Lenguajes de Programación

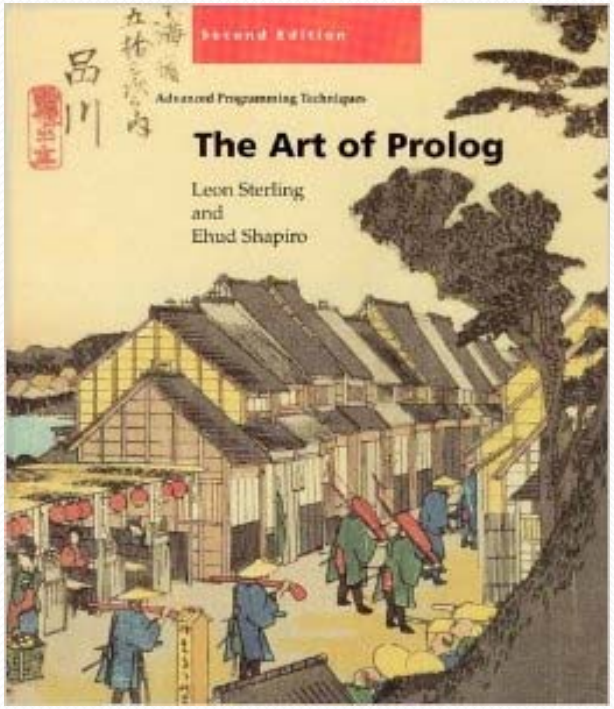
Dr. Carlos A. Coello Coello
Departamento de Computación
CINVESTAV-IPN
ccoello@cs.cinvestav.mx

PROLOG (Antecedentes)

- La programación en los lenguajes orientados a la lógica es no procedural.

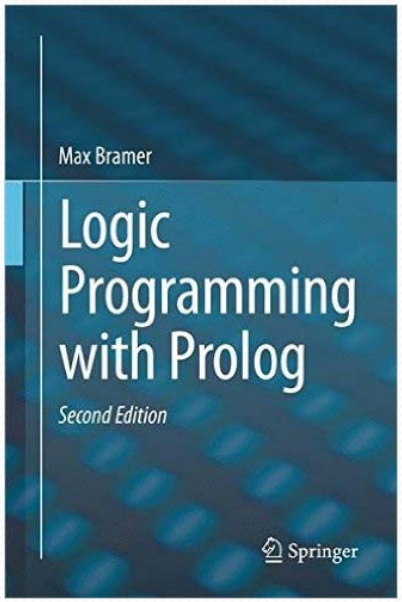
- Los programas en estos lenguajes no especifican exactamente “cómo” calcular un cierto resultado, sino que más bien describen la forma que debe tener dicho resultado.

PROLOG (Antecedentes)



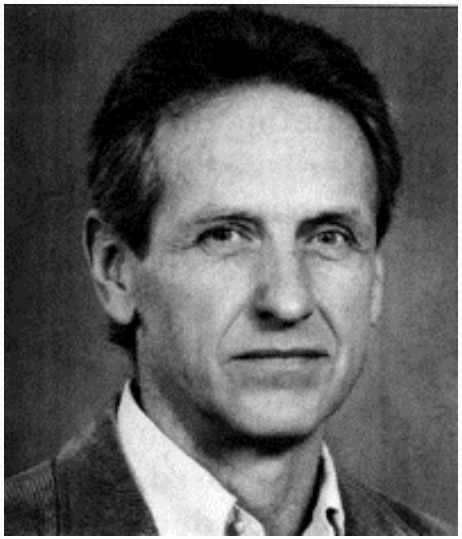
- Para proporcionar estas capacidades, se requieren medios concisos de suministrar a la computadora tanto la información relevante como un método de inferencia para obtener los resultados deseados.

PROLOG (Antecedentes)



- El cálculo de predicados proporciona la forma básica de comunicación con la computadora, y el método de demostración conocido como “resolución” (desarrollado por Robinson en 1965) proporciona el mecanismo de inferencia.

