Listas

Objetivos

- Las listas son una estructura de datos recursiva fundamental en Prolog
- member/2 predicado Prolog fundamental para la definición de listas
- Metodología para la construcción de predicados sobre listas

Listas

Una lista es una secuencia finita de elementos

Ejemplos:

[mia, vicente, julio, yolanda]

[mia,>, X, 2, mia]

[ ]

[[ ], a(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

Listas

- Los elementos de una lista se encierran entre corchetes.
- Cualquier término Prolog puede ser un elemento de una lista.
- Una lista puede tener elementos de distinto tipo mezclados.
- Existe una lista especial:

  la lista vacía : [ ]

Cabeza y Resto

- Una lista no vacía es un término complejo con 2 argumentos:
  - Cabeza de la lista (1er elemento)
  - Resto: todo lo demás
    - es siempre una lista,
    - eventualmente la lista vacía

Cabeza y Resto

[mia, vicente, julio, yolanda]

C:

R:

Cabeza y Resto

[mia, vicente, julio, yolanda]

C: mia
R:

Cabeza y Resto

[mia, vincent, jules, yolanda]

C: mia
R: [vicente, julio, yolanda]

Cabeza y Resto

[[ ], a(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

C:

R:

Cabeza y Resto

[[ ], a(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

C: [ ]
R:

Cabeza y Resto

[[ ], a(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

C: [ ]
R: [a(z), [2, [b,c]], [ ], Z, [2, [b,c]]]

Cabeza y Resto

[a(z)]

C:

R:

Cabeza y Resto

[a(z)]

C: a(z)
R:

Cabeza y Resto

[a(z)]

C: a(z)
R: [ ]

La lista vacía

- La lista vacía no tiene ni cabeza ni resto
- [ ] es un átomo, no tiene estructura interna

- La Lista vacía será el caso base en las definiciones recursivas de predicados Prolog

El operador |

- Prolog tiene un operador especial ‘|’ que puede usarse para descomponer una lista en Cabeza y Resto

- El operador | es una herramienta clave en la construcción de predicados que manipulan listas

El operador |

```
?- [C|R] = [mia, vicente, julio, yolanda].

C = mia
R = [vicente,julio,yolanda]
yes

?-
```

El operador |

```
?- [X|Y] = [mia, vicente, julio, yolanda].

X = mia
Y = [vicente,julio,yolanda]
yes

?-
```

El operador |

```
?- [X|Y] = [ ].

no

?-
```

El operador |

```
?- [X,Y|R] = [[ ], a(z), [2, [b,c]], [], Z, [2, [b,c]]] .

X = [ ]
Y = a(z)
R = [[2, [b,c]], [ ], Z, [2, [b,c]]]
yes

?-
```

Variable anónima

Supongamos que estamos interesados en el 2do y 4to elemento de una lista

```
?- [X1,X2,X3,X4|R] = [mia, vicente, ana, laura, yolanda].
X1 = mia
X2 = vicente
X3 = ana
X4 = laura
R = [yolanda]
yes

?-
```

Variable anónima

There is a simpler way of obtaining only the information we want:

```
?- [ _,X2, _,X4|_ ] = [mia, vicente, ana, laura, yolanda].
X2 = vicente
X4 = laura
yes

?-
```

‘ _ ’ : variable anónima

Variable anónima

- Necesitamos una variable, no nos interesa en qué valor se instancie.

- Cada ocurrencia de la variable anónima es independiente : distintas ocurrencias pueden instanciarse en distintos valores

Member

- Predicado que dado un elemento X y una lista L nos dice si X es o no un elemento de L
- Se llama usualmente *member*
- Suele estar definido en los intérpretes

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?-
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(yolanda,[yolanda,juan,vicente,julio]).
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(yolanda,[yolanda,ana,1,2]).
yes
?-
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(vicente,[yolanda,ana,vicente,julio]).
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(vicente,[yolanda,ana,vicente,julio]).
yes
?-
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(luis,[yolanda,ana,vicente,julio]).
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(luis,[yolanda,ana,vicente,julio]).
no
?-
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(X,[a,b,c]).
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(X,[a,b,c]).
X = a ;
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(X,[a,b,c]).
X = a ;
X = b ;
```

member/2

```
member(X,[X|T]).
member(X,[H|T]):- member(X,T).
```

```
?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c.
```

member/2 con variables anónimas

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

Recorriendo recursivamente listas

- El predicado member/2 recorre recursivamente la lista
  - Procesando la cabeza
  - Procesando recursivamente el resto
- Esta técnica es muy utilizada

ejemplo: a2b/2

El predicado a2b/2, con 2 listas como argumentos, se satisface si

- El 1er argumento es una lista de a's y
- El 2do argumento es una lista de b's del mismo largo

```
?- a2b([a,a,a,a],[b,b,b,b]).
yes
?- a2b([a,a,a,a],[b,b,b]).
no
?- a2b([a,c,a,a],[b,b,b,t]).
no
```

Definiendo a2b/2: paso 1

```
a2b([],[]).
```

En primer lugar, la situación más simple posible

En este caso: la lista vacía

Definiendo a2b/2: paso 2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

- El caso general
- Especificamos la propiedad de a2b dando condiciones para las Cabezas y los Restos de ambas listas

Testeando a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b([a,a,a],[b,b,b]).
yes
?-
```

Testeando a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b([a,a,a,a],[b,b,b]).
no
?-
```

Testeando a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b([a,t,a,a],[b,b,b,c]).
no
?-
```

Testeando a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b([a,a,a,a,a], X).
X = [b,b,b,b,b]
yes
?-
```

Testeando a2b/2

```
a2b([],[]).
a2b([a|L1],[b|L2]):- a2b(L1,L2).
```

```
?- a2b(X,[b,b,b,b,b,b,b]).
X = [a,a,a,a,a,a,a]
yes
?-
```

Resumen

- Listas
  - Sintaxis
  - Lista vacía
  - Operador |
- Predicados recursivos sobre listas
  - member/2
  - a2b/2
- Variable anónima

Próxima

Aritmética *extra-lógica* en Prolog

- Predefinidos Prolog para aritmética (no es práctico trabajar en unario!!)
- Aritmética y Listas
- Introducción a acumuladores