

Estructuras de Datos en Prolog

Lógica para Ciencias de la Computación

Primer Cuatrimestre de 2008

– Material Adicional –

Tipos de Datos en Prolog

- En Prolog no existe el concepto de **tipo de dato**.
- Las variables denotan un cierto objeto, pero ese objeto puede ser **cualquiera** de los objetos considerados.
- La carencia de **tipos de datos** implica una mayor responsabilidad para el programador.

Estructuras de Datos en Prolog

- No disponer de tipos de datos no implica dejar de lado a las **estructuras de datos**.
- Por otro lado, la **recursión** es la principal estructura de control en Prolog.
- Las **estructuras de datos recursivas** se integran naturalmente a los programas escritos en Prolog.

Estructuras de Datos Recursivas

- Una estructura de datos se puede decir **recursiva** si cada una de sus distintas instancias satisfacen:
 - ▶ Ser una **instancia básica**, la cual puede ser definida de forma directa.
 - ▶ Ser una **instancia general**, la cual se define como una elaboración de otra u otras instancias más simples.

Codificación Recursiva para los Números Naturales

- La codificación estándar parece **no satisfacer** las condiciones anteriores:
 - ▶ 0, 1, 2, 3, 4, ...
- La **notación unaria** sin duda es recursiva:
 - ▶ 1, 11, 111, 1111, 11111, ...
- La **notación $s^n(0)$** también es recursiva:
 - ▶ 0, s(0), s(s(0)), s(s(s(0))), s(s(s(s(0)))) , ...

Ejemplo

Dado un número natural obtener el siguiente:

$\text{next}(X, s(X)).$

?- $\text{next}(0, 0).$

no

?- $\text{next}(0, Rsta).$

$Rsta = s(0)$

Ejemplo

Verificar si el argumento suministrado es un número natural en codificación $s^n(0)$:

`nn(0).`

`nn(s(X)) :- nn(X).`

`?- nn(s(s(0))).`

`yes`

`?- nn(Rsta).`

`...`

Estrategia de Resolución

- Las soluciones recursivas a problemas basados en estructuras de datos recursivas son **elegantes** y **concisas**, pero encontrarlas es una tarea **no necesariamente simple**.
- Estas soluciones pueden construirse de manera **directa** una vez encontrado un **planteo recursivo del problema**.

¡Los matemáticos hacen esto hace milenios!

Planteo Recursivo del Problema

• Consta de dos partes o casos:

➤ **Caso Base:** se define en forma directa la solución a una **instancia base** o trivial del problema.

➤ **Caso Recursivo:** se define la solución a una **instancia gral.** del problema **en términos de** la solución a una **instancia menor** (ie, más cercana a la instancia base).

Ejemplo

• Obtener la suma de dos número naturales en notación $s^n(0)$. *Pista: el pred. y suc. de un número en notación $s^n(0)$ se obtienen fácilmente.*

$$a + b = \begin{cases} b, \text{ cuando } a \text{ es } 0. & \text{(CB)} \\ \text{suc}(\text{pred}(a) + b), \text{ caso contrario.} & \text{(CR)} \end{cases}$$

suma(A, B, Rdo) :- A = 0, Rdo = B.

suma(A, B, Rdo) :- A = s(PA),
suma(PA, B, SPAB),
Rdo = s(SPAB).

Crítica a la Solución Anterior

$\text{suma}(A, B, \text{Rdo}) :- A = 0, \text{Rdo} = B.$

$\text{suma}(A, B, \text{Rdo}) :-$

$A = s(\text{PA}), \text{suma}(\text{PA}, B, \text{SPAB}), \text{Rdo} = s(\text{SPAB}).$

Como las variables denotan **incógnitas**, el hecho de que esas incógnitas sean luego **develadas** hace que las variables en cuestión resulten **redundantes**.

$\text{suma}(0, B, B).$

$\text{suma}(s(\text{PA}), B, s(\text{PAB})) :- \text{suma}(\text{PA}, B, \text{PAB}).$

Declarativo vs. No Declarativo

- Este ejemplo ilustra una característica interesante de Prolog, ya que se trata de una **solución declarativa**.
- El adjetivo declarativo denota que el **significado** del programa en cuestión **es evidente**.
- Prolog **fomenta** la elaboración de soluciones declarativas.