PROLOG (Antecedentes)

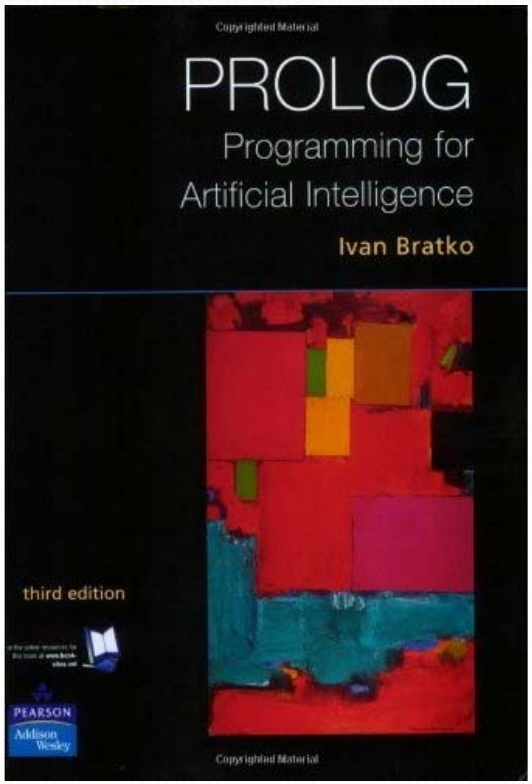


- A principios de los 1970s, Alain Colmerauer y Philippe Roussel, del Grupo de Inteligencia Artificial de la Universidad de Aix-Marseille (en Francia), junto con Robert Kowalski, del Departamento de Inteligencia Artificial de la Universidad de Edimburgo, desarrollaron el diseño fundamental de PROLOG.

PROLOG (Antecedentes)

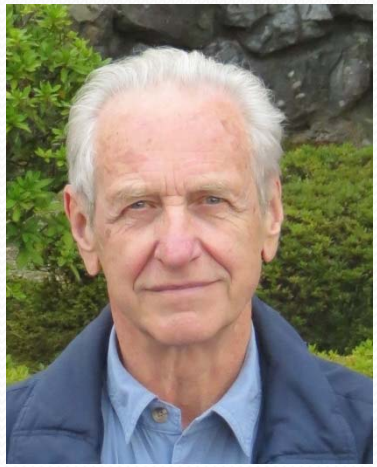
- El primer intérprete de PROLOG se desarrolló en Marsella en Algol-W, en 1972. Luego hubo una versión en FORTRAN, en 1973 y otra en Pascal en 1976.
- La versión del lenguaje que se desarrolló en aquel entonces se describe en una publicación de Roussel de 1975.
- El nombre PROLOG se deriva de "**Programming in Logic**".

PROLOG (Antecedentes)



- La programación no procedural resulta estar relacionada con un área de investigación de IA muy activa en aquel entonces: la demostración automática de teoremas.

PROLOG (Antecedentes)



- La programación lógica hace uso explícito de la observación, hecha a inicios de los 1970s por Pat Hayes, Robert Kowalski, Cordell Green y otros de que el aplicar métodos de deducción estándar frecuentemente tiene los mismos efectos que ejecutar un programa.

PROLOG (Antecedentes)

- Los programas en PROLOG se expresan en forma de proposiciones que aseveran la existencia del resultado deseado.

- El demostrador de teoremas del lenguaje debe entonces construir el resultado deseado para demostrar su existencia.

Perfil de PROLOG

- Un programa en PROLOG consiste de 3 componentes:

  1) Un conjunto de hechos

  2) Un conjunto de reglas o relaciones

  3) Una consulta (*query*)

Perfil de PROLOG

- El intérprete de PROLOG intenta “resolver” o responder la consulta usando los hechos y las reglas proporcionadas.
- Un hecho es una sentencia que expresa una relación entre objetos; una regla relaciona grupos de hechos.
- Una regla es una cláusula de Horn y un hecho es un átomo (en el Cálculo de Predicados).

