Lenguajes de Programación

José de Jesús Lavalle Martínez

Otoño de 2008

Índice general

Capítulo 1

Descripción de lenguajes de programación

En este capítulo se estudia como se conforman los paradigmas de programación y se establecen los conceptos y principios que los caracterizan.

Es conveniente aclarar que dichos conceptos pueden tomar otros nombres en diferentes lenguajes de programación, de cualquier manera un buen libro que enseñe alguno de estos lenguajes normalmente hace las anotaciones pertinentes. Se prefirió no adoptar la terminología de algún lenguaje en particular para mantener cierta generalidad.

1.1. Antecedentes

Aquí no se analizará la historía de los lenguajes de programación, sino más bien los principios de lenguajes de programación (*PLP*) que han sido más relevantes y que propiamente conllevan a aquellos principios que definen a los paradigmas de programación imperativa, dado que los siguientes dos capítulos tratarán con sendos paradigmas de programación declarativa.

Un PLP es la respuesta a una necesidad práctica que acerca a escribir programas de manera más parecida a como mentalmente se concibe la solución de un problema y por lo tanto cada principio marca un cambio cualitativo en la práctica de la programación.

Un conjunto específico de PLP conforman un paradigma de programación (*PP*), finalmente un PP es un enfoque a como se deben realizar cálculos con una computadora.

Como es bien sabido para que una computadora realice un cálculo se le deben dar una serie de instrucciones, por si fuera poco sólo entiende 0's y 1's. Así la primera necesidad práctica fue evitar escribir código en 0's y 1's, por lo que el primer Principio fue el llamado *Principio de Etiquetamiento*, que significa poder asignarle una etiqueta o identificador tanto al código en binario de una instrucción como a su(s) operando(s). Creándose implícitamente los primeros

Lenguajes de Programación, es decir, los Lenguajes Ensambladores, ahora llamados de Bajo Nivel. Para poder hacer realidad este principio se tuvo que crear un nuevo tipo de programas de computadora, es decir, los programas traductores.

Note que el trabajo que tenian que hacer los primeros traductores era pasar de código escrito con etiquetas al respectivo conjunto de 0's y 1's que iba a entender un hardware en particular, además en aquellos tiempos las computadoras no eran tan accesibles como ahora, por lo que el código que se escribía en un Centro de Cálculo difícilmente serviría en otro Centro de Cálculo ya que de seguro tenian Hardwares distintos.

Ante dicha necesidad surge el *Principio de portabilidad*, o lo que es lo mismo, poder ejecutar el código de algún programa en cualquier Centro de Cálculo, sin tener que hacer modificaciones o ajustes; de esta manera surgen los Lenguajes de Alto Nivel[1].

A la par de este desarrollo evoluciona también la electrónica y se empiezan a reducir los precios de los equipos gracias a la producción en serie del hardware. Lo que permite que se extienda la práctica de la programación, con lo que ahora es deseable no tener que escribir código que ya ha sido producido por otra persona[2], surge así el *Principio de Reusabilidad*.

Pero sucedió que no era tan fácil simplemente tomar código ya escrito y componer un nuevo programa, con frecuencia el programa compuesto no funcionaba, lo que es peor al tratar de adecuarlo resultaba que con mayor frecuencia el código era **ilegible e inentendible**. Es así como investigadores de la talla de Dijkstra, Hoare y Wirth se dan a la tarea de analizar por qué, algo aparentemente tan natural, era tan difícil.

Los resultados de sus investigaciones arrojaron que la responsable de tremendo fracaso era la instrucción `GOTO`, más o menos afirman lo siguiente:

> ... esta instrucción causa que no nos disciplinemos a la hora de escribir un programa, ya que somos muy dados a que ante un problema nos sentemos a escribir código inmediatamente; como no hemos meditado suficientemente una estrategia para la solución, muy probablemente nuestro primer intento fallará y de lo primero que nos auxiliaremos es de la instrucción GOTO; mandaremos así el control del programa desde donde encontramos el error hasta algún lugar recondito 10 páginas despues; probaremos nuevamente el código, volverá a fallar, etc, etc, etc. Lo que es peor, cada uno de los ciclos anteriores irá introduciendo nuevos errores y seguramente ocultando aún más, los ya existentes ...

Sus conclusiones causaron revuelo en el medio computacional y partieron a los programadores en dos bandos, los defensores de la GOTO y sus detractores. Por la reacción tan fuerte que hubo a sus conclusiones sabían que el caso merecía

[1] Aqui existe una bifurcación considerable sobre los lenguajes que surgieron y los rumbos que tomaron, pero como se advirtió, no atenderemos aqui este aspecto.

[2] Obviamente código que calculaba lo que tenía que calcular.

tomar medidas extremas, es decir, definir lenguajes de programación que no tuvieran la instrucción GOTO y que las instrucciones de control de flujo tuvieran una sintaxis y semántica tal, que fuera evidente la *estructura* de un programa simplemente viendo su listado.

Para terminar de completar el poder de dichos lenguajes se debia permitir al usuario definir sus propios estructuras de datos, de tal forma que pudiera representar lo más fielmente posible lo que estaba modelando, para poder implementar a su vez los algoritmos más adecuados que iban a actuar sobre dichas estructuras; requerir esto es pedir que se cuente con el *Principio de Abstracción de Datos*. Un lenguaje que cuente con los principios analizados hasta aqui se dice que pertenece al *Paradigma Estructurado*.

Gracias a estas herramientas se empezó a demandar la automatización de tareas cada vez más cotidianas; así que la creación de programas empezó a ser un negocio jugoso, pero nuevamente surgieron problemas, de los cuales se distinguen tres:

1. El tamaño del código es muy grande, por lo que se contrata a mucha gente para terminar rápido o se tarda mucho tiempo en terminarlo.
2. Los programas en el 99 % de los casos no se entregaban a tiempo.
3. Ya entregados no se usaban debido a:
   a) No funcionaban.
   b) Funcionaban, pero no calculaban lo que debian calcular, es decir, no se había entendido para qué se quería.
   c) El problema que resolvía había evolucionado, por lo que el programa era ya obsoleto.

todo lo anterior constituye la llamada *Crisis del Software*, se desprende de las características anteriores que el dinero que se invierte en la creación de software va directamente al cajón del escritorio de algún directivo de la compañia que demandaba dicho software.

Prácticamente es en este momento de la historia donde toma auge la *Ingeniería de Software*, con el propósito de crear metodologías de construción de grandes sistemas, de aquí surge el concepto de *Ciclo de Vida del Software*, que plantea seguir los siguientes pasos para crear una solución computacional a un problema dado:

1. Entender el problema (dadas las especificaciones del usuario).
2. Analizar lo qué el sistema deba hacer.
3. Diseñar la mejor solución.
4. Implantación computacional del diseño (Codificación).
5. Instalación del sistema y pruebas del mismo.

si en algún momento se detectaba una inconsistencia se volvía al paso anterior, hasta que no se encontraba inconsistencia alguna.

Para realizar la parte de implementación, por el tamaño que tendría el sistema se repartía en grupos de programadores, a cada grupo se le daba una especificación de lo que su código debia calcular y un juego de pruebas de control. Como en el ciclo de vida, el código era liberado hasta que no se encontraban más errores; una vez que se liberaba el código de un grupo de programadores era sometido a un grupo especial de (por así decir) *Rompe Códigos* que como su nombre sugiere se encargaban de encontrar errores no percibidos por los propios programadores, si los rompe códigos lograban su objetivo se devolvía el código a sus desarrolladores y así sucesivamente, hasta que nadie encontraba más errores.

Una vez que cada subsistema aprovaba este control de calidad se pasaba a la fase de integración y contrario a lo que puede suponerse el resultado era que el sistema completo no funcionaba.

Entre otros, Parnas se plantea el reto de analizar por qué sucedia dicho fenómeno, los aspectos más comunes que encontró son:

1. Sintácticos:
   - *a*) Identificadores duplicados.
   - *b*) Incompatibilidad de tipos.
   - *c*) Incompatibilidad en el número de argumentos en un llamado a subprograma.
2. Lógicos:
   - *a*) Variables “globales” no iniciadas.
   - *b*) Variables “globales” modificadas accidentalmente,
   - *c*) Argumentos de subprogramas pasados por referencia creyendose que era por valor.
   - *d*) Modificación del código por otro(s) grupo(s) distinto(s) al que lo creó.

por lo que la metodología planteada no era suficiente para la construcción de software de gran tamaño, así que para evitar los problemas ya enumerados, surge la necesidad de robustecer la abstracción de datos, es decir, tener algún mecanismo en los lenguajes de programación que permita no sólo poder modelar la estructura de datos que se quiera y las operaciones sobre dicha estructura, sino que también permita *ocultar* a los usuarios de dichas estructuras tanto la implementación de ellas como el código que las manipula, así surge el *Principio de Ocultación*.

Nótese que el Principio de Ocultación encapsula datos y código en una unidad lógica a la que se le da el nombre de *Objeto*, como consecuencia de este principio sólo se conoce la interficie que un objeto tiene con el entorno, dicha interficie especifica el comportamiento que tiene un objeto con los objetos con los que coexiste, en términos computacionales, se conoce únicamente el nombre y los parámetros de los subprogramas que afectan o consultan el estado de un

objeto, pero no se conocen ni los nombres de las variables que representan el estado, ni el código con el que se llevan al cabo las operaciones, por tanto *no se puede acceder directamente al estado de un objeto*. Los lenguajes de programación que implanten de una u otra manera los principios analizados hasta aqui, se dice que se soportan el *Paradigma Basado en Objetos*. También debemos observar que por el análisis hecho por Parnas, los lenguajes en este Paradigma son estrictos en cuanto al manejo de tipos, es decir, los elementos involucrados en una expresión deben ser necesariamente del mismo tipo.

