Tema 3:

1.- Conceptos básicos

1.1.- Un poco de sintaxis

- **Identificadores:** se definen utilizando caracteres alfabéticos (en minúsculas y mayúsculas), dígitos y subrayado '_'. El primer caracter debe ser alfabético. No se pueden usar palabras reservadas, funciones predefinidas ni nombres de operadores como identificadores.

- **Números enteros:** pueden representarse en decimal, hexadecimal u octal. Por ejemplo, 0, 005, 123, 0x1b7 y 0o777 son constantes numéricas válidas.

- MiniZinc distingue entre MAYÚSCULAS y minúsculas. Así, *solve* sirve para resolver restricciones, pero si en su lugar utilizamos *Solve* obtenemos un error

Conjuntos de símbolos importantes:

- **Palabras reservadas**: ann, annotation, any, array, assert, bool, constraint, enum, float, function, in, include, int, list, of, op, output, minimize, maximize, par, predicate, record, set, solve, string, test, tuple, type, var y where.

- **Operadores**

  - **Unarios:** *not*, +, y -

  - **Binarios:** <->, ->, <-, \/, xor, /\, <, >, <=, >=, ==, =, !=, in, subset, superset, union, diff, symdiff, .., intersect, ++, +, -, *, /, div (división entera) y mod (*a mod b* es el resto que se obtiene al hacer la división entera entre *a* y *b*. El resultado siempre tiene el mismo signo que *a*).

- **Funciones predefinidas:** abort, abs, acosh, array intersect, array union, array1d, array2d, array3d, array4d, array5d, array6d, asin, assert, atan, bool2int, card, ceil, concat, cos, cosh, dom, dom array, dom size, fix, exp, floor, index set, index set 1of2, index set 2of2, index set 1of3, index set 2of3, index set 3of3, int2float (convierte un entero en un número), is fixed, join, lb, lb array, length, ln, log, log2, log10, min, max, pow, product, round, set2array, show, show int, show float, sin, sinh, sqrt, sum, tan, tanh, trace, ub y ub array.

1.2.- Ficheros de datos y aserciones

En el ejemplo de la fundición (tema 1), cada vez que cambia el valor de algún parámetro como el precio del mineral *a* o el número de días de los que disponemos tenemos que cambiar el programa. Para evitar esto MiniZinc ofrece la posibilidad de dar valor a los parámetros en un fichero separado con extensión *.dzn*. Así podemos tener distintos ficheros dando valores a los parámetros sin tener que tocar el programa.
El programa fuente "fundicion4.mzn" queda:

Fundición. Versión 4:

```
%%%%%%%% fundición
```

```
float: dias;
float: cant_h;
float: cant_c;
float: precio_a;
float: precio_b;

% cantidades a fabricar de cada producto
var float: a;
var float: b;

% límite de tiempo de uso de horno
constraint  a*3.0+b*4.0<=dias;
% no se puede superar la cant. de hierro almacenado
constraint 14.0*a+2.1*b<=cant_h;
% ni la de carbón
constraint 2.0*a+11.0*b<=cant_c;
% cantidades no negativas
constraint a>=0.0;
constraint b>=0.0;

% maximizar el beneficio
solve maximize precio_a*a+precio_b*b;

%mostrar el resultado
output ["A=", show(a), "\t B=", show(b),  "\tganancia:", show(18.5*a+20.7*b),
          "\nSobra carbón: ", show(cant_c-(2.0*a+11.0*b)), "\tSobra hierro: ", show(cant_h-(14.0*a+2.1*b)),
         "\n Sobran días de horno:", show(dias-(a*3.0+b*4.0)),"\n"];
```

Y el de parámetros, "fundicion4.dzn":

```
dias = 15.0;
cant_h = 40.0;
cant_c=13.0;
precio_a=20.7;
precio_b=18.5;
```

Ahora se utiliza el fichero de parámetros de la siguiente forma:

```
[~/docencia/1213/pr] $ mzn-g12mip fundicion4.mzn fundicion4.dzn
A=2.7550066755674227     B=0.6809078771695596   ganancia:65.06241655540721
Sobra carbón: 0.0       Sobra hierro: 7.105427357601002e-15
 Sobran días de horno:4.011348464619493
```