Perfil de PROLOG

- Ejemplo:

```
mujer(annette).
mujer(marilyn).
mujer(audrey).
padres(annete, fred, marilyn).
padres(audrey, fred, marilyn).
padres(marilyn, john, liz).
hermana(X,Y):-mujer(X),padres(X,M,F),padres(Y,M,F).
```

Perfil de PROLOG

- Podemos preguntar ahora “¿son hermanas annette y audrey?”:

  ?:-hermana(annette, audrey).

- La respuesta es: **yes.**

Perfil de PROLOG

- Pero si preguntamos si annette y marilyn son hermanas:

  ?:-hermana(annette, marilyn).

- La respuesta es: **No.**

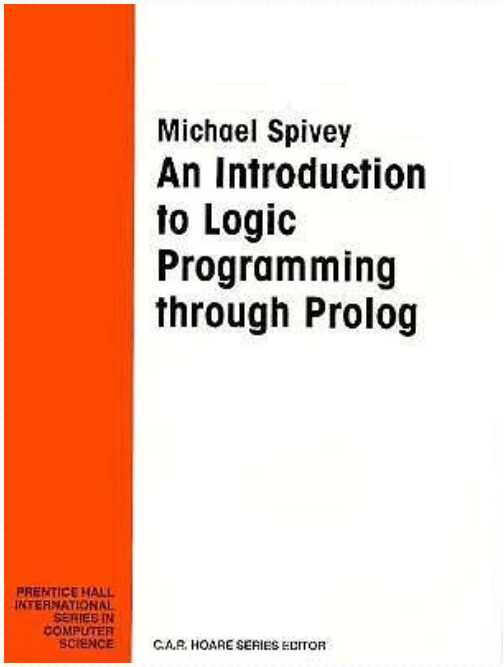
Perfil de PROLOG

- PROLOG puede encontrarse tanto en forma de intérprete como de compilador.
- Por lo tanto, es posible escribir consultas de manera interactiva (en el intérprete) y esperar a que se nos proporcionen respuestas de manera directa.

Perfil de PROLOG

- El programa en PROLOG antes indicado, contiene un número de hechos que indican el sexo y parentesco de diferentes objetos.
- También contiene una sola regla que relaciona estos hechos.
- El término “hermana (X,Y)” es llamado la “cabeza” de la regla; los términos a la derecha del “:-” constituyen el “cuerpo” de la regla.

Perfil de PROLOG



- Las letras mayúsculas se usan en PROLOG para denotar variables (a las que suele denominarse "variables lógicas", porque obtienen un solo valor vía el proceso de unificación.

Perfil de PROLOG

- El intérprete respondió que **sí** a la primera consulta, porque existe una instanciación consistente de variables X & Y, en las reglas proporcionadas que hacen que la relación “hermana” pueda derivarse.

- El intérprete respondió **no** a la segunda consulta porque las reglas requeridas no existen en la base de datos (o conocimiento).

Perfil de PROLOG

- Una consulta puede contener también variables lógicas.
- La respuesta que retorna el intérprete en este caso son TODAS las asociaciones que correspondan a la variable utilizada.

Perfil de PROLOG

- De tal forma, si preguntamos:

  ?:-hermana(X, audrey).

- El intérprete responderá:

  X=annette

Perfil de PROLOG

- También es posible dejar un término sin nombre.
- Esto suele usarse cuando queremos saber únicamente si hay una respuesta a nuestra consulta (p.ej., queremos saber si audrey tiene hermanas), pero no nos interesa la respuesta en sí (p.ej., los nombres de las hermanas de audrey).

Perfil de PROLOG

- Ejemplo:

  ?:-hermana(_, audrey).