Con el Paradigma Basado en Objetos se pueden desarrollar grandes sistemas de cálculo, pero con frecuencia se daba el siguiente fenómeno; el objeto con el que se está modelando un problema se parece en mucho a algún otro objeto que ya está implementado, sólo varían detalles como:

- El estado del nuevo objeto es más complejo, por lo que se necesitan más variables para poder representarlo.
- Al tener un estado más complejo, debe exibir también un comportamiento más complejo.

por lo que sería deseable *Compartir el Comportamiento* de objetos ya existentes y sólo desarrollar las nuevas características que tiene un objeto nuevo. Estamos hablando así del *Principio de Compartición de Comportamiento*.

Es conveniente analizar con mayor detalle que significa que un objeto encapsule estado y código. Cuando estamos tratando de automatizar alguna tarea determinada, se deben distinguir los objetos alli involucrados, el siguiente paso es *abstraer* las características esenciales que tiene cada objeto para el problema en cuestión, es decir, si bien en un primer acercamiento sólo se pueden observar objetos concretos, se tiene que pensar en el conjunto de características que hacen a cada objeto concreto una instancia del objeto abstraido.

Por ejemplo, probablemente el primer automóvil del que tuvimos conciencia era uno muy particular, tangible, era rojo, pequeño, tenia un faro roto, los sillones estaban relucientes, olía al perfume de Papá, tenía cuatro llantas, cuatro puertas y siempre arrancaba, de toda está información para definir el concepto de automóvil seguro no nos sirve saber que olía al perfume de Papá o que tenia los sillones relucientes, pero si que tenía un color (no impota cual), ciertas dimensiones como alto, ancho, largo, un determinado número de puertas y otras características de las que no nos habiamos percatado como peso, tipo de motor, etc.

Una vez que se ha concebido al objeto abstracto se debe determinar el conjunto de cosas que se pueden hacer con él a lo que se le llama el comportamiento del objeto, note que la determinación del comportamiento también es independiente de un objeto concreto, es decir, al igual que en la determinación del estado, se debe de buscar aquello que todos los objetos con determinadas características pueden hacer sin importar alguno en particular, en el ejemplo del automóvil, aunque puedan existir autos que nunca han chocado, en principio existe la posibilidad de que eso suceda para cualquier instancia del objeto abstracto automóvil.

Así que en el POO estado y comportamiento son una unidad indivisible y el estado de un objeto sólo puede cambiar si está definido que tenga ese comportamiento, por lo que no podemos cambiar el estado si la propia definición del objeto no lo permite, volviendo al automóvil, implica que un auto sólo puede cambiar de color, si éste es pintado, chocado o ha incidido el sol mucho tiempo en él, *no pude cambiar su color espontáneamente*, cosa que accidentalmente si podriamos hacer sin el principio de ocultación.

Para que un objeto pueda manifestar un comportamiento se le debe "pedir" que así lo haga mediante un mensaje, por lo que al conjunto de mensajes que atiende un objeto se le conoce con el nombre de *Protocolo*.

Una de las formas de implementar este principio es mediante la herencia simple, a lo que aqui se le ha llamado objeto abstracto en el POO se le llama *clase* y a cada instancia particular de una clase se le llama objeto, así que en lo único que potencialmente cambia un objeto respecto de otros objetos de la misma clase es en los valores particulares que cada elemento de su estado tiene, en la práctica lo que sucede cuando un objeto es creado es que sólo se le aparta la memoria necesaria para que pueda almacenar su estado y obviamente alguna manera de acceder a los métodos que implementan su comportamiento.

En la herencia simple cada clase debe tener una y sólo una superclase, de la que heredará tanto su estado como su comportamiento, de esta manera podemos expresar el hecho de que una clase sea muy parecida a otra y sólo se diferencie por tener un estado y/o comportamiento más complejo. De esta manera todas las clases[3] se ven como una especialización de otra clase, por ejemplo, un automóvil deportivo es la especialización de un automóvil, éste a su vez es la especialización de un vehículo de motor, que a su vez es la especialización de un vehículo de transporte y así sucesivamnete.

En el ejemplo anterior es claro que la diferenciación de los objetos puede tener diversos niveles de aproximación, por decir, en cierto nivel un automóvil y una bicicleta pueden verse como instancias de la misma clase (la clase vehículo de transporte), pero en un nivel más fino un Ferrari y un Volks Wagen pueden no ser instancias de la misma clase, ya que el Ferrari pertenece a la clase automóvil deportivo y el Volks Wagen a la clase automóvil.

Como consecuencia de lo anterior se desprende que en el POO se pueden ir modelando problemas a diferentes niveles de complejidad, en dependencia del conocimiento que se tenga del fenómeno en estudio, en otras palabras esto quiere decir que, se pueden hacer aproximaciones sucesivas a una solución y entre más conocimiento tengamos sobre el fenómeno más iremos enriqueciendo nuestro sistema; donde lejos de presentar problemas el profundizar sobre el fenómeno, se enriquece la jerarquía de lo que se puede modelar.

Si bien como ya se mencionó una clase hereda el estado y el comportamiento de su superclase, también es posible que para la clase hija no sea adecuado exibir determinado comportamiento que ha heredado, por lo que se permite a la clase hija que *redefina* ese comportamiento acorde a sus necesidades, por ejemplo, un automóvil seguramente tiene una velocidad límite a la que puede avanzar, si de

[3]Excepto la más superior de la jerarquía.

esta clase derivamos la clase automóvil deportivo, es obvio que no debe de tener la misma velocidad límite que la clase automóvil, por lo que es deseable redefinir el método que permite hacerlo avanzar a la velocidad límite haciendo énfasis en que para los automóviles deportivos la velocidad límite es considerablemente mayor.

Como un objeto muestra comportamiento es frecuente que objetos de distintas clases tengan el mismo comportamiento, si bien para alcanzar dicho comportamiento el código para lograrlo es distinto, por ejemplo, diversas clases de objetos pueden recibir el mensaje suma, pero dependiendo del objeto que reciba el mensaje en tiempo de ejecución (*ligadura tardía*) se activará el método correspondiente a la clase del objeto que lo recibió. Con esto se dice que el mensaje es *polimorfo* y que los métodos que responden a él están *sobrecargados*.

Cada vez que se crea un nuevo objeto se le asigna un identificador llamado *identificador de objeto o ido*, se garantiza que ningún otro objeto ha tenido o tendrá el mismo ido, incluso si el objeto muere a ningún otro objeto se le asignará el mismo ido. Por otro lado las variables no contienen objetos sino que son apuntadores a ellos, así que un objeto muere cuando ninguna variable hace referencia a él, cabe mencionar que al usuario no le compete liberar los recursos que ocupaba un objeto que ha muerto, por lo que debe existir un recolector de basura que haga el trabajo, note que las variables pueden incluso ser parte del estado de otro objeto, por lo que es natural representar objetos complejos, es decir, objetos que están constituidos por otros objetos.

De lo anterior se desprenden dos cosas; primero, si cambia el estado de un objeto ese cambio automáticamente queda reflejado en los objetos que hacian referencia a él (*integridad referencial*), ya que como se dijo las variables no contienen objetos sino apuntadores a éstos; segundo, el concepto de igualdad tiene dos acepciones, a saber:

Identidad Si dos objetos tienen el mismo ido.

Similaridad Si a su vez son iguales los objetos que constituyen el estado de cada uno de los dos objetos que se están comparando.

Gracias a los conceptos anteriores también es muy fácil lograr que un objeto *mute* a algún objeto de otra clase, basta con que quien hacía referencia al objeto que va a mutar, haga referencia ahora al otro objeto y que quien hacía referencia al otro objeto ahora lo haga al que mutó. La característica anterior puede parecer extraña pero por la forma en que el POO concibe un cálculo es más necesaria de lo que pudiera parecer, por ejemplo, considere que se quiere modelar un arbol binario, la manera correcta de hacerlo en el POO es definiendo dos clases `árbolBinarioVacío` y `árbolBinarioNoVacío`, hijas de la clase abstracta `árbolBinario`, entonces cuando se crea un árbol que no tenga algo en absoluto, se deberá crear en realidad una instancia de `árbolBinarioVacío` y cuando se necesite insertar elmentos en él, dicho objeto deberá mutar para convertirse en una instancia de `árbolBinarioNoVacío`, conversamente cuando borremos todos los elementos de una instancia de `árbolBinarioNoVacío` ésta deberá mutarse a una instancia de `árbolBinarioVacío`

Finalmente un concepto muy útil dentro del POO es el de *clase abstracta*, la cual define todo su protocolo pero no lo implementa completamente, ya que dicha implementación dependerá de la representación del estado de sus subclases. Como ejemplo, considere todas las clases de objetos que cumplan una relación de orden, podemos definir la clase abstracta BienOrdenado y derivar de allí a los enteros, reales, caracteres ascii, cadenas, etc., y definir e implementar en BienOrdenado los métodos para los mensajes <=, >, >=, etc., pero sólo definir el mensaje < en BienOrdenado indicando que cada subclase lo implementará de la manera que mejor le convenga a sus intereses.

