Kpress-Calc: Programación Avanzada

. Expresiones y reglas

Esencialmente hablando, el núcleo de KPress-Calc manipula *expresiones*[1] según una serie de *reglas de transformación*[2], a la que llamamos *programa*.

Veamos un ejemplo: en KPress-Calc, una secuencia de objetos se representa mediante una expresión de la forma

$$[objeto_1\ ,\ objeto_2\ ,\ .\ .\ .\ .\ ,\ objeto_n]$$

por ejemplo `[pera,manzana,fresa]`. Dichas expresiones son conocidas con el nombre de *listas*. Internamente, KPress-Calc representa las listas mediante una expresión de la forma

$$[objeto_1\ |\ [objeto_2\ |\ [\ .\ .\ .\ [objeto_n\ |[]]\ .\ .\ .\ ]]]$$

Así que si escribimos en el área de trabajo (no olvidar el **;** que indica a KPress-Calc final de expresión)

```
[pera|[manzana|[fresa|[]]]];
```

y presionamos el botón F2 , obtendremos

```
[pera,manzana,fresa]
```

Ahora supongamos que queremos una función que extraiga el primer elemento de una lista. Para ello escribimos en el área de trabajo:

```
primer_elemento([X|Y]) := X;

primer_elemento([pera,manzana,fresa]);
```

y tras presionar F2 , obtenemos

```
pera
```

¿Qué ha pasado? KPress-Calc ejecuta secuencialmente las expresiones que hay en el área de trabajo. La primera que encuentra es de la forma $expresión_1 := expresión_2$. Esto indica a KPress-Calc que se trata de una regla, y lo que hace es incorporarla al programa. En adelante, nos referiremos a la $expresión_1$ como la *cabeza de la regla* y a la $expresión_2$ como la *cola de la regla*. Es importante observar que los símbolos que comienzan por (o son) una letra mayúscula, como X e Y, son los *símbolos de variable*.

---

[1] funcionales
[2] reescritura

A continuación KPress-Calc ejecuta `primer_elemento([pera,manzana,fresa])`. Como no es una regla, le aplica las del programa. El proceso de aplicación de reglas, se conoce como proceso de *reescritura*. En este caso, consiste en lo siguiente: la *cabeza* de la regla que hemos escrito, esto es la expresión `primer_elemento([X|Y])`, encaja[3] con `primer_elemento([pera|[manzana|[fresa|[]]]])` mediante la asignación

```
X =>  pera
Y => [manzana|[fresa|[]]]
```

Entonces, el resultado de aplicar la regla, es el valor que toma la cola de la regla (es decir **X**), tras dicha asignación (es decir **pera**).

¿Qué se obtiene al ejecutar el siguiente programa?

```
resto_de_elementos([X|Y]) := Y;

resto_de_elementos([pera,manzana,fresa]);
```

Sólo resta observar que:

1. El resultado que muestra KPress-Calc, es una expresión a la que no puede aplicar ninguna regla del programa, esto es, cuando no hay ninguna regla cuya cabeza unifique con la expresión o con alguna subexpresión. Por ejemplo al escribir

   ```
   primer_elemento([X|Y]) := X;
   resto_de_elementos([X|Y]) := Y;

   primer_elemento(resto_de_elementos([pera,manzana,fresa]));
   ```

   obtenemos `manzana`, para ello KPress-Calc ha realizado las siguientes transformaciones:

   ```
   primer_elemento(resto_de_elementos([pera,manzana,fresa])) →
   primer_elemento([manzana,fresa]) →
   manzana
   ```

2. En el caso de que sean aplicables varias reglas, KPress-Calc aplica la que se encuentre en primer lugar dentro del programa.

. ¿Cómo sacar el último de la lista?

