (\mathit{Proposición} = \mathit{Proposición}) \\
& | & (\neg \mathit{Proposición})
\end{array}
$$

La gramática utilizada es muy limitada en el sentido de que las precedencias de evaluación entre operadores, así como sus asociatividades, deben ser indicadas explícitamente por el programador mediante el uso de paréntesis. Ello hace de esta gramática un procedimiento engorroso de descripción, aunque desde un punto de vista meramente orientativo sea válida.

3.1 Evaluación de proposiciones: tablas de verdad

Con la escritura de la gramática ha surgido una cuestión importante: la de la *evaluación de proposiciones*. Ya hemos visto que las proposiciones complejas surgen de la combinación de proposiciones más simples, las cuales pueden tomar valor $\mathcal{T}$ o $\mathcal{F}$. El modo de obtener el valor de la proposición resultante consistirá en realizar una evaluación ascendente de las proposiciones a partir de sus elementos más simples. Los resultados de las evaluaciones de estos se reflejan para cada operador en una tabla denominada *tabla de verdad*. En la tabla 3.2 aparecen representadas las tablas de verdad correspondientes a los cinco operadores lógicos introducidos anteriormente.

p	q	$(\neg p)$	$(p \wedge q)$	$(p \vee q)$	$(p \Rightarrow q)$	$(p = q)$
$\mathcal{T}$	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
$\mathcal{T}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{F}$
$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{F}$
$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$

Tabla 3.2: Tablas de verdad para los operadores lógicos definidos

3.1.1 Evaluación de proposiciones constantes

Por *proposición constante* se entiende aquella que no posee identificadores, es decir, que está construida mediante la aplicación de los operadores directamente sobre los valores $\mathcal{T}$ y $\mathcal{F}$. Ejemplos de proposiciones constantes son: $(\mathcal{T})$, $(\mathcal{T} \vee \mathcal{F})$, $((\mathcal{F} \wedge \mathcal{T}) \Rightarrow \mathcal{T})$.

La evaluación de este tipo de proposiones es sencilla, ya que tan sólo hay que ir sustituyendo, desde los paréntesis interiores hacia los exteriores, la aplicación de un operador por su resultado, que se obtiene directamente de la tabla de verdad.

Ejemplo 3.1.1 *Tomemos la proposición*

$$(((\mathcal{T} \vee (\neg\mathcal{T})) \wedge \mathcal{F}) \vee \mathcal{T})$$

que se transforma en

$$(((\mathcal{T} \vee \mathcal{F}) \wedge \mathcal{F}) \vee \mathcal{T})$$

al sustituir $(\neg\mathcal{T})$ por su resultado según la tabla de verdad, que es $\mathcal{F}$. Al sustituir la primera disyunción obtenemos

$$((\mathcal{T} \wedge \mathcal{F}) \vee \mathcal{T})$$

Ahora, sustituyendo la conjunción, llegamos a

$$(\mathcal{F} \vee \mathcal{T})$$

para obtener finalmente $\mathcal{T}$.□

3.1.2 Evaluación de proposiciones en estados

Cuando las proposiciones no son constantes, hay que tener en cuenta el valor de las variables en el momento en que se evalúa la proposición. Por ejemplo, la proposición

$$(casa\_roja \wedge puerta\_verde)$$

se evalúa a $\mathcal{T}$ si es cierto que nos hallamos ante una casa roja con una puerta verde. Pero este no tiene porqué ser el caso en todo momento.

El conjunto de los valores de todas las variables definidas constituye el *estado* en el que se evalúan las proposiciones. Más formalmente, diremos que un estado S es una función que lleva un conjunto de identificadores en el conjunto $\{\mathcal{T}, \mathcal{F}\}$:

$$S : \{identificadores\} \longrightarrow \{\mathcal{T}, \mathcal{F}\}$$

Sin embargo, no siempre es posible evaluar una proposición en un estado, simplemente porque es posible que no todos los identificadores implicados en la misma estén definidos. Es por ello necesario introducir el concepto de *proposición bien definida*. Así, decimos que una proposición está bien definida en un estado si cada identificador de la proposición está asociado bien a $\mathcal{T}$ o bien a $\mathcal{F}$ en dicho estado.

Ejemplo 3.1.2 *Sea el estado S igual a $\{(p, \mathcal{F}), (q, \mathcal{T})\}$, entonces las siguientes proposiciones están bien definidas:*

$$(p \Rightarrow q)$$

$$(\neg q)$$

$$(p \vee ((p \Rightarrow q) \wedge (\neg p)))$$

mientras que las siguientes no lo están:

$$(\neg a)$$

$$((p \wedge q) \vee a)$$

$$(p \vee ((p \Rightarrow b) \wedge (\neg p)))$$

porque las variables a y b no están definidas en el estado S. □

El concepto de estado es muy importante en lenguajes de programación, ya que durante la ejecución de un programa los valores de las variables van cambiando, provocando la modificación del estado de la máquina. Ello implica que al evaluar una misma expresión en varios lugares de un programa se pueden obtener valores diferentes, ya que el estado de la máquina difiere de un lugar a otro. Así, dado un estado S y una proposición e bien definida en ese estado, suele notarse por $S(e)$ el valor obtenido al reemplazar todas las ocurrencias de los identificadores[2] en el estado por sus valores correspondientes y evaluar las proposiciones constantes resultantes. En este caso, se dice que $S(e)$ es el resultado de evaluar e en S.

Ejemplo 3.1.3 *El resultado de evaluar la proposición*

$$(((\neg p) \vee q) \wedge \mathcal{T})$$

en el estado

$$S = \{(p, \mathcal{T}), (q, \mathcal{F})\}$$

es el siguiente:

$$S\left(((\neg p) \vee q) \wedge \mathcal{T}\right) = S\left(((\neg \mathcal{T}) \vee \mathcal{F}) \wedge \mathcal{T}\right) = \mathcal{F}$$

Este proceso se puede representar mediante la utilización de una tabla de verdad en la que las columnas representan cada una de las subexpresiones que se van evaluando en cada paso. En este ejemplo, dicha tabla tendría el aspecto mostrado en la tabla 3.3. □

3.1.3 Evaluación de proposiciones sin paréntesis

En las expresiones totalmente parentizadas, como son todas las mostradas hasta el momento, los paréntesis se utilizan para definir expresamente el orden de evaluación de las distintas subexpresiones contenidas en una proposición. Dado que se trata de un mecanismo relativamente incómodo, parece natural

[2] esto es, variables booleanas.

p	q	$(\neg p)$	$((\neg p) \vee q)$	$(((\neg p) \vee q) \wedge \mathcal{T})$
$\mathcal{T}$	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$
$\mathcal{T}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$
$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$

Tabla 3.3: Tabla de verdad para la expresión $(((\neg p) \vee q) \wedge \mathcal{T})$

restringir su utilización mediante la definición de precedencias por defecto, tal y como ocurre en los lenguajes de programación. De este modo, el uso de paréntesis quedaría reducido a aquellos casos en los que se desee evaluar las subexpresiones en un orden distinto al establecido por defecto.

Las reglas que se van a considerar en el proceso de establecimiento de las precedencias por defecto son las siguientes:

1. Las subexpresiones formadas por la aplicación consecutiva del mismo operador son evaluadas de izquierda a derecha. De este modo, la expresión $p \vee q \vee r$ es equivalente a $((p \vee q) \vee r)$. Esto es, estamos suponiendo una asociatividad por la izquierda, por defecto, para los operadores lógicos.

2. En las expresiones formadas por la aplicación de distintos operadores, las subexpresiones se evalúan atendiendo a la prioridad de los operadores. La siguiente lista ordenada define dichas prioridades: "$\neg \ \wedge \ \vee \ \Rightarrow =$", donde "$\neg$" es el operador con mayor prioridad y "$=$" el de menor prioridad.

En este momento, una vez establecidas explícitamente las prioridades y asociatividades de los operadores lógicos, podemos eliminar los paréntesis de la gramática inicialmente considerada,

para pasar a la siguiente :

$$\begin{array}{rcl} \mathit{Proposición} & \rightarrow & \mathcal{T} \\ & | & \mathcal{F} \\ & | & \mathit{Identificador} \\ & | & \mathit{Proposición} \wedge \mathit{Proposición} \\ & | & \mathit{Proposición} \vee \mathit{Proposición} \\ & | & \mathit{Proposición} \Rightarrow \mathit{Proposición} \\ & | & \mathit{Proposición} = \mathit{Proposición} \\ & | & \neg \mathit{Proposición} \\ & | & (\mathit{Proposición}) \end{array}$$

donde la última regla permite el cambio explícito de prioridades mediante el uso de paréntesis.

Otra forma de eliminar la obligatoriedad en el uso de paréntesis es la introducción implícita de asociatividades y prioridades en la gramática utilizada para describir el lenguaje de las proposiciones. Esto es, podemos escribir una nueva gramática que contemple las prioridades y asociatividades por defecto. Ello hará que esta sea determinista en el sentido de que sólo sea factible un camino para analizar cada proposición, justamente la causa que nos obligó en un principio a incluir sistemáticamente el uso de los paréntesis. Una posible alternativa es la que se muestra a continuación :

$$\begin{array}{rcl} \mathit{Proposición} & \rightarrow & \mathit{Imp\_exp} \\ & | & \mathit{Proposición} = \mathit{Imp\_exp} \\ \mathit{Imp\_exp} & \rightarrow & \mathit{Exp} \\ & | & \mathit{Imp\_exp} \Rightarrow \mathit{Exp} \\ \mathit{Exp} & \rightarrow & \mathit{Término} \\ & | & \mathit{Exp} \vee \mathit{Término} \\ \mathit{Término} & \rightarrow & \mathit{Factor} \\ & | & \mathit{Término} \wedge \mathit{Factor} \\ \mathit{Factor} & \rightarrow & \neg \mathit{Factor} \\ & | & (\mathit{Proposición}) \\ & | & \mathcal{T} \\ & | & \mathcal{F} \\ & | & \mathit{Identificador} \end{array}$$

En este punto, es importante observar que la precedencia entre operadores queda establecida por el nivel de profundidad en el que aparecen en la gramática. Así, cuanto mayor es esta, mayor es la prioridad. De la misma forma, la asociatividad por la izquierda se introduce implícitamente en la recursividad por la izquierda de las reglas consideradas.

Al igual que en el caso de la gramática inicial con paréntesis, es posible dar una definición de la evaluación de proposiciones dirigida por la sintaxis de la nueva gramática y de carácter recursivo, tal como se muestra a continuación:

$$\begin{array}{rcl} S(< Proposición >) & = & S(< Imp\_exp >) \\ S(< Proposición >) & = & (S(< Proposición >) = S(< Imp\_exp >)) \\ & \vdots & \\ S(< Factor >) & = & S(< Identificador >) \end{array}$$

Evidentemente, cuando se llega al nivel de las expresiones constantes, se utilizarán las tablas de verdad para proceder a su evaluación.

3.1.4 Tautologías

Hemos visto que el resultado de la evaluación de una proposición depende del estado en el cual se realiza dicha evaluación. Sin embargo, hay ciertas expresiones que siempre producen el mismo resultado independientemente del estado en el que se evalúen. Este tipo de proposiciones es particularmente importante por cuanto se corresponden con el concepto intuitivo de *fórmula lógica*. Desde un punto de vista práctico, estas son las proposiciones en las que estamos interesados: aquellas que expresan relaciones ciertas entre objetos, cualesquiera que sean los valores manipulados.

Ejemplo 3.1.4 *Vamos a evaluar la proposición* $(p \Rightarrow q) = (\neg p \vee q)$ *para todos los posibles valores de sus operandos. Esto es equivalente a evaluar la proposición en todos los posibles estados en los que esté bien definida. Para ello basta con construir su tabla de verdad, tal y como se muestra en la tabla 3.4. A partir*

de ésta, es fácilmente verificable la verdad de la proposición, cualquiera que sea el estado en el que sea evaluada, siempre y cuando nuestra proposición esté bien definida. □

p	q	$\neg p$	$\neg p \vee q$	$p \Rightarrow q$	$(p \Rightarrow q) = (\neg p \vee q)$
$\mathcal{T}$	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
$\mathcal{T}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$
$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$

Tabla 3.4: Tabla de verdad para la proposición $(p \Rightarrow q) = (\neg p \vee q)$

A una proposición que, como en el caso del anterior ejemplo 3.1.4, posee un valor constante $\mathcal{T}$ cualquiera que sea el estado en el que esté bien definida se le denomina *tautología*. Dado que éstas constituyen las fórmulas válidas para el cálculo de proposiciones es necesario plantearse una metodología con el objeto de verificar el carácter tautológico de las proposiciones. En relación a este problema, podemos distinguir dos filosofías bien diferenciadas en razón del objetivo que se persiga y teniendo presentes las herramientas introducidas hasta el momento:

- Para demostrar el carácter tautológico tendremos que probar que para todos los casos posibles la evaluación de la proposición da como resultado $\mathcal{T}$. Esto conlleva la construcción y verificación explícita de las tablas de verdad, lo que con toda probabilidad constituirá un proceso muy costoso desde el punto de vista computacional.

- Si lo que pretendemos es descalificar la conjetura tautológica, el proceso es en principio mucho más sencillo, puesto que es suficiente con encontrar un solo caso de estado en el que la proposición esté bien definida y se evalúe a $\mathcal{F}$, para probar que no es una tautología. El proceso resulta computacionalmente menos costoso que en el caso anterior,

al menos cuando seamos capaces de proponer algún sistema efectivo de búsqueda de este tipo de estados.

En cualquier caso, parece claro que el formalismo de las tablas de verdad, si bien extremadamente simple e intuitivo, no es el más conveniente para la puesta en práctica del cálculo de proposiciones. En este punto se hace necesaria la introducción de un entorno de cálculo más manejable y que no exija un retorno continuo a la evaluación al más bajo nivel, representado por las tablas de verdad.

3.2 Reglas de reescritura

Hasta ahora hemos visto un estilo de razonamiento con proposiciones consistente en la utilización de tablas de verdad y en la evaluación exhaustiva. Este tipo de razonamiento tiene un carácter de procesamiento aritmético/lógico, ya que trata de llegar a expresiones constantes que son resueltas aplicando las tablas de verdad para cada operador. Existe un modo de razonamiento basado en proposiciones que utiliza una manipulación de más alto nivel en las mismas, consistente en la derivación de expresiones equivalentes.

La idea general consiste en transformar una proposición compleja en otra más simple que sea equivalente a la anterior, es decir, que el resultado de evaluar ambas en cualquier estado en el que estén bien definidas sea el mismo. Formalmente, diremos que dos proposiciones P y Q son *equivalentes* si y sólo si $P = Q$ es una tautología. En tal caso, se dice que $P = Q$ es una *equivalencia lógica*.

Ejemplo 3.2.1 *La expresión $p \vee q \wedge p$ es equivalente a p. Obsérvese que esta última es más simple, más fácil de manejar.* □

3.2.1 Leyes de equivalencia

No ganaríamos nada con una posible simplificación si para transformar una proposición en otra equivalente tuviésemos que verificar mediante tablas de verdad la validez de tal transformación. Afortunadamente, existe un conjunto de equivalencias, fácilmente comprobables, que se utilizan habitualmente como base para realizar las transformaciones de equivalencia. A continuación se enumeran estas equivalencias, que tradicionalmente reciben el nombre de *Leyes de Equivalencia*:

1. Leyes conmutativas:
   $(E_1 \wedge E_2) = (E_2 \wedge E_1)$
   $(E_1 \vee E_2) = (E_2 \vee E_1)$
   $(E_1 = E_2) = (E_2 = E_1)$

2. Leyes asociativas:
   $E_1 \vee (E_2 \vee E_3) = (E_1 \vee E_2) \vee E_3$
   $E_1 \wedge (E_2 \wedge E_3) = (E_1 \wedge E_2) \wedge E_3$

3. Leyes distributivas:
   $E_1 \vee (E_2 \wedge E_3) = (E_1 \vee E_2) \wedge (E_1 \vee E_3)$
   $E_1 \wedge (E_2 \vee E_3) = (E_1 \wedge E_2) \vee (E_1 \wedge E_3)$

4. Leyes de De Morgan:
   $\neg(E_1 \wedge E_2) = \neg E_1 \vee \neg E_2$
   $\neg(E_1 \vee E_2) = \neg E_1 \wedge \neg E_2$

5. Ley de la negación:
   $\neg(\neg E) = E$

6. Ley del medio excluido:
   $E \vee \neg E = \mathcal{T}$

7. Ley de la contradicción:
   $E \wedge \neg E = \mathcal{F}$

8. Ley de la implicación:
   $E_1 \Rightarrow E_2 = \neg E_1 \vee E_2$
9. Ley de la igualdad:
   $(E_1 = E_2) = (E_1 \Rightarrow E_2) \wedge (E_2 \Rightarrow E_1)$
10. Leyes de la simplificación del $\vee$:
    $E \vee E = E$
    $E \vee \mathcal{T} = \mathcal{T}$
    $E \vee \mathcal{F} = E$
    $E_1 \vee (E_1 \wedge E_2) = E_1$
11. Leyes de la simplificación del $\wedge$:
    $E \wedge E = E$
    $E \wedge \mathcal{T} = E$
    $E \wedge \mathcal{F} = \mathcal{F}$
    $E_1 \wedge (E_1 \vee E_2) = E_1$
12. Ley de la identidad:
    $E = E$

Como hemos comentado con anterioridad, se puede comprobar fácilmente la veracidad de cada una de las equivalencias en relación a la lógica introducida en la discusión precedente, construyendo la correspondiente tabla de verdad. Sin embargo, es importante señalar que el conjunto de leyes enumeradas también podría ser considerado como el conjunto de axiomas a partir del cual definimos nuestra nueva lógica y que en principio no tendría porqué coincidir con la definida por las tablas de verdad. Por otro lado, es necesario ahora completar este núcleo de leyes con un conjunto de reglas capaces de inferir nuevas tautologías a partir de los conocimientos disponibles, esto es, de permitirnos avanzar en nuestro conocimiento.

3.2.2 Reglas de sustitución, resolución y transitividad

Para poder razonar con proposiciones mediante la aplicación de las leyes de equivalencia vamos a introducir tres reglas de

inferencia: la *regla de sustitución*, la de *resolución*, y la de *transitividad*. Ellas nos permitirán ensanchar el horizonte de nuestras posibilidades en lo que al cálculo de proposiciones se refiere y dejarán el camino abierto para el cálculo de predicados.

Regla de sustitución

Formalmente, la regla de sustitución establece que si E_1 y E_2 son proposiciones equivalentes, y $E(p)$ es una proposición escrita como función de uno de sus identificadores[3] p, entonces se cumple que $E(E_1) = E(E_2)$ y $E(E_2) = E(E_1)$, es decir, que $E(E_1)$ y $E(E_2)$ son equivalentes. Utilizando la notación habitual en las reglas de inferencia, podemos escribir la de sustitución en la forma

$$\frac{E_1 = E_2}{E(E_1) = E(E_2), E(E_2) = E(E_1)}$$

Ejemplo 3.2.2 *Tomemos la equivalencia* $(a \Rightarrow b) = (\neg a \vee b)$ *y la proposición* $E(p) = c \vee p$. *Entonces, aplicando la regla de sustitución y tomando:*

$$E_1 = a \Rightarrow b$$
$$E_2 = \neg a \vee b$$

se cumple que $c \vee (a \Rightarrow b) = c \vee (\neg a \vee b)$ *es una equivalencia.* □

Regla de resolución

La regla de resolución será una pieza clave en la construcción de demostradores de teoremas, o lo que es lo mismo, en la construcción de intérpretes lógicos. Formalmente, la regla de inferencia en cuestión es la siguiente:

$$\frac{(\neg a \vee b) \wedge (a \vee c)}{b \vee c}$$

[3] diremos que una proposición esta escrita como función de un identificador si ese identificador aparece en la proposición en la forma de un operando.

Para demostrar la veracidad de esta regla, tendremos que probar que si las premisas son ciertas, entonces también lo es la conclusión. En principio, dos casos son posibles:

1^o caso: Si $\neg a = \mathcal{F}$, entonces puesto que estamos suponiendo que la premisa es cierta, tendremos que $b = \mathcal{T}$. En consecuencia la conclusión es también cierta.

2^o caso: Si $a = \mathcal{F}$, entonces puesto que estamos suponiendo que la premisa es cierta, tendremos que $c = \mathcal{T}$. Por tanto, $b \vee c = \mathcal{T}$.

por lo que en cualquier caso la conclusión es cierta.

Regla de transitividad

Esta regla nos permite encadenar sucesivas aplicaciones de las leyes de equivalencia y de las reglas de inferencia para pasar de una proposición inicial a otra más simplificada, o incluso a $\mathcal{T}$ en el caso de que se trate de una tautología. Formalmente, la regla de transitividad establece que si $E_1 = E_2$ y $E_2 = E_3$ son equivalencias, entonces $E_1 = E_3$ es una equivalencia.

Claramente, esto indica que si de E_1 podemos pasar, mediante la aplicación de las reglas de equivalencia y/o sustitución a E_2, y de esta última por el mismo procedimiento somos capaces de pasar a E_3, entonces el paso de E_1 a E_3 es válido, esto es, E_1 y E_3 son equivalentes. Esto es, tenemos que:

$$\frac{E_1 = E_2, E_2 = E_3}{E_1 = E_3}$$

Ejemplo 3.2.3 *Se trata de demostrar la equivalencia* $(b \Rightarrow c) = (\neg c \Rightarrow \neg b)$*. Para ello se van aplicando sucesivamente diferentes reglas de equivalencia:*

$$\begin{array}{lll} b \Rightarrow c & = \neg b \vee c & (Implicación) \\ & = c \vee \neg b & (Conmutatividad) \\ & = \neg c \Rightarrow \neg b & (Implicación) \end{array}$$

En cada paso se ha explicitado, a modo de aclaración, la regla de equivalencia aplicada. La utilización de signos "=" entre todas las proposiciones indica que se puede aplicar la transitividad desde la primera de ellas hasta la última. □

Ejemplo 3.2.4 *Demostrar la ley de la contradicción. Es suficiente probar que* $\neg(e \wedge \neg e) = \mathcal{T}$. *Para ello partimos de* $\neg(e \wedge \neg e)$ *y vamos buscando proposiciones equivalentes hasta llegar a* $\mathcal{T}$*:*

$$\begin{array}{lll} \neg(e \wedge \neg e) & = \neg e \vee \neg\neg e & (De\ Morgan) \\ & = \neg e \vee e & (Negación) \\ & = e \vee \neg e & (Conmutatividad) \\ & = \mathcal{T} & (Medio\ excluido) \end{array}$$

□

Transformación de implicaciones

Un caso particular muy importante, en relación con la ley de la transitividad, es el de la transformación de implicaciones. Supongamos que tenemos una proposición del tipo

$$E_1 \wedge E_2 \wedge \ldots \wedge E_n \Rightarrow E$$

y que queremos demostrar que se trata de una tautología. Evidentemente, la solución consiste en aplicar las leyes de equivalencia hasta llegar a la proposición constante $\mathcal{T}$. Sin embargo, dicha tarea puede simplificarse notablemente si aplicamos tales leyes en un cierto orden: primero la ley de la implicación y seguidamente la ley de De Morgan. El proceso sería el siguiente:

$$\begin{array}{lll} E_1 \wedge E_2 \ldots \wedge E_n \Rightarrow E & & \\ \quad = (E_1 \wedge E_2 \wedge \ldots \wedge E_n) \Rightarrow E & & \\ \quad = \neg(E_1 \wedge E_2 \wedge \ldots \wedge E_n) \vee E & (Implicación) \\ \quad = \neg E_1 \vee \neg E_2 \vee \ldots \vee \neg E_n \vee E & (De\ Morgan) \end{array}$$

La proposición final es cierta si al menos una de las proposiciones $\neg E_1$, $\neg E_2$, …, $\neg E_n$, E lo es. Estas últimas serán más simples

que la original, por lo que el proceso de comprobación de la tautología será computacionalmente menos costoso. Esta técnica será aplicada, con profusión, en el motor de los intérpretes lógicos.

Ejemplo 3.2.5 *Probar que* $(\neg(b \Rightarrow c) \wedge \neg(\neg b \Rightarrow (c \vee d))) \Rightarrow (\neg c \Rightarrow d)$ *es una tautología. Para ello se realizan los siguientes pasos:*

$$
\begin{array}{ll}
(\neg(b \Rightarrow c) \wedge \neg(\neg b \Rightarrow (c \vee d))) \Rightarrow (\neg c \Rightarrow d) & \\
= \neg\neg(b \Rightarrow c) \vee \neg\neg(\neg b \Rightarrow (c \vee d)) \vee (\neg c \Rightarrow d) & (Implicación\ y\ De\ Morgan) \\
= (b \Rightarrow c) \vee (\neg b \Rightarrow (c \vee d)) \vee (\neg c \Rightarrow d) & (Negación) \\
= \neg b \vee c \vee b \vee c \vee d \vee c \vee d & (Implicación\ y\ De\ Morgan) \\
= \neg b \vee c \vee b \vee d & (Simplificación\ del\ \vee\ ) \\
= \mathcal{T} \vee c \vee d & (Medio\ excluido) \\
= \mathcal{T} & (Simplificación\ del\ \vee\ )
\end{array}
$$

□

3.2.3 Axiomas y teoremas

En la discusión precedente hemos hablado en algún momento de axiomas para referirnos a las leyes de equivalencia que hemos introducido como alternativa para definir nuestra lógica. Nuestro objetivo ahora es el de formalizar el concepto. Así, denominaremos *axioma* al resultado de sustituir proposiciones en las expresiones de las leyes de equivalencia. Esto es, a aquellas proposiciones cuya veracidad es consecuencia inmediata de la aplicación de una o varias sustituciones en las leyes de equivalencia. En la misma línea, denominaremos *teorema* al resultado de aplicar una o varias veces las reglas de sustitución y transitividad sobre un axioma o sobre un teorema. En este sentido, se dice que dichas reglas sirven para inferir teoremas.

Capítulo 4

Cálculo de Predicados

Hemos visto en el capítulo anterior que la lógica definida para las proposiciones permitía expresar conocimiento sobre los hechos del mundo a la vez que proporcionaba un medio de manipular dicho conocimiento para inferir nuevos hechos. Sin embargo, el poder expresivo de las proposiciones está limitado por su incapacidad de generalización. Se pueden indicar los hechos de cada elemento individual del mundo, pero no se puede utilizar una única expresión que abarque a todos los elementos. Tomando un ejemplo clásico, la lógica de proposiciones permite expresar que Manolo, Miguel y Alberto son hombres y son mortales mediante las proposiciones *hombre_Manolo*, *hombre_Miguel*, *hombre_ Alberto*, *mortal_Manolo*, *mortal_Miguel* y *mortal_Alberto*, pero no permite expresar que todos los hombres son mortales por el hecho de ser hombres. Para obtener la capacidad de generalización debemos abandonar el mundo de las proposiciones y penetrar en un nivel más avanzado de la lógica, la *lógica de predicados de primer orden*. Adicionalmente, el uso de predicados nos permitirá utilizar expresiones booleanas[1] en todos aquellos lugares en donde puedan aparecer identificadores en la lógica proposicional.

[1] esto es, expresiones de cualquier tipo con una única característica en común: el resultado de su evaluación debe ser un valor booleano.

4.1 Extensión del concepto de estado

El cálculo de predicados constituye una ampliación del de proposiciones, dotándolo de mayor potencia y capacidad expresiva. Estas mejoras provocan la necesidad de completar conceptos ya definidos en relación al cálculo de proposiciones. Uno de ellos es la noción de *estado*. Así, consideraremos que un *estado* es una función que va de un conjunto de identificadores a un conjunto de valores, eventualmente infinito, que incluye $\mathcal{T}$ y $\mathcal{F}$.

$$S : \{identificadores\} \longrightarrow \{\mathcal{T}, \mathcal{F}, \ldots\}$$

Intuitivamente, un estado representa el contexto en el cual se evalúa un predicado. En tal contexto, cada identificador se encuentra ligado a un *tipo* al que pertenece. Un *tipo* define un conjunto de elementos. El valor de un identificador en un estado deberá estar comprendido en el conjunto de los valores definidos por su tipo. Algunos de los tipos más utilizados se muestran en la tabla 4.1.

Tipo	**Conjunto de valores**
Booleano	$\{\mathcal{T}, \mathcal{F}\}$
Natural	{0,1,2, ...}
Entero	{..., -2, -1, 0, 1, 2, ...}
⋮	⋮

Tabla 4.1: Algunos tipos usados comúnmente

Además es necesario poder extender el conjunto de expresiones manejadas por nuestra lógica, hasta ahora estrictamente limitado a las booleanas. En este sentido denominaremos *expresión atómica* a aquella que no se puede descomponer en otras más simples según la gramática de proposiciones, pero a la que se puede asociar el valor $\mathcal{T}$ o $\mathcal{F}$. Intuitivamente, este tipo de expresiones nos permite ampliar el cálculo proposicional con toda una variedad de fórmulas

no exclusivamente lógicas. De este modo la limitación antes expuesta puede ser fácilmente superada.

Ejemplo 4.1.1 *La subexpresión $x < y$ en la expresión $(x < y) \vee (x > y)$, donde x e y son identificadores que están definidos en el estado en el cual se evalúa esta última, es una expresión atómica.* □

Una vez llegados a este punto, ya no queda más que incluir de forma efectiva el concepto de expresión atómica en nuestra lógica. Con este fin, definiremos un *predicado* como una expresión resultante de sustituir en una proposición, los identificadores por expresiones booleanas o atómicas.

Ejemplo 4.1.2 *A partir de la proposición $a \wedge b$ se puede construir el predicado $(x < y) \wedge b$, resultado de sustituir a por $x < y$.* □

Como consecuencia de la inclusión del concepto de expresión atómica, también será necesario redefinir la noción de evaluación, con el objeto de extenderla al caso de los predicados. Así, diremos que la *evaluación* del predicado e en el estado S, denotada por $S(e)$, es el valor obtenido al sustituir todos los identificadores por su valor en dicho estado. Evidentemente, los identificadores no tienen porqué ser ahora obligatoriamente booleanos como en el caso más restrictivo de las proposiciones. Además en este caso no estamos exigiendo que e esté bien definido en S.

Ejemplo 4.1.3 *Sea P el predicado $(x \leq y \wedge y < z) \vee (x + y < z)$ y S el estado definido por los siguientes pares identificador/valor: $\{(x, 1), (y, 3), (z, 5)\}$. Entonces el resultado de evaluar P en S es el siguiente:*

$$\begin{aligned} S((x \leq y \wedge y < z) \vee (x + y < z)) &= S((\mathcal{T} \wedge \mathcal{T}) \vee \mathcal{T}) \\ &= \mathcal{T} \end{aligned}$$

□

4.2 Los operadores *cand* y *cor*

En la definición de la evaluación de predicados, no se exigía que cada predicado estuviese bien definido en el estado en que se realizaba dicha evaluación. Por consiguiente, en principio, se puede evaluar un predicado en un estado en el cual no todos los identificadores involucrados están definidos. Con el objeto de formalizar este tipo de situaciones se hace necesario introducir un nuevo valor atómico, que denominaremos $\mathcal{U}$, por *indefinido*[2], que acompañe a $\mathcal{T}$ y $\mathcal{F}$ como posible resultado de la evaluación de una expresión booleana.

Como expresión de la utilidad práctica de la posibilidad de evaluación con variables no ligadas a ningún valor, se introducen dos nuevas operaciones denominadas *conjunción condicional* y *disyunción condicional*, y que denotaremos `cand` y `cor`. En definitiva, se trata de variantes del $\wedge$ y $\vee$ respectivamente, que realizan la conjunción y la disyunción sobre la lógica trivaluada resultante. Las tablas de verdad correspondientes a ambos operadores lógicos se muestran en la tabla 4.2. Las operaciones

a	b	a *cand* b	a *cor* b
$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
$\mathcal{T}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$
$\mathcal{T}$	$\mathcal{U}$	$\mathcal{U}$	$\mathcal{T}$
$\mathcal{F}$	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{T}$
$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$
$\mathcal{F}$	$\mathcal{U}$	$\mathcal{F}$	$\mathcal{U}$
$\mathcal{U}$	$\mathcal{T}$	$\mathcal{U}$	$\mathcal{U}$
$\mathcal{U}$	$\mathcal{F}$	$\mathcal{U}$	$\mathcal{U}$
$\mathcal{U}$	$\mathcal{U}$	$\mathcal{U}$	$\mathcal{U}$

Tabla 4.2: Tablas de verdad para los operadores `cand` y `cor`

`cand` y `cor` están implementadas en la mayoría de los lenguajes

[2] *undefined* en inglés.

de programación. Se suelen utilizar preferentemente en la condición de las instrucciones de control tales como `if` o `while`, en las cuales se busca determinar lo antes posible el valor de dichas expresiones.

Estos operadores verifican una serie de interesantes propiedades que de algún modo representan un claro paralelismo en relación a algunas de las leyes de equivalencia ya introducidas. Antes de enumerarlas, es conveniente realizar algunas puntualizaciones acerca de la noción de equivalencia entre predicados cuando el valor $\mathcal{U}$ está presente. Así, decimos que:

- $a \simeq b$, si $a = b$ en los estados en los que a y b están bien definidos.
- $a \equiv b$, si $a = b$ en todos los estados, aun cuando aparezcan valores $\mathcal{U}$ en las expresiones.

A continuación se da una lista de las propiedades de `cand` y `cor`:

1. Leyes asociativas:

   $E_1 \; cand \; (E_2 \; cand \; E_3) \equiv (E_1 \; cand \; E_2) \; cand \; E_3$
   $E_1 \; cor (E_2 \; cor \; E_3) \equiv (E_1 \; cor \; E_2) \; cor \; E_3$

2. Leyes distributivas:

   $E_1 \; cand \; (E_2 \; cor \; E_3) \equiv (E_1 \; cand \; E_2) \; cor \; (E_1 \; cand \; E_3)$
   $E_1 \; cor \; (E_2 \; cand \; E_3) \equiv (E_1 \; cor \; E_2) \;\; cand \; (E_1 \; cor \; E_3)$

3. Leyes de De Morgan:

   $\neg(E_1 \; cand \; E_2) \equiv \neg E_1 \; cor \; \neg E_2$
   $\neg(E_1 \; cor \; E_2) \equiv \neg E_1 \; cand \; \neg E_2$

4. Ley del medio excluido:

   $E_1 \; cor \; \neg E_1 \simeq \mathcal{T}$

5. Ley de la contradicción:

   $E_1 \; cand \; \neg E_1 \simeq \mathcal{F}$

6. Leyes de la simplificación del `cor`:
   E_1 *cor* $E_1 \simeq E_1$
   E_1 *cor* $\mathcal{T} \simeq \mathcal{T}$
   E_1 *cor* $\mathcal{F} \simeq E_1$
   E_1 *cor* $(E_1$ *cand* $E_2) \simeq E_1$
7. Leyes de la simplificación del `cand`:
   E_1 *cand* $E_1 \simeq E_1$
   E_1 *cand* $\mathcal{T} \simeq E_1$
   E_1 *cand* $\mathcal{F} \simeq \mathcal{F}$
   E_1 *cand* $(E_1$ *cor* $E_2) \simeq E_1$

4.3 Cuantificadores

Una de las características de los predicados, y que les confiere mayor fuerza expresiva, es la posibilidad de cuantificar los identificadores. Ello permite expresar propiedades que se refieren a más de un elemento sin tener que referenciar explícitamente dicha propiedad para cada individuo involucrado. Los dos cuantificadores básicos son el *cuantificador universal* y el *cuantificador existencial*. A estos añadiremos un tercero, el *cuantificador numérico* que generaliza los dos anteriores.

4.3.1 El cuantificador universal

El *cuantificador universal* se utiliza para expresar una propiedad que se verifica para todos los elementos de un conjunto. Formalmente aplicaremos la definición que sigue.

Definición 4.3.1 *Sean $m, n \in \mathcal{Z}$, con $m < n$. Entonces*

$$\frac{E_m \wedge E_{m+1} \wedge \ldots \wedge E_{n-1}, i \in [m, n)}{E_i}$$

puede expresarse mediante el uso del cuantificador universal en la forma :

$$(\forall i : m \leq i < n : E_i)$$

donde el conjunto $[m, n)$ *constituye el* rango del cuantificador i. *Esta notación es equivalente a:*

$$\left(\forall_{i=m}^{n-1} E_i\right)$$

□

También podemos dar una definición recursiva del cuantificador universal en la forma:

1. $(\forall i : m \leq i < m : E_i) = \mathcal{T}$
2. $(\forall i : m \leq i < k+1 : E_i) =$
   $(\forall i : m \leq i < k : E_i) \wedge E_k,$ con $k \geq m$

Ejemplo 4.3.1 *Sea el conjunto* $A = \{3, 6, 9, 12, 15\}$. *Para expresar la propiedad de que todos los elementos de A son divisibles por 3, se utiliza el siguiente predicado:*

$$(\forall i : 1 \leq i < 6 : (A_i \bmod 3) = 0)$$

donde A_i *representa al elemento que ocupa la posición* i *en el conjunto A. También se suele escribir*

$$\left(\forall_{i=1}^{5} (A_i \bmod 3) = 0\right)$$

□

4.3.2 El cuantificador existencial

El *cuantificador existencial* permite expresar la existencia de al menos un elemento en un conjunto, que cumple un predicado. Formalmente la definición es la que sigue.

Definición 4.3.2 *Sean* $m, n \in \mathcal{Z}$, *con* $m < n$. *Entonces*

$$\frac{E_i, i \in [m, n)}{E_m \vee E_{m-1} \vee \ldots \vee E_{n-1}}$$

puede expresarse mediante la utilización del cuantificador existencial en la forma :

$$(\exists i : m \leq i < n : E_i)$$

que también podemos denotar en la forma siguiente:

$$\left(\exists_{i=m}^{n-1} E_i\right)$$

□

A imagen de la definición recursiva considerada para el cuantificador universal, en este caso podemos considerar la que sigue:

1. $(\exists i : m \leq i < m : E_i) = \mathcal{F}$
2. $(\exists i : m \leq i < k+1 : E_i) =$
   $(\exists i : m \leq i < k : E_i) \vee E_k,$ con $k \geq m$

Tanto el cuantificador universal como el existencial se utilizan profusamente en el cálculo de predicados y en la aplicación de este a la programación lógica, tal y como veremos en la continuación de este texto. No es ese el caso, en general, del cuantificador numérico que introducimos a continuación.

4.3.3 El cuantificador numérico

Un cuantificador menos conocido que los anteriores es el *cuantificador numérico*, que se utiliza en aquellos casos en los que se desea explicitar el número de elementos dentro de un rango que cumplen un determinado predicado. La notación es

$$(\mathcal{N} i : m \leq i < n : E_i)$$

donde $m, n \in \mathcal{Z}$, con $m < n$.

Ejemplo 4.3.2 *Supongamos que se desea expresar que k es el más pequeño de los números naturales tales que $E_k = \mathcal{T}$. Se puede expresar de la siguiente forma, haciendo uso del cuantificador universal:*

$$(\{i, k\} \subset \mathcal{N}) \wedge (\forall i : 0 \leq i < k : \neg E_i) \wedge E_k$$

pero también lo podemos hacer considerando el cuantificador numérico en la forma

$$(\{i,k\} \subset \mathcal{N}) \wedge ((\mathcal{N}i : 0 \leq i < k : E_i) = 0) \wedge E_k$$

Con esta última expresión se indica que el número de valores que cumplen E_i es 0 en el rango que va de 0 a $k-1$ y que E_k se cumple. □

Ejemplo 4.3.3 *Supongamos que deseamos expresar el hecho de que k es el segundo número natural más pequeño tal que $E_k = \mathcal{T}$. Utilizando el cuantificador universal, el predicado resultante sería:*

$$(\{i,j,k\} \subset \mathcal{N}) \wedge (\forall i : 0 \leq i < j : \neg E_i) \wedge E_j \\ \wedge (\forall i : j+1 \leq i < k : \neg E_i) \wedge E_k$$

Mediante el uso del cuantificador numérico el predicado resultante es más reducido y comprensible:

$$(\{i,k\} \subset \mathcal{N}) \wedge ((\mathcal{N}i : 0 \leq i < k : E_i) = 1) \wedge E_k$$

ya que se expresa explícitamente que hay un solo número i tal que E_i se cumple en el rango $[0,k)$, y que E_k se verifica igualmente. □

En este momento es importante señalar que, como han mostrado los ejemplos 4.3.2 y 4.3.3, el cuantificador numérico no es más que una facilidad notacional para abreviar situaciones que se pueden perfectamente expresar mediante los cuantificadores universal y existencial.

4.3.4 Equivalencias entre cuantificadores

Los distintos cuantificadores se encuentran relacionados unos con otros. De entre todas estas relaciones, una de las más importantes es la equivalencia de cuantificadores que se establece en la siguiente discusión:

Lema 4.3.1 *Sean $m, n \in \mathcal{Z}$. Entonces se cumple que:*

$$(\forall i : m \leq i < n : E_i) = \neg\,(\exists i : m \leq i < n : \neg E_i)$$

Demostración:

Bastará aplicar las definiciones de los cuantificadores y las leyes de equivalencia. Así, tenemos que:

$$\begin{array}{ll}
(\forall i : m \leq i < n : E_i) & \\
\quad = E_m \wedge E_{m+1} \wedge \ldots \wedge E_{n-1} & \\
\quad = \neg\neg(E_m \wedge E_{m+1} \wedge \ldots \wedge E_{n-1}) & (Negación) \\
\quad = \neg(\neg E_m \vee \neg E_{m+1} \vee \ldots \vee \neg E_{n-1}) & (De\ Morgan) \\
\quad = \neg(\exists i : m \leq i < n : \neg E_i) &
\end{array}$$

□

En relación al cuantificador numérico, este se encuentra estrechamente ligado a los cuantificadores existencial y universal, tal y como se había apuntado con anterioridad en los ejemplos 4.3.2 y 4.3.3. En realidad, estos dos últimos pueden ser sustituidos por el cuantificador numérico, ya que se verifican las siguientes igualdades:

$$\begin{array}{lcl}
(\exists i : m \leq i < n : E_i) & = & ((\mathcal{N} i : m \leq i < n : E_i) \geq 1) \\
(\forall i : m \leq i < n : E_i) & = & ((\mathcal{N} i : m \leq i < n : E_i) = n - m)
\end{array}$$

Por otro lado, las expresiones que incluyen cuantificadores pueden ser simplificadas con frecuencia, cuando estos son del mismo tipo. Este es el caso de los rangos consecutivos, tal y como se muestra a continuación:

$$\begin{array}{lcl}
(\exists i : m \leq i < n : E_i) & \vee & (\exists i : n \leq i < p : E_i) \\
 & & \quad = (\exists i : m \leq i < p : E_i) \\
(\forall i : m \leq i < n : E_i) & \wedge & (\forall i : n \leq i < p : E_i) \\
 & & \quad = (\forall i : m \leq i < p : E_i) \\
(\mathcal{N} i : m \leq i < n : E_i) & + & (\mathcal{N} i : n \leq i < p : E_i) \\
 & & \quad = (\mathcal{N} i : m \leq i < p : E_i)
\end{array}$$

4.3.5 Cuantificación sobre rangos infinitos

Hasta ahora se han considerado únicamente rangos finitos en los cuantificadores, lo que limita la expresividad de los predicados. Se trata, por tanto, de introducir el concepto de *rango infinito*, teniendo en cuenta que tales predicados no pueden, en ocasiones, ser calculados mediante programas reales en un tiempo finito. Como primer paso, se generalizan los cuantificadores existencial y universal en la forma:

$$(\exists i : R : E)$$
$$(\forall i : R : E)$$

donde i es un identificador, y R y E son predicados. Declarativamente leeríamos, respectivamente:

"Existe un i que verifica R tal que E es $\mathcal{T}$."
"Para todo i que verifica R, E es $\mathcal{T}$."

