Lenguajes de Programación

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

Estructuras de Control

- PROLOG permite un control limitado sobre la forma en que se realiza el retroceso, vía el operador "**cut**" (!).

- Una regla de la forma:

  r(X):-f(X), !, g(X).

- Indica que una vez que f(X) se satisfaga, no queremos que se efectúe más retroceso.

Estructuras de Control

- La asociación para X derivada de satisfacer la primera cláusula no puede ser revocada incluso si la cláusula g(X) falla.

- Por ejemplo, podemos usar “cut” para definir una expresión condicional de la forma:

f(X)= if p(X) then g(X) else h(X)

Estructuras de Control

- Esto se hace usando:

```
f(X,Y):-p(X), !, g(X,Y).
f(X,Y):-h(X,Y).
```

- Y se almacena el resultado de la condición.

Estructuras de Control

- Si el predicado p(X) arroja cierto, es imposible que se evalúe la rama correspondiente al “falso” de la condición anterior.
- Al incorporar “cut” dentro del programa, hemos hecho explícita esta aseveración.
- El operador “cut” juega un papel importante en PROLOG ya que restringe el proceso de búsqueda y elimina la generación de resultados indeseables.

Estilo Procedural

- El hecho de que PROLOG tenga un orden de búsqueda predefinido combinado con funciones impuras como “cut”, permiten que un programador adopte un estilo de programación muy procedural en PROLOG. Por ejemplo:

```
squares_to(N) :- asserta(current(1)),
                 repeat,
                 one_step,
                 current(N).
```

Estilo Procedural

```
one_step :-  current(K),
             Sq is K*K,
             write(Sq), nl,
             NewK is K+1,
             retract(current(K)),
             asserta(current(NewK)),
             !.
```

Estilo Procedural

- El predicado '**repeat**', que se incluye en la mayor parte de las versiones de PROLOG existentes, se define mediante las cláusulas siguientes:

```
repeat.
repeat :- repeat.
```

* Esta definición luce bastante extraña, pues dice algo como lo siguiente: *repeat* es cierto; así mismo, *repeat* es cierto si *repeat* es cierto.

Estilo Procedural

- Veamos cómo funciona el programa anterior si la meta es:

```
:-squares_to(5).
1
4
9
16
yes
```

Estilo Procedural

- Lo que hace este programa es imprimir los cuadrados de los números del 1 a un valor previo al parámetro (5 en este caso).

- Veamos cómo hace PROLOG para satisfacer esta meta. Para satisfacer ':- squares_to(5)', PROLOG debe satisfacer, en orden, las siguientes sub-metas:

```
:- asserta(current(1)),repeat,one_step,current(5).
```

Estilo Procedural

- El comando ‘asserta’ introduce el hecho ‘current(1)’ en la base de datos y se reporta éxito. Luego se ejecuta el ‘repeat’ y también se logra el éxito, porque el hecho ‘repeat’ se encuentra en la base de datos. Procedemos ahora a satisfacer ‘one_step’. Primero se ejecuta ‘current(1)’. Luego hacemos que Sq se asocie con 1*1 (elevamos al cuadrado el valor de K). La cláusula siguiente imprime ‘1’ y se ejecuta un “newline” (nl) para pasar a la línea siguiente. Las siguientes líneas eliminar ‘current(K)’ de la base de datos y agregan ‘current(2)’, porque NewK es K+1 (y K=1).

Estilo Procedural

- El cut (!) evita que se re-ejecute ‘one_step’ y regresamos el control a ‘squares_to’. Intentamos satisfacer ‘current(5)’, pero como ‘current(2)’ es el único hecho relevante en la base de datos, este intento falla. Ahora debemos hacer el retroceso (backtracking). Puesto que ya usamos ‘cut’ en ‘one_step’, no podemos ejecutarla de nuevo y nos vamos al ‘repeat’. Como ‘repeat’ es un hecho, podemos aplicar la regla del ‘repeat’, pero de una forma diferente a la vez anterior. Este ‘repeat’ permite invocar de nuevo ‘one_step’, que imprimirá 4 (2*2). El proceso se repite hasta que se satisface ‘squares_to(5)’, que es donde se detiene el programa.