Capítulo 2

Programación lógica

Kowalski caracterizó el análisis de un algoritmo mediante la "ecuación":

$$Algoritmo = Lógica + Control$$

es decir, dado un algoritmo que resuelve un problema particular se deben tener dos componentes: el *componenete lógico* que especifica el conocimiento que se tiene para resolver el problema y el *componente de control* que determina la manera en que este conocimiento se usa para llegar a la solución del problema.

En un sistema de programación lógica un programador especifica el componente lógico de un algoritmo, llamado *programa lógico*, por medio de la lógica matemática. El componente de control está dado por alguna implementación del procedimiento de refutación. Es por esto que la programación lógica es un paradigma representativo de la *programación declarativa*, el programador no da ordenes para administrar los recursos de la computadora, sino que declara las relaciones existentes entre los componentes de la solución al problema.

2.1. Lógica de primer orden

Definición 1 El alfabeto de primer orden para la lógica de primer orden consiste de los siguientes símbolos:

1. Delimitador: , (coma)
2. Paréntesis: (,)
3. Conectivos primitivos: $\neg$, $\rightarrow$
4. Cuantificador universal: $\forall$
5. Variables individuales: $x, y, z, x_1, y_1, z_1, \cdots$
6. Constantes individuales: $a, b, c, a_1, b_1, c_1, \cdots$

7. Para cada número natural n, símbolos predicado de aridad n:
   $P^n, Q^n, R^n, P_1^n, Q_1^n, R_1^n, \cdots$

8. Para cada número natural n, símbolos función de aridad n:
   $f^n, g^n, h^n, f_1^n, g_1^n, h_1^n, \cdots$

Definición 2 Los términos son expresiones que inductivamente se definen mediante:

1. Una variable o constante individual es un término,

2. Si f es un símbolo función de aridad n y $t_1, \cdots, t_n$ son términos entonces $f(t_1, \cdots, t_n)$ es un término,

3. Una expresión es un término sólo si se puede demostrar que lo es mediante las condiciones anteriores.

Definición 3 Si P es un símbolo predicado de aridad n y $t_1, \cdots, t_n$ son términos entonces $P(t_1, \cdots, t_n)$ es una *fórmula atómica o átomo o literal positiva*. Una *literal negativa* es una fórmula de la forma $\neg A$ donde A es un átomo. Una *literal* es positiva o negativa.

Definición 4 Las fórmulas bien formadas (fbf) o simplemente fórmulas de la lógica de primer orden se definen de la siguiente manera:

1. Toda fórmula atómica es una fórmula,

2. Si F es una fórmula entonces $\neg F$ es una fórmula,

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

4. Si F y G son fórmulas entonces $F \to G$ es una fórmula,

5. Una expresión es una fórmula sólo si se puede generar usando alguna(s) de las cuatro condiciones anteriores.

Ejercicio 1 Compruebe que los conectivos lógicos $\vee$, $\wedge$ y $\leftrightarrow$ se pueden expresar en términos de los conectivos $\neg$ y $\to$ de la siguiente manera:

$$\begin{aligned} F \vee G &\equiv \neg F \to G, \\ F \wedge G &\equiv \neg(F \to \neg G), \\ F \leftrightarrow G &\equiv \neg((F \to G) \to \neg(G \to F)). \end{aligned}$$

Definición 5 Un *cuantificador existencial*, denotado $\exists$, se introduce y define de la siguiente manera: $\exists x(F)$ queda definido mediante $\neg(\forall x(\neg F))$.

Definición 6 En las fórmulas $\exists x(F)$ y $\forall y(G)$, se dice que F y G son el *ámbito* de los cuantificadores $\exists x$ y $\forall y$ respectivamente.

Definición 7 Dado un alfabeto de primer orden, el *lenguaje de primer orden* comprende al conjunto de todas las fórmulas construidas con los símbolos del alfabeto.

Definición 8 Se dice que una ocurrencia de una variable x en una fórmula F está *acotada* (o que x está *acotada* en F) si $\forall x$ ocurre en F o x cae en el ámbito de un cuantificador $\forall x$ en F. Si la ocurrencia de x en F no está acotada se dice que su ocurrencia está *libre* en F (o simplemente que x está *libre* en F). Una ocurrencia de una variable en una fórmula puede ser tanto libre como acotada. Tal caso se puede evitar renombrando simultaneamente las variables, en el cuantificador y sus ocurrencias acotadas asociadas, por nuevas variables.

Definición 9 Una fórmula sin variables libres se llama *fórmula cerrada* (o *sentencia* o *enunciado*). Si $x_1, \cdots, x_n$ son todas las variables libres de F entonces a la fórmula $\forall x_1 \cdots \forall x_n F$ se le llama la *cerradura* de F y se abrevia $\forall F$.

Ejemplo 1 Suponga que $F = \forall x(P(x, y))$ y que $G = \forall x(P(x, y) \rightarrow \forall y(Q(y)))$. La variable x está acotada en F y en G. La variable y está libre en F, en G su primera ocurrencia está libre, pero la segunda y tercera ocurrencias están acotadas.

El uso de la notación $F[x_1, \cdots, x_n]$ enfatiza que $x_1, \cdots, x_n$ son algunas de las variable libres de F. La substitución de los términos $t_1, \cdots, t_n$ por las ocurrencias libres de, respectivamente, $x_1, \cdots x_n$ se denota mediante $F[x_1/t_1, \cdots, x_n/t_n]$.

Definición 10 Se dice que un término t está *libre para una variable* x en una fórmula F si ninguna ocurrencia libre de x cae dentro del ámbito de cualquier cuantificador $\forall y$, donde y es una variable que ocurre en t.

Ejemplo 2 Considere a la fórmula $\forall x(P(x, y)) \rightarrow Q(z)$ y al término $f(a, x)$. El término está libre para z en la fórmula, pero no está libre para y en la misma fórmula.

Definición 11 Una interpretación $\mathcal{I}$ de un lenguaje de primer orden consiste de lo siguiente:

1. Un conjunto no vacío D,
2. Una asignación a cada símbolo predicado de aridad n a una relación de aridad n en D,
3. Una asignación a cada símbolo función de aridad n a una función con dominio D^n y codominio D,
4. Una asignación a cada constante individual a un elemento fijo de D.

En una interpretación a los conectivos lógicos y a los cuantificadores se les da su significado usual y se considera que las variables tomarán valores en D. Si e es una expresión cerrada (símbolo función o predicado) entonces $\mathcal{I}(e)$ denota a la asignación corrspondiente dada por $\mathcal{I}$. En especial si t es un término cerrado de la forma $f(t_1, \cdots, t_n)$ entonces la asignación correspondiente dada por $\mathcal{I}$ es $\mathcal{I}(f)(\mathcal{I}(t_1), \cdots, \mathcal{I}(t_n))$.

Definición 12 Suponga que $\mathcal{I}$ es una interpretación con dominio D y que $t \in D$. Entonces $\mathcal{I}_{(x_i/t)}$ es una interpretación la cual es exactamente la misma que $\mathcal{I}$ excepto que la i-ésima variable x_i siempre toma el valor t en vez de que varíe sobre el dominio entero.

Definición 13 Sea $\mathcal{I}$ una interpretación con dominio D. Sea Σ el conjunto de todas las secuencias de elementos de D. Para una secuencia dada $S = (s_1, s_2, \cdots) \in \Sigma$ y para un término t considere la *asignación término* de t con respecto a $\mathcal{I}$ y S denotada como $S^*(t)$, de la siguiente manera:

1. Si t es una variable x_j entonces su asignación término es $s_j \in D$,
2. Si t es una constante a entonces su asignación término es $\mathcal{I}(a) \in D$,
3. Si $r_1, \cdots, r_n$ son las asignaciones término de $t_1, \cdots, t_n$ respectivamente y f' es la asignación del símbolo función f de aridad n (es decir, $S^*(t_i) = r_i, 1 \leq i \leq n$ y $\mathcal{I}(f) = f'$) entonces $f'(r_1, \cdots, r_n) \in D$ es la asignación término de $f(t_1, \cdots, t_n)$.

Definición 14 La satisfacción de una fórmula con respecto a una secuencia y a una interpretación se define inductivamente de la siguiente manera:

1. Si F es una fórmula atómica $P(t_1, \cdots, t_n)$ entonces la secuencia $S = (s_1, s_2, \cdots)$ *satisface* a F si y sólo si $P'(r_1, \cdots, r_n)$ se cumple (es decir, la n-tupla $(r_1, \cdots, r_n)$ está en la relación P'), donde P' es la correspondiente relación de aridad n de la interpretación de P (es decir, $S^*(t_i) = r_i, 1 \leq i \leq n$ y $\mathcal{I}(P) = P'$),
2. S *satisface* a $\neg F$ si y sólo si S no *satisface* a F,
3. S *satisface* a $F \wedge G$ si y sólo si S *satisface* a F y S *satisface* a G,
4. S *satisface* a $F \vee G$ si y sólo si S *satisface* a F o S *satisface* a G,
5. S *satisface* a $F \rightarrow G$ si y sólo si S no *satisface* a F o S *satisface* a G,
6. S *satisface* a $F \leftrightarrow G$ si y sólo si S *satisface* tanto a F como a G o S no *satisface* ni a F ni a G,
7. S *satisface* a $\exists x_i(F)$ si y sólo si existe una secuencia S' que difiere de S a lo más en la i-ésima componente tal que S' *satisface* a F[1],

[1]En otras palabras, una secuencia $S = (s_1, s_2, \cdots, s_i, \cdots)$ *satisface* a $\exists x_i(F)$ si y sólo si existe un elemento c en el dominio D tal que la secuencia $S' = (s_1, s_2, \cdots, c, \cdots)$ *satisface* a F. Aquí, $(s_1, s_2, \cdots, c, \cdots)$ denota la secuencia obtenida de $(s_1, s_2, \cdots, s_i, \cdots)$ al reemplazar la componente i-ésima s_i por c.

8. S *satisface* a $\forall x_i(F)$ si y sólo si toda *secuencia* que difiera de S a lo más en la i-ésima componente *satisface* a F[2].

Definición 15 Una fbf F es *verdadera para la interpretación* $\mathcal{I}$ (alternativamente se dice que a F se le puede dar el *valor de verdad* T), lo cual se escribe como $\models_{\mathcal{I}} F$ o $\mathcal{I}(F) = \mathrm{T}$, si y sólo si toda secuencia en Σ satisface a F. Se dice que una fbf F es *falsa para la interpretación* $\mathcal{I}$ si y sólo si ninguna secuencia en Σ satisface a F.

Si una fórmula no es cerrada entonces alguna secuencia podría satisfacer a la fórmula pero el resto podrían no hacerlo. El valor de verdad de una fórmula cerrada no depende de la secuencia de interpretación, en tal situación la satisfacción de la fórmula sólo es con respecto a la interpretación.

Definición 16 Sea $\mathcal{I}$ una interpretación de un lenguaje de primero orden $\mathcal{L}$. Se dice que $\mathcal{I}$ es un *modelo* para una fbf cerrada F si F es verdadera con respecto a $\mathcal{I}$. Se dice que $\mathcal{I}$ es un *modelo* para un conjunto Γ de fbfs cerradas de $\mathcal{L}$ si y sólo si toda fbf en Γ es verdadera con respecto a $\mathcal{I}$.

Definición 17 Sea Γ un conjunto de fbfs cerradas de un lenguaje de primer orden $\mathcal{L}$. Entonces

1. Γ puede satisfacerse si y sólo si $\mathcal{L}$ tiene al menos una interpretación la cual es un modelo para Γ,
2. Γ es *válida* si y sólo si toda interpretación de $\mathcal{L}$ es un modelo para Γ,
3. Γ no puede satisfacerse si y sólo si ninguna interpretación de $\mathcal{L}$ es un modelo para Γ.

Definición 18 Sea F una fbf cerrada de un lenguaje de priner orden $\mathcal{L}$. Se dice que una fbf cerrada G está *implicada por* F (equivalentemente que F *implica a* G) si y sólo si para toda interpretación $\mathcal{I}$ de $\mathcal{L}$, que $\mathcal{I}$ sea un modelo para F implica que $\mathcal{I}$ es un modelo para G. Se dice que dos fbfs cerradas F y G son *equivalentes* si y sólo si cada una implica a la otra.

Definición 19 Sea Γ un conjunto de fbfs cerradas de $\mathcal{L}$. Se dice que una fbf cerrada F es una *consecuencia lógica de* Γ (que se escribe $\Gamma \models F$) si y sólo si para toda interpretación $\mathcal{I}$ de $\mathcal{L}$, $\mathcal{I}$ es un modelo para Γ implica que $\mathcal{I}$ es un modelo para F. Por tanto, $\Gamma \models F$ significa que toda fórmula en Γ implica a F.

Ejemplo 3 Considere las siguientes dos fórmulas en un lenguaje de primer orden $\mathcal{L}$:

$$P(a)$$
$$\forall x(P(x) \rightarrow Q(f(x)))$$

[2]En otras palabras, una secuencia $S = (s_1, s_2, \cdots, s_i, \cdots)$ *satisface* a $\forall x_i(F)$ si y sólo si para cada elemento c del dominio D la secuencia $(s_1, s_2, \cdots, c, \cdots)$ *satisface* a F. Note que, si S satisface a $\forall x_i(F)$, entonces, como un caso especial, S satisface a F.

Ahora considere una interpretación de $\mathcal{L}$ como sigue (concentrándose sólo en los símbolos que aparecen en las dos fórmulas anteriores):

1. El dominio de interpretación es el conjunto de todos los números naturales,
2. Asigne a a 0,
3. Asigne f a la función sucesor,
4. Suponga que P y Q son asignados a P' y a Q' respectivamente bajo la interpretación siguiente. Un número natural x está en la relación P' si y sólo si x es par. Un número natural x está en la relación Q' si y sólo si x es impar.

Las dos fórmulas son verdaderas bajo la interpretación anterior. Si el símbolo función f hubiera sido interpretado como la función f' donde $f'(x) = x + 2$ entonces la segunda fórmula habría sido falsa.

2.2. Formas normales de cláusulas

Definición 20 Se dice que una fórmula está en *forma normal prenex* si es de la forma

$$\mathcal{Q}_1 x_1 \cdots \mathcal{Q}_n x_n B \tag{2.1}$$

donde cada $\mathcal{Q}_i$ es $\forall$ o $\exists$ y B está libre de cuantificadores. A la fórmula B se le llama matriz.

Definición 21 Se dice que una fórmula en forma normal prenex está en *forma normal conjuntiva de Skolem* si tiene la forma

$$\forall x_1 \cdots \forall x_n B \tag{2.2}$$

donde la matriz B está en forma normal conjuntiva, es decir, una conjunción de disyunción de literales.

Definición 22 Se dice que una fórmula en forma normal conjuntiva de Skolem es una *cláusula* si tiene la forma

$$\forall x_1 \cdots \forall x_n (L_1 \vee \cdots \vee L_n) \quad n \geq 0 \tag{2.3}$$

donde cada L_i es una literal y $x_1, \cdots x_n$ son las variables libres de la disyunción $L_1 \vee \cdots \vee L_n$. Una fórmula se dice que está en *forma clausular* si es una cláusula.

Por conveniencia, escribiremos una cláusula sin sus cuantificadores como la disyunción de literales $L_1 \vee \cdots \vee L_n$ o como el conjunto de literales $L_1, \cdots, L_n$. De tal manera que cuando se dé una cláusula C como una disyunción $L_1 \vee \cdots \vee L_n$ o como un conjunto $L_1, \cdots, L_n$, donde cada L_i es una literal y $x_1, \cdots x_n$ son todas las variables libres que ocurren en las literales L_i, entonces se pensará que tiene la forma de cláusula (2.3).

2.3. Unificación y el principio de resolución

Definición 23 En el contexto de unificación y resolución consideraremos a los términos y a las literales como las únicas *expresiones bien formadas* (o simplemente *expresiones*).

Definición 24 Una *substitución* θ es un conjunto finito de pares de variables y términos, denotado mediante $\{x_1/t_1, \ldots, x_n/t_n\}$, donde las x_is son distintas y cada t_i es diferente de x_i. El término t_i se llama *acotación* (o *instancia* o *ejemplar*) para x_i. θ se llama *substitución básica* si cada t_i es un término básico (sin variables). La substitución dada por el conjunto vacío se llama *substitución vacía* (o *substitución identidad*) y se denota por $\{\}$ o por ε.

Definición 25 En una substitución $\{x_1/t_1, \ldots, x_n/t_n\}$ a $x_1, \cdots, x_n$ se les llama las *variables de la substitución* y a $t_1, \cdots, t_n$ los *términos de la substitución.*

Definición 26 Sean $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ una substitución y E una expresión. La *aplicación* de θ a E, denotada por $E\theta$, es la expresión obtenida al reemplazar simultaneamente cada ocurrencia de la variable x_i en E por el término t_i. En este caso a $E\theta$ se le llama la *instancia* de E dada por θ. Si $E\theta$ es un término básico entonces a $E\theta$ se le llama *instancia básica* de E. También, se refiere a E como una *generalización* de $E\theta$.

Ejercicio 2 Aplique $\theta = \{x/b, y/h(x)\}$ a $E = P(x, f(x), y, g(a))$

Definición 27 Sean $\theta = \{x_1/t_1, \ldots, x_m/t_m\}$ y $\phi = \{y_1/s_1, \ldots, y_n/s_n\}$ dos substituciones. La *composición* $\theta\phi$, de θ y ϕ, es la substitución que se obtiene del conjunto

$$\{x_1/t_1\phi, \cdots, x_m/t_m\phi, y_1/s_1, \cdots, y_n/s_n\}$$

al borrar todo par $x_i/t_i\phi$ tal que $x_i = t_i\phi$ y borrando también todo par y_i/s_i cuando $y_i \in \{x_1, \cdots, x_m\}$.

Ejercicio 3 Sean $\theta = \{x/b, y/h(z)\}$ y $\phi = \{z/c\}$. Obtenga la composición $\theta\phi$.

Lema 1 Para cualquier substitución θ se cumple que $\varepsilon\theta = \theta\varepsilon = \theta$.

Lema 2 Sean E cualquier cadena y α y β dos substituciones arbitrarias. Entonces $(E\alpha)\beta = E(\alpha\beta)$.

Lema 3 Sean α y β dos substituciones arbitrarias. Si $E\alpha = E\beta$ para toda cadena E entonces $\alpha = \beta$.

Lema 4 La composición de substituciones es asociativa, es decir, para cualquiera substituciones α, β, γ se tiene que $(\alpha\beta)\gamma = \alpha(\beta\gamma)$. De tal manera que al escribir una composición de substituciones los paréntesis se pueden omitir.

Definición 28 Sea S cualquier conjunto $\{E_1, \cdots, E_n\}$ de expresiones bien formadas. Entonces el *conjunto desacuerdo* de S, denotado por $\mathcal{D}(\mathrm{S})$, se obtiene localizando la posición del primer símbolo en el que no todas las E_i $(1 \leq i \leq n)$ tienen el mismo símbolo, después se extraen las subexpresiones t_i de E_i que empiezan con dicho símbolo. El conjunto $\{t_1, \cdots, t_n\}$ es el conjunto desacuerdo de S.

Ejercicio 4 Sea $\mathrm{S} = \{P(a, b, x, y), P(a, b, f(x, y), z), P(a, b, g(h(x)), y)\}$. Calcule $\mathcal{D}(\mathrm{S})$.

Definición 29 Un *unificador* de dos expresiones E y E' es una substitución θ tal que $E\theta$ es sintácticamente idéntica a $E'\theta$. Si las dos expresiones no tienen un unificador entonces no son *unificables*. Un unificador θ se llama *unificador más general (umg)* para dos expresiones si para cada unificador α de E y E' existe una substitución β tal que $\alpha = \theta\beta$.

Ejemplo 4 Un umg para las expresiones $P(x, f(a, y))$ y $P(b, z)$ es $\{x/b, z/f(a, y)\}$. Un unificador de estas expresiones es $\{x/b, y/c, z/f(a, c)\}$.

Un umg es único salvo renombramiento de variables. Debido a esta propiedad el umg de dos expresiones se usa muy frecuentemente. El siguiente algoritmo encuentra un umg para un conjunto de expresiones (si existe).

Algoritmo 1 Algoritmo de unificación

Entrada: Un conjunto S de expresiones bien formadas.

Paso 1: Sean $i = 0, \theta_0 = \varepsilon$.

Paso 2: Si $\mathrm{S}\theta_i$ es un conjunto con un sólo elemento entonces $\theta_{\mathrm{S}} = \theta_i$ y regresa θ_{S} como un umg para S.

Paso 3: Si no existen elementos x_i y e_i en $\mathcal{D}(\mathrm{S}\theta_i)$ tal que x_i es una variable y x_i no ocurre en e_i entonces para ya que S no es unificable.

Paso 4: Defina $\theta_{i+1} = \theta_i\{x_i/e_i\}$.

Paso 5: Defina $i = i + 1$ y vaya al paso 2.

Ejercicio 5 Encuentre un umg para $\mathrm{S} = \{P(x, g(y), f(g(b))), P(a, z, f(z))\}$ usando el algoritmo de unificación y diga que pasos va siguiendo.

Ejercicio 6 Encuentre un umg para $\mathrm{S} = \{P(x, x), P(f(a), g(a))\}$ usando el algoritmo de unificación y diga que pasos va siguiendo.

Teorema 1 Sea S cualquier conjunto finito no vacío de expresiones bien formadas. Si S es unificable entonces el algoritmo de unificación (Algoritmo 1) siempre termina en el paso 2 y θ_{S} es un umg de S.

Definición 30 Suponga que un subconjunto del conjunto de todas las literales que ocurren en una cláusula C tiene el mismo signo y que éstas tiene un umg θ. Entonces Cθ se llama un *factor* de C. Si Cθ es una cláusula unitaria entonces se llama *factor unitario* de C.

Ejercicio 7 Sea C $= P(x, a) \vee P(f(y), z) \vee Q(y, z)$. Encuentre un factor para C y diga si es o no factor unitario.

Ejercicio 8 Sea C $= P(x, a) \vee P(f(y), z) \vee P(f(b), a)$. Encuentre un factor para C y diga si es o no factor unitario.

Definición 31 Suponga que C y D son cláusulas sin variables en común, también suponga que L y M son literales que ocurren respectivamente en C y D, que L y M son complementarias y que unifican con un umg θ. Entonces la cláusula $(\mathrm{C}\theta - \{L\theta\}) \cup (\mathrm{D}\theta - \{M\theta\})$ se llama un *resolvente binario* de C y D. Las literales L y M se llaman las *literales resueltas* y C y D son las *cláusulas madre* de la operación de resolución.

Ejemplo 5 Suponga que C $= P(x, z) \vee Q(f(a)) \vee Q(z)$ y D $= \neg Q(f(y)) \vee \neg R(y)$. Considere respectivamente a L y M como $Q(f(a))$ y $\neg Q(f(y))$. Entonces $\theta = \{y/a\}$ y $(\mathrm{C}\theta - \{L\theta\}) \cup (\mathrm{D}\theta - \{M\theta\}) = P(x, z) \vee Q(z) \vee \neg R(a)$.

Definición 32 Un resolvente de las cláusulas C y D es un resolvente binario de C_1 y D_1, donde C_1 es C o un factor de C y D_1 es D o un factor de D. Así un resolvente binario de dos cláusulas C y D es también un resolvente de C y D.

Ejemplo 6 Suponga que C $= P(x, z) \vee Q(f(a)) \vee Q(z)$ y D $= \neg Q(f(y)) \vee \neg R(y)$. Entonces el resolvente $P(x, f(a)) \vee \neg R(a)$ de C y D es un resolvente binario entre el factor de C $P(x, f(a)) \vee Q(f(a))$ y D.

2.4. Teorema de resolución

Definición 33 Si S es cualquier conjunto de cláusulas entonces la *resolución* de S, denotada Res(S), es el conjunto de todas las cláusulas que consisten de los miembros de S junto con todos los resolventes de todos los pares de miembros de S.

Definición 34 Si S es cualquier conjunto de cláusulas entonces la resolución n-ésima de S, denotada Res^n(S), se define como:

$$\begin{aligned} Res^0(\mathrm{S}) &= \mathrm{S} \\ Res^{n+1}(\mathrm{S}) &= Res(Res^n(\mathrm{S})), n = 0, 1, \cdots \end{aligned}$$

De la definición anterior es claro que:

$$Res^0(\mathrm{S}) \subseteq Res^1(\mathrm{S}) \subseteq \cdots \subseteq Res^n(\mathrm{S}) \subseteq \cdots \tag{2.4}$$

Lema 5 Si S es un conjunto finito de cláusulas básicas entonces no todas las inclusiones en la cadena (2.4) son propias.

Teorema 2 (Teorema de resolución básico) Si S es cualquier conjunto finito de cláusulas básicas entonces S es insatisfactible si y sólo si $Res^n(\mathrm{S})$ contiene a la cláusula vacía (denotada por $\square$), para algún $n \geq 0$.

Teorema 3 (Teorema de resolución) Si S es cualquier conjunto finito de cláusulas entonces S es insatisfactible si y sólo si $Res^n(\mathrm{S})$ contiene a la cláusula vacía, para algún $n \geq 0$.

2.5. Procedimiento de refutación

Ahora se puede definir, sobre la base del teorema de resolución (2), un procedimiento para derivar la cláusula vacía a partir de un conjunto insatisfactible de cláusulas.

Definición 35 Sea S un conjunto de cláusulas (llamadas *cláusulas de entrada*). Una *derivación* (o *deducción*) en S es una secuencia de cláusulas $C_1, C_2, \cdots$ tal que cada C_i está en S o es un resolvente de C_j y C_k, donde $1 \leq i \leq n$, $1 \leq j \leq i$ y $1 \leq k \leq i$. A C_i se le llama *cláusula derivada*. Una derivación es *finita* o *infinita* de acuerdo a la longitud de su secuencia.

Definición 36 Una *refutación* de S es una derivación finita $C_1, \cdots, C_n$ en S tal que $C_n = \square$.

El siguiente teorema es el teorema de completitud de un sistema lógico cuya única regla de inferencia es el principio de resolución.

Teorema 4 Un conjunto finito S de cláusulas es insatisfactible si y sólo si existe una refutación de S.

Una de las maneras de encontrar una refutación, de un conjunto de cláusulas insatisfactibles, es calculando la secuencia S, $Res(\mathrm{S}), Res^2(\mathrm{S}), \cdots$ hasta que $Res^n(\mathrm{S})$ contenga $\square$, para algún $n \geq 0$.

Ejemplo 7 Dado el siguiente conjunto S buscaremos una refutación para él:

$$
\begin{aligned}
\mathrm{S} = \quad & 1. \quad \neg P(a) \\
& 2. \quad P(x) \vee \neg Q(x) \\
& 3. \quad P(x) \vee \neg R(f(x)) \\
& 4. \quad Q(a) \vee R(f(a))
\end{aligned}
$$

$$Res(\mathrm{S}) = \mathrm{S} \quad \cup \quad \begin{array}{llll} 5. & \neg Q(a) & 1 \text{ y } 2, & \theta = \{x/a\} \\ 6. & \neg R(f(a)) & 1 \text{ y } 3, & \theta = \{x/a\} \\ 7. & P(a) \vee R(f(a)) & 2 \text{ y } 4, & \theta = \{x/a\} \\ 8. & P(a) \vee Q(a) & 3 \text{ y } 4, & \theta = \{x/a\} \end{array}$$

$$Res^2(\mathrm{S}) = Res(\mathrm{S}) \quad \cup \quad \begin{array}{llll} 9. & R(f(a)) & 1 \text{ y } 7, & \theta = \{\} \\ 10. & Q(a) & 1 \text{ y } 8, & \theta = \{\} \\ 11. & P(a) & 2 \text{ y } 8, & \theta = \{x/a\} \\ 12. & P(a) & 3 \text{ y } 7, & \theta = \{x/a\} \\ 13. & R(f(a)) & 4 \text{ y } 5, & \theta = \{\} \\ 14. & Q(a) & 4 \text{ y } 6, & \theta = \{\} \end{array}$$

$$Res^3(\mathrm{S}) = Res^2(\mathrm{S}) \quad \cup \quad 15. \quad \square \quad 1 \text{ y } 11, \quad \theta = \{\}$$

Como encontramos una refutación para S el conjunto es insatisfactible.

Ejemplo 8 Dado el siguiente conjunto S buscaremos una refutación para él:

$$\mathrm{S} = \begin{array}{ll} 1. & P(x) \vee Q(x) \\ 2. & \neg P(x) \vee \neg Q(x) \\ 3. & \neg P(x) \vee Q(x) \\ 4. & \neg Q(a) \end{array}$$

$$Res(\mathrm{S}) = \mathrm{S} \quad \cup \quad \begin{array}{llll} 5. & Q(x) \vee \neg Q(x) & 1 \text{ y } 2, & \theta = \{\} \\ 6. & Q(x) & 1 \text{ y } 3, & \theta = \{\} \\ 7. & P(a) & 1 \text{ y } 4, & \theta = \{x/a\} \\ 8. & \neg P(x) & 2 \text{ y } 3, & \theta = \{\} \\ 9. & \neg P(a) & 3 \text{ y } 4, & \theta = \{x/a\} \end{array}$$

$$
\begin{array}{lllll}
Res^2(\mathrm{S}) = Res(\mathrm{S}) & \cup & 10. & P(x) \vee Q(x) & 1 \text{ y } 5, \quad \theta = \{\} \\
 & & 11. & Q(x) & 1 \text{ y } 8, \quad \theta = \{\} \\
 & & 12. & Q(a) & 1 \text{ y } 9, \quad \theta = \{x/a\} \\
 & & 13. & \neg P(x) \vee \neg Q(x) & 2 \text{ y } 5, \quad \theta = \{\} \\
 & & 14. & \neg P(x) & 2 \text{ y } 6, \quad \theta = \{\} \\
 & & 15. & \neg Q(a) & 2 \text{ y } 7, \quad \theta = \{x/a\} \\
 & & 16. & \neg P(x) \vee Q(x) & 3 \text{ y } 5, \quad \theta = \{\} \\
 & & 17. & Q(a) & 3 \text{ y } 7, \quad \theta = \{x/a\} \\
 & & 18. & \neg Q(a) & 4 \text{ y } 5, \quad \theta = \{x/a\} \\
 & & 19. & \square & 4 \text{ y } 6, \quad \theta = \{x/a\}
\end{array}
$$

Como encontramos una refutación para S el conjunto es insatisfactible.

Ejercicio 9 Diga si el siguiente conjunto es satisfactible o insatisfactible:

$$
\begin{array}{lll}
\mathrm{S} = & 1. & P(x) \\
 & 2. & P(a) \\
 & 3. & \neg P(x) \vee R(x) \\
 & 4. & \neg R(a)
\end{array}
$$

Ejercicio 10 Diga si el siguiente conjunto es satisfactible o insatisfactible:

$$
\begin{array}{lll}
\mathrm{S} = & 1. & P(x) \vee R(f(a)) \\
 & 2. & \neg P(x) \vee \neg R(g(x)) \\
 & 3. & P(b) \\
 & 4. & R(g(a))
\end{array}
$$

2.6. Cláusulas programa y metas

Una cláusula no vacía se puede reescribir como

$$M_1 \vee \cdots \vee M_p \vee \neg N_1 \vee \cdots \vee \neg N_q, p + q \geq 1 \tag{2.5}$$

donde las M_is y las N_js son literales positivas y las variables están implicitamente cuantificadas universalmente sobre toda la disyunción. La cláusula anterior se puede escribir en la forma de *cláusula programa* de la siguiente manera

$$M'_1 \vee \cdots \vee M'_k \leftarrow N_1 \wedge \cdots \wedge N_q \wedge \neg M'_{k+1} \wedge \cdots \wedge \neg M'_p, k \geq 1, q \geq 0, p \geq 0 \tag{2.6}$$

o en la forma de una *meta* como

$$\leftarrow N_1 \wedge \cdots \wedge N_q \wedge \neg M'_1 \wedge \cdots \wedge \neg M'_p, p + q \geq 1 \tag{2.7}$$

donde cada M'_i es una M_j, para algún j.

Definición 37 Una *cláusula programa* (también *cláusula general*, *regla* o simplemente *cláusula*) es una fórmula de la forma:

$$A_1 \vee \cdots \vee A_m \leftarrow L_1 \wedge \cdots \wedge L_n, m \geq 1, n \geq 0 \tag{2.8}$$

donde $A_1 \vee \cdots \vee A_m$ es la *cabeza* (también *conclusión* o *consecuente*) de la cláusula programa y $L_1 \wedge \cdots \wedge L_n$ es el *cuerpo* (también *condición* o *antecedente*). Cada A_i es un átomo y cada L_j es un átomo (una *condición positiva*) o es un átomo negado (una *condición negativa*). Se asume que cualquier variable que aparezca en $A_1, \cdots, A_m, L_1, \cdots, L_n$ está cuantificada universalmente sobre la fórmula completa. Se dice que cada A_i (respectivamente L_i) ha *ocurrido* en la cabeza (respectivamente en el cuerpo) de la cláusula (2.8).

Dependiendo de los diferentes valores que tomen m y n en la definición de cláusula general (2.8) se obtienen los siguientes tipos de clásulas:

1. Si $m = 1$ y $n = 0$ la cláusula (2.8) toma la forma

   $$A \leftarrow \tag{2.9}$$

   donde el cuerpo es vacío y la cabeza es un solo átomo. Esta cláusula se llama *cláusula unitaria* o *aserción unitaria*.

2. Si $m \geq 1$ y $n = 0$ la cláusula (2.8) toma la forma

   $$A_1 \vee \cdots \vee A_m \leftarrow \tag{2.10}$$

   donde el cuerpo es vacío y la cabeza es una disyunción de átomos. Esta cláusula se llama *cláusula positiva* o *aserción*.

3. Si $m = 1$, $n \geq 0$ y cada L_i es un átomo la cláusula (2.8) toma la forma

   $$A \leftarrow B_1 \wedge \cdots \wedge B_n \tag{2.11}$$

   donde A y las B_js son átomos. Esta cláusula se llama *cláusula definida* o *cláusula de Horn*.

4. Si $m = 1$ y $n \geq 0$ la cláusula (2.8) toma la forma

   $$A \leftarrow L_1 \wedge \cdots \wedge L_n \tag{2.12}$$

   donde donde A es un átomo y las L_js son literales. Esta cláusula se llama *cláusula normal*.

5. Si $m > 1$ y $n \geq 0$ la cláusula (2.8) toma la forma

   $$A_1 \vee \cdots \vee A_m \leftarrow B_1 \wedge \cdots \wedge B_n \tag{2.13}$$

   donde las A_is y las B_js son átomos. Esta cláusula se llama *cláusula disyuntiva* o *cláusula no Horn*.

Al escribir cláusulas unitarias o aserciones se omite el símbolo '$\leftarrow$'.

Ejercicio 11 Diga de qué tipo son las siguientes cláusulas y qué enuncian:

1. $Par(0)$
2. $Positivo(x) \vee Negativo(x)$
3. $Par(s(x)) \leftarrow Entero(x) \wedge Impar(x)$
4. $Par(x) \leftarrow Entero(x) \wedge \neg Impar(x)$
5. $Par(x) \vee Impar(x) \leftarrow Entero(x)$
6. $Entero(x) \vee Fracción(x) \leftarrow Real(x) \wedge \neg Irracional(x)$
7. $Par(s(s(x))) \leftarrow Entero(x) \wedge Par(x)$

Definición 38 Una *meta normal* (también *negación* o simplemente *meta*) es una fórmula de la forma:

$$\leftarrow L_1 \wedge \cdots \wedge L_n, n \geq 1 \tag{2.14}$$

donde cada L_i es un átomo o un átomo negado. La conjunción $L_1 \wedge \cdots \wedge L_n$ es el *cuerpo* de la meta. Las variables que ocurran en una meta se asumen que están cuantificadas universalmente sobre toda la meta. Cuando todas las literales en la meta anterior son positivas, es decir, cuando la meta (2.14) toma la forma

$$\leftarrow B_1 \wedge \cdots \wedge B_n, n \geq 1 \tag{2.15}$$

donde cada B_i es un átomo, se le llama *meta definida*.

Definición 39 Una *submeta de una meta* G es una literal que ocurre en G. Una *submeta de una regla* R es una literal que ocurre en el cuerpo de R.

Definición 40 Un *programa general* (o simplemente *programa*) es un conjunto finito de cláusulas generales. Un *programa disyuntivo* es un conjunto finito de cláusulas disyuntivas. Un *programa normal* es un conjunto finito de cláusulas normales, Un *programa definido* es un conjunto finito de cláusulas definidas.

Si $x_1, \cdots, x_p$ son todas las variables libres en una meta de la forma (2.14), entonces la meta en el contexto de un programa P se interpreta como una petición para hacer una prueba constructiva para la fórmula:

$$\exists x_1, \cdots, \exists x_p (L_1 \wedge \cdots \wedge L_n) \tag{2.16}$$

Lo cual significa que uno debe encontrar una substitución θ para las variables libres en la meta tal que $(L_1 \wedge \cdots \wedge L_n)\theta$ es verdadera de acuerdo a la semántica del programa P.