Al predicado R se le denomina el *rango del cuantificador*.

Ejemplo 4.3.4 *Dados los siguientes conceptos atómicos y predicados asociados:*

Conceptos atómicos	**Predicados**
p es hombre	$hombre(p)$
x es mortal	$mortal(x)$

entonces, el enunciado "todo hombre es mortal" *puede expresarse mediante el predicado*

$$(\forall p : hombre(p) : mortal(p))$$

□

Ejemplo 4.3.5 *Podemos establecer que el máximo de un número real y su opuesto es el valor absoluto de ese número real, en la forma:*

$$(\forall n : real(n) : abs(n) = max(n, -n))$$

donde $real(n)$ es cierto si n es un número real. □

En general, el tipo del identificador cuantificado viene dado explícitamente por su rango, como en el caso del ejemplo anterior con $real(n)$. Sin embargo, en los casos en que dicho test es obvio, se suele omitir. Así, la expresión del ejemplo anterior se podría escribir:

$$(\forall n : abs(n) = max(n, -n))$$

Continuando con el mismo ejemplo, resulta evidente que

$$abs(n) = max(n, -n)$$

para cualquier real n. Esto significa que nos encontramos ante una tautología, puesto que es $\mathcal{T}$ en cualquier estado en el que esté bien definida. Esto es lo que se expresa con

$$(\forall n : real(n) : abs(n) = max(n, -n))$$

En general, una tautología $E(i_1, i_2, \ldots, i_n)$, donde los i_j son los identificadores contenidos en E, puede considerarse como una abreviatura de un predicado en el cual todos los identificadores se encuentren cuantificados universalmente. Más concretamente:

$$E = (\forall i_1, i_2, \ldots, i_n : E)$$

4.4 Identificadores libres y ligados

En el cálculo de predicados, hay dos clases diferentes de identificadores que se distinguen por su visibilidad:

- Los *identificadores libres* son aquellos cuyo ámbito abarca todos los predicados.
- Los *identificadores ligados* o *acotados* son aquellos cuyo valor sólo es visible dentro de la parte acotada por un cuantificador.

Para aclarar más los conceptos, tomamos como ejemplo el predicado:

$$(\forall i : m \leq i < n : x + i = 0)$$

En este caso, el identificador i está acotado por el cuantificador universal y sólo sirve para indicar elementos individuales dentro de su rango, por lo que su utilización fuera del ámbito del cuantificador no tiene sentido. Podría decirse que i sólo está definido en el contexto local del cuantificador. En cambio, x es un identificador libre cuyo valor forma parte del contexto global y puede ser utilizado sin problemas en cualquier otro predicado. A continuación se describe formalmente cuándo un identificador es libre en una expresión:

1. i es libre en la expresión i.
2. i es libre en la expresión (E), si es libre en E.
3. i es libre en *op* E, donde *op* es un operador unario, si es libre en E.
4. i es libre en E_1 *op* E_2, donde *op* es un operador binario, si es libre en E_1 o en E_2.
5. i es libre en $(\forall j : m \leq j < n : E)$, $(\exists j : m \leq j < n : E)$ y $(\nexists j : m \leq j < n : E)$ si lo es en m, en n o en E, siempre que $i \neq j$.

Igualmente, a continuación se describe formalmente cuándo un identificador está acotado en una expresión:

1. i está acotado en (E) si lo está en E.
2. i está acotado en *op* E, donde *op* es un operador unario, si lo está en E.
3. i está acotado en E_1 *op* E_2 si lo está en E_1 o en E_2.
4. i está acotado en $(\forall i : m \leq i < n : E)$, $(\exists i : m \leq i < n : E)$ y $(\nexists i : m \leq i < n : E)$. En este caso, los predicados precedentes delimitan la *visibilidad* del identificador i.
5. i está acotado en $(\forall j : m \leq j < n : E)$, $(\exists j : m \leq j < n : E)$ y $(\nexists j : m \leq j < n : E)$ si lo está en m, n o E.

Con el objeto de evitar problemas en la identificación de las variables, se establece la regla de que un identificador no puede estar acotado simultáneamente por dos cuantificadores. Esto quiere decir que no se puede dar el siguiente caso:

$$(\forall i : m \leq i < n : i * 3 = 6) \wedge (\exists i : s \leq i < t : i > 0)$$

ya que i está acotado tanto por el cuantificador universal como por el existencial. Este caso se puede evitar reescribiendo la expresión anterior para obtener otra en la cual la variable acotada por uno de los cuantificadores ha sido renombrada:

$$(\forall i : m \leq i < n : i * 3 = 6) \wedge (\exists j : s \leq j < t : j > 0)$$

con lo cual podemos asegurar que nuestro convenio no supone en ningún momento una pérdida de generalidad en la potencia expresiva de los predicados.

4.5 Sustitución textual

Una operación fundamental en la transformación de predicados es la sustitución de un identificador por una expresión, o *sustitución textual*. Ello nos permitirá particularizar una tautología, adaptándola al problema concreto a tratar en cada caso. Formalmente:

Definición 4.5.1 *Sean E y e expresiones y x un identificador. Entonces, la notación E_e^x representa la expresión obtenida al sustituir en E todas las ocurrencias libres de x por e.* □

Ejemplo 4.5.1 *Se desea sustituir la variable x por la expresión z en $(x + y)$. El resultado es el siguiente:*

$$(x + y)_z^x = (z + y)$$

□

Como advertencia, debemos hacer notar que en general es aconsejable utilizar paréntesis en la expresión que va a sustituir

al identificador, puesto que en ciertos casos su ausencia puede hacer que la expresión resultante agrupe operaciones de forma no prevista ni deseada.

Ejemplo 4.5.2 *Se desea sustituir el identificador y en la expresión $(x * y)$ por la expresión $a + b$. Si no se utilizan paréntesis rodeando a esta última expresión, el resultado sería:*

$$(x * y)^y_{a+b} = (x * a + b)$$

La mayor prioridad del $*$ frente al $+$ hace que la operación del producto se realice primero. Por el contrario, con paréntesis se obtendría:

$$(x * y)^y_{(a+b)} = (x * (a + b))$$

En este caso, los paréntesis salvaguardan el orden de las operaciones a realizar, y con ello la semántica de la expresión. □

Evidentemente, si no hay ninguna ocurrencia libre del identificador que va a ser reemplazado, la expresión resultante será igual a la original, pues tal y como quedó establecido en la definición 4.5.1, tan sólo pueden ser reemplazadas ocurrencias libres de identificadores.

Ejemplo 4.5.3 *Sea $E = x < y \wedge (\forall i : 0 \leq i < n : b[i] < y)$. Entonces $E^i_k = E$, puesto que i no es libre en E.* □

Enlazando la presente discusión con la relativa al cálculo de proposiciones, podrá observarse que las sustituciones textuales ya habían sido utilizadas previamente, aunque con otra notación, cuando se había introducido la *regla de sustitución*. En efecto, si E es una expresión en función de x, se cumple que $E^x_z = E(z)$.

En esta línea, de la misma forma que en el caso de las proposiciones se podía aplicar la regla de sustitución de forma consecutiva, en el caso que ahora nos ocupa se pueden encadenar múltiples sustituciones textuales. Esto significa que se pueden sustituir identificadores en una expresión resultado de una sustitución previa. Los paréntesis identifican el orden en que

se realizan tales sustituciones: primero se sustituyen los niveles más internos de paréntesis.

Ejemplo 4.5.4 *Sea E la expresión $x < y \wedge (\forall i : 0 \leq i < n : b[i] < y)$. Entonces:*

$$\left(E^{y}_{(w*z)}\right)^{z}_{(a+u)} = (x < (w * z) \wedge (\forall i : 0 \leq i < n : b[i] < (w * z)))^{z}_{(a+u)}$$

□

Una observación importante es que al realizar sustituciones debemos tener en cuenta que la expresión resultante sea sintácticamente correcta. Así, con la expresión E del anterior ejemplo 4.5.4 no se podría hacer E^{b}_{c+1} puesto que daría lugar a la aparición de expresiones del tipo $c + 1[i]$ que son incorrectas. Un problema adicional que surge al aplicar sustituciones está relacionado con los posibles conflictos que puedan surgir con los identificadores acotados por cuantificadores. Como ilustración, supongamos que en el ejemplo anterior deseamos expresar que x y los elementos del vector b son menores que $y - i$, donde i es un identificador libre. En principio, expresaríamos la situación mediante:

$$E^{y}_{(y-i)} = x < (y - i) \wedge (\forall i : 0 \leq i < n : b[i] < (y - i))$$

lo cual no se corresponde con la idea original. En definitiva, se debe de tener cuidado con que los identificadores que aparezcan en la expresión por la que se sustituye no estén ligados en la expresión original. Una forma sencilla de evitar este problema consiste en renombrar los identificadores de forma automática, considerando únicamente las sustituciones a variables libres. Todo ello nos lleva a matizar la definición de la notación E^{x}_{e}.

Definición 4.5.2 *E^{x}_{e} denota el predicado creado por el reemplazamiento de toda ocurrencia libre de la variable x en la expresión e. Además, para ser válida, debe ser sintácticamente correcta. Finalmente, si la sustitución provocase la acotación*

de alguna variable en e, los identificadores de las expresiones cuantificadas en E deberán ser reemplazadas de forma conveniente. □

Ejemplo 4.5.5 *Sea E la expresión* $x < y \wedge (\forall i : 0 \leq i < n : b[i] < y)$. *El resultado de sustituir en E el identificador y por* $y - i$ *es el siguiente:*

$$E^{y}_{(y-i)} = x < (y - i) \wedge (\forall k : 0 \leq k < n : b[k] < (y - i))$$

donde se ha cambiado el nombre del antiguo identificador ligado i por el de k, para evitar que el nuevo i estuviese ligado.□

4.6 Sustitución simultánea

Sin embargo, el concepto de sustitución textual no será suficiente para la continuación de nuestro trabajo, por cuanto su acción se limita a una sola variable. Ello representa una seria barrera para la expresividad de lo que serán nuestros programas lógicos, en los que la translación de información entre sus diferentes componentes probablemente exija la manipulación simultánea de un conjunto de sustituciones. Es por tanto necesario introducir un concepto más potente, el de *sustitución simultánea.*

Definición 4.6.1 *Sean* $\bar{x} = \{x_i, i = 1, \ldots, n\}$ *y* $\bar{e} = \{e_i, i = 1, \ldots, n\}$ *dos vectores de distintos identificadores x y de expresiones e, respectivamente. Ambos vectores tienen la misma longitud. Entonces definimos* $E^{\bar{x}}_{\bar{e}}$, *ó* $E^{x_1,x_2,\ldots,x_n}_{e_1,e_2,\ldots,e_n}$, *como la* sustitución simultánea *de todas las ocurrencias libres de* x_i *por su correspondiente* e_i *en la expresión E.* □

Ejemplo 4.6.1 *A continuación se muestran varios ejemplos de sustituciones simultáneas:*

- $(x + x + y)^{x,y}_{a+b,c} = a + b + a + b + c$
- $(x + x + y)^{x,y}_{x+y,z} = x + y + x + y + z$

- $(\forall i : 0 \leq i < n : b(i) \vee c(i+1))^{n,b}_{n+i,d}$
  $= (\forall k : 0 \leq k < n + i : d(k) \vee c(k+1))$

□

Debemos observar que, en general, una sustitución simultánea no es quivalente a la realización de sucesivas sustituciones simples, esto es, $E^{x,y}_{u,v}$ no es quivalente a $(E^x_u)^y_v$. Por ejemplo:

$$(x + x + y)^{x,y}_{x+y,z} = x + y + x + y + z$$

mientras que

$$\Big((x + x + y)^x_{x+y}\Big)^y_z = (x + y + x + y + y)^y_z = x + z + x + z + z$$

Capítulo 5

Prolog y Cálculo de Predicados

En este capítulo se aborda la relación existente entre el cálculo de predicados y el lenguaje de programación Prolog, aplicando la teoría explicada en los capítulos anteriores. Más exactamente, construiremos un demostrador de teoremas cuyo funcionamiento se corresponderá casi exactamente con el presentado por el lenguaje citado. A la vez, aprovecharemos dicho proceso para introducir la terminología propia de Prolog, así como sus conceptos básicos.

5.1 Regla de resolución

Recordemos ahora la regla de resolución del cálculo de proposiciones:

$$\frac{(\neg a \vee b) \wedge (a \vee c)}{b \vee c}$$

Veamos el modo en que se puede aplicar dicha regla al cálculo de predicados. Para ello será suficiente, en general, con tener en cuenta las siguientes reglas de inferencia:

$$\text{Regla de particularización:} \qquad \frac{(\forall\ i : R : E_i)}{E_i} \tag{5.1}$$

$$\text{Regla de generalización:} \qquad \frac{E_1, E_2, \ldots, E_n}{E_1 \wedge E_2 \wedge \ldots \wedge E_n} \tag{5.2}$$

La primera de ellas recibe el nombre de *regla de sustitución* en numerosos textos, no así en el presente en el que se ha reservado esa denominación para una regla fundamental del cálculo de proposiciones. Debemos observar, sin embargo, que la regla de resolución, tal y como ha sido enunciada, no permite su aplicación a casos como el dado por las expresiones:

$$(\forall\ X, Z : \neg a(X) \vee b(Z))$$

y

$$(\forall\ W, Y : a(W) \vee c(Y))$$

puesto que no se trata de proposiciones, sino claramente de predicados. Sin embargo, dicha aplicación es posible si antes hemos considerado la adecuada aplicación de la ecuación 5.1, dada en este caso por la sustitución[1] que asigna W a X, y que notaremos:

$$\sigma \equiv \{X \leftarrow W\}$$

con lo que

$$\frac{\neg a(X) \vee b(Z)}{\neg a(W) \vee b(Z)}\ \sigma$$

de donde tendremos que tanto

$$\neg a(W) \vee b(Z)$$

como

$$a(W) \vee c(Y)$$

son tautologías. En este caso, diremos que los términos $a(X)$ y $a(W)$ *unifican* mediante la sustitución σ. Veamos ahora cómo podemos aplicar la regla de resolución. Teniendo en cuenta que el conjunto de valores posibles para las variables Y, Z y W es

[1] el término sustitución constituye aquí un abuso del lenguaje, puesto que realmente se trata de un simple renombramiento de variables.

en la práctica finito[2], podemos aplicar a cada triple de posibles valores la ecuación 5.2 y tendremos que

$$\frac{\neg a(W) \vee b(Z), a(W) \vee c(Y)}{(\neg a(W) \vee b(Z)) \wedge (a(W) \vee c(Y))}$$

y por tanto

$$\frac{(\forall\ W, Z : \neg a(W) \vee b(Z)), (\forall\ W, Y : a(W) \vee c(Y))}{(\forall\ W, Z : \neg a(W) \vee b(Z)) \wedge (\forall\ W, Y : a(W) \vee c(Y))}$$

lo cual, teniendo en cuenta la concatenación de rangos en los cuantificadores, es lo mismo que escribir

$$\frac{(\forall\ W, Z : \neg a(W) \vee b(Z)), (\forall\ W, Y : a(W) \vee c(Y))}{(\forall\ W, Y, Z : \neg a(W) \vee b(Z), a(W) \vee c(Y))}$$

Ahora sí podemos aplicar la regla de resolución en el interior del cuantificador para cada valor de W, Y y Z, con lo que tendremos:

$$\frac{(\forall\ W, Y, Z : \neg a(W) \vee b(Z)), (\forall\ W, Y : a(W) \vee c(Y))}{(\forall\ Y, Z : b(Z) \vee c(Y))}$$

que notacionalmente es lo mismo que escribir

$$\frac{\neg a(W) \vee b(Z), a(W) \vee c(Y)}{b(Z) \vee c(Y)}$$

con lo que resumiendo el proceso aplicado, tendremos que:

$$\frac{\neg a(X) \vee b(Z), a(W) \vee c(Y)}{b(Z) \vee c(Y)}$$

lo que significa que hemos extendido la aplicación de la regla de resolución de las proposiciones al caso de los predicados.

5.2 El proceso de resolución

Con el objeto de no sobrecargar al lector con más definiciones formales, utilizaremos un sencillo ejemplo para ilustrar y guiar la discusión en cada caso.

[2] no hay que olvidar que en la práctica los recursos de un ordenador son finitos, y por tanto el rango también.

5.2.1 Un ejemplo simple

En primer lugar consideraremos las dos expresiones:

$$\begin{array}{lcl} hombre('Adan') & \Leftarrow & \mathcal{T} \\ mortal(Persona) & \Leftarrow & hombre(Persona) \end{array}$$

que denominaremos *cláusulas* y que constituirán nuestro *programa lógico* o lo que es lo mismo, la base de datos que representa nuestro conocimiento del mundo. En cuanto al *significado declarativo* del programa, podemos entender estas dos cláusulas en la forma:

" Adán es un hombre. "
" Todo hombre es mortal. "

En un futuro, utilizaremos la notación

```
hombre('Adan').
mortal(Persona) :-hombre(Persona).
```

para abreviar y facilitar el manejo de las expresiones. De hecho, esta será la representación Prolog del programa considerado, donde los símbolos $\mathcal{T}$ a la derecha de las implicaciones son obviados y donde además el símbolo "$\Leftarrow$" de la implicación lógica ha sido cambiado por ":-". También consideraremos el predicado dado por la expresión:

$$\mathcal{F} \Leftarrow mortal(Individuo)$$

que representaremos en forma abreviada como

```
:-mortal(Individuo).
```

y que constituye la *pregunta* que el usuario realiza al programa. Más exactamente, el significado declarativo de esta pregunta sería:

"¿ Existe algún individuo mortal ?"

Formalmente la inclusión de esta última cláusula en nuestra base de datos significa que estamos suponiendo que

$$\neg(\exists\ Individuo : mortal(Individuo)) = \mathcal{T}$$

o lo que es lo mismo, estamos suponiendo que no existe ningún valor de $Individuo$ para el que

$$mortal(Individuo) = \mathcal{T}$$

Esto implica que nuestra estrategia de demostración es por *reducción al absurdo*. Más exactamente, se trata de demostrar que

$$\neg(\exists\ Individuo : mortal(Individuo)) = \mathcal{F}$$

con lo que

$$(\exists\ Individuo : mortal(Individuo)) = \mathcal{T}$$

lo cual se corresponde con el significado declarativo que habíamos dado a nuestra pregunta. Podemos considerar ahora el conjunto de cláusulas en juego, junto con su interpretación en lógica de predicados:

1. `hombre('Adan').`
   $\mathcal{T} : hombre('Adan')$
2. `mortal(Persona) :- hombre(Persona).`
   $(\forall\ Persona : \mathcal{T} : mortal(Persona) \Leftarrow hombre(Persona))$
   $= (\forall\ Persona : mortal(Persona) \Leftarrow hombre(Persona))$
3. `:- mortal(Individuo).`
   $(\forall\ Individuo : \mathcal{T} : \neg mortal(Individuo))$
   $= (\forall\ Individuo : \neg mortal(Individuo))$
   $= \neg(\exists\ Individuo : mortal(Individuo))$

donde la primera cláusula recibe el nombre de *hecho* por cuanto expresa una verdad sin condiciones.

A partir de aquí, podemos realmente comenzar el *proceso de resolución*, o lo que es lo mismo, el proceso de

demostración de nuestro candidato a teorema[3]. Así, el término `mortal(Individuo)` de la pregunta

```
:- mortal(Individuo).
```

unifica con la *cabeza de la cláusula*[4]

```
mortal(Persona) :- hombre(Persona).
```

mediante la sustitución

$$\sigma_1 \equiv \{\texttt{Individuo} \leftarrow \texttt{Persona}\}$$

Aplicando ahora la regla de resolución obtendremos

$$\frac{\neg mortal(Persona),\ mortal(Persona) \Leftarrow hombre(Persona)}{\neg hombre(Persona)}$$

a cuya conclusión podemos aplicarle de nuevo el mismo razonamiento, unificando de nuevo con la cabeza de la primera cláusula mediante la sustitución

$$\sigma_2 \equiv \{\texttt{Persona} \leftarrow \texttt{'Adan'}\}$$

Ahora, aplicando la regla de resolución de forma análoga al caso de las proposiciones[5], obtenemos

$$\frac{\neg hombre('Adan'),\ hombre('Adan')}{\mathcal{F}}$$

lo cual es una *inconsistencia lógica*, puesto que $\mathcal{F}$ es un valor que nunca puede ser cierto. Ello representa un absurdo, que es consecuencia directa de haber supuesto que

$$\neg(\exists\ Individuo : mortal(Individuo)) = \mathcal{T}$$

por tanto, podemos concluir que

$$(\exists\ Individuo : mortal(Individuo)) = \mathcal{T}$$

[3] lo que a su vez equivale a encontrar una respuesta válida para la pregunta inicial.
[4] esto es, la parte izquierda de la cláusula.
[5] ya no hay variables sin instanciar en los términos.

y este `Individuo` viene dado por la sustitución

$$\texttt{Individuo} \leftarrow \texttt{Persona} \leftarrow \texttt{'Adan'}$$

lo que constituye la *respuesta*[6] a nuestra pregunta, tal como pretendíamos.

El proceso comentado suele representarse abreviadamente en forma arborescente, en lo que habitualmente se denomina *árbol de resolución*. En él se indican las sustituciones utilizadas en las unificaciones que nos han conducido a la obtención de la respuesta final, así como el conjunto de expresiones a las que han sido aplicadas. En este contexto, cada una de las expresiones que constituyen los nodos del árbol se denominan *objetivos* y desde un punto de vista intuitivo representan el conjunto de cuestiones en las que hemos descompuesto el problema representado por la pregunta inicial para su resolución. Cuando no queda ningún objetivo por resolver es que hemos conseguido demostrar nuestro teorema, o lo que es lo mismo responder a la pregunta inicial. En nuestro caso, el árbol de resolución es el representado en la figura 5.1. A cada nodo del árbol de resolución se le denomina *resolvente* y representa el conjunto de objetivos a resolver en el momento de la construcción de dicho nodo.

$$\begin{array}{c} \{\neg mortal(Individuo)\} \\ \mid\ \sigma_1 \\ \{\neg hombre(Persona)\} \\ \mid\ \sigma_2 \\ \{\} \end{array}$$

Figura 5.1: Resolución de `:- mortal(Individuo).`

[6] a veces también se le denomina *contraejemplo*, puesto que efectivamente constituye un contraejemplo a nuestra falsa suposición inicial.

5.2.2 Un ejemplo completo

Una vez introducida, al menos informalmente, la técnica de resolución que representa el motor de Prolog, el siguiente paso consistirá en mostrar un ejemplo más completo. El problema elegido es el de la definición del tipo lista, tal y como se considera en los lenguajes clásicos de programación. En este caso, consideraremos que una lista puede representarse de dos formas distintas:

- Mediante la simple explicitación de todos sus elementos separados por comas y encerrados entre corchetes. Así, por ejemplo la lista cuyos elementos son 1 y 2 se representaría en la forma
$$[1, 2]$$
- Mediante la diferenciación entre los primeros elementos de la lista y el resto de los elementos. En este caso la lista
$$[1, 2]$$
también se representaría en la forma
$$[1 \mid [2]]$$
o también
$$[1, 2 \mid [\,]]$$
donde | es el operador que separa los n primeros elementos de la lista, del resto.

Ejemplo 5.2.1 *La definición del tipo lista en programación lógica se corresponde con las dos cláusulas que siguen:*

```
lista([ ]).
lista ([Car|Cdr]) :- lista(Cdr).
```

que declarativamente pueden ser leídas en la forma:

> "La lista vacía es una lista."
> "Una estructura cuyos elementos están expresados entre corchetes es una lista si contiene un primer elemento `Car` y lo que queda define también una lista."

Si ahora interrogamos al programa con la pregunta

```
:- lista([1,2]).
```

formalmente estamos afirmando que

$$\neg lista([1,2]) = \mathcal{T}$$

Veremos que esta suposición nos lleva a un absurdo, con lo que podremos concluir que

$$lista([1,2]) = \mathcal{T}$$

El proceso de resolución sería el siguiente:

1. *El objetivo* $lista([1,2])$ *unifica con la cabeza de la segunda cláusula mediante la sustitución*

$$\sigma_1 \equiv \{Car_1 \leftarrow 1, Cdr_1 \leftarrow [2]\}$$

2. *Aplicando ahora la regla de resolución, obtendremos*

$$\frac{\neg lista([1,2]), lista([1 \mid [2]]) \Leftarrow lista([2])}{\neg lista([2])}$$

3. *De nuevo, unificaremos el objetivo*

$$lista([2])$$

con la cabeza de la segunda cláusula mediante la sustitución

$$\sigma_2 \equiv \{Car_2 \leftarrow 2, Cdr_2 \leftarrow [\,]\}$$

con lo que podemos aplicar de nuevo la regla de resolución:

$$\frac{\neg lista([2]), lista([2 \mid [\,]) \Leftarrow lista([\,])}{\neg lista([\,])}$$

4. *Ahora unificamos con la primera cláusula y aplicando de nuevo la regla de resolución, tendremos*

$$\frac{\neg lista([\,]), lista([\,])}{\mathcal{F}}$$

con lo que hemos llegado a una inconsistencia lógica, cuyo origen no es otro que el de haber supuesto

$$\neg lista([1,2]) = \mathcal{T}$$

por lo tanto tendremos que

$$lista([1,2]) = \mathcal{T}$$

siendo en este caso $\mathcal{T}$ la respuesta obtenida a nuestra pregunta.

□

El ejemplo siguiente muestra la complejidad del proceso de resolución a la vez que la monotonía de la técnica aplicada en su puesta en práctica. Ambos factores justifican el interés de una implementación informática para el mismo.

Ejemplo 5.2.2 *El problema abordado ahora es el de la inversión, al primer nivel, de estructuras de tipo lista, tal como han sido definidas con anterioridad. La idea es simple, dada una lista*

$$[1,2]$$

el resultado de la inversión será

$$[2,1]$$

En el caso de listas con más de un nivel de profundidad, tales como

$$[1,[2,3],4]$$

el efecto de la inversión sólo se reflejará en el primero de ellos, de modo que obtendríamos

$$[4,[2,3],1]$$

Un posible conjunto de cláusulas capaz de implementar esta funcionalidad viene dado por

```
concatenar([], Lista, Lista).
concatenar([Car|Cdr], Lista, [Car|Resultado]) :-
                    concatenar(Cdr, Lista, Resultado).
invertir([ ], [ ]).
invertir([Car|Cdr], Invertir) :-
                    invertir(Cdr, Invertir_Cdr),
                    concatenar(Invertir_Cdr,[Car],Invertir).
```

donde el predicado `concatenar` *representa la implementación del concepto de concatenación de listas y el predicado* `invertir` *el de la inversión propiamente dicha. Más exactamente consideraremos que*

`concatenar(Lista_1, Lista_2, Resultado)`

es cierto si `Resultado` *es el resultado de concatenar las listas* `Lista_1` *y* `Lista_2`*. De forma análoga, supondremos que*

`invertir(Lista, Invertir)`

es cierto si `Invertir` *es el resultado de la inversión, al primer nivel, de la lista* `Lista`*. Declarativamente, el significado de nuestro programa sería el siguiente, interpretando cláusula a cláusula:*

> "Si concatenamos la lista vacía con otra lista, esta última es el resultado."
> "El resultado de concatenar una lista, cuyo primer elemento viene dado por `Car` y el resto de los elementos por `Cdr`, con otra lista `Lista` es igual a concatenar `Cdr` con `Lista` y al resultado así obtenido anteponerle `Car`."

> "El resultado de invertir la lista vacía es ella misma."
> "El resultado de invertir una lista de primer elemento `Car` es igual a invertir el resto de sus elementos `Cdr`, y al resultado así obtenido concatenarle la lista formada por el primer elemento `Car` de la lista original."

Supongamos ahora que planteamos la pregunta

```
:- invertir([1,2], Invertir).
```

Esto es, estamos interrogando al programa sobre el resultado del proceso de inversión de la lista `[1, 2]`*. Primero comenzaremos por traducir el conjunto de cláusulas a su expresión exacta como predicados. Así, tenemos que:*

```
invertir([ ],[ ]). = T
```

```
invertir([Car|Cdr],Invertir) :-
            invertir(Cdr, Invertir_Cdr),
            concatenar(Invertir_Cdr, [Car], Invertir).
```

$$= (\forall Car, Cdr, Invertir, Invertir\_Cdr : invertir(Cdr, Invertir\_Cdr) \wedge concatenar(Invertir\_Cdr, [Car], Invertir) \Rightarrow invertir([Car \mid Cdr], Invertir))$$

```
:- invertir([1,2],Invertir).
```

$$= (\forall Invertir : \neg invertir([1,2], Invertir))$$
$$= \neg(\exists Invertir : invertir([1,2], Invertir))$$

De forma totalmente análoga podríamos traducir las cláusulas del predicado `concatenar`*. Se trata pues de demostrar que*

$$\neg(\exists Invertir : invertir([1,2], Invertir)) = \mathcal{F}$$

lo que es equivalente a probar que

$$(\exists Invertir : invertir([1,2], Invertir)) = \mathcal{T}$$

Si consideramos la sustitución[7]

$$\sigma_1 \equiv \{ \; Car \leftarrow 1, Cdr \leftarrow [2], Invertir \leftarrow Invertir_1, \; Invertir\_Cdr \leftarrow Invertir\_Cdr_1\}$$

y aplicándola sobre la segunda cláusula, tendremos que por aplicación de la regla de particularización:

[7] aquí $Invertir\_Cdr_1$ es el resultado de renombrar el $Invertir\_Cdr$ en la segunda cláusula del predicado *invertir* e $Invertir_1$ es el resultado de renombrar el $Invertir$ de la pregunta.

$$\frac{\begin{array}{l}(\forall Car, Cdr, Invertir, Invertir\_Cdr : \\ \quad invertir([Car \mid Cdr], Invertir) \Leftarrow \\ \quad\quad invertir(Cdr, Invertir\_Cdr) \\ \quad\quad \wedge\ concatenar(Invertir\_Cdr, [Car], Invertir))\end{array}}{\begin{array}{l}(\forall Invertir_1, Invertir\_Cdr_1 : \\ \quad invertir([1 \mid [2]], Invertir_1) \Leftarrow \\ \quad\quad invertir([2], Invertir\_Cdr_1) \\ \quad\quad \wedge\ concatenar(Invertir\_Cdr_1, [1], Invertir_1))\end{array}}\ \sigma_1$$

Ahora, aplicando la regla de generalización, obtenemos:

$$\frac{\begin{array}{l}(\forall Invertir_1, Invertir\_Cdr_1 : \\ \quad invertir([1 \mid [2]], Invertir_1) \Leftarrow \\ \quad\quad invertir([2], Invertir\_Cdr_1) \\ \quad\quad \wedge\ concatenar(Invertir\_Cdr_1, [1], Invertir_1)), \\ (\forall Invertir_1 : \neg invertir([1 \mid [2]], Invertir_1))\end{array}}{\begin{array}{l}(\forall Invertir_1, Invertir\_Cdr_1 : \\ \quad invertir([1 \mid [2]], Invertir_1) \Leftarrow \\ \quad\quad invertir([2], Invertir\_Cdr_1) \\ \quad\quad \wedge\ concatenar(Invertir\_Cdr_1, [1], Invertir_1)) \\ \wedge\ (\forall Invertir_1 : \neg invertir([1 \mid [2]], Invertir_1))\end{array}}$$

Concatenando los rangos de los cuantificadores:

$$\begin{array}{l}(\forall Invertir_1, Invertir\_Cdr_1 : \\ \quad invertir([1 \mid [2]], Invertir_1) \Leftarrow \\ \quad\quad invertir([2], Invertir\_Cdr_1) \\ \quad\quad \wedge\ concatenar(Invertir\_Cdr_1, [1], Invertir_1) \\ \quad \wedge\ \neg invertir([1 \mid [2]], Invertir_1))\end{array}$$

y aplicando la regla de resolución a nivel de los predicados simples, tendremos que finalmente

$$\frac{\begin{array}{l}(\forall Invertir_1, Invertir\_Cdr_1 : \\ \quad invertir([1 \mid [2]], Invertir_1) \Leftarrow \\ \quad\quad invertir([2], Invertir\_Cdr_1) \\ \quad\quad \wedge\ concatenar(Invertir\_Cdr_1, [1], Invertir_1)), \\ (\forall Invertir_1 : \neg invertir([1 \mid [2]], Invertir_1))\end{array}}{\begin{array}{l}(\forall Invertir_1, Invertir\_Cdr_1 : \\ \quad \neg(invertir([2], Invertir\_Cdr_1) \\ \quad\quad \wedge\ concatenar(Invertir\_Cdr_1, [1], Invertir_1)))\end{array}}$$

Esto es, la nueva resolvente es:

$$\begin{array}{l}\neg(\exists Invertir_1, Invertir\_Cdr_1 : \\ \quad invertir([2], Invertir\_Cdr_1) \\ \quad \vee\ concatenar(Invertir\_Cdr_1, [1], Invertir_1))\end{array}$$

donde, considerando la sustitución[8]

$$\begin{array}{l}\sigma_2 \equiv \{\ \ Car \leftarrow 2, Cdr \leftarrow [\,], Invertir \leftarrow Invertir\_Cdr_1, \\ \qquad Invertir\_Cdr \leftarrow Invertir\_Cdr_2\}\end{array}$$

y aplicándola a la segunda cláusula, tendremos que por aplicación de la regla de particularización

$$\frac{\begin{array}{l}(\forall Car, Cdr, Invertir, Invertir\_Cdr : \\ \quad invertir([Car \mid Cdr], Invertir) \Leftarrow \\ \quad\ invertir(Cdr, Invertir\_Cdr) \\ \quad\ \wedge\ concatenar(Invertir\_Cdr, [Car], Invertir))\end{array}}{\begin{array}{l}(\forall Invertir\_Cdr_1, Invertir\_Cdr_2 : \\ \quad invertir([2 \mid [\,]], Invertir\_Cdr_1) \Leftarrow \\ \quad\ invertir([\,], Invertir\_Cdr_2) \\ \quad\ \wedge\ concatenar(Invertir\_Cdr_2, [2], Invertir\_Cdr_1))\end{array}}\ \sigma_2$$

Aplicando de nuevo la regla de particularización tendremos que:

$$\frac{\begin{array}{l}(\forall Invertir\_Cdr_1, Invertir\_Cdr_2 : \\ \quad invertir([2 \mid [\,]], Invertir\_Cdr_1) \Leftarrow \\ \quad\ invertir([\,], Invertir\_Cdr_2) \\ \quad\ \wedge\ concatenar(Invertir\_Cdr_2, [2], Invertir\_Cdr_1)), \\ (\forall Invertir_1, Invertir\_Cdr_1 : \\ \quad \neg(invertir([2 \mid [\,]], Invertir\_Cdr_1) \\ \quad\ \wedge\ concatenar(Invertir\_Cdr_1, [1], Invertir_1)))\end{array}}{\begin{array}{l}(\forall Invertir\_Cdr_1, Invertir_1, Invertir\_Cdr_2 : \\ \quad \neg(invertir([\,], Invertir\_Cdr_2) \\ \quad\ \wedge\ concatenar(Invertir\_Cdr_2, [2], Invertir\_Cdr_1)) \\ \quad \vee\ \neg concatenar(Invertir\_Cdr_1, [1], Invertir_1))\end{array}}$$

[8] aquí $Invertir\_Cdr_2$ es el resultado de renombrar el $Invertir\_Cdr$ en la segunda cláusula del predicado *invertir*.

Lo que es equivalente a

$$
\begin{array}{l}
(\forall Invertir\_Cdr_1, Invertir_1, Invertir\_Cdr_2 : \\
\quad \neg invertir([\,], Invertir\_Cdr_2) \\
\quad \vee \neg concatenar(Invertir\_Cdr_2, [2], Invertir\_Cdr_1) \\
\quad \vee \neg concatenar(Invertir\_Cdr_1, [1], Invertir_1))
\end{array}
$$

que es la nueva resolvente. Si aplicamos ahora a la conclusión obtenida la sustitución

$$\sigma_3 \equiv \{Invertir\_Cdr_2 \leftarrow [\,]\}$$

y aplicando la regla de particularización tendremos que

$$
\frac{\begin{array}{l}
(\forall Invertir\_Cdr_1, Invertir_1, Invertir\_Cdr_2 : \\
\quad \neg invertir([\,], Invertir\_Cdr_2) \\
\quad \vee \neg concatenar(Invertir\_Cdr_2, [2], Invertir\_Cdr_1) \\
\quad \vee \neg concatenar(Invertir\_Cdr_1, [1], Invertir_1))
\end{array}}{\begin{array}{l}
(\forall Invertir\_Cdr_1, Invertir_1 : \\
\quad \neg invertir([\,], [\,]) \\
\quad \vee \neg concatenar([\,], [2], Invertir\_Cdr_1) \\
\quad \vee \neg concatenar(Invertir\_Cdr_1, [1], Invertir_1))
\end{array}}\sigma_3
$$

Al considerar la aplicación de la regla de generalización a dicha conclusión en la primera cláusula del predicado invertir*, tendremos:*

$$
\frac{\begin{array}{l}
(\forall Invertir\_Cdr_1, Invertir_1 : \\
\quad \neg invertir([\,], [\,]) \\
\quad \vee \neg concatenar([\,], [2], Invertir\_Cdr_1) \\
\quad \vee \neg concatenar(Invertir\_Cdr_1, [1], Invertir_1)), \\
invertir([\,], [\,])
\end{array}}{\begin{array}{l}
(\forall Invertir\_Cdr_1, Invertir_1 : \\
\quad \neg invertir([\,], [\,]) \\
\quad \vee \neg concatenar([\,], [2], Invertir\_Cdr_1) \\
\quad \vee \neg concatenar(Invertir\_Cdr_1, [1], Invertir_1)) \\
\wedge\ invertir([\,], [\,])
\end{array}}
$$

y al concatenar dominios

$$
\begin{array}{l}
(\forall Invertir\_Cdr_1, Invertir_1 : \\
\quad (\neg invertir([\,], [\,]) \\
\quad\ \ \vee \neg concatenar([\,], [2], Invertir\_Cdr_1) \\
\quad\ \ \vee \neg concatenar(Invertir\_Cdr_1, [1], Invertir_1)) \\
\quad \wedge\ invertir([\,], [\,]))
\end{array}
$$

Teniendo ahora en cuenta la regla de resolución en el interior del cuantificador universal, tendremos que finalmente:

$$\frac{\begin{array}{l}(\forall Invertir\_Cdr_1, Invertir_1, Invertir\_Cdr_2 : \\ \quad \neg invertir([\,], Invertir\_Cdr_2) \\ \quad \vee \neg concatenar(Invertir\_Cdr_2, [2], Invertir\_Cdr_1) \\ \quad \vee \neg concatenar(Invertir\_Cdr_1, [1], Invertir_1))\end{array}}{\begin{array}{l}(\forall Invertir\_Cdr_1, Invertir_1 : \\ \quad \neg concatenar([\,], [2], Invertir\_Cdr_1) \\ \quad \vee \neg concatenar(Invertir\_Cdr_1, [1], Invertir_1))\end{array}}$$

El proceso continúa ahora de forma similar para `concatenar`*, siendo los valores obtenidos para las variables en juego*

$$Invertir\_Cdr_1 \leftarrow [2]$$

y

$$Invertir_1 \leftarrow [2, 1]$$

con lo que nuestra respuesta final será

$$Invertir \leftarrow [2, 1]$$

Evidentemente podemos generar el árbol de resolución correspondiente, que es el indicado en la figura 5.2, donde

$$\sigma = \{Invertir\_Cdr_1 \leftarrow [2], Invertir_1 \leftarrow [2, 1]\}$$

□

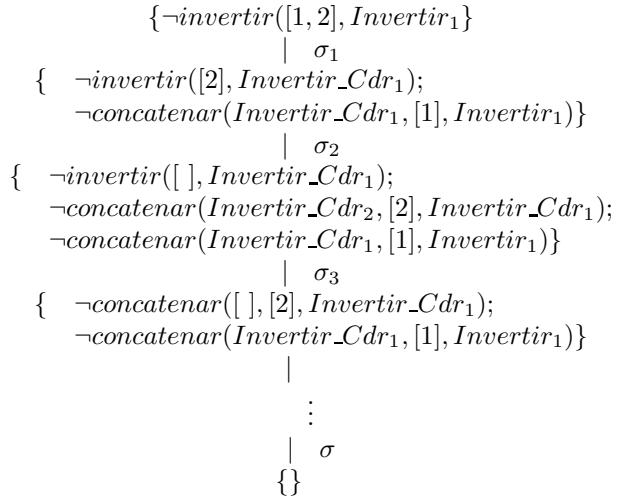


Figura 5.2: Resolución de `:- invertir([1,2], Invertir).`

Parte III

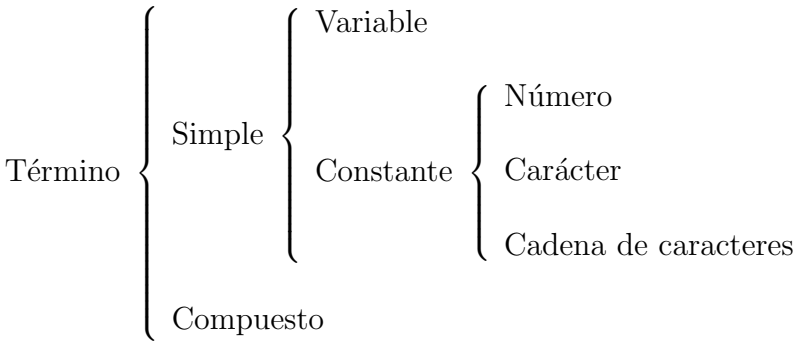
El Intérprete Lógico

Capítulo 6

Conceptos Fundamentales

6.1 Objetos en programación lógica

La clase más general de objeto manejado en programación lógica es el *término*. La jerarquía de datos a partir de dicha estructura viene indicada por el siguiente diagrama:



Las *variables* suelen indicarse mediante un identificador cuyo primer carácter es una letra mayúscula. Los *términos compuestos* son los objetos estructurados del lenguaje. Se componen de un *funtor*[1] y una secuencia de uno o más términos llamados *argumentos*. Un funtor se caracteriza por su *nombre*, que es un átomo, y su *aridad* o número de argumentos. Gráficamente, el objeto puede representarse en

[1] llamado el *funtor principal del término*.

este caso mediante un árbol en el que la raíz está etiquetada con el nombre del funtor y los hijos con los argumentos del mismo. Cuando el término compuesto no expresa ningún tipo de relación lógica[2], suele dársele el nombre de *función*. Una *constante lógica* es una función de aridad cero.

Ejemplo 6.1.1 *Podemos considerar el caso de un objeto estructurado* `fecha` *con tres argumentos:* `Año`, `Mes` *y* `Día`. *Su representación será la siguiente:*

$$fecha(Año, Mes, Día)$$

En este caso, el funtor puede caracterizarse usando la notación `fecha/3` *en referencia a su nombre y a la aridad considerada. Dicha representación se suele conocer como la* firma del funtor. *La representación en forma de árbol sería la indicada en la figura 6.1.* □

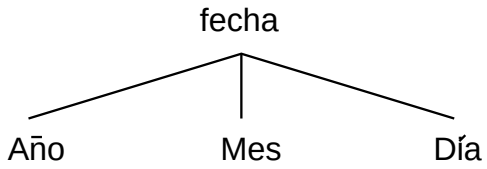


Figura 6.1: Representación arborescente del término *fecha(Año, Mes, Día)*.