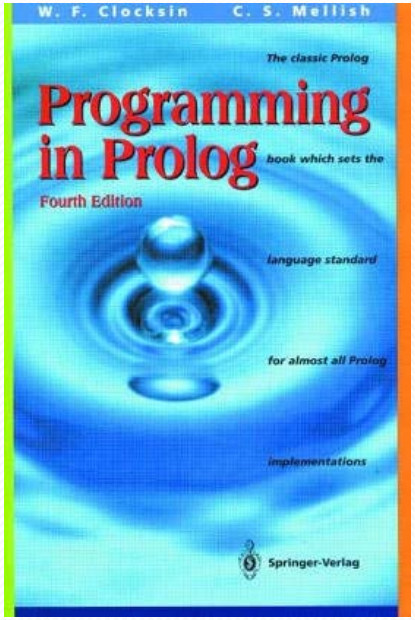
- La respuesta es: **Yes.**

- El símbolo "_" se denomina "variable anónima".

Tipos de Datos

- Los únicos tipos de datos primitivos en PROLOG son los enteros, los caracteres y las cadenas.
- No existe un tipo específico para los Booleanos, aunque pueden simularse ciertas operaciones Booleanas.
- El PROLOG puro no puede expresar la negación, pero la mayor parte de las implementaciones proporcionan una interpretación de la misma.

Tipos de Datos



- PROLOG soporta el conjunto estándar de operadores relacionales, mejor conocidos como reglas, las cuales tienen éxito si la relación necesaria se cumple y fracasan, si no es así.

Tipos de Datos

- El tipo estructurado principal en PROLOG es la lista (list).
- Una lista es una secuencia ordenada de términos (puede incluir listas anidadas) escritos entre corchetes rectangulares.
- Consideremos un ejemplo en el que queremos determinar si un cierto elemento es miembro de una lista:

Tipos de Datos

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

- El símbolo "|" se usa para dividir una lista en su "cabeza" y su "cola".
- Este programa dice: "X es un miembro de una lista cuya cabeza es X. X también es miembro de una lista cuya cola es Y, si X es miembro de Y".

Tipos de Datos

- Ejemplo de uso:

  ?:-member(foo, [bar, bam, zoom, foo, blat]).

- Regresa: **Yes.**

Tipos de Datos

- Otro ejemplo es la definición de “append”, que ilustra el uso del proceso de sustitución consistente (o “unificación”) para escribir un procedimiento “ordinario” (en contraposición a una función Booleana que puede interpretarse naturalmente como una consulta):

Tipos de Datos

```
append([ ], L, L).
append([X|L1], L2, [X|L3]):-append(L1, L2, L3).
```

- El primer hecho puede ser interpretado como: agregar una lista vacía a otra lista L, produce L.

Tipos de Datos

- La regla significa:

- “Si el “append” de una lista cuya cabeza es X y cuya cola es L1 con la lista L2 es una lista cuya cabeza es X y cuya cola es L3, entonces el “append” de L1 y L2 es L3”.

Tipos de Datos

- La “consulta” o invocación a procedimiento:

  ?:-append([a, b, c], [d, e, f], X).

- Regresa:

- X=[a,b,c,d,e,f].

Tipos de Datos

- Algo interesante es que “append” puede usarse también para resolver el problema inverso:

- “Dada una lista L1 y una lista L2, regresa la lista L3 tal que L2 sea el “append” de L1 y L3”.

Tipos de Datos

- La consulta es:

  ?:-append([a,b,c], X, [a,b,c,d,e,f]).

- La respuesta es:

- X=[d,e,f].

Tipos de Datos

- El proceso de unificación es insensible a la posición de las variables en una consulta.
- Este proceso simplemente intenta encontrar una sustitución consistente basada en los hechos y las reglas proporcionadas.

Estructuras de Datos

- En PROLOG no existen constructores de estructuras de datos.
- En vez de eso, las estructuras de datos se definen implícitamente mediante sus propiedades.
- Este mismo tipo de enfoque puede usarse para definir todos los tipos de datos, incluyendo datos primitivos tradicionales como los enteros.

Estructuras de Datos

- Por ejemplo, los números naturales y la aritmética sobre ellos puede definirse mediante cláusulas como las siguientes:

```
sum(succ(X),Y,succ(Y)) :- sum(X,Y,Z).
sum(o,X,X).
dif(X,Y,Z) :- sum(Z,Y,X).
```

Estructuras de Datos

- La primera cláusula dice que la suma del sucesor de X más Y es el sucesor de Z, si la suma de X y Y es Z.
- La segunda cláusula dice que la suma de cero y X es X.
- La tercera cláusula dice que la diferencia de X y Y es Z, si la suma de Z y Y es X. En efecto, esta es una definición recursiva de la aritmética.