2.7. Semántica procedural para programas definidos

En el contexto de la programación lógica hay más de una semántica que se puede asignar a un programa P. Para el propósito del curso es suficiente la semántica procedural, esta semántica se basa en un tipo especial de resolución que se llama *resolución-SLD* y se aplica a programas definidos.

Definición 41 Sean P un programa definido y G una meta. Una *derivación-SLD sin restricciones* de $P \cup \{G\}$ consiste de una secuencia $G_0 = G, G_1, \cdots$ de metas, una secuencai $C_1, C_2, \cdots$ de variantes de cláusulas en P (llamadas las *cláusulas de entrada* de la derivación) y una secuencia $\theta_1, \theta_2, \cdots$ de substituciones. Cada meta no vacía G_i contiene un átomo, el cual es el *átomo seleccionado* de G_i. Se dice que la cláusula G_{i+1} es derivada de G_i y C_i con substitución θ_i y se obtiene como sigue.

Suponga que G_i es

$$\leftarrow A_1 \wedge \cdots \wedge A_k \wedge \cdots \wedge A_m, m \geq 1$$

y que A_k es el átomo seleccionado. Sea $C_i = A \leftarrow B_1 \wedge \cdots \wedge B_n$ una cláusula en P tal que A y A_k son unificables con algún unificador θ. Entonces G_{i+1} es

$$\leftarrow (A_1 \wedge \cdots \wedge A_{k-1} \wedge B_1 \wedge \cdots \wedge B_n \wedge A_{k+1} \wedge \cdots \wedge A_m)\theta$$

y defina θ_{i+1} como θ. Una *refutación-SLD sin restricciones* es una derivación que termina con la cláusula vacía.

Definición 42 Las definiciones de *derivación-SLD* y *refutación-SLD* son exactamente las mismas que en la definición (41), pero en lugar de considerar cualquier unificador θ, del átomo seleccionado (A_k) y de la cabeza de la cláusula de entrada (A), se toma al umg de ellos.

Lema 6 Sean P un programa definido, $G = \leftarrow A_1 \wedge \cdots \wedge A_k \wedge \cdots \wedge A_m, m \geq 1$ una meta y θ una substitución. Si existe una refutación de $P \cup \{G\theta\}$ con $A_k\theta$ como el primer átomo seleccionado, entonces existe una refutación de $P \cup \{G\}$ con A_k como el primer átomo seleccionado.

Definición 43 Sean P un programa definido y G una meta o una cláusula vacía. Un *árbol-SLD* para $P \cup \{G\}$ tiene a G como raíz y cada nodo es o una meta o una cláusula vacía. Suponga que

$$\leftarrow A_1 \wedge \cdots \wedge A_k \wedge \cdots \wedge A_m, m \geq 1$$

es un nodo no vacío con átomo seleccionado A_k. Entonces este nodo tiene un descendiente para toda cláusula $A \leftarrow B_1 \wedge \cdots \wedge B_n$ si A y A_k son unificables con un umg θ. El descendiente es

$$\leftarrow (A_1 \wedge \cdots \wedge A_{k-1} \wedge B_1 \wedge \cdots \wedge B_n \wedge A_{k+1} \wedge \cdots \wedge A_m)\theta$$

Cada ruta en un árbol-SLD es una derivación-SLD y una ruta que termina en la cláusula vacía se llama *derivación-SLD exitosa*. El espacio de búsqueda es la totalidad de derivaciones construidas.

Definición 44 Una derivación-SLD puede ser finita o infinita. Una refutación-SLD es una derivación-SLD exitosa. Una *derivación-SLD fallida* es aquella que termina en una meta no vacía, con la propiedad de que el átomo seleccionado en dicha meta no unifica con la cabeza de ninguna cláusula del programa. Las ramas correspondientes con derivaciones exitosas en un árbol-SLD se llaman *ramas exitosas*, las ramas correspondientes con derivaciones infinitas se llaman *ramas infinitas* y las ramas correspondientes con derivaciones fallidas se llaman *ramas fallidas*. Un árbol-SLD se llama *fallido finitamente* si todas sus ramas son fallidas.

En general, dado $\mathrm{P} \cup \{\mathrm{G}\}$ se pueden tener diferentes árboles-SLD dependiendo de que átomos sean considerados los átomos seleccionados.

Ejercicio 12 Considere el siguiente programa P:

C1 $P(x, y) \leftarrow Q(x, y)$

C2 $P(x, y) \leftarrow Q(x, z) \wedge P(z, y)$

C3 $Q(a, b)$

C4 $Q(b, c)$

y la meta $\mathrm{G} = \leftarrow P(a, y)$. Construya al menos tres árboles-SLD para $\mathrm{P} \cup \{\mathrm{G}\}$.

2.8. Semántica procedural para programas normales

Recuerde que una cláusula normal es aquella que tiene un solo átomo en la cabeza y una o más literales en el cuerpo, es decir, que puede tener átomos negados en el cuerpo. También recuerde que la resolución-SLD sólo se puede aplicar a literales positivas, pero notemos que si $\mathrm{P} \cup \{\leftarrow A\}$ tiene un árbol-SLD fallido finitamente entonces A no es una consecuencia lógica de P y así $\neg A$ se puede inferir a partir de P. A esta forma de interpretar la negación se le llama *negación como falla*. A continuación se define como extender la resolución-SLD para implementar la negación como falla, por lo que cuando se diga programa y meta se debe entender como programa y meta normales, al menos que se especifique otra cosa.

Ejercicio 13 ¿Es suficiente la negación como falla para modelar la negación en el sentido del cálculo de predicados? Justifique.

Definición 45 Sean una meta G $=\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$ y una cláusula $C = A \leftarrow M_1 \wedge \cdots \wedge M_q$. Suponga que L_m es una literal, llamada el *átomo seleccionado*, y que θ es el umg de L_m y A. Entonces se dice que la meta

$$\leftarrow (L_1 \wedge \cdots \wedge L_{m-1} \wedge M_1 \wedge \cdots \wedge M_q \wedge L_{m+1} \wedge \cdots \wedge L_k)\theta$$

se ha derivado de G y C usando el umg θ. Una meta así derivada es un resolvente de G y C. Si $k = 1$ y $q = 0$ entonces el resolvente es la cláusula vacía.

Definición 46 Sean P un programa y G una meta. Una *refutación-SLDNF de rango 0* de $\mathrm{P} \cup \{\mathrm{G}\}$ consiste de una secuencia $\mathrm{G} = \mathrm{G}_0, \mathrm{G}_1, \cdots, \mathrm{G}_n = \square$ de metas, una secuencia $C_1, \cdots, C_n$ de variantes de cláusulas de P y una secuencia $\theta_1, \cdots, \theta_n$ de umgs tales que G_{i+1} es derivada de G_i y C_{i+1} usando θ_{i+1}.

Definición 47 Sean P un programa y G una meta. Un *árbol-SLDNF fallido finitamente de rango 0* para $\mathrm{P} \cup \{\mathrm{G}\}$ es un árbol finito que satisface lo siguiente:

1. El nodo raíz es G,
2. cada nodo del árbol es una meta y no una cláusula vacía,
3. en los nodos del árbol sólo los átomos son seleccionados,
4. suponga que $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k (k \geq 1)$ es un nodo interno del árbol y que L_m es la literal positiva seleccionada. Entonces este nodo tiene un descendiente, por cada cláusula $A \leftarrow M_1 \wedge \cdots \wedge M_q$ en P tal que L_m y A son unificables, de la forma:

   $$\leftarrow (L_1 \wedge \cdots \wedge L_{m-1} \wedge M_1 \wedge \cdots \wedge M_q \wedge L_{m+1} \wedge \cdots \wedge L_k)\theta$$

   donde θ es un umg de L_m y A,
5. suponga que $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k (k \geq 1)$ es un nodo hoja en el árbol y que la literal positiva seleccionada de esta meta es L_m. Entonces no existe cláusula en P cuya cabeza unifique con L_m.

Definición 48 Sean P un programa y G una meta. Una *refutación-SLDNF de rango $r + 1$* de $\mathrm{P} \cup \{\mathrm{G}\}$ consiste de una secuencia $\mathrm{G} = \mathrm{G}_0, \mathrm{G}_1, \cdots, \mathrm{G}_n = \square$ de metas, una secuencia $C_1, \cdots, C_n$ de variantes de cláusulas de P y una secuencia $\theta_1, \cdots, \theta_n$ de umgs tales que una de las siguientes condiciones se cumple:

1. Cada G_{i+1} es derivada de G_i y C_{i+1} usando θ_{i+1},
2. suponga que $\mathrm{G}_i = L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k (k \geq 1)$, que la literal seleccionada L_m es una literal negativa básica $\neg A$ y que existe un árbol-SLDNF fallido finitamente de rango r para $\mathrm{P} \cup \{\leftarrow A\}$. Entonces G_{i+1} es

   $$\leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k,$$

   donde θ_{i+1} es la substitución identidad y C_{i+1} es L_m.

Definición 49 Sean P un programa y G una meta. Un *árbol-SLDNF fallido finitamente de rango* $r+1$ para $\mathrm{P} \cup \{\mathrm{G}\}$ es un árbol finito que satisface lo siguiente:

1. El nodo raíz es G,
2. cada nodo del árbol es una meta y no una cláusula vacía,
3. suponga que $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k (k \geq 1)$ es un nodo interno del árbol y que L_m es la literal positiva seleccionada. Entonces este nodo tiene un descendiente, por cada cláusula $A \leftarrow M_1 \wedge \cdots \wedge M_q$ en P tal que L_m y A son unificables, de la forma:

   $$\leftarrow (L_1 \wedge \cdots \wedge L_{m-1} \wedge M_1 \wedge \cdots \wedge M_q \wedge L_{m+1} \wedge \cdots \wedge L_k)\theta$$

   θ es un umg de L_m y A,
4. suponga que $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k (k \geq 1)$ es un nodo interno del árbol, que la literal seleccionada L_m es una literal negativa básica de la forma $\neg A$ y que existe un árbol-SLDNF fallido finitamente de rango r para $\mathrm{P} \cup \{\leftarrow A\}$, entonces el único descendiente para este nodo es

   $$\leftarrow (L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k)$$

5. Sea $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k (k \geq 1)$ un nodo hoja en el árbol y suponga que la literal seleccionada es L_m, entonces una de las siguientes se cumple:
   a) L_m es positiva y no existe cláusula en P cuya cabeza unifique con L_m,
   b) L_m es una literal negativa básica $\neg A$ y existe una refutación-SLDNF de rango r para $\mathrm{P} \cup \{\leftarrow A\}$,
   c) un nodo con la cláusula vacía no tiene descendientes.

Definición 50 Una *refutación-SLDNF* de $\mathrm{P} \cup \{\mathrm{G}\}$ es una refutación-SLDNF de rango r de $\mathrm{P} \cup \{\mathrm{G}\}$, para algún r. Un *árbol-SLDNF fallido finitamente* para $\mathrm{P} \cup \{\mathrm{G}\}$ es un árbol-SLDNF fallido finitamente de rango r para $\mathrm{P} \cup \{\mathrm{G}\}$, para algún r.

Definición 51 Sean P un programa y G una meta. Una *derivación-SLDNF* de $\mathrm{P} \cup \{\mathrm{G}\}$ consiste de una secuencia de metas $\mathrm{G} = \mathrm{G}_0.\mathrm{G}_1, \cdots$, una secuencia de variantes de cláusulas de P o de literales negativas $C_1, C_2, \cdots$ y una secuencia de substituciones $\theta_1, \theta_2, \cdots$ que satisfacen lo siguiente:

1. Si la literal seleccionada de G_i es un átomo entonces G_{i+1} se deriva de G_i y C_{i+1} usando θ_{i+1},

2. si $G_i = \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$, la literal seleccionada L_m es una literal negativa básica $\neg A$ y existe un árbol-SLDNF fallido finitamente para $P \cup \{\leftarrow A\}$, entonces θ_{i+1} es la substitución identidad, C_{i+1} es $\neg A$ y G_{i+1} es
$$\leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k$$

3. si la secuencia $G_0.G_1, \cdots$ es finita entonces una de las siguientes se cumple

   a) la última meta es vacía,

   b) la última meta es $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$, la literal seleccionada L_m es positiva y no existe cláusula programa cuya cabeza unifique con L_m,

   c) la última meta es $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$, la literal seleccionada L_m es una literal negativa básica $\neg A$ y existe una refutación-SLDNF de $P \cup \{\leftarrow A\}$.

Una derivación-SLDNF es finita si consiste de una secuencia finita de metas, de otro modo es infinita. Una derivación-SLDNF es exitosa si es finita y la última meta es la meta vacía. Así una derivación-SLDNF exitosa es una refutación-SLDNF. Una derivación-SLDNF es fallida si es finita y la última meta no es la meta vacía.

Definición 52 Sea P un programa y G una meta. Un *árbol-SLDNF* para $P \cup \{G\}$ es un árbol que satisface lo siguiente:

1. Cada nodo del árbol es una meta o una cláusula vacía,

2. el nodo raíz es G,

3. suponga que $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k (k \geq 1)$ es un nodo interno del árbol y que L_m es la literal positiva seleccionada. Entonces este nodo tiene un descendiente, por cada cláusula $A \leftarrow M_1 \wedge \cdots \wedge M_q$ en P tal que L_m y A son unificables, de la forma:
$$\leftarrow (L_1 \wedge \cdots \wedge L_{m-1} \wedge M_1 \wedge \cdots \wedge M_q \wedge L_{m+1} \wedge \cdots \wedge L_k)\theta$$
   donde θ es un umg de L_m y A,

4. suponga que $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k (k \geq 1)$ es un nodo interno del árbol, que la literal seleccionada L_m es una literal negativa básica de la forma $\neg A$ y que existe un árbol-SLDNF fallido finitamente para $P \cup \{\leftarrow A\}$, entonces el único descendiente para este nodo es
$$\leftarrow (L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k)$$

5. sea $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k (k \geq 1)$ un nodo hoja en el árbol y suponga que la literal seleccionada es L_m, entonces una de las siguientes se cumple:

a) L_m es positiva y no existe cláusula en P cuya cabeza unifique con L_m,

b) L_m es una literal negativa básica $\neg A$ y existe una refutación-SLDNF para $\mathrm{P} \cup \{\leftarrow A\}$,

c) un nodo con la cláusula vacía no tiene descendientes.

En un árbol-SLDNF una rama que termina en una meta vacía es una *rama exitosa*, una rama que no termina es una *rama infinita* y una rama que termina con una meta no vacía es una *rama fallida*. Un árbol-SLDNF para el cual toda rama es una rama fallida se llama *árbol-SLDNF fallido finitamente*. Cada rama de un árbol-SLDNF se corresponde con una derivación-SLDNF.

2.9. Prolog

Prolog, acrónimo de Pro(gramming) (in) Log(ic), es un lenguaje de programación lógica creado por Colmerauer y colaboradores en 1973. Un *programa Prolog normal* es un programa lógico normal, es decir, un conjunto finito de cláusulas normales, similarmente una *meta Prolog normal* es una meta normal. A Prolog se le han incorporado algunas características que no están sustentadas por la lógica de primer orden, por lo que se les llama *características extralógicas*, éstas (entre otras) permiten:

1. Mejorar el desempeño al reducir el espacio de búsqueda, es decir, podando ramas redundantes al árbol de derivación (por ejemplo, usando el símbolo *cut*),
2. "Incrementar legibilidad" (por ejemplo, usando los constructores *if-then/or*),
3. Operaciones de entrada/salida (por ejemplo, usando los predicados *read/write*),
4. Administrar cláusulas (por ejemplo, usando los predicados *assert/retract*),

2.9.1. Sintaxis

La sintaxis de Prolog es muy simple y queda definida por las siguientes reglas:

1. En general, las constantes y los símbolos función y predicado de un programa Prolog empiezan con una letra minúscula (o por el símbolo de 'pesos'), si se desea usar cualquier cadena de caracteres se debe encerrar entre comillas sencillas,
2. las variables empiezan al menos con una letra mayúscula (o por el símbolo de 'guión bajo'). El símbolo '_' es un tipo especial de variable llamada 'no sé'. Las ocurrencias de esta variable particular en una cláusula programa o en una meta se consideran diferentes unas de otras,
3. los símbolos para '$\wedge$' (conjunción), '$\vee$' (disyunción), '$\leftarrow$' (implicación) y '$\neg$' (negación) respectivamente son ',', ';', ':-' y '`not`',

4. los símbolos de funciones aritméticas (por ejemplo, '+', '-', '*', '/') y algunos predicados para comparar (por ejemplo, '<', '>') se pueden escribir en forma infija. Por ejemplo, la suma de dos números `a` y `b` usualmente se expresa como `a+b` en lugar de `+(a,b)` (aunque ésta es también una forma correcta de escribirla),

5. la asignación aritmética se logra mediante el predicado `is` y la igualdad aritmética se representa mediante el símbolo '=:=',

6. la unificación explícita de dos términos se logra usando el símbolo '=', para checar si dos términos son sintácticamente iguales se usa el símbolo '=='.

Al describir una meta, una submeta o una pieza de código, la ocurrencia de una expresión de la forma `type` o `type_n` se puede reemplazar por una expresión arbitraria que sea del tipo `type`. Por ejemplo, una submeta `square(<integer_1>, <integer_2>)` significa que los argumentos del predicado `square` son enteros arbitrarios.

Con las anteriores convenciones podemos expresar la sintaxis de una cláusula Prolog normal mediante:

```
<atom>:- <literal_1>, <literal_2>, ..., <literal_n>.
```

donde `n`≥ 0. Toda cláusula en un programa o meta Prolog se termina con el símbolo '.'. Como ejemplo, la cláusula normal

$$Even(x) \leftarrow Integer(x) \wedge \neg Odd(x)$$

en sintaxis de la lógica de primer orden se expresa en la sintaxis de Prolog como

```
even(X):- integer(X), not odd(X).
```

Una meta para un programa Prolog se expresa por el símbolo '?-' seguida de una conjunción de literales y terminada con el símbolo '.'. Así una meta Prolog normal tiene la forma

```
?- <literal_1>, <literal_2>, ...,  <literal_n>.
```

donde `n`≥ 1. Como ejemplo, la meta para encontrar todos los números pares no cuadrados es

```
?- even(X), not square(X).
```

Las ordenes a un intérprete Prolog se dan en forma de metas.