Intuitivamente, el lector puede pensar en los términos compuestos de la programación lógica como si de un *tipo registro* en los lenguajes clásicos de programación se tratara. De hecho, el principio es el mismo con la salvedad de que aquí la recuperación y asignación de lo que serían los campos del registro no se realiza mediante funciones de acceso, sino utilizando el concepto de *unificación* que introduciremos más adelante.

[2]ello dependerá tan solo del contexto expresado por el programa lógico.

6.2 Programas lógicos, cláusulas y preguntas

La unidad fundamental de un programa lógico es el *átomo*, el cual es un tipo especial de término compuesto, distinguido sólo por el contexto en el cual aparece en el programa[3]. Su estructura es en principio la misma que la de un término compuesto, pero en este caso el funtor principal recibe el nombre de *predicado*. Su equivalente en la programación clásica sería el de una llamada a un proceso. A partir del concepto de átomo, definimos el de *literal*, que no es otra cosa que un átomo o su negación.

En este punto, podemos ya definir la noción de *cláusula* como un conjunto de literales ligados por conectores y cuantificadores lógicos. En nuestro caso, nos limitaremos a un tipo especial de cláusulas denominadas de Horn. Una *cláusula de Horn* es una estructura compuesta por una *cabeza* y un *cuerpo*. La cabeza consiste o bien de un simple átomo, o bien está vacía. El cuerpo está formado por una secuencia de cero o más átomos. Cuando la cabeza está vacía se dice que la cláusula es una *pregunta*. La cabeza y el cuerpo de una misma cláusula están separados por un símbolo de implicación lógica cuya representación notacional puede variar según el autor considerado. En cuanto a los átomos del cuerpo, estos se separan por operadores lógicos conjuntivos[4], soliendo indicarse el final del cuerpo mediante un punto. Esto es, en general representaremos una cláusula en la forma :

$$P \ :- \ Q_1, Q_2, \ldots, Q_n.$$

que podemos leer bien *declarativamente*:

> *"P es cierto si Q_1 es cierto, Q_2 es cierto, ..., y Q_n es cierto."*

bien *operacionalmente*:

> *"Para satisfacer P, es necesario satisfacer los átomos Q_1, Q_2, ..., y Q_n."*

[3] esto es, expresa una relación lógica.
[4] y en determinados casos disyuntivos.

En este contexto, un *programa lógico* se define simplemente como una secuencia de cláusulas.

Ejemplo 6.2.1 *El siguiente programa define recursivamente el conjunto de los números naturales:*

```
numero_natural(0).
numero_natural(siguiente(Numero)) :- numero_natural(Numero).
```

Se puede observar que hemos utilizado una función, de nombre `siguiente`*, que no expresa ningún tipo de relación lógica. De hecho su utilidad es meramente notacional, en contra de lo que ocurre con el predicado* `numero_natural`*. Además el argumento de la regla recursiva es una variable, puesto que se trata de una regla que debe expresar de forma general la naturaleza de un número natural. Declarativamente, podemos leer ambas cláusulas en la forma:*

> "El cero es un número natural."
> "El número `siguiente(Numero)` es natural, si `Numero` también lo es."

Evidentemente, la notación considerada en este caso para los números naturales no es la habitual, sino la dada por la tabla 6.1. □

Natural	**Notación**
0	`0`
1	`siguiente(0)`
2	`siguiente(siguiente(0))`
⋮	⋮
n	`siguiente(`$\overset{n}{\ldots}$`(siguiente(0))...)`

Tabla 6.1: Notación para la representación de los números naturales.

6.3 Concepto de unificación: sustituciones e instancias

Dos conceptos, *sustitución* e *instanciación* son, junto con el de *resolución*, nociones básicas del motor de un intérprete lógico y determinan inequívocamente el estilo de programación propio de los lenguajes de este tipo.

Definición 6.3.1 *Una* sustitución *es una lista de pares* (variable, término)*. Utilizaremos la notación*

$$\Theta \equiv \{X_1 \leftarrow T_1, \ldots, X_n \leftarrow T_n\}$$

para representar la sustitución Θ *que asocia las variables* X_i *a los términos*

$$T_i,\ i \in \{1, 2, \ldots, n\}$$

La aplicación de una sustitución Θ *a un término lógico* T *será denotada* $T\Theta$ *y se dirá que es una* instancia *del término* T*.* □

Ahora podemos considerar una definición formal recursiva de la *semántica declarativa* de las cláusulas, que sirve para indicarnos aquellos objetivos que pueden ser considerados ciertos en relación a un programa lógico:

> *"Un objetivo es cierto si es una instancia de la cabeza de alguna de las cláusulas del programa lógico considerado y cada uno de los objetivos que forman el cuerpo de la cláusula instanciada son a su vez ciertos."*

En este punto, es importante advertir que la semántica declarativa no hace referencia al orden explícito de los objetivos dentro del cuerpo de una cláusula, ni al orden de las cláusulas dentro de lo que será el programa lógico. Este orden es, sin embargo, fundamental para la semántica operacional de Prolog. Ello es la causa fundamental de las divergencias entre ambas semánticas en dicho lenguaje y fuente de numerosos errores de programación, que además no siempre son fáciles de detectar.

Las sustituciones son utilizadas en programación lógica para, mediante su aplicación a las variables contenidas en una cláusula, obtener la expresión de la veracidad de una relación lógica particular a partir de la veracidad de una relación lógica más general incluida en el programa. Más formalmente, dicho concepto se conoce con el nombre de *unificación*, que pasamos a definir inmediatamente.

Definición 6.3.2 *Un* unificador *de dos términos lógicos T_1 y T_2 es una sustitución Θ, tal que $T_1\Theta = T_2\Theta$. Cuando al menos existe un unificador para dos términos lógicos T_1 y T_2, existe un unificador particular Θ llamado el* unificador más general (mgu)[5] de T_1 y T_2*, tal que para cualquier otro unificador Θ', existe una sustitución σ tal que $\Theta' = \Theta\sigma$.* □

Intuitivamente, el $mgu(T_1, T_2)$ representa el número mínimo de restricciones a considerar sobre dos términos para hacerlos iguales.

Ejemplo 6.3.1 *Dados los términos lógicos:*

$$T_1 = p(A, 3, f(Y))$$
$$T_2 = p(B, C, f(g(Z)))$$

un conjunto de posibles unificadores es el siguiente:

$$\Theta_1 \equiv \{A \leftarrow 5, B \leftarrow 5, C \leftarrow 3, Y \leftarrow g(Z)\}$$
$$\Theta_2 \equiv \{B \leftarrow A, C \leftarrow 3, Y \leftarrow g(0), Z \leftarrow 0\}$$
$$\Theta_3 \equiv \{A \leftarrow B, C \leftarrow 3, Y \leftarrow g(Z)\}$$

donde el $mgu(T_1, T_2)$ es Θ_3. □

Desde un punto de vista práctico, el mgu nos permitirá efectuar nuestro razonamiento lógico conservando la mayor generalidad posible en nuestras conclusiones. Es por ello que el mgu es el unificador utilizado en el proceso de demostración que constituye un intérprete lógico. Para su obtención, la mayoría de los dialectos Prolog actuales consideran el algoritmo de Robinson [22], que pasamos a describir inmediatamente.

[5] de **m**ost **g**eneral **u**nificator.

Algoritmo 6.3.1 *El siguiente seudocódigo describe el* método de unificación de Robinson.

Entrada: *Dos términos lógicos* T_1 *y* T_2.

Salida: *El* $mgu(T_1, T_2)$, *si existe; en otro caso* `fail`.

Algoritmo:
inicio
$\Theta := \emptyset$;
`push` $(T_1 = T_2$, `Pila)` ;
mientras `Pila` $\neq \emptyset$ **hacer**
  $(X = Y)$ `:= pop(Pila)` ;
  caso
    $X \notin Y$: `sustituir` (X, Y);
      `añadir` $(\Theta, X \leftarrow Y)$
    $Y \notin X$: `sustituir` (Y, X);
      `añadir` $(\Theta, Y \leftarrow X)$
    $(X \leftarrow Y)$: **nada**
    $(X \leftarrow f(X_1, \ldots, X_n))$ **y** $(Y \leftarrow f(Y_1, \ldots, Y_n))$:
      para i `:=` n **hasta** *1* **paso** *-1* **hacer**
        `push(`$X_i = Y_i$`,` `Pila)`
      fin para
    sino : **devolver** `fail`
  fin caso
fin mientras ;
devolver Θ
fin

donde la función `push(`$T_1 = T_2$`,` `Pila)` *introduce la ecuación lógica* $T_1 = T_2$ *y la función* `pop(Pila)` *extrae la ecuación lógica* $X = Y$ *de la estrucura LIFO*[6] `Pila`. *La función* `sustituir`(X, Y) *sustituye la variable* X *por* Y *en la pila y en la sustitución* Θ. *Finalmente, la función* `añadir`(Θ, σ) *añade al unificador* Θ *la sustitución* σ. □

[6] **L**ast **I**nput **F**irst **O**utput.

Ejemplo 6.3.2 *El siguiente ejemplo describe la unificación de los términos:*

$$T_1 = f(X, g(X, h(Y)))$$
$$T_2 = f(Z, g(Z, Z))$$

Como aplicación directa del algoritmo 6.3.1 tenemos la siguiente secuencia de configuraciones de la pila usada para la unificación:

$$\Theta \equiv \{\} \quad \boxed{T_1 = T_2} \quad \vdash \quad \Theta \equiv \{\} \quad \boxed{\begin{array}{c} X = Z \\ \hline g(Z,Z) = g(X,h(Y)) \end{array}} \quad \vdash$$

$$\vdash \quad \Theta \equiv \{X \leftarrow Z\} \quad \boxed{g(Z,Z) = g(Z,h(Y))} \quad \vdash \quad \Theta \equiv \{X \leftarrow Z\} \quad \boxed{\begin{array}{c} Z = Z \\ \hline Z = h(Y) \end{array}} \quad \vdash \quad \Theta \equiv \{X \leftarrow Z\} \quad \boxed{Z = h(Y)}$$

Esto es, el resultado final es el dado por la sustitución

$$\Theta \equiv \{X \leftarrow Z \leftarrow h(Y), Z \leftarrow h(Y)\}$$

□

Un aspecto importante a señalar en relación al algoritmo de unificación es el comúnmente llamado *test de ciclicidad* y que en el algoritmo 6.3.1 se representa mediante las expresiones $X \notin Y$ y $Y \notin X$. En efecto, en la práctica y dado el elevado costo que representaría la aplicación de este test, la mayoría de los intérpretes Prolog lo eliminan. Como consecuencia, el programador puede incurrir en errores de difícil localización, tal y como demuestra el siguiente ejemplo.

Ejemplo 6.3.3 *Consideremos el siguiente programa lógico, que implementa el concepto de igualdad entre dos términos:*

```
igual(X,X).
```

cuya semántica declarativa viene dada por

> "Dos términos son iguales si unifican."

Interrogamos ahora a nuestro programa con la pregunta

```
:- igual(Y,f(Y)).
```

Veremos que nuestro algoritmo de unificación, sin test de ciclicidad, entra en un ciclo sin fin. En efecto, la secuencia de configuraciones en la pila que sirve de sustento al algoritmo, es la que sigue:

$$\underset{\Theta \equiv \{\}}{\boxed{igual(X,X) = igual(Y,f(Y))}} \vdash \underset{\Theta \equiv \{\}}{\begin{array}{|c|}\hline X = Y \\ \hline X = f(Y) \\ \hline \end{array}} \vdash \underset{\Theta \equiv \{X \leftarrow Y\}}{\boxed{Y = f(Y)}}$$

Observemos que en la última de las configuraciones Y aparece en f(Y). *Si continuamos con el proceso ocurrirá que sustituiremos toda ocurrencia de* Y *por* f(Y), *y como consecuencia obtendremos:*

$$\begin{aligned} X \quad & \leftarrow Y \leftarrow f(Y) \leftarrow f(f(Y)) \leftarrow f(f(f(Y))) \\ & \leftarrow f(f(f(f(Y)))) \leftarrow \ldots \end{aligned}$$

Esto es, la unificación ha entrado en un ciclo. □

6.4 Un intérprete lógico simple

La mayoría de los intérpretes Prolog actuales están basados en un método de resolución lógica denominado SLD[7] [8] y que pasamos a describir ahora.

Algoritmo 6.4.1 *El siguiente seudocódigo describe el* método de resolución SLD.

Entrada: *Un programa lógico P y una pregunta Q.*

Salida: *$Q\Theta$, si es una instancia de Q deducible a partir de P; en otro caso* `fail`.

[7] por **S**electing a literal, using a **L**inear strategy and searching the space of possible deductions **D**epth-first.

Algoritmo:

inicio
`Resolvente` := $\{Q\}$;
mientras `Resolvente` $\neq \emptyset$ **hacer**
  $A \in$ `Resolvente` ;
  si $\exists\, P : -Q_1, \ldots, Q_n$ *tal que* $\exists\, \Theta = mgu(A, P)$ **entonces**
    `borrar (Resolvente,` A`)` ;
    `añadir (Resolvente,` $Q_1\Theta, \ldots, Q_n\Theta$`)` ;
    `aplicar (`Θ`, Resolvente)`
  sino
    devolver `fail`
  fin si
fin mientras ;
devolver Θ
fin

donde la función `borrar(Resolvente, A)` *borra el objetivo* `A` *de* `Resolvente`*, mientras que la función* `añadir(Resolvente,` $Q_1\Theta, \ldots, Q_2\Theta$`)` *añade los objetivos indicados a* `Resolvente`*. La función* `aplicar(`Θ`, Resolvente)` *aplica sobre el conjunto de objetivos de* `Resolvente` *la restricción representada por la sustitución* Θ*. En general consideraremos que una* resolvente *contiene en cada instante el conjunto de objetivos a resolver.* □

De hecho, existe un formalismo arborescente para representar las derivaciones efectuadas por nuestro algoritmo 6.4.1. En dicho árbol, la existencia de una rama cuya última hoja es la cláusula vacía se traduce en la existencia de una instancia que es consecuencia lógica del programa, esto es, de una respuesta. En otro caso, no existe ninguna instancia que sea consecuencia lógica del programa.

El árbol en cuestión se denomina *árbol de resolución*. En él, cada uno de los nodos representa el estado actual de la resolvente, mientras que las ramas están etiquetadas con los valores de las sustituciones aplicadas en el algoritmo 6.4.1.

Definición 6.4.1 *Sean P un programa lógico y Q una pregunta para el mismo, entonces el* árbol de resolución *para el programa P, dada la pregunta Q, se construye en la forma siguiente:*

1. *La raíz del árbol está constituida por la pregunta Q, que es el primer valor de la resolvente.*
2. *Una vez seleccionado un átomo A en la resolvente, que llamaremos el* objetivo a resolver*, la construcción del árbol en cada nodo continúa como sigue:*
   (a) *Si el átomo seleccionado A se puede unificar con la cabeza de cada una de las cláusulas*

$$\begin{aligned} &P_1 : -Q_1^1, \ldots, Q_1^{m_1}. \\ &P_2 : -Q_2^1, \ldots, Q_2^{m_2}. \\ &\vdots \\ &P_n : -Q_n^1, \ldots, Q_n^{m_n}. \end{aligned}$$

   mediante las sustituciones $\{\Theta_i$, i=1,2,...,n$\}$, entonces construimos n ramas para el nodo considerado. En cada una de esas ramas escribimos la nueva resolvente derivada de la cláusula C_i y de la sustitución Θ_i, renombrando[8] *automáticamente las variables de los nuevos objetivos incorporados a la resolvente. Los átomos de la cola de la cláusula unificada se añaden a la resolvente, mientras que el objetivo a resolver es eliminado de la misma.*
   (b) *Si el átomo seleccionado no se puede unificar con la cabeza de ninguna de las cláusulas de P, se paraliza la construcción de la rama del árbol considerada y se devuelve* `fail` *como respuesta.*
   (c) *Si la resolvente resultante está vacía es porque no queda ningún objetivo por resolver. En este caso, también se*

[8] dicho renombramiento puede efectuarse simplemente mediante la incorporación de un subíndice a los nombres de las variables. Es importante indicar que esta técnica es necesaria para indicar qué variables de igual nombre en cláusulas distintas son diferentes, evitando de este modo confusiones en las futuras sustituciones.

paraliza la construcción de dicha rama y se devuelve `true`*, además del resultado de la composición de las sustituciones* $\tilde{\Theta}_1\tilde{\Theta}_2 \ldots \tilde{\Theta}_l$ *que han sido consideradas desde la raíz hasta el nodo en cuestión. Ello constituye la respuesta a la pregunta* Q*.*

□

En este punto, es importante señalar que en la práctica la construcción del árbol de resolución en un intérprete Prolog impone un orden fijo en la elección tanto del objetivo a resolver dentro de la resolvente en un momento dado, como de la cláusula que será aplicada en primer lugar. Ello nos obliga a redefinir el concepto de árbol de resolución, tal y como es considerado en la práctica por los intérpretes Prolog. Esta carácterística marca la diferencia fundamental entre lo que entendemos por intérprete lógico basado en la resolución SLD e intérprete Prolog.

Definición 6.4.2 *Sean* P *un programa lógico y* Q *una pregunta para el mismo, entonces el* árbol de resolución Prolog *para el programa* P*, dada la pregunta* Q*, se construye de la misma forma que en la anterior definición 6.4.1, salvo que:*

1. *El objetivo a resolver es siempre el primer átomo* A *de la resolvente* Q*, si consideramos esta como una lista ordenada de izquierda a derecha.*

2. *En el proceso de unificación del objetivo a resolver, con las cabezas de las cláusulas del programa, se impondrá un orden de exploración de estas que será de arriba hacia abajo.*

3. *Al añadir a la resolvente los átomos de la cola de la cláusula unificada con el objetivo a resolver, estos sustituyen la posición que tenía en la resolvente el objetivo resuelto, a la vez que conservan el orden local que mantenían en la cola de la cláusula.*

□

Los dos conceptos introducidos en las definiciones 6.4.1 y 6.4.2 están muy próximos. Sin embargo, los detalles que las distinguen tendrán una importancia fundamental en el establecimiento de técnicas prácticas de programación lógica, así como en el alejamiento de esta de la declaratividad que en principio justificó su introducción.

Ejemplo 6.4.1 *Veamos el programa que define recursivamente el tipo número natural, que viene dado por las cláusulas:*

```
numero_natural(0).
numero_natural(siguiente(Numero)) :- numero_natural(Numero).
```

Consideremos la pregunta

```
:- numero_natural(siguiente(siguiente(0))).
```

Entonces, la aplicación directa de la definición 6.4.2 puede representarse gráficamente en forma de árbol de resolución, cuyo valor exacto para este caso es el indicado en la figura 6.2.

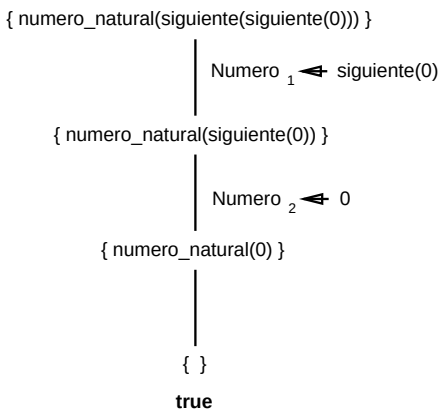


Figura 6.2: Árbol de resolución para la pregunta `:- numero_natural(siguiente(siguiente(0))).`

□

Es absolutamente necesario detenerse en algunas reflexiones acerca del trabajo expuesto hasta el momento. En efecto, el ejemplo 6.4.1 muestra un caso muy simple de resolución desde todo punto de vista:

- La pregunta realizada[9] no contiene variables y por tanto la respuesta no se expresa en función de las sustituciones aplicadas.
- El árbol de resolución es determinista, en el sentido en que cada nodo del mismo sólo posee un hijo en su descendencia. Ello equivale a decir que sólo existe una posible respuesta.
- Finalmente, el orden de lectura considerado para las cláusulas y objetivos de la resolvente no ha perturbado en forma alguna el proceso de resolución.

Ello no es, ni mucho menos, lo habitual, como vamos a demostrar en la discusión que sigue.

6.5 El concepto de retroceso

Como hemos comentado anteriormente, el recorrido del árbol de resolución Prolog se efectúa en profundidad. Este método consiste en un descenso directo a la rama más a la izquierda en el árbol de resolución. En este punto, es necesario llamar la atención sobre tres posibles casos:

1. Si nos encontramos con una resolvente vacía al extremo de la nueva rama, hemos encontrado una de las respuestas. Entonces, dependiendo de las implementaciones, Prolog se detiene y espera nuestras instrucciones o continúa buscando otras respuestas.
2. Si caemos en una rama infinita no podemos hacer otra cosa más que tratar de eliminar nuestro proceso desde el sistema

[9] `:- numero_natural(siguiente(siguiente(0))).`

operativo o esperar a que la pila de ejecución del intérprete se sature.

3. Si nos encontramos con un `fail` al extremo de la rama, entonces Prolog automáticamente continúa con la exploración de la rama inexplorada del árbol de resolución que esté más próxima por la derecha.

En los casos 1 y 3, la exploración de una nueva rama del árbol implica el retroceder paso a paso por la rama acabada de construir. En cada nodo revisitado, se pueden plantear dos casos:

- Si existe una rama no explorada por la derecha, esto es, queda alguna cláusula aplicable como alternativa a la anteriormente escogida para resolver el primer objetivo de la resolvente de ese nodo, entonces esa nueva posibilidad es explorada.
- En otro caso, remontamos un nodo más en nuestra rama y recomenzamos el proceso para el mismo, hasta haber agotado todas las posibilidades. Si esto último ocurre devolvemos `fail` como respuesta.

El *retroceso*, tal como ha sido descrito, no sólo se desencadena de forma automática por el intérprete Prolog cuando se encuentra con `fail` en el extremo de la rama del árbol de resolución que se explora en ese momento, sino que el usuario puede a voluntad forzar el retroceso en una rama en la que se ha obtenido una respuesta con éxito. La utilidad en este caso se reserva para la obtención de varias respuestas para una misma pregunta. Para forzar el retroceso una vez obtenida una respuesta, la mayoría de los intérpretes lógicos preveen la instrucción **;** tecleada directamente sobre el intérprete.

Ejemplo 6.5.1 *Volviendo al programa del ejemplo 6.4.1, consideremos una pregunta que incluya variables en su estructura:*

```
:- numero_natural(X).
```

Declarativamente, el significado de la pregunta es el siguiente:

> "Cuáles son los valores `Val` de la variable `X` para los que el átomo `numero_natural(Val)` es deducible a partir del programa. Esto es, los valores para los cuales `X` es un número natural."

Una cosa es, en principio, evidente: El conjunto de posibles soluciones es infinito. Veremos cómo se enfrenta nuestro intérprete a un problema de tal magnitud. En este caso, teniendo en cuenta que el árbol de resolución es el representado en la figura 6.3, concluimos que la profundidad del mismo es infinita y que a cada nodo de su estructura le corresponden dos hijos:

1. *El situado más a la izquierda es el resultado de la unificación final con la primera de las cláusulas del programa. Es sólo en estas ramas en las que el proceso de resolución devolverá una respuesta.*

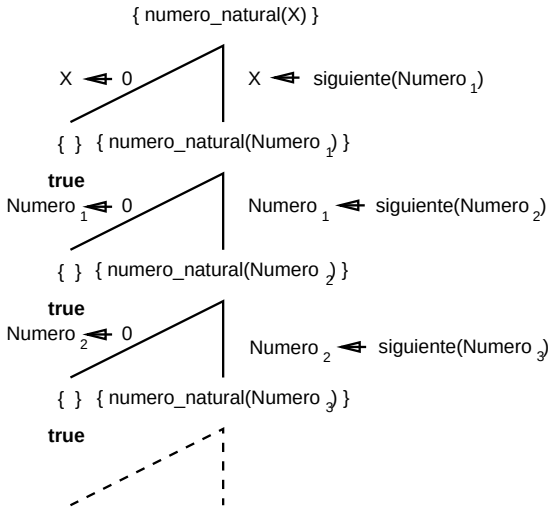


Figura 6.3: Resolución de `:- numero_natural(X).`

2. *El hijo restante es el resultado de la aplicación de la relación recursiva representada por la segunda de las cláusulas del programa.*

Para recuperar las soluciones, bastará con aplicar las sustituciones aplicadas sobre cada una de las ramas terminales. Así, obtendremos:

$$
\begin{array}{ll}
X & \leftarrow 0 \\
X & \leftarrow \texttt{siguiente}(\texttt{Numero}_1) \\
 & \leftarrow \texttt{siguiente}(0) \\
X & \leftarrow \texttt{siguiente}(\texttt{Numero}_1) \\
 & \leftarrow \texttt{siguiente}(\texttt{siguiente}(\texttt{Numero}_2)) \\
 & \leftarrow \texttt{siguiente}(\texttt{siguiente}(0)) \\
X & \leftarrow \texttt{siguiente}(\texttt{Numero}_1) \\
 & \leftarrow \texttt{siguiente}(\texttt{siguiente}(\texttt{Numero}_2)) \\
 & \leftarrow \texttt{siguiente}(\texttt{siguiente}(\texttt{siguiente}(\texttt{Numero}_3))) \\
 & \leftarrow \texttt{siguiente}(\texttt{siguiente}(\texttt{siguiente}(0))) \\
\vdots &
\end{array}
$$

Entre resultado y resultado, el usuario deberá forzar el retroceso en el intérprete Prolog. En relación a la cuestión de cuándo finalizará el cálculo, la respuesta es tan contundente como simple: Cuando la memoria reservada por el ordenador para uso de nuestro intérprete se agote. □

Ejemplo 6.5.2 *Se trata ahora de implementar el concepto de número natural par. Para ello bastará, en principio, con considerar la función* `siguiente` *del ejemplo 6.4.1. El esqueleto del nuevo programa podría venir dado por las dos reglas que siguen:*

```
par(0).
par(siguiente(siguiente(Numero))) :- par(Numero).
```

De hecho la semántica declarativa es la correcta, puesto que:

> "El cero es un número natural par."
> "El siguiente al siguiente de un número natural `Numero` es par, si `Numero` lo es."

Este programa se comporta conforme a lo esperado ante preguntas como

```
:- par(X).
```

que nos devuelve todos los posibles valores para los números naturales pares. La respuesta para preguntas con argumentos que no incluyan variables es igualmente satisfactoria, como sería el caso de:

```
:- par(siguiente(siguiente(0))).
```

que produce una respuesta afirmativa. □

Ejemplo 6.5.3 *De forma análoga, podemos implementar el concepto de número natural impar. Siguiendo la misma técnica que la del anterior ejemplo, nuestro programa podría ser el siguiente:*

```
impar(siguiente(0)).
impar(siguiente(siguiente(Numero))) :- impar(Numero).
```

cuya semántica declarativa es la dada por:

> "El uno es un número natural impar."
> "El siguiente al número natural `siguiente(Numero)` es impar, si `Numero` lo es."

Este programa se comporta conforme a lo esperado ante preguntas como

```
:- impar(X).
```

que nos devuelve todos los posibles valores para los números naturales impares. Como en el caso del ejemplo 6.5.2 la respuesta es igualmente satisfactoria para preguntas cuyos argumentos no incluyen variables. □

Ejemplo 6.5.4 *Como muestra más completa de programa lógico vamos a extender el programa del ejemplo 6.4.1 con el objeto de implementar el concepto de suma de números naturales. Para ello basta considerar el programa lógico siguiente:*

```
numero_natural(0).
numero_natural(siguiente(Numero)) :- numero_natural(Numero).

suma(0,Numero,Numero) :- numero_natural(Numero).
suma(siguiente(Numero_1),Numero_2,siguiente(Numero_3)) :-
                          suma(Numero_1,Numero_2,Numero_3).
```

donde `suma(Sumando_1,Sumando_2,Resultado)` *es la notación de usuario que determina la semántica declarativa del predicado* `suma`*, cuyo significado será:*

> "El resultado de la suma de `Sumando_1` y `Sumando_2` es `Resultado`."

y a partir de la cual hemos construido las dos cláusulas del predicado `suma` *en la forma:*

> "La suma del cero y de un número cualquiera, es dicho número si este es natural."
> "La suma del siguiente al número `Numero_1` y un número arbitrario `Numero_2`, es el siguiente al número `Numero_3`, si `Numero_3` es el resultado de sumar `Numero_1` y `Numero_2`."

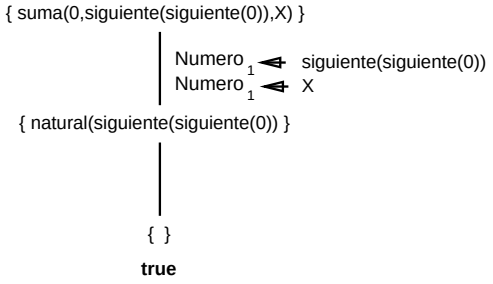


Figura 6.4: Resolución de `:- suma(0,siguiente(siguiente(0)),X).`

Una vez el programa definido, consideremos la pregunta:

```
:- suma(0,siguiente(siguiente(0)),X ).
```

Esto es, estamos preguntando por el valor de la suma de 0 y 2. El proceso de resolución es en este caso el representado en la figura 6.4, cuya única solución es:

$$X \leftarrow \texttt{Numero}_1 \leftarrow \texttt{siguiente}(\texttt{siguiente}(0))$$

□

6.6 Significado de un programa lógico

Es fundamental resaltar dos cuestiones importantes que afectarán a la congruencia declarativo/operacional de Prolog. En efecto, desde un punto de vista meramente operacional:

1. La elección de los objetivos de la resolvente que en cada caso deberán ser resueltos, se realizará en un orden determinado. Ello tendrá como consecuencia que el funcionamiento de los programas Prolog sea directamente dependiente de dicho orden, cuando desde un punto de vista meramente declarativo ello es irrelevante.

2. Lo mismo se puede decir sobre el orden de búsqueda de las cláusulas cuya cabeza unificará con el objetivo a resolver.

Esta situación se traduce formalmente en una posible diferencia entre el *significado*[10] *de un programa lógico* P que notaremos por $S(P)$, y su *significado deseado*[11] $D(P)$. Concretamente, definimos los conceptos de *corrección* y *completud* de un programa lógico.

Definición 6.6.1 *Sea P un programa lógico, entonces decimos que P es* correcto respecto a una significación deseada $D(P)$ *si $S(P) \subseteq D(P)$. Decimos que P es* completo respecto a $D(P)$ *si $D(P) \subseteq S(P)$.*

[10] esto es, el conjunto de objetivos deducibles aplicando el algoritmo de resolución.

[11] es decir, el conjunto de objetivos que el programador desearía que fuesen deducibles a partir del programa.

Intuitivamente, un programa es correcto cuando todas las soluciones calculadas entran dentro del conjunto de las previstas por el programador, mientras que es completo cuando todas las soluciones previstas por el programador pueden ser efectivamente calculadas por el programa. □

Lejos de constituir casos aislados, las incongruencias declarativo/operacionales son, en la práctica, el talón de aquiles de la programación lógica. La única solución práctica hoy por hoy es la consideración de la semántica declarativa de las cláusulas como una primera aproximación a la semántica operacional del programa, que en definitiva es la que cuenta[12]. Eso si, ya hemos perdido parte de la magia de un lenguaje lógico puro.

Con el fin de presentar un caso típico de incongruencia declarativo/operacional, vamos a intentar aplicar toda la potencia de la semántica declarativa a nuestro programa del ejemplo 6.5.4.

Ejemplo 6.6.1 *Continuando con el programa del ejemplo 6.5.4, consideraremos ahora la pregunta*

```
:- suma(X,Y,siguiente(siguiente(0))).
```

Ello equivale a intentar resolver la ecuación

$$X + Y = 2$$

Obtenemos el árbol de resolución representado en la figura 6.5,

[12] en este punto, es muy importante dejar claro que los distintos métodos de resolución considerados en programación lógica son siempre correctos y completos. Es en el momento de su implementación cuando, atendiendo a razones de economía de recursos y de eficiencia general del proceso, el diseñador puede optar por limitar la completud del algoritmo. En todo caso, la corrección debe estar siempre asegurada.

y por tanto las respuestas son en este caso las dadas por:

$$\begin{cases} X \leftarrow 0 \\ Y \leftarrow \texttt{Numero\_1}_1 \leftarrow \texttt{siguiente}(\texttt{siguiente}(0)) \end{cases}$$

$$\begin{cases} X \leftarrow \texttt{siguiente}(\texttt{Numero\_1}_1) \leftarrow \texttt{siguiente}(0) \\ Y \leftarrow \texttt{Numero\_2}_1 \leftarrow \texttt{siguiente}(0)) \end{cases}$$

$$\begin{cases} X & \leftarrow \texttt{siguiente}(\texttt{Numero\_1}_1) \leftarrow \texttt{siguiente}(\texttt{siguiente}(\texttt{Numero\_1}_2)) \\ & \leftarrow \texttt{siguiente}(\texttt{siguiente}(0)) \\ Y & \leftarrow \texttt{Numero\_2}_1 \leftarrow \texttt{Numero\_2}_2 \leftarrow 0 \end{cases}$$

□

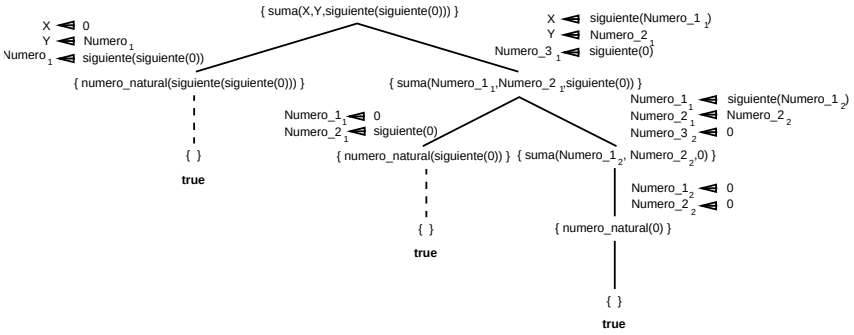


Figura 6.5: Resolución de `:- suma(X,Y,siguiente(siguiente(0))).`

Está claro que en el caso de la pregunta considerada en el ejemplo 6.6.1 el programa no presenta mayores problemas, y en principio todo parece funcionar como estaba previsto en la semántica declarativa. Vamos a ver que desgraciadamente ello es tan solo un simple espejismo. Comenzamos con las incongruencias. Para ello, y antes de pasar al ejemplo concreto, debemos subrayar el hecho de que con el significado declarativo considerado, nuestro programa debería en principio ser capaz de resolver cualquier ecuación de números naturales de la forma

$$X + Y = Z$$

cuando veremos que desde un punto de vista operacional ello no es ni remotamente posible.

Ejemplo 6.6.2 *Supongamos que al programa considerado en el ejemplo 6.5.4 se le interroga con la pregunta*

```
:- suma(X,Y,Z)
```

Esto es, nos interesamos por las soluciones de la ecuación

$$X + Y = Z$$

Desde un punto de vista exclusivamente declarativo, nuestro programa debería ser capaz de responder; sin embargo dicho empeño sólo podrá ser posible en parte, como muestra la figura 6.6 que representa el correspondiente árbol de resolución.

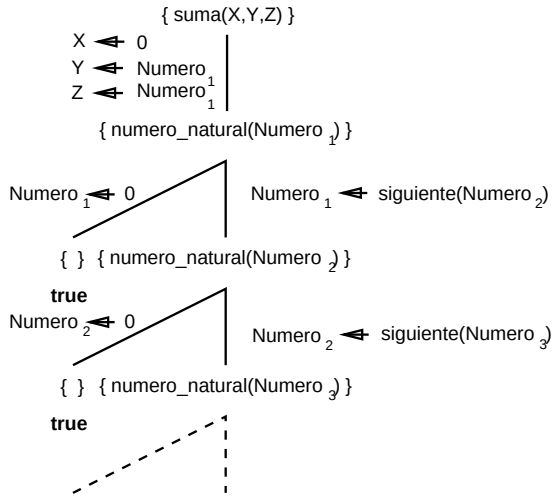


Figura 6.6: Resolución de `:- suma(X,Y,Z).`

En efecto, ante la pregunta considerada, el programa jamás llega a proporcionar ni una sola de las respuestas válidas de la ecuación. En síntesis, el problema consiste en que debido

al método particular aplicado para la exploración del árbol de resolución, en profundidad, el algoritmo de resolución es incapaz de salirse del proceso que implica la determinación de la naturaleza del primero de los sumandos X, *como número natural.* □

En definitiva, podemos concluir que en sí mismo, la implementación del algoritmo SLD considerado en la práctica por Prolog, no es completo. De hecho la completud se sacrifica en aras tanto de la eficiencia del sistema como de la economía de recursos. El problema así planteado es endémico en programación lógica, y su resolución todavía está abierta. En este sentido, los esfuezos se concentran en el diseño de nuevos esquemas de resolución [15, 26], para los que la exigencia de completud en los cálculos no se traduzca en una explotación masiva de recursos, a la vez que se asegura un buen comportamiento temporal del sistema.

Profundizando ahora en las razones de la incompletud operacional de la mayoría de los dialectos Prolog, diremos que el *paradigma declarativo* en el que se basa la programación lógica sólo establece relaciones entre los objetos manejados en un programa, pero en ningún caso un orden en el tratamiento de estas. En este sentido, es necesario hacer hincapié en el hecho de que ni el método de unificación de Robinson del algoritmo 6.3.1, ni el intérprete lógico del algoritmo 6.4.1 establecen ningún tipo de orden en el tratamiento de los objetos y relaciones del programa lógico. El aspecto expuesto es determinante para la comprensión de los problemas planteados en la práctica por la programación lógica y nuestro objetivo será ahora el de ilustrar mediante ejemplos simples las situaciones provocadas por este alejamiento del paradigma declarativo. Como primer factor causante de tales incongruencias nos referiremos al orden de tratamiento de los objetivos durante el proceso de resolución establecido por el algoritmo 6.4.1 y que fue fijado en la definición 6.4.2 como el significado en la práctica del concepto de árbol de resolución. Como segundo factor, nos referiremos

al orden de exploración del conjunto de cláusulas que forman nuestro programa y que recordamos también fue fijado en la definición 6.4.2.

A pesar de las limitaciones expuestas en el ejemplo 6.6.2 con respecto al funcionamiento de la definición lógica propuesta para la suma, esta puede servir de base para la creación de una pequeña biblioteca matemática que considere las operaciones numéricas fundamentales, tal y como se mostrará en los capítulos siguientes.

6.6.1 El orden de tratamiento de los objetivos

Para ilustrar la influencia de este factor en la incompletud operacional de Prolog plantearemos un problema sencillo, el de la definición del concepto de persona. Nuestra semántica declarativa será la siguiente:

> *"Alguien es una persona si tiene una madre y esta es una persona."*

Claramente, la definición considerada es válida y el orden de las condiciones establecidas es irrelevante. Esto es, el orden de tratamiento de los conceptos:

- Tener una madre.
- La madre es una persona.

no altera la relación establecida en nuestra definición del concepto. Sin embargo, veremos que este orden sí influye en el tratamiento que en la práctica realiza Prolog. Así, una posible implementación es la dada por la regla

```
persona(Persona) :- persona(Madre), madre(Persona, Madre).
```

donde estamos suponiendo que `persona(Persona)` es cierto si `Persona` es un individuo humano. Evidentemente, es necesario completar nuestra base de datos con el fin de establecer las relaciones `madre(Persona, Madre)` pertinentes. Un primer acercamiento podría ser el programa:

```
persona(Persona) :- persona(Madre), madre(Persona, Madre).

madre('Javier', 'Rosa').
madre('Juan', 'Maria').
madre('Eva', 'Nuria').

persona('Rosa').
persona('Maria').
persona('Nuria').
```

para el cual, considerando el orden de tratamiento de los objetivos de la definición 6.4.2, el usuario no obtendrá respuesta para la pregunta

```
:- persona('Juan').
```

por agotamiento de los recursos de memoria. La razón es simple y estriba en que para responder a la misma, el intérprete aplica la siguiente semántica operacional:

> *"El individuo* `Persona` *es una persona si se puede demostrar que el individuo* `Madre` *es una persona y que además es la madre de* `Persona`*."*

El problema radica en que para responder al objetivo `persona(Madre)` el intérprete volverá a aplicar la primera de las cláusulas, entrando en un bucle infinito. Sin embargo, la situación varía simplemente con cambiar el orden de los términos de la primera cláusula. En efecto, consideremos ahora el conjunto de cláusulas siguiente:

```
persona(Persona) :- madre(Persona, Madre), persona(Madre).

madre('Javier', 'Rosa').
madre('Juan', 'Maria').
madre('Eva', 'Nuria').

persona('Rosa').
persona('Maria').
persona('Nuria').
```

La semántica declarativa es la misma del programa anterior, pero la operacional ha cambiado sustancialmente debido a la

alteración en el orden de tratamiento de los objetivos a tratar durante el proceso de resolución. Ahora la semántica operacional es:

> *"El individuo* `Persona` *es una persona si se puede demostrar que tiene una madre* `Madre` *y esta es una persona."*

Más exactamente, la diferencia consiste en que cuando ahora planteamos la pregunta

```
:- persona('Juan').
```

en primer lugar buscamos la identidad de su madre, en caso de existir, y que en este caso es `'Maria'`. Cuando ahora intentamos resolver el objetivo `persona(Madre)`, la variable `Madre` ha sido instanciada al valor `'Maria'` y su resolución es inmediata y afirmativa. Esto es, la respuesta a nuestra pregunta inicial es ahora `true`.

6.6.2 El orden de aplicación de las cláusulas

Con el fin de discutir la influencia de este nuevo aspecto utilizaremos el mismo ejemplo usado para ilustrar la influencia del orden de tratamiento de los objetivos en el proceso de resolución: la definición del concepto de persona. En efecto, aun sin cambiar el orden de los términos en las cláusulas, podemos reconstruir nuestro programa de forma que podamos obtener una respuesta congruente para la pregunta

```
:- persona('Juan').
```

El razonamiento es muy simple: se trata de conseguir que cuando se realice la resolución del objetivo `persona(Madre)`, ello no conlleve la aplicación de la cláusula recursiva, que era la responsable de la entrada en un bucle infinito. Una vez fijado el orden de aplicación de las cláusulas del programa por la definición 6.4.2, consideremos el nuevo programa:

```
persona('Rosa').
persona('Maria').
persona('Nuria').

persona(Persona) :- persona(Madre), madre(Persona, Madre).

madre('Javier', 'Rosa').
madre('Juan', 'Maria').
madre('Eva', 'Nuria').
```

cuya semántica declarativa es idéntica a la del original. La respuesta a nuestra pregunta es ahora `true`, aun cuando la estructura interna de las cláusulas no ha variado en absoluto.

En este punto es conveniente clarificar qué es lo que realmente puede esperar el programador de un lenguaje como Prolog y cuáles son sus ventajas en relación a los lenguajes clásicos. Así, por un lado es evidente que el atractivo fundamental estriba en la declaratividad que podemos imprimir a nuestro estilo de programación. Ello es fácilmente verificable en la práctica, en la que unas cuantas cláusulas Prolog suelen ser equivalentes a páginas completas de código escrito en otros lenguajes clásicos de probada eficacia.

