Programación para la Inteligencia Artificial

Prolog

Dr. Alejandro Guerra-Hernández

Instituto de Investigaciones en Inteligencia Artificial
Universidad Veracruzana
Campus Sur, Calle Paseo Lote II, Sección Segunda No 112,
Nuevo Xalapa, Xalapa, Ver., México 91097

mailto:aguerra@uv.mx
https://www.uv.mx/personal/aguerra/pia

Maestría en Inteligencia Artificial 2023

Universidad Veracruzana

Instalación

Prolog: `https://www.swi-prolog.org`

Emacs: `https://www.gnu.org/software/emacs/`

Modo: `https://bruda.ca/emacs/prolog_mode_for_emacs`

Universidad Veracruzana

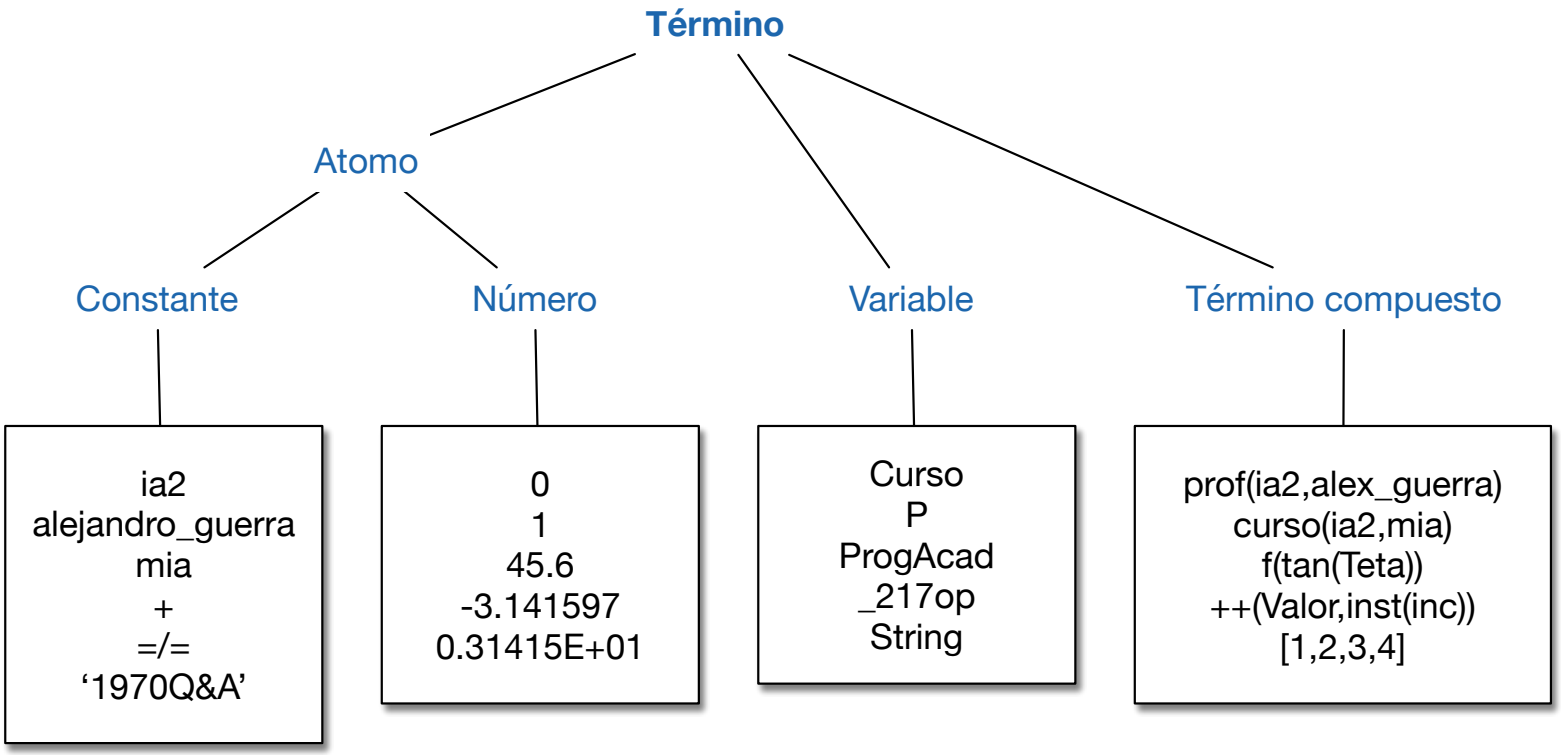
Bibliografía básica

- Bratko [1] está orientado a Inteligencia Artificial.
- Sterling y Shapiro [3] es una buena introducción genérica.
- The Power of Prolog, por Markus Triska:

  `https://www.metalevel.at/prolog?ref=hackr.io`
- Clocksin y Melish [2] presenta el ISO-estándar.

Universidad Veracruzana

¿De quién podemos hablar? Términos



Universidad Veracruzana

¿Qué podemos decir? Hechos

- Hechos, relaciones incondicionales entre términos.
- Ej. Sobre ciudades y sus localizaciones:

```
1  % loc_en(X,Y): La ciudad X está localizada en Y.
2
3  loc_en(atlanta,georgia).
4  loc_en(houston,texas).
5  loc_en(austin,texas).
6  loc_en(boston,massachussets).
7  loc_en(xalapa,veracruz).
8  loc_en(veracruz,veracruz).
```

Universidad Veracruzana

¿Qué podemos computar? Metas

- `true`. Se puede deducir y la meta no tiene variables:

```
?- loc_en(xalapa,veracruz).
true.
```

- `false`. No se puede deducir. Se asume el CWA:

```
?- loc_en(xalapa,texas).
false.
```

- Substitución de respuesta. Se puede deducir y la meta tiene variables:

```
?- loc_en(Ciudad,texas).
Ciudad = houston ;
Ciudad = austin ;
false.
```

Universidad Veracruzana

¿Qué más se puede decir? Reglas

- Relaciones condicionadas entre términos.
- Extienden la semántica de localizado en:

```
1  % loc_en(X,Y): La ciudad X está localizada en el país Y
2
3  loc_en(X,usa) :- loc_en(X,georgia).
4  loc_en(X,usa) :- loc_en(X,texas).
5  loc_en(X,usa) :- loc_en(X,massachussets).
6  loc_en(X,mex) :- loc_en(X,veracruz).
7
8  % loc_en(X,norteamerica): La ciudad X esta en norteamérica
9
10 loc_en(X,norteamerica) :- loc_en(X,usa).
11 loc_en(X,norteamerica) :- loc_en(X,mexico).
```

Universidad Veracruzana

Más sobre las metas

- El mismo predicado `loc_en/2` nos sirve para computar diferentes consultas, dependiendo de donde aparecen las variables.

```
?- loc_en(xalapa,usa).
false.
?- loc_en(xalapa,norteamerica).
true
?- loc_en(C,mexico).
C = xalapa ;
C = veracruz ;
false.
?- loc_en(xalapa,Loc).
Loc = veracruz ;
Loc = mexico ;
Loc = norteamerica ;
false.
```

Universidad Veracruzana

Metas negativas

- Se puede usar el predicado not (\+), con algunas sorpresas.

```
?- \+ loc_en(xalapa,usa).
true.
?- \+ loc_en(C,usa).
false.
```

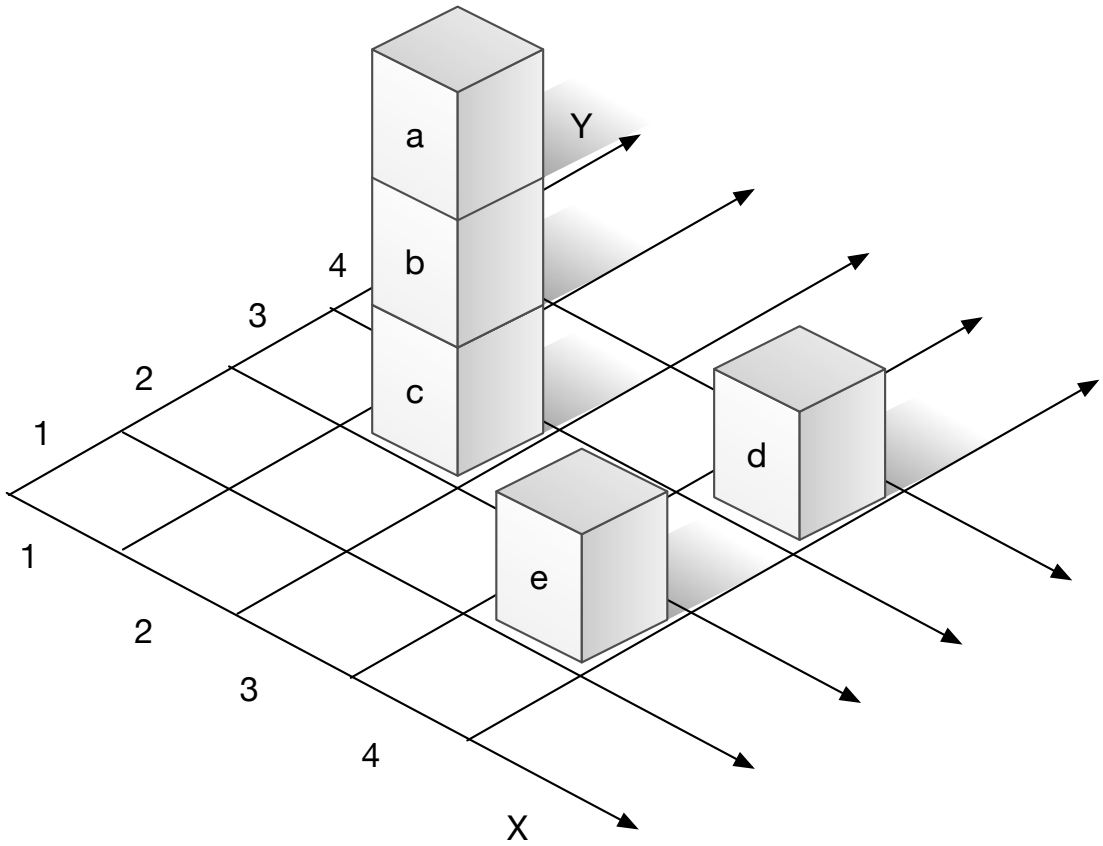
- Pero:

```
?- loc_en(C,norteamerica), \+ loc_en(C,usa).
C = xalapa ;
C = veracruz ;
false.
```

Universidad Veracruzana

El mundo de los bloques

- Asumamos un medio ambiente como el siguiente:



Universidad Veracruzana

Base de conocimientos

- Se puede representar el mundo como:

```
1  %% see(Block,X,Y)
2  % Block is observed by the camera at coordinates X, Y
3  % From Bratko, 4th edition.
4
5  see(a,2,3).
6  see(d,4,4).
7  see(e,4,2).
8
9  %% on(Block,Object)
10 %  Block is standing on Object
11
12 on(a,b).
13 on(b,c).
14 on(c,table).
15 on(d,table).
16 on(e,table).
```

Universidad Veracruzana

Algunas metas sobre bloques

- ¿Qué bloques hay en el mundo?

```
?- on(Block,_).
Block = a ;
Block = b ;
Block = c ;
Block = d ;
Block = e.
```

- ¿Qué bloques están en la misma coordenada X?

```
?- see(B1,X,_),see(B2,X,_), dif(B1,B2).
B1 = d,
X = 4,
B2 = e
```

Universidad Veracruzana

Otras metas sobre bloques

- ¿Qué bloques no son visibles?

```
?- on(B,_), on(_,B).
B = b ;
B = c ;
false.
```

- O de manera alternativa (cuidado con la negación):

```
?- on(B,_), \+ see(B,_,_).
B = b ;
B = c ;
false.
```

- ¿Cual es el bloque visible más a la izquierda?

```
?- see(B,X,_), \+ (see(B2,X2,_), X2 < X).
B = a,
X = 2 ;
false.
```

Universidad Veracruzana

¿Y la coordenada Z?

- Definición recursiva:

```
26  % z(B,Z) returns the Z coordinate of block B
27  z(B,1) :-
28      on(B,table).
29  z(B,Z) :-
30      on(B,B1),
31      z(B1,Z1),
32      Z is Z1 + 1.
33
34  % coords(B,X,Y,Z) the coordinates in 3D of block B
35
36  coords(B,X,Y,Z) :-
37      see(B,X,Y),
38      z(B,Z).
```

Universidad Veracruzana

Más metas

- Sobre las coordenadas Z:

```
1  ?- z(a,Z).
2  Z = 3
3  ?- coords(a,X,Y,Z).
4  X = 2,
5  Y = Z, Z = 3
```

Universidad Veracruzana

Definición de lista

- Las listas son términos compuestos.
- Una lista se define de manera recursiva:
  - Una lista vacía se denotada como [].
  - Una lista no vacía se denota como $[X|Xs]$, donde X se conoce como cabeza de la lista y Xs como cola de la lista, que es a su vez una lista.
- Lo anterior puede verificarse vía unificación:

```
1 ?- [1,2,3] = [1 | [2 | [3 | []]]].
2 true.
```

- Nota: SWI-Prolog no usa el punto como functor para denotar listas:

```
1 ?- X = '[|]'(1, []).
2 X = [1].
```

Universidad Veracruzana

Acceso a los elementos de una lista

- Se explota su estructura recursiva y la unificación: Toda lista tiene una cabeza y una cola:

```
1  ?- [X|Xs] = [1,2,3].
2  X = 1
3  Xs = [2, 3] ;
4  No
5  ?- [X|Xs] = [1].
6  X = 1
7  Xs = [] ;
8  No
9  ?- [X,Y|Xs] = [1,2,3].
10 X = 1
11 Y = 2
12 Xs = [3]
13 Yes
14 ?- [X|Xs] = [].
15 false.
```

Universidad Veracruzana

Recorriendo listas

- Todas las ciudades de la lista son gringas:

```
1  gringa(X) :- loc_en(X,usa).
2
3  gringas([X]) :- gringa(X).
4  gringas([X|Xs]) :-
5    gringa(X),
6    gringas(Xs).
```

```
1  ?- gringas([boston]).
2  true
3  ?- gringas([]).
4  false.
5  ?- gringas([boston,houston,xalapa]).
6  false.
7  ?- gringas(L).
8  L = [atlanta] ;
9  ...
10 L = [atlanta, atlanta] ...
```

Universidad Veracruzana

Funciones de orden superior

- O con maplist/2:

```
?- maplist(gringa, [atlanta, houston, austin]).
true
?- maplist(gringa, [atlanta, houston, xalapa]).
false
```

- Si quieren saber más sobre un predicado predefinido, pueden usar:

```
?- help(maplist).
```

- Si no están seguros del predicado que buscan pueden usar:

```
?- apropos(map).
```

Universidad Veracruzana

Otro recorrido

- Miembro de una lista:

```
1  % miembro(X,L): El elemento X es miembro de la lista L.
2  miembro(X,[X|_]).
3  miembro(X,[_|Xs]) :- miembro(X,Xs).
```

```
1  ?- miembro(1,[]).
2  false.
3  ?- miembro(1,[1]).
4  true
5  ?- miembro(1,[3,2,1]).
6  true
7  ?- miembro(X,[1,2]).
8  X = 1 ;
9  X = 2 ;
10 false.
11 ?- miembro(1,L).
12 L = [1|_G1534] ;
13 L = [_G1533, 1|_G1537] ...
```

Universidad Veracruzana

Usos de miembro

- Podemos computar permutaciones:

```
?- L=[_,_,_], miembro(a,L), miembro(b,L), miembro(c,L).
L = [a, b,c] ;
L = [a, c, b] ;
L = [b, a, c] ;
L = [c, a, b] ;
L = [b, c, a] ;
L = [c, b, a] ;
false.
```

- Podemos buscar en un diccionario:

```
?- Dict = [p(uno,one), p(two,dos), p(three,tres)],
|  miembro(p(two,Esp), Dict).
Dict = [p(uno, one), p(two, dos), p(three, tres)],
Esp = dos
```

Universidad Veracruzana

Mapeo de listas a enteros

- Longitud de una lista:

```
1  % long(+L,-S): S es la longitud de la lista L
2
3  long([],0).
4  long([_|Xs],L) :-
5    long(Xs,L1), L is L1+1.
```

```
1  ?- long([1,2,3,4],L).
2  L = 4;
3  false.
4  ?- long([],L).
5  L = 0;
6  false.
```

Universidad Veracruzana

Traza de programas

- Es posible depurar nuestros programas con un modelo basado en eventos:

Call	Un evento de llamada ocurre cuando Prolog comienza a tratar de satisfacer una meta.
Exit	Un evento de salida ocurre cuando alguna meta es satisfecha.
Redo	Un evento de reintento ocurre cuando Prolog reviene sobre una meta, tratando de satisfacerla nuevamente.
Fail	Un evento de fallo ocurre cuando una meta falla.

Universidad Veracruzana

¿Qué tan complejo es miembro/2?

- La traza de `miembro/2`:

```
?- trace, miembro(1,[3,2,1]).
   Call: (9) miembro(1, [3, 2, 1]) ?
   Call: (10) miembro(1, [2, 1]) ?
   Call: (11) miembro(1, [1]) ?
   Exit: (11) miembro(1, [1]) ?
   Exit: (10) miembro(1, [2, 1]) ?
   Exit: (9) miembro(1, [3, 2, 1]) ?
true

[trace]  ?- notrace.
true.

[debug]  ?- nodebug.
true.

?-
```

Universidad Veracruzana

Entendiendo la recursividad

- La traza de `long/2`:

```
1  ?- trace.
2  [trace]  ?- long([1,2],L).
3     Call: (7) long([1, 2], _G309) ?
4     Call: (8) long([2], _L350) ?
5     Call: (9) long([], _L369) ?
6     Exit: (9) long([], 0) ?
7  ^  Call: (9) _L350 is 0+1 ?
8  ^  Exit: (9) 1 is 0+1 ?
9     Exit: (8) long([2], 1) ?
10 ^  Call: (8) _G309 is 1+1 ?
11 ^  Exit: (8) 2 is 1+1 ?
12    Exit: (7) long([1, 2], 2) ?
13 L = 2
```

- ¿Qué pasa con la pila de ejecución de Prolog?

Universidad Veracruzana

Recursividad a la cola

- Consideren el uso de acumuladores *Acc*:

```
1  longTR(Xs,L) :- long(Xs,0,L).
2
3  long([],Acc,Acc).
4  long([_|Xs],Acc,L) :-
5    Acc1 is Acc + 1,
6    long(Xs,Acc1,L).
```

- Su comportamiento es idéntico a `long/2`:

```
1  ?- longTR([1,2,3,4],L).
2  L = 4;
3  No
4  ?- longTR([],L).
5  L = 0;
6  No
```

Universidad Veracruzana

Recursividad a la cola

- Pero observen la traza:

```
1  [trace]  ?- longTR([1,2,3,4],L).
2     Call: (7) longTR([1, 2, 3, 4], _G315) ?
3     Call: (8) long([1, 2, 3, 4], 0, _G315) ?
4  ^  Call: (9) _L368 is 0+1 ?
5  ^  Exit: (9) 1 is 0+1 ?
6     Call: (9) long([2, 3, 4], 1, _G315) ?
7  ^  Call: (10) _L388 is 1+1 ?
8  ^  Exit: (10) 2 is 1+1 ?
9     Call: (10) long([3, 4], 2, _G315) ?
10 ^  Call: (11) _L408 is 2+1 ?
11 ^  Exit: (11) 3 is 2+1 ?
12    Call: (11) long([4], 3, _G315) ?
13 ^  Call: (12) _L428 is 3+1 ?
14 ^  Exit: (12) 4 is 3+1 ?
15    Call: (12) long([], 4, _G315) ?
16    Exit: (12) long([], 4, 4) ? ...
```

Universidad Veracruzana

A pensar

- La recursión a la cola es más eficiente que la no a la cola: tamaño de stack $O(1)$ vs $O(n)$.
- ¿Cómo probamos experimentalmente si esto es cierto?
- ¿Qué hace esta función misterio?

```
1  mist(0,[]).
2  mist(N,[N|Ns]) :-
3    N1 is N-1,
4    mist(N1,Ns).
```

- Busquen en el manual de Prolog qué hace `time/1`.

Universidad Veracruzana

Resultados

- Con estas herramientas podemos evaluar:

```
?- creaLista(1000,L), time(long(L,R)).
% 2,002 inferences, 0.000 CPU in 0.000 seconds (97% CPU, 11846154 Lips)
L = [1000, 999, 998, 997, 996, 995, 994, 993, 992|...],
R = 1000
```

- Desempeño (time/1):

Método	N	infs	CPU	secs	Lips
long	1000	2002	0.000	0.000	11846154
longTR	1000	1002	0.000	0.000	9920792
long	10000	20001	0.003	0.003	6667000
longTR	10000	10002	0.001	0.001	10572939
long	100000	200001	0.025	0.031	8076606
longTR	100000	100202	0.005	0.005	20222851
long	10000000	5592108	0.665	0.814	**out stack**
longTR	10000000	10000002	0.488	0.489	20488449

Universidad Veracruzana

Operaciones sobre listas

- Agregando dos listas:

```
1  % conc(L1,L2,L3): La lista L3 es el resultado de agregar
2  % L1 a L2.
3
4  conc([],Ys,Ys).
5  conc([X|Xs], Ys, [X|Zs]) :-  conc(Xs,Ys,Zs).
```

```
1  ?- conc([1,2,3],[4,5,6],L).
2  L = [1, 2, 3, 4, 5, 6].
3  ?- conc(X,Y,[1,2,3]).
4  X = [],
5  Y = [1, 2, 3] ;
6  X = [1],
7  Y = [2, 3] ;
8  X = [1, 2],
9  Y = [3] ;
10 X = [1, 2, 3],
11 Y = [] ;
12 false.
```

- Nota. Predefinido como append/3.

Universidad Veracruzana

Otras operaciones sobre listas

```
% add(X,L,R): R es la lista que resulta de agregar X a L

add(X,L,[X|L]).

% del(X,L,R): R es la lista que resulta de eliminar X de L

del(X,[X|Cola],Cola).
del(X,[Y|Cola],[Y|Cola1]) :-
  del(X,Cola,Cola1).
```

```
?- add(1,[],R).
R = [1].
?- add(1,[2,3],R).
R = [1, 2, 3].
?- del(1,[2,3,1,4,5],R).
R = [2, 3, 4, 5]
```

Universidad Veracruzana

Más operaciones sobre listas

- Partes de una lista:

```
1 prefijo(Xs,Ys) :- append(Xs,_,Ys).
2 sufijo(Xs,Ys) :- append(_,Xs,Ys).
3 sublista(Xs,Ys):- prefijo(Aux,Ys), sufijo(Xs,Aux).
```

```
1 ?- prefijo([1,2],[1,2,3]).
2 true
3 ?- sufijo([2,3],[1,2,3]).
4 true
5 ?- sublista([2,3],[1,2,3,4]).
6 true
```

Universidad Veracruzana

Reverso de una lista

- Es posible definir versiones recursivas naïf y a la cola:

```
1  reverso([],[]).
2  reverso([X|Xs],Res):-
3    reverso(Xs,XsReverso),
4    append(XsReverso,[X],Res).
5
6  reversoTR(L,Res):-
7    nonvar(L),
8    reversoTRaux(L,[],Res).
9
10 reversoTRaux([],Acc,Acc).
11 reversoTRaux([X|Xs],Acc,Res) :-
12   reversoTRaux(Xs,[X|Acc],Res).
```

- Prueben con `time/1` cual es más demandante.

Universidad Veracruzana

Notación

- Normalmente usamos notación infija: $2 * a + b * c$.
- Pero Prolog usa notación prefija: $+(*(2, a), *(b, c))$.
- Sin embargo, podemos definir operadores de diferentes maneras:
  - Operadores infijos: `xfx`, `xfy`, `yfx`.
  - Operadores prefijos: `fx`, `fy`.
  - Operadores posfijos: `xf`, `yf`.
- Se asume que `x` tiene menor precedencia que `y`.

Universidad Veracruzana

Ejemplo: Lógica proposicional

- Los operadores proposicionales podrían ser:

```
% operadores proposicionales

:- op(800,xfx,equiv).
:- op(700,xfy,or).
:- op(600,xfy,and).
:- op(500,fy,not).
```

```
?- write_canonical(p and q or not r).
or(and(p,q),not(r))
?- write_canonical(not (p and q) equiv not p or not q).
equiv(not(and(p,q)),or(not(p),not(q)))
```

- ¿Como podría definir la semántica?

Universidad Veracruzana

Ejemplo: Deducción natural

- Extendemos el programa con la semántica de los operadores y las premisas (hechos conocidos):

```
1  % semántica de los operadores
2  or(X,Y) :- X;Y.
3  and(X,Y) :- X,Y.
4  not(X) :- \+ X.
5  equiv(X,Y) :- (\+ X;Y), (\+Y;X).
6
7  % premisas
8  p.
9  q.
```

- $p, q \models \neg(p \wedge q) \equiv \neg p \vee \neg q$

```
1  ?- not (p and q) equiv not p or not q.
2  true
```

Universidad Veracruzana

Datos compuestos

- Los términos compuestos funcionan como estructuras de datos.
- Por ejemplo, el término $punto(X, Y)$ representa un punto con sus coordenadas.
- El término $seg(P1, P2)$ representa un segmento cuyos extremos son los puntos $P1$ y $P2$.
- Las relaciones vertical y horizontal:

```
1 horizontal(seg(punto(X,_), punto(X,_))).
2
3 vertical(seg(punto(_,Y), punto(_,Y))).
```

Universidad Veracruzana

Consulta

- La segunda consulta indica que P es un punto cuya primera coordenada no está instanciada y la segunda es 2.

```
?- vertical(seg(punto(1,2), punto(3,2))).
true

?- vertical(seg(punto(1,2),P)).
P = punto(_G240, 2) ;
false
```

Universidad Veracruzana

Operadores de unificación

- = verifica la unificación (no es un operador de asignación); \= verifica la no unificación.
- Prolog no tiene chequeo de ocurrencias, por eso la última meta incluye $Y = f(Y)$.

```
1  ?- f(X,X) = f(a,Y).
2  X = a
3  Y = a.
4  ?- f(X,X) \= f(a,Y).
5  false.
6  ?- p(f(X),g(Z,X)) = p(Y, g(Y,a)).
7  X = a
8  Z = f(a)
9  Y = f(a).
10 ?- p(X,X) = p(Y,f(Y)).
11 X = Y, Y = f(Y).
```

Universidad Veracruzana

Para implementar la unificación

- El operador =.. convierte predicados en listas:

```
?- papa(X,juan) =.. L.
L = [papa, X, juan].
?- 2 + 3 =.. [Op|Args].
Op =  (+),
Args = [2, 3].
?- R =.. [+,2,3].
R = 2+3.
```

- Prolog tiene definidos predicados para saber el tipo de una fbf: atomic, var, nonvar, etc. (ver manual).

Universidad Veracruzana

Operadores aritméticos

- Prolog predefine los operadores aritméticos básicos:

Operador	Semántica
+	suma
-	resta
/	división
**	potencia
//	división entera
mod	módulo

- También define las funciones trigonométricas estándar y otras funciones matemáticas, p.ej., *sin*\1, *cos*\1, *tan*\1, *atan*\1, *log*\1, *exp*\1, etc.

Universidad Veracruzana

Uso de los operadores aritméticos

- Las operaciones aritméticas son excepcionales, por lo que es necesario indicar a Prolog que se ejecutará una operación de este tipo.
- El operador infijo `is` unifica la expresión a su izquierda, con la evaluación de la expresión a su derecha.

```
1  ?- R is 3+5.
2  R= 8
```

- No confundir el operador *is* con la asignación convencional. X is $X + 1$ no tiene sentido en Prolog.
- El operador $=$ unifica sus argumentos, pero no los evalúa.

```
1  ?- R = 3+5.
2  R= 3+5
```

Universidad Veracruzana

Operadores aritméticos en las reglas

- Es posible usar operadores en las reglas.

```
1  par(X) :- 0 is X mod 2.
2  impar(X) :- 1 is X mod 2.
3  suma(X,Y,Z) :- Z is X+Y.
```

```
1  ?- par(3).
2  false.
3  ?- par(4).
4  true.
5  ?- suma(3,4,R).
6  R=7
```

- Observen que se pierde generalidad.

```
1  ?- suma(3,Y,7).
2  ERROR: is/2: Arguments are not sufficiently instantiated
```

Universidad Veracruzana

Operadores de comparación

- Los operadores para comparaciones incluyen:

Operador	Semántica
`X > Y`	X es mayor que Y.
`X < Y`	X es menor que Y.
`X >= Y`	X es mayor o igual que Y.
`X =< Y`	X es menor o igual que Y.
`X =:= Y`	Los valores de X e Y son iguales.
`X =/= Y`	Los valores de X e Y son diferentes.

- Observen que menor o igual no se escribe como es usual.

Universidad Veracruzana

Uso de comparaciones

- Aplican entre términos numéricos:

```
?- 3+5 > 2+6.
false.
?- 3+5 >= 2+6.
true.
?- 3+5 =\= 2+6.
false.
?- 3 =:= 1 + 2.
true.
```

Universidad Veracruzana

Aritmética declarativa

- Una aproximación declarativa a la aritmética hace uso de las librerías para manejo de restricciones en dominios finitos (clpfd) Triska [4] o
- enteros (clpz): `https://github.com/triska/clpz`
- En swi-prolog es necesario llamar a la siguiente librería:

```
1 :- use_module(library(clpfd)).
```

al inicio de nuestro programa (o agregarlo a la configuración global de Prolog: `/.config/swi-prolog/init.pl`).

Universidad Veracruzana

Restricciones

- Las restricciones más importantes definidas por clpfd son:

Restricción	Semántica
`A #= B`	A es igual a B
`A #< B`	A es menor que B
`A #> B`	A es mayor que B
`A #\= B`	A no es igual a B

donde `A` y `B` son expresiones aritméticas:

- Si $\alpha \in \mathit{Vars}$ entonces α es una expresión aritmética.
- Si α es un número, entonces α es una expresión aritmética.
- Si α y β son expresiones aritméticas, también lo son $\alpha + \beta$, $\alpha - \beta$, y $\alpha * \beta$.

- Ejemplos:

```
1  ?- X #= 5 + 3.
2  X = 8.
3  ?- 2 #= X + 9.
4  X = -7.
```

Universidad Veracruzana

Problema CLP

- Un granjero tiene un total de 30 animales, entre vacas y gallinas. Entre todos ellos tienen 74 patas ¿Cuantas gallinas y vacas tiene el granjero?

```
1  :- use_module(library(clpfd)).
2
3  sol(Gallinas,Vacas) :-
4     Gallinas + Vacas #= 30,
5     Gallinas * 2 + Vacas * 4 #= 74,
6     Gallinas in 0..sup,
7     Vacas in 0..sup.
```

- Ejecución:

```
1  ?- sol(G,V).
2  G = 23,
3  V = 7.
```

Universidad Veracruzana

Ejemplo clásico: Factorial

- Versión naïf:

```
1  fact(0,1).
2  fact(N,F) :-
3    N>0,
4    N1 is N-1,
5    fact(N1,F1),
6    F is N* F1.
```

- Versión recursiva a la cola:

```
1  fact_tr(X,F) :- fact_tr(X,1,F).
2
3  fact_tr(0,Acc,Acc).
4  fact_tr(N,Acc,R) :-
5    N>0,
6    N1 is N-1,
7    Acc1 is Acc*N,
8    fact_tr(N1,Acc1,R).
```

Universidad Veracruzana

Una versión con restricciones

- Versión por restricciones:

```
1  fact_fd(0, 1).
2  fact_fd(N, F) :-
3      N #> 0,
4      N1 #= N - 1,
5      F #= N * F1,
6      fact_fd(N1, F1).
```

que puede computar:

```
1  ?- fact_fd(N,F).
2      N = 0, F = 1
3   ;  N = 1, F = 1
4   ;  N = 2, F = 2
5   ;  N = 3, F = 6
6   ;  N = 4, F = 24
7   ;  ... .
```

Universidad Veracruzana

Combinando

- Una versión recursiva a la cola con restricciones:

```
1  fact_fd_tr(N,F) :-
2      fact_fd_tr(N,1,F).
3
4  fact_fd_tr(0,Acc,Acc).
5  fact_fd_tr(N,Acc,F) :-
6      N #> 0,
7      N1 #= N - 1,
8      Acc1 #= Acc * N,
9      fact_fd_tr(N1,Acc1,F).
```

Universidad Veracruzana

Comparación usando time

- Mismo número de inferencias, tiempo de convergencia diferente:

N	Alg	No. Infs	CPU %	T(segs)
1000	fact	3000	0.003	0.006
	fact_tr	3001	0.001	0.001
	fact_fd_tr	3001	0.002	0.002
10000	fact	30,000	0.053	0.058
	fact_tr	30,001	0.018	0.018
	fact_fd_tr	30,001	0.021	0.022
100000	fact	300,000	6.922	7.007
	fact_tr	300,001	1.657	1.718
	fact_fd_tr	300,001	1.587	1.627

- Usen una variable anónima para que Prolog no imprima el resultado: `fact(100000,_).`

Universidad Veracruzana

Valores a partir de listas

- Estas funciones no utilizan la recursión a la cola, no pueden ser optimizadas.

```
1  sum([],0).
2  sum([X|Xs],S) :-
3    sum(Xs,Sc),
4    S is Sc + X.
5
6  long([],0).
7  long([X|Xs],L):-
8    long(Xs,Lc),
9    L is Lc + 1.
```

```
1  ?- sum([1,2,3,4,5],R).
2  R = 15.
3  ?- long([1,2,3,4,5],R).
4  R = 5.
```

Universidad Veracruzana

Listas a partir de valores

- Genera una lista creciente cuyo primer valor es X y el último valor es Y.

```
1  intervalo(X,X,[X]).
2  intervalo(X,Y,[X|Xs]) :-
3    X < Y, Z is X + 1,
4    intervalo(Z,Y,Xs).
```

```
1  ?- intervalo(1,5,R).
2  R= [1, 2, 3, 4, 5]
3  Yes
4  ?- intervalo(3,1,R).
5  false.
```

Universidad Veracruzana

Listas por filtrado

- *pares*/2 filtra los elementos pares en su primer argumento.

```
1  pares([],[]).
2  pares([X|Xs],[X|Ys]) :-
3    par(X), pares(Xs,Ys).
4  pares([X|Xs],Ys) :-
5    impar(X), pares(Xs,Ys).
```

- Aquí se combina con *intervalo*/3 en una meta compuesta.

```
1  ?- intervalo(1,5,Aux), pares(Aux, Resp).
2  Aux = [1, 2, 3, 4, 5]
3  Resp = [2, 4]
```

Universidad Veracruzana

¿Lista ordenada?

- El predicado *ordenada*/1 es verdadero si su argumento es una lista ordenada:

```
1 ordenada([]).
2 ordenada([_]).
3 ordenada([X,Y|Ys]) :- X<Y, ordenada([Y|Ys]).
```

```
1 ?- ordenada([1,2,3]).
2 true
3 ?- ordenada([1,3,2]).
4 false.
```

Universidad Veracruzana

Ordenando una lista (poco eficiente)

- Declarativamente, una permutación ordenada:

```
?- permutation([2,3,4,1,5],L),ordenada(L).
L = [1, 2, 3, 4, 5]
```

- Se puede generalizar el caso:

```
% sortPerm(L1,L2): L2 es la lista L1 ordenada

sortPerm(L1,L2) :-
  permutation(L1,L2),
  ordenada(L2).
```

Universidad Veracruzana

Inserción en una lista ordenada

- *inserta*/3 pone el primer elemento en la lista ordenada segundo argumento, en el lugar correspondiente.

```
1  inserta(X,[],[X]).
2  inserta(X,[Y|Ys],[X,Y|Ys]) :-
3    X < Y.
4  inserta(X,[Y|Ys],[Y|Zs]) :-
5    X >= Y, inserta(X,Ys,Zs).
```

```
1  ?- inserta(4,[1,2,3,5],R).
2  R = [1, 2, 3, 4, 5]
```

Universidad Veracruzana

Ordenando listas (por inserción)

- Ahora podemos ordenar una lista por inserción:

```
1  insertSort([],[]).
2  insertSort([X|Xs],S):-
3    insertSort(Xs,S1),
4    inserta(X,S1,S).
```

```
1  ?- insertSort([1,5,3,2,4],R).
2  R = [1, 2, 3, 4, 5]
```

Universidad Veracruzana

O por burbuja

```
1  bubbleSort(L,S):-
2    swap(L,L1),!,
3    write(L1), nl,
4    bubbleSort(L1,S).
5  bubbleSort(S,S).
6
7  swap([X,Y|Ys],[Y,X|Ys]):- X>Y.
8  swap([Z|Zs],[Z|Zs1]):- swap(Zs,Zs1).
```

```
1  ?- bubbleSort([1,5,3,2,4],R).
2  [1, 3, 5, 2, 4]
3  [1, 3, 2, 5, 4]
4  [1, 2, 3, 5, 4]
5  [1, 2, 3, 4, 5]
6  R = [1, 2, 3, 4, 5].
```

Universidad Veracruzana

O por QuickSort

```
1  quickSort([],[]).
2
3  quickSort([X|Xs],Res) :-
4     split(X,Xs,MasPeques,MasGrandes),
5     quickSort(MasPeques,PequesOrd),
6     quickSort(MasGrandes,GrandesOrd),
7     append(PequesOrd,[X|GrandesOrd],Res).
8
9  split(_,[],[],[]).
10 split(X,[Y|Ys],[Y|Peques],Grandes) :-
11    X > Y, !,
12    split(X,Ys,Peques,Grandes).
13 split(X,[Y|Ys],Peques,[Y|Grandes]) :-
14    split(X,Ys,Peques,Grandes).
```

```
1  ?- quickSort([1,5,3,2,4],R).
2  R = [1, 2, 3, 4, 5]
```

Universidad Veracruzana

Y podemos compararlos:

- Por número de elementos a ordenar:

Algoritmo	N=100	200	500
bubbleSort	0.28	2.05	32.33
ordenaIns	0.01	0.04	00.27
quickSort	0.00	0.00	00.01
msort	0.00	0.00	00.00

- ¿Que algoritmo usan *sort*/2 y *msort*/2 en SWI-Prolog?

Universidad Veracruzana

Nuevos datos

N	Algoritmo	No. Infs	t(segs)
1000	bubbleSort	224,247,047	39.62
	ordenaIns	749,400	0.193
	quickSort	31,323	0.007
	sort	1	0.001
	msort	1	0.000
10000	quickSort	461,025	0.133
	sort	1	0.004
100000	quickSort	13,494,872	3.779
	sort	1	0.032

Universidad Veracruzana

Quicksort concurrente!

```
1  quickSortConc(Xs, Ys) :-
2      quickSortConcLevel(Xs, Ys, 0).
3
4  quickSortConcLevel([X|Xs], Ys, Level) :-
5      split(X,Xs,Left,Right),
6      NextLevel = Level + 1,
7      (    NextLevel < 8
8      ->   concurrent(2, [quickSortConcLevel(Left, Ls, NextLevel),
9                          quickSortConcLevel(Right, Rs, NextLevel)], [])
10         ;
11         quickSort(Left, Ls),
12         quickSort(Right, Rs)
13     ),
14     append(Ls, [X|Rs], Ys).
15
16 quickSortConcLevel([], [], _).
```

Universidad Veracruzana

Benchmark para los Quicksort

```
1  bench :-
2      randseq(200000, 500000, L),
3      write('Normal:'), nl,
4      time(quickSort(L,_Sol)),
5      write('Concurrente:'), nl,
6      time(quickSortConcurrent(L,_SolC)),
7      write('Sort:'), nl,
8      time(sort(L,_SolS)).
```

```
1  ?- bench.
2  Normal:
3  % 12,068,420 inferences, 3.257 CPU in 3.371 seconds (97% CPU, 3705763 Lips)
4  Concurrente:
5  % 558,947 inferences, 0.093 CPU in 1.490 seconds (6% CPU, 6029763 Lips)
6  Sort:
7  % 1 inferences, 0.167 CPU in 0.171 seconds (98% CPU, 6 Lips)
```

Universidad Veracruzana

Codificador

- Supongan que queremos una función para compactar una lista de caracteres:

```
1  comprime([],[]).
2  comprime([X|Xs],Ys) :- comp(Xs,X,1,Ys).
3
4  comp([],C,N,[cod(C,N)]).
5  comp([X|Xs],X,N,Ys) :- N1 is N+1,
6      comp(Xs,X,N1,Ys).
7  comp([X|Xs],Y,N,[cod(Y,N)|Ys]):- N\=Y,
8      comp(Xs,X,1,Ys).
```

- Su uso es:

```
1  ?- comprime([1,1,1,1,0,0,0,1,1],Res).
2  Res = [cod(1, 4), cod(0, 3), cod(1, 2)]
```

Universidad Veracruzana

Sub-término

- Se puede verificar si un término es sub-término de otro:

```
1  subtermino(T,T).
2  subtermino(ST,T) :-
3      compound(T),
4      functor(T,F,A),
5      subtermino(A,ST,T).
6
7  subtermino(A,ST,T) :-
8      A>1, A1 is A-1, subtermino(A1,ST,T).
9  subtermino(A,ST,T) :-
10     arg(A,T,Arg), subtermino(ST,Arg).
```

- Una definición doblemente recursiva.

Universidad Veracruzana

Corrida subtérmino

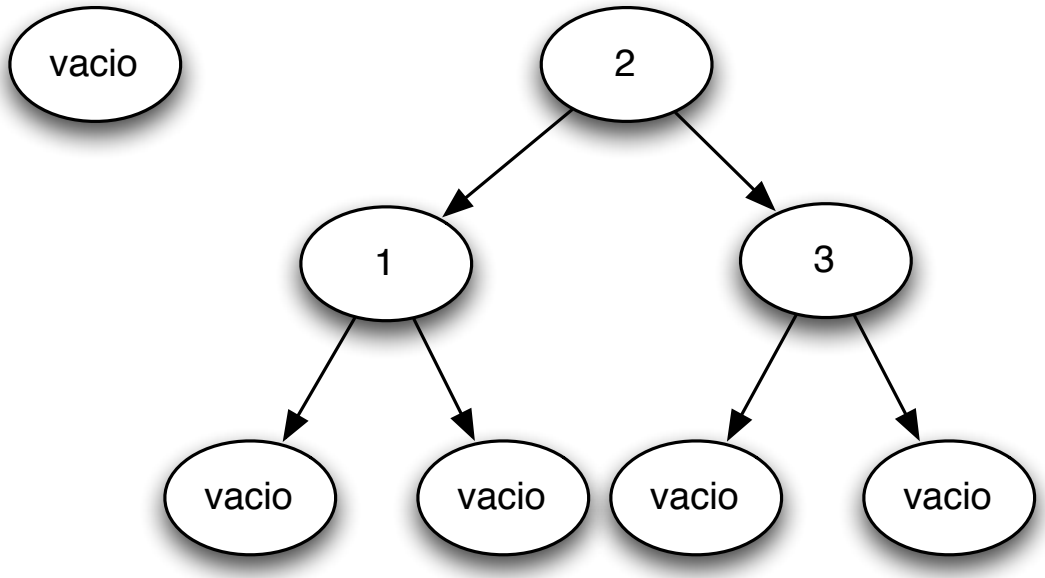
- Ejemplos de consultas:

```
?- subtermino(f(a),g(f(a),b)).
true
?- subtermino(x,g(f(a),b)).
false.
?- subtermino(X,g(f(a),b)).
X = g(f(a), b) ;
X = f(a) ;
X = a ;
X = b ;
false.
```

Universidad Veracruzana

Arboles binarios

- Un árbol es un nodo vacío, o un nodo con dos hijos que son árboles.



Universidad Veracruzana

Insertar un elemento

```
1  % insertaArbol(E,A,A1): inserta elemento E en árbol A, resultando
2  % en árbol A1.
3
4  insertaArbol(X,vacio,arbol(X,vacio,vacio)).
5
6  insertaArbol(X,arbol(X,A1,A2),arbol(X,A1,A2)).
7
8  insertaArbol(X,arbol(Y,A1,A2),arbol(Y,A1N,A2)) :-
9      X<Y, insertaArbol(X,A1,A1N).
10 insertaArbol(X,arbol(Y,A1,A2),arbol(Y,A1,A2N)) :-
11     X>Y, insertaArbol(X,A2,A2N).
```

```
1  ?- Insertaarbol(1,vacio,Arbol1), insertaArbol(2,Arbol1,Arbol2).
2  Arbol1 = arbol(1, vacio, vacio),
3  Arbol2 = arbol(1, vacio, arbol(2, vacio, vacio))
```

Universidad Veracruzana

Lista a Arbol

```
% lista2arbol(L,A): Crea el árbol A a partir de la lista L

lista2arbol(L,A) :-
    creaArbol(L,vacio,A).

creaArbol([],A,A).
creaArbol([X|Xs],AAux,A) :-
    insertaArbol(X,AAux,A2),
    creaArbol(Xs,A2,A).
```

```
?- lista2arbol([2,1,3],A).
A = arbol(2, arbol(1, vacio, vacio), arbol(3, vacio, vacio))
```

Universidad Veracruzana

Nodos de un Arbol

```
1  % nodos(A,L): L es la lista de los nodos en A.
2
3  nodos(vacio,[]).
4  nodos(arbol(X,A1,A2),Xs) :-
5      nodos(A1,Xs1),
6      nodos(A2,Xs2),
7      append(Xs1,[X|Xs2],Xs).
```

```
1  ?- lista2arbol([1,2,3],A), nodos(A,L).
2  A = arbol(1, vacio,arbol(2, vacio, arbol(3, vacio, vacio))),
3  L = [1, 2, 3]
```

- ¿Cómo se pueden listar los nodos en pre-orden o post-orden?

Universidad Veracruzana

Ordena Lista por Inserción en Arbol

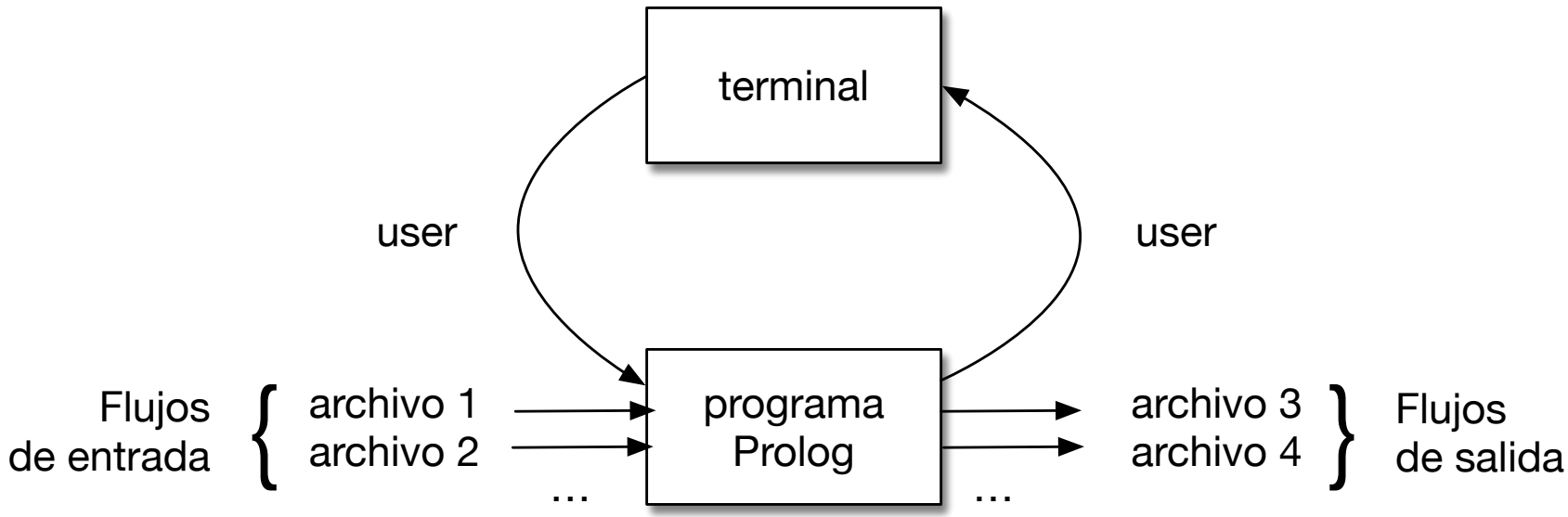
```
1  % ordenaLista(L1,L2): L2 es la lista L1 ordenada
2
3  ordenaLista(L1,L2) :-
4      lista2arbol(L1,A),
5      nodos(A,L2).
```

```
1  ?- ordenaLista([4,2,3,1,5],L).
2  L = [1, 2, 3, 4, 5]
```

Universidad Veracruzana

Entradas y Salidas en Prolog

- Además de las metas, el usuario puede interactuar con Prolog usando flujos de entrada y salida:



- El teclado y el monitor son los flujos (`user`) por default.

Universidad Veracruzana

write/1, nl/0 y tab/1

- *write*/1 toma un argumento que debe ser un término Prolog.
- La evaluación de este predicado causa (bye bye programación lógica pura) que su argumento sea escrito en el flujo de salida actual (por default el monitor).
- *nl*/0 inserta un salto de línea en el flujo de salida. nueva línea en el flujo de salida.
- *tab*/1 toma un argumento entero n, e inserta n espacios en blanco en el flujo de salida.

Universidad Veracruzana

Ejemplos

```
?- write(38).
38
true.
?- write('una cadena de caracteres').
una cadena de caracteres
true.
?- write([a,b,c,d,[e,f,g]]), nl.
[a, b, c, d, [e, f, g]]
true.
?- write('Maestría en'), nl, write('IA'),nl.
Maestría en
IA
true.
?- write('Centro de Investigación'),tab(5),write('en IA').
Centro de Investigación-----en IA
true.
```

Universidad Veracruzana

writeq/1

- Observen que el apóstrofo se pierde en la salida al usar *write*/1. Si es importante conservarlo, se puede usar *writeq*/1:

```
?- writeq('una cadena de caracteres').
'una cadena de caracteres'
true.

?- writeq('mira mis apóstrofos').
'mira mis apóstrofos'
true.

?- write('ahora no los ves').
ahora no los ves
true.
```

Universidad Veracruzana

Ejemplo: writelist/1

```
1  % writelist(L): escribe un miembro de L por renglón
2
3  writelist([]).
4  writelist([H|T]) :-
5      write(H), nl,
6      writelist(T).
```

```
1  ?- writelist([[1,2,3],[4,5,6],[7,8,9]]).
2  [1,2,3]
3  [4,5,6]
4  [7,8,9]
5  true.
```

Universidad Veracruzana

Ejemplo: Gráfica de barras

```
1  % bars(L): gráfica barras con los valores en la lista L
2
3  bars([]).
4  bars([H|T]) :-
5      stars(H),
6      bars(T).
7
8  stars(N) :-
9      N>0,
10     write(*),
11     N1 is N-1,
12     stars(N1).
13
14 stars(_) :- nl.
```

```
1  ?- bars([2,5,3]).
2  **
3  *****
4  ***
5  true
```

Universidad Veracruzana

read/1

- El predicado *read*/1 provee la lectura de términos de entrada.
- Su único argumento debe ser un término, normalmente una variable.
- Su evaluación causa que su argumento unifique con el valor leído en el flujo de entrada actual (por default el teclado).
- La lectura del término termina con un punto "." y un carácter en blanco, por ejemplo, un salto de línea o un espacio.
- Al momento de la lectura el *prompt* de Prolog suele cambiar, por ejemplo a:

```
1 |:
```

Universidad Veracruzana

Ejemplos

```
?- read(X).
|: juan.
X = juan.
?- read(X).
|: mi_predicado_tonto(b).
X = mi_predicado_tonto(b).
?- read(p(X)).
|: p(5).
X = 5.
?- read(p(X)).
|: 5.
false.
```

- Si la variable ya tuviese un valor asociado, la meta tiene éxito sólo si el nuevo valor unifica con el anterior!

```
?- X=alfredo, read(X).
|: juan.
false.
```

Universidad Veracruzana

ASCII

- Para algunas aplicaciones es más conveniente bajar el nivel de las operaciones de E/S a caracteres, usando código ASCII para manipular cadenas.

valor	signo	valor	signo	valor	signo	valor	signo
9	tab	40	(	59	;	94	^
10	eof	41	)	60	<	95	_
32	espacio	42	*	61	=	96	'
33	!	43	+	62	>	97–122	a – z
34	"	44	,	63	?	123	{

- UTF no está definido en el estándar Prolog.

Universidad Veracruzana

Salida de caracteres

- Se utiliza en predicado *put*/1 cuyo argumento debe ser un número en 0 . . . 255

```
?- put(97), nl.
a
true.
?- put(122), nl.
z
true.
?- put(64), nl.
@
true.
```

Universidad Veracruzana

Entrada de caracteres

- Se utilizan los predicados *get*0/1 y *get*/1. El primero puede aceptar caracteres en blanco. El segundo no.

```
?- get0(X).
|: a
X = 97.
?- get(X).
|: a
X = 97.
?- get0(X).
|:
X = 32.
?- get(X).
|:
|: .
X = 46.
```

Universidad Veracruzana

Preguntita

- ¿Qué hace el siguiente programa lógico con la meta *readin*?

```
1  readin :-
2      get0(X),
3      process(X).
4
5  process(42).
6  process(X) :-
7      X =\= 42,
8      write(X), nl,
9      readin.
```

Universidad Veracruzana

Corrida

- El predicado lee una secuencia de caracteres hasta encontrar "*". Imprime uno a uno de estos caracteres excluyendo "*".

```
?- readin.
|: hola*
104
111
108
97
```

Universidad Veracruzana

Contador de caracteres

```
1  cuenta(Total) :-
2      cuenta_aux(0,Total).
3  cuenta_aux(Acc,Res) :-
4      get0(X), process(X,Acc,Res).
5
6  process(42,Acc,Acc).
7  process(X,Acc,Res) :-
8      X=\=42, Acc1 is Acc+1,
9      cuenta_aux(Acc1,Res).
```

```
1  ?- cuenta(X).
2  |: hola*
3  X = 4
```

Universidad Veracruzana

Cuenta vocales

```
1  cuenta_vocales(Total) :-
2      cuenta_vocales_aux(0,Total).
3  cuenta_vocales_aux(Acc,Res) :-
4      get0(X), process_vocales(X,Acc,Res).
5
6  process_vocales(42,Acc,Acc).
7  process_vocales(X,Acc,Res) :-
8      X=\=42, processChar(X,Acc,N),
9      cuenta_vocales_aux(N,Res).
10
11 processChar(X,Acc,N) :-
12     vocal(X), N is Acc+1.
13 processChar(_,Acc,Acc).
14
15 vocal(65). %%% Mayúsculas
16 vocal(69). vocal(73).
17 vocal(79). vocal(85).
18 vocal(97). %%% Minúsculas
19 vocal(101). vocal(105).
20 vocal(111). vocal(117).
```

Universidad Veracruzana

Corridas cuenta vocales

```
?- cuenta_vocales(X).
|: andaba herrado*
X = 6
?- cuenta_vocales(Z).
|: kdvzktvsk*
Z = 0
```

Universidad Veracruzana

Salida: tell/1, told/0 y telling/1

- Para cambiar el flujo de salida actual se usa *tell*/1 cuyo argumento es una variable o un átomo representando un nombre de archivo. Ej: *tell*('*salida.txt*')
- El flujo de salida estándar es *user*.
- El predicado *told*/0 cierra el flujo actual de salida y vuelva a establecer *user* como el flujo actual de salida.
- El argumento de *telling*/1 suele ser una variable que unifica con el nombre del flujo de salida actual.
- Ejemplo:

```
1  ?- telling(X). X = user.
```

Universidad Veracruzana

Entrada: see/1, seen/0 y seeing/1

- El argumento de *see*/1 suele ser una variable o un átomo representando el nombre de un archivo. Ej: *see*('*mientrada.txt*')
- Al evaluar este predicado, el archivo se vuelve el flujo de entrada actual. Si no es posible abrir el archivo se genera un error.
- El flujo de entrada estándar también es *user*.
- Para cerrar el flujo actual de entrada se usa *seen*/0.
- El argumento de *seeing* /1 suele ser una variable que unifica con el identificador del flujo de entrada actual.

Universidad Veracruzana

Lectura de archivos

- El predicado *read*/1 normalmente toma como argumento una variable que unifica con la línea leída del flujo actual de entrada.
- Si el final del archivo es alcanzado al usar *read*/1 la variable unifica con el átomo *end_of_file*
- Si el final de archivo se alcanzo usando *get*0/1 ó *get*/1, el valor unificado con la variable depende de la implementación, pero siempre está fuera del rango de los caracteres ASCII,
por ejemplo -1.

Universidad Veracruzana

Ejemplo

- Contar vocales de un archivo.

```
1  vocs_file(Arch,Total) :-
2      see(Arch),
3      cuenta_vocales_file(0,Total),
4      seen.
5
6  cuenta_vocales_file(Acc,Res) :-
7      get0(X), process_vocales_file(X,Acc,Res).
8
9  process_vocales_file(-1,Acc,Acc).
10 process_vocales_file(X,Acc,Res) :-
11     X =\= -1, processChar(X,Acc,N),
12     cuenta_vocales_file(N,Res).
```

```
1  ?- vocs_file('/Users/aguerra/intro.tex',R).
2  R = 2184
```

Universidad Veracruzana

Ejemplo

- El predicado *copychars*/1 copia una secuencia de caracteres que termina en "!" de la terminal a un archivo especificado:

```
1  copychars(Salida):-
2      telling(Actual), tell(Salida),
3      copychars_aux, told, tell(Actual).
4
5  copychars_aux :-
6      get0(C), process_copy(C).
7
8  process_copy(33).
9  process_copy(C) :-
10     C =\= 33, put(C), copychars_aux.
```

Universidad Veracruzana

Leyendo archivos CSV

```
1  % load_exs(CSV_file,Atts,Exs): read the CSV to obtain the list of
2  % attributes Atts for the learning setting, and the list of training
3  % examples Exs.
4
5  load_exs(CSV_file,Atts,Exs) :-
6      csv_read_file(CSV_file,[AttsAux|ExsAux], [strip(true)]),
7      AttsAux =.. [_|Atts],
8      maplist(proc_ex,ExsAux,Exs),
9      length(Exs,NumExs),
10     write(NumExs), write(' training examples loaded.'), nl.
11
12 % proc_ex(Ex,Args): return the arguments Args of a training example Ex.
13
14 proc_ex(Ex,Args) :-
15     Ex =.. [_|Args].
```

Universidad Veracruzana

Un conocido CSV

```
outlook, temperature, humidity, wind, class
sunny, hot, high, weak, no
sunny, hot,  high,  strong, no
overcast, hot,  high, weak, yes
rain, mild,  high, weak, yes
rain, cool, normal, weak, yes
rain, cool, normal, strong, no
overcast, cool, normal, strong, yes
sunny, mild, high, weak, no
sunny, cool, normal, weak, yes
rain, mild, normal, weak, yes
sunny, mild, normal, strong, yes
overcast, mild, high, strong, yes
overcast, hot, normal, weak, yes
rain, mild, high, strong, no
```

Universidad Veracruzana

Leyendo el archivo CSV

```
?- load_exs('tenis.csv',Atts,Exs).
14 training examples loaded.
Atts = [outlook, temperature, humidity, wind, class],
Exs = [[sunny, hot, high, weak, no], [sunny, hot, high, strong, no],
       [overcast, hot, high, weak, yes], [rain, mild, high, weak, yes],
       [rain, cool, normal, weak|...], [rain, cool, normal|...],
       [overcast, cool|...], [sunny|...], [...|...]|...].
```

Universidad Veracruzana

O con Scryer

- Utilizando gramáticas definitivas, mapeos y funciones anónimas:

```
1  :- use_module(library(pio)).
2  :- use_module(library(charsio)).
3  :- use_module(library(csv)).
4  :- use_module(library(lambda)).
5
6  load_exs(CSV_file, Atts, Exs) :-
7      phrase_from_file(parse_csv(frame(Header,Data)),CSV_file),
8      maplist(string_to_term,Header,Atts),
9      maplist(\\X^(maplist(string_to_term,X)),Data,Exs).
10
11  string_to_term(S,T) :-
12      append(S,".", S_aux),
13      read_from_chars(S_aux,T).
```

Universidad Veracruzana

Computando los dominios de los atributos

```
1  % domain_atts(Atts,Exs,Doms): return the domains Doms of the attributes
2  % Atts, given the examples Exs.
3
4  domain_atts(Atts,Exs,Doms) :-
5      findall([Att,Dom],(member(Att,Atts),
6                         nth0(Idx,Atts,Att),
7                         column_exs(Idx,Exs,Vals),
8                         list_to_set(Vals,Dom)),
9              Doms).
10
11 % column(N,Exs,Vals): returns the values Vals for the Nth attribute in
12 % the examples Exs.
13
14 column_exs(N,Exs,Vals) :-
15     maplist(nth0(N),Exs,Vals).
```

Universidad Veracruzana

Ejemplo de corrida

```
?- load_exs('tenis.csv',Atts,Exs), domain_atts(Atts,Exs,Doms).
14 training examples loaded.
Atts = [outlook, temperature, humidity, wind, class],
Exs = [[sunny, hot, high, weak, no], [sunny, hot, high, strong, no],
       [overcast, hot, high, weak, yes], [rain, mild, high, weak, yes],
       [rain, cool, normal, weak|...], [rain, cool, normal|...],
       [overcast, cool|...], [sunny|...], [...|...]|...],
Doms = [[outlook, [sunny, overcast, rain]], [temperature, [hot, mild, cool]],
        [humidity, [high, normal]], [wind, [weak, strong]],
        [class, [no, yes]]].
```

Universidad Veracruzana

Paquetes

- SWI-Prolog puede usar paquetes de terceros:

  `https://www.swi-prolog.org/pack/list`

- El listado se puede consultar con la siguiente meta:

```
?- pack_list(bib).
% Contacting server at https://www.swi-prolog.org/pack/query ... ok
p bibtex@0.1.8                  - Parser and predicates for BibTeX files
true.
```

- Y el paquete se puede instalar con:

```
?- pack_install(bibtex).
```

Universidad Veracruzana

Uso de paquete BibTex

- Se puede cargar un archivo `bib` como hechos:

```
1  :- use_module(library(bibtex_cmd)).
2  :- dynamic entry/3.
3
4  % Assert a list of entries
5
6  assert_entries_list([]).
7  assert_entries_list([E|Es]) :-
8    assertz(E),
9    assert_entries_list(Es).
10
11 % Assert entries from file Bib
12
13 assert_entries(Bib) :-
14   bibtex:bibtex_file(Bib, LstEntries),
15   assert_entries_list(LstEntries).
```

Universidad Veracruzana

Corrida

- Cargar una bibliografía:

```
?- assert_entries("biblio.bib").
true.
```

- Cada entrada bibliográfica tiene la forma:

```
?- entry(Type,Key,Fields).
Type = book,
Key = "Flach2007",
Fields = [field(address, "Bristol, UK"), field(author, "Flach, Peter"),
field(publisher, "Wiley \\& Sons Ltd"),
field(title, "Simply Logical: Intelligent Reasoning by Example"),
field(year, "2007")]
```

Universidad Veracruzana

Colaboradores

- Después de programar las funciones de acceso:

```
1  get_collaborators(Author,Collabs) :-
2    get_keys_by_author(Author, Keys),
3    maplist(get_author, Keys, Authors),
4    foldl(union, Authors, [], CollabsAux),
5    delete(CollabsAux,Author,Collabs).
```

- Corrida:

```
1  ?- get_collaborators("Guerra-Hernández, Alejandro",C), length(C, Num_collabs).
2  C = ["Hübner, F Jomi", "Bordini, Rafael H", |...],
3  Num_collabs = 61 ;
4  false.
```

Universidad Veracruzana

DCG

- Las Gramática de Cláusulas Definitivas (DCG) describen secuencias.
- Se usan para analizar (*parse*), generar, completar y greenvalidar secuencias representadas como listas.
- Su estructura general es:

```
1  Cabeza --> Cuerpo.
```

- El Cuerpo de estas reglas puede incluir:

Terminales.	Una lista, denota los elementos que contiene.
No terminales.	Una DCG u otro constructor gramatical, denota los elementos que describe.
Metas gramaticales.	Una meta tiene definitiva invocada entre llaves, e.g., { Meta }.

Universidad Veracruzana

Ejemplo

- La siguiente DCG describe sequencias de 'as':

```
1  as --> [].
2  as --> [a], as.
```

- Una DCG se invoca usando *phrase*/2:

```
1  ?- phrase(as, Ls).
2     Ls = []
3  ;  Ls = [a]
4  ;  Ls = [a,a]
5  ;  Ls = [a,a,a]
6  ;  Ls = [a,a,a,a]
7  ;  ... .
```

- Si queremos representar las cadenas de texto como listas de caracteres, podemos usar:

```
1  ?- set_prolog_flag(double_quotes, chars).
2  true.
```

Universidad Veracruzana

Otros usos de la DCG

- Podemos verificar si una cadena de texto es una expresión válida de la gramática:

```
?- phrase(as, "aaa").
   true
;  false.

?- phrase(as, "bcd").
   false.
```

- Podemos completar una cadena de texto para que satisfaga la gramática:

```
?- phrase(as, [a,X,a]).
   X = a
;  false.
```

Universidad Veracruzana

seq//1

- El no terminal *seq*//1 describe una secuencia de elementos. Su único argumento denota la secuencia que describe, representada como una lista:

```
1  seq([])     --> [].
2  seq([E|Es]) --> [E], seq(Es).
```

- Ejemplo.

```
1  ?- phrase(("Hola, ",seq(Cs),"!"), "Hola, todos!").
2     Cs = "todos"
3  ;  false.
```

Universidad Veracruzana

seqq/1

- El no terminal *seqq*/1 describe secuencia de secuencias!

```
1  seqq([]) --> [].
2  seqq([Es|Ess]) -->
3          seq(Es),
4          seqq(Ess).
```

- Ejemplo.

```
1  ?- phrase(seqq(["ab","cd","ef"]), Ls).
2     Ls = "abcdef".
```

Universidad Veracruzana

.../0

- El no terminal .../0 describe cualquier secuencia.

```
1  ... --> [] | [_], ... .
```

- Ejemplo.

```
1  phrase((...,[Ultimo]), "Hola").
2     Ultimo = a
3  ;  false.
```

- Todos estos operadores están definidos en la librería *dcgs* de scryer-prolog.
- Para más información ver la sección correspondiente de *The Power of Prolog* de Markus Triska.

Universidad Veracruzana

Y la bibliografía revisitada

- Un pequeño programa para leer archivos bib:

```
entry([Kind,Author,Title]) --> ..., "@", seq(Kind), "{",..., "author =
↪  {",seq(Author),"}", ... ,"title = {",seq(Title),"}", ..., "}}", ... .

entries([[K,A,T]|Es]) --> entry([K,A,T]), entries(Es).
entries([]) --> [].

get_biblio_from_file(File, L) :-
    phrase_from_file(entries(L), File).
```

- Que funciona así:

```
?- get_biblio_from_file('biblio.bib', L).
   L = [["book","Flach, Peter","Simply Logical: I ..."] ...
```

Universidad Veracruzana

Referencias I

[1] I Bratko. *Prolog programming for Artificial Intelligence*. Fourth. Essex, England: Pearson, 2012.

[2] WF Clocksin y CS Melish. *Programming in Prolog, using the ISO standard*. Berlin-Germany: Springer-Verlag, 2003.

[3] L Sterling y E Shapiro. *The Art of Prolog*. Cambridge, MA, USA: The MIT Press, 1999.

[4] M Triska. "The Finite Domain Constraint Solver of SWI-Prolog". En: *FLOPS 2012*. Ed. por T Schrijvers y P Thiemann. Vol. 7294. Lecture Notes in Computer Sciences. Berlin, Germany: Springer-Verlag, 2012, págs. 307-316.

Universidad Veracruzana