Estructuras de Datos

- Si ahora escribimos:
- :- sum(succ(succ(0)),succ(succ(succ(0))),A).
- (Esto significa '2+3=A'), PROLOG hará lo siguiente:
- A = succ(succ(succ(succ(succ(0)))))
- (el resultado es A=5)

Estructuras de Datos

- Aunque estas definiciones de la suma y la resta son correctas, serían muy ineficientes en la práctica, puesto que todas las computadoras implementan directamente la aritmética de enteros.

- Por tanto, todas las implementaciones de PROLOG cuentan con ciertos predicados y funciones para realizar operaciones aritméticas básicas.

Estructuras de Datos

- Desafortunadamente, la implementación de dichas operaciones suele comprometer las propiedades lógicas del lenguaje.

- Por ejemplo, la suma no puede realizarse “hacia atrás” como en nuestra implementación de “dif” mostrada anteriormente.

- El pequeño número de tipos de datos y operaciones nativas de PROLOG ilustra el **Principio de Simplicidad**. El tratamiento uniforme de todos los tipos de datos como predicados y términos, sigue el **Principio de Regularidad**.

Estructuras Complejas

- En PROLOG es posible definir estructuras complejas de manera directa usando términos o predicados.
- Por ejemplo, supongamos que queremos escribir un programa para realizar diferenciación simbólica.
- En la mayor parte de los lenguajes de programación, sería necesario definir estructuras de datos que representen los árboles con las expresiones algebraicas.

Estructuras Complejas

- En PROLOG sólo tenemos que usar términos compuesto. Por ejemplo, la expresión 'x²+bx+c' puede representarse como:

  plus(plus(sup(x,2),times(b,x)),c)

- Puesto que muchos dialectos de PROLOG interpretan a los operadores aritméticos como funciones para construir términos compuestos, podríamos escribir:

  - x^2 + b*x + c

Estructuras Complejas

- Dada la notación infija que usa PROLOG para sus términos compuestos, es bastante fácil escribir y leer un programa para realizar diferenciación simbólica (el código se muestra en el acetato siguiente).

- En este caso, los términos se representan a sí mismos. No hay necesidad para una estructura de datos separada que manipule el programa.

- PROLOG sigue el **Principio de Automatización**.

Estructuras Complejas

```
d(X,U+V,DU+DV) :- d(X,U,DU), d(X,V,DV).
d(X,U-V, DU-DV) :- d(X,U,DU), d(X,V, DV).
d(X,U*V, DU*V+U*DV) :- d(X,U,DU), d(X,V,DV).
d(X, U/V, (DU*V-U*DV)/V^2) :- d(X,U,DU), d(X,V,DV).
d(X, U^C, C*U^(C-1)*DU) :- d(X,U,DU), atomic(C), C \= X.
d(X,X,1).
d(X,C,0) :- atomic(C), C \= X.
```

Predicados para Representar Estructuras

- Los términos compuestos pueden reemplazarse frecuentemente por predicados.
- Por ejemplo, en vez de tener términos como ‘plus(X,Y)’, ‘times(X,Y)’, etc., para representar expresiones aritméticas, podríamos usar los predicados ‘sum(X,Y,Z)’, ‘prod(X,Y,Z)’, etc.
- Consideremos las consecuencias de esto en más detalle.

Predicados para Representar Estructuras

- Consideremos el término compuesto '(x+y)*(y+1)' que representa una expresión aritmética. Esta es una descripción de una estructura de datos (un árbol), pues puede escribirse de la siguiente forma en notación prefija:

  times(plus(x,y),plus(y,1))

- Esto es análogo a la siguiente estructura en LISP:

  (times (plus x y) (plus y 1))

Predicados para Representar Estructuras

- ¿Cómo podemos expresar este mismo tipo de relaciones en términos de predicados?
- En efecto, necesitamos describir un árbol en términos de relaciones entre sus nodos.
- Podemos describir las relaciones en un árbol de expresiones aritméticas mediante predicados tales como ‘sum’ y ‘prod’.