Sin embargo, y a la vista de los resultados presentados en esta sección, el programador debería considerar este aspecto declarativo sólo como una potente facilidad orientativa en su trabajo. En ningún caso deberá olvidar que el lenguaje tiene una componente operacional importante, inducida principalmente por los dos factores comentados anteriormente. En este sentido, es importante señalar también que estas dos limitaciones al paradigma declarativo son generalmente superables, desarrollándose un estilo de programación característico que consiste en tener en cuenta el orden de los términos y de las cláusulas del programa en relación al algoritmo de resolución del intérprete lógico. En resumen, dos reglas son suficientes en la mayor parte de los casos para resolver esta situación:

1. Dado que los objetivos se resuelven de izquierda a derecha según el orden preestablecido en Prolog por la definición 6.4.2, el programador debería asegurarse de que ello asegura la instanciación del mayor número posible de variables en los objetivos inmediatos a resolver.

2. En la misma línea, el programador deberá asegurarse de que la exploración de la base de datos constituida por las cláusulas, de arriba hacia abajo, asegura que los casos particulares sean tratados con la mayor brevedad posible, evitando en lo posible la caída en ciclos infinitos durante la construcción del árbol de resolución.

Aparte de los factores ya señalados, existen otros que influyen también de forma definitiva en el alejamiento, cuando programamos en Prolog, de la programación lógica con respecto al paradigma declarativo. Se trata fundamentalmente de la introducción en el lenguaje del concepto de *predicado no lógico*. Esto es, de predicados con un significado lógico irrelevante, que no expresan ningún tipo de relación sino tan sólo comportamientos operacionales. Este es típicamente el caso de las estructuras de control o de la gestión de las entradas y salidas.

Capítulo 7

Control en Prolog: el Corte

Hemos visto la importancia de entender, en la práctica, el orden en el que Prolog resuelve las cuestiones planteadas. En particular, hemos intuido lo interesante de la posibilidad de controlar de algún modo dicho orden. En este sentido, el *corte* es el método más utilizado para establecer un control de la resolución en un programa Prolog. Pero no nos engañemos, su utilización abrirá a su vez nuevos interrogantes.

7.1 Semántica operacional

Básicamente, el corte es un predicado cuya utilización provoca un efecto colateral sobre la estructura del árbol de resolución, forzando a que ciertas ramas que en principio podrían ser exploradas en caso de retroceso, no lo sean en la práctica. Esto es, se trata de una forma muy simple de determinizar nuestra búsqueda de soluciones.

Definición 7.1.1 *El* corte *es un predicado sin argumentos, que se verifica siempre y que suele representarse mediante la notación* **!**. *Se ejecuta en el momento en el que es llamado, esto es, en el momento en que se encuentra como primer objetivo*

a la izquierda de nuestra resolvente. Como efecto colateral, suprime en el nodo actual de nuestro árbol de resolución todas las alternativas que puedan quedar por explorar para los predicados que se encuentren entre la cabeza de la cláusula en la que aparece, y el lugar en el que el corte se ha ejecutado. □

$$
\begin{array}{ll}
a :- b_1, \ldots, b_{i-1}, !, b_i, \ldots, b_r. & b_{i-1} :- c^1_{i-1,1}, c^1_{i-1,2}, \ldots, c^1_{i-1,m_{i-1,1}}. \\
b_1 :- c^1_{1,1}, c^1_{1,2}, \ldots, c^1_{1,m_{1,1}}. & \vdots \\
\vdots & b_{i-1} :- c^{l_{i-1}}_{i-1,1}, c^{l_{i-1}}_{i-1,2}, \ldots, c^{l_{i-1}}_{i-1,m_{i-1,l_{i-1}}}. \\
b_1 :- c^{l_1}_{1,1}, c^{l_1}_{1,2}, \ldots, c^{l_1}_{1,m_{1,l_1}}. & \vdots \\
\vdots & b_{i-1} :- c^{n_{i-1}}_{i-1,1}, c^{n_{i-1}}_{i-1,2}, \ldots, c^{n_{i-1}}_{i-1,m_{i-1,n_{i-1}}}. \\
b_1 :- c^{n_1}_{1,1}, c^{n_1}_{1,2}, \ldots, c^{n_1}_{1,m_{1,n_1}}. & b_i :- c^1_{i,1}, c^1_{i,2}, \ldots, c^1_{i,m_{i,1}}. \\
b_2 :- c^1_{2,1}, c^1_{2,2}, \ldots, c^1_{2,m_{2,1}}. & \vdots \\
\vdots & b_i :- c^{n_i}_{i,1}, c^{n_i}_{i,2}, \ldots, c^{n_i}_{i,m_{i,n_i}}. \\
b_2 :- c^{l_2}_{2,1}, c^{l_2}_{2,2}, \ldots, c^{l_2}_{2,m_{2,l_2}}. & \vdots \\
\vdots & b_r :- c^1_{r,1}, c^1_{r,2}, \ldots, c^1_{r,m_{r,1}}. \\
b_2 :- c^{n_2}_{2,1}, c^{n_2}_{2,2}, \ldots, c^{n_2}_{2,m_{2,n_2}}. & \vdots \\
\vdots & b_r :- c^{n_r}_{r,1}, c^{n_r}_{r,2}, \ldots c^{n_r}_{r,m_{r,n_r}}.
\end{array}
$$

Figura 7.1: Un programa Prolog genérico

El concepto de corte es tan importante como difícil de asimilar, es por ello que será objeto de especial atención a lo largo del presente texto. Para comenzar, ilustraremos su utilización en el entorno de un programa abstracto. Más exactamente, consideremos un programa lógico incluyendo el conjunto de cláusulas representado en la figura 7.1. y supongamos que en un momento dado nuestra resolvente es de la forma

$$\{a, \ldots\}$$

verificándose además todos los objetivos

$$b_1, b_2, \ldots b_{i-1}$$

y siendo las correspondientes primeras cláusulas para las que dichos objetivos se verifican[1] aquellas indicadas con los superíndices

$$l_1, l_2, \ldots l_{i-1}$$

respectivamente. Entonces, una vez el corte se ha ejecutado, todas las alternativas de las cláusulas cuya cabeza se encuentra en el conjunto

$$b_1, b_2, \ldots b_{i-1}$$

y que todavía no han sido exploradas, se cortan para la resolución del objetivo `a`. Esto es, en caso de un retroceso sobre las ramas construidas entre el nodo del árbol de resolución indicado por $\{a, \ldots\}$ y aquel en el que el corte es el primer objetivo, ninguna de las alternativas indicadas por las cláusulas

$$b_1^{j_1}, b_2^{j_2}, \ldots, b_{i-1}^{j_{i-1}}, \text{ tal que } j_1 > l_1, j_2 > l_2, \ldots, j_{i-1} > l_{i-1}$$

se tendrán en cuenta en el proceso de resolución. Informalmente, la inclusión de un corte en una cláusula es una forma de decir:

> *"Si hemos llegado hasta aquí, es que estamos en el buen camino y por tanto es inútil intentar otras alternativas en el camino ya recorrido."*

Ejemplo 7.1.1 *Como primer ejemplo para mostrar las posibilidades del corte, consideremos la implementación de la multiplicación de números naturales en Prolog.*

La semántica declarativa del predicado `mult` *viene dada por la notación de usuario* `mult(Factor_1,Factor_2,Resultado)` *cuyo significado será:*

> "El resultado de la multiplicación de `Factor_1` por `Factor_2` es `Resultado`."

[1] observar de nuevo la importancia del orden de exploración del conjunto de reglas del programa.

y a partir de la cual hemos construido las cláusulas del predicado `mult` *en la forma:*

> "La multiplicación del cero por un número natural cualquiera es siempre cero."
> "La multiplicación del uno por un número natural cualquiera es ese mismo número."
> "La multiplicación del siguiente al número `Numero_1` por un número arbitrario `Numero_2`, es el número `Numero_3`, donde `Numero_3` es el valor de sumar `Numero_2` con el resultado de la multiplicación de `Número_1` por `Número_2`."

```
numero_natural(0).
numero_natural(siguiente(Numero)) :- numero_natural(Numero).

suma(0,Numero,Numero) :- numero_natural(Numero).
suma(siguiente(Numero_1),Numero_2,siguiente(Numero_3)) :-
                          suma(Numero_1,Numero_2,Numero_3).

mult(0,Numero,0) :- numero_natural(Numero).
mult(siguiente(0),Numero,Numero) :- !, numero_natural(Numero).
mult(siguiente(Numero_1),Numero_2,Numero_3) :-
                          mult(Numero_1,Numero_2,Numero_4),
                          suma(Numero_4,Numero_2,Numero_3).
```

Una vez el programa construido, consideremos como una posible pregunta:

```
:- mult(siguiente(0),0,X).
```

esto es, estamos preguntando por el valor de

$$X = 1 * 0$$

cuyo árbol de resolución está representado en la figura 7.3. La única respuesta obtenida será la dada por

$$X \leftarrow \texttt{Numero}_1 \leftarrow 0$$

Es importante observar que, en principio, es posible un retroceso sobre la segunda rama de la figura 7.3, pero ello es evitado por la presencia del corte. Supongamos que la definición considerada para el predicado `mult` *fuese la siguiente:*

```
mult(0,Numero,0) :- numero_natural(Numero).
mult(siguiente(0),Numero,Numero) :- numero_natural(Numero).
mult(siguiente(Numero_1),Numero_2,Numero_3) :-
                        mult(Numero_1,Numero_2,Numero_4),
                        suma(Numero_4,Numero_2,Numero_3).
```

entonces la semántica declarativa de nuestro programa es exactamente la misma, pero la operacional ha cambiado. Veamos lo que ocurre cuando interrogamos al intérprete con nuestra nueva base de datos. De hecho estaremos en condiciones de obtener dos soluciones forzando el retroceso, sin embargo este esfuerzo de cálculo es inútil puesto que la segunda respuesta es la misma que la primera. Más exactamente, las respuestas obtenidas son:

$$X \leftarrow \texttt{Numero}_1 \leftarrow 0$$
$$X \leftarrow \texttt{Numero\_3}_1 \leftarrow 0$$

siendo el correspondiente árbol de resolución el mostrado en la figura 7.2. □

Este último ejemplo 7.1.1 muestra una de las aplicaciones fundamentales del corte, a saber, la determinización del proceso de resolución con el objetivo final de eliminar soluciones que bien por la naturaleza del problema[2], bien por la naturaleza del código[3], hemos decidido no tener en cuenta.

Ejemplo 7.1.2 *A partir de la definición de multiplicación de números naturales considerada en el anterior ejemplo 7.1.1, podemos construir un predicado*

```
exp(Base, Exponente, Resultado)
```

[2]del conjunto de posibles soluciones, puede interesarnos únicamente encontrar una.

[3]la consideración de cláusulas especializadas en la resolución de un determinado caso en un problema, ya que puede conducirnos a obtener varias veces una misma solución si no tenemos en cuenta un método de control de la resolución.

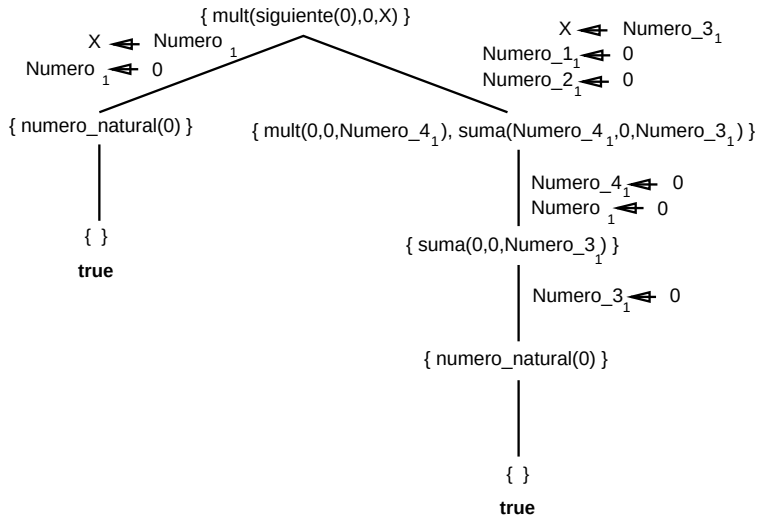


Figura 7.2: Resolución de `:- mult(siguiente(0),0,X).`, sin corte

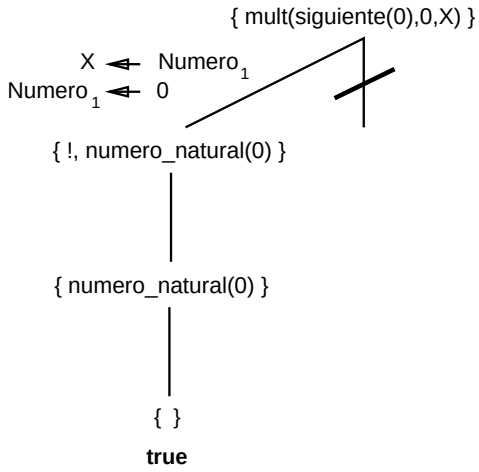


Figura 7.3: Resolución de `:- mult(siguiente(0),0,X).`, con corte

que se verifique cuando

$$Base^{Exponente} = Resultado$$

Esto es, se trata de implementar la exponenciación de números naturales. El programa propuesto posee una estructura similar a la del ejemplo 7.1.1, si nos limitamos al caso de los números naturales, y viene dado por las cláusulas siguientes:

```
numero_natural(0).
numero_natural(siguiente(Numero)) :- numero_natural(Numero).

suma(0,Numero,Numero) :- numero_natural(Numero).
suma(siguiente(Numero_1),Numero_2,siguiente(Numero_3)) :-
                         suma(Numero_1,Numero_2,Numero_3).

mult(0,Numero,0) :- numero_natural(Numero).
mult(siguiente(0),Numero,Numero) :- !, numero_natural(Numero).
mult(siguiente(Numero_1),Numero_2,Numero_3) :-
                         mult(Numero_1,Numero_2,Numero_4),
                         suma(Numero_4,Numero_2,Numero_3).

exp(Numero, siguiente(0), Numero) :- numero_natural(Numero), !.
exp(Numero, 0, siguiente(0)).
exp(Numero, siguiente(Exponente), Resultado) :-
                         exp(Numero, Exponente, Parcial),
                         mult(Parcial, Numero, Resultado).
```

donde las únicas cláusulas introducidas en relación al ejemplo 7.1.1 son las tres últimas, cuya semántica declarativa viene dada por:

> "El resultado de elevar un número natural `Numero` a la potencia uno, es ese mismo número."
> "El resultado de elevar un valor numérico cualquiera a la potencia cero, es el uno."
> "El resultado `Resultado` de elevar un número `Numero` al número natural siguiente a `Exponente`, es igual a calcular la potencia `Exponente` de dicho número `Numero` y multiplicar el resultado `Parcial` de dicha operación nuevamente por `Numero`."

En este caso, la razón por la que hemos introducido un corte en la parte derecha de la primera cláusula del predicado `exp`*, es para evitar retrocesos sobre la tercera cuando el segundo argumento es un uno.* □

Siguiendo la misma pauta, podemos ahora implementar el concepto matemático de resto. Analíticamente, decimos que un número natural `Resto` es el resto módulo `Y` de un número `X` si el resto de dividir `X` por `Y` es `Resto`. Como en los ejemplos anteriores, nos limitaremos por comodidad al caso de los números naturales.

Ejemplo 7.1.3 *Para implementar el concepto matemático de resto, basta considerar el siguiente conjunto de reglas:*

```
numero_natural(0).
numero_natural(siguiente(Numero)) :- numero_natural(Numero).

suma(0,Numero,Numero) :- numero_natural(Numero).
suma(siguiente(Numero_1),Numero_2,siguiente(Numero_3)) :-
                         suma(Numero_1,Numero_2,Numero_3).

mult(0,Numero,0) :- numero_natural(Numero).
mult(siguiente(0),Numero,Numero) :- !, numero_natural(Numero).
mult(siguiente(Numero_1),Numero_2,Numero_3) :-
                         mult(Numero_1,Numero_2,Numero_4),
                         suma(Numero_4,Numero_2,Numero_3).

inferior(0, 0) :- !, fail.
inferior(0, Numero) :- numero_natural(Numero).
inferior(siguiente(Numero_1), siguiente(Numero_2)) :-
                         inferior(Numero_1, Numero_2).

modulo(Dividendo, Divisor, Dividendo) :-
                         inferior(Dividendo, Divisor), !.
modulo(Dividendo, Divisor, Resto) :-
                         suma(Parcial, Divisor, Dividendo),
                         modulo(Parcial, Divisor, Resto).
```

□

Esta no es, sin embargo, la única aplicación del corte. En efecto, el uso más general dado a esta estructura fundamental de control es el de optimizar la búsqueda de soluciones, eliminando posibilidades que el programador sabe que no pueden ser correctas y que el algoritmo de resolución podría decidir explorar[4]. En resumen, se trata de guiar al intérprete por el buen camino, impidiéndole entretenerse en cálculos infructuosos.

Ejemplo 7.1.4 *Se trata de implementar un programa capaz de deducir el número de progenitores paternos de una persona, teniendo en cuenta que Adán y Eva no tuvieron ninguno. Para ello, consideraremos el predicado*

```
numero_de_padres(Persona, Numero)
```

que será cierto si el número de progenitores de `Persona` *es* `Numero`*. En este sentido, una primera versión de nuestro programa podría ser:*

```
numero_de_padres('Adan',0).
numero_de_padres('Eva',0).
numero_de_padres(Persona,2).
```

cuya semántica declarativa es la siguiente:

> "El número de padres de Adán es cero."
> "El número de padres de Eva es cero."
> "Cualquier persona tiene dos padres."

sin embargo, la posibilidad de efectuar retrocesos imprevistos hace que nuestra semántica operacional difiera de nuestra idea original. En efecto, consideremos la pregunta:

```
:- numero_de_padres('Adan',X).
```

obtendremos dos respuestas distintas si el retroceso es forzado:

$$X \leftarrow 0$$
$$X \leftarrow 2$$

[4]en la exploración de tales alternativas se obtendría finalmente un `fail` como respuesta.

de las cuales la segunda es claramente incorrecta. El correspondiente árbol de resolución puede verse en la figura 7.4. La solución en principio está en cortar la última posibilidad explorada por el intérprete; para ello bastaría con corregir el programa inicial en la forma:

```
numero_de_padres('Adan',0) :- !.
numero_de_padres('Eva',0) :- !.
numero_de_padres(Persona,2).
```

con lo que pudiéramos pretender que la semántica operacional fuese:

> "El número de padres de Adán es cero."
> "El número de padres de Eva es cero."
> "Cualquier persona que no sea ni Eva ni Adán, tiene dos padres."

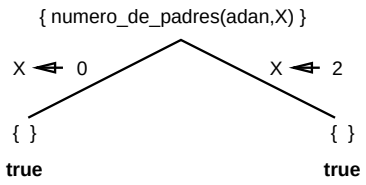


Figura 7.4: Resolución de `:- numero_de_padres('Adan',X).`, sin corte

Sin embargo, no es ese el caso. Aunque a la pregunta anterior

```
:- numero_de_padres('Adan',X).
```

obtenemos como respuesta

$$X \leftarrow 0$$

cuando planteamos la nueva pregunta

```
:- numero_de_padres('Adan',2).
```

la respuesta obtenida es `true`*. La explicación está en que no hemos previsto todos los casos posibles de unificación.*

Más exactamente, no hemos guiado adecuadamente a nuestro intérprete durante la resolución. Una solución efectiva sería la siguiente:

```
numero_de_padres('Adan',N) :- !, N=0.
numero_de_padres('Eva',N) :- !, N=0.
numero_de_padres(Persona,2).
```

donde el predicado "=" es un predicado no lógico[5] *que implica en este caso la unificación de la variable* `N` *con el valor 0. Debe observarse que ahora no hemos limitado ninguna posibilidad de unificación en cuanto al número de padres, ni para Eva ni para Adán. Eso sí, una vez verificado el hecho de que la persona en cuestión es uno de los dos personajes bíblicos, cortamos todas las otras posibilidades para el predicado* `numero_de_padres`. □

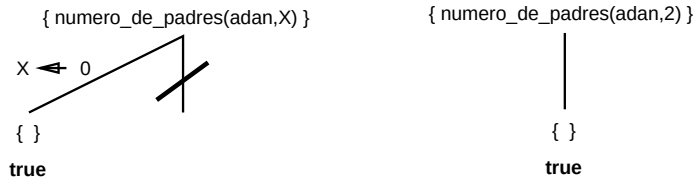


Figura 7.5: Árboles con corte para `:- numero_de_padres('Adan',X).` y `:- numero_de_padres('Adan',2).`

Una tercera posibilidad de aplicación del predicado de corte, es su combinación con el predicado predefinido `fail`. En este caso, se pretende emular de algún modo el funcionamiento de la negación lógica, pero ello constituye por mérito propio un capítulo aparte.

Un caso típico de utilización del corte es la generación de estructuras de control a semejanza de las existentes en los lenguajes clásicos de programación: `if`, `while`, `repeat`, ... Ello recalca el papel del corte como estructura fundamental de control en programación lógica. Para ilustrar este tipo de aplicaciones, consideremos un caso concreto de implementación.

[5] serán introducidos más tarde.

Ejemplo 7.1.5 *Se trata de implementar una estructura clásica de control, la instrucción* `if_then_else`*. Su semántica será la habitual. Así, definiremos el predicado* `if_then_else(P,Q,R)` *en la forma:*

"Si P es cierto, entonces probar Q, sino probar R."

y cuya implementación es la siguiente:

```
if_then_else(P,Q,R) :- P, !, Q.
if_then_else(P,Q,R) :- R.
```

En este caso, el corte indica que una vez probado el objetivo P*, el único camino a seguir es el indicado por la primera cláusula de la definición, esto es, probar* Q*.* □

Ejemplo 7.1.6 *Un ejercicio simple consiste en implementar un predicado* `intentar(P)` *que demuestre la veracidad de* `P` *si ello es posible. En otro caso, la respuesta a nuestra pregunta debe ser* `true`*. La funcionalidad requerida requiere tan sólo dos cláusulas:*

```
intentar(Objetivo) :- Objetivo,!.
intentar(Objetivo).
```

cuya semántica declarativa viene dada por:

"Si `Objetivo` es cierto, demostrarlo."
"Si `Objetivo` no es cierto, devolver `true`."

La razón que justifica en este caso la consideración de un corte como último literal en la parte derecha de la primera cláusula, es que cuando `Objetivo` *ha sido demostrado ya sabemos que la respuesta debe ser afirmativa, y ello sin tener que recurrir a la relación expresada por la segunda cláusula.* □

7.2 Tipos de cortes

Un punto importante en el que merece la pena detenerse en relación al uso de los cortes, es su clasificación en función de los efectos que su desaparición puede provocar en el significado de un programa. Se trata, en definitiva, de medir el impacto del uso de cortes en la mantenibilidad de los programas.

7.2.1 El corte verde

Informalmente, diremos que un corte es un *corte verde*, cuando su presencia o no, no quita ni pone nada al significado de un programa. Ello quiere decir simplemente que este tipo de corte expresa determinismo, esto es, se trata en todo caso de eliminar respuestas repetidas. Tal es el caso del corte del ejemplo 7.1.1, comentado ya ampliamente.

7.2.2 El corte rojo

En el caso de un *corte rojo*, su presencia o no en el programa provoca un cambio en el significado del mismo. Ello implica que no se trata simplemente de determinizar un programa, sino que la inclusión del corte se justifica exclusivamente desde un punto de vista operacional. Un caso simple es el considerado en el ejemplo 7.1.4. En la práctica, cualquier utilización de un corte rojo debería estar claramente indicada en el código, en forma de comentario.

En definitiva, la utilización de cortes es objeto de continuas controversias. En efecto, su uso pretende por un lado limitar el espacio de búsqueda en los árboles de resolución, no sólo con el objeto de mejorar las prestaciones, sino también para permitir ciertas funcionalidades[6] que de otro modo no serían posibles. Sin embargo, por otro lado, su introducción abre todavía más la brecha existente entre las semánticas declarativa y operacional de nuestros programas lógicos. A este respecto, conviene no olvidar dos cuestiones importantes, y enfrentadas:

1. Gran parte de las aplicaciones del corte no pueden entenderse desde otro punto de vista que no sea el operacional.

2. El lenguaje Prolog pretende ser un lenguaje declarativo.

[6] como veremos en el caso de la negación.

En este sentido, está claro que el uso del corte debe ser moderado con el fin de mejorar la eficacia sin por ello comprometer la claridad de nuestros programas.

7.3 Las variables anónimas

La mayoría de los programas lógicos son susceptibles de incluir en algunas de sus cláusulas variables que son irrelevantes para el proceso de resolución, pero cuya presencia implica una complicación innecesaria de tal proceso. Como muestra tomemos el caso de la instrucción `if_then_else` considerada en el ejemplo 7.1.5:

```
if_then_else(P,Q,R) :- P, !, Q.
if_then_else(P,Q,R) :- R.
```

En efecto, tanto `R` en la primera cláusula, como `P` y `Q` en la segunda, aparecen una sola vez en la estructura de sus respectivas cláusulas y por tanto se trata de variables irrelevantes a nivel del proceso de unificación, puesto que no son incluidas en ninguna de las ecuaciones consideradas en el algoritmo 6.3.1. Sin embargo, una vez introducidas en la cláusula, el intérprete lógico no siempre puede detectar su irrelevancia. Como consecuencia, son asimiladas en el proceso de resolución como una variable más, sujeta a renombramientos, inclusión en las sustituciones, ... En definitiva, las variables en cuestión sólo consiguen dificultar innecesariamente el trabajo del intérprete, puesto que desde un punto de vista lógico su funcionalidad es nula.

Con el objeto de evitar este tipo de situaciones, la mayoría de los intérpretes Prolog incluyen el concepto de *variable anónima*, denotada "`_`". Este tipo de variable, simplemente es ignorada en el proceso de resolución. Así, volviendo a nuestro ejemplo del predicado `if_then_else`, podríamos reescribir el programa anterior en la forma:

```
if_then_else(P,Q,_) :- P, !, Q.
if_then_else(_,_,R) :- R.
```

Suponiendo ahora que interrogamos a una cualquiera de las dos versiones consideradas para la definición del predicado `if_the_else`, con la pregunta

```
:- if_then_else(fail,fail,true).
```

la respuesta obtenida es `true` en los dos casos. Sin embargo, el lector puede comprobar que los correspondientes árboles de resolución mostrados en la figura 7.6, aun siendo en esencia los mismos, se distinguen por el número total de variables locales manejadas, que es inferior en el caso del uso de variables anónimas. Si bien es evidente que este ejemplo es sumamente simple, con el fin de facilitar la comprensión, el lector puede imaginar fácilmente que un abuso innecesario en el uso de variables que bien pudieran ser anónimas disminuye la eficiencia del intérprete lógico tanto a nivel temporal como espacial.

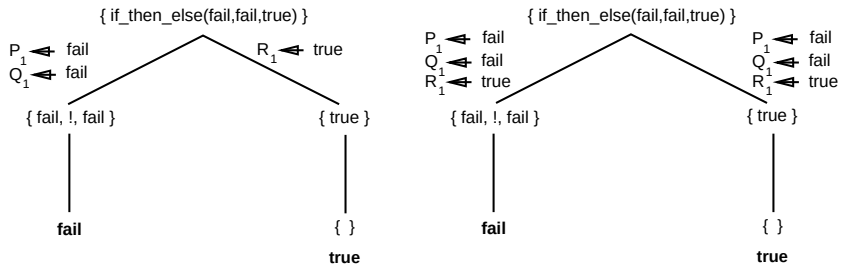


Figura 7.6: Resolución de `:- if_then_else(fail,fail,true).`, con y sin variables anónimas

El ejemplo 7.1.5 no es el único de los anteriormente mostrados que es susceptible de incluir el concepto de variable anónima. Así por ejemplo, la noción es perfectamente aplicable al ejemplo 7.1.6 que implementaba el predicado `intentar`, como se ilustra en el ejemplo que sigue.

Ejemplo 7.3.1 *En el código correspondiente al ejemplo 7.1.6, la segunda cláusula incluye un argumento cuya unificación explícita es irrelevante para el proceso de resolución. Así, las reglas a considerar pueden ser las siguientes:*

```
intentar(Objetivo) :- Objetivo,!.
intentar(_).
```

donde el argumento de la segunda cláusula es ahora una variable anónima. □

Capítulo 8

La Negación

La definición de la negación en programación lógica es, todavía hoy, un problema abierto. De hecho, constituye una de las muestras emblemáticas de la diferencia existente entre lógica estricta y programación lógica en la práctica.

8.1 El predicado `fail`

La totalidad de los dialectos Prolog incluye un predicado predefinido, en general denominado `fail`, que devuelve siempre `fail` como respuesta. La utilidad esencial de tal estructura es la de dar una posibilidad al programador de expresar el concepto de falsedad en programación lógica. En efecto, por definición nuestros programas se construyen a partir de cláusulas de Horn y en consecuencia es imposible el afirmar la falsedad de nuestras afirmaciones. El nuevo predicado, contribuye en parte a paliar esta carencia. Los ejemplos que siguen ilustran el problema.

Ejemplo 8.1.1 *Se trata de implementar un programa Prolog capaz de representar el concepto "inferior o igual a ..." en los números naturales. Para ello, podemos considerar el siguiente conjunto de cláusulas:*

```
numero_natural(0).
numero_natural(siguiente(Numero)) :- numero_natural(Numero).

inferior_o_igual(0,Numero) :- numero_natural(Numero).
inferior_o_igual(siguiente(Numero_1), siguiente(Numero_2) :-
                inferior_o_igual(Numero_1,Numero_2).
```

cuya semántica declarativa viene indicada por el predicado

`inferior_o_igual(Numero_1,Numero_2)`

que suponemos cierto si $\texttt{Numero}_1 \leq \texttt{Numero}_2$. □

Ejemplo 8.1.2 *Supongamos ahora, que deseamos implementar el concepto "inferior a ..." en los números naturales. Es obvio que gran parte del programa será en esencia común al considerado en el ejemplo 8.1.1, con una salvedad: necesitamos afirmar que el cero no es inferior a sí mismo. El predicado* `fail` *nos permitirá hacerlo. Una primera versión podría ser la dada por:*

```
numero_natural(0).
numero_natural(siguiente(Numero)) :- numero_natural(Numero).

inferior(0,0) :- fail.
inferior(0,Numero) :- numero_natural(Numero).
inferior(siguiente(Numero_1), siguiente(Numero_2)) :-
                              inferior(Numero_1,Numero_2).
```

Sin embargo esta no es todavía una implementación correcta. En efecto, si consideramos la pregunta

```
:- inferior(0,0).
```

la respuesta obtenida será `true`*, esto es, todo lo contrario de lo que pretendíamos. Para comprender la razón basta considerar el correspondiente árbol de resolución en la figura 8.1. En efecto, la presencia del* `fail` *en la resolvente de la primera rama fuerza el retroceso, y como consecuencia entramos en la alternativa representada por la segunda cláusula del predicado* `inferior`*, lo que nos lleva a obtener una respuesta afirmativa. La solución está en cortar dicha rama, para ello bastará con incluir un corte justo después de estar seguros de que los dos números a comparar son cero, esto es, justo después del literal de cabeza:*

```
numero_natural(0).
numero_natural(siguiente(Numero)) :- numero_natural(Numero).

inferior(0,0) :- !, fail.
inferior(0,Numero) :- numero_natural(Numero).
inferior(siguiente(Numero_1), siguiente(Numero_2) :-
                              inferior(Numero_1,Numero_2).
```

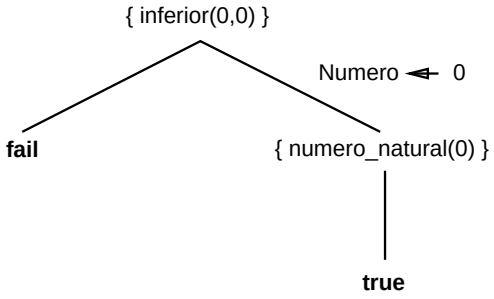


Figura 8.1: Resolución de `:- inferior(0,0).`, sin corte

El nuevo árbol de resolución se muestra en la figura 8.2. □

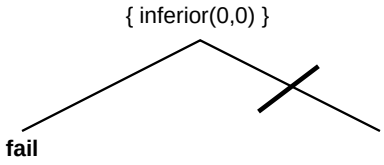


Figura 8.2: Resolución de `:- inferior(0,0).`, con corte

La combinación corte/fallo es un clásico de la programación lógica y su utilización es amplia. A este respecto, es interesante señalar que los cortes en este tipo de combinación pueden ser tanto verdes como rojos. Así, el caso expuesto en el ejemplo 8.1.2

incluye un corte rojo, puesto que hemos visto que su eliminación conlleva un cambio en el significado del programa.

Ejemplo 8.1.3 *En el siguiente ejemplo, implementaremos un predicado* `diferente(X,Y)` *que sea cierto cuando sus dos argumentos no sean unificables. Un posible programa capaz de resolver el problema sería el siguiente:*

```
diferente(X,X):-!,fail.
diferente(_,_).
```

donde la primera cláusula asegura que en el caso de unificación de los dos argumentos del predicado, se produzca un fallo sin posibilidad de retroceso sobre la segunda cláusula que sólo será accesible si los argumentos no son unificables. □

8.2 La negación por fallo

Una de las formas más extendidas de implementar la negación en lógica es mediante la utilización del predicado `fail`. En este contexto, la utilización del corte y de dicho predicado, permiten la definición de un predicado `not(P)` que es cierto cuando P no se puede demostrar y es falso en otro caso. A este tipo de negación se le suele llamar *negación por fallo* y en general no coincide con la definición de negación en lógica estricta. Además debe utilizarse con extrema prudencia, puesto que la semántica operacional de la implementación práctica de dicho predicado no coincide con la semántica declarativa de la negación por fallo. En particular, veremos que el orden en la resolución de los objetivos puede alterar las respuestas finales de un programa incluyendo esta implementación.

Ante todo, intentemos razonar cuál debe ser el proceso de construcción de nuestro predicado de negación. En principio, podríamos pensar en una definición del tipo:

```
not(P) :- P, fail.
not(P).
```

que podría corresponderse a la semántica declarativa que constituye nuestro objetivo, a saber:

"Si `P` *es cierto, entonces devolvemos* `fail`*."*
"En otro caso, `not(P)` *es cierto."*

Sin embargo nuestro programa está lejos de conseguir este objetivo. Para demostrarlo basta considerar la pregunta siguiente:

```
:- not(true).
```

En efecto, la respuesta obtenida es `true`. De hecho, justamente lo contrario de aquello que pretendíamos. Para entender la razón es necesario profundizar en la semántica operacional de nuestro programa a partir del árbol de resolución correspondiente a nuestra pregunta, tal y como lo muestra la figura 8.3.

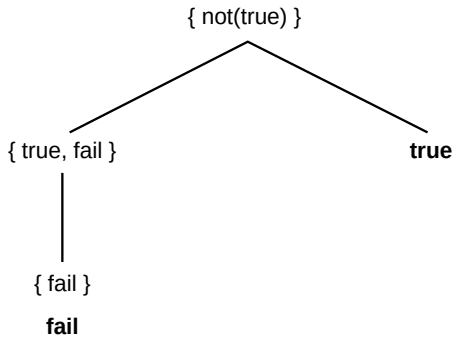


Figura 8.3: Resolución de `:- not(true).`, sin corte

La solución es, en principio, simple. Se trata de cortar la alternativa situada más a la derecha en el árbol de resolución, pero sólo cuando `P` sea cierto. Esto es, colocaremos un corte inmediatamente después de haber verificado que `P` es cierto.

```
not(P) :- P, !, fail.
not(P).
```

Ahora, una vez interrogado nuestro programa con la pregunta

```
:- not(true).
```

su comportamiento parece ser el deseado de acuerdo con nuestra semántica declarativa, tal y como se muestra en la figura 8.4.

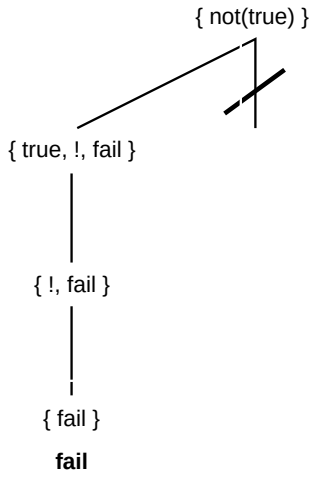


Figura 8.4: Resolución de `:- not(true).`, con corte

8.3 Anomalías de la negación por fallo

El hecho de que nuestra negación, por construcción, devuelva `fail` cuando su argumento pueda probarse, plantea problemas serios en su utilización. Más exactamente, la semántica declarativa de nuestra negación no siempre coincide con la operacional, como mostramos en los ejemplos que siguen.

Ejemplo 8.3.1 *El problema planteado es el de implementar un programa capaz de respondernos acerca de las características estéticas de un conjunto dado de personas. Para ello, consideremos el conjunto de cláusulas que sigue:*

```
igual(X,X).

guapa('Luisa').
guapa('Carmen').

fea(Persona):-not(guapa(Persona)).
```

cuya semántica operacional[1] *es la siguiente:*

[1] el predicado `igual` se utilizará en las preguntas.

"Dos términos son iguales, si unifican."

"Luisa es guapa."
"Carmen es guapa."

"Una persona es fea, si no se puede probar que es guapa."

y que en el caso de la última cláusula, tendemos a interpretar declarativamente en la forma

"Una persona es fea, si no es guapa."

Veremos que el olvido de este pequeño matiz puede provocar comportamientos imprevistos por el programador en el proceso de resolución. Consideremos las tres preguntas siguientes:

```
:- fea(X), igual(X,'Teresa').
:- fea('Teresa').
:- igual(X,'Teresa'),fea(X).
```

En principio, podríamos pensar que la semántica operacional para las tres fuese la misma:

"¿ Es Teresa fea ?."

lo cual sería cierto sólo si la resolución Prolog fuese independiente del orden de las cláusulas. En nuestro caso, ninguna de las tres preguntas coincide en su respuesta. En efecto, las respuestas son respectivamente:

```
fail
true
X ← 'Teresa'
```

Para ilustrar el porqué de este comportamiento, basta analizar los correspondientes árboles de resolución. Consideremos por ejemplo el correspondiente a la primera pregunta. Tal y como se representa en la figura 8.5, el problema se presenta porque en el momento de intentar probar el objetivo `guapa(Persona`$_1$`)` *la*

variable `Persona`$_1$ *no está instanciada, por lo que realmente no nos planteamos la veracidad de* `guapa('Teresa')` *como en un principio pretendemos. De hecho, el problema expuesto para la resolución de la pregunta se resuelve con la instanciación*

$$\texttt{Persona}_1 \leftarrow \texttt{'Luisa'}$$

que en principio nada tiene que ver con la cuestión considerada.

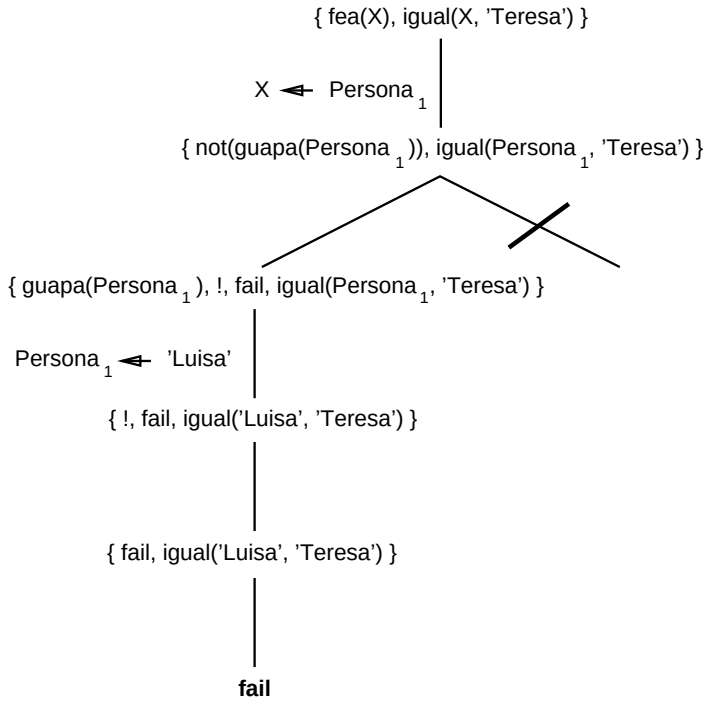


Figura 8.5: Resolución de `:- fea(X), igual(X,'Teresa').`

En el caso de la segunda pregunta, el comportamiento operacional es el expresado por el árbol de resolución de la figura 8.6, cuyo resultado es totalmente contrario al experimentado en la resolución anterior. Esencialmente, ello se debe a que ahora, la resolución del objetivo relativo a la negación

se efectúa con un predicado que no contiene variables. En esta situación, la implementación considerada para la negación por fallo se comporta como la negación lógica clásica y el resultado es el esperado a partir de nuestra base de datos, en la que Teresa no aparece como una persona guapa.

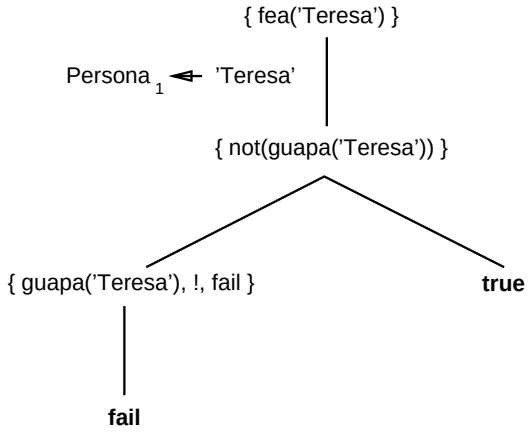


Figura 8.6: Resolución de `:- fea('Teresa').`

El último caso considerado es ilustrativo como ejemplo de dos situaciones distintas en relación al proceso de resolución de un objetivo:

1. *La influencia del orden de resolución de los objetivos en la resolvente. Para ello basta ver la forma de la pregunta en relación a la primera de las formuladas en este ejemplo.*

2. *El hecho de que, respondiendo afirmativamente a la pregunta como en el segundo caso, aquí obtengamos además como respuesta una sustitución.*

Como es habitual, es suficiente con centrar nuestra atención en el correspondiente árbol de resolución para detectar las causas del comportamiento observado. Así, a partir de la figura 8.7 el

lector puede observar que la causa de que la negación funcione como se espera es de nuevo el hecho de que su resolución se realiza con un predicado en el que no hay variables libres. En efecto, la variable en principio presente en dicho predicado queda instanciada una vez resuelto el objetivo `igual(X,'Teresa')`. *La razón por la que finalmente obtenemos una sustitución como respuesta es justamente porque en la pregunta aparecen variables libres.* □

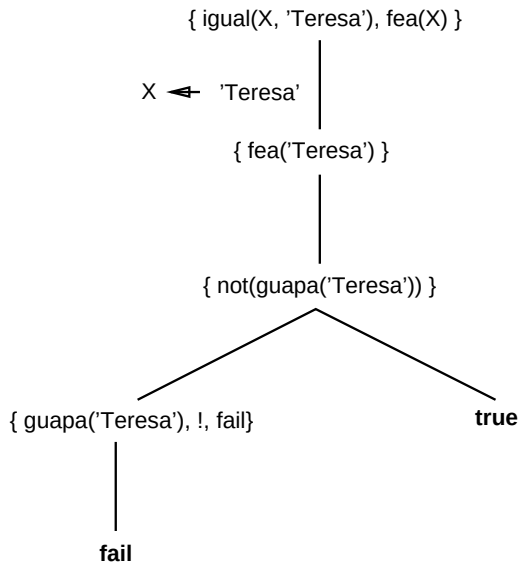


Figura 8.7: Resolución de `:- igual(X,'Teresa'), fea(X).`

Como conclusión fundamental a considerar a partir del ejemplo 8.3.1 está el hecho de que debe evitarse la resolución de negaciones cuando estas contienen variables sin instanciar. La forma en que ello pueda hacerse dependerá en cada caso del tipo de dialecto Prolog considerado[2]. En caso de no disponer

[2]por ejemplo, ciertos dialectos incluyen la posibilidad de inclusión por parte del programador de mecanismos denominados de *evaluación perezosa* que permiten no evaluar un objetivo hasta que no se cumplan ciertas condiciones en su entorno.

de ninguno de estos mecanismos, el orden de evaluación de los objetivos es determinante tal y como hemos visto en el último ejemplo 8.3.1.