Aparentemente, sacar el último de la lista debería ser similar a sacar al primero. Sin embargo en cuanto lo intentamos . . . ¡vemos que no hay forma! . . . bueno, si utilizamos varias reglas . . .

```
ultimo_elemento([X1]) := X1;
ultimo_elemento([X1,X2]) := X2;
```

[3] unifica

```
ultimo_elemento([X1,X2,X3]) := X3;
...
```

Sólo tenemos que escribir un número infinito de reglas . . . o adoptar otra estrategia. La idea es utilizar una regla, no para obtener directamente el resultado, sino para plantear una situación más sencilla, cuyo resultado coincida con lo que buscamos. Veamos, sabemos que:

```
ultimo_elemento([pera,manzana,fresa]) = ultimo_elemento([manzana,fresa])
                                      = ultimo_elemento([fresa])
```

```
ultimo_elemento([X,Y|Z]) := ultimo_elemento([Y|Z]);

ultimo_elemento([melocoton,platano,pera,manzana,fresa]);
```

y tras presionar F2 , obtenemos

```
ultimo_elemento([fresa])
```

Casi lo tenemos . . . ya que nuestra regla que dice "*el último de una lista con al menos dos elementos, coincide con el último de la lista resultante de quitarle el primero*" reduce el problema al de obtener el último de una lista formada por un único elemento. Así que si escribimos:

```
ultimo_elemento([X,Y|Z]) := ultimo_elemento([Y|Z]);
ultimo_elemento([X]) := X;

ultimo_elemento([melocoton,platano,pera,manzana,fresa]);
```

y tras presionar F2 , obtenemos

```
fresa
```

Observación: Vemos que el alcance de una variable es la regla en la que aparece, por ello la X de la primera regla no tiene nada que ver con la X de la segunda.

¡Te atreves con los primeros elementos! es decir

```
primeros_elementos([pera,manzana,fresa]) → [pera,manzana]
```

Aquí

```
primeros_elementos([pera,manzana,fresa] ≠ primeros_elementos([manzana,fresa])
```

¿Qué hacer? . . . nos plantemos la siguiente pregunta: ¿Podemos construir fácilmente

```
primeros_elementos([pera,manzana,fresa])
```

a partir de `primeros_elementos([manzana,fresa])`? . . . un modo de hacerlo es:

```
primeros_elementos([pera,manzana,fresa]) = [pera,manzana] = [pera|[manzana]]
```

```
= [pera | primeros_elementos([manzana,fresa])]
```

Ya lo tenemos: si escribimos en el área de trabajo

```
primeros_elementos([X,Y|Z]) := [X|primeros_elementos([Y|Z])];
primeros_elementos([X]) := [];

primeros_elementos([melocoton,platano,pera,manzana,fresa]);
```

tras presionar F2 , obtenemos `[melocoton,platano,pera,manzana]`.

. Jugando con listas

Aunque no hemos visto todo el lenguaje, sí lo esencial. Antes de seguir es fundamental asimilar lo expuesto, y el único modo de hacerlo es programando. A continuación discutiremos unos ejemplos, donde es importante, una vez entendido el planteamiento, intentar dar con la solución antes de leer la explicación. Con ello, en un momento dado, particular de cada persona, se produce un “clic mental” (como cuando se logra ver en 3D un estereograma), a partir del cual se es capaz de programar recursivamente.

juntar(X,Y)

Queremos programar una función que junte dos listas, es decir,

```
juntar([melocoton,platano],[pera,manzana,fresa])
                    ↓
[melocoton,platano,pera,manzana,fresa]
```

. . . parece elemental . . . si escribimos en el área de trabajo

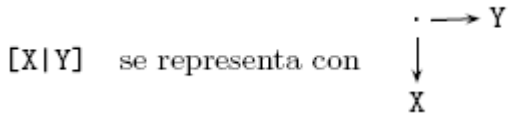
```
juntar(X,Y) := [X|Y];

juntar([melocoton,platano],[pera,manzana,fresa]);
```

tras presionar F2 obtenemos ¡¡¡ `[[melocoton,platano],pera,manzana,fresa]` !!! . . . a lo mejor no hemos entendido bien, probemos con la coma . . .

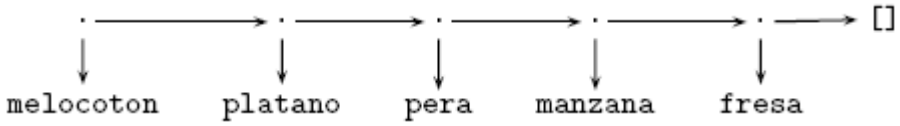
```
juntar(X,Y) := [X,Y];

juntar([melocoton,platano],[pera,manzana,fresa]);
```

... y tras presionar F2 obtenemos ¡¡¡ `[[melocoton,platano],[pera,manzana,fresa]]` !!!

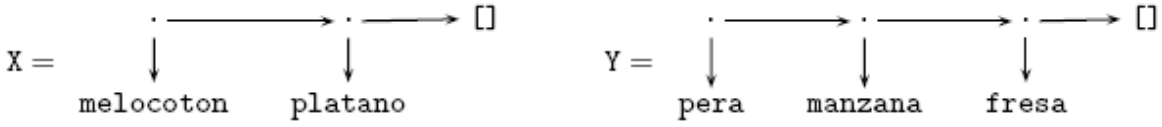
Para entender lo que está ocurriendo, tenemos que ver las listas del mismo modo en que lo hace KPress-Calc. Para ello adoptaremos la siguiente forma de representar gráficamente las listas:



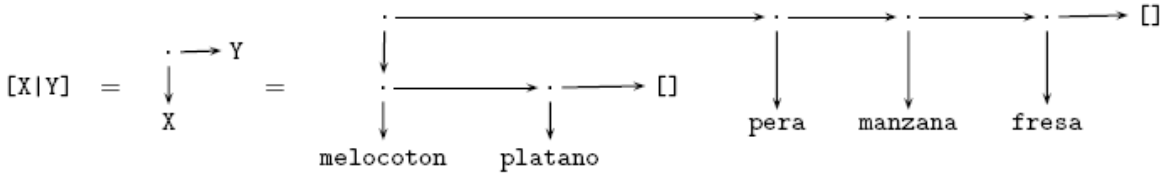
El resultado deseado es `[melocoton|[platano|[pera|[manzana|[fresa|[]]]]]]`, luego su representación gráfica es:



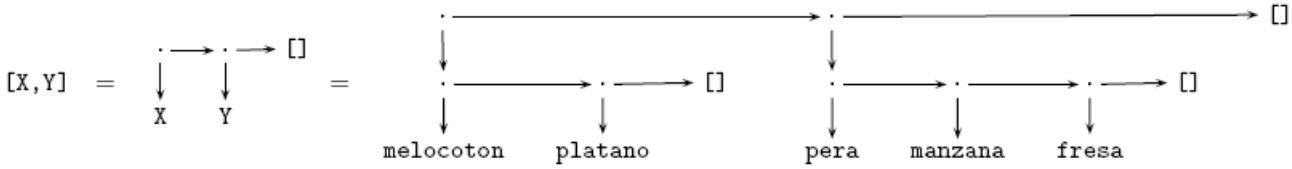
Ahora veamos porqué no hemos obtenido este resultado. Sabiendo que



por un lado tendremos que



y por otro



Volvamos al problema, `juntar(X,Y)` significa encadenar `Y` al final `X`. Ahora tenemos dos argumentos, entonces para reducir el problema, ¿Descomponemos el primero, . . . el segundo, . . . o ambos? . . .Probando la primera opción, vemos que

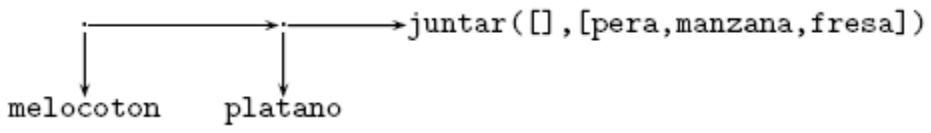
```
juntar([melocoton,platano],[pera,manzana,fresa]) =

                    = [melocoton,platano,pera,manzana,fresa]
                    = [melocoton | [platano,pera,manzana,fresa]]
                    = [melocoton | juntar([platano],[pera,manzana,fresa])]
```

Ensayemos entonces la siguiente regla reductora:

```
juntar([X|Y],Z) := [X|juntar(Y,Z)];
```

```
juntar([melocoton,platano],[pera,manzana,fresa]);
```

y tras presionar F2 , obtenemos
`[melocoton,platano|juntar([],[pera,manzana,fresa])]`, es decir,



Sólo tenemos que añadir una regla para juntar la lista vacía `[]` con cualquier otra:

```
juntar([X|Y],Z) := [X|juntar(Y,Z)];
juntar([],Z) := Z;

juntar([melocoton,platano],[pera,manzana,fresa]);
```

y tras presionar F2 , finalmente obtenemos `[melocoton,platano,pera,manzana,fresa]`.

voltear(X)

Queremos una función que ponga en orden inverso los elementos de una lista. Por ejemplo

```
voltear([melocoton,platano,pera,manzana,fresa])
                          ↓
[fresa,manzana,pera,platano,melocoton]
```

Apliquemos nuestra estrategia: ¿Nos ayuda en algo ...?

```
voltear([platano,pera,manzana,fresa]
```

Desde luego que sí:

```
voltear([melocoton,platano,pera,manzana,fresa]) = [fresa,manzana,pera,platano,melocoton]

      voltear([platano,pera,manzana,fresa]) = [fresa,manzana,pera,platano]
```

Así que para obtener el resultado deseado sólo debemos colocar `melocoton` al final del resultado de `voltear([platano,pera,manzana,fresa]`. Para ello podemos utilizar la función juntar:

```
voltear([melocoton,platano,pera,manzana,fresa]) =

      juntar( voltear([platano,pera,manzana,fresa]) , [melocoton] )
```

Ya podemos construir la regla de reducción que actúa siempre que la lista sea no vacía. Luego si añadimos la regla que voltea la lista vacía, habremos resuelto el problema:

```
voltear([X|Y]) := juntar(voltear(Y),[X]);
voltear([]) := [];

juntar([X|Y],Z) := [X|juntar(Y,Z)];
```

```
juntar([],Z) := Z;

voltear([melocoton,platano,pera,manzana,fresa]);
```

y tras presionar F2 obtenemos `[fresa,manzana,pera,platano,melocoton]`.
¿Qué hemos aprendido?

a) Se puede invocar a una función (juntar), para generar el resultado final (`[fresa,manzana,pera,platano,melocoton]`), a partir del de el problema reducido (`voltear([platano,pera,manzana,fresa])`).
b) El orden relativo en que se definen las reglas de funciones diferentes, es irrelevante. Sólo es importante que definirlas antes de usarlas: por ejemplo

```
voltear([X|Y]) := juntar(voltear(Y),[X]);
voltear([]) := [];

voltear([melocoton,platano,pera,manzana,fresa]);

juntar([X|Y],Z) := [X|juntar(Y,Z)];
juntar([],Z) := Z;
```

tras presionar F2 obtenemos

```
juntar(juntar(juntar(juntar(juntar([],[fresa]),[manzana]),[pera]),[
platano]),[melocoton])
```

sublistas(X)

Queremos una función que genere todas las sublistas de una dada, por ejemplo

`sublistas([1,2,3])` $\longrightarrow$ `[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]`

¿Qué podemos obtener de `sublistas([2,3])`?. . . aquellas sublistas de `[1,2,3]` que no incluyen al `1`. Pero aún hay algo más:

`sublistas([2,3]) = [[2,3],[2],[3],[]]`

y las que nos faltan para `completar` `sublistas([1,2,3])` son precisamente

`[[1,2,3],[1,2], [1,3],[1]]`

es decir, el resultado de `incluir` el `1` en cada una de las `sublistas([2,3])`. ¡Ya sabemos todo lo necesario para programar `sublistas(X)`! . . . es decir, `completar`, `incluir` y `juntar`:

```
sublistas([X|Y]) := completar(X,sublistas(Y));
sublistas([]) := [[]];

completar(X,L) := juntar(incluir(X,L),L);
```

```
incluir(X,[S|L]) := [[X|S]|incluir(X,L)];
incluir(X,[]) := [];

juntar([X|Y],Z) := [X|juntar(Y,Z)];
juntar([],Z) := Z;

sublistas([1,2,3]);
```

y tras presionar F2 obtenemos `[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]`.

. Funciones booleanas

Hay una familia de expresiones en KPress-Calc sin necesidad de reglas de transformación. Se trata de las que invocan a funciones predefinidas. Por ejemplo, al escribir

```
1 < 2;
```

tras presionar F2 obtenemos `true`.

Gran parte de las funciones booleanas predefinidas, son expresables mediante reglas y han sido predefinidas para facilitar la programación. A continuación vemos algunas de ellas:

```
not true := false;
not false := true;
true and true := true;
true and false := false;
false and true := false;
false and false := false;
true or true := true;
true or false := true;
false or true := true;
false or false := false;
X = X := true;
X = Y := false;
```

Observación: Para entender la función de igualdad (X=Y), hay que recordar que KPress-Calc prueba secuencialmente las reglas referidas a una misma función. Entonces, si *expresión*$_1$=*expresión*$_2$ no unifica con X=X, necesariamente *expresión*$_1$ es distinta de *expresión*$_2$, que es lo que se encarga de establecer la segunda regla.

Aunque estrictamente hablando no es booleana, añadimos a este grupo la siguiente función de control:

```
if true then X else Y := X;
if false then X else Y := Y;
```

Hay, sin embargo, funciones booleanas predefinidas que no pueden ser expresadas mediante reglas, por ejemplo la de orden `X<Y`, `isnumeric(X)` para determinar si `X` es un número, o `isinteger(X)` para determinar si `X` es un número entero.

Ordenación de listas

Una de las cosas interesantes que podemos hacer con `X<Y` es ordenar una lista. Pero antes de hacerlo, comencemos por una función más sencilla. La función se llama `merge(X,Y)`, y construye una lista ordenada a partir de dos listas ordenadas. Por ejemplo

`merge([2,4,6],[1,2,8])` ⟶ `[1,2,2,4,6,8]`

Lo interesante de `merge(X,Y)` es que el primer elemento de la lista resultante, o bien es el primer elemento de `X` o bien el primer elemento de `Y`. Además una vez retirado, seguimos teniendo dos listas ordenadas, con lo que podemos aplicar el mismo criterio para sacar el siguiente. En fin, podemos programarla fácilmente del siguiente modo:

```
merge([A|X],[B|Y]) := if A < B then [A|merge(X,[B|Y])]
                      else if B < A then [B|merge([A|X],Y)]
                      else [A,B|merge(X,Y)];
merge([],Y) := Y;
merge(X,[]) := X;

merge([2,4,6],[1,2,8]);
```

y obtenemos `[1,2,2,4,6,8]`.

La función `merge(X,Y)` genera una lista ordenada, a partir de dos más pequeñas también ordenadas . . . bueno, normalmente buscamos este tipo de situación para programar recursivamente. Así que para ordenar una lista

1. Dividimos la lista en dos trozos.
2. Ordenamos cada trozo.
3. Aplicamos `merge` a los trozos ordenados.

Lo ideal es que los dos trozos sean de tamaño similar, y no habiendo más requerimientos, podemos hacerlos del modo más simple (desde el punto de vista de programación) posible. Por ejemplo, el primer trozo puede contener los elementos que ocupan una posición impar:

```
impar([A,B|X]) := [A|impar(X)];
impar([A]) := [A];
impar([]) := [];
```

y el otro los que están en lugar par:

```
par([A|X]) := impar(X);
par([]) := [];
```

Y ahora sólo falta escribir el método de ordenación descrito:

```
ordena([A,B|X]) := merge(ordena([A|impar(X)]),ordena([B|par(X)]));
ordena([A]) := [A];
ordena([]) := [];
```

Si ponemos todo en el área de trabajo junto con

```
ordena([melocoton,platano,pera,manzana,fresa]);
```

tras presionar F2 obtenemos `[fresa,manzana,melocoton,pera,platano]`.

. Aritmética

Entre las funciones predefinidas, las más importantes son sin duda las aritméticas. En efecto, si escribimos

```
2*(3+2);
```

naturalmente obtenemos `10`.

Con ellas, por ejemplo, podemos contar el número de elementos de una lista:

```
numero_de_elementos([X|Y]) := 1+numero_de_elementos(Y);
numero_de_elementos([]) := 0;

numero_de_elementos([melocoton,platano,pera,manzana,fresa]);
```

y tras presionar F2 obtenemos `5`.

Observación: Para que una expresión invoque a una función predefinida, es necesario que sus argumentos sean adecuados. Por ejemplo `7*5` → `35`, porque los argumentos 7 y 5 son números. Sin embargo `7*[]` → `7*[]` porque `[]` no es un número, y en consecuencia la expresión `7*[]` no invoca a la función predefinida producto de números.

Vectores y matrices

Aunque en KPress-Calc no está predefinida el álgebra matricial, es muy sencillo implementarla. Comencemos con los vectores: adoptando que un vector es una lista de números, la suma de vectores puede programarse del siguiente modo:

```
[A|X]+[B|Y] := [A+B|X+Y];
[]+[] := [];

[1,2,3]+[2,-1,-3];
```

y tras presionar F2 obtenemos `[3,1,0]`.

Es interesante prestar atención al funcionamiento de este programa:

Al evaluar la expresión `[1|[2|[3|[]]]] + [2|[-1|[-3|[]]]]`, KPress-Calc encuentra que unifica con la cabeza de regla `[A|X] + [B|Y] := [A+B|X+Y]`, mediante la asignación

```
A  →  1
X  →  [2|[3|[]]]
B  →  2
X  →  [-1|[-3|[]]]
```

Así que remplaza dicha expresión por `[1+2 | [2|[3|[]]] + [-1|[-3|[]]] ]`.

Como esta expresión no unifica con la cabeza de ninguna regla, KPress-Calc procede a evaluar las subexpresiones. La primera que encuentra es `1+2`, que tras invocar a la *suma de números* es remplazada por 3. Con ello la expresión resultante es `[3 | [2|[3|[]]] + [-1|[-3|[]]] ]`.

Repetimos este proceso con la subexpresión `[2|[3|[]]] + [-1|[-3|[]]]`, hasta que deja de ser posible aplicar alguna regla o alguna función predefinida.

Al evaluar la expresión `[1|[2|[3|[]]]] + [2|[-1|[-3|[]]]]`, KPress-Calc encuentra que unifica con la cabeza de regla `[A|X] + [B|Y] := [A+B|X+Y]`, mediante la asignación

Si decidimos representar a una matriz mediante la lista de sus vectores fila, por ejemplo

$$\begin{bmatrix} 2 & 5 \\ 3 & 8 \\ 1 & 0 \end{bmatrix}$$

mediante `[[2,5],[3,8],[1,0]]`, para programar la suma de matrices . . . ¡no tenemos que hacer nada! ya que:

```
[A|X]+[B|Y] := [A+B|X+Y];
[]+[] := [];

[[2,5],[3,8],[1,0]]+[[1,1],[2,2],[0,0]];
```

tras presionar F2 obtiene `[[3,6],[5,10],[1,0]]`.