Estilo Procedural

- Este programa tiene ciertamente un estilo totalmente procedural. El predicado ‘current’ es realmente una variable y el uso de ‘retract’ y ‘assert’ lo que hacen es implementar asignaciones a esta variable. El predicado ‘repeat’ es el que hace que se siga iterando.

- En el siguiente acetato mostramos el código equivalente en Pascal, el cual, como se verá, es mucho más fácil de comprender que el programa en PROLOG que acabamos de ver.

Estilo Procedural

```
procedure SquaresTo(N: integer);
  var current: integer;
begin
   current := 1;
   repeat
      writeln(current*current);
      current := current+1;
    until current = N
end;
```

Negación

- ¿Cuál es el problema con la negación en PROLOG?

- Supongamos que queremos definir el predicado ‘can_marry(X,Y)’ (X se puede casar, legalmente, con Y) de la siguiente forma:

can_marry(X,Y) :- X \= Y, nonsibling(X,Y),noncousin(X,Y).

* El símbolo \= significa ‘diferente’. Supongamos que tenemos la base de datos que se muestra en el acetato siguiente.

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
sibling(X,Y) :- mother(M,X), mother(M,Y),
                father(F,X), father(F,Y), X \= Y.
cousin(X,Y) :- parent(U,X), parent(V,Y), sibling(U,V).

father(albert,jeffrey).
mother(alice,jeffrey).
father(albert, george).
mother(alice,george).
father(john, mary).
mother(sue, mary).
father(george, cindy).
mother(mary,cindy).
father(george,victor).
mother(mary, victor).
```

Negación

- Definamos ahora el predicado 'nonsibling(X,Y)':

```
nonsibling(X,Y) :- X = Y.
nonsibling(X,Y) :- mother(M1,X), mother(M2,Y), M1 \= M2.
nonsibling(X,Y) :- father(F1,X), father(F2,Y), F1 \= F2.
```

- Veamos ahora qué pasa si usamos:

```
:- nonsibling(albert,alice).
```

no

Negación

- De esta respuesta, se puede inferir que Albert y Alice son hermanos.

- Sin embargo, la base de datos no contiene información que nos permite aseverar eso. El problema es que, de hecho, la base de datos no indica quiénes son los padres de Albert y de Alice. Antes la premisa del mundo cerrado, PROLOG determina que lo que no existe en la base de datos no puede ser cierto y de ahí la respuesta que retorna.

Negación

- Una forma de lidiar con este problema es definir predicados que consideren la posibilidad de que una persona no tenga padres:

```
nonsibling(X,Y) :- no_parent(X).
nonsibling(X,Y) :- no_parent(Y).
```

- El problema ahora es cómo expresar que X no tiene padres. No existe un hecho positivo que exprese la ausencia de padre. Claro que podemos agregar el hecho 'no_parent(albert)', pero eso no resuelve el problema fundamental de la negación.

Negación

- El problema lo podemos resumir de la siguiente manera: Los hechos en una cláusula de Horn establecen una serie de relaciones que se establecen entre un cierto número de individuos.

- Las reglas permiten que se deduzcan un cierto número de hechos adicionales a partir de los hechos dados.

- Por lo tanto, a partir del hecho de que existe una cierta relación, sólo podemos concluir que existen otras relaciones, pero no podemos concluir que no existen.

Negación

- A partir de relaciones positivas, sólo podemos deducir relaciones positivas.
- Es por eso que tenemos problemas para definir 'nonsibling' (lo mismo pasaría con 'noncousin').
- La única forma de lidiar con este problema es poner los hechos de manera negativa (como con 'no_parent'), pero esto suele conducir a una explosión de conocimiento, pues se tienen que definir muchos hechos negativos para que el programa funcione.

Negación

- Para poder resolver este problema, debemos ir más allá de las cláusulas de Horn.
- Una posible solución es el uso de la combinación “cut-fail”.
- El predicado “fail”, como su nombre lo indica, siempre falla, y existe en la mayor parte de las versiones de PROLOG.

Negación

- Las reglas quedarían de la forma siguiente:

nonsibling(X,Y) :- sibling(X,Y), !, fail.
nonsibling(X,Y).

Si ahora consultamos:

:- nonsibling(albert,alice).
yes

Negación

- El sistema intentará satisfacer la primera cláusula. Esta cláusula requiere que se satisfaga ‘sibling(albert,alice)’.
- Como esta cláusula no se puede satisfacer, el sistema hace retroceso y se va a la siguiente regla.
- Esa regla devuelve cierto, con lo cual el sistema responde que la consulta es cierta.

Negación

- Si ahora usamos la consulta:

:- nonsibling(jeffrey,george).

no

- En este caso, se intenta satisfacer ‘sibling(jeffrey,george)’. Esto devuelve “cierto”, por lo que pasamos al “cut” e inmediatamente después al “fail”.

- Como el “cut” impide el retroceso, la cláusula falla.

Negación

- Este tipo de situación es tan común, que muchas versiones de PROLOG proporcionan un predicado “not” que tiene éxito si “C” falla y viceversa.

- Este predicado se puede definir de la forma siguiente:

```
not(C) :- call(C), !, fail.
not(C).
```

- call(C) intenta satisfacer la sub-meta representada por la estructura “C”.

Negación

- Ahora es muy fácil definir “nonsibling”:

```
nonsibling :- not(sibling(X,Y)).
```

- Esto significa que “nonsibling(X,Y)” se satisface si “sibling(X,Y)” no se satisface y viceversa. También podemos definir fácilmente “can_marry”:

```
can_marry(X,Y) :- X \= Y, not(sibling(X,Y)), not(cousin(X,Y)).
```

Negación

- Esta definición de “not” tiene, sin embargo, su lado oscuro. Consideremos la siguiente meta que enumera todas las formas posibles de hacer “append” a dos listas para producir “[a, b, c, d]”:

```
:- append(X,Y, [a,b,c,d]).
X = [], Y = [a,b,c,d];
X = [a], Y = [b,c,d];
X = [a,b], Y = [c,d];
X = [a,b,c], Y = [d];
X = [a,b,c,d], Y = [];
no
```

Negación

- Cada punto y coma hace que el sistema haga retroceso y encuentre instanciaciones adicionales de las variables.

- Consideremos ahora la meta siguiente:

:- not(not(append(X,Y, [a,b,c,d]))).

* En lógica, el usar “not(not)” se cancela, por lo que esperaríamos que esta consulta produjera el mismo resultado que la cláusula que vimos anteriormente.

Negación

- Sin embargo, este no es el caso. Veamos qué ocurre. El "not" más externo intenta satisfacer la sub-meta 'not(append(X,Y,[a,b,c,d])' que, a la vez, genera la sub-meta 'append(X,Y,[a,b,c,d])' la cual se puede satisfacer. Al satisfacerse, la combinación "cut-fail" hace que falle el "not" interno.

- Debido a la falla, debemos hacer un retroceso, por lo que cualquier asociación que se intente realizar para satisfacer la sub-meta no está definida. Como el "not" interno falla, el externo tiene éxito, pero X y Y no tienen valores asociados.

Negación

- Por tanto, lo que ocurre es lo siguiente:

:- not(not(append(X,Y,[a,b,c,d]))).

X = _0, Y = _1;

no

Los símbolos "_0" y "_1" los utiliza PROLOG para denotar que las variables X y Y no tienen valores asociados.

Negación

- ¿Por qué ocurre esto?
- El problema es que el “not” de PROLOG no es un “not” lógico, sino que significa “no satisfacible”.
- En lógica, existe una diferencia importante entre poder demostrar que algo es falso y no poder demostrar que algo es cierto.
- Este es un buen ejemplo de cómo PROLOG se aleja en varios aspectos de la programación lógica.

Comentarios Finales sobre PROLOG

- Uno de los beneficios de la programación lógica es que es de muy alto nivel y orientado a las aplicaciones.
- Puesto que los programas se describen en términos de predicados e individuos del dominio del problema, los programas casi están auto-documentados.
- Estas características promueven una programación clara, rápida y precisa.

Comentarios Finales sobre PROLOG

- Una de las principales críticas a PROLOG es que suele alejarse en diferentes aspectos de la programación lógica (la negación es un buen ejemplo de esto).
- PROLOG no es un lenguaje eficiente, y muchos del trabajo de investigación en torno a este lenguaje se ha concentrado, durante muchos años, en mejorar su eficiencia.
- Hoy en día, existen versiones muy eficientes de PROLOG.

Comentarios Finales sobre PROLOG

- Quizás la característica más importante de los programas lógicos puros es que constituyen ejemplos de programación no procedural.

- Durante muchos años, se tuvieron diversas discusiones sobre la programación no procedural, pero fueron los programas lógicos los primeros en demostrar su factibilidad.

- Claro que todavía hay mucho qué hacer para acercar al PROLOG a la programación lógica pura.

Principios de Diseño de Lenguajes

- Es natural preguntarse si existe un lenguaje de programación perfecto.
- Si existiera, la siguiente pregunta sería: ¿para qué perder nuestro tiempo con lenguajes imperfectos?