Ejemplo 8.3.2 *Consideremos el programa dado por el siguiente conjunto de cláusulas:*

```
soltero(Persona) :- varon(Persona), not(casado(Persona)).

casado('Juan').

varon('Juan').
varon('Pepe').
```

cuya semántica declarativa es la dada por

> "Una persona es un soltero si no está casada y además es un varón."

El lector puede verificar fácilmente que la semántica operacional coincide con la declarativa en este caso, así para las preguntas

```
:- soltero(X).
:- soltero('Juan').
:- soltero('Pepe').
```

obtenemos las respuestas

```
X ← 'Pepe'
fail
true
```

respectivamente. Consideremos ahora un pequeño cambio en el orden de los términos del predicado `soltero`*, para obtener el nuevo programa*

```
soltero(Persona) :- not(casado(Persona)), varon(Persona).

casado('Juan').

varon('Juan').
varon('Pepe').
```

cuya semántica declarativa es la misma del anterior, puesto que esta no varía con el orden de los términos. Entonces en relación a las mismas preguntas, las respuestas respectivas obtenidas son las siguientes:

```
fail
fail
true
```

Esto es, la primera no coincide con la obtenida para el programa anterior, ni con la semántica declarativa considerada. La razón vuelve a ser de nuevo el hecho de que cuando se resuelve la negación, en el primer caso, el objetivo correspondiente contiene una variable `Persona` *sin instanciar. Observemos que ese no es el caso de la segunda y tercera preguntas, en las que además el resultado coincide con el obtenido para las mismas cuestiones en el programa anterior.* □

Capítulo 9

Control de la Evaluación

En este capítulo se va a tratar un conjunto de predicados que permiten retardar la evaluación de los objetivos hasta que cierta información que se considera necesaria esté disponible. Este tipo de técnicas se suele conocer bajo la denominación de *evaluación perezosa*. Con ello se consigue una mejora en el rendimiento, a la vez que proporciona un mecanismo que facilita la definición del comportamiento requerido.

9.1 La primitiva `freeze`

Cuando se trabaja con Prolog, en ciertas ocasiones es conveniente retrasar el proceso de evaluación hasta que se disponga de toda la información que se precisa para llevar a cabo con éxito el proceso de resolución. Esto es, se trata de asegurar que ciertas variables involucradas en el proceso estén instanciadas en el momento de proceder a la evaluación. Para ello se utiliza la primitiva `freeze`, que no se encuentra disponible en todas las implementaciones de Prolog[1].

Cuando está disponible, podemos utilizar `freeze` para asegurar la corrección de los programas en aquellos casos en los que se sabe *a priori* que el proceso de resolución de un objetivo

[1] la primitiva `freeze` está incluida en la sintaxis de Prolog II [24].

no podrá terminar con éxito si alguna de las variables no está instanciada.

Ejemplo 9.1.1 *Se trata de construir un predicado que realice la suma de dos valores. Para ello podríamos utilizar directamente el predicado no lógico* `is`*, obteniendo la siguiente definición de* `suma`*:*

```
suma(X,Y,Z) :- Z is X + Y.
```

El problema surge cuando `suma` *forma parte de la cola de una cláusula, puesto que se puede dar el caso de que el intérprete Prolog intente resolverlo antes de que las variables* `X` *y* `Y` *hayan sido instanciadas. En tal caso, se obtendrá* `fail` *como resultado de la evaluación de la suma.*

El problema tendría fácil solución si existiese algún modo de indicar que para realizar la suma es preciso esperar a que tanto `X` *como* `Y` *hayan tomado valores. Esta es la tarea encomendada a* `freeze`*. La regla:*

```
sumar(X,Y,Z) :- freeze(X,freeze(Y,suma(X,Y,Z))).
```

indica que si las variables `X` *y* `Y` *no están instanciadas se proceda a* congelar *la evaluación de* `sumar` *hasta que dichas variables se instancien durante el proceso de resolución de la cláusula en cuya cola está incluida la aparición de* `suma`*.* □

Más exactamente, cuando se debe satisfacer un objetivo de la forma `freeze(Variable,Objetivo)`, donde `Variable` es una variable y `Objetivo` es un objetivo, hay que considerar los siguientes casos:

1. Si `Variable` está instanciada, entonces se satisface `Objetivo` de la forma usual, esto es, como si la primitiva `freeze` no existiese.
2. Si `Variable` no está instanciada:
   - El objetivo completo `freeze(Variable,Objetivo)` se satisface, por lo que la evaluación de los siguientes objetivos proseguirá normalmente.

- La unificación de `Objetivo` se retrasa hasta que `Variable` sea instanciada. Hasta que esto ocurra, se dice que la variable `Variable` está *congelada*.

A continuación se muestra cómo la utilización de la primitiva `freeze` nos puede ayudar a resolver los problemas que surgen como consecuencia de las anomalías en la negación por fallo que mostramos en el capítulo 8.

Ejemplo 9.1.2 *Retomando el caso mostrado en el ejemplo 8.3.2, en el cual tratábamos de establecer un predicado que indicase si una persona estaba soltera, podemos reescribir el conjunto de reglas de la siguiente manera:*

```
soltero(Persona) :- freeze(Persona,not(casado(Persona))),
                    varon(Persona).
casado('Juan').

varon('Juan').
varon('Pepe').
```

cuya semántica declarativa continúa siendo la misma que entonces:

> "Una persona es un soltero si no está casada y además es un varón"

Sin embargo, en este caso el lector puede comprobar que la semántica operacional coincide con la declarativa independientemente del orden de los términos del predicado `soltero`*. Por tanto, al formular las preguntas*

```
:- soltero(X).
:- soltero('Juan').
:- soltero('Pepe').
```

obtendremos las respuestas

```
X ← 'Pepe'
fail
true
```

La razón de este comportamiento estriba en el hecho de que la resolución de la negación se retarda hasta el momento en que la variable `Persona` *esté instanciada, con lo cual, si en la pregunta se utiliza una variable, se resolverá primero el término correspondiente a* `varon`.□

Un caso distinto se plantea con el del ejemplo 8.3.1, puesto que al estar constituida la cola de la cláusula encabezada por `fea(Persona)` por un único término, no es posible congelarlo en espera de que el proceso de resolución instancie la variable `Persona`. En este caso la solución viene dada por la utilización de declaraciones `wait`.

9.2 La declaración `wait`

Además del mecanismo de la congelación de variables, en ciertas versiones de Prolog[2] existe la posibilidad de establecer de antemano el comportamiento del proceso de evaluación de predicados según el estado en que se encuentren las variables involucradas. Para ello se utiliza la declaración `wait`.

Los predicados que no poseen declaraciones `wait` se comportan como predicados Prolog estándar, mientras que aquellos sobre los que se ha aplicado `wait` pueden ver retardada su evaluación. Dado un predicado P con n variables, la declaración `wait` correspondiente tendría la forma `wait` P`(`$s_1, s_2, \ldots, s_n$`)`, donde los s_i pueden tomar el valor 1 ó 0. Su significado es el siguiente:

- Si s_i es 0, entonces es preciso que la variable que ocupa la posición i se encuentre instanciada en el momento de proceder a la evaluación del predicado P. Si no es este el caso, la evaluación de dicho predicado se congela hasta el momento en que el proceso de resolución proporcione un valor a dicha variable, típicamente como resultado de

[2]como en el caso de Mu-Prolog [19].

la evaluación de alguno de los restantes términos de la cláusula.

- Si s_i es 1, entonces en proceso de resolución, en lo concerniente a la variable que ocupa la posición, se realizará siguiendo el formalismo estándar de Prolog.

Ejemplo 9.2.1 *Veamos ahora cómo podemos solucionar los ploblemas planteados en el ejercicio 8.3.1 mediante la utilización de declaraciones* `wait`. *Recordemos que partíamos del conjunto de cláusulas*

```
igual(X,X).

guapa('Luisa').
guapa('Carmen').

fea(Persona) :- not(guapa(Persona)).
```

y que la semántica declarativa de la última regla era

> "Una persona es fea, si no se puede probar que es guapa"

El problema surgía al realizar las preguntas

```
:- fea(X),igual(X,'Teresa').
:- igual(X,'Teresa'),fea(X).
```

para las que obtenemos dos respuestas distintas, ya que el diferente orden de evaluación provoca variaciones en la instanciación de las variables. Concretamente, el problema surge cuando se evalúa primero el predicado `fea`, *ya que su variable no puede ser instanciada. Utilizando la declaración*

```
wait fea(0).
```

antes de definir el predicado `fea`, *retardamos la evaluación de* `fea` *hasta que su variable esté instanciada, hecho que ocurre cuando se lleva a cabo la evaluación del predicado* `igual`.□

Como se ha podido ver, tanto la primitiva `freeze` como la declaración `wait` proporcionan un medio para evitar incongruencias declarativo/operacionales, a la vez que se mantiene una estructura legible para los programas.

Parte IV

Programación Lógica

Capítulo 10

Las Listas

Si bien los lenguajes de programación lógica en general, y Prolog en particular, no han sido optimizados para el tratamiento de listas, estas constituyen un magnífico campo de experimentación para las técnicas de programación lógica, en especial para el concepto de unificación. Comenzaremos por definir lo que es una *lista*. Esencialmente, se trata de estructuras de almacenamiento de datos con dos características que las diferencian de otros tipos de estructuras:

- Una *lista* es una estructura de datos dinámica, esto es, su tamaño puede variar en el curso de la sesión de trabajo.
- Es además una estructura de tipo secuencial. Esto es, para acceder a un elemento contenido en una lista es necesario recorrer previamente todos los que le preceden.

En nuestro caso, una lista se representará por la sucesión de sus elementos separados por comas, y encerrada entre corchetes. Así por ejemplo, la lista cuyo primer elemento es **a**, cuyo segundo elemento es **b** y cuyo tercer elemento es **c**, se representará en la forma:

```
[a,b,c]
```

La implementación física de la estructura de datos lista se centra en el concepto de *célula* o *doblete*. Elemento mínimo de una lista, la célula, está compuesta de dos partes, el **car** y el **cdr**:

- El **car** de una célula se corresponde con el contenido de la dirección de memoria de dicha célula. Esto es, el **car** de una lista es su primer elemento.

- El **cdr** de una célula es el contenido de la dirección siguiente a la representada por el **car**. Esto es, el **cdr** de una lista es la lista una vez eliminado su primer elemento.

- En el caso de Prolog, la estructura de célula se representa mediante el operador | en la forma indicada en la figura 10.1, donde el símbolo `[ ]` representa el fin de lista y habitualmente se conoce con el nombre[1] de lista vacía.

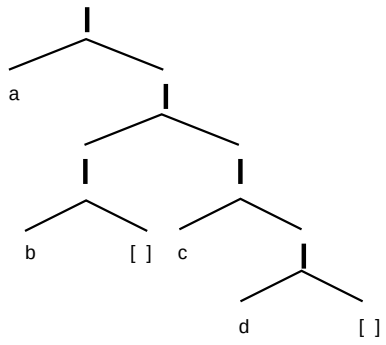


Figura 10.1: Representación interna de la lista `[a,[b],c,d]`

Ello permite que consideremos dos tipos de representación Prolog para una misma lista, una enumerando sus elementos separados por comas y otra utilizando el operador |. Así por ejemplo, la lista `[a,b,c]` puede representarse también en las formas:

```
[a|[b,c]]
[a,b|[c]]
[a,b,c|[ ]]
```

[1] totalmente inadecuado.

10.1 Definición del tipo lista

Si bien el tipo de dato lista suele estar predefinido en los intérpretes Prolog, nosotros podríamos definirlo fácilmente a partir de los datos expuestos sobre su naturaleza y la forma en que suponemos la vamos a representar notacionalmente. Una posible implementación sería la siguiente:

```
lista([ ]).
lista([_|Cdr]) :- lista(Cdr).
```

cuya semántica declarativa viene dada por:

> *"La lista vacía es una lista."*
> *"Un conjunto de elementos entre corchetes es una lista, si una vez eliminado el primero, lo que queda continúa siendo una lista."*

que como ya veremos no coincide plenamente con la semántica operacional del programa considerado.

10.2 Operaciones fundamentales sobre listas

En este apartado, comentaremos mediante ejemplos cómo se pueden implementar las operaciones fundamentales para el manejo de listas, y de hecho definiremos el núcleo de un pequeño intérprete Lisp[2], en Prolog. El objetivo es, ante todo, práctico y nos servirá para poner en evidencia algunos problemas que serán tratados más tarde con cierta profundidad. Asimismo, esperamos adentrar al lector en el manejo de las listas, una estructura de trabajo especialmente útil. Finalmente, y para aquellos lectores ya familiarizados con el manejo de lenguajes funcionales o seudofuncionales como Lisp, este apartado marcará sin duda una sutil línea de separación entre ambas concepciones de la programación declarativa.

Es en este tipo de programas, simples y potentes a la vez, donde realmente se diseña el estilo de programación propio de

[2]el lenguaje orientado a listas por excelencia.

Prolog. En efecto, no debemos centrar nuestra atención en razonamientos operacionales como en los lenguajes clásicos, al menos mientras ello sea posible, sino intentar aplicar en toda su potencia el mecanismo de unificación de las variables lógicas. Mecanismo que, no lo olvidemos, es ofrecido de forma gratuita por el intérprete. Comenzaremos por las dos funciones básicas de acceso a los elementos de una lista: los predicados `car` y `cdr`.

Ejemplo 10.2.1 *Se trata pues de definir los predicados que nos den acceso al primero y al resto de los elementos de una lista, respectivamente. Como ya hemos comentado, la consideración del mecanismo de unificación será fundamental. Tomemos las definiciones de predicado:*

```
car([Car|_], Car).
cdr([_|Cdr], Cdr).
```

cuya semántica declarativa es la que sigue:

> "El primer elemento de una lista es el **car** de la misma."
> "Todo aquello que sigue al primer elemento de una lista es el **cdr** de la misma."

Observemos que para escribir nuestro programa sólo hemos tenido en cuenta la notación utilizada para representar una lista, y sobre todo, el mecanismo de unificación que nos asegura que las respuestas al primer predicado serán instancias de la variable `Car`*, y las del segundo instancias de* `Cdr`*.*

También es curioso hacer señalar que si bien `Car` *y* `Cdr` *son variables que aparecen una sola vez por cláusula, no tendría sentido el considerarlas como variables anónimas puesto que son necesarias para obtener la respuesta deseada en cada caso.* □

En este punto es importante señalar que los predicados `car` y `cdr` no serán nunca usados tal y como han sido definidos en el ejemplo 10.2.1. En efecto, ambos conceptos son directamente aplicables mediante el concepto de unificación. En ese sentido, el anterior ejemplo tiene sólo un valor testimonial tal y como se muestra en el siguiente.

Ejemplo 10.2.2 *El problema consiste ahora en definir un predicado de la forma* `miembro(Elemento, Lista)` *que sea cierto cuando el objeto* `Elemento` *pertenezca a la lista* `Lista`*. La idea es de hecho muy simple y sirve de modelo a buen número de programas Prolog para el tratamiento de listas. Se trata en resumen de:*

1. *Ver si* `Elemento` *unifica con el car de* `Lista`*, en cuyo caso el problema tendrá una respuesta afirmativa.*

2. *En caso contrario, realizar una llamada recursiva sobre el predicado* `miembro` *con el* `cdr` *de* `Lista`*.*

Ello constituye la semántica declarativa de nuestra definición para el predicado en cuestión, que podemos traducir en forma de dos cláusulas:

```
miembro(Elemento,[Elemento|_]) :- !.
miembro(Elemento,[_|Cdr]) :- miembro(Elemento,Cdr).
```

Es importante subrayar ciertos detalles de la implementación, en relación a su semántica operacional, que además no coincide plenamente con la declarativa, aunque en este caso ello ha sido una decisión nuestra con objeto de mejorar la eficiencia, como se comenta inmediatamente:

- *La inclusión de un corte en la primera de las cláusulas tiene como objetivo evitar la duplicidad de soluciones en preguntas en las que* `Elemento` *aparece varias veces en el cuerpo de* `Lista`*. Se trata por tanto de un corte verde.*

- *La inclusión de variables anónimas tiene por único objetivo facilitar el proceso de resolución y más exactamente el de unificación.*

- *Los predicados* `car` *y* `cdr` *definidos en el ejemplo 10.2.1 no son utilizados explícitamente, si bien ambos conceptos son aplicados implícitamente mediante unificación.*

Es importante indicar que no todas las preguntas posibles obtendrán la respuesta esperada[3]*, así por ejemplo*

```
:- miembro(X,[1,2,3]).
```

obtiene como única respuesta

$$X \leftarrow \texttt{Elemento}_1 \leftarrow 1$$

cuando en principio todos los elementos de la lista considerada serían válidos. En este caso, la razón del desajuste es simple, tal y como se muestra en la figura 10.2. En efecto, el corte

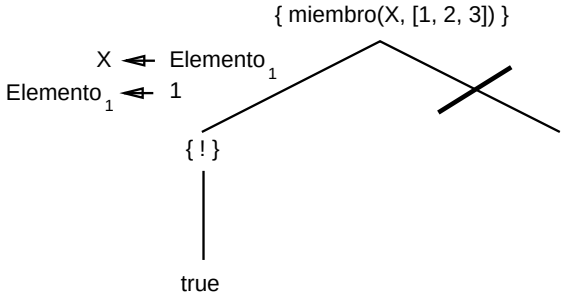


Figura 10.2: Resolución de `:- miembro(X,[1,2,3]).`, con corte

es la causa última de este comportamiento, puesto que corta toda posibilidad de búsqueda de otras soluciones. Ello podría plantear la supresión del mismo, lo cual resolvería este problema en particular, pero plantearía el de la duplicidad de soluciones con preguntas del tipo

```
:- miembro(1,[1,2,1,3,1]).
```

en caso de retroceso, tal y como habíamos comentado en un principio. Esto es, sacrificamos la completud de nuestro programa en favor de su eficiencia. Ello ilustra la confrontación, habitual, entre eficiencia y congruencia declarativo/operacional

[3] de nuevo una incongruencia declarativo/operacional.

en los programas Prolog. La decisión final sólo depende del criterio del programador en cada caso. Sin embargo, no todo son limitaciones en la aplicación práctica de la programación lógica. El mecanismo de unificación es tan simple como útil, y la potencia de cálculo que de él se deriva es con frecuencia arrolladora. Sirva como ejemplo el de la pregunta

```
:- miembro(X,Y).
```

que interroga a nuestro programa acerca del conjunto de listas que incluyen a un elemento arbitrario en su estructura. Está claro que dicho conjunto es infinito, aunque nuestras dos líneas de código son más que suficientes para obtener la respuesta adecuada:

$$\begin{cases} X \leftarrow \texttt{Elemento}_1 \\ Y \leftarrow [\texttt{Elemento}_1 \mid \_] \end{cases}$$

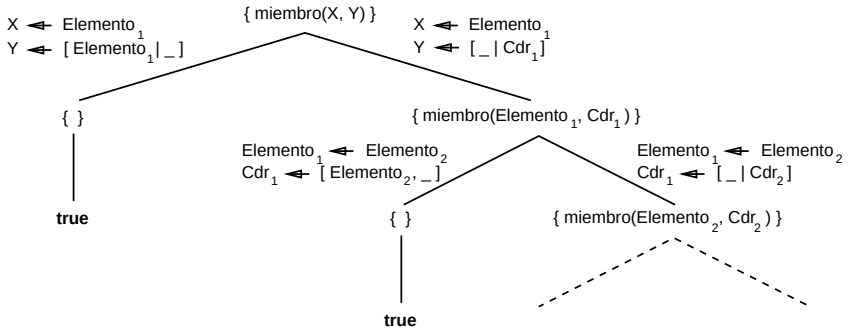


Figura 10.3: Resolución de `:- miembro(X,Y).`, sin corte

Evidentemente faltan respuestas, pero ello no es más que una consecuencia del uso del corte con objeto de acelerar la resolución. En efecto, considerando la versión sin corte

```
miembro(Elemento,[Elemento|_]).
miembro(Elemento,[_|Cdr]) :- miembro(Elemento,Cdr).
```

el conjunto de respuestas obtenido para la misma pregunta es el deseado:

$$\left\{\begin{array}{l} X \leftarrow \texttt{Elemento}_1 \\ Y \leftarrow [\texttt{Elemento}_1 \mid \_] \end{array}\right. \qquad \left\{\begin{array}{l} X \leftarrow \texttt{Elemento}_1 \\ Y \leftarrow [\_, \texttt{Elemento}_1 \mid \_] \end{array}\right.$$

$$\left\{\begin{array}{l} X \leftarrow \texttt{Elemento}_1 \\ Y \leftarrow [\_, \_, \texttt{Elemento}_1 \mid \_] \end{array}\right. \qquad \left\{\begin{array}{l} X \leftarrow \texttt{Elemento}_1 \\ Y \leftarrow [\_, \_, \_, \texttt{Elemento}_1 \mid \_] \end{array}\right.$$

...

tal y como muestra la figura 10.3. □

En este punto, sería conveniente que el lector se plantease cuántas líneas de código de un lenguaje imperativo serían necesarias para obtener la potencia de cálculo descrita en los ejemplo anteriores. La respuesta, clarificadora por sí sola, constituye una de las razones por las que, a pesar de todas las limitaciones prácticas de los lenguajes lógicos actuales, la investigación en lo que se refiere al desarrollo de nuevos esquemas de interpretación [2, 3, 4, 6, 11, 23, 25], e incluso de compilación [14, 15, 26], sigue concentrando los esfuerzos de buen número de investigadores.

Casos como el planteado en el ejemplo 10.2.2, en los que la aplicación de la unificación se traduce en el desarrollo de potencialidades desconocidas en otros formalismos, son habituales en programación lógica. De hecho, si bien es una completa exageración afirmar que los algoritmos de resolución lógica traducen el conjunto de procesos que conforman el pensamiento humano, el lector convendrá en que este estilo de programación conjuga la simplicidad de razonamiento con un incremento del espectro aplicativo.

Con el fin de subrayar la importancia del concepto de unificación en la definición de la semántica de un programa lógico, introduciremos nuevos ejemplos en los que el número de variables en juego se incremente de forma gradual. Este es un aspecto importante a recordar:

"Las semánticas declarativa y operacional de un

> *programa lógico dependen en gran parte del proceso de unificación aplicado en la resolución."*

Ejemplo 10.2.3 *El ejemplo a considerar ahora es la implementación de un predicado* `concatenar(Lista`$_1$`,Lista`$_2$`, Resultado)` *describiendo la concatenación de listas* `Lista`$_1$ *y* `Lista`$_2$ *para obtener la nueva lista* `resultado`*. La idea no es otra que la de recorrer la primera lista hasta el final, para luego concatenarle los elementos de la segunda. Formalmente, la implementación podría ser la siguiente:*

```
concatenar([],Lista,Lista).
concatenar([Car|Cdr],Lista,[Car|Resultado]) :-
                              concatenar(Cdr,Lista,Resultado).
```

cuya semántica declarativa puede ser

> "La concatenación de la lista vacía con otra lista cualquiera, es esta última."
> "La concatenación de una lista `[Car | Cdr]` con otra lista arbitraria `Lista` es el resultado de concatenar el `Car` al resultado `Resultado` previamente obtenido de la concatenación de `Cdr` con `Lista`."

También en este caso podemos constatar la potencia de cálculo imprimida por la aplicación de la unificación a nuestro programa, que describe en apenas dos líneas la concatenación de listas. Sirva como muestra el comportamiento del programa ante una pregunta de la forma

```
:- concatenar(X,Y,Z).
```

obteniendo como respuestas

$$\begin{cases} X \leftarrow [\,] \\ Y \leftarrow \texttt{Lista}_1 \\ Z \leftarrow \texttt{Lista}_1 \end{cases} \qquad \begin{cases} X \leftarrow [\texttt{Car}_1] \\ Y \leftarrow \texttt{Lista}_1 \\ Z \leftarrow [\texttt{Car}_1 \mid \texttt{Lista}_1] \end{cases}$$

$$\begin{cases} X \leftarrow [\texttt{Car}_1\texttt{Car}_2] \\ Y \leftarrow \texttt{Lista}_1 \\ Z \leftarrow [\texttt{Car}_1, \texttt{Car}_2 \mid \texttt{Lista}_1] \end{cases} \qquad \begin{cases} X \leftarrow [\texttt{Car}_1\texttt{Car}_2\texttt{Car}_3] \\ Y \leftarrow \texttt{Lista}_1 \\ Z \leftarrow [\texttt{Car}_1, \texttt{Car}_2, \texttt{Car}_3 \mid \texttt{Lista}_1] \end{cases}$$

...

De nuevo sería interesante que el lector intentase trasladar la semántica declarativa considerada, al caso de un lenguaje imperativo para constatar que el mecanismo de unificación evita de forma elegante la inclusión de estructuras de control condicionales tan corrientes en ese tipo de lenguajes. □

El objetivo ahora es el de introducir distintas técnicas de programación, haciendo hincapié en sus ventajas e inconvenientes. Esto es, una vez introducida la esencia del estilo de programación lógico, que pudiéramos definir como orientado por la unificación y naturalmente recursivo, modelaremos estas técnicas básicas justificando en cada caso su utilización. Para ello consideraremos un mismo problema que será tratado desde distintas perspectivas, dando lugar a diferentes implementaciones. Más exactamente, la cuestión planteada será la inversión de una lista en su primer nivel de profundidad. Así por ejemplo, dada la lista

$$[1, [2, 3], [4, 5], 6]$$

pretendemos invertirla en la forma

$$[6, [4, 5], [2, 3], 1]$$

El predicado será notado `invertir(Lista,Inversa)` que consideraremos se verificará si `Inversa` es la lista inversa al primer nivel de profundidad de `Lista`.

Ejemplo 10.2.4 *Trataremos en este ejemplo de dar una primera aproximación al problema de la inversión de listas. En este sentido, un algoritmo fácilmente comprensible sería el siguiente:*

```
invertir([ ],[ ]).
invertir([Car|Cdr],Inversa) :-
                  invertir(Cdr,Inverso_Cdr),
                  concatenar(Inverso_Cdr,[Car],Inversa).
```

cuya semántica declarativa viene dada por

> "El resultado de invertir la lista vacía es la misma lista vacía."
> "Para invertir una lista `[Car|Cdr]` es suficiente con invertir `Cdr` pegándole luego `Car`."

A diferencia de ejemplos anteriores como el 10.2.2 o el 10.2.3, el uso del corte en la primera cláusula no es aquí necesario para evitar la duplicación de soluciones. Ello es debido a que si una lista unifica con la lista vacía, no lo hará con una lista no vacía[4]*.*

Si bien el programa descrito parece cumplir su cometido, es ineficiente a más no poder. De hecho, estamos recorriendo la lista a invertir dos veces completamente, y varias parcialmente. Ello es debido a que el predicado recursivo `invertir`*, que recorre totalmente su primer argumento, incluye una llamada a otro predicado recursivo* `concatenar` *que como hemos visto en el ejemplo 10.2.3, recorre totalmente el primero de sus argumentos para concatenarlo con el segundo. En consecuencia, podríamos decir que nuestra primera definición de* `invertir` *es un tanto inocente y despreocupada. Tendremos que refinarla.* □

A continuación pasamos a describir la operación `aplanar` sobre estructuras de tipo lista. Fundamentalmente, se tratará de eliminar todas las sublistas de una lista dada. Así, consideraremos que el resultado de aplanar la lista

```
[1,[2],[3,[4]]]
```

es la nueva lista `[1,2,3,4]`. En este caso, utilizaremos dos predicados no lógicos, `atomic` y $\backslash ==$, que serán descritos en detalle más adelante.

Ejemplo 10.2.5 *Proponemos implementar el aplanamiento de una lista mediante el predicado* `aplanar(Lista,Resultado)` *que supondremos cierto cuando* `Resultado` *sea el resultado de aplanar la lista* `Lista`*. Para ello consideraremos la semántica declarativa siguiente:*

[4] lo cual subraya el hecho de que una lista vacía no es realmente una lista.

> "El resultado de aplanar la lista vacía es la misma lista vacía."
> "El resultado de aplanar un elemento atómico, es una lista formada por ese único elemento."
> "El resultado del aplanamiento de una lista arbitraria es el de concatenar el aplanamiento de su **car** con el aplanamiento de su `cdr`."

que formalmente podemos traducir en las tres cláusulas siguientes

```
aplanar([],[]).
aplanar(Atomo,[Atomo]) :- atomic(Atomo), Atomo \== [].
aplanar([Car|Cdr], Resultado) :-
            aplanar(Car,Car_aplanado),
            aplanar(Cdr,Cdr_aplanado),
            concatenar(Car_aplanado,Cdr_aplanado,Resultado).
```

donde hemos utilizado dos predicados no lógicos, que pasamos a describir de forma breve:
`atomic(X)` *es cierto si el argumento* `X` *es un átomo.*
`X \ == Y` *es cierto si los argumentos* `X` *y* `Y` *no están instanciados a un mismo objeto.*

Es preciso aclarar el significado, en principio contradictorio, de la segunda de las cláusulas en la definición del predicado `aplanar`. *En efecto, el hecho de que exijamos la generación de una lista en el caso del aplanamiento de un objeto atómico permite garantizar en todos los casos el sentido de la llamada a* `concatenar` *en la cláusula recursiva.* □

Otro ejemplo de función esencial en el manejo de listas es el cálculo de su longitud, que pasamos a detallar inmediatamente. En esta ocasión, y para subrayar la potencia del concepto de unificación, implementaremos una versión del predicado que calcula la longitud de una lista y que considera únicamente técnicas lógicas. A este respecto, es conveniente recordar que por el momento el concepto de operación aritmética tradicional no ha sido contemplado en este texto.

Ejemplo 10.2.6 *Vamos ahora a considerar que el predicado* `longitud(Lista,Longitud)` *es cierto cuando* `Longitud` *sea la longitud de la lista* `Lista`*. Una vez señalada la representación notacional, la semántica declarativa es en este caso especialmente simple. En efecto:*

> "La longitud de la lista vacía es cero."
> "La longitud de una lista `[Car|Cdr]` es igual al valor del número natural siguiente al de la longitud de `Cdr`."

que podemos traducir en las dos cláusulas que siguen

```
longitud([],0).
longitud([Car|Cdr],siguiente(Longitud)) :-
                          longitud(Cdr,Longitud).
```

□

Ejemplo 10.2.7 *Abordamos en este ejemplo el problema de la duplicación de listas. Se trata de implementar un predicado* `duplicar(Lista, Duplicación)` *que sea cierto sólo en el caso en el que* `Duplicación` *sea el resultado de concatenar la lista* `Lista` *consigo mismo. Una posible utilización sería*

```
:- duplicar([1,2], X).
```

para la que deberíamos obtener como respuesta

$$X \leftarrow [1,2,1,2]$$

Claramente, una posible solución pasaría por usar directamente el predicado `concatenar` *anteriormente definido en el ejemplo 10.2.3. Sin embargo, y por razones exclusivamente didácticas, no lo usaremos. En ese caso, una posible solución es la dada por el programa:*

```
duplicar(Lista,Dup) :- duplicar_aux(Lista,Lista,Dup).

duplicar_aux([],Lista,Lista).
duplicar_aux([Car|Cdr],Lista,[Car|Duplicado_Cdr]) :-
                        duplicar_aux(Cdr,Lista,Duplicado_Cdr).
```

cuya semántica declarativa es obvia. □

10.3 Operaciones sobre conjuntos

Como ejemplo recapitulatorio de esta sección dedicada al tratamiento de listas, proponemos la implementación del concepto de conjunto así como de las operaciones fundamentales que tiene asociadas: inclusión, unión, intersección y producto cartesiano.

10.3.1 El tipo conjunto

Para comenzar, debemos formalizar la definición del tipo conjunto. Ello implica dos tareas complementarias:

1. El diseño de una estructura de datos que represente al conjunto.
2. La implementación de una definición de tipo, que verifique sobre la estructura del apartado anterior si se cumplen o no las condiciones exigidas al conjunto.

En relación al primer punto, convendremos en representar un conjunto mediante una lista en la que no aparezcan elementos repetidos. La implementación exigida en 2. queda recogida en la definición del predicado

`conjunto(Lista)`

que será cierto si `Lista` verifica que no contiene elementos repetidos. Analíticamente la implementación se reduce a las tres cláusulas siguientes:

```
conjunto([]).
conjunto([Car|Cdr]) :- miembro(Car,Cdr),!,fail.
conjunto([_|Cdr]) :- conjunto(Cdr).
```

donde el predicado `miembro` es el mismo definido en el ejemplo 10.2.2 y la semántica declarativa viene dada por:

> *"La lista vacía representa al conjunto vacío."*
> *"Si el primer elemento,* `Car`*, de una lista* `[Car|Cdr]`

> *aparece repetido en su* `Cdr`*, entonces la lista no representa a un conjunto."*
> *"Si el primer elemento,* `Car`*, de una lista* `[Car|Cdr]` *no aparece repetido en su* `Cdr`*, entonces la lista representará a un conjunto si dicho* `Cdr` *no contiene elementos repetidos."*

El uso del corte en la segunda de las cláusulas se justifica por el hecho de que es necesario evitar un posible retroceso sobre la tercera cláusula cuando el `Car` no aparezca en el `Cdr`. Dicho retroceso es forzado en el momento de la ocurrencia del predicado `fail`. Evidentemente se trata de un corte rojo.

10.3.2 La unión

El siguiente paso será la implementación de la operación unión de conjuntos. Si bien dicha operación se suele considerar para un número arbitrario de argumentos, en nuestro caso y con el fin de simplificar la exposición, nos restringiremos al caso de la unión de dos conjuntos. Para ello, implementaremos el predicado

```
union(Conjunto_1,Conjunto_2,Union)
```

que será cierto si `Union` es el resultado de la unión de los conjuntos `Conjunto_1` y `Conjunto_2`. Analíticamente, consideraremos el programa dado por las tres cláusulas que siguen

```
union(Conjunto,[],Conjunto).
union(Conjunto,[Car|Cdr],Union_con_Cdr) :-
                        miembro(Car,Union_con_Cdr),
                        union(Conjunto,Cdr,Union_con_Cdr),
                        !.
union(Conjunto,[Car|Cdr],[Car|Union_con_Cdr]) :-
                        union(Conjunto,Cdr,Union_con_Cdr).
```

cuya semántica declarativa viene dada por:

> *"La unión de un conjunto cualquiera y el conjunto vacío, da como resultado el primer conjunto."*

> *"La unión de un conjunto con otro de la forma* `[Car|Cdr]`*, da como resultado el mismo que el de la unión con* `Cdr`*, en el caso en el que* `Car` *sea un elemento de la unión de dicho conjunto con* `Cdr`*."*
> *"La unión de un conjunto con otro de la forma* `[Car|Cdr]`*, da como resultado* `[Car|Union_con_Cdr]` *donde* `Union_con_Cdr` *es la unión con* `Cdr`*, en el caso en el que* `Car` *no sea un elemento de* `Union_con_Cdr`*."*

La consideración del corte en la segunda de las cláusulas viene justificada por la necesidad de cortar las posibilidades de retroceso cuando `Car` sea un elemento del resultado de la unión de `Conjunto` y `Cdr`. En este sentido, es necesario recordar que la última de las cláusulas sólo tiene sentido cuando `Car` no pertenece a dicha unión. Se trata en definitiva de un nuevo caso de corte rojo.

10.3.3 La inclusión

En relación a la operación inclusión de conjuntos, la implementación es muy sencilla y se limita a la consideración de un recorrido recursivo de la lista representando al conjunto cuya inclusión se desea verificar, donde en cada llamada recursiva se aplicará un test de pertenencia sobre los elementos del otro conjunto. Analíticamente, las dos cláusulas siguientes son suficientes:

```
inclusion([Car|Cdr],Conjunto) :- miembro(Car,Conjunto),
                                 inclusion(Cdr,Conjunto).
inclusion([],Conjunto).
```

donde `miembro` es el predicado anteriormente definido en el ejemplo 10.2.2.

10.3.4 La intersección

Para la implementación de la operación intersección de conjuntos, podríamos hacer la misma consideración que en el caso de la unión. Esto es, para simplificar la notación del

programa sólo consideraremos el caso de la intersección de dos conjuntos. Ello no supone, en cualquier caso, ningún tipo de restricción real, ni pérdida de generalidad. Para ello consideraremos el predicado

```
interseccion(Conjunto_1,Conjunto_2,Interseccion)
```

que será cierto si `Interseccion` es el resultado de la intersección de los conjuntos `Conjunto_1` y `Conjunto_2`. Desde un punto de vista analítico, consideraremos el siguiente conjunto de cláusulas

```
interseccion([Car|Cdr],Conjunto,[Car|Inter]) :-
                          miembro(Car,Conjunto),!,
                          interseccion(Cdr,Conjunto,Inter).
interseccion([_|Cdr],Conjunto,Inter) :-
                          interseccion(Cdr,Conjunto,Inter).
interseccion([],Conjunto,[]).
```

cuya semántica declarativa bien pudiera ser la siguiente:

> *"El resultado de la intersección de un conjunto de la forma* `[Car|Cdr]` *con un conjunto arbitrario, es igual a la intersección de este con* `Cdr` *más* `Car`*, cuando* `Car` *pertenece al mismo."*
> *"El resultado de la intersección de un conjunto de la forma* `[Car|Cdr]` *con un conjunto arbitrario, es igual a la intersección de este con* `Cdr`*, cuando* `Car` *no pertenece al mismo."*
> *"La intersección del vacío con cualquier conjunto da como resultado el vacío."*

Es también importante señalar la consideración de variables anónimas en el caso de la segunda cláusula. Ello se justifica por el hecho de que el **car** del primer argumento no aparece en ningún otro lugar de la cláusula. Esto indica su absoluta inutilidad al nivel del proceso de resolución, como ya habíamos comentado en la sección 7.3.

De forma análoga a implementaciones anteriores, el corte de la primera cláusula es un corte rojo cuya utilización se justifica por el hecho de que se deben evitar posibles retrocesos sobre la segunda cláusula cuando `Car` pertenezca a `Conjunto`.

10.3.5 El producto cartesiano

La última operación sobre conjuntos que vamos a tratar es el producto cartesiano de dos conjuntos. Para ello consideraremos el predicado

`cartesiano(Conjunto_1,Conjunto_2,Cartesiano)`

que será cierto cuando `Cartesiano` sea el producto cartesiano de los conjuntos `Conjunto_1` y `Conjunto_2`. Analíticamente, el programa viene dado por las cláusulas siguientes

```
cartesiano([],Conjunto,[]).
cartesiano([Car|Cdr],Conjunto,Resultado) :-
                            linea(Car,Conjunto,Linea),
                            cartesiano(Cdr,Conjunto,Resto),
                            concatenar(Linea,Resto,Resultado).
linea(Elemento,[],[]).
linea(Elemento,[Car|Cdr],[[Elemento,Car]|Resto]) :-
                            linea(Elemento,Cdr,Resto).
```

cuya semántica declarativa viene dada por:

> *"El producto cartesiano del conjunto vacío y cualquier otro, es el mismo conjunto vacío."*
> *"El producto cartesiano de un conjunto de la forma* `[Car|Cdr]` *con otro conjunto, es el resultado de concatenar el resultado del producto cartesiano de* `Cdr` *con dicho conjunto y a cuyo resultado concatenamos la línea del cartesiano correspondiente a* `Car`*."*
>
> *"La línea correspondiente a un elemento arbitrario es la vacía si el conjunto por el que realizamos el producto es el vacío."*
> *"En otro caso, el conjunto tiene la forma* `[Car|Cdr]` *y para calcular la línea resultante es suficiente con añadir el par* `[Elemento|Car]` *a la línea resultante del producto de* `Elemento` *por* `Cdr`*."*

Capítulo 11

Técnicas de Programación

El problema planteado en el anterior ejemplo 10.2.4, no es una situación aislada, sino bastante común entre los programadores inexpertos para los que la máxima

> *"Si el ordenador lo hace, es que necesariamente el algoritmo es rápido."*

es válida, lo cual supone una velocidad infinita del ordenador ... grave error. En efecto, la aplicación de la recursividad no debe ser ciega y debería estar supeditada a criterios de complejidad algorítmica, al menos cuando la aplicación a desarrollar exija un mínimo de eficiencia tanto temporal como espacial bien por la complejidad de la misma, bien por la escasez de los recursos disponibles. Ello sería objeto de un estudio más detallado, pero por el momento nos limitaremos a mostrar que no todos los problemas exigen el uso de la recursividad y que esta puede, a veces, ser eliminada si ello fuese requerido.

11.1 Los acumuladores

Una técnica bastante extendida para la eliminación de la recursividad es la del uso de los denominados *acumuladores*. Esencialmente se trata de utilizar una variable que se inicializa a un conjunto vacío para luego añadirle en cada llamada recursiva

la parte de la solución construida en dicha llamada. Ello permite en muchos casos eliminar las situaciones de doble recursividad como la comentada con `invertir` y `concatenar`, tal y como mostraremos inmediatamente.

Ejemplo 11.1.1 *Aplicaremos la técnica de los acumuladores para eliminar la llamada al predicado* `concatenar` *en la definición de* `invertir` *del ejemplo 10.2.4. Consideremos esta nueva definición del predicado* `invertir`*:*

```
invertir(Lista, Inversa) :- invertir(Lista,[ ],Inversa).
invertir([ ],Acumulador,Acumulador).
invertir([Car|Cdr],Acumulador,Inversa) :-
                    invertir(Cdr,[Car|Acumulador],Inversa).
```

cuya semántica declarativa es la siguiente:

> "Invertir una lista según el predicado `invertir/2` equivale a invertirla con el predicado `invertir/3`, con un acumulador inicial que es la lista vacía."
> "El resultado de invertir una lista vacía con un acumulador inicial dado, es este mismo acumulador."
> "El resultado de invertir una lista `[Car|Cdr]` con un acumulador inicial dado, es igual al resultado de invertir `Cdr` con un nuevo acumulador que es el resultado de añadir `Car` en cabeza del anterior acumulador."

La semántica operacional es idéntica a la del código del ejemplo 10.2.4, si bien la complejidad espacial y temporal es considerablemente inferior en la nueva versión. □

Es importante observar que el uso del acumulador es en este programa transparente para el usuario. Ello facilita su utilización y hace el código más elegante. Por otro lado, y en relación a nuestro objetivo inicial, es evidente que hemos eliminado la llamada al predicado `concatenar` y con ello los recorridos inútiles de nuestra lista argumento a invertir.

Siguiendo en la misma tónica, presentamos ahora un nuevo ejemplo, que de hecho es continuación del anterior 10.2.5 en el

que definíamos el predicado `aplanar` para las listas. En este caso, aplicaremos nuevamente la técnica de los acumuladores para eliminar la llamada a `concatenar`.