Del mismo modo que hemos programado la suma, podemos programar la resta y el opuesto:

```
[A|X]-[B|Y] := [A-B|X-Y];
[]-[] := [];

-[A|X] := [-A|-X];
-[] := [];
```

. . . y nos sirven tanto para vectores como para matrices. Incluso, podemos programar una función universal que determine si un número,vector o matriz es nulo:

```
es_nulo(X) := (X=X-X);
```

Ahora supongamos que queremos hacer lo mismo con el producto. Comencemos por el producto por escalares:

```
A*[B|X] := [A*B|A*X];
A*[] := [];
```

Con este programa funciona tanto para el producto de un número por un vector `3*[1,2,3]` ⟶ `[3,6,9]`, como para el producto de un número por una matriz `3*[[1,2,3],[1,0,0]]` ⟶ `[[3,6,9],[3,0,0]]`. ¿Pero, qué pasa si ponemos dos matrices? por ejemplo

$$\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$$

obtenemos ¡¡¡ `[[[[1,0],[0,1]]*1,[[1,0],[0,1]]*0], [[[1,0],[0,1]]*0,[[1,0],[0,1]]*1]]`!!!

Por tanto tenemos que decir al programa, que la reglas del producto únicamente deben utilizarse cuando el primer argumento es un número . . . Bien, entonces si escribimos

```
A*[B|X] := if isnumeric(A) then [A*B|A*X] else A*[B|X];
A*[] := if isnumeric(A) then [] else A*[];

[[1,0],[0,1]]*[[1,0],[0,1]];
```

tras presionar F2 ... ¡¡¡ KPress-Calc entra en un bucle infinito !!!

Los guardas

Hasta ahora, la única forma de decidir si una regla se aplica o no, es a través de la forma de las expresiones. Con la regla del producto necesitamos decidir, en función del contenido. Este tipo de situaciones se resuelven en KPress-Calc mediante el uso de *guardas*. Una regla con guarda es una expresión de la forma

$$\textit{expresión}_1 := \textit{expresión}_2 \;\; <== \;\; \textit{expresión}_3$$

El *guarda* (la *expresión*$_3$) sirve para invocar a una función boolena. Y una regla con guarda, únicamente es aplicada si el resultado de dicha función es *true*.

Ahora ya podemos escribir la regla del producto:

```
A*[B|X] := [A*B|A*X] <== isnumeric(A);
A*[] := [] <== isnumeric(A);
```

entonces `[[1,0],[0,1]]*[[1,0],[0,1]]` ⟶ `[[1,0],[0,1]]*[[1,0],[0,1]]`

Ahora, podemos añadir a esta regla, otra para el producto escalar de dos vectores

$$(a_1, \cdots, a_n) * (b_1, \cdots, b_n) = a_1 b_1 + \cdots a_n b_n$$

```
[A,A2|X]*[B,B2|Y] := [A*B|[A2|X]*[B2|Y]] <== isnumeric(A);
[A]*[B] := [A*B] <== isnumeric(A);

[1,1,1]*[1,1,1];
```

obtiene 3.

Lo interesante, es que si añadimos a estas reglas, las de la suma de vectores-matrices y el producto por escalares, ¡tendremos programado el producto de un vector por una matriz (dando como resultado un vector)!

$$(1,1)\begin{bmatrix} 1 & 2 & 3 \\ 2 & 2 & 2 \end{bmatrix} = (3,4,5)$$

es decir, `[1,1]*[[1,2,3],[2,2,2]]` $\longrightarrow$ `[3,4,5]`. Por tanto casi tenemos construido el producto de matrices "*la filas del producto son obtenidas multiplicado cada fila de la primera matriz por la segunda*". Por tanto si añadimos

```
[X|R]*Y->[X*Y|R*Y] <== not isnumeric(X);
[]*Y -> [];

[[1,0],[0,1]]*[[1,0],[0,1]];
```

obtendremos `[[1,0],[0,1]]`.

Además, tendremos definido el producto de una matriz por un vector (que habitualmente se "pinta" en forma de columna)

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 2 & 2 \end{bmatrix}\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 6 \\ 6 \end{pmatrix}$$

es decir `[[1,2,3],[2,2,2]]*[1,1,1]` $\longrightarrow$ `[6,6]`.