Predicados para Representar Estructuras

- Por ejemplo, hagamos que 'sum(X,Y,Z)' signifique que la suma de X y Y es Z. Entonces la expresión '(x+y)*(y+1)' se describe mediante 3 hechos:

```
sum(x,y,t1).
sum(y,1,t2).
prod(t1,t2,t3).
```

Puesto que podemos decir que la suma de 'x' y 'y' es 't1', la suma de 'y' y '1' es 't2' y el producto de 't1' y 't2' es 't3'.

Predicados para Representar Estructuras

- Este ejemplo ilustra la primera dificultad de representar estructuras de datos como predicados: la falta de legibilidad.

- Supongamos que quisiéramos re-escribir nuestro programa para diferenciación simbólica de manera que funcione con expresiones representadas como predicados.

Predicados para Representar Estructuras

- Las reglas para la suma y el producto se verían de la forma siguiente:

```
d(X,W,Z) :- sum(U,V,W),d(X,U,DU),d(X,V,DV), sum(DU,DV,Z).
d(X,W,Z) :- prod(U,V,W), d(X,U,DU), d(X,V,DV).
                prod(DU,V,A), prod(U,DV,B), sum(A,B,Z).
d(X,X,1).
D(X,C,0) :- atomic(C), C \= X.
```

Predicados para Representar Estructuras

- La regla de la suma no se ve mal, es bastante legible: “La derivada con respecto a X de W es Z, si: W es la suma de U y V, y la derivada con respecto a X de U es DU, y …”.

- La regla del producto, sin embargo, no luce tan bien como la de la suma. Las variables ‘A’ y ‘B’ no ayudan mucho a su legibilidad. La única razón por la que aparecen es porque las necesitamos como nodos intermedios del árbol.

Predicados para Representar Estructuras

- Sin embargo, hay problemas más sutiles. Supongamos que tenemos el siguiente hecho en nuestra base de datos:

  sum(x,1,z).

- Esto dice que la suma de 'x' y 1 es 'z'. Ahora supongamos que ponemos la meta siguiente:

  :- d(x,z,D).

Predicados para Representar Estructuras

- Esto le pide al sistema que encuentre una ‘D’ tal que la derivada con respecto ‘x’ de ‘z’ sea ‘D’.
- Puesto que ‘z’ es la suma de ‘x’ y 1, esperamos que ‘D’ sea la suma de 1 y 0.
- ¿Se logrará satisfacer esta meta?

Predicados para Representar Estructuras

- Sin embargo, en este caso se unifica la cabeza de la regla de la suma con la asignaciones X = x, W = z, Z = D, lo cual produce las siguientes sub-metas:

:- sum(U,V,z), d(x,U,DU), d(x,V,DV), sum(DU,DV,D).

- La primera sub-meta se satisface mediante U=x y V = 1, puesto que tenemos en nuestra base de datos 'sum(x,1,z)'.

Predicados para Representar Estructuras

- Por tanto, obtenemos las sub-metas:

  :- d(x,x,DU), d(x,1,DV), sum(DU,DV,D).

- Ahora las primeras dos sub-metas se satisfacen mediante DU = 1 y DV = 0, de manera que sólo queda una sub-meta:

  :- sum(1,0,D).

Predicados para Representar Estructuras

- Parece que nuestro programa ha terminado, pero de hecho, esta sub-meta fallará, puesto que no existe ningún hecho (o regla) de la forma '(sum(1,0,D))' en la base de datos.

- Claro que si hubiésemos tenido en nuestra base de datos lo siguiente:

  sum(1,0,a).

- El sistema habría devuelto la respuesta correcta: 'D = a'.

Predicados para Representar Estructuras

- Sin embargo, no es muy razonable que hubiésemos anticipado esto.

- Además, si la respuesta hubiese sido más complicada, la situación habría sido peor, porque habríamos tenido que anticipar toda su expresión, pues de lo contrario, el sistema habría fallado.

La Premisa del Mundo Cerrado

- ¿Por qué tiene PROLOG este comportamiento anti-intuitivo?
- Las reglas deductivas de PROLOG se basan en la llamada premisa del mundo cerrado.
- Esto significa que, en lo que a PROLOG concierne, todo lo que es cierto acerca del mundo es lo que puede demostrarse con base en los hechos y reglas contenidos en su base de datos.

La Premisa del Mundo Cerrado

- En muchas aplicaciones, esta es una suposición razonable.
- En muchos casos, tiene sentido suponer que no existe un objeto que tenga una cierta propiedad, si éste no se encuentra en nuestra base de datos.
- Esta premisa es muy razonable en aplicaciones orientadas a los objetos, o sea aplicaciones en las cuales los objetos y sus relaciones modelan los objetos y relaciones del mundo real.

La Premisa del Mundo Cerrado

- Sin embargo, la premisa del mundo cerrado no funciona adecuadamente en problemas matemáticos.
- En matemáticas, se suele suponer que un objeto existe si su existencia no contradice los axiomas que tenemos.
- Matemáticamente, es natural suponer que una expresión de la forma '1+0' existe cuando se le necesite.

La Premisa del Mundo Cerrado

- Para un humano, su existencia no tiene que establecerse explícitamente ni tiene que demostrarse a partir de un conjunto de hipótesis.

- Evidentemente, podríamos axiomatizar automáticamente la existencia de todas las expresiones aritméticas que necesitemos, pero esto sería anti-natural e implicaría un enorme esfuerzo que no parece valer la pena realizar.

La Premisa del Mundo Cerrado

- Para resumir, es posible reemplazar términos compuestos por predicados que conecten sus partes, pero la premisa del mundo cerrado dificulta que podamos garantizar que los objetos correctos estén ahí cuando los necesitemos.

- Lo más conveniente es usar los predicados para definir objetos que modelen el mundo real, pero no para operaciones matemáticas.

Estructuras de Control

- Los programas lógicos no tienen estructuras de control en el sentido usual.
- En un lenguaje de programación convencional, las estructuras de control determinan el orden en el cual se ejecutarán las acciones que conforman un programa.
- Este orden es esencial para la ejecución correcta del programa. Por tanto, la lógica de un programa se relaciona íntimamente con su control.

Estructuras de Control

- En contraste, la programación lógica realiza una mayor separación de la lógica y el control.
- El orden en que las cláusulas de un programa se escriben, no debiera tener efecto en el significado del programa.
- En otras palabras, la lógica del programa es determinada por las interrelaciones lógicas de las cláusulas y no por su relación física.

Estructuras de Control

- El control afecta el orden en el cual ocurren las acciones en el tiempo.
- Las únicas acciones que ocurren en la ejecución de un programa lógico son la generación y la unificación de sub-metas.
- Sin embargo, este orden tiene un enorme efecto en la eficiencia del programa.

Estructuras de Control

- Por lo tanto, en la programación lógica el control afecta sólo el desempeño de los programas (o sea, su eficiencia), pero no su significado.

- La separación de la lógica y el control es una importante aplicación del Principio de Ortogonalidad.

- Significa que un programa puede ser desarrollado en dos fases distintas: el análisis lógico y el análisis del control.

Estructuras de Control

- El análisis lógico se refiere únicamente a verificar que el programa sea correcto (o sea, a producir las cláusulas correctas para generar la salida deseada).

- El análisis de control se enfoca exclusivamente en la eficiencia del programa. Esta parte puede realizarla directamente la implementación del lenguaje lógico o puede ser manipulada (al menos parcialmente) por el programador.