Ejemplo 11.1.2 *La técnica para implementar el aplanamiento de listas con acumuladores es similar a la aplicada en el anterior ejemplo 11.1.1, esto es, se trata de aplicar el mismo algoritmo que el utilizado en la versión que incluía a* `concatenar`*, pero sustituyéndola aquí por la utilización de los acumuladores. Así, obtenemos por traslación casi automática el siguiente conjunto de cláusulas:*

```
aplanar(Lista,Resultado) :- aplanar_ac(Lista,[],Resultado).

aplanar_ac([],Acumulador,Acumulador).
aplanar_ac(Atomo,Acumulador,[Atomo|Acumulador]) :-
                      atomic(Atomo), Atomo \== [].
aplanar_ac([Car|Cdr],Acumulador,Resultado) :-
                      aplanar_ac(Cdr,Acumulador,Cdr_aplanado),
                      aplanar_ac(Car,Cdr_aplanado,Resultado).
```

donde la primera es un simple protocolo de interfaz que inicializa el acumulador a la lista vacía.

Más exactamente, lo que estamos haciendo es actualizando el valor del acumulador en cada llamada recursiva. Para ello consideramos que `Acumulador` *es en cada momento el valor de la parte ya aplanada de la lista en ese instante de la ejecución del programa. Esto es, en la siguiente llamada recursiva el valor del acumulador será el resultado de aplanar todo el* **cdr** *de la lista a aplanar en la llamada actual. En este sentido, la tercera de las cláusulas en el programa considerado es especialmente ilustrativa del concepto de acumulación.* □

Como nuevo ejemplo ilustrativo del uso de acumuladores, consideraremos una nueva versión para la implementación del producto cartesiano de conjuntos, propuesta en la sección 10.3.

Ejemplo 11.1.3 *Para considerar una versión con acumuladores de la implementación propuesta en la sección 10.3, eliminaremos la utilización que aquel hacía del*

predicado `concatenar`*, incluyendo el uso de un acumulador que en cada momento representa la porción calculada del producto cartesiano, o del conjunto de pares en el caso del predicado* `linea`*. Analíticamente, el conjunto de cláusulas es el siguiente:*

```
cartesiano(Conj_1,Conj_2,Result) :-
              cartesiano_ac(Conj_1,Conj_2,[],Result).

cartesiano_ac([],Conj,Acumulador,Acumulador).
cartesiano_ac([Car|Cdr],Conj,Acumulador,Result) :-
              linea(Car,Conj,Acumulador,Lineas),
              cartesiano_ac(Cdr,Conj,Lineas,Result).

linea(Elem,[],Acumulador,Acumulador).
linea(Elem,[Car|Cdr],Acumulador,Lineas) :-
              linea(Elem,Cdr,[[Elem,Car]|Acumulador],Lineas).
```

□

La consideración del concepto de acumulador, no es privativa de las listas. Se trata de una técnica general, aplicable a cualquier tipo de estructura de datos. Para ilustrarlo, abordaremos un problema numérico recursivo típico: la definición del factorial de un número. En primer lugar, ilustraremos la definición recursiva del concepto para después pasar a la implementación con acumuladores. Ante todo, debemos recordar que la definición matemática de factorial de un número natural es la siguiente:

$$n! = \begin{cases} 1 & \textbf{si} \quad n = 0 \\ n * (n-1)! & \textbf{en otro caso} \end{cases}$$

Ejemplo 11.1.4 *El programa Prolog correspondiente a la definición recursiva, y natural, del factorial es el dado por las reglas:*

```
numero_natural(0).
numero_natural(siguiente(Numero)) :- numero_natural(Numero).
```

```
suma(0,Numero,Numero) :- numero_natural(Numero).
suma(siguiente(Numero_1),Numero_2,siguiente(Numero_3)) :-
                suma(Numero_1,Numero_2,Numero_3).

mult(0,Numero,0) :- numero_natural(Numero).
mult(siguiente(0),Numero,Numero) :- !, numero_natural(Numero).
mult(siguiente(Numero_1),Numero_2,Numero_3) :-
                mult(Numero_1,Numero_2,Numero_4),
                suma(Numero_4,Numero_2,Numero_3).

factorial(0, siguiente(0)).
factorial(siguiente(Numero), Factorial) :-
                factorial(Numero, Parcial),
                mult(siguiente(Numero), Parcial, Factorial).
```

El nuevo predicado introducido es `factorial(Numero, Factorial)` *que será cierto cuando* `Numero`$! =$ `Factorial`*. En este caso, la semántica declarativa se corresponde exactamente con el concepto matemático expuesto con anterioridad y no requiere una explicación a mayores.* □

Ejemplo 11.1.5 *Una posible versión del factorial con uso de acumuladores es la que sigue, que utiliza el predicado* `factorial_ac` *como estructura auxiliar de cálculo para el almacenamiento de los resultados parciales acumulados. Las cláusulas del programa están en la página 170.*

El lector no debe en este caso sentirse confundido por la utilización de la recursividad del predicado `factorial_ac`*. En efecto, se trata de una recursividad final, y por tanto el programa es iterativo en sentido estricto.* □

Otro ejemplo interesante y en la misma línea del anterior 11.1.5 es una versión del predicado `longitud` que utiliza un acumulador aritmético.

Ejemplo 11.1.6 *Una posible alternativa a la resolución del problema de la longitud de una lista, es la consideración de un acumulador aritmético. Eso si, sacrificando toda la elegancia de la programación lógica pura, mediante la introducción de*

```
numero_natural(0).
numero_natural(siguiente(Numero)) :- numero_natural(Numero).

suma(0,Numero,Numero) :- numero_natural(Numero).
suma(siguiente(Numero_1),Numero_2,siguiente(Numero_3)) :-
                suma(Numero_1,Numero_2,Numero_3).

mult(0,Numero,0) :- numero_natural(Numero).
mult(siguiente(0),Numero,Numero) :- !, numero_natural(Numero).
mult(siguiente(Numero_1),Numero_2,Numero_3) :-
                mult(Numero_1,Numero_2,Numero_4),
                suma(Numero_4,Numero_2,Numero_3).

factorial(Numero,Factorial) :-
                factorial_ac(Numero,siguiente(0),Factorial).

factorial_ac(0, Factorial, Factorial).
factorial_ac(Numero, Acumulador, Factorial) :-
                mult(Numero, Acumulador, Parcial),
                suma(X, siguiente(0), Numero),
                factorial_ac(X, Parcial, Factorial).
```

nuevos predicados no lógicos. El programa aplica el mismo formalismo notacional que el del ejemplo 10.2.6, y viene dado por las cláusulas siguientes

```
longitud([],0).
longitud([Car|Cdr],Longitud) :- longitud (Cdr,Longitud_cdr),
                                Longitud is 1 + Longitud_cdr.
```

que declarativamente podemos traducir en:

> "La longitud de la lista vacía es cero."
> "La longitud de una lista `[Car|Cdr]` es uno más la longitud de `Cdr`."

y donde hemos introducido un nuevo predicado no lógico y habitualmente predefinido en la mayoría de los intérpretes Prolog. Se trata de

```
X is Y
```

que asigna a `X` *el valor resultante de la evaluación de la expresión aritmética* `Y`. □

11.2 Estructuras de datos incompletas

La implementación del predicado `concatenar` propuesta con anterioridad en el ejemplo 10.2.3 se ha mostrado como altamente ineficiente en ejemplos como el 10.2.4, puesto que su uso implica forzosamente recorrer todos los elementos de su primera lista argumento, generando una copia de esta que se concatena a otra copia del segundo de sus argumentos.

El origen de este comportamiento tan poco afortunado no es otro que el hecho de que una lista es, tal y como se ha descrito hasta el momento en el texto, un puntero sobre el primero de sus elementos, y donde cada elemento de la misma guarda a su vez un puntero sobre el elemento siguiente. Frente a este diseño estructural que fuerza el acceso secuencial a los elementos de la lista, el usuario desearía, en ocasiones, poder acceder de forma directa al último de los elementos de la lista. Tal es el caso del predicado `concatenar`, cuya semántica, que nos servirá para

introducir el concepto de *diferencia de listas*, es un caso típico para ilustrar el uso de *estructuras de datos incompletas*.

11.2.1 Diferencias de listas

Esencialmente una *diferencia de listas* puede ser interpretada de una forma sencilla, como una lista indicada por dos punteros: uno a su primer elemento y otro al último de sus elementos. Sobre el papel, este planteamiento suprime de raíz los problemas que planteaba el uso del `concatenar` tradicional. La notación utilizada será la siguiente:

```
X \ Y
```

donde el operador \ recibe el nombre de *operador diferencia de listas*, y no suele ser incluido de forma estándar en los intérpretes Prolog. Ello implicará que en general tendremos que definirlo sistemáticamente antes de su utilización.

Una vez en este punto, estamos en condiciones de definir la semántica declarativa del nuevo predicado `concatenar_dl` que utilizará al operador diferencia de listas ya definido. El predicado se describe de forma analítica en la forma:

```
concatenar_dl(X \ Y,Y \ Z,X \ Z)
```

siendo su semántica declarativa, ilustrada por la figura 11.1, la siguiente:

> *"El resultado de concatenar la lista cuyo primer elemento es el apuntado por* `X` *y cuyo último elemento es el apuntado por* `Y`*, con la lista cuyo primer elemento es el apuntado por* `Y` *y cuyo último elemento es el apuntado por* `Z`*; es la lista cuyo primer elemento es el apuntado por* `X` *y cuyo último elemento es el apuntado por* `Z`*."*

Este nuevo formalismo permite una gran flexibilidad en la representación de listas. Así, considerando por ejemplo la estructura

```
[1,2]
```

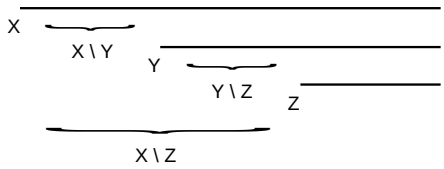


Figura 11.1: Representación gráfica de la relación entre listas tradicionales y diferencias de listas.

podemos asociarle un número infinito de expresiones equivalentes, utilizando diferencias de listas:

$$
\begin{array}{l}
[1,2] \setminus [\,] \\
[1,2,3] \setminus [3] \\
[1,2,3,4] \setminus [3,4] \\
\vdots \\
[1,2 \mid X] \setminus X
\end{array}
$$

Como nuevo ejemplo ilustrativo, consideraremos en primer lugar una implementación del predicado `invertir` cuya semántica fue ya descrita en los ejemplos 10.2.4 y 11.1.1, pero utilizando ahora la técnica de las diferencias de listas.

Ejemplo 11.2.1 *La nueva versión del predicado* `invertir`*, mediante aplicación del concepto de lista incompleta, queda definida por el siguiente conjunto de reglas:*

```
invertir(Lista,Invertir) :- invertir_dl(Lista,Invertir\[]).

invertir_dl([Car|Cdr],Invertir\Cola) :-
                    invertir_dl(Cdr,Invertir\[Car|Cola]).
invertir_dl([],Invertir\Invertir).
```

que describimos ahora de forma breve y por orden:

- *La primera cláusula es simplemente una regla de interfaz para la introducción del operador diferencia de listas.*

- *En la segunda, indicamos que para invertir la lista* `[Car|Cdr]` *basta con concatenar al inverso de* `Cdr` *el primer elemento* `Car` *de la lista considerada.*

- *Finalmente, la tercera cláusula indica simplemente que en la nueva notación, la lista vacía puede representarse mediante la diferencia de dos listas cualesquiera.*

Es importante subrayar el hecho de que es necesario haber definido previamente el operador diferencia de listas, `\`. *Para ello, utilizaremos la cláusula*

```
op(600, xfy, [\]).
```

que utiliza la declaración no lógica *predefinida* `op` *para la construcción de nuevos operadores del lenguaje. Dicha facilidad será descrita detalladamente más adelante. Aquí nos limitaremos a admitir que dicha regla equivale a la definición del operador diferencia de listas como un operador asociativo por la derecha y con una prioridad 600 en el conjunto total de operadores del lenguaje.*

Es importante observar que el algoritmo base es el mismo considerado en el ejemplo 10.2.4, simplemente se trata de una versión más económica para la implementación de la concatenación de listas. □

En este punto, es importante subrayar que la semántica declarativa de los programas que utilizan las diferencias de listas para la implementación de la concatenación de listas es la misma que la considerada en las versiones utilizando el predicado clásico `concatenar`, como ilustra el ejemplo precedente. Ello puede servir de referencia al programador inexperto en su familiarización con la nueva técnica.

Ejemplo 11.2.2 *Consideramos ahora como nuevo ejemplo, el del predicado* `aplanar` *que implementa el concepto de*

aplanamiento de una lista, y que ya había sido objeto de nuestra atención en los ejemplos anteriores 10.2.5 y 11.1.2. En este caso, la semántica permanece invariable en relación al ejemplo inicial 10.2.5, que además puede servirnos de referencia para la introducción de las diferencias de listas, como ya habíamos comentado. Analíticamente, el nuevo programa viene dado por las cláusulas que siguen:

```
aplanar(Lista,Resultado) :- aplanar_dl(Lista,Resultado\[]).

aplanar_dl([],Lista\Lista).
aplanar_dl(Atomo,[Atomo|Cdr]\Cdr) :- atomic(Atomo),
                                     Atomo \== [].
aplanar_dl([Car|Cdr],Cabeza\Dif) :-
                    aplanar_dl(Car,Cabeza\Dif_car),
                    aplanar_dl(Cdr,Dif_car\Dif).
```

que como hemos indicado, se corresponden exactamente con la versión que utilizaba `concatenar`*. A señalar la utilización del predicado predefinido* `atomic(X)`*, que se verifica cuando* `X` *está instanciada a un valor que es un átomo o un número. La figura 11.2 muestra un ejemplo de árbol de resolución para el predicado* `aplanar` *tal y como acabamos de definir. La pregunta considerada es*

```
:- aplanar([a,[b]],Resultado).
```

y la respuesta obtenida es la dada por la sustitución:

$$\texttt{Resultado} \leftarrow \texttt{Resultado}_1 \leftarrow \texttt{Cabeza}_1 \leftarrow \texttt{[a|Cdr}_2\texttt{]} \leftarrow \texttt{[a|Cabeza}_3\texttt{]}$$
$$\leftarrow \texttt{[a|Cabeza}_4\texttt{]} \leftarrow \texttt{[a|[b|Cdr}_5\texttt{]]} \leftarrow \texttt{[a|[b|Dif\_car}_3\texttt{]]}$$
$$\leftarrow \texttt{[a|[b|[]]]} = \texttt{[a,b]}$$

Evidentemente, es necesario haber definido previamente el operador $\backslash$ *mediante la declaración* `op`*, tal y como hemos hecho en el ejemplo 11.2.1.* □

Siguiendo en la misma línea, plantearemos ahora la resolución del producto cartesiano de dos conjuntos, cuestión ya considerada anteriormente en la sección 10.3 y en el ejemplo 11.1.3, pero utilizando ahora las diferencias de listas.

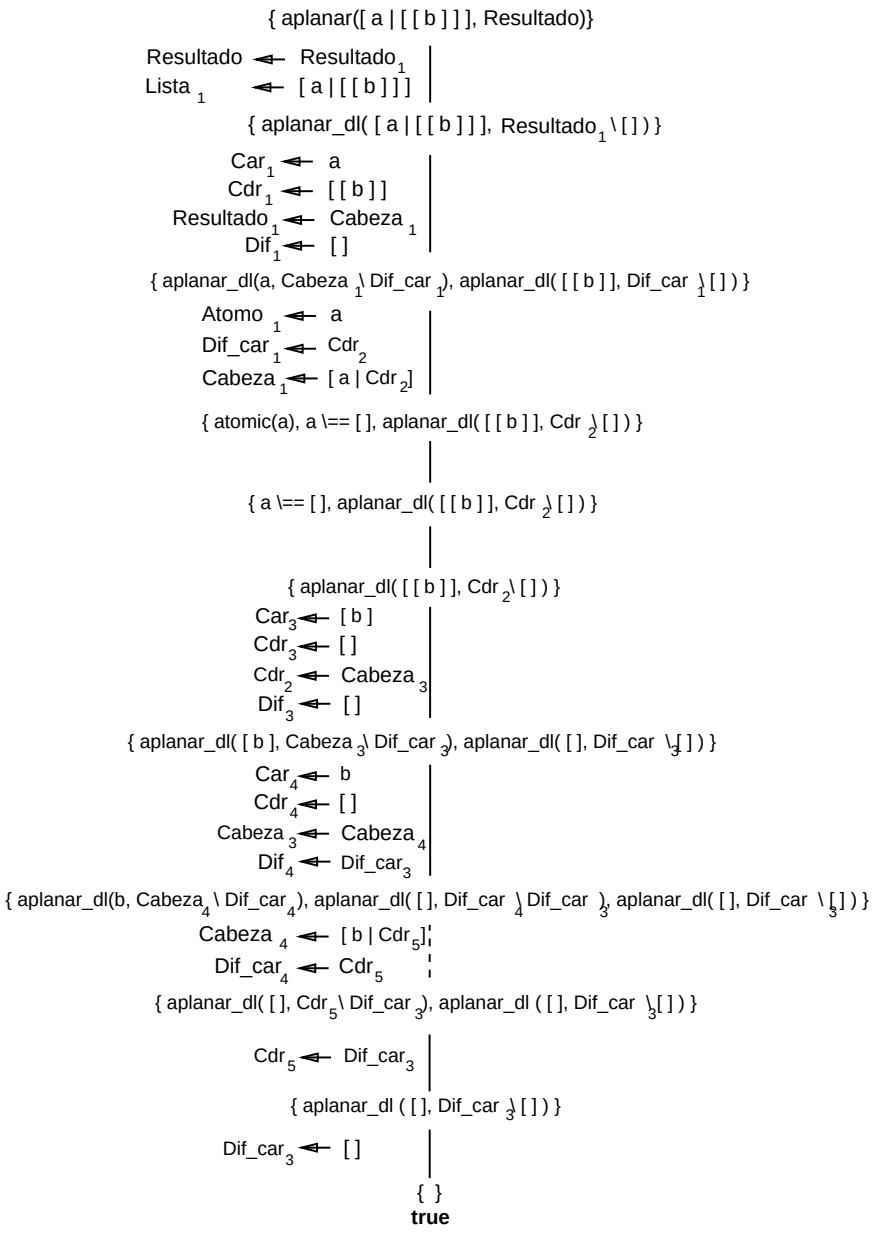


Figura 11.2: Resolución de `:- aplanar([a,[b]],Resultado).`, con diferencias de listas.

Ejemplo 11.2.3 *El proceso para la inclusión de las diferencias de listas en la resolución del producto cartesiano de dos conjuntos, es similar al aplicado en los ejemplos anteriores 11.2.1 y 11.2.2. En efecto, en este caso la semántica declarativa es la misma que la considerada en la implementación de la sección 10.3, en la cual la utilización de* `concatenar` *ha sido directamente sustituida por la consideración de diferencias de listas. Analíticamente, el programa resultante es el siguiente:*

```
cartesiano(Conj_1, Conj_2, Result) :-
                cartesiano_dl(Conj_1, Conj_2, Result \ []).

cartesiano_dl([], Conj, Dif \ Dif).
cartesiano_dl([Car|Cdr], Conj, Lineas \ Resto) :-
                linea(Car, Conj, Lineas \ Acum),
                cartesiano_dl(Cdr, Conj, Acum \ Resto).

linea(Elem, [], Dif \ Dif).
linea(Elem, [Car|Cdr], Acum \ Resto) :-
                linea(Elem, Cdr, Acum \ [[Elem,Car]|Resto]).
```

habiendo definido previamente el operador $\setminus$ *tal y como hemos hecho con anterioridad en el ejemplo 11.2.1.* □

Ejemplo 11.2.4 *Consideraremos ahora el problema de obtener a partir de una lista arbitraria, otra nueva que esté formada por los mismos elementos de la primera a los que se les ha añadido el inverso de dicha lista. Así por ejemplo, si denominamos* `concatenar_inverso` *al nuevo predicado, la respuesta ante una pregunta del tipo*

```
:- concatenar_inverso([1,2], X).
```

sería el valor de `X` *dado por:*

$$X \leftarrow [1, 2, 2, 1]$$

Evidentemente, un predicado así sería fácilmente programable a partir de los anteriormente definidos `concatenar` *e* `invertir`*. Sin embargo, lo haremos sin hacer un uso explícito de estos, aplicando simplemente el concepto de diferencia de listas. Una posible solución es la dada por las cláusulas que siguen:*

```
concatenar_inverso(Lista, Resultado) :-
            concatenar_inverso_dl(Lista, Resultado\[]).

concatenar_inverso_dl([], Lista\Lista).
concatenar_inverso_dl([Car|Cdr], [Car|Resultado]\Cola) :-
            concatenar_inverso_dl(Cdr,Resultado\[Car|Cola]).
```

Básicamente la idea es muy simple. En efecto, tomando como base el programa inicial de inversión de listas del ejemplo 11.2.1, lo que hacemos es añadir los elementos de la lista, a la lista misma en proceso de inversión en la cláusula recursiva. □

11.2.2 Diferencias de estructuras

El concepto básico aplicado en la noción de diferencias de listas es generalizable a otras estructuras. Este es el principio de las llamadas *estructuras de datos incompletas* o *diferencias de estructuras* para representar los resultados parciales de un cálculo. Uno de sus usos más representativos es la normalización de expresiones numéricas y su consiguiente aplicación a la resolución de ecuaciones. Intuitivamente, una expresión normalizada es aquella que se corresponde a la consideración de un patrón predefinido para su construcción. Ello permite la aplicación posterior de métodos que presuponen la existencia de estos patrones.

Como ejemplo ilustrativo, podemos considerar las expresiones aritméticas, a las que supondremos las prioridades establecidas por el uso explícito de paréntesis. Así por ejemplo, a la expresión

```
(a+b) + (c+d)
```

podemos asociarle la correspondiente estructura sintáctica representada en forma de árbol a la izquierda de la figura 11.3. De la misma forma, podemos considerar la expresión

```
((a+b) + c) + d
```

que se corresponde con el uso de las prioridades habituales para las sumas aritméticas, y cuya estructura sintáctica es la representada a la derecha de la figura 11.3.

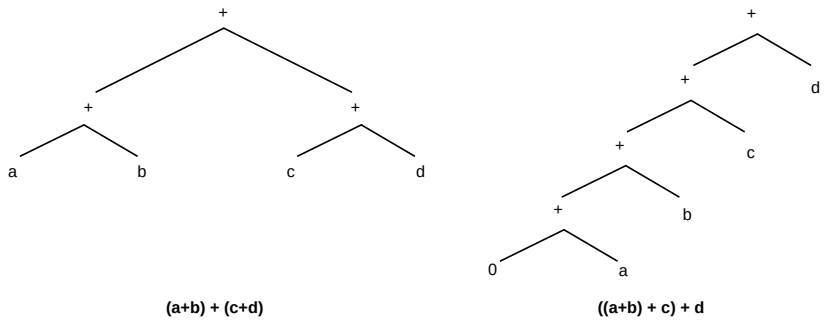


Figura 11.3: Árboles para las expresiones `(a+b)+(c+d)` y `((a+b)+c)+d`.

Frente a este tipo de estructuras sintácticas, irregulares en el sentido de que sus representaciones no responden a un patrón determinado, consideraremos la construcción de expresiones aritméticas normalizadas a la derecha[1] para las sumas. Se trata en definitiva de obtener árboles sintácticos para estas operaciones, en los que los operadores se sitúen sobre la vertiente derecha del árbol, esto es, estamos suponiendo que la operación es asociativa por la derecha. Así, en el caso de las expresiones consideradas en la figura 11.3, el árbol normalizado por la derecha es el representado en la figura 11.4.

El problema planteado es, en definitiva, el de la implementación de un programa capaz de normalizar en el sentido descrito, las sumas aritméticas. Para ello, consideraremos la introducción del operador *diferencia de sumas*, que representaremos por `++`. La semántica de la que vamos a dotar al nuevo operador será similar a aquella considerada en el caso de las diferencias de listas. Se trata en definitiva de representar una suma mediante dos punteros: uno indicando su principio y otro su final. Así por ejemplo, la suma

```
1+2
```

[1] las prioridades habituales resultan de una normalización a la izquierda.

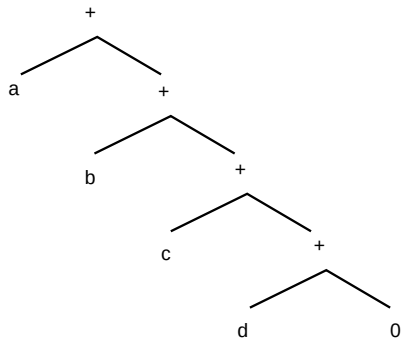


Figura 11.4: Árbol sintáctico asociativo por la derecha para `a+b+c+d`.

puede representarse de formas tan variadas como las siguientes:

$$\begin{array}{l} 1+2+3\ ++3 \\ 1+2+3+4\ ++3+4 \\ \vdots \\ 1+2+X\ ++X \end{array}$$

Particularmente interesante es la última expresión considerada, puesto que siendo `X` una variable, su valor es arbitrario. Esto es, en cada momento su valor puede asignarse a aquel que más nos convenga. Es igualmente evidente que cualquier expresión aritmética

```
X
```

puede representarse mediante

```
X ++ 0
```

Gráficamente, obtenemos una interpretación similar a la de la diferencia de listas y que en este caso representamos en la figura 11.5.

En base a la utilización del operador `++`, introducimos el predicado

```
normalizar(Suma,Norma)
```

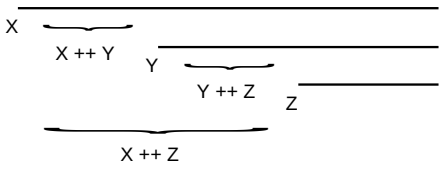


Figura 11.5: Representación gráfica de la relación entre sumas tradicionales y diferencias de sumas.

que será cierto si `Norma` es la expresión normalizada por la derecha de la suma aritmética contenida en `Suma`. Las siguientes cláusulas constituyen el programa:

```
normalizar(Suma, Norma) :- normalizar_ds(Suma, Norma ++ 0).

normalizar_ds(Suma_1 + Suma_2, Norma ++ Resto) :-
                    normalizar_ds(Suma_1, Norma ++ Resto_1),
                    normalizar_ds(Suma_2, Resto_1 ++ Resto).
normalizar_ds(Atomo, Atomo + Resto ++ Resto) :- atomic(Atomo).
```

cuya semántica declarativa con respecto al predicado `normalizar_ds` viene dada[2] por:

> *"Normalizar una suma de la forma* `Suma_1 + Suma_2` *es equivalente a normalizar el primer sumando y añadir al árbol resultante, el obtenido de la normalización del segundo sumando".*
> *"Expresa el resultado trivial* `X ++ 0 = X`*, en el caso de átomos".*

Como en el caso ya estudiado en el ejemplo 11.2.1, hemos de definir previamente el operador `++`, por ejemplo mediante

```
op(600,xfy,[++]).
```

lo cual equivale a la definición del operador `++`, como asociativo por la derecha y con una prioridad de 600 en el registro de operadores del lenguaje.

[2]la primera cláusula es simplemente de interfaz para el operador `++`.

Capítulo 12

Predicados No Lógicos

Los programas lógicos basados exclusivamente en los conceptos de unificación y resolución son sin duda elegantes, pero no es menos cierto que su ineficacia es manifiesta desde un punto de vista exclusivamente temporal, en muchos casos. Así por ejemplo, la implementación de la operación suma en el ejemplo 6.5.4 se encuentra a unos niveles de eficiencia, en cuanto a la velocidad de tratamiento se refiere, muy por debajo de las posibilidades ofrecidas por los microprocesadores de uso común mediante la aplicación de instrucciones apropiadas.

A este respecto, los distintos dialectos de Prolog incluyen una serie de predicados no lógicos predefinidos cuya función es la de implementar funcionalidades que no expresan ningún tipo de relación lógica entre objetos.

12.1 Predicados aritméticos

Entre los predicados no lógicos más importantes encontramos las interfaces para las operaciones aritméticas habituales, que nos permiten acceder directamente a la capacidad aritmética del ordenador. Evidentemente, el precio a pagar es la pérdida de generalidad que otorgan las programas puramente lógicos. Se trata de los *predicados aritméticos*.

La mayoría de los intérpretes Prolog incluyen el predicado de evaluación aritmética `is`, cuya funcionalidad es la evaluación de expresiones aritméticas representadas por los operadores habituales: $+, *, /, -$. La sintaxis correspondiente viene dada por

```
X is Y op Z
```

la cual instancia `X` al valor resultante de evaluar la expresión `Y op Z`, donde `op` es un operador aritmético binario. También podemos utilizar `is` con operadores unarios en la forma

```
X is op Z
```

o simplemente para asignar un valor numérico a una variable

```
X is 5
```

Evidentemente, cuando la expresión a la derecha del `is` no es evaluable en el momento de la ejecución[1], se produce un error. Ello plantea una diferencia fundamental con los programas lógicos puros. En efecto, en estos últimos la semántica está completamente definida y por tanto no existe el concepto de error en tiempo de ejecución, circunstancia siempre posible cuando utilizamos una interfaz directa con el procesador del ordenador. Más exactamente, nos encontramos nuevamente con los problemas clásicos derivados de la programación imperativa.

Para evitar este tipo de problemas, algunos dialectos Prolog incluyen mecanismos de control de la evaluación tales como la declaración `wait` [19] o la primitiva `freeze` [24] que pueden retardar la evaluación de un término. Sin embargo, al no tratarse de mecanismos estándar, es conveniente evitar su utilización con el fin de hacer nuestro código lo más portable posible.

En la misma línea, la mayoría de los dialectos Prolog suelen incluir predicados predefinidos en relación al resto de las

[1] esto es, contiene alguna variable sin instanciar o se le ha aplicado un valor no numérico.

operaciones aritméticas habituales; así tenemos:

`X = Y`	Cierto si `X` es igual a `Y`.
`X < Y`	Cierto si `X` es menor que `Y`.
`X =< Y`	Cierto si `X` es menor o igual que `Y`.
`X > Y`	Cierto si `X` es mayor que `Y`.
`X >= Y`	Cierto si `X` es mayor o igual que `Y`.

12.2 Entradas y salidas

Una clase importante de predicados interesantes sólo por sus efectos colaterales son aquellos encargados de la gestión de las interfaces de entrada y salida de datos. En efecto, esta labor sólo puede entenderse desde el punto de vista del sistema y de hecho, el modelo de ejecución de Prolog excluye las entradas y salidas de la componente pura del lenguaje.

El predicado fundamental de entrada es `read(X)`, el cual lee un término del flujo actual de entrada[2]. El término así leído se unifica con `X` y `read` es cierto. En cualquier caso, la instanciación de `X` a un valor exterior al programa se realiza fuera del modelo lógico considerado, puesto que a cada llamada de `read(X)` el resultado es cierto con un valor eventualmente distinto de la variable.

El predicado fundamental de salida es `write(X)`, el cual escribe el término `X` en el flujo de salida actual[3] definido por el sistema operativo subyacente.

Otro predicado interesante es `system(X)`, predicado sistema por autonomasia puesto que su misión es ejecutar sobre el sistema operativo subyacente el comando `X`. Así por ejemplo, la llamada

```
:- system("date").
```

tiene como efecto colateral, la ejecución del comando `date` en el sistema operativo, que en este caso supondremos Unix. El efecto

[2] habitualmente el teclado.
[3] habitualmente el terminal.

es el de la salida por pantalla[4] de la fecha actual, que podría ser:

```
Thu Feb 10 10:32:34 MET 1994
```

Finalmente hemos de introducir los predicados `halt` de salida de Prolog, y `trace` para visualizar el proceso de resolución. Asimismo `consult(Programa)` para cargar un programa en el intérprete y `reconsult(Programa)`, con la misma finalidad, pero actualizando los posibles predicados predefinidos. El predicado `untrace` provoca la salida del modo de traza.

12.3 Predicados de memorización

Los *predicados de memorización* salvaguardan los resultados de cálculos intermedios realizados durante la ejecución de un programa, cuando estos puedan resultar útiles para cálculos posteriores. Ello permite la construcción de programas "inteligentes" capaces de aprender de su propia experiencia. De forma similar, será posible eliminar ciertos conocimientos cuando se estime que ya no son útiles para la resolución del problema tratado.

Claramente, a este tipo de comportamientos no podemos asignarle una semántica congruente en Prolog puesto que no expresan ningún tipo de relación lógica. Por el contrario, se trata de desencadenar efectos colaterales sobre el código fuente del programa. Para ello se utilizan tres predicados fundamentalmente:

```
asserta(X)
```

que introduce la regla `X` en cabeza del conjunto de cláusulas que se encuentran en la memoria del intérprete,

```
assert(X)
```

que introduce la regla `X` al final del conjunto de cláusulas que se encuentran en la memoria del intérprete, y

```
retract(X)
```

[4] o más exactamente, por el flujo de salida actual.

que retira la primera regla que unifique con `X` en la memoria del intérprete, en un orden de búsqueda de arriba hacia abajo.

Como ejemplo ilustrativo, consideremos el caso de la función de Fibonacci, que analíticamente viene definida en la forma siguiente:

$$\text{fibonacci}(X) = \begin{cases} 0 & \textbf{si} X = 0 \\ 1 & \textbf{si} X = 1 \\ \text{fibonacci}(X-1) + \text{fibonacci}(X-2) & \textbf{en otro caso} \end{cases}$$

y que podemos implementar con el conjunto de cláusulas:

```
fib(0,0) :- !.
fib(1,1) :- !.
fib(N,R) :- N1 is N-1, fib(N1,R1),
            N2 is N-2, fib(N2,R2), R is R1+R2.
```

donde el predicado `fib(N,R)` es cierto cuando fibonacci$(N) = R$. Claramente, la semántica de la implementación es correcta, sin embargo la interpretación del código presenta ciertas características indeseables. Así, para el cálculo de fibonacci(2) será necesario conocer fibonacci(1) y fibonacci(0), valores que ya están calculados en las dos primeras cláusulas. El problema se complica cuando queremos calcular fibonacci(3), puesto que necesitaremos conocer fibonacci(2) y fibonacci(1), mientras que en principio sólo tenemos acceso directo en nuestro programa a fibonacci(1), lo cual implica el cálculo explícito de fibonacci(2). Si seguimos avanzando en el valor considerado para el argumento de la función de Fibonacci, los cálculos van a multiplicarse de forma innecesaria. Como ejemplo, consideremos el cálculo del valor de

$$\text{fibonacci}(4) = \text{fibonacci}(3) + \text{fibonacci}(2)$$

Puesto que ninguno de los valores a la derecha de la igualdad aparecen calculados directamente en el programa, tendremos que calcularlos a partir de la relación recursiva dada por la última cláusula. Ello implica la duplicación del cálculo de fibonacci(2):

1. Una vez para el cálculo de fibonacci(2).

2. Otra vez para el cálculo de fibonacci(3).

La misma situación se repite para la resolución de fibonacci(X), $X \geq 3$. Evidentemente, podríamos evitar estas duplicaciones si una vez calculado un valor que previsiblemente va a ser utilizado en un futuro, este se incluyera en la memoria del intérprete. La solución viene de la mano del programa siguiente:

```
fib(0,0) :- !.
fib(1,1) :- !.
fib(N,R) :- N1 is N-1, fib(N1,R1), N2 is N-2, fib(N2,R2),
            R is R1+R2, asserta((fib(N,R) :- !)).
```

El procedimiento anterior permite en principio una optimización de nuestra aplicación. Sin embargo, puede ocurrir que a partir de un momento dado, las cláusulas incluidas dinámicamente en la memoria del intérprete puedan no ser útiles para cálculos futuros. Así por ejemplo, una vez calculado fibonacci(4), podemos eliminar la regla correspondiente a fibonacci(2) puesto que no nos será necesaria en el cálculo de fibonacci(X), $X > 4$. El nuevo programa que incluye la eliminación de dichas reglas viene dado por el siguiente conjunto de cláusulas:

```
if_then_else(P,Q,R) :- P,!,Q.
if_then_else(P,Q,R) :- R.

fib(0,0) :- !.
fib(1,1) :- !.
fib(N,R) :- N1 is N-1, fib(N1,R1), N2 is N-2, fib(N2,R2),
            R is R1+R2, asserta((fib(N,R) :- !)),
            if_then_else(N>3,retract((fib(N2,_) :- !)),true).
```

Un problema importante en relación al uso de los predicados de memorización es que los retrocesos no tienen ningún tipo de efecto sobre ellos. La razón estriba en la naturaleza misma de los predicados en cuestión. Más claramente, una vez hemos introducido una cláusula mediante la utilización de un predicado como `asserta`, cualquiera que sea el retroceso en el que se vea involucrado el proceso de resolución la cláusula no será borrada. Ello puede ser contraproducente en tanto que el programa queda

modificado en cualquier caso, aún cuando las relaciones que hayan establecido la oportunidad de dicha modificación hayan sido invalidadas. En consecuencia, el programador deberá tener presente esta posibilidad en todo momento para asegurar la corrección de sus programas. Una posible alternativa pasa por la redefinición de estos predicados con el fin de hacerlos sensibles a los retrocesos, tal y como ocurre con los predicados lógicos.

Ejemplo 12.3.1 *En este ejemplo mostraremos cómo el programador puede redefinir el predicado* `asserta` *con el fin de hacerlo sensible a los retrocesos. Esto es, si en el transcurso del proceso de resolución se produce un retroceso sobre el objetivo* `asserta(Clausula)` *que había determinado la inclusión de la cláusula* `Clausula` *en nuestra base de datos, el intérprete debe ser capaz de detectarlo y eliminar dicha cláusula de la memoria del mismo.*

Usaremos la notación `assertb` *para designar al nuevo predicado. Una posible implementación es la dada por las cláusulas:*

```
assertb(Clausula) :- asserta(Clausula).
assertb(Clausula) :- retract(Clausula), fail.
```

cuya semántica operacional viene dada por:

> "La primera vez que se ejecuta, `assertb(Clausula)` funciona tal y como lo haría `asserta(Clausula)`."
> "En caso de retroceso sobre el objetivo `assertb(Clausula)`, retiramos la cláusula `Cláusula` de la base de datos del intérprete y continuamos con el retroceso."

Es importante señalar que la inclusión del predicado `fail` *al final de la segunda alternativa del predicado* `assertb` *es necesaria con el objeto de conseguir que el proceso natural de retroceso del intérprete no se vea interrumpido. En efecto, si eliminamos dicho* `fail` *el resultado será que el proceso de retroceso se pare en la segunda alternativa de* `assertb` *puesto que* `retract(Clausula)` *se verificará siempre por definición*

de `retract`. *A este respecto, baste observar que en caso de haber utilizado un predicado* `asserta` *clásico, el retroceso hubiese simplemente continuado puesto que no habría alternativa posible a la ya aplicada.* □

A continuación se mostrará la utilización de los predicados de memorización mediante uno de los ejemplos clásicos de programación en inteligencia artificial: las Torres de Hanoi.

Ejemplo 12.3.2 *En el juego de las Torres de Hanoi se parte de un escenario en el que se dispone de los siguientes elementos:*

- *Tres palos en posición vertical.*
- *Un conjunto de discos de diferente diámetro. Todos ellos disponen de un orificio en la parte central mediante el cual pueden ser insertados en cualquiera de los palos.*

La dinámica del juego consiste en mover un determinado número de discos desde el palo en el que están iniciamente situados a un palo de destino. El tercer palo puede utilizarse para colocar temporalmente los discos. En todo momento del juego debe satisfacerse la siguiente condición:

> "Sobre un disco cualquiera solamente pueden colocarse discos de menor diámetro."

Aquí es donde reside la dificultad del juego, puesto que todos los movimientos deben realizarse de tal forma que nunca sea necesario situar un disco sobre otro de inferior tamaño. Como elemento ilustrativo, en la figura 12.1 mostramos los movimientos necesarios para mover dos discos del palo A al palo C, utilizando el palo B como elemento intermedio. En este caso, basta realizar cuatro movimientos. Sin embargo, la complejidad del juego aumenta a medida que se incrementa el número de discos. El número de movimientos necesarios para alcanzar la solución no es proporcional al número de discos, sino que la relación es de tipo exponencial. En una primera aproximación al problema mediante Prolog, definimos el siguiente conjunto de reglas:

```
concatenar([],Lista,Lista) :- !.
concatenar([Car|Cdr],Lista,[Car|Resultado]) :-
                                concatenar(Cdr,Lista,Resultado).
hanoi(1,A,B,_,[A a B]).
hanoi(N,A,B,C,Movimientos) :-
  N > 1,
  N1 is N - 1,
  hanoi(N1,A,C,B,Movimientos_1),
  hanoi(N1,C,B,A,Movimientos_2),
  concatenar(Movimientos_1,[A a B|Movimientos_2],Movimientos).
```

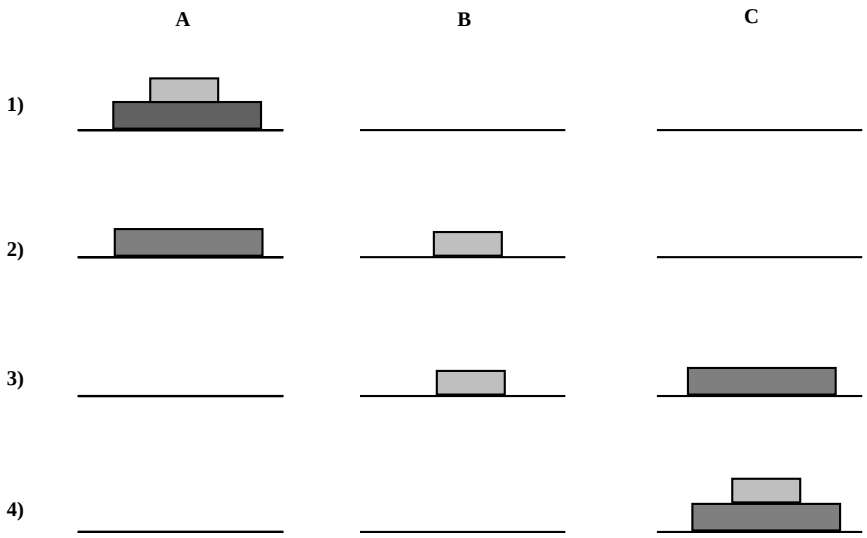


Figura 12.1: Juego de las Torres de Hanoi para dos discos.

