3 PROLOG

Este capítulo está concebido como un tutorial breve sobre el lenguaje de programación lógica Prolog. Después de abordar brevemente la instalación del lenguajes, introduciremos los conceptos básicos de la programación en Prolog, incluyendo: Términos y cláusulas, listas, árboles binarios como términos compuestos, operaciones de entrada y salida y corte. Es de esperar que con estos antecedentes, nos sea más natural atacar los problemas propios de la IA con este lenguaje de programación.

3.1 INSTALACIÓN

En este tutorial utilizaremos SWI-Prolog, en su versión 7.4.2. La página oficial de esta implementación se encuentra en `http://www.swi-prolog.org`. Los instaladores para Windows y OS X, pueden bajarse desde esta página. Las distribuciones para Linux están disponibles desde los instaladores de paquetes de las diferentes versiones de este sistema operativo. También es posible instalar el lenguaje en macOS, usando `Homebrew` [1], con la fórmula:

```
> brew install --with-xpce --with-jpl swi-prolog
```

La opción `with-xpce` es necesaria para incluir las herramientas gráficas de SWI-Prolog en la instalación. Estas herramientas requieren que este instalado un manejador de ventanas X, p.ej., `XQuartz` [2]. En que lo sigue, asumiremos que SWI-Prolog ha sido instalado en MacOS, con soporte para las herramientas gráficas. Si se desea también usar SWI-Prolog con Java, es necesario incluir la opción `with-jpl`. La fórmula:

```
> brew info swi-prolog
```

provee información sobre otras opciones de instalación.

3.1.1 Uso en una términal

Una vez ejecutado el método de instalación correspondiente, hay varias formas de acceder a Prolog. Desde una terminal del sistema operativo, SWI-Prolog puede ejecutarse mediante el comando `swipl`:

```
> swipl
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.4.2)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

1 https://brew.sh/index_es.html
2 https://www.xquartz.org

Prolog se ejecuta en un ciclo REPL (Read-Eval-Print Loop), lo que significa que una vez ejecutado, el sistema está a la espera de una instrucción para evaluarla e imprimir el resultado. Eso se indica con el prompt `?-`. Asumiendo que existe el siguiente programa bajo el nombre `abuelo.pl`:

```
%%% padre(X,Y): X es padre de Y

padre(luis,ana).
padre(juan,luis).

%%% abuelo(X,Y): X es abuelo de Y

abuelo(X,Y) :-
  padre(X,Z), padre(Z,Y).
```

el programa puede cargarse en Prolog de la siguiente manera:

```
?- [abuelo].
true.
```

Una vez cargado un programa, se le pueden plantear metas al mismo. Por ejemplo:

```
?- padre(luis,ana).
true.

?- padre(luis,X).
X = ana.

?- abuelo(X,ana).
X = juan.
```

Si el programa se encuentra en otro directorio, puede ser cargado mediante una cadena de texto que indique el camino completo hasta el programa:

```
?- ["code/prolog/ia2/cap03/abuelo.pl"].
true.

?- abuelo(X,ana).
X = juan.
```

Para salir de Prolog, use el comando `halt`:

```
?- halt.
>
```

3.1.2 Uso desde la aplicación SWI-Prolog

En algunos sistemas operativos, SWI-Prolog instala una aplicación ejecutable que incluye un ambiente de desarrollo. En el caso de macOS, la aplicación se encuentra en la carpeta de aplicaciones, y una vez ejecutada abre una terminal propia, con un menú que permite trabajar con archivos Prolog, configurar el ambiente de desarrollo, ejecutar programas y acceder al depurador y ayudas del sistema (Ver Figura 3.1). De esta forma, además del comportamiento mostrado en la terminal del sistema operativo (sub sección anterior), es posible cargar un programa desde el menú `File ->Consult...`, etc.

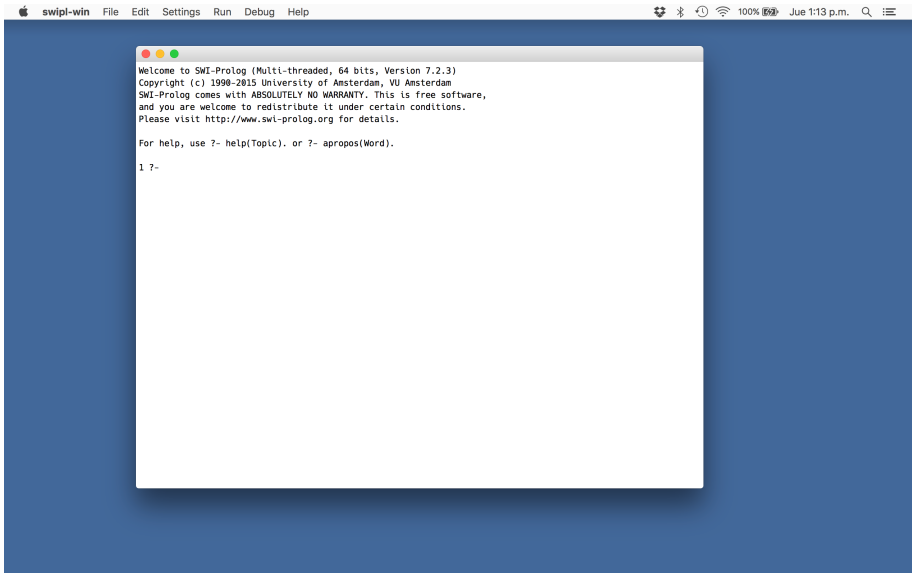

Figura 3.1: El ambiente provisto por la aplicación SWI-Prolog en OS X.

3.1.3 Uso desde Emacs

Nuestra forma favorita para usar SWI-Prolog es desde el editor Emacs. En el sistema operativo OS X puede usarse Aquamacs (`http://aquamacs.org`) con este fin. Esta versión de `emacs` viene configurada con un modo para trabajar con archivos Prolog. La Figura 3.2 muestra una ventada de Aquamacs con dos marcos, uno para editar el código de `abuelo.pl` y otro para hacer consultas al mismo.

3.2 TÉRMINOS Y CLÁUSULAS

El programa `abuelo.pl` se refiere a un universo de discurso cuyos miembros son `juan`, `luis` y `ana`. Estos= son los términos constantes del programa. Las relaciones entre ellos se definen en términos de cláusulas: `padre/2` se usa para definir con dos hechos (cláusulas sin cuerpo), quien es padre de quien. La cláusula `abuelo/2` se define como una regla (cláusula completa), en términos de la relación padre – Alguien (X) es abuelo de alguien más (Y), si tiene un hijo (Z) que es padre de ese alguien más (Y). Las consultas que hicimos para saber si luis era padre de ana, de quién es padre luis y quien es abuelo de ana, son metas (cláusulas sin cabeza). La figura 3.3 muestra los **términos** que podemos usar en Prolog.

Términos Prolog

3.2.1 Hechos

Los **hechos** expresan relaciones entre los objetos del universo de discurso, que son verdaderas incondicionalmente. Recuerden que son cláusulas con el cuerpo vacío. Por ejemplo, los siguientes hechos son utilizados para representar que ciertas ciudades están ubicadas en ciertos estados:

Hechos Prolog

```
1 %%% loc_en(X,Y)/2
2 %%% La ciudad X está localizada en el estado Y
3
4 loc_en(atlanta,georgia).
```

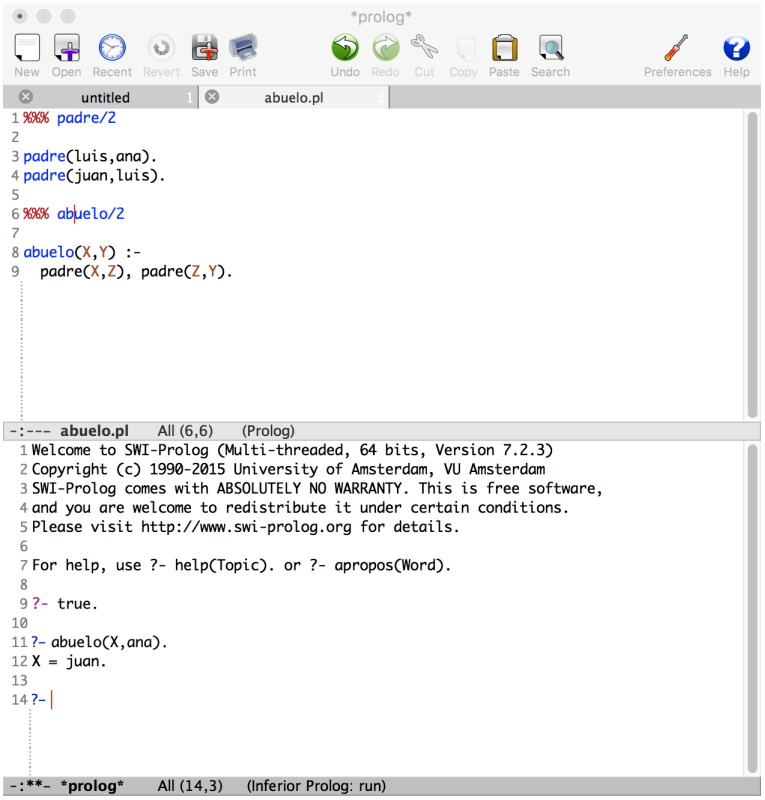

Figura 3.2: Prolog ejecutado desde Emacs en OS X.

```
loc_en(houston,texas).
loc_en(austin,texas).
loc_en(boston,massachussets).
loc_en(xalapa,veracruz).
loc_en(veracruz,veracruz).
```

3.2.2 Metas

Si cargamos en Prolog el programa `tutorialProlog01.pl`, que incluye estos hechos, podremos pedirle al sistema que resuelva metas (cláusulas sin cabeza) sobre estas ciudades y sus ubicaciones. Por ejemplo, si la ciudad de Xalapa se encuentra en Veracruz:

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

O si Boston, se encuentra en Veracruz:

```
?- loc_en(boston,veracruz).
false.
```

Observen que Prolog responde falso debido al **Supuesto del Mundo Cerrado** (*CWA* por sus siglas en inglés: *Closed World Assumption*) que implica que todo aquello que no sea derivable de un programa lógico es falso; y no a que el programa tenga representado el hecho de que Boston, no está en Veracruz (a esto se le llama a veces negación fuerte). *CWA*

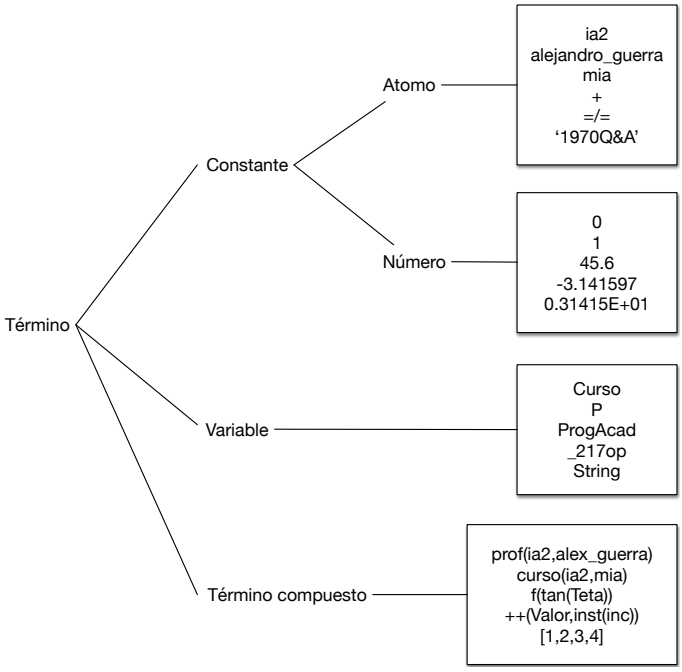

Figura 3.3: Los términos definidos en Prolog.

Si usamos variables, nuestras metas pueden ser más interesantes: ¿Qué ciudades se encuentran en Texas? Puede responderse con la siguiente meta. El punto y coma luego de la primer respuesta, le pide a Prolog que busque si hay más respuestas posibles:

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

3.2.3 Reglas

Las reglas son verdades condicionadas (cláusulas con cabeza y cuerpo). El siguiente conjunto de reglas expande la semántica de nuestro predicado `loc_en`:

```
%%% loc_en(X,Y)/2
%%% La ciudad X está localizada en el país Y

loc_en(X,usa) :- loc_en(X,georgia).
loc_en(X,usa) :- loc_en(X,texas).
loc_en(X,usa) :- loc_en(X,massachussets).

loc_en(X,mexico) :- loc_en(X,veracruz).

%%% loc_en(X,Y)/2
%%% La ciudad X está en norteamérica

loc_en(X,norteamerica) :- loc_en(X,usa).
loc_en(X,norteamerica) :- loc_en(X,usa).
loc_en(X,norteamerica) :- loc_en(X,mexico).
```

De forma que ahora podemos plantear metas más interesantes. Por ejemplo ¿Xalapa está en Estados Unidos?

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

¿Xalapa está en Norteamérica?

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

¿Qué ciudades se encuentran en México?

```
?- loc_en(Ciudad,mexico).
Ciudad = xalapa ;
Ciudad = veracruz.
```

¿Cual es la localización exacta de Xalapa?

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

Observen que el predicado `loc_en` ha sido utilizado para responder a todas estas preguntas.

3.2.4 Consultas negativas

Dada la semántica de las metas, las consultas que involucran negaciones pueden resultar curiosas. Si la meta en cuestión es de base, no hay problema alguno. Por ejemplo, para saber si Xalapa no está en los Estados Unidos, puedo usar la siguiente meta:

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

Pero, siguiendo el estilo de esta meta, si quiero saber qué ciudades no están en Estados Unidos, podría intentar algo como:

```
?- \+ loc_en(Ciudad,usa).
false.
```

Cuyo resultado no es el esperado. Esto se debe a que tal meta en realidad pregunta si no hay ciudades en Estados Unidos, a lo que Prolog responde con falso, puesto que las hay. La siguiente meta funciona mejor para este caso:

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

Esta meta tiene la siguiente interpretación: ¿Qué ciudades localizadas en Norte América, no se encuentran en los Estados Unidos? Observen que al procesar la negación, la variable *Ciudad* siempre está instanciada.

3.2.5 Términos compuestos como estructuras de datos

Es posible usar los términos compuestos como estructuras de datos. Por las cláusulas `horizontal` y `vertical`:

```
horizontal(seg(punto(X,Y),
               punto(X1,Y))).

vertical(seg(punto(X,Y),
             punto(X,Y1))).
```

usamos el término compuesto `punto/2` para representar datos sobre puntos en un espacio 2D; y `seg/2` para representar segmentos de línea entre dos puntos. De forma que podemos hacer consultas como:

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

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

La segunda meta es interesante, ¿Cómo debe ser el segundo punto del segmento para que éste sea horizontal? Cualquier punto con coordenada Y=2, satisface esta meta. Haremos uso de estas facilidades más adelante, para trabajar con árboles binarios.

3.2.6 Aritmética

Las computadoras suelen usarse para calcular operaciones aritméticas y comparar valores numéricos. Prolog provee operadores para ello. Consideren los operadores de **comparación** que se muestran en el Cuadro 3.1. *Comparar*

Operación	Semántica
X =:= Y	X e Y son el mismo número.
X =/= Y	X e Y son diferentes números.
X < Y	X es menor que Y.
X > Y	X es mayor que Y.
X =< Y	X es menor o igual que Y.
X >= Y	X es mayor o igual que Y.

Cuadro 3.1: Operadores aritméticos de comparación.

Observen que el operador menor o igual está invertido, esto es porque la expresión <= está reservada para otros usos en Prolog.

Prolog provee un operador para la **evaluación** de expresiones aritméticas. Propiamente hablando, el operador infijo *is* tiene como segundo argumento un término que se interpreta como una expresión aritmética, que una vez evaluada de acuerdo a las reglas de la aritmética, se unificará con el primer argumento. EL Cuadro 3.2 muestra los operadores aritméticos básicos de Prolog. *Evaluar*

Operación	Semántica
X + Y	la suma de X e Y.
X − Y	la resta de X e Y.
X * Y	la multiplicación de X e Y.
X / Y	la división de X e Y.
X // Y	el cociente entero de X entre Y.
X mod Y	el resto de dividir X entre Y.

Cuadro 3.2: Operadores aritméticos de Prolog.

Por ejemplo, la siguiente cláusula define `factorial`:

```
/* factorial */

```

```
fact(0,1).
fact(N,F) :- N>0, N1 is N-1, fact(N1,F1), F is N* F1.
```

i

Esta definición puede usare para computar el factorial de un número, cinco por ejemplo:

```
?- fact(5,R).
R = 120
```

pero no para computar que número tiene como factorial 120:

```
?- fact(X,120).
ERROR: >/2: Arguments are not sufficiently instantiated
```

esto se debe a que *is* solo evalúa su operador derecho, esto es, no se trata de un operador que resuelva ecuaciones.

A continuación definimos los conceptos de par e impar haciendo uso de los operadores vistos:

```
par(X) :- 0 is X mod 2.

impar(X) :- 1 is X mod 2.

%%% impar con negación

nimpar(X) :- \+ par(X).
```

3.3 LISTAS

Listas

Prolog provee una sintaxis particular para manejar listas. Una **lista** vacía es una lista y se denota por la constante especial `[]`. La expresión `[X|Xs]` es una lista, si `Xs` es una lista. Se dice que `X` es la cabeza de la lista y `Xs` su cola. La definición de esta estructura de datos es recursiva. Observen la siguiente unificación:

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

La expresión de la izquierda nos es más familiar, pero solo se trata de una abreviatura de la de la derecha [3]. Lo que es interesante de la notación que usa la barra es que nos permite acceder a elementos de una lista vía unificación. Consideren los siguientes ejemplos:

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

?- [X1,X2|Xs] = [1,2,3].
X1 = 1,
X2 = 2,
Xs = [3].

?- [X|Xs] = [1].
X = 1,
Xs = [].

?- [X|Xs] = [].
false.
```

3 Que a su vez es una abreviatura de una expresión más fea: `'[|]'(1, '[|]'(2,...))`. El operador de concatenación solía ser el punto, de forma que la expresión anterior solía ser `.(1,.(2,.(3,[])))`, pero esto cambió en la versión 7 de SWI-Prolog para poderle dar al punto otros usos, como en `alumno.apellido`.

Hay dos formas de usar cadenas de texto en SWI-Prolog. Si la cadena está delimitada por dos acentos invertidos, ésta será interpretada como una lista de códigos ascii. Por ejemplo:

```
?- L = `hola`.
L = [104, 111, 108, 97].
```

Este suele ser el comportamiento esperado de las cadenas de texto en Prolog y se suelen delimitar por comillas. Sin embargo, a partir de la versión 7 de SWI-Prolog, las cadenas de texto delimitadas por comillas crean términos de la clase `String`, de forma que la unificación anterior tiene un efecto diferente:

```
?- L = "hola".
L = "hola".
```

3.3.1 Recorriendo listas

Las listas suelen usarse para guardar colecciones de datos. Un patrón común al usar listas es recorrerlas para ver si los elementos de la lista cumplen con cierta propiedad. Por ejemplo:

```
%%% gringas/1
%%% Las ciudades en la lista X son todas gringas

gringas([X]) :- loc_en(X,usa).
gringas([X|Xs]) :- loc_en(X,usa), gringas(Xs).
```

define una cláusula que verifica si todas las ciudades de una lista están localizadas en Estados Unidos. Observen la definición recursiva de esta cláusula. El caso que detiene la recursividad es cuando la lista analizada tiene un elemento. El resultado depende de si ese elemento está en Estados Unidos o no. El caso recursivo es cuando la lista tiene más de un elemento: Se debe verificar si el primer elemento de la lista cumple con la propiedad deseada (estar en Estados Unidos) y en ese caso hacer la llamada recursiva sobre el resto de la lista; Si ese no es el caso, la cláusula falla. Las consultas que se pueden hacer incluyen:

```
?- gringas([boston]).
true

?- gringas([boston,atlanta,xalapa]).
false.

?- gringas(L).
L = [atlanta] ;
L = [houston] ;
L = [austin] ;
L = [boston] ;
L = [atlanta, atlanta] ;
L = [atlanta, houston]

?- gringas([]).
false.
```

Intente entender la razón de las dos últimas respuestas.

Otro recorrido clásico es el que define a una cláusula que busca si un elemento es miembro de una lista:

```
%%% miembro(X,Ys)
%%% El elemento X es miembro de la lista Ys

miembro(X,[X|_]).
miembro(X,[_|Ys]) :- miembro(X,Ys).
```

En esta definición recursiva, el caso terminal es cuando el elemento que buscamos es el primer elemento de la lista; si ese no es el caso, debe de ser que el elemento es miembro del resto de la lista. En cualquier otro caso, la búsqueda falla. Algunas consultas incluyen:

```
?- miembro(1,[]).
false.

?- miembro(1,[1]).
true

?- miembro(1,[3,2,1]).
true

?- miembro(X,[1,2]).
X = 1 ;
X = 2 ;
false.

?- miembro(1,L).
L = [1|_G1534] ;
L = [_G1533, 1|_G1537]
```

3.3.2 Mapeos de listas a enteros

Otro patrón de procesamiento de listas muy común es recorrer una lista para computar un valor entero. Por ejemplo, calcular la longitud de una lista (cuantos miembros tiene):

```
%%% long(Xs,L)
%%% L es la longitud de la lista Xs

long([],0).
long([_|Xs],L) :- long(Xs,L1), L is L1+1.
```

El caso terminal de esta definición es cuando la lista está vacía en cuyo caso tiene longitud 0. Si la lista no es una lista vacía, su longitud es 1 más la longitud del resto de la lista. Las posibles consultas incluyen:

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

?- long([],L).
L = 0.

?- long(L,2).
L = [_G199193, _G199196]
```

El modo trace

Un paréntesis necesario ¿Han entendido las llamadas recursivas? SWI-Prolog provee un modelo para la **depuración** [4] de nuestros programas, que incluye una herramienta para trazar la ejecución de los mismos. Esta basado en las derivaciones que Prolog hace para tratar se satisfacer una meta. *Depuración*

El cómputo no determinista de Prolog y su mecanismo de *backtracking*, complican un poco la historia de depurar un programa lógico. En los lenguajes de programación tradicionales, el punto de interés a revisar un programa es la entrada y salida

4 El modelo se debe a Lawrence Byrd, de hecho se le conoce como *Byrd Box Model*. La descripción que presentamos está basada en el texto de Clocksin y Melish [18], sección 8.3.

de una función; en Prolog es necesario saber cuando y porque se está entrando o saliendo de una función. Para ello, la descripción se hace en términos de cuatro clases de **eventos**:

Eventos del depurador

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.

De esta forma podemos saber que evento está sucediendo sobre que meta, en todo momento. Para diferenciar las metas, estas reciben un identificador numérico único, su **número de invocación**. A continuación se ejemplifica el uso de la traza con nuestra cláusula `long/2`:

Número de invocación

```
?- trace.
[trace]  ?- long([1,2],L).
   Call: (7) long([1, 2], _G309) ?
   Call: (8) long([2], _L350) ?
   Call: (9) long([], _L369) ?
   Exit: (9) long([], 0) ?
   ^  Call: (9) _L350 is 0+1 ?
   ^  Exit: (9) 1 is 0+1 ?
   Exit: (8) long([2], 1) ?
   ^  Call: (8) _G309 is 1+1 ?
   ^  Exit: (8) 2 is 1+1 ?
   Exit: (7) long([1, 2], 2) ?
L = 2
```

Observen como las diferentes llamadas a `long/2` están diferenciadas por el número de invocación. Para cerrar el modo de traza y depuración, hacer lo siguiente:

```
[trace] ?- notrace.
[debug] ?- nodebug.
?-
```

3.3.3 Recursividad a la cola

Observen en la traza de `long2` que la pila de llamadas crece hasta alcanzar un tamaño igual a la longitud de la lista cuya longitud queremos medir. Esto se debe a que la operación que suma 1 a la longitud del resto de la lista, para resolver nuestra meta, queda pendiente hasta satisfacer la llamada a la longitud del resto de la lista. Esto se puede evitar usando una técnica llamada **recursividad a la cola**. Se trata de no dejar operaciones pendientes luego de hacer una llamada recursiva, con ayuda de una variable auxiliar donde vayamos guardando el resultado de interés, para cada llamada a una submeta. Esta variable se conoce como **acumulador**. A continuación se lista una versión recursiva a la cola de longitud, llamada `longTR`:

Recursividad a la cola

Acumulador

```
47 longTR(Xs,L) :- long(Xs,0,L).
48
49 long([],Acc,Acc).
50 long([_|Xs],Acc,L) :- Acc1 is Acc + 1, long(Xs,Acc1,L).
```

Observen primero que `longTR/2` es meramente una interfaz para llamar a `long/3`, es el realmente la cláusula recursiva a la cola. El caso terminal, cuando el primer argumento es una lista vacía, regresa el valor del acumulador *Acc*. Como la llamada inicial se hace con el acumulador igual a cero (línea 1), si el `longTR` es llamada con la lista vacía, el resultado será cero:

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

Lo interesante ocurre tras bambalinas. Observen la traza de esta meta:

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

El tamaño de la pila de ejecución es siempre $O(1)$, mientras que en la versión no a la cola era de $O(n)$. Para comprender la relevancia de este resultado usaremos la siguiente cláusula para generar listas de tamaño n:

```
%%% creaLista(N,L)
%%% crea una lista L de N elementos (para descreídos)

creaLista(0,[]).
creaLista(N,[N|Ns]) :- N1 is N-1, creaLista(N1,Ns).
```

De forma que si queremos saber cuantos recursos (tiempo y espacio) usan ambas versiones, `long` y `longTR`, para computar una lista de cinco mil elementos, ejecutamos las siguientes metas:

```
?- creaLista(5000,Lista), time(long(Lista,Long)).
% 10,002 inferences, 0.001 CPU in 0.002 secs (79% CPU, 7583017 Lips)
Lista = [5000, 4999, 4998, 4997, 4996, 4995, 4994, 4993, 4992|...],
Long = 5000

?- creaLista(5000,Lista), time(longTR(Lista,Long)).
% 5,002 inferences, 0.000 CPU in 0.000 secs (98% CPU, 12381188 Lips)
Lista = [5000, 4999, 4998, 4997, 4996, 4995, 4994, 4993, 4992|...],
Long = 5000
```

No olviden apagar el modo de traza y depuración antes de ejecutar estas metas. Pueden abortar la traza tecleando `a`. El Cuadro 3.3 resume estos datos para diferentes longitudes de lista.

Aunque los números exactos pueden variar, dependiendo de la cantidad de memoria asignada a Prolog, observen como `long` no puede procesar una lista de un millón de elementos, mientras que `longTR` lo hace sin problemas. De hecho, en mi nuevo equipo puedo procesar 10 millones de elementos con `longTR` en 0.448 segundos; mientras que `long` solo puede procesar hasta aproximadamente 2750000 elementos. Mismo equipo.

Podemos usar recursividad a la cola para redefinir nuestra definición de factorial, `fact`, que presenta los mismos problemas que `long` al no ser recursiva a la cola.

Método	N	No. Infs	CPU %	t(segs)	Lips
long	1000	2001	0.00	0.00	200100
longTR	1000	2002	0.00	0.00	Infinite
long	10000	20001	0.02	0.02	1000050
longTR	10000	20002	0.01	0.02	2000200
long	100000	200001	0.17	0.22	1176476
longTR	100000	200202	0.14	0.16	1428586
long	1000000	348726	0.29	1.85	**out stack**
longTR	1000000	2000002	2.09	2.31	956939

Cuadro 3.3: Estadísticas sobre ejecución de long y longTR sobre listas de diferente longitud.

```
6  factTR(X,F) :-
7          fact(X,1,F).
8  fact(0,Acc,Acc).
9  fact(N,Acc,R) :-
10         N>0,
11         N1 is N-1,
12         Acc1 is Acc*N,
13         fact(N1,Acc1,R).
```

3.3.4 Listas a partir de valores

También es posible generar listas a partir de ciertos valores. Por ejemplo, la cláusula `intervalo` genera una lista de valores comprendidos entre un valor inicial y uno final:

```
40 /* listas a partir de valores */
41
42 intervalo(X,X,[X]).
43 intervalo(X,Y,[X|Xs]) :- X < Y, Z is X + 1, intervalo(Z,Y,Xs).
```

Esta cláusula puede usarse dela siguiente manera:

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

3.3.5 Lista a partir de listas (filtrado)

También es posible filtrar una lista para obtener otra. Por ejemplo, la cláusula `filtra/2` recibe una lista de enteros y regresa una lista que solo incluye los elementos pares de su primer argumento.

```
47 pares([],[]).
48 pares([X|Xs],[X|Ys]) :- par(X), pares(Xs,Ys).
49 pares([X|Xs],Ys) :- impar(X), pares(Xs,Ys).
```

Observe que este patrón de procesamiento requiere tres casos: Uno de terminación, el caso donde el elemento debe ser guardado (es par) y el caso cuando no (no es par). Para probar su uso, recurriremos a `intervalo/3`. La siguiente meta obtiene los pares comprendidos entre 1 y 5:

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

3.3.6 Operaciones sobre listas

La siguiente cláusula permite unir dos listas en una tercera:

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

Su funcionamiento es como sigue:

```
?- append([1,2,3],[4,5,6],L).
L = [1, 2, 3, 4, 5, 6].
?- append(X,Y,[1,2,3]).
X = [],
Y = [1, 2, 3] ;
X = [1],
Y = [2, 3] ;
X = [1, 2],
Y = [3] ;
X = [1, 2, 3],
Y = [] ;
false.
```

Una vez que hemos definido `append`, es muy sencillo definir cláusulas para computar si una lista es sufijo, prefijo o sublista de otra:

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

Las llamadas a estas cláusulas pueden ser como sigue:

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

A continuación, dos versiones para obtener el reverso de una lista. Una sigue una estrategia recursiva simple y la otra a la cola:

```
reverso([],[]).
reverso([X|Xs],Res):-
  reverso(Xs,XsReverso),
  append(XsReverso,[X],Res).

reversoTR(L,Res):-
  nonvar(L),
  reversoTRaux(L,[],Res).

reversoTRaux([],Acc,Acc).
reversoTRaux([X|Xs],Acc,Res) :-
  reversoTRaux(Xs,[X|Acc],Res).
```

3.3.7 Unificación con listas

Prolog provee operadores para computar la unificación entre dos términos. El operador = verifica si la unificación se satisface; el operador \= se satisface cuando éste no es el caso. Veamos algunos ejemplos:

```
?- f(X,X) = f(a,Y).
X = a
Y = a.
?- f(X,X) \= f(a,Y).
false.
?- p(f(X),g(Z,X)) = p(Y, g(Y,a)).
X = a
Z = f(a)
Y = f(a).
?- p(X,X) = p(Y,f(Y)).
X = f(**)
Y = f(**).
```

SWI-Prolog no implementa chequeo de ocurrencias, por lo que la última unificación genera un ciclo infinito, indicado por $**$.

Es posible acceder a los componente de un término compuesto mediante unificación, gracias al operador `=..`. A continuación un ejemplo de su uso:

```
?- papa(X,juan) =.. L.
L = [papa, X, juan].
```

Observen que L ha sido unificada con una lista que incluye los componentes de $papa(X, juan)$.

3.3.8 Ordenando listas

Una operación muy común es ordenar una lista. En esta sección revisaremos varios algoritmos de ordenamiento. Comencemos por definir una cláusula que es verdadera si su argumento es una lista ordenada:

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

Observen el patrón en la definición de la línea tres: Se recuperan los dos primeros elementos de la lista porque será necesario compararlos; pero la llamada recursiva se hace sobre toda la lista, menos el primer elemento. De forma que:

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

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

?- ordenada([]).
true.
```

El orden ascendente se debe a la primer sub-meta de la línea tres. Si usamos mayor en lugar a menor, el orden a verificar sería descendente. Definamos ahora una función para insertar un elemento en una lista ordenada y mantener el orden en ella:

```
inserta(X,[],[X]).
inserta(X,[Y|Ys],[X,Y|Ys]) :- X < Y.
inserta(X,[Y|Ys],[Y|Zs]) :- X >= Y, inserta(X,Ys,Zs).
```

Observen que esta cláusula pasa por tres casos: Insertar un elemento en una lista vacía; insertar un elemento en una lista cuyo primer elemento es mayor que el elemento; e insertar un elemento en una lista cuando este no es el caso. Veamos algunas llamadas a la cláusula `inserta`:

```
?- inserta(1,[],L).
L = [1]

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

Por supuesto que se asume que la lista donde vamos a insertar el elemento está en orden ascendente, vean lo que ocurre cuando este no es el caso:

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

La meta tiene éxito, aunque su semántica no es la que el programador pretendía. Por ello es necesario hacer pruebas exhaustivas sobre las posibles llamadas a una cláusula al momento de definirla. Si se deja la verificación para más adelante, la depuración de nuestros programas se volverá rápidamente intratable. Sería conveniente comentar la definición de `inserta` para que quede claro lo que se espera de su segundo argumento.

Ordenamiento por inserción

El primer algoritmo de ordenamiento que definiremos hace uso de las definición de `inserta`, si quieres ordenar una lista simplemente inserta en orden sus elementos en otra:

```
ordenaIns([],[]).
ordenaIns([X|Xs],Ys) :- ordenaIns(Xs,Xs0), inserta(X,Xs0,Ys).
```

De forma que:

```
?- ordenaIns([],L).
L = [].

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

Observen que el peor desempeño de este algoritmo será cuando la lista a ordenar esté en orden inverso. Observan también que esta definición no es recursiva a la cola ¿Se les ocurre como hacer que lo sea?

Método de burbuja

Veamos la definición del algoritmo de ordenamiento por burbujas:

```
bubbleSort(L,S):- swap(L,L1),!,
%        write(L1),nl,
        bubbleSort(L1,S).
bubbleSort(S,S).

swap([X,Y|Ys],[Y,X|Ys]):- X>Y.
swap([Z|Zs],[Z|Zs1]):- swap(Zs,Zs1).
```

Observen el uso de `write` para observar el comportamiento del programa: Gracias a esa meta, veremos las burbujas flotar:

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

Observen que el caso terminal está definido después de la cláusula recursiva ¿Qué sucede si invierten ese orden?

3.4 ARBOLES BINARIOS COMO TÉRMINOS COMPUESTOS

Un árbol binario es un nodo vacío, o un nodo con dos hijos que son árboles, tal y como se muestra en la figura 3.4. Esto puede representarse con términos: La constante `vacio` representa un árbol binario vacío y el término compuesto `arbol/3` un árbol no vacío, donde el primer elemento es el valor que almacena en árbol y los otros dos son sus hijos izquierdo y derecho (árboles también). A continuación revisaremos algunas operaciones comunes sobre esta representación.

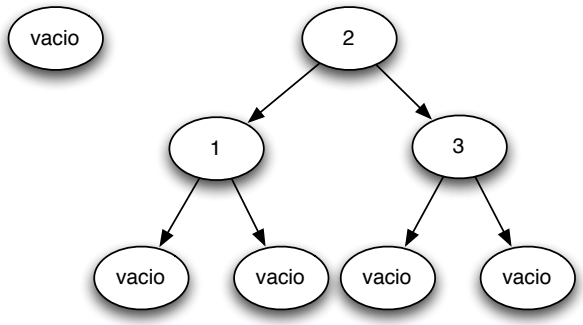

Figura 3.4: Arboles binarios: vacío y con dos hijos.

3.4.1 Inserción de un elemento en un árbol

La inserción de un elemento en un árbol binario se define como sigue:

```
1   insertaArbol(X,vacio,arbol(X,vacio,vacio)).
2
3   insertaArbol(X,arbol(X,A1,A2),arbol(X,A1,A2)).
4
5   insertaArbol(X,arbol(Y,A1,A2),arbol(Y,A1N,A2)) :-
6           X<Y, insertaArbol(X,A1,A1N).
7   insertaArbol(X,arbol(Y,A1,A2),arbol(Y,A1,A2N)) :-
8           X>Y, insertaArbol(X,A2,A2N).
```

De forma que la consulta siguiente, ejemplifica el resultado de insertar 1 en un árbol vacío y, posteriormente, 2 en el árbol resultante:

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

3.4.2 Conversión de lista a árbol

Sería de gran utilidad un predicado que convirtiera una lista a un árbol y así no tener que escribir `arbol(... arbol(...` cada vez que lo necesitemos. La definición de ese preciado es como sigue:

```
1   creaArbol([],A,A).
2   creaArbol([X|Xs],AAux,A) :- insertaArbol(X,AAux,A2),
3           creaArbol(Xs,A2,A).
4
5   lista2arbol(Xs,A) :- creaArbol(Xs,vacio,A).
```

Su uso se ilustra a continuación:

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

3.4.3 Recorrido del árbol

El siguiente procedimiento colecta los nodos del árbol en una visita en orden.

```
nodos(vacio,[]).
nodos(arbol(X,A1,A2),Xs) :-
  nodos(A1,Xs1),
  nodos(A2,Xs2),
  append(Xs1,[X|Xs2],Xs).
```

La siguiente llamada ilustra su uso:

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

3.4.4 Ordenamiento de una lista por inserción en árbol binario

Todo lo anterior puede ser usado para ordenar una lista mediante la inserción de sus elementos en un árbol binario y el recorrido en orden de los nodos de éste:

```
ordenaLista(L1,L2) :-
  lista2arbol(L1,A),
  nodos(A,L2).
```

Su uso es como sigue:

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

3.5 OPERACIONES DE ENTRADA Y SALIDA

Prolog provee una serie de funciones de entrada y salida en consola y archivos. A continuación revisaremos ambos. También es posible definir interfaces gráficas con SWI-Prolog, usando la librería XPCE, pero esto queda fuera del alcance de este tutorial.

3.5.1 Términos de salida

Flujo

Prolog provee la función `write/1` para escribir términos en el **flujo** de salida. Por lo tanto, el argumento de esta función debe ser un término. Por defecto, el flujo de salida es el monitor, aunque como veremos, éste puede cambiarse a otro dispositivo o archivo. También hemos utilizado la función `nl/0` que escribe un salto de línea en el flujo de salida activo. A continuación algunos ejemplos del uso de estas funciones:

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

```
Maestría en
IA
true.
```

Si es necesario conservar las comillas, se puede usar la función `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.
```

Observen que estas funciones tienen efectos colaterales, además de satisfacerse, tienen un efecto en el flujo de salida. En el mismo sentido, si se encuentra un problema al acceder al flujo de salida, estas funciones no se satisfacen, independientemente de su sentido lógico.

3.5.2 Términos de entrada

La función `read/1` permite la captura de términos de entrada. Su único argumento debe ser una variable. La evaluación de esta función tiene el efecto de intentar unificar la variable en cuestión, con el término capturado desde el flujo de entrada. El flujo de entrada por defecto es el teclado. De manera análogo a las funciones de salida, el flujo de entrada puede cambiarse a otros dispositivos o archivos. La captura del término desde el teclado termina cuando escribimos un punto y un carácter en blanco (espacio, salto de línea, etc.). Mientas estamos capturando el término, el *prompt* de Prolog cambia como se puede ver a continuación:

```
?- read(X).
|: juan.
X = juan.
?- read(X).
|: mi_predicado_tonto(b).
X = mi_predicado_tonto(b).
?- read(Y).
|: 'alejandro guerra'.
Y = 'alejandro guerra'.
```

Recuerden que esta función de salida trata de unificar la variable que pasamos como su argumento, contra el término capturado desde el teclado. Si esto no fuese posible, la función falla:

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

3.5.3 Entradas y salidas con caracteres

En algunas ocasiones es conveniente bajar de nivel en las operaciones de entrada y salida para trabajar con caracteres en lugar de términos. Para ello podemos hacer uso del código ASCII o, si nuestra implementación de Prolog lo permite, con alguna variante de Unicode. En estos casos la función `put/1` permite escribir caracteres en el flujo de salida. Su argumento debe ser un número (código ASCII) entre 0 y 255:

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

La captura de caracteres desde el teclado puede hacerse con las funciones `get/1` y `get0/1`. La segunda puede aceptar caracteres en blanco, mientras que la primera no. Veamos algunos ejemplos:

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

La siguiente función me permite capturar caracteres, hasta que se introduzca el asterisco. Posteriormente, los códigos de los caracteres son listados:

```
readin :- get0(X), process(X).
process(42).
process(X) :- X =\= 42, write(X), nl, readin.
```

Un ejemplo de su ejecución sería:

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

Este patrón general de procesamiento es bastante común la procesar entradas. A continuación otro ejemplo, en este caso se trata de una función que cuenta los caracteres capturados:

```
cuenta(Total) :- cuenta_aux(0,Total).
cuenta_aux(Acc,Res) :-
  get0(X), process(X,Acc,Res).

process(42,Acc,Acc).
process(X,Acc,Res) :-
  X=\=42, Acc1 is Acc+1,
  cuenta_aux(Acc1,Res).
```

Su ejecución es como sigue:

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

Otro ejemplo un poco más elaborad, una función que cuenta vocales en los caracteres capturados:

```
cuenta_vocales(Total) :- cuenta_vocales_aux(0,Total).
cuenta_vocales_aux(Acc,Res) :-
  get0(X), process_vocales(X,Acc,Res).

process_vocales(42,Acc,Acc).
process_vocales(X,Acc,Res) :-
  X=\=42, processChar(X,Acc,N),
  cuenta_vocales_aux(N,Res).

processChar(X,Acc,N) :- vocal(X), N is Acc+1.
processChar(_,Acc,Acc).

vocal(65). %%% Mayúsculas
vocal(69). vocal(73).
vocal(79). vocal(85).
vocal(97). %%% Minúsculas
vocal(101). vocal(105).
vocal(111). vocal(117).
```

Y su ejecución:

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

3.5.4 Archivos

Tanto dispositivos como archivos son flujos para Prolog. De forma que si queremos leer o escribir en un archivo, simplemente hay que cambiar de flujo. Para cambiar el flujo de salida se usa la función `tell/1` cuyo argumento es una variable o un átomo que representa el nombre del archivo donde queremos escribir. El **flujo de salida estándar** es `user`. El predicado `told/0` cierra el flujo de salida actual, regresando al flujo de salida anterior. El predicado `telling/1` unifica su argumento, que debe ser una variable, con el nombre del flujo de salida actual.

Flujo de Salida Estándar

Tenemos operadores análogos para las entradas. El predicado `see/1` tiene como argumento una variable o un átomo que representa el nombre del archivo que queremos leer. Al evaluar este predicado satisfactoriamente, el archivo en cuestión se vuelve el flujo de entrada activo. Si no es posible abrir el archivo, el predicado falla. El **flujo de entrada estándar** también se llama `user`. Para cerrar el flujo de entrada activo se usa `seen/0` y `seeing/1` unifica su argumento, normalmente una variable, con el identificador del flujo de entrada activo.

Flujo de Entrada Estándar

El siguiente programa cuenta las vocales de un archivo dado:

```
  vocs_file(Arch,Total) :-
          see(Arch),
          cuenta_vocales_file(0,Total),
          seen.

  cuenta_vocales_file(Acc,Res) :-
          get0(X), process_vocales_file(X,Acc,Res).

  process_vocales_file(-1,Acc,Acc).
  process_vocales_file(X,Acc,Res) :-
          X =\= -1, processChar(X,Acc,N),
          cuenta_vocales_file(N,Res).
```

Un ejemplo de su ejecución es como sigue:

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

Para terminar, el siguiente programa copia una secuencia de caracteres capturados que termina con signo de admiración en el archivo indicado en su argumento:

```
copiaCars(Salida):-
  telling(Actual), tell(Salida),
  copiaCars_aux, told, tell(Actual).

copiaCars_aux :- get0(C), copia(C).

copia(33).
copia(C) :-
  C =\= 33, put(C), copiaCars_aux.
```

3.5.5 Trabajando con archivos CSV

Ejemplificaré la lectura de archivos con un caso de minería de datos. Es común utilizar Weka [97] para estas tareas, donde los conjuntos de entrenamiento se guardan en un formato conocido como ARFF. Es posible exportar estos archivos a un formato separado por comas (CSV) que podemos leer fácilmente en Prolog. Por ejemplo, el conocido conjunto de entrenamiento tenis.csv quedaría almacenado en un archivo de texto como:

```
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
```

La línea uno del archivo CSV incluye el nombre de los atributos del problema, la clase incluida. El resto de las líneas son ejemplos de entrenamiento, es decir, valores específicos para esos atributos y clase. Observen que al convertir el archivo ARFF a CSV hemos perdido información sobre el dominio de los atributos, que deberemos recuperar más adelante.

Conociendo la estructura de este archivo, es posible definir un procedimiento para leer estos datos y convertirlos a un formato amigable con Prolog, por ejemplo, listas:

```
  % load_exs(CSV_file,Atts,Exs): read the CSV to obtain the list of
  % attributes Atts for the learning setting, and the list of training
  % examples Exs.

  load_exs(CSV_file,Atts,Exs) :-
    csv_read_file(CSV_file,[AttsAux|ExsAux], [strip(true)]),
    AttsAux =.. [_|Atts],
    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].
16  \end{minipageted}
17
18  La idea de convertir el archivo ARFF a CSV no es fortuita. Prolog, provee el
19  predicado \texttt{csv\_read\_file} para leer este tipo de archivos. Su argumento
20  \texttt{strip(true)} hace que los espacios en blanco no se tomen en cuenta al
21  leer el archivo. La siguiente consulta ilustra su uso directamente sobre
22  \texttt{tenis.csv}:
23
24  \begin{minted}{logtalk}
25    ?- csv_read_file('tenis.csv',Lines,[strip(true)]).
26    Lines = [row(outlook, temperature, humidity, wind, class), row(sunny,
27    hot, high, weak, no), row(sunny, hot, high, strong, no), row(overcast,
28    hot, high, weak, yes), row(rain, mild, high, weak, yes), row(rain,
29    cool, normal, weak, yes), row(rain, cool, normal, strong, no),
30    row(overcast, cool, normal, strong, yes), row(...)|...].
```

Observen que la salida es una lista de predicados `row`, donde cada uno de ellos tiene como argumentos los valores de cada línea del archivo. La idea es entonces quedarse solo con una lista de estos valores tanto para los atributos como para los ejemplos. Para ello, los predicados `row` se parten en dos listas: la cabeza se corresponde con los atributos *AttsAux* y el resto con los ejemplos *ExsAux*. Obtener la lista de atributos es muy sencillo, usando el operador `=..` en la línea 7, de forma que *Atts* guarda ahora la lista de los nombres atributos del conjunto de entrenamiento. Algo similar debe hacerse con todos los ejemplos, por lo que definimos un procedimiento `proc_ex` (línea 14) y lo aplicamos a todos los miembros de *ExsAux* con la ayuda del predicado `maplist`, para obtener los ejemplos como listas de valores *Exs*. Un consulta al archivo CSV se realiza como sigue:

```
1  ?- load_exs('tenis.csv',Atts,Exs).
2  14 training examples loaded.
3  Atts = [outlook, temperature, humidity, wind, class],
4  Exs = [[sunny, hot, high, weak, no], [sunny, hot, high, strong, no],
5         [overcast, hot, high, weak, yes], [rain, mild, high, weak, yes],
6         [rain, cool, normal, weak|...], [rain, cool, normal|...],
7         [overcast, cool|...], [sunny|...], [...|...]|...].
```

De forma que tenemos los atributos y los ejemplos almacenados como listas. Ahora podemos procesar la información que perdimos al pasar de ARFF a CSV, es decir, computar los dominios de cada atributo. Básicamente el dominio de un atributo es el conjunto de valores que aparece en su columna de valores. El siguiente procedimiento implementa esta definición casi directamente:

```
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],
6          (member(Att,Atts),
7           nth0(Idx,Atts,Att),
8           column_exs(Idx,Exs,Vals),
9           list_to_set(Vals,Dom)),
10   Doms).
11
12 % column(N,Exs,Vals): returns the values Vals for the Nth attribute in
```

```
% the examples Exs.

column_exs(N,Exs,Vals) :-
  maplist(nth0(N),Exs,Vals).
```

El predicado auxiliar `column_exs/3` obtiene una lista de los valores *Vals* de la *N*-ésima columna en los ejemplos *Exs*. El predicado principal `domain_atts/3` obtiene los dominios buscando el conjunto correspondiente (`list_to_set` a la columna (`column_exs`) de cada atributo *Att* en *Atts*. Su uso se ilustra a continuación:

```
?- 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]]].
```

3.6 LECTURAS Y EJERCICIOS SUGERIDOS

Existen numerosos libros de introducción a Prolog. Muchos de los ejemplos utilizados en este capítulo están tomados de dos referencias: el libro de Bratko [9], que además de introducir al lenguaje, ejemplifica su uso en numerosos problemas de la IA y el libro de Sterling y Shapiro [88], otra excelente introducción al lenguaje. Clocksin y Melish [18] proveen una buena introducción, más breve que las dos anteriores. Clocksin [17] complementa el libro anterior con una revisión de técnicas avanzadas y casos de estudio dirigidos al programador que necesita usar Prolog rápidamente. Covington, Nute y Avellino [24] hacen también una revisión en profundidad de los usos de Prolog en IA. Una introducción más reciente, en mi opinión inferior a las anteriores, se encuentra en el libro de Bramer [7]. Otra revisión reciente de Prolog y sus aplicaciones es el libro de Stobo [89]. Wielemaker y col. [96] presentan una panorámica interesante del desarrollo de SWI-Prolog y las tecnologías que ha adoptado. Covington y col. [25] nos ofrecen una guía de estilo para la programación en Prolog.

Ejercicios

Ejercicio 3.1. *Implemente en Prolog las siguientes operaciones sobre conjuntos representados como listas:*

- *Subconjunto:*

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

- *Intersección:*

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

- *Unión:*

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

- *Diferencia:*

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

Ejercicio 3.2. *Escriba un programa que tome una lista de números como primer argumento y regrese una lista de los mismos números incrementados en 1. Por ejemplo:*

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

Ejercicio 3.3. *Escriba un programa que regrese en su segundo argumento la lista de todas las permutaciones de la lista que es su primer argumento. Por ejemplo:*

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

Ejercicio 3.4. *Escriba un programa en que elimine todas las ocurrencias de un elemento (primer argumento) en una lista (segundo argumento); y regrese la lista filtrada como tercer argumento. Por ejemplo:*

```
?- eliminar(3,[1,3,2,4,5,3,6,7],R).
R = [1, 2, 4, 5, 6, 7]
```

Ejercicio 3.5. *Escribe un predicado que convierta números naturales de Peano (¿Recuerdan? $s(s(s(0))) = 3$) a su equivalente decimal. Por ejemplo:*

```
?- peanoToNat(s(s(s(0))),N).
N = 3.
?- peanoToNat(0,N).
N = 0.
```

Ejercicio 3.6. *Modifique el predicado nodos/2 para que los nodos de un árbol binario sean colectados en una visita en pre-orden y otra en post-orden.*

Ejercicio 3.7. *Escriban un predicado triangulo/1 cuyo argumento es un entero positivo y su salida es como sigue:*

```
?- triangulo(5).
    *
   * *
  * * *
 * * * *
* * * * *
true.
```