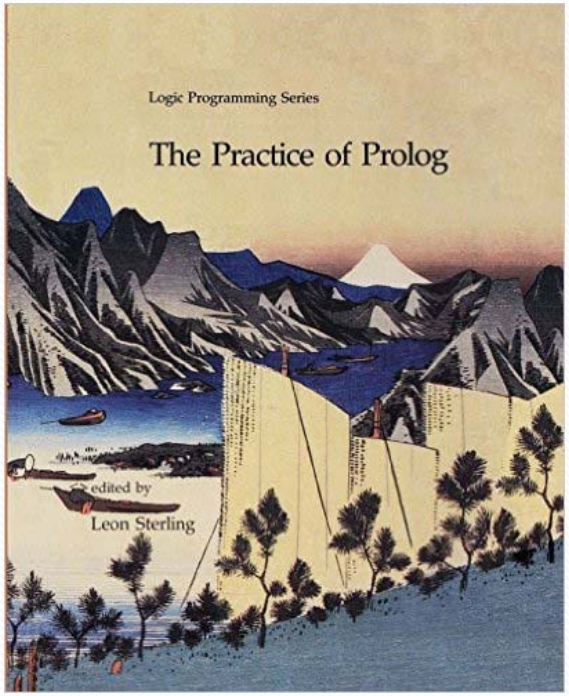
Estructuras de Control

- Cada una de las cláusulas de una consulta se conoce como “meta”.
- El intérprete de PROLOG intenta satisfacer cada meta encontrada en la consulta usando los hechos y reglas proporcionadas.

Estructuras de Control

- Una meta se satisface si:
  - Existe un hecho que se unifique con ella, o
  - Existe una regla cuya cabeza se unifique con la meta de tal forma que cada cláusula en el cuerpo de la regla sea satisfecha (con base en el conjunto actual de asociaciones).

Estructuras de Control



- Si una cierta meta no puede satisfacerse, entonces el intérprete hace “retroceso” (*backtracking*) e intenta encontrar un medio alternativo por el cual pueda satisfacerse la meta.

Estructuras de Control

- Ejemplo:

```
hombre(edward).
hombre(henry).
madre(edward, victoria).
madre(henry, liz).
reina(liz).
principe(X):-hombre(X), madre(X,Z), reina(Z).
```

Estructuras de Control

- La consulta es:

  ?:-principe(Y).

- La respuesta es:

- Y=henry.

Estructuras de Control

- Para resolver esta consulta, el intérprete unifica la cabeza de la regla definiendo lo que significa ser un príncipe con una sola meta en la consulta.
- El efecto de esta unificación es una asociación de Y a X.
- Posteriormente, intenta satisfacer cada cláusula en el cuerpo de la regla.

Estructuras de Control

- Primero, usa “edward” y se da cuenta que “victoria” es su madre.
- Sin embargo, puesto que no hay cláusula que unifique “victoria” con “reina”, se hace retroceso para tratar de obtener otra “madre” para “edward”.
- Como no hay ninguna, entonces se hace retroceso para escoger “henry” y finalmente resuelve la consulta después de lograr unificar todas las cláusulas.

Estructuras de Control

- La estrategia de evaluación del intérprete está basada en una búsqueda primero en profundidad, de arriba hacia abajo.
- El intérprete intenta satisfacer cada cláusula en una meta de izquierda a derecha, buscando los hechos y reglas relevantes proporcionados en el programa.
- El recorrido es de arriba hacia abajo.

Estructuras de Control

- Si en el ejemplo anterior hubiésemos intercambiado la posición de los hechos uno y dos, no hubiese ocurrido ningún “retroceso”.

- En PROLOG, aunque los elementos de un programa son meras declaraciones de hechos y reglas, el orden en que éstas aparecen es altamente significativo para el comportamiento del programa.

Estructuras de Control

- Ejemplo:

```
q(foo).
P(X):-p(X).
P(X):-q(X).
```

- La consulta: **?:-p(Z)** nunca terminará (la segunda cláusula se elegirá eternamente).

Estructuras de Control

- Sin embargo, si intercambiamos las posiciones de la segunda y la tercera cláusula, la consulta producirá la respuesta correcta:

Z = foo