- A continuación, daré las razones por las cuales considero que no existe un lenguaje de programación perfecto.

Principios de Diseño de Lenguajes

- ¿Cómo sería un lenguaje de programación perfecto?
- Intuitivamente, podríamos decir que sería un lenguaje que resultara ideal para situaciones diferentes (para todos los usuarios, para todas las aplicaciones y para todas las plataformas de hardware).
- Este punto suena imposible de satisfacerse. Por ejemplo, para explotar mejor las características de una computadora en particular, es deseable desarrollar herramientas más específicas para la misma.

Principios de Diseño de Lenguajes

- ¿Qué nos lleva a esta diversidad de cosas?

- Podemos hacer las siguientes observaciones generales sobre las situaciones en las cuales ocurren los lenguajes de programación:

1) Hay muchas clases diferentes de **usos** de los lenguajes de programación.

2) Hay muchas clases diferentes de **usuarios** de los lenguajes de programación.

3) Hay muchas clases diferentes de **computadoras** en las cuales se pueden implementar los lenguajes de programación.

Principios de Diseño de Lenguajes

- Cada una de estas clases tiene características diferentes, que dictan diseños diferentes.

- Consideremos esta analogía con un avión: Un avión para entrenamiento debiera ser más tolerante a los errores del piloto que un avión normal, aún si esto hace que tenga un menor desempeño. Un avión de carga debe ser capaz de llevar mucho peso, aún si esto lo hace más lento. Un avión de combate debe tener una enorme maniobrabilidad y mayor velocidad, aún si su capacidad de carga es menor.

Principios de Diseño de Lenguajes

- Cada clase de usos, usuarios y computadoras, nos lleva a decisiones de diseño que suelen ser inapropiadas para otras clases.

- La diversidad de situaciones en las cuales se usan los lenguajes de programación es tan grande que cualquier lenguaje que intente acomodarlas todas, presentará un compromiso muy pobre entre ellas.

- Una mejor solución es identificar clases amplias de usos, usuarios y computadoras y crear lenguajes cuyas decisiones de diseño estén optimizadas para dichas clases.

Principios de Diseño de Lenguajes

- Nótese, sin embargo, que lo que sí es posible es determinar un entorno para un lenguaje perfecto.
- La motivación para dicho entorno es que, aunque los lenguajes difieren en sus detalles, suelen ser similares en su estructura general.
- Por ejemplo, un lenguaje científico podría tener los tipos real, integer y complex, muchas operaciones matemáticas y un manejo extenso de arreglos. Un lenguaje para manipulación de cadenas, tendrá los tipos character y string, records y operaciones para comparar cadenas.

Principios de Diseño de Lenguajes

- Independientemente de su uso, los lenguajes de programación comparten cierta funcionalidad (p.ej., iteradores definidos e indefinidos, selectores (if, case, etc.), declaraciones de funciones o procedimientos y bloques).

- Esta funcionalidad suele ser independiente de la computadora en la que se implemente el lenguaje, así como de los usos y usuarios del lenguaje.

- A esto se le denomina el **entorno del lenguaje** (*language framework*).

Principios de Diseño de Lenguajes

- Si bien la meta de llegar a tener un lenguaje de programación perfecto sea tal vez utópica, lo que sí podríamos tener son lenguajes muy flexibles y fáciles de portar a diferentes plataformas, capaces de incorporar diversas funciones y de poder tener una amplia aplicabilidad.

- Algunos tal vez piensen que lenguajes como Java proporcionan esta flexibilidad, como alguna vez muchos pensaron lo mismo de LISP.