Inversión de Predicados

El predicado `suma/3` introducido antes presenta un comportamiento muy interesante:

?- `suma(s(0), s(0), Rsta).`

`Rsta = s(s(0))`

?- `suma(A, B, s(0)).`

`A = 0, B = s(0);`

`A = s(0), B = 0`

?- `suma(Rsta, s(0), s(s(0))).`

...

Cuidado

¿Qué pasa al invertir un predicado que haga uso del predicado predefinido `is/2`?

```
suma(X, Y, Z) :- Z is X + Y.
```

```
?- suma(Rsta, 1, 2).
```

...

Listas en Prolog

- La lista es una estructura de datos **recursiva**:
 - ➔ La **lista vacía** es una lista.
 - ➔ Agregarle un **elemento** a una **lista** genera otra lista.
- Codificación arbitraria (no de Prolog):
 - ➔ La constante **vacía** denota a la lista vacía.
 - ➔ La estructura **poner/2** denota poner el primer argumento a la cabeza de la lista suministrada como segundo argumento.

Codificación Arbitraria

- La lista vacía:
 - ➔ `vacía`
- La lista conteniendo al número 1:
 - ➔ `poner(1, vacía)`
- La lista conteniendo a 1, 2, 3:
 - ➔ `poner(1, poner(2, poner(3, vacía)))`
- La lista conteniendo a la lista con el 1:
 - ➔ `poner(poner(1, vacía), vacía)`

Ejemplo

Obtener la longitud (en notación $s^n(0)$) de una lista codificada de acuerdo al esquema anterior:

`largo(vacia, 0).`

`largo(poner(Elto, SubLista), s(LargoSL)) :-
largo(SubLista, LargoSL).`

`?- largo(poner(1, poner(2, poner(3, vacia))), Rsta).`

`Rsta = s(s(s(0)))`

Codificación Estándar

- La lista es una estructura fundamental para la programación en lógica.
- Por esta razón, al diseñar la sintáxis de PROLOG se incluyó una codificación especial para listas que resulta más cómoda para el usuario, (sin tantos “poner”).
- Codificación propia de Prolog:
 - ▶ La constante `[]` denota a la lista vacía.
 - ▶ La estructura `[H | T]` denota agregar al elemento `H` al comienzo de la lista `T`.

Codificación Estándar

- La lista conteniendo a 1, 2, 3:

➔ [1 | [2 | [3 | []]]]

- Comparandola con nuestra notación “casera”:

➔ poner(1 , poner(2, poner(3 , vacia)))

➔ [1 | [2 | [3 | []]]]

- Para mayor comodidad aún, PROLOG permite expresarla directamente así:

➔ [1, 2, 3]

... no obstante, el intérprete transforma automáticamente esta representación a la representación recursiva.

Listas de listas

- Los elementos de una lista son términos, en particular pueden ser listas!

- La lista conteniendo a la lista que contiene al 1:

 - $[[1 \mid []] \mid []]$ *(rep. recursiva)*

 - $[[1]]$

- La lista conteniendo 2 veces a la lista que contiene al 1:

 - $[[1 \mid []] \mid [[1 \mid []] \mid []]]$ *(rep. recursiva)*

 - $[[1], [1]]$

Ejemplo

Obtener la longitud de una lista codificada en notación estándar de Prolog:

```
largo([], 0).
```

```
largo([Head | Tail], s(LargoT)) :-
```

```
    largo(Tail, LargoT).
```

```
?- largo([1, 2, 3], Rsta).
```

```
Rsta = s(s(s(0)))
```

```
?- largo(Lista, s(s(0))).
```

...

Concatenación

Dadas dos listas, obtener la lista que resulta de concatenar ambas listas:

`concat([], Xs, Xs).`

`concat([X | Xs], Ys, [X | Zs]) :- concat(Xs, Ys, Zs).`

?- `concat([1, 2], [3, 4], Rsta).`

`Rsta = [1, 2, 3, 4]`

?- `concat(Xs, Ys, [1, 2, 3, 4]).`

...