También se puede introducir el fichero de datos a través de la linea de comandos:

```
[~/docencia/1213/pr/ejemplos] $ mzn-g12mip fundicion4.mzn -D "dias = 15.0; cant_h = 40.0; cant_c=13.0; precio_a=20.7; precio_b=18.5;"
A=2.7550066755674227     B=0.6809078771695596   ganancia:65.06241655540721
Sobra carbón: 0.0       Sobra hierro: 7.105427357601002e-15
 Sobran días de horno:4.011348464619493
```

Un fichero modelo puede acompañarse de tantos ficheros de parámetros como se desee, pero cada parámetro debe tomar valor en exactamente uno de los ficheros

Una idea habitual en cualquier modelo de programación es que hay que comprobar que los valores introducidos por el usuario son válidos. Para esto MiniZinc proporciona un mecanismo de aserciones. La sintaxis de una aserción es:

Si no se cumple la condición la ejecución del modelo se detiene y se muestra el error correspondiente por pantalla. En nuestro caso las condiciones impidan que los parámetros tomen valores negativos:

Fundición. Versión 5:

```
%%%%%%%% fundición
float: dias;
constraint assert(dias >= 0.0,"Fichero de parámetros no válido: " ++
                                    "cantidad negativa de días");
float: cant_h;
constraint assert(cant_h >= 0.0,"Fichero de parámetros no válido: " ++
                                    "cantidad negativa de hierro");
float: cant_c;
constraint assert(cant_c >= 0.0,"Fichero de parámetros no válido: " ++
                                    "cantidad negativa de carbón");
float: precio_a;
constraint assert(precio_a >= 0.0,"Fichero de parámetros no válido: " ++
                                    "precio negativo para producto A");
float: precio_b;
constraint assert(precio_b >= 0.0,"Fichero de parámetros no válido: " ++
                                    "precio negativo para producto B");

% cantidades a fabricar de cada producto
var float: a;
var float: b;

% límite de tiempo de uso de horno
constraint  a*3.0+b*4.0<=dias;
% no se puede superar la cant. de hierro almacenado
constraint 14.0*a+2.1*b<=cant_h;
% ni la de carbón
constraint 2.0*a+11.0*b<=cant_c;
% cantidades no negativas
constraint a>=0.0;
constraint b>=0.0;

% maximizar el beneficio
solve maximize precio_a*a+precio_b*b;

%mostrar el resultado
output ["A=", show(a), "\t B=", show(b),  "\tganancia:", show(18.5*a+20.7*b),
          "\nSobra carbón: ", show(cant_c-(2.0*a+11.0*b)), "\tSobra hierro: ", show(cant_h-(14.0*a+2.1*b)),
         "\n Sobran días de horno:", show(dias-(a*3.0+b*4.0)),"\n"];
```

Ahora el modelo detecta parámetros erróneos:

```
[~/docencia/1213/pr/ejemplos] $ mzn-g12mip fundicion5.mzn -D "dias = 15.0; cant_h = 40.0; cant_c=13.0; precio_a=-20.7; precio_b=18.5;"
mzn2fzn:
  error:
  fundicion5.mzn:15
  In constraint.
  In 'assert' expression.
  Assertion failure: "Fichero de parámetros no válido: precio negativo para producto A"
```

1.3.- Estructura de un modelo

Un modelo puede contener los siguientes tipos de elementos distintos:

- Instrucciones *include(nombrefichero);* donde *nombrefichero* es una cadena de caracteres.
  Un ejemplo típico es *alldifferent*, que comprueba que todos los elementos de una lista son diferentes.

```
include "alldifferent.mzn";
...
constraint alldifferent([A,B,C,D,E]);
...
```

- Declaración de variables, ya sean variables de decisión o parámetros.
- Declaración de constraints.
- Llamada a *solve*.
- Instrucción *output*
- Declaración de predicados
- Anotaciones

2.- Arrays y conjuntos

2.1.- Declaración

Declaración: array[ini1..fin1,..., inik..fink] of tipo:nombre;

Donde *