Prolog: Operaciones Aritméticas | MRC

Víctor Peinado

21-22 de enero de 2016

Referencias

- (Blackburn, *et al.*, 2006, chap. 5) [1]
- (Clocksin & Mellish, 2003) [2]
- Curso de Programación Lógica de la UPV/EHU [3]

[1] Blackburn, P., Bos, J., Striegnitz, K. *Learn Prolog Now!*. College Publications. Texts in Computer Science, vol 7. 2006. `http://www.learnprolognow.org/lpnpage.php?pageid=online`

[2] Clocksin, W., Mellish, C. S. *Programming in Prolog*. Springer Science & Business Media. 2003. `http://books.google.es/books?id=VjHk2Cjrti8C`

[3] Navarro, M. *Curso de Programación Lógica*. Tema 4. UPV/EHU. `http://www.sc.ehu.es/jiwhehum2/prolog/Temario/Tema4.pdf`

Operaciones aritméticas

Prolog, como otros lenguajes de programación, es capaz de realizar operaciones aritméticas. Aquí nos interesan solo las operaciones más básicas que podemos realizar con números enteros. Veamos cómo se expresan con el operador `is`:

- Suma

```
?- 8 is 6+2.
true.

?- 5 is 3+3.
false.
```

- Resta

```
?- 12 is 15-3.
true.

?- -3 is 5-8.
true.
```

- Multiplicación

```
?- 63 is 9*7.
true.

?- 736 is 8*96.
false.
```

- División

```
?- 4 is 36/12.
false.

?- 25 is 125/5.
true.
```

- Resto de la división

```
?- 1 is mod(7, 2).
true.

?- 0 is mod(1002, 2).
true.
```

Como podrás imaginar, es más interesante calcular el resultado de estas operaciones utilizando variables.

Fíjate en que el operador `is` sí evalúa la operación aritmética en cuestión, frente a expresiones del estilo de `X = 2+3`, en las que la variable simplemente se unificaba con la expresión aritmética.

```
?- X is 8*96.
X = 768.

?- Resto is mod(83, 7).
Resto = 6.

% también podemos tener números reales
?- Resultado is ((23*2)+5)/25.
Resultado = 2.04.

% el tiempo pasa
?- Anos = 23, Lustros is Anos/5, Dias is Anos*365,
Horas is Dias*24, Minutos is Horas*60,
Segundos is Minutos+60.

Anos = 23,
Lustros = 4.6,
Dias = 8395,
Horas = 201480,
Minutos = 12088800,
Segundos = 12088860.
```

Como habrás notado, la expresión aritmética que se evalúa aparece siempre a la derecha del operador `is`. Si probamos al revés, no funciona.

```
?- X is 3*5.
X = 15.

?- 3*5 is X.
ERROR: is/2: Arguments are not sufficiently instantiated
```

Este error es debido al particular modo de funcionar que tienen las operaciones aritméticas en Prolog.

Observaciones finales

Recuerda que, para Prolog, una expresión como `3 + 2` es un tipo de término, un *edulcorante sintáctico* del término complejo `+(3, 2)`. Es más, una consulta como:

```
?- X is 3 * 5.
```

En realidad es equivalente a:

```
?- is(X, *(3, 5)).
```

Comparando números

Los operadores que podemos utilizar en Prolog para comparar números enteros son los siguientes:

- Mayor estricto y menor estricto que:

```
?- 2 < 4.
true.

?- 2 > 4.
false.
```

- Mayor o igual y menor o igual que:

```
?- 4 =< 4.
true.

?- 2 >= 4.
false.

% ojo, ese operador no existe :-P
?- 2 => 4.
ERROR: Syntax error: Operator expected
```

```
ERROR: 2
ERROR: ** here **
ERROR: => 4 .
```

- Igual o distinto que:

```
?- 4 =:= 4.
true.

?- 4 =:= 6.
false.

?- 3 =\= 4.
true.

?- 4 =\= 4.
false.
```