Las dos primeras reglas definen el predicado `concatenar` *introducido con anterioridad en el ejemplo 10.2.3. Las que están encabezadas por* `hanoi` *son las que realmente se encargan de resolver el núcleo del problema, esto es, el movimiento de los discos.*

Mediante la primera regla `hanoi` *se establece directamente la solución para el caso trivial, el cual no es otro que aquel en el que se dispone de un único disco. La segunda regla es más compleja y resuelve los casos no triviales del problema. En la*

variable `Movimientos` *se van almacenando, en forma de lista, los movimientos que se deben realizar para trasladar* `N` *discos desde el palo* `A` *hacia el* `B`*, utilizando el* `C` *como palo intermedio, para lo cual es necesario realizar previamente los siguientes pasos:*

1. *Mover* `N-1` *discos de* `A` *hacia* `C` *utilizando* `B` *como palo intermedio. Con ello dejamos en* `A` *un único disco: el que estaba en la base, soportando a todos los demás.*
2. *Mover el disco situado en* `A` *a* `B`*.*
3. *Trasladar los* `N-1` *discos situados en* `C` *a* `B`*, para lo cual se puede hacer uso de* `A` *como palo intermedio.*

Para que el programa funcione correctamente es necesario declarar el operador `a`*, que se utiliza para representar los movimientos. Para ello se utiliza la declaración no lógica* `op`*:*

```
op(600,yfx,a).
```

Si se desea obtener una visualización en pantalla de los movimientos aplicados, se deben introducir en los puntos adecuados llamadas al predicado no lógico `write`*. Una posible solución es la que se muestra en el siguiente conjunto de cláusulas:*

```
concatenar([],Lista,Lista) :- !.
concatenar([Car|Cdr],Lista,[Car|Resultado]) :-
                              concatenar(Cdr,Lista,Resultado).

hanoi(1,A,B,_,Nom_A,Nom_B,_,[A a B]) :-
            write(Nom_A),write('->'),write(Nom_B),write(', ').
hanoi(N,A,B,C,Nom_A,Nom_B,Nom_C,Movimientos) :- N > 1,
  N1 is N - 1,
  hanoi(N1,A,C,B,Nom_A,Nom_C,Nom_B,Movimientos_1),
  write(Nom_A),write('->'),write(Nom_B),write(', '),
  hanoi(N1,C,B,A,Nom_C,Nom_B,Nom_A,Movimientos_2),
  concatenar(Movimientos_1,[A a B|Movimientos_2],Movimientos).
```

Estas implementaciones presentan un problema: la repetición de soluciones previamente alcanzadas, ya que para alcanzar la solución relativa al movimiento de N discos es preciso trasladar

dos veces $N-1$ discos. Evidentemente, la solución para estos dos últimos traslados es la misma. Sin embargo, el programa realiza los cálculos dos veces, una vez para cada caso. El remedio para evitar la duplicidad de cálculos consiste en incorporar al programa un mecanismo de aprendizaje que le permita recordar la solución obtenida para aquellos problemas ya resueltos.

Bajo estas consideraciones, el nuevo programa consta del siguiente conjunto de reglas, en el cual, por simples razones de claridad de texto, no se han incluido las llamadas al predicado no lógico `write`*:*

```
concatenar([],Lista,Lista) :- !.
concatenar([Car|Cdr],Lista,[Car|Resultado]) :-
                              concatenar(Cdr,Lista,Resultado).

hanoi(1,A,B,_,[A a B]).
hanoi(N,A,B,C,Movimientos) :-
  N > 1,
  N1 is N - 1,
  hanoi(N1,A,C,B,Movimientos_1),
  asserta((hanoi(N1,A,C,B,Movimientos_1) :- !)),
  hanoi(N1,C,B,A,Movimientos_2),
  concatenar(Movimientos_1,[A a B|Movimientos_2],Movimientos).
```

Por inducción, se puede demostrar que la complejidad del problema es de 2^{N-1} movimientos para N discos. Por ejemplo, en el caso de $N = 10$, el número total de movimientos utilizando esta versión es de:

$$2^{10-1} = 2^9 = 512$$

□

12.4 Predicados metalógicos

Los *predicados metalógicos* son un tipo especial de los no lógicos, cuya misión es interrogar al sistema acerca del estado actual del proceso de resolución. Para introducir al lector en la conveniencia de disponer de un conjunto tal de facilidades, reconsideraremos el problema del ejemplo 6.6.2.

Como se recordará, en esa ocasión quedó de manifiesto que un acercamiento exclusivamente lógico a determinados problemas no siempre es garantía para la obtención de los resultados deseados. Más exactamente, en aquel caso veíamos que la resolución de ecuaciones del tipo

$$X + Y = Z$$

no siempre está garantizado utilizando el acercamiento mostrado en el ejemplo 6.5.4, que consideraba sólo mecanismos lógicos para la resolución. A este respecto, el ejemplo a desarrollar podría ser una alternativa en cuanto a la técnica empleada.

Ejemplo 12.4.1 *En esta ocasión, intentaremos abordar la resolución de ecuaciones de la forma* $X + Y = Z$ *evitando la utilización de funciones lógicas. Esto es, no consideraremos ningún tipo de notación para la representación de los argumentos. Como en el caso del ejemplo 6.5.4 nos limitaremos al caso de los números naturales. Además, y con objeto de facilitar la comprensión de las ideas fundamentales, consideraremos que la ecuación contiene sólo una variable.*

La última de las hipótesis permite una simplificación notable del problema. En efecto, si dispusiésemos de un predicado capaz de indicarnos cuándo una variable está instanciada o no, nosotros podríamos preveer el comportamiento en función de dicha respuesta. Así por ejemplo, si estuviésemos en condiciones de afirmar que tanto la variable X *como la* Z *están instanciadas a algún valor, podríamos preveer que la solución a la ecuación sería el valor* $Y = Z - X$*. Un rendimiento tal, es únicamente posible mediante la utilización de un predicado metalógico. En nuestro caso particular, la solución viene de la mano del predicado predefinido*

```
nonvar(X)
```

que es cierto cuando `X` *es una variable instanciada. También utilizaremos el predicado no lógico* `is`*. El programa resultante está formado por las cláusulas siguientes:*

```
suma(X,Y,Z) :- nonvar(X), nonvar(Y), Z is X+Y.
suma(X,Y,Z) :- nonvar(X), nonvar(Z), Y is Z-X.
suma(X,Y,Z) :- nonvar(Y), nonvar(Z), X is Z-Y.
```

□

El ejemplo comentado, ilustra perfectamente la utilidad fundamental del tipo de predicados estudiado: el *test metalógico*. Más exactamente, se trata de utilizar un predicado para decidir, en función del estado actual del intérprete, qué opción de resolución debemos seguir. Si bien el conjunto de predicados metalógicos ofertados depende en general del dialecto Prolog utilizado, los más usuales son los siguientes:

`nonvar(X)`	Es cierto cuando X es una variable instanciada.
`var(X)`	Es cierto cuando X es una variable sin instanciar.
`atom(X)`	Es cierto cuando X está instanciada a un valor que es un átomo.
`number(X)`	Es cierto cuando X está instanciada a un valor que es un número.
`integer(X)`	Es cierto cuando X está instanciada a un valor que es un número entero.
`float(X)`	Es cierto cuando X está instanciada a un valor que es un número real.
`atomic(X)`	Es cierto cuando X está instanciada a un valor que es un átomo o un número.

12.5 Comentarios

Si bien los comentarios no merecen el calificativo de predicados, puesto que simplemente son ignorados por el intérprete, hemos decidido presentarlos en este capítulo aún siendo conscientes de la contradicción que ello representa. En efecto, como en la mayoría de los lenguajes de programación, este tipo de estructuras son simplemente ignoradas incluso a nivel léxico, sin

llegar en la inmensa mayoría de los casos a la fase de análisis sintáctico[5].

En lo que se refiere a su uso práctico, simplemente indicar que en un lenguaje fuertemente declarativo como Prolog, su utilización se hace aconsejable para evidenciar los posibles conflictos declarativo/operacionales de los programas. En todo caso, un programa bien comentado es un programa con garantías de continuidad.

En la mayoría de los dialectos Prolog, los comentarios suelen delimitarse mediante "/*" al principio y "*/" al final, no siendo habitualmente multilíneas. Como ejemplo sirva el siguiente:

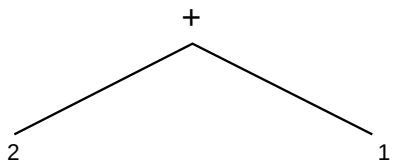
```
/* Esto es un simple comentario */
```

[5]sólo sistemas en los que el analizador sintáctico tenga asociada una aplicación que exija la recuperación íntegra del texto a partir del árbol de derivación, requieren que los comentarios tengan un tratamiento específico a este nivel. Tal es el caso del entorno de generación de lenguajes *Centaur* [9].

Capítulo 13

Los Operadores

Un operador es, en Prolog, simplemente una conveniencia notacional. Así por ejemplo, la expresión $2+1$ podría ser notada como $+(2,1)$ para representar al árbol



y no como parecería natural, al número 3. En Prolog, existen tres tipos de operadores:

Infijos, aquellos que aparecen entre sus dos argumentos.

Prefijos, aquellos que aparecen delante de su único argumento.

Sufijos, aquellos que aparecen detrás de su único argumento.

Cada operador tiene una prioridad, la cual indica el orden de evaluación de los operadores, y que en Prolog viene dada por un número entre 1 y 1200 siendo el 1 el indicativo de mayor

prioridad. Así por ejemplo, en C-Prolog la suma posee una prioridad 500 y la multiplicación una prioridad 400. Además es necesario asociar a cada operador un tipo de asociatividad. La necesidad de fijar estos dos parámetros ya había sido justificada en la sección 2.3. También se habían planteado en ese momento las dos alternativas posibles para resolver el problema. Por un lado una vía explícita caracterizada por la dinamicidad de su aplicación, y otra implícita que pasaba por modificar la gramática subyacente al lenguaje que incorporaba al operador.

En nuestro caso particular, es evidente que un cambio directo en la gramática que define el lenguaje mismo de programación no representa un acercamiento práctico. En este sentido, se impone la solución representada por la definición explícita de prioridades y asociatividades.

13.1 Definición de operadores

La discusión precedente ilustra perfectamente la necesidad de definir una prioridad y una asociatividad para cada operador considerado en el lenguaje. En consecuencia, cualquier posibilidad de introducción de nuevos operadores pasa por el cumplimiento de tres requisitos previos:

1. Definición de la representación física del operador, esto es, de su notación.

2. Declaración de su prioridad en relación a los demás operadores presentes en el lenguaje.

3. Declaración del tipo de asociatividad del operador.

En relación a la prioridad, ya hemos comentado que en Prolog viene dada por un número entero entre 1 y 1200. Respecto a la declaración de la asociatividad distinguiremos entre los casos correspondientes a los operadores binarios[1] y los unarios[2].

[1] esto es, aquellos que tienen asociados dos operandos.
[2] aquellos que tienen asociado un único operando.

13.1.1 Operadores binarios

En este caso, el lenguaje considera tres tipos distintos de operador binario:

- Operadores del tipo `xfx`.
- Operadores del tipo `xfy`.
- Operadores del tipo `yfx`.

donde:

`f` representa al operador binario, respecto al cual intentamos definir la asociatividad.

`x` indica que dicha subexpresión debe tener una prioridad estrictamente menor[3] que la del funtor `f`.

`y` indica que dicha subexpresión puede tener una prioridad menor o igual que la del funtor `f`.

considerando que la prioridad de una expresión viene dada por la prioridad de su funtor principal, esto es, del funtor que aparece en la raíz de su árbol de análisis sintáctico. En definitiva, lo que estamos diciendo es que un operador del tipo `xfy` tendrá una asociatividad por la derecha, mientras que un operador de tipo `yfx` tendrá un asociatividad por la izquierda.

En este punto, lo único que necesitamos es un predicado no lógico capaz de permitirnos la introducción de un nuevo operador en el lenguaje a partir de una notación, una prioridad y una asociatividad. Este operador tiene la sintaxis siguiente:

```
op(Prioridad, Asociatividad, Lista_de_notaciones)
```

Así por ejemplo,

```
op(500,yfx,[+])
```

declara al operador + como asociativo por la izquierda y con una prioridad de 500. Además, dado que se trata de un simple

[3] es decir, un indicativo de prioridad mayor.

predicado de interfaz con el sistema, no cabe esperar respuestas congruentes a preguntas del tipo

```
:- op(Prioridad,yfx,[+]).
```

o

```
:- op(500,Asociatividad,[+]).
```

En cualquier caso, la definición de nuevos operadores es una tarea delicada que debe tener muy en cuenta el comportamiento deseado de la nueva estructura en relación al comportamiento de los operadores ya existentes.

13.1.2 Operadores unarios

Hasta ahora hemos visto cómo se definen los operadores binarios, que son realmente los más utilizados. Sin embargo, también es posible definir en Prolog un operador unario, es decir, un operador que cuenta con un único operando. Para ello se utiliza la misma declaración `op` que en el caso de los operadores binarios. Las variaciones vienen dadas por el valor que se le pasa al segundo parámetro de `op`, mediante el cual se establece la asociatividad. Los otros dos parámetros, que se refieren a la asociatividad y a la notación, mantienen su significado.

Respecto a la asociatividad, existen cuatro tipos de operadores unarios:

- Operadores unarios del tipo `fx`.
- Operadores unarios del tipo `fy`.
- Operadores unarios del tipo `xf`.
- Operadores unarios del tipo `yf`.

Los dos primeros se refieren a operadores prefijos, mientras que los dos últimos a operadores sufijos. El significado de los tipos es análogo al de los operadores binarios, ya que:

`f` representa al operador en sí mismo. Si aparece a la izquierda indica que el operador es prefijo. Consecuentemente, si está situado a la derecha indica el carácter sufijo del operador.

`x` indica que dicha subexpresión debe tener una prioridad menor que la del funtor `f`.

`y` indica que dicha subexpresión debe tener una prioridad menor o igual que la del funtor `f`.

Todo operador *op* declarado de tipo `fy` o `yf` tiene carácter asociativo, por lo que es válido escribir[4]

op op ... op operando

puesto que la expresión *op operando* tiene igual prioridad que *op* y en consecuencia puede ser utilizada como operando de este último operador. En cambio, un operador *op* definido como `fx` o `xf` no puede ser utilizado asociativamente, puesto que una expresión como

op op ... op operando

no será válida al no ser *op operando* de menor prioridad que *op*, lo cual implica que no puede ser utilizada como operando de este último operador.

Como comentario, diremos que es posible definir más de un operador con el mismo nombre, siempre que sean de diferente tipo. Es lo que se conoce como *sobrecarga* de un operador, del cual se dice que está *sobrecargado*. El intérprete Prolog identifica el operador concreto que se está utilizando mediante el examen de los operandos en análisis sintáctico. La utilización de operadores sobrecargados debe restringirse en lo posible.

[4] en este caso suponemos que *op* es de tipo `fy`, es decir, un operador unario prefijo.

13.2 Operadores y potencia expresiva

La definición de operadores no debe ser considerada en nigún caso como una práctica limitada en Prolog. Lejos de eso, su introducción facilita la lectura declarativa de los programas y por tanto su comprensión y mantenibilidad.

Ejemplo 13.2.1 *En este ejemplo, mostraremos la utilidad de la potencia de las notaciones lógicas y en particular de los operadores. Para ello, consideraremos un problema clásico en inteligencia artificial: el* Problema de la Analogía.

Esencialmente se trata de establecer analogías entre formas geométricas. Como ejemplo concreto, a partir del conjunto de formas representadas en la figura 13.1, podemos considerar la existencia de algún tipo de relación entre ellas. Nuestro

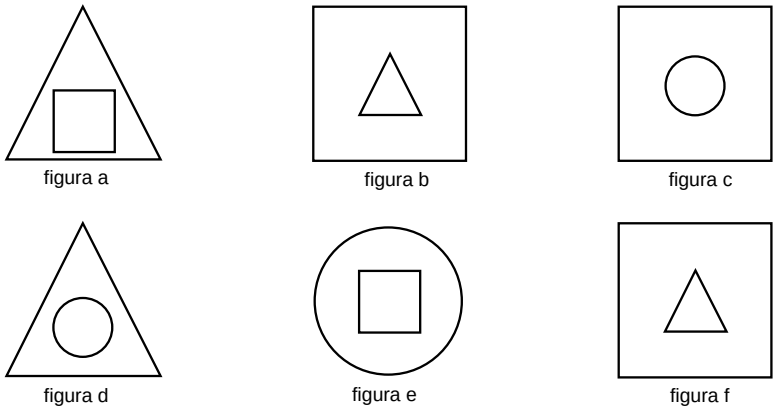


Figura 13.1: Un conjunto de figuras para el problema de la analogía

objetivo será el de diseñar un programa capaz de detectar las relaciones, indicando su naturaleza. Más exactamente, a partir de una relación entre dos objetos y un tercer objeto, se trata de encontrar el análogo a este tercer objeto según la relación establecida entre los dos primeros. Para alcanzar este objetivo, la forma de proceder es sistemática: se expresan en

forma de hechos todas las posibles relaciones y se establece una regla de deducción para encontrar las analogías. Para ello, consideraremos el predicado

```
analogia( es_a(Figura_1,Figura_2),
          es_a(Figura_3,Figura_4), Relación)
```

que será cierto si la figura `Figura_1` *es a la* `Figura_2` *lo que la figura* `Figura_3` *es a la* `Figura_4` *mediante la analogía* `Relación`*. También consideraremos el predicado auxiliar*

```
verifican(Figura_1,Figura_2,Relacion)
```

que es cierto cuando las figuras `Figura_1` *y* `Figura_2` *se relacionan mediante* `Relacion`*. Una primera solución podría venir dada por las cláusulas de la página 204.*

Como muestra del correcto comportamiento del programa, el lector puede comprobar que las respuestas a la pregunta

```
:- analogia(es_a(X,a),es_a(b,Y),Relacion).
```

vienen dadas por los triples

$$\begin{cases} \texttt{X} \leftarrow b \\ \texttt{Y} \leftarrow a \\ \texttt{Relacion} \leftarrow \texttt{inversion} \end{cases} \quad \begin{cases} \texttt{X} \leftarrow d \\ \texttt{Y} \leftarrow c \\ \texttt{Relacion} \leftarrow \texttt{contorno} \end{cases}$$

$$\begin{cases} \texttt{X} \leftarrow d \\ \texttt{Y} \leftarrow f \\ \texttt{Relacion} \leftarrow \texttt{contorno} \end{cases} \quad \begin{cases} \texttt{X} \leftarrow e \\ \texttt{Y} \leftarrow f \\ \texttt{Relacion} \leftarrow \texttt{interior} \end{cases}$$

$$\begin{cases} \texttt{X} \leftarrow f \\ \texttt{Y} \leftarrow a \\ \texttt{Relacion} \leftarrow \texttt{inversion} \end{cases}$$

Sin embargo, el programa anterior es insatisfactorio por varias razones, entre ellas:

1. *Un mecanismo de deducción de las relaciones muy poco flexible. En efecto, procediendo como hasta ahora estamos obligados a considerar de forma exhaustiva todas las*

```
analogia(es_a(X,Y),es_a(Z,W),Relacion) :-
                              verifican(X,Y,Relacion),
                              verifican(Z,W,Relacion).

verifican(a,b,inversion).
verifican(a,d,contorno).
verifican(a,e,interior).
verifican(a,f,inversion).

verifican(b,a,inversion).
verifican(b,c,contorno).
verifican(b,f,contorno).
verifican(b,f,interior).
verifican(b,f,igualdad).

verifican(c,b,contorno).
verifican(c,d,interior).
verifican(c,e,inversion).
verifican(c,f,contorno).

verifican(d,a,contorno).
verifican(d,c,interior).

verifican(e,a,interior).
verifican(e,c,inversion).

verifican(f,a,inversion).
verifican(f,b,contorno).
verifican(f,b,interior).
verifican(f,b,igualdad).
verifican(f,c,contorno).
```

relaciones posibles entre todas las figuras consideradas e incluirlas en el programa en forma de hechos. Ello puede provocar fallos en el comportamiento final del programa simplemente por omisión de alguna de las relaciones.

2. *La consideración de la función* `es_a` *como tal, y no como operador infijo dificulta la lectura declarativa del programa y por tanto su comprensión.*

Figura	**Descripción**
a	`cuadrado dentro_de triangulo`
b	`triangulo dentro_de cuadrado`
c	`circulo dentro_de cuadrado`
d	`circulo dentro_de triangulo`
e	`cuadrado dentro_de circulo`
f	`triangulo dentro_de cuadrado`

Tabla 13.1: Tabla de nombres para el problema de la analogía

Para solucionar el primer punto, incluiremos la descripción de cada figura, una información que ahora vamos a utilizar para la resolución de nuestro problema. El protocolo de descripción es el propuesto en la tabla 13.1, a partir de las denominaciones ya introducidas en la figura 13.1. Declararemos por lo tanto `dentro_de` *como operador infijo mediante*

```
op(200,xfy,[dentro_de])
```

En relación a la segunda cuestión, la solución pasa por la declaración como operador infijo de `es_a`*, por ejemplo mediante*

```
op(300,xfy,[es_a])
```

Es importante observar que hemos definido `es_a` *como un operador con menor prioridad que* `dentro_de`*. Ello es imprescindible para la buena marcha del programa, puesto que*

queremos que `es_a` *se evalúe más tarde que* `dentro_de` *en las expresiones en las que aparecen juntos ambos operadores.*

Ahora ya podemos escribir nuestro nuevo programa:

```
analogia(X es_a Y, Z es_a W, Relacion) :-
          figura(X,FormaX), figura(Y,FormaY), X \== Y,
          figura(Z,FormaZ), figura(W,FormaW), Z \== W,
          verifican(FormaX,FormaY,Relacion),
          verifican(FormaZ,FormaW,Relacion).

verifican(Figura_1 dentro_de Figura_2,
          Figura_1 dentro_de Figura_2,igualdad).
verifican(Figura_1 dentro_de Figura_2,
          Figura_2 dentro_de Figura_1,inversion).
verifican(Figura_1 dentro_de Figura_2,
          Figura_1 dentro_de Figura_3,interior).
verifican(Figura_1 dentro_de Figura_2,
          Figura_3 dentro_de Figura_2,contorno).

figura(a, cuadrado dentro_de triangulo).
figura(b, triangulo dentro_de cuadrado).
figura(c, circulo dentro_de cuadrado).
figura(d, circulo dentro_de triangulo).
figura(e, cuadrado dentro_de circulo).
figura(f, triangulo dentro_de cuadrado).
```

que ante la pregunta

```
:- analogia(X es_a a, b es_a Y, Relacion).
```

proporciona las respuestas esperadas:

$$\begin{cases} \texttt{X} \leftarrow b \\ \texttt{Y} \leftarrow a \\ \texttt{Relacion} \leftarrow \texttt{inversion} \end{cases} \quad \begin{cases} \texttt{X} \leftarrow d \\ \texttt{Y} \leftarrow c \\ \texttt{Relacion} \leftarrow \texttt{contorno} \end{cases}$$

$$\begin{cases} \texttt{X} \leftarrow d \\ \texttt{Y} \leftarrow f \\ \texttt{Relacion} \leftarrow \texttt{contorno} \end{cases} \quad \begin{cases} \texttt{X} \leftarrow e \\ \texttt{Y} \leftarrow f \\ \texttt{Relacion} \leftarrow \texttt{interior} \end{cases}$$

$$\begin{cases} \texttt{X} \leftarrow f \\ \texttt{Y} \leftarrow a \\ \texttt{Relacion} \leftarrow \texttt{inversion} \end{cases}$$

□

Capítulo 14

Lógica y Análisis Sintáctico

Una vez establecidos los mecanismos fundamentales de la programación lógica, nuestro objetivo es ahora el de centrarnos en una de sus aplicaciones fundamentales: el análisis de lenguajes naturales[1]. Ello nos llevará a explotar la relación existente entre el diseño de compiladores lógicos para programas constituidos por cláusulas de Horn y el análisis sintáctico de un lenguaje de contexto libre tradicional. Este aspecto fue observado por primera vez por Colmerauer [5], quien introdujo el concepto de *gramática de cláusulas definidas*, más conocido por DCG[2] [5, 20].

Intuitivamente, las DCG's son una generalización de las gramáticas de contexto libre, pero a diferencia de estas no son simples formalismos denotacionales sino también operacionales, esto es, pueden ejecutarse directamente. Más exactamente, podemos centrar las principales diferencias en torno a las siguientes características:

- Los no terminales pueden ser términos compuestos, por lo que no están restringidos a ser simples átomos.

[1] esto es, los lenguajes de comunicación humana.
[2] por **D**efinite **C**lause **G**rammar.

- El lado derecho de las reglas puede contener, además de terminales y no terminales, llamadas a procedimientos mediante los cuales es posible expresar condiciones adicionales que se deben satisfacer para que la regla sea considerada válida. Esto es lo que habitualmente hemos denominado *objetivos*.

La utilización de DCG's constituye un potente mecanismo para la construcción de analizadores de lenguajes. En esta capacidad juega un papel muy importante la forma en que se va construyendo la estructura correspondiente al árbol sintáctico. En efecto, Prolog construye estas estructuras "por trozos", tratando como variables aquellas partes que no pueden ser especificadas en un momento dado. La estructura puede seguir siendo construida, ya que el proceso de unificación instancia las variables que han quedado libres a medida que las partes anteriormente especificadas se pueden construir.

14.1 Un acercamiento intuitivo

Introduciremos este nuevo concepto de forma práctica, con un ejemplo de programación muy simple. Se trata de implementar un reconocedor de palíndromos, o lo que es lo mismo, de palabras cuya lectura es simétrica de derecha a izquierda, y de izquierda a derecha. Algunos ejemplos podrían ser los siguientes:

abba
aba
aa

Como en la mayoría de los problemas planteados en informática, su resolución pasa por el diseño de una gramática en la que las sentencias del lenguaje generado consituyan las soluciones buscadas. Se trata pues simplemente de describir formalmente la naturaleza de estas. La siguiente gramática es perfectamente

válida para este cometido:

$$\begin{array}{lllll} (0) & P & \rightarrow & C\ P\ C \\ (1) & P & \rightarrow & C \\ (2) & P & \rightarrow & \varepsilon \\ (3) & C & \rightarrow & \textit{carácter} \end{array}$$

con el condicionante adicional de que los caracteres C de la regla (0) deben ser idénticos. Esta última exigencia es de tipo contextual y no puede ser resuelta de forma simple utilizando un formalismo de contexto libre tradicional. En nuestro caso, su resolución será gratuita por unificación.

14.1.1 Una técnica simple

Como primera alternativa de solución para el problema planteado, utilizaremos para la implementación de nuestro analizador sintáctico una función de firma `p/3`, que servirá para representar las estructuras sintácticas generadas. La base de la implementación es el predicado

```
palindromo(Arbol,Inicial,Final)
```

que será cierto si `Arbol` es el árbol sintáctico correspondiente al análisis de la cadena considerada entre los caracteres de posiciones `Inicial` y `Final`. Incluiremos además el predicado

```
caracter(X,Inicial,Final)
```

que será cierto si `X` es un símbolo carácter entre las posiciones `Inicial` y `Final` de la cadena considerada. Analíticamente el programa propuesto tiene la lectura declarativa siguiente:

> *"Una cadena es un palíndromo si el primer y el último carácter coinciden y el resto de la cadena es un palíndromo."*
> *"La cadena vacía es un palíndromo."*
> *"Un sólo carácter es por sí mismo un palíndromo."*

```
palindromo(p(C,P,C),X,W) :- caracter(C,X,Y),
                            palindromo(P,Y,Z),
                            caracter(C,Z,W).
palindromo(p(nil),X,X).
palindromo(p(C),X,Y) :- Y is X+1, caracter(C,X,Y).

caracter(a,0,1).
caracter(b,1,2).
caracter(a,2,3).
caracter(a,3,4).
caracter(a,4,5).
caracter(b,5,6).
caracter(a,6,7).
```

Es importante observar que la técnica aplicada obliga a un conocimiento exacto de las posiciones de los caracteres en la cadena de entrada, posiciones que además deben indicarse en el programa al tiempo que se define el diccionario, que en nuestro caso está representado por las siete últimas cláusulas. Ello implica que el programa es distinto para cada cadena de caracteres considerada en entrada. Además estamos obligados a conocer incluso el número de caracteres de la frase a analizar. Así, las llamadas son del tipo

```
:- palindromo(Arbol,0,7).
```

Esto es, la cadena de entrada no se incluye en la llamada sino explícitamente en el programa, lo cual supone una limitación evidente. La respuesta obtenida viene dada por:

```
Arbol ← p(a,p(b,p(a,p(a),a),b),a)
```

Un punto importante a subrayar, es el de la resolución del problema contextual planteado por la identidad de los caracteres de la regla (0) de la gramática de los palíndromos. En efecto, se ha resuelto simplemente mediante unificación en la primera de las cláusulas del programa.

14.1.2 Una técnica flexible

Está claro, por las razones expuestas anteriormente, que el método descrito no es el más apropiado desde una perspectiva de simple usuario. Como alternativa, la utilización de diferencias de listas exige únicamente la creación de un diccionario de términos, independizando el programa lógico de la cadena de entrada considerada en cada caso. Analíticamente, el conjunto de cláusulas a considerar puede ser el siguiente:

```
analiza(Arbol, F) :- palindromo(Arbol,F\[]).

palindromo(p(C,P,C),P0\P3) :- caracter(C,P0\P1),
                              palindromo(P,P1\P2),
                              caracter(C,P2\P3).
palindromo(p(nil),P0\P0).
palindromo(p(C),P0\P1) :- caracter(C,P0\P1).

caracter(a,[a|X]\X).
caracter(b,[b|X]\X).
```

donde previamente habremos definido el operador diferencia de listas tal y como se ha hecho en el ejemplo 11.2.1. Ahora, las dos últimas cláusulas definen el diccionario, pero sin referenciar de forma expresa una cadena de entrada en particular. Ante la pregunta

```
:- analiza(Arbol,[a,b,a,a,a,b,a]).
```

obtenemos la respuesta

```
Arbol ← p(a,p(b,p(a,p(a),a),b),a)
```

Esto es, la misma obtenida con la técnica anterior. Observemos que la única exigencia en relación a la introducción de la cadena de caracteres de entrada es la utilización de una lista cuyos elementos son justamente dichos caracteres. Es importante también indicar cuál es en este último caso la lectura declarativa:

> *"Tenemos un palíndromo entre las posiciones* `P0` *y* `P3` *cuyo árbol sintáctico viene dado por la función* `p(C,P,C)`*, si tenemos:*

1. *Un carácter* `C` *entre las posiciones* `P0` *y* `P1` *y cuyo árbol es* `C`.
2. *Un palíndromo entre las posiciones* `P1` *y* `P2`*, cuyo árbol viene dado por* `P`.
3. *El mismo carácter* `C` *entre las posiciones* `P2` *y* `P3`*, con un árbol asociado que también viene dado por* `C`*."*

"En todo caso, la palabra vacía es un palíndromo."
"Tenemos un palíndromo entre las posiciones `P0` *y* `P1` *con un árbol asociado* `p(C)`*, si tenemos un carácter* `C` *entre esas mismas posiciones."*

"El carácter `a` *puede constituir el primer elemento de una lista. Ello se traduce en este caso en que es una palabra del diccionario que deseamos reconocer. Esto es,* `a` *es uno de los caracteres que pueden formar parte de los palíndromos que queremos reconocer. El árbol reconocido es en este caso el mismo carácter* `a`*."*
"El carácter `b` *verifica las mismas condiciones que el carácter* `a`*."*

14.2 El problema de la recursividad por la izquierda

El problema que tratamos de abordar ahora es el de la implementación de un analizador sintáctico para las expresiones con paréntesis bien equilibrados. Como paso previo, es necesario el diseño de una gramática que genere como lenguaje a las mismas. Una primera posibilidad sería la siguiente:

$$
\begin{array}{lrl}
(0) & Frase & \rightarrow \ (\ Frase\ ) \\
(1) & Frase & \rightarrow \ Frase\ Frase \\
(2) & Frase & \rightarrow \ \varepsilon \\
(3) & Frase & \rightarrow \ \text{carácter}
\end{array}
$$

Esta gramática es interesante por varias razones, entre ellas el hecho de que pueden generarse infinitos árboles sintácticos para una cadena de entrada de longitud finita. Ello queda de manifiesto en la figura 14.1, que muestra los infinitas posibilidades para las estructuras sintácticas de salida dada la cadena de entrada `()`, donde las líneas discontinuas representan las ambigüedades, las flechas los ciclos en los árboles, y los iconos los nodos compartidos. Si bien esta característica no impide una

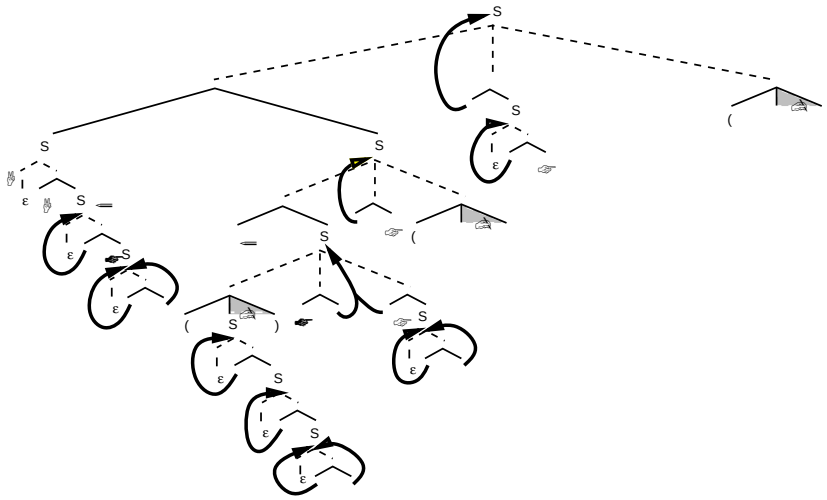


Figura 14.1: Bosque sintáctico para la cadena `()`, con infinitos árboles

implementación válida en Prolog, sí lo hace el hecho de que se trate de una gramática recursiva por la izquierda. En efecto, en la sección 6.4 habíamos hecho hincapié en el hecho de que el algoritmo de resolución construía los correspondientes árboles en profundidad. Ello impedía la terminación de los programas cuando estos eran recursivos por la izquierda. En este caso, el hecho de que la gramática de partida posea esta característica, se traduce en la generación de un programa Prolog igualmente recursivo por la izquierda y por tanto incompleto desde un punto de vista operacional, tal y como se había justificado en la

sección 6.6. En efecto, el analizador sintáctico correspondiente vendría dado por el siguiente conjunto de cláusulas:

```
analiza(F, Arbol) :- frase(F\[], Arbol).

frase(P0\P3, f(Abierto,F,Cerrado)) :-
                          parentesis_abierto(P0\P1, Abierto),
                          frase(P1\P2, F),
                          parentesis_cerrado(P2\P3, Cerrado).
frase(P0\P2, f(F1,F2)) :- frase(P0\P1, F1),
                           frase(P1\P2, F2).
frase(P0\P1, f(T)) :- terminal(P0\P1, T).
frase(P0\P0, f(nil)).

terminal([a|X]\X, a).

parentesis_abierto(['('|X]\X, '(').

parentesis_cerrado([')'|X]\X, ')').
```

el cual ante la pregunta

```
:- analiza(['(',')'], Arbol).
```

entra en un bucle infinito que provoca el agotamiento de la memoria. La solución pasa por suprimir la recursividad por la izquierda en la gramática de partida. En este punto, tenemos dos alternativas posibles:

1. Diseñar una nueva gramática no recursiva por la izquierda.
2. Transformar, de forma sistemática y automática, la gramática de la que disponemos en otra que genere el mismo lenguaje, pero en la que la recursividad por la izquierda sea eliminada.

Teniendo en cuenta lo habitual de la situación discutida, haremos una completa descripción del proceso de transformación de una gramática recursiva por la izquierda a otra directamente implementable en Prolog. Ante todo, presentamos un resultado muy simple que permite la eliminación de este tipo de recursividad cuando es directa.

Teorema 14.2.1 *Sea* $\mathcal{G} = (N, \Sigma, P, S)$ *una gramática de contexto libre, y sean* $A \to A\alpha_1 \mid \ldots \mid A\alpha_r$ *todas las A-producciones de* $\mathcal{G}$ *para las cuales A aparece como primer símbolo del lado derecho de la producción. Sean finalmente* $A \to \beta_1 \mid \ldots \mid \beta_s$ *el resto de las A-producciones. Entonces la gramática* $\mathcal{G}_1 = (N_1, \Sigma_1, P_1, S_1)$ *en la cual* $N_1 = N \cup \{B\}$, $S_1 = S$, $\Sigma_1 = \Sigma$, *y donde las A-producciones de P han sido sustituidas por:*

$$\left.\begin{array}{lcl} A & \to & \beta_i \\ A & \to & \beta_i B \end{array}\right\} 1 \leq i \leq s \qquad \left.\begin{array}{lcl} B & \to & \alpha_i \\ B & \to & \alpha_i B \end{array}\right\} 1 \leq i \leq r$$

genera el mismo lenguaje que $\mathcal{G}$. □

El método general para eliminar cualquier tipo de recursividad por la izquierda es considerablemente más complicado y consiste en trasformar la gramática en una equivalente escrita en la Forma Normal de Greibach [1]. Las gramáticas escritas siguiendo este método se caracterizan por incrementar la longitud del prefijo de terminales con la aplicación de cada regla. En consecuencia, el primer elemento de la parte derecha de toda regla debe ser un terminal. Evitando la aparición de no terminales en tal posición, se evita completamente la aparición de recursividades a la izquierda. El precio a pagar viene dado por la obtención de una gramática final muy alejada, en cuanto a su forma, de la diseñada originalmente por el programador.

A continuación se muestran los distintos conjuntos de reglas que se van obteniendo como consecuencia de la aplicación de los pasos conducentes a la normalización de la gramática de partida:

1. El primero consiste en crear un nuevo símbolo inicial que tan sólo aparezca en el lado izquierdo de una sola regla. Esto es, se trata de generar la correspondiente gramática aumentada. Con ello obtenemos las siguientes

producciones:

$$\begin{array}{lll}
(0) & S & \rightarrow \quad Frase \\
(1) & Frase & \rightarrow \quad (\ Frase\ ) \\
(2) & Frase & \rightarrow \quad Frase\ Frase \\
(3) & Frase & \rightarrow \quad \varepsilon \\
(4) & Frase & \rightarrow \quad \text{carácter}
\end{array}$$

2. En el siguiente paso se deben eliminar todas las $\varepsilon-$reglas. Si $\varepsilon \in \mathcal{L}(\mathcal{G})$, como es nuestro caso, se permite la existencia de la regla $S \rightarrow \varepsilon$. El nuevo conjunto de reglas que se obtiene es el siguiente:

$$\begin{array}{lll}
(0) & S & \rightarrow \quad Frase \\
(1) & S & \rightarrow \quad \varepsilon \\
(2) & Frase & \rightarrow \quad (\ ) \\
(3) & Frase & \rightarrow \quad (\ Frase\ ) \\
(4) & Frase & \rightarrow \quad Frase\ Frase \\
(5) & Frase & \rightarrow \quad \text{carácter}
\end{array}$$

3. A continuación se debe proceder a transformar las reglas de transmisión simple. Una vez eliminadas, la gramática queda como sigue:

$$\begin{array}{lll}
(0) & S & \rightarrow \quad \varepsilon \\
(1) & S & \rightarrow \quad (\ ) \\
(2) & S & \rightarrow \quad (\ Frase\ ) \\
(3) & S & \rightarrow \quad Frase\ Frase \\
(4) & S & \rightarrow \quad \text{carácter} \\
(5) & Frase & \rightarrow \quad (\ ) \\
(6) & Frase & \rightarrow \quad (\ Frase\ ) \\
(7) & Frase & \rightarrow \quad Frase\ Frase \\
(8) & Frase & \rightarrow \quad \text{carácter}
\end{array}$$

4. Puesto que en la gramática anterior no hay símbolos inútiles, ya que cada uno de los no terminales contribuye a la generación de las sentencias del lenguaje, podemos pasar a transformar la gramática a una forma normal intermedia denominada Forma Normal de Chomsky [1], en la que cada

regla tiene una de las formas siguientes:

$$\begin{array}{lcl} A & \to & a \\ A & \to & B\ C \\ S & \to & \varepsilon \end{array}$$

donde $A, B \in N$, $a \in \Sigma$, y S es el axioma de la gramática. El conjunto de reglas de la gramática escrita de esta forma es ahora el siguiente:

(0)	S	$\to$	ε	(6)	$Frase$	$\to$	$Pi\ T$
(1)	S	$\to$	$Pi\ Pd$	(7)	$Frase$	$\to$	$Frase\ Frase$
(2)	S	$\to$	$Pi\ T$	(8)	$Frase$	$\to$	carácter
(3)	S	$\to$	$Frase\ Frase$	(9)	Pi	$\to$	(
(4)	S	$\to$	carácter	(10)	Pd	$\to$	)
(5)	$Frase$	$\to$	$Pi\ Pd$	(11)	T	$\to$	$Frase\ Pd$

5. Para la realización de los siguientes pasos es preciso establecer un orden entre los símbolos no terminales. En nuestro caso, el orden elegido ha sido el siguiente: S, $Frase$, T, Pi, Pd. Una vez hecho esto, los pasos restantes se encaminan hacia la consecución de una gramática en la que las reglas sean de la forma

$$\begin{array}{lcl} A & \to & a\ A_1\ A_2\ \ldots\ A_n \\ A & \to & a \\ S & \to & \varepsilon \end{array}$$

donde $A \in N$, $a \in \Sigma$, y S es el axioma.

6. A continuación se procederá a eliminar la recursividad izquierda directa en $Frase$, el primer no terminal en el orden establecido para el que existe este tipo de recursividad, aplicando el teorema 14.2.1. Con ello obtenemos las siguientes reglas:

(0)	S	$\to$	ε	(8)	$Frase$	$\to$	$Pi\ Pd\ Z$
(1)	S	$\to$	$Pi\ Pd$	(9)	$Frase$	$\to$	$Pi\ T\ Z$
(2)	S	$\to$	$Pi\ T$	(10)	$Frase$	$\to$	carácter Z
(3)	S	$\to$	$Frase\ Frase$	(11)	Pi	$\to$	(
(4)	S	$\to$	carácter	(12)	Pd	$\to$	)
(5)	$Frase$	$\to$	$Pi\ Pd$	(13)	T	$\to$	$Frase\ Pd$
(6)	$Frase$	$\to$	$Pi\ T$	(14)	Z	$\to$	$Frase\ Z$
(7)	$Frase$	$\to$	carácter	(15)	Z	$\to$	$Frase$

7. Al transformar la regla 13 para evitar que el primer elemento de la parte derecha de la regla sea un no terminal con menor orden que T, obtenemos una gramática cuyas reglas son el resultado de copiar todas las de la gramática anterior excepto la 13, la cual es reemplazada por el conjunto de reglas obtenido mediante la sustitución de $Frase$ por cada una de las producciones que tienen a dicho no terminal como lado izquierdo. Puesto que en todos los casos el primer elemento de la parte derecha de las reglas encabezadas por $Frase$ tiene orden mayor que T o bien son símbolos terminales, no será preciso volver a repetir este paso para las nuevas reglas obtenidas. En el caso hipotético de que hubiésemos obtenido una regla con recursión directa sobre T[3] sería preciso aplicar de nuevo el paso anterior para eliminarla. El número de iteraciones por estos dos pasos viene acotado por el orden de T. La gramática resultante se muestra a continuación:

(0)	S	$\rightarrow$	ε				
(1)	S	$\rightarrow$	$Pi\ Pd$	(11)	Pi	$\rightarrow$	(
(2)	S	$\rightarrow$	$Pi\ T$	(12)	Pd	$\rightarrow$	)
(3)	S	$\rightarrow$	$Frase\ Frase$	(13)	T	$\rightarrow$	$Pi\ Pd\ Pd$
(4)	S	$\rightarrow$	carácter	(14)	T	$\rightarrow$	$Pi\ T\ Pd$
(5)	$Frase$	$\rightarrow$	$Pi\ Pd$	(15)	T	$\rightarrow$	carácter Pd
(6)	$Frase$	$\rightarrow$	$Pi\ T$	(16)	T	$\rightarrow$	$Pi\ Pd\ Z\ Pd$
(7)	$Frase$	$\rightarrow$	carácter	(17)	T	$\rightarrow$	$Pi\ T\ Z\ Pd$
(8)	$Frase$	$\rightarrow$	$Pi\ Pd\ Z$	(18)	T	$\rightarrow$	carácter $Z\ Pd$
(9)	$Frase$	$\rightarrow$	$Pi\ T\ Z$	(19)	Z	$\rightarrow$	$Frase\ Z$
(10)	$Frase$	$\rightarrow$	carácter Z	(20)	Z	$\rightarrow$	$Frase$

8. En el último paso sustituimos los no terminales que aparecen como primer símbolo del lado derecho de las reglas por la parte derecha de las producciones en la cual dicho no terminal aparece como lado izquierdo. Esto es, dadas

[3] este caso se daría si existiese una regla de $Frase$ que tuviese a T como primer símbolo del lado derecho, lo cual es en principio posible ya que T tiene mayor orden que $Frase$.

las reglas

$$\begin{array}{lcl} A & \rightarrow & B\ C_1,\ C_2\ \ldots\ C_n \\ B & \rightarrow & w_1 \\ & | & w_2 \\ & & \vdots \\ & | & w_m \end{array}$$

donde $A, B, C_i \in N$ y $w_i \in \epsilon$, la primera de ellas se transformará en

$$\begin{array}{lcl} A & \rightarrow & w_1\ C_1,\ C_2\ \ldots\ C_n \\ & | & w_2\ C_1,\ C_2\ \ldots\ C_n \\ & & \vdots \\ & | & w_m\ C_1,\ C_2\ \ldots\ C_n \end{array}$$

El proceso de transformación comenzará por las reglas con el lado izquierdo de mayor orden. Una vez terminado todo el proceso obtenemos una gramática en Forma Normal de Greibach, que en nuestro caso está constituida por el siguiente conjunto de producciones :

(0)	S	$\rightarrow$	ε	(16)	Pi	$\rightarrow$	(
(1)	S	$\rightarrow$	(Pd	(17)	Pd	$\rightarrow$	)
(2)	S	$\rightarrow$	(T	(18)	T	$\rightarrow$	($Pd\ Pd$
(3)	S	$\rightarrow$	carácter	(19)	T	$\rightarrow$	($T\ Pd$
(4)	S	$\rightarrow$	($Pd\ Frase$	(20)	T	$\rightarrow$	carácter Pd
(5)	S	$\rightarrow$	($T\ Frase$	(21)	T	$\rightarrow$	($Pd\ Z\ Pd$
(6)	S	$\rightarrow$	carácter $Frase$	(22)	T	$\rightarrow$	($T\ Z\ Pd$
(7)	S	$\rightarrow$	($Pd\ Z$	(23)	T	$\rightarrow$	carácter $Z\ Pd$
(8)	S	$\rightarrow$	($T\ Z$	(24)	Z	$\rightarrow$	(Pd
(9)	S	$\rightarrow$	carácter Z	(25)	Z	$\rightarrow$	(T
(10)	$Frase$	$\rightarrow$	(Pd	(26)	Z	$\rightarrow$	carácter
(11)	$Frase$	$\rightarrow$	(T	(27)	Z	$\rightarrow$	($Pd\ Z$
(12)	$Frase$	$\rightarrow$	carácter	(28)	Z	$\rightarrow$	($T\ Z$
(13)	$Frase$	$\rightarrow$	($Pd\ Z$	(29)	Z	$\rightarrow$	carácter Z
(14)	$Frase$	$\rightarrow$	($T\ Z$	(30)	Z	$\rightarrow$	($Pd\ Z\ Z$
(15)	$Frase$	$\rightarrow$	carácter Z	(31)	Z	$\rightarrow$	($T\ Z\ Z$
				(32)	Z	$\rightarrow$	carácter $Z\ Z$

Tal como se hizo anteriormente, podemos traducir directamente las producciones de la gramática en reglas de Prolog, obteniendo

el programa de las páginas 221 y 222. Esta implementación ya no produce bucles infinitos. Como ejemplo, volvamos a tomar la pregunta conflictiva mostrada anteriormente:

```
:- analiza(['(',')'],Arbol).
```

La respuesta que obtenemos es el árbol correcto correspondiente a la expresión `()`:

```
Arbol ← s((, pd()))
```

Supongamos ahora que la pregunta es la dada por la cláusula:

```
:- analiza(['(','(','a',')',')','(','a',')'],Arbol).
```

entonces la respuesta es la expresada por la sustitución

```
Arbol ← s((, t((, t(a, pd())), pd())), frase((, t(a, pd()))))
```

Obsérvese que la estructura de los árboles de las respuestas, aunque correcta, se asemeja poco a la pensada para la gramática original y resulta difícilmente reconocible[4].

[4] este es el inconveniente fundamental de aplicar un método sistemático de eliminación de la recursividad por la izquierda.

```
analiza(F,Arbol) :- s(F\[],Arbol).

s(P0\P0,s(nil)).
s(P0\P2,s(Abierto,Pd)) :-
        parentesis_abierto(P0\P1,Abierto), pd(P1\P2,Pd).
s(P0\P2,s(Abierto,T)) :-
        parentesis_abierto(P0\P1,Abierto), t(P1\P2,T).
s(P0\P1,s(Terminal)) :- terminal(P0\P1,Terminal).
s(P0\P3,s(Abierto,Pd,Frase)) :-
        parentesis_abierto(P0\P1,Abierto),
        pd(P1\P2,Pd), frase(P2\P3,Frase).
s(P0\P3,s(Abierto,T,Frase)) :-
        parentesis_abierto(P0\P1,Abierto),
        t(P1\P2,T), frase(P2\P3,Frase).
s(P0\P2,s(Terminal,Frase)) :-
        terminal(P0\P1,Terminal), frase(P1\P2,Frase).
s(P0\P3,s(Abierto,Pd,Z)) :-
        parentesis_abierto(P0\P1,Abierto),
        pd(P1\P2,Pd), z(P2\P3,Z).
s(P0\P3,s(Abierto,T,Z)) :-
        parentesis_abierto(P0\P1,Abierto),
        t(P1\P2,T), z(P2\P3,Z).
s(P0\P2,s(Terminal,Z)) :-
        terminal(P0\P1,Terminal), z(P1\P2,Z).

frase(P0\P2,frase(Abierto,Pd)) :-
            parentesis_abierto(P0\P1,Abierto), pd(P1\P2,Pd).
frase(P0\P2,frase(Abierto,T)) :-
            parentesis_abierto(P0\P1,Abierto), t(P1\P2,T).
frase(P0\P1,frase(Terminal)) :- terminal(P0\P1,Terminal).
frase(P0\P3,frase(Abierto,Pd,Z)) :-
            parentesis_abierto(P0\P1,Abierto),
            pd(P1\P2,Pd), z(P2\P3,Z).
frase(P0\P3,frase(Abierto,T,Z)) :-
            parentesis_abierto(P0\P1,Abierto),
            t(P1\P2,T), z(P2\P3,Z).

pi(P0\P1,pi(Abierto)) :- parentesis_abierto(P0\P1,Abierto).
pd(P0\P1,pd(Cerrado)) :- parentesis_cerrado(P0\P1,Cerrado).
```

```
t(P0\P3,t(Abierto,Pd,Pd)) :-
        parentesis_abierto(P0\P1,Abierto),
        pd(P1\P2,Pd), pd(P2\P3,Pd).
t(P0\P3,t(Abierto,T,Pd)) :-
        parentesis_abierto(P0\P1,Abierto),
        t(P1\P2,T), pd(P2\P3,Pd).
t(P0\P2,t(Terminal,Pd)) :-
        terminal(P0\P1,Terminal), pd(P1\P2,Pd).
t(P0\P4,t(Abierto,Pd,Z,Pd)) :-
        parentesis_abierto(P0\P1,Abierto),
        pd(P1\P2,Pd), z(P2\P3,Z), pd(P3\P4,Pd).
t(P0\P4,t(Abierto,T,Z,Pd)) :-
        parentesis_abierto(P0\P1,Abierto),
        t(P1\P2,T), z(P2\P3,Z), pd(P3\P4,Pd).
t(P0\P3,t(terminal,Z,Pd)) :-
        terminal(P0\P1,Terminal), z(P1\P2,Z), pd(P2\P3,Pd).

z(P0\P2,z(Abierto,Pd)) :-
        parentesis_abierto(P0\P1,Abierto), pd(P1\P2,Pd).
z(P0\P2,z(Abierto,T)) :-
        parentesis_abierto(P0\P1,Abierto), t(P1\P2,T).
z(P0\P1,z(Terminal)) :- terminal(P0\P1,Terminal).
z(P0\P3,z(Abierto,Pd,Z)) :-
        parentesis_abierto(P0\P1,Abierto),
        pd(P1\P2,Pd), z(P2\P3,Z).
z(P0\P3,z(Abierto,T,Z)) :-
        parentesis_abierto(P0\P1,Abierto),
        t(P1\P2,T), z(P2\P3,Z).
z(P0\P2,z(Terminal,Z)) :-
        terminal(P0\P1,Terminal), z(P1\P2,Z).
z(P0\P4,z(Abierto,Pd,Z,Z)) :-
        parentesis_abierto(P0\P1,Abierto),
        pd(P1\P2,Pd), z(P2\P3,Z), z(P3\P4,Z).
z(P0\P4,z(Abierto,T,Z,Z)) :-
        parentesis_abierto(P0\P1,Abierto),
        t(P1\P2,T), z(P2\P3,Z), z(P3\P4,Z).
z(P0\P3,z(Terminal,Z,Z)) :-
        terminal(P0\P1,Terminal), z(p1\P2,Z).

terminal([a|X]\X,a).
parentesis_abierto(['('|X]\X,'(').
parentesis_cerrado([')'|X]\X,')')
```

Capítulo 15

Análisis del Lenguaje Natural

El objetivo de este capítulo es el de abordar el análisis práctico de textos escritos en lenguaje natural, mediante la utilización de Prolog. Para ello aplicaremos el concepto de DCG, aunque este caso particular presente características propias que lo hacen especialmente interesante, lo que justifica un tratamiento aparte. Más concretamente, nuestro objetivo será el de mostrar la adecuación de Prolog para el tratamiento de ciertos problemas específicos:

- La ambigüedad, a todos los niveles: léxico, sintáctico y semántico.
- La existencia de dependencias contextuales, lo cual implica la introducción de restricciones semánticas a nivel de las reglas sintácticas.

Para ilustrar la discusión que sigue, construiremos una gramática que describa un pequeño subconjunto del español, así como el analizador sintáctico capaz de tratarlo. Nuestro propósito será ante todo didáctico, por lo que en ningún momento pretendemos presentar los resultados expuestos como un sistema completo en cuanto a su diseño. Por el contrario,

nuestro ánimo es el de proponer un conjunto de técnicas cuya extensión, en cuanto a su aplicación al problema concreto, se deja al lector.

15.1 Análisis sintáctico

Antes de introducirnos de lleno en el mundo de las DCG's, examinaremos una pequeña gramática de contexto libre que nos permitirá analizar frases sencillas tales como

"un hombre ama a una mujer"

En primer lugar, debemos escribir las reglas de la gramática que genere el lenguaje deseado y que son las que se muestran a continuación:

(1)	*Frase*	→	*Frase_nominal Frase_verbal*
(2)	*Frase_nominal*	→	*Determinante Nombre Clausula_relativa*
(3)		|	"a" *Determinante Nombre Clausula_relativa*
(4)		|	*Nombre*
(5)	*Frase_verbal*	→	*Verbo_transitivo Frase_nominal*
(6)		|	*Verbo_intransitivo*
(7)	*Clausula_relativa*	→	"que" *Frase_verbal*
(8)		|	*Frase_verbal*
(9)		|	ε
(10)	*Determinante*	→	"un"
		|	"una"
(11)	*Nombre*	→	"hombre"
		|	"mujer"
(12)	*Verbo_transitivo*	→	"ama"

Mediante esta gramática podemos generar sentencias que están formadas por una frase nominal más una frase verbal. Las frases nominales desempeñan las funciones de sujeto o de complementos del verbo. En su forma más sencilla están formadas únicamente por un sustantivo, o un determinante y un sustantivo, a los que puede preceder la preposición "*a*" cuando desempeñan la función de complemento del verbo. Adicionalmente, puede poseer una cláusula relativa, que generalmente se construye anteponiendo el relativo "*que*" a una frase verbal, aunque puede estar constituida tan solo por

esta última. Una frase verbal está formada por un verbo, que en el caso de tener carácter transitivo debe ir seguido por un complemento directo en la forma de una frase nominal. Con el objeto de simplificar la presentación de los conceptos fundamentales, no se contempla la existencia de complementos indirectos ni circunstanciales.

15.1.1 Una técnica simple

Siguiendo una técnica estándar de contrucción de DCG's, vamos a asociar a cada no terminal de la gramática un predicado binario con su mismo nombre, en el cual los argumentos representan las posiciones entre las que se sitúa la categoría sintáctica referida. Los argumentos de tales predicados representan los puntos de inicio y final de la cadena de caracteres generada por el no terminal correspondiente. La técnica aplicada en este caso es exactamente la misma considerada para el problema del palíndromo en la subsección 14.1.1. El resultado obtenido es el programa de la página 226.

La semántica declarativa de su primera regla es

> *"Podemos reconocer una frase entre las posiciones* `S0` *y* `S`*, si podemos reconocer una frase nominal entre* `S0` *y* `S1`*, y una verbal entre* `S1` *y* `S`*. "*

La semántica de las demás cláusulas es fácilmente deducible por analogía. Para representar los símbolos terminales se ha utilizado un predicado ternario, denominado `conecta`. La utilización de `conecta(S0,T,S1)` significa

> *"El símbolo terminal* `T` *está localizado entre las posiciones* `S0` *y* `S1` *de la cadena de entrada."*

Cada frase que se desee analizar deberá ser transformada en su correspondiente conjunto de predicados `conecta`. Por ejemplo, la sentencia

> *"un hombre ama a una mujer"*

```
frase(S0,S) :- frase_nominal(S0,S1), frase_verbal(S1,S).

frase_nominal(S0,S) :- determinante(S0,S1),
                       nombre(S1,S2),
                       clausula_relativa(S2,S).
frase_nominal(S0,S) :- conecta(S0,'a',S1),
                       determinante(S1,S2),
                       nombre(S2,S3),
                       clausula_relativa(S3,S).
frase_nominal(S0,S) :- nombre(S0,S).

frase_verbal(S0,S) :- verbo_transitivo(S0,S1),
                      frase_nominal(S1,S).
frase_verbal(S0,S) :- verbo_intransitivo(S0,S).

clausula_relativa(S0,S) :- conecta(S0,que,S1),
                           frase_verbal(S1,S).
clausula_relativa(S0,S) :- frase_verbal(S0,S).
clausula_relativa(S,S).

determinante(S0,S) :- conecta(S),'un',S).
determinante(S0,S) :- conecta(S),'una',S).

nombre(S0,S) :- conecta(S0,'hombre',S).
nombre(S0,S) :- conecta(S0,'mujer',S).

verbo_transitivo(S0,S) :- conecta(S0,'ama',S).
```

se traduce en[1]

```
conecta(1,'un',2).
conecta(2,'hombre',3).
conecta(3,'ama',4).
conecta(4,'a',5).
conecta(5,'una',6).
conecta(6,'mujer',7).
```

Para probar ahora que esta frase es gramaticalmente correcta, deberá realizársele al intérprete Prolog la siguiente pregunta:

```
:- frase(1,7).
```

cuya semántica declarativa viene dada por:

> *" ¿ Es posible reconocer una frase entre las posiciones 1 y 7 del texto ? "*

Es importante resaltar que la representación de una gramática de contexto libre es independiente de los datos, con este tipo de técnica. En efecto, la representación real de la cadena que va a ser analizada no es conocida, sino que se tiene en cuenta el objetivo establecido por la pregunta y el predicado `conecta`.

15.1.2 Una técnica flexible

Si en vez de utilizar un entero para etiquetar un determinado punto en una cadena, utilizásemos una lista con los símbolos ubicados a partir de ese punto en la cadena, no sería necesario emplear una cláusula `conecta` individual para cada símbolo de la cadena. En su lugar, se podría definir con una única cláusula el predicado `conecta`, en la forma:

```
conecta([P | S]\S,P).
```

que denotacionalmente significa

[1] debemos hacer notar que no se aplica factorización alguna en el análisis léxico para reducir el tamaño del diccionario, sino que éste contiene todas las formas derivadas de cada palabra. Esta última funcionalidad es perfectamente implementable en la práctica, pero por cuestiones exclusivamente didácticas hemos decidido limitarnos al tratamiento de la componente sintáctica del lenguaje.

> *"La posición de la cadena etiquetada por la lista con cabeza P y cola S está conectada mediante el símbolo P a la posición de la cadena etiquetada como S."*

En consecuencia, podríamos reescribir la pregunta formulada anteriormente de la siguiente manera:

```
:-frase(['un','hombre','ama','a','una','mujer']\[]).
```

Se trata en esencia de repetir el razonamiento que habíamos hecho para el caso del palíndromo en la subsección 14.1.2.

Un diccionario más compacto

Una de las primeras ventajas que se obtiene al utilizar DCG's es la posibilidad de definir el diccionario de una manera más compacta y eficiente. Para ello, en vez de utilizar cláusulas del tipo

categoría :- palabra.

utilizaremos cláusulas de la forma:

categoría :- P, es_categoría(P, $S_1, \ldots, S_n$*).*

De este modo, la cláusula

```
nombre(S0\S) :- 'gato'.
```

se traduce[2] en

```
nombre(Palabra) :-  conecta(S0\S,Palabra),
                    es_nombre(Palabra).
```

con un conjunto adicional de predicados en los que se especifican los valores válidos para que una palabra sea considerada un nombre. Por ejemplo, mediante

[2] los argumentos $S_1, \ldots, S_n$ se considerarán opcionales, y se reservarán para la introducción de las restricciones semánticas derivadas de la consideración de diversos accidentes gramaticales, tales como las concordancias de género y número.

```
es_nombre('gato').
es_nombre('perro').
es_nombre('hombre').
es_nombre('mujer').
```

establecemos como hechos que *gato*, *perro*, *hombre* y *mujer* son nombres.

La construcción de estructuras sintácticas

La utilización de argumentos adicionales en los no terminales permite construir estructuras que representen el avance del proceso de análisis sintáctico de las oraciones. La estructura generada por un no terminal se construye cuando se expande dicho no terminal como consecuencia del proceso de resolución. En la página 230 se muestra una versión modificada, para tal fin, de la gramática presentada anteriormente.

La semántica declarativa de la primera cláusula es

> *"Una frase con estructura* `f(FN,FV)` *está formada por una frase nominal que da lugar a la estructura* `FN` *y una frase verbal que da lugar a* `FV`*."*

Como se puede observar, la estructura construida a partir de las cláusulas se corresponde con una representación del árbol de análisis sintáctico. En nuestro caso, la raíz de dicho árbol está etiquetada con el funtor `f`.

15.2 Análisis semántico

En los lenguajes de comunicación humana existen multitud de aspectos contextuales que no pueden ser representados adecuadamente mediante la utilización de gramáticas de contexto libre. Tal es el caso de las concordancias de género y número. La utilización de un formalismo DCG nos permite introducir restricciones semánticas en las cláusulas mediante una adecuada utilización de las llamadas a procedimientos que explota de forma gratuita el concepto de unificación.

```
frase(f(FN,FV),S0\S) :-
      frase_nominal(FN,S0\S1), frase_verbal(FV,S1\S).

frase_nominal(fn(Det,Nom,Rel),S0\S) :-
      determinante(Det,S0\S1), nombre(Nom,S1\S2),
      clausula_relativa(Rel,S2\S).
frase_nominal(fn('a',Det,Nom,Rel),S0\S) :-
      conecta(S0\S1,'a'), determinante(Det,S1\S2),
      nombre(Nom,S2\S3), clausula_relativa(Rel,S3\S).
frase_nominal(fn(Nom),S0\S) :- nombre(Nom,S0\S).

frase_verbal(fv(VT,FN),S0\S) :-
      verbo_transitivo(VT,S0\S1), frase_nominal(FN,S1\S).
frase_verbal(fv(VI),S0\S) :- verbo_intransitivo(VI,S0\S).

clausula_relativa(rel('que',FV),S0\S) :-
      conecta(S0\S1,'que'), frase_verbal(FV,S1\S).
clausula_relativa(rel(FV),S0\S) :- frase_verbal(FV,S0\S).
clausula_relativa(rel(nil),S\S).

determinante(det(Palabra),S0\S) :-
      conecta(S0\S,Palabra), es_determinante(Palabra).

nombre(nom(Palabra),S0\S) :-
      conecta(S0\S,Palabra), es_nombre(Palabra).

verbo_transitivo(vt(Palabra),S0\S) :-
      conecta(S0\S,Palabra), es_verbo_trans(Palabra).

es_determinante('mi').
es_determinante('un').
es_determinante('una').

es_nombre('gato').
es_nombre('hombre').
es_nombre('mujer').

es_verbo_trans('ama').
```

15.2.1 Concordancias de género y número

Mediante la utilización de las llamadas a procedimientos incluidas en el formalismo de las DCG's es posible representar tales concordancias al nivel de análisis sintáctico, incluyendo un conjunto de restricciones semánticas.

Incorporación del número

Para establecer las concordancias de número, algunos de los no terminales deberán poseer argumentos adicionales cuyo rango de valores posibles viene representado por el conjunto

$$\{singular, plural, neutro\}$$

Así mismo, los predicados mediante los cuales se representa el diccionario deberán ser modificados, ya que es preciso incluir un nuevo argumento para indicar el número de la palabra correspondiente y otro más que proporcione el lema de la palabra[3]. En el entorno que estamos considerando, no es conveniente utilizar el lexema en lugar del lema debido al gran número de irregularidades existentes en el español.

Tras aplicar los cambios mencionados a las cláusulas involucradas, obtenemos el conjunto de cláusulas de la página 232.

Ahora ya podemos analizar frases del tipo *"un gato come ratones"* o *"unos gatos comen un ratón"*. Resulta interesante estudiar la semántica declarativa de algunas de las cláusulas. Por ejemplo, en la primera tenemos que:

> *"Una frase es correcta si está formada por una frase nominal y una frase verbal que concuerdan en número."*

[3]el lema es la forma que encabeza el grupo de palabras derivadas de una dada. En el caso de sustantivos, adjetivos y artículos, el lema lo constituye la forma masculina y singular; en el caso de los verbos, el infinitivo.

```
frase(f(FN,FV),S0\S) :-
  frase_nominal(Numero,FN,S0\S1),frase_verbal(Numero,FV,S1\S).

frase_nominal(Numero,fn(Det,Nom,Rel),S0\S) :-
  determinante(Numero,Det,S0\S1), nombre(Numero,Nom,S1\S2),
  clausula_relativa(Numero,Rel,S2\S).
frase_nominal(Numero,fn('a',Det,Nom,Rel),S0\S) :-
  conecta(S0\S1,'a'), determinante(Numero,Det,S1\S2),
  nombre(Numero,Nom,S2\S3),clausula_relativa(Numero,Rel,S3\S).
frase_nominal(Numero,fn(Nom),S0\S) :- nombre(Numero,Nom,S0\S).

frase_verbal(Numero_verbo,fv(VT,FN),S0\S) :-
  verbo_transitivo(Numero_verbo,VT,S0\S1),
  frase_nominal(Numero_nombre,FN,S1\S).
frase_verbal(Numero,fv(VI),S0\S) :-
  verbo_intransitivo(Numero,VI,S0\S).

clausula_relativa(Numero,rel('que',FV),S0\S) :-
  conecta(S0\S1,'que'), frase_verbal(Numero,FV,S1\S).
clausula_relativa(Numero,rel(FV),S0\S) :-
  frase_verbal(Numero,FV,S0\S).
clausula_relativa(Numero,rel(nil),S\S).

determinante(Numero,det(Palabra),S0\S) :-
  conecta(S0\S,Palabra), es_determinante(Numero,Palabra).
nombre(Numero,nom(Palabra),S0\S) :-
  conecta(S0\S,Palabra), es_nombre(Numero,Palabra,Lema).
verbo_transitivo(Numero,vt(Palabra),S0\S) :-
  conecta(S0\S,Palabra), es_verbo_trans(Numero,Palabra,Lema).

es_determinante(singular,'un','un').
es_determinante(singular,'una','una').
es_determinante(plural,'unos','un').
es_determinante(plural,'unas','un').
es_nombre(singular,'gato','gato').
es_nombre(singular,'gata','gato').
es_nombre(plural,'gatos','gato').
es_nombre(plural,'gatas','gato').
es_verbo_trans(singular,'come','comer').
es_verbo_trans(plural,'comen','comer').
```

Un caso distinto aparece en la tercera cláusula relativa a las frases verbales, ya que expresa

> *"Una frase verbal es correcta si está constituida por un verbo transitivo y una frase nominal."*

La diferencia radica en que ya no se fuerza la concordancia, puesto que es lícito construir frases como *"un gato come ratones"* , en la que el verbo *"come"* está en singular mientras que la frase nominal *"ratones"* está en plural. Es por ello que resulta preciso distinguir entre `Numero_verbo` y `Numero_nombre`.

Incorporación del género

Con el último conjunto de cláusulas obtenido, hemos creado una gramática que permite construir frases del tipo *"una gato come ratones"* gracias a la carencia total de restricciones en cuanto al género. El tratamiento de las concordancias de género se realiza de manera análoga al comentado para las concordancias de número. Por consiguiente, se introduce un argumento adicional en los no terminales que contendrá un valor del conjunto

$$\{masculino, femenino, neutro\}$$

Así mismo, los predicados mediante los cuales se construye el diccionario verán aumentada su aridad en una unidad para incorporar un argumento en el cual se almacene el género de la palabra. El lema se sigue almacenando en el argumento creado para establecer las concordancias de número. Una vez realizados estos cambios, obtenemos el conjunto de cláusulas de las páginas 234 y 235.

En este caso ha sido preciso modificar menos cláusulas que cuando habíamos introducido el número, puesto que existen construcciones para las cuales la separación en géneros no es aplicable. Concretamente, las frases verbales no tienen dependencias de género, puesto que su núcleo, el verbo[4], no

[4]que determina el número de la frase verbal.

```
frase(f(FN,FV),S0\S) :- frase_nominal(Numero,Genero,FN,S0\S1),
                        frase_verbal(Numero,Genero,FV,S1\S).

frase_nominal(Numero,Genero,fn(Det,Nom,Rel),S0\S) :-
                   determinante(Numero,Genero,Det,S0\S1),
                   nombre(Numero,Genero,Nom,S1\S2),
                   clausula_relativa(Numero,Genero,Rel,S2\S).
frase_nominal(Numero,Genero,fn('a',Det,Nom,Rel),S0\S) :-
                   conecta(S0\S1,'a'),
                   determinante(Numero,Genero,Det,S1\S2),
                   nombre(Numero,Genero,Nom,S2\S3),
                   clausula_relativa(Numero,Rel,S3\S).
frase_nominal(Numero,Genero,fn(Nom),S0\S) :-
                   nombre(Numero,Genero,Nom,S0\S).

frase_verbal(Numero_verbo,fv(VT,FN),S0\S) :-
                   verbo_transitivo(Numero_verbo,VT,S0\S1),
                   frase_nominal(Numero_nombre,Genero,FN,S1\S).
frase_verbal(Numero,fv(VI),S0\S) :-
                   verbo_intransitivo(Numero,VI,S0\S).

clausula_relativa(Numero,rel('que',FV),S0\S) :-
                   conecta(S0\S1,'que'),
                   frase_verbal(Numero,FV,S1\S).
clausula_relativa(Numero,rel(FV),S0\S) :-
                   frase_verbal(Numero,FV,S0\S).
clausula_relativa(Numero,rel(nil),S\S).
```

```
determinante(Numero,Genero,det(Palabra),S0\S) :-
                  conecta(S0\S,Palabra),
                  es_determinante(Numero,Genero,Palabra,Lema).

nombre(Numero,Genero,nom(Palabra),S0\S) :-
                  conecta(S0\S,Palabra),
                  es_nombre(Numero,Genero,Palabra,Lema).

verbo_transitivo(Numero,vt(Palabra),S0\S) :-
                  conecta(S0\S,Palabra),
                  es_verbo_trans(Numero,Palabra,Lema).

es_determinante(singular,masculino,'un','un').
es_determinante(singular,femenino,'una','una').
es_determinante(plural,masculino,'unos','un').
es_determinante(plural,femenino,'unas','un').

es_nombre(singular,masculino,'gato','gato').
es_nombre(singular,femenino,'gata','gato').
es_nombre(plural,masculino,'gatos','gato').
es_nombre(plural,femenino,'gatas','gato').
es_nombre(singular,'raton','raton')
es_nombre(singular,'ratones','raton').

es_verbo_trans(singular,'come','comer').
es_verbo_trans(plural,'comen','comer').
```

las posee a este nivel. Consecuentemente, al no ser aplicable el género a las frases verbales, tampoco lo es a las frases relativas.

15.2.2 El significado de las frases

El problema que abordamos ahora es, sin duda, el más complejo en lo que al tratamiento de los lenguajes naturales se refiere: la representación de su semántica. Se trata, sin embargo, de una cuestión inevitable en problemas tan cercanos como la traducción automática entre lenguas naturales o la generación de interfaces en lengua natural. La solución que a continuación esbozamos es, pese a sus innegables limitaciones, una técnica de aplicación común entre la comunidad especializada.

En esencia la idea es simple y atractiva, se trata de encontrar una representación formal de los conceptos usados en los lenguajes de comunicación humana. Una vez establecido el objetivo, la experiencia nos permite asegurar que:

- La utilización de lenguajes de programación lógica parece adaptarse adecuadamente a la implementación práctica de analizadores de lenguajes naturales.
- La base que sirve de fundamento a los lenguajes lógicos, el cálculo de predicados, presenta todas las garantías de formalidad.
- Los conceptos expresados en el cálculo de predicados constituyen un lenguaje universal, y por tanto aplicable a cualquier tipo de noción, independientemente de cuál sea la forma externa de expresar dicha noción.

En estas condiciones, la idea de considerar el cálculo de predicados como punto de partida para la representación de la semántica de los lenguajes de comunicación humana parece reunir buenas espectativas de éxito. Para ilustrar mejor este proceso volveremos a nuestra pequeña gramática del español y trataremos de construir un traductor que convierta frases en español como

"todo hombre que vive ama a una mujer"

en sus equivalentes en lógica de predicados, en este caso:

$$(\forall\ X : (hombre(X)\ \&\ vive(X)\ \Rightarrow\ \exists\ Y\ \&\ ama(X,Y)))$$

donde ":","&" y "⇒" son operadores binarios infijos que deben ser definidos como tales en Prolog. Para ello utilizaremos las siguientes declaraciones:

```
:- op(900, yfx, [=>]).
:- op(800, yfx, [&]).
:- op(300, yfx, [:]).
```

en las que se establece que los tres son operadores binarios asociativos por la izquierda y con proridades decrecientes. Mostramos el programa en la página 238. Para facilitar la comprensión de los conceptos que se tratan de inculcar aquí, hemos decidido no tener en cuenta ciertos aspectos considerados anteriormente, como son las concordancias de género y número y la construcción del árbol sintáctico, puesto que su presencia aumenta el número de argumentos en los predicados y no son indispensables en este ejemplo.

Cada no terminal posee uno o más argumentos, con el objeto de proporcionar la interpretación de la frase correspondiente. Para facilitar la compresión, las variables `X` e `Y` se refieren precisamente a las variables que posteriormente serán acotadas por el ∀ y el ∃, respectivamente. Las variables cuyo nombre comienza por `P` representan predicados parciales cuya composición dará lugar al predicado final contenido en la variable que se llama simplemente `P`.

Para mostrar cómo se construye la interpretación de una frase, comenzaremos por las partes más simples. Por ejemplo, el verbo *"ama"* tiene por interpretación `amar(X,Y)`, la cual depende de `X` e `Y`, cuyos valores dependen a su vez de las restantes partes de la frase. Un caso un poco más complejo tiene lugar con la palabra *"todo"*, que tiene la interpretación

```
forall(X) : (P1 => P2)
```

```
conecta([P|S]\S,P).

frase(P, S0\S) :- frase_nominal(X, P2, P, S0\S1),
                  frase_verbal(X, P2, S1\S).

frase_nominal(X, P2, P2, S0\S) :- nombre_propio(X,S0\S).
frase_nominal(X, P2, P, S0\S) :-
                      determinante(X, P1, P2, P, S0\S1),
                      nombre(X, P3, S1\S2),
                      clausula_relativa(X, P3, P1, S2\S).

clausula_relativa(X, P3, P3 & P4, S0\S)  :-
                      conecta(S0\S1, 'que'),
                      frase_verbal(X, P4,S1\S).
clausula_relativa(_,P3,P3,S0\S0).

frase_verbal(X,P2,S0\S) :- verbo_intransitivo(X, P2, S0\S).
frase_verbal(X,P2,S0\S) :- verbo_transitivo(X, Y, P5, S0\S1),
                           prep_a(S1\S2),
                           frase_nominal(Y, P5, P2, S2\S).

prep_a(S0\S) :- conecta(S0\S, 'a').
prep_a(S\S).

determinante(X,P1, P2, forall(X) : (P1 => P2), S0\S) :-
                                      conecta(S0\S, 'todo').
determinante(X, P1, P2, exists(X) : (P1 & P2), S0\S) :-
                                      conecta(S0\S, 'una').

nombre(X, hombre(X), S0\S) :- conecta(S0\S, 'hombre').
nombre(X, mujer(X), S0\S) :- conecta(S0\S, 'mujer').

nombre_propio('Miguel',S0\S) :-  conecta(S0\S, 'Miguel').
nombre_propio('Adriana',S0\S) :-  conecta(S0\S, 'Adriana').

verbo_transitivo(X,Y,amar(X,Y),S0\S) :- conecta(S0\S, 'ama').
verbo_intransitivo(X,vive(X),S0\S) :- conecta(S0\S, 'vive').
```

en el contexto de las propiedades `P1` y `P2` de un individuo `X`, en donde la propiedad `P1` se corresponde con la frase nominal que contiene la palabra *"todo"* y la propiedad `P2` vendrá del resto de la frase.

Como ejemplo ilustrativo, analizamos la frase

"todo hombre vive"

mediante la pregunta:

```
:- frase(P,[todo,hombre,vive]\[]).
```

obteniendo como respuesta

```
P ← forall(_A):(hombre(_A)=>vive(_A))
```

Podemos observar cómo la presencia del determinante *"todo"* produce la interpretación

```
forall(_A) : (P1 => P2),
```

donde `_A` representa el valor sin instanciar de la variable `X`, universalmente cuantificada por `forall`. A continuación, el sustantivo *"hombre"* provocará la identificación de `P1` con `hombre(_A)`, mientras que el verbo *"vive"* provocará a su vez la identificación de `P2` con `vive(_A)`.

El orden en el que se realiza el análisis sintáctico es tal que algunas partes de la traducción de la frase no están disponibles cuando esa traducción se construye. Es el caso de nuestro ejemplo, en el cual la determinación de la estructura

```
forall(X) : (P1 => P2)
```

es establecida nada más leer la primera palabra, momento en el cual los valores para `P1` y `P2` son desconocidos. En general, en nuestro programa la estructura de la interpretación `P` de una frase es producida por el sujeto, pero depende a su vez de la interpretación, todavía no conocida, de la frase verbal que completa la sentencia a analizar. Tales elementos desconocidos se convierten en variables que se instancian posteriormente, como es el caso de `P1` y `P2` en este ejemplo.

Para finalizar, consideremos un ejemplo algo más complejo, planteándole al programa el análisis de la frase

"todo hombre que vive ama a una mujer"

La pregunta correspondiente es:

```
:- frase(P,[todo,hombre,que,vive,ama,a,una,mujer]\[]).
```

cuya respuesta se corresponde con lo que se pretendía obtener:

```
P ← forall(_B):(hombre(_B) & vive(_B)
          => exists(_A):(mujer(_A) & amar(_B,_A)))
```

Mediante este sencillo programa no sólo podemos interpretar una cierta clase de frases, sino que podemos aprovechar la potencia de la programación lógica para generar aquellas frases que son válidas de acuerdo con el programa, así como sus interpretaciones. Esto podemos conseguirlo realizando la pregunta

```
:- frase(P,Q).
```

cuya primera respuesta es la siguiente:

```
{ P ← vive('Miguel')
{ Q ← ['Miguel',vive|_A]\_A
```

El programa nos dice que *"Miguel vive"*, menos mal. Veamos la siguiente respuesta:

```
{ P ← amar('Miguel','Miguel')
{ Q ← ['Miguel',ama,a,'Miguel'|_A]\_A
```

Parece que este chico es un poco narcisista. Sigamos:

```
{ P ← amar('Miguel','Adriana')
{ Q ← ['Miguel',ama,a,'Adriana'|_A]\_A
```

Parece que el programa, además de inteligente, también es un poco cotilla. Continuemos:

```
{ P ← forall(_A):(hombre(_A) & vive(_A) => amar('Miguel',_A))
{ Q ← ['Miguel',ama,a,todo,hombre,que,vive|_B]\_B
```

El tal Miguel es realmente un chico encantador. Prosigamos:

```
{ P ← forall(_A):(hombre(_A) & amar(_A,'Miguel')
                        => amar('Miguel',_A))
  Q ← ['Miguel',ama,a,todo,hombre,que,ama,a,'Miguel'|_B]\_B
```

Amigo de sus amigos, como debe ser. Veamos que más nos dice el programa:

```
{ P ← forall(_A):(hombre(_A) & amar(_A,'Adriana')
                        => amar('Miguel',_A))
  Q ← ['Miguel',ama,a,todo,hombre,que,ama,a,'Adriana'|_B]\_B
```

Una de dos, o Miguel no es celoso, o el programa ha cometido un pequeño fallo esta vez. Dejemos este asunto aparte y pasemos a la siguiente respuesta:

```
{ P ← forall(_A):(hombre(_A) & forall(_B):
  (hombre(_B) & vive(_B) => amar(_A,_B)) => amar('Miguel',_A))
  Q ← ['Miguel',ama,a,todo,hombre,que,ama,a,todo,hombre,que,
                        vive|_C]\_C
```

Entre pacifistas, todos hermanos. Este pequeño conjunto de respuestas puede servirnos para ver la potencia del lenguaje. Evidentemente, el tipo de frase que puede ser analizada y generada es reducido, pero podemos ampliar el dominio de aplicación del programa mediante la simple adición de nuevas cláusulas que sigan la misma estructura. Si además añadimos otros aspectos semánticos, como por ejemplo las concordancias vistas anteriormente, podremos mejorar notablemente la exactitud del análisis y de la interpretación de las frases.

Referencias

[1] **Aho, A. V. and Ullman, J. D.**
The Theory of Parsing, Translation and Compiling.
1973. Prentice-Hall, Englewood Cliffs, vol. 1,2, New Jersey, U.S.A.

[2] **Bancilchon, F.; Maier, D.; Sagiv, Y.; and Ullman, J. D.**
Magic Sets: Algorithms and Examples.
1985. MCC Technical Report DB-168-85.

[3] **Beeri, C.; and Ramakristan, R.**
On the Power of Magic.
1987. Proc. of the 6^{th} ACM SIGACT-SIGMOD-SIGART Syposium on Principles of Database Systems (pp. 269-283), San Diego, California, USA.

[4] **Bird, R. S.**
Tabulation Techniques for Recursive Programs.
1980. Computing Survey, vol. 12, $n^o 4$ (pp.403-417).

[5] **Colmerauer, A.**
Metamorphosis Grammars.
1978. Natural Language Communication with Computers, L. Bolc ed., Springer LNCS 63.

[6] **Dietrich, S. W.**
Extension Tables: Memo Relations in Logic Programming.
1987. Proc. of the 4^{th} Symposium on Logic Programming (pp. 264-272), San Francisco, California, USA.

[7] **Giannesini, F. and Cohen, J.**
Parser Generation and Grammar Manipulations Using Prolog's Infinite Trees.
1984. Journal of Logic Programming vol.1, n^o3 (pp. 253-265).

[8] **Hill, R.**
LUSH-Resolution and its Completeness
1974. DCL Memo 78, Departament of Artificial Intelligence, University of Edimburgh.

[9] **Jacobs, I.**
The Centaur 1.2 Manual.
1992. INRIA Sophia-Antipolis, France.

[10] **Kay, M.**
Unification in Grammar.
1984. Proc. of the First International Workshop on Natural Language Understanding and Logic Programming, Rennes, France (pp. 233-240).

[11] **Kim, W.; Reiner, D. S. and Batory, D. S.**
Query Processing in Database Systems
1985. Springer-Verlag, New York, U.S.A.

[12] **Kowalski, R. A.**
Logic for Problem Solving.
1980. North-Holland Publishing Company, New York, U.S.A.

[13] **Kowalski, R. A. and Van Emden, M. H.**
The Semantics of Predicate Logic as a Programming Language.
1976. Journal of the ACM, vol. 23, n^o4 (pp. 133-742).

[14] **Lang, B.**
Datalog Automata.
1988. Proc. of the 3^{rd} International Conference on Data and Knowledge Bases, C. Beeri, J. W. Schmidt, U. Dayal (eds), Morgan Kaufmann Pub., Jerusalem, Israel (pp. 389-404).

[15] **Lang, B.**
Complete Evaluation of Horn Clauses, an Automata Theoretic Approach.
1988. Research Report n^o913, INRIA Rocquencourt, France.

[16] **Lang, B.**
Towards a Uniform Formal Framework for Parsing.
1991. Current Issues in Parsing Technology, M. Tomita ed., Kluwer Academic Publishers (pp. 153-171).

[17] **Lloyd, J. W.**
Foundations of Logic Programming.
1987. Springer-Verlag, 2^{nd} edition.

[18] **Maier, D. and Warren, D. S.**
Computing with Logic
1988. The Benjamin/Cummings Publishing Company Inc., Menlo Park, California, U.S.A.

[19] **Naish, L.**
MU-Prolog 3.2. Reference Manual
1990. Technical Report 85/11. Dept. of Computer Sciences, University of Melbourne, Parkville, Victoria 3052, Australia.

[20] **Pereira, F. C. N. and Warren, D. H. D.**
Definite Clause Grammars for Language Analysis. A Survey of the Formalism and a Comparison with Augmented Transition Networks.
1980. Artificial Intelligence, vol.13 (pp. 231-278).

[21] **Pereira, F. C. N. and Warren, D. H. D.**
Parsing as Deduction.
1983. Proc. of the 21^{st} Annual Metting of the Association for Computational Linguistics, Cambridge, Massachusetts, U.S.A (pp. 137-144).

[22] **Robinson, J. A.**
A Machine-Oriented Logic Based on the Resolution Principle
1965. Journal of the ACM, vol. 12.

[23] **Tamaki, H. and Sato, T.**
OLD Resolution with Tabulation.
1986. Proc. of the 3^{rd} International Conference on Logic Programming, London, United Kingdom, Springer LNCS 225 (pp. 84-98).

[24] **Van Emdem, M. H., and Lloyd, J. W.**
A Logical Reconstruction of Prolog II
1990. The Journal of Logic Programming, vol. 1, n^o2.

[25] **Vieille, L.**
Database-Complete Proof Procedures Based on SLD Resolution.
1987. Proc. of the 4^{th} International Conference on Logic Programming, Melbourne, Australy.

[26] **Villemonte de la Clergerie, E.**
Automates à Piles et Programmation Dynamique.
1993. Doctoral Thesis, University of Paris VII, France.

[27] **Vilares Ferro, M.**
Efficient Incremental Parsing for Context-Free Languages.
1992. Doctoral thesis, University of Nice, France.

Índice