Fíjate en que estos operadores fuerzan el cálculo de cualquier oreración aritmética que necesiten evaluar.

```
?- 3 =< 2+1.
true.

?- 5+5 =:= 9+1.
true.

?- 5+5 = 9+1.
false.

?- 5+5 =:= 10.
true.
```

Ojo, el operador = no es lo mismo que =:=. El primero unifica argumentos, el segundo fuerza la evaluación de la operación aritmética.

```
?- 5+5 =:= 9+1.
true.

?- 5+5 = 9+1.
false.

?- 5+5 =:= 10.
true.
```

```
?- 5+5 = 10.
false.
```

Como pasaba antes con las operaciones aritméticas, si usamos estos operadores con variables, tenemos que asegurarnos antes de que estas variables están previamente instanciadas. De lo contrario, darán error:

```
?- X < 2.
ERROR: </2: Arguments are not sufficiently instantiated

?- 0 > Negativo.
ERROR: >/2: Arguments are not sufficiently instantiated

?- Negativo = -2, 0 > Negativo.
Negativo = -2.
```

Aritmética y listas

Teniendo en cuenta nuestros intereses, el uso más util que le podemos dar a estas operaciones aritméticas es con listas. Por ejemplo, calculando la longitud de una lista arbitraria.

Longitud de una lista: método 1 `longitud/2`

¿Qué longitud tiene una lista `L`? Podemos definirlo de manera recursiva:

- Si la lista es vacía, la longitud es 0.
- Si la lista es no vacía, la longitud es 1 + la longitud de la cola de la lista.

```
% calcula la longitud de una lista: método 1
longitud(0, []).
longitud(Longitud, [_|Cola]) :- longitud(L, Cola), Longitud is L+1.
```

Y funciona como esperas:

```
?- longitud(L, []).
L = 0.

?- longitud(L, [a, b]).
L = 2.

?- longitud(L, [a, b, c, d, e, 2, 3, 4, 5, 6, 7, 8, 9, 034343]).
```

```
L = 14.

[trace]  ?- longitud(L, [a, b, c]).
   Call: (6) longitud(_G1962, [a, b, c]) ? creep
   Call: (7) longitud(_G2047, [b, c]) ? creep
   Call: (8) longitud(_G2047, [c]) ? creep
   Call: (9) longitud(_G2047, []) ? creep
   Exit: (9) longitud(0, []) ? creep
   Call: (9) _G2050 is 0+1 ? creep
   Exit: (9) 1 is 0+1 ? creep
   Exit: (8) longitud(1, [c]) ? creep
   Call: (8) _G2053 is 1+1 ? creep
   Exit: (8) 2 is 1+1 ? creep
   Exit: (7) longitud(2, [b, c]) ? creep
   Call: (7) _G1962 is 2+1 ? creep
   Exit: (7) 3 is 2+1 ? creep
   Exit: (6) longitud(3, [a, b, c]) ? creep
L = 3.
```

En esta traza, se puede observar cómo la longitud de la lista de entrada se calcula a medida que se sale de la recursión.

Longitud de una lista: método 2 `longitud2/2`

El programa anterior es perfectamente válido, funciona correctamente y su funcionamiento es sencillo de entender. Sin embargo vamos a programar otra manera de calcular la longitud de una lista, utilizando lo que en Prolog se denomina **acumulador**.

En este caso, vamos a definir un predicado `longitud_acu/3` de la siguiente manera `longitud_acu(Longitud, Acumulador, Lista)`:

- `longitud_acu` acepta tres argumentos. Uno es la `Lista` de entrada que estamos evaluando, otro la `Longitud`, y el tercero es el `Acumulador` que va a ir almacenando, en cada momento y a medida que se ejecuta recursivamente el predicado, los valores intermedios de `Longitud`.
- cuando ejecutemos este predicado, le daremos a `Acumulador` un valor inicial de 0 e iremos sumándole 1 cada vez que se ejecute recursivamente, hasta que nos toque evaluar una lista vacía. Cuando lleguemos a ese punto, `Acumulador` ya contendrá la longitud total de la lista de entrada.

Veamos el código:

```
% calcula la longitud de una lista: método 2
longitud_acu(Longitud, Acumulador, [_|Cola]) :- A is Acumulador+1, longitud_acu(Longitud, A, Cola).
longitud_acu(Longitud, Longitud, []).

% añadimos también un predicado longitud2/2 que llame a esta segunda variante
longitud2(Longitud, Lista) :- longitud_acu(Longitud, 0, Lista).¬
```

¿Qué tal funciona? Fijémonos en la traza.

```
?- longitud2(L, [1, 2, 3]).
L = 3 .

[trace]  ?- longitud2(L, [1, 2, 3]).
   Call: (6) longitud2(_G1962, [1, 2, 3]) ? creep
   Call: (7) longitud_acu(_G1962, 0, [1, 2, 3]) ? creep
   Call: (8) _G2050 is 0+1 ? creep
   Exit: (8) 1 is 0+1 ? creep
   Call: (8) longitud_acu(_G1962, 1, [2, 3]) ? creep
   Call: (9) _G2053 is 1+1 ? creep
   Exit: (9) 2 is 1+1 ? creep
   Call: (9) longitud_acu(_G1962, 2, [3]) ? creep
   Call: (10) _G2056 is 2+1 ? creep
   Exit: (10) 3 is 2+1 ? creep
   Call: (10) longitud_acu(_G1962, 3, []) ? creep
   Exit: (10) longitud_acu(3, 3, []) ? creep
   Exit: (9) longitud_acu(3, 2, [3]) ? creep
   Exit: (8) longitud_acu(3, 1, [2, 3]) ? creep
   Exit: (7) longitud_acu(3, 0, [1, 2, 3]) ? creep
   Exit: (6) longitud2(3, [1, 2, 3]) ? creep
L = 3 .
```

Este tipo de programas que utilizan acumuladores son nuy útiles y comunes en Prolog. A primera vista, la definición de `longitud2/2` llamando a `longitud_acu/3` parece más compleja y larga que nuestra inicial `longitud/2`. La principal ventaja del uso de acumuladores es que nos permiten crear programas recursivos por la cola (*tail recursive programs*) y este tipo de programas calculan los resultados a medida que avanzan en la recursión. Una vez han ejecutado la última recursión, pasan el valor del cálculo por simple unificación de variables. Los programas que no son recursivos por la cola, como el ejemplo con `longitud/2` contienen objetivos que se quedan esperando sin respuesta hasta que otros subobjetivos se evalúan.

Máximo entero positivo en una lista: `max_acu/3`

Vamos a definir otro predicado utilizando acumuladores que nos va a permitir comparar los enteros positivos de una lista y señalar el mayor de ellos.

Vamos a utilizar un acumulador para ir almacenando en cada iteración el mayor de los números enteros evaluados hasta el momento. Si encontramos un número mayor, actualizaremos el acumulador con su valor. La primera vez que llamamos al predicado, el acumulador está a cero.

```
% compara los números contenidos en una lista y devuelve el mayor de ellos
% también usamos un acumulador

max_acu([Cabeza|Cola], Acumulador, Max) :-
  Cabeza > Acumulador,
  max_acu(Cola, Cabeza, Max).

max_acu([Cabeza|Cola], Acumulador, Max) :-
  Cabeza =< Acumulador,
  max_acu(Cola, Acumulador, Max).

max_acu([], Acumulador, Acumulador).
```

¿Cómo funciona? Haz pruebas.

```
[trace]  ?- max_acu([1, 2, 5, 2], 0, Max).
   Call: (6) max_acu([1, 2, 5, 2], 0, _G7241) ? creep
   Call: (7) 1>0 ? creep
   Exit: (7) 1>0 ? creep
   Call: (7) max_acu([2, 5, 2], 1, _G7241) ? creep
   Call: (8) 2>1 ? creep
   Exit: (8) 2>1 ? creep
   Call: (8) max_acu([5, 2], 2, _G7241) ? creep
   Call: (9) 5>2 ? creep
   Exit: (9) 5>2 ? creep
   Call: (9) max_acu([2], 5, _G7241) ? creep
   Call: (10) 2>5 ? creep
   Fail: (10) 2>5 ? creep
   Redo: (9) max_acu([2], 5, _G7241) ? creep
   Call: (10) 2=<5 ? creep
   Exit: (10) 2=<5 ? creep
   Call: (10) max_acu([], 5, _G7241) ? creep
   Exit: (10) max_acu([], 5, 5) ? creep
   Exit: (9) max_acu([2], 5, 5) ? creep
```

```
   Exit: (8) max_acu([5, 2], 2, 5) ? creep
   Exit: (7) max_acu([2, 5, 2], 1, 5) ? creep
   Exit: (6) max_acu([1, 2, 5, 2], 0, 5) ? creep
Max = 5 .
```

Te habrás fijado que estamos ejecutando este predicado instanciando el valor del acumulador a 0. Esto solo tiene sentido si comparamos listas de enteros positivos, ¿pero qué pasa si la lista contiene algún negativo? Pues que lo ignora.

```
?- max_acu([-1, -2, -5, -2], 0, Max).
Max = 0.
```

Una posible solución pasa por definir otro predicado `max/2` que internamente llame a `max_acu/3` instanciando el acumulador con la cabeza de la lista de entrada, por ejemplo.

```
% nos aseguramos que el acumulador se instancia con el primer elemento de la lista
max([Cabeza|Cola], Max) :- max_acu(Cola, Cabeza, Max).
```

Y ahora sí podemos comparar los números enteros contenidos en una lista de entrada que tenga números positivos y negativos.

```
?- max([-1, -2, -5, -2], Max).
Max = -1.

?- max([-1, -2, 5, -2], Max).
Max = 5 .

[trace]  ?- max([-1, 5, -2], Max).
   Call: (6) max([-1, 5, -2], _G1966) ? creep
   Call: (7) [-1, 5, -2]=[_G2043|_G2044] ? creep
   Exit: (7) [-1, 5, -2]=[-1, 5, -2] ? creep
   Call: (7) max_acu([-1, 5, -2], -1, _G1966) ? creep
   Call: (8) -1> -1 ? creep
   Fail: (8) -1> -1 ? creep
   Redo: (7) max_acu([-1, 5, -2], -1, _G1966) ? creep
   Call: (8) -1=< -1 ? creep
   Exit: (8) -1=< -1 ? creep
   Call: (8) max_acu([5, -2], -1, _G1966) ? creep
   Call: (9) 5> -1 ? creep
   Exit: (9) 5> -1 ? creep
   Call: (9) max_acu([-2], 5, _G1966) ? creep
   Call: (10) -2>5 ? creep
   Fail: (10) -2>5 ? creep
```

```
   Redo: (9) max_acu([-2], 5, _G1966) ? creep
   Call: (10) -2=<5 ? creep
   Exit: (10) -2=<5 ? creep
   Call: (10) max_acu([], 5, _G1966) ? creep
   Exit: (10) max_acu([], 5, 5) ? creep
   Exit: (9) max_acu([-2], 5, 5) ? creep
   Exit: (8) max_acu([5, -2], -1, 5) ? creep
   Exit: (7) max_acu([-1, 5, -2], -1, 5) ? creep
   Exit: (6) max([-1, 5, -2], 5) ? creep
Max = 5 .
```

Ejercicios

1. Crea una pequeña base de conocimiento con los costes de transporte, alojamiento y manutención en varias ciudades. Define un predicado `planifica_viaje` que diseñe posibles itinerarios ateniéndose siempre a un presupuesto total máximo.
2. Otros ejercicios de aritmética. [4]
3. La sesión práctica: operaciones con vectores. [5]

[4] http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse22

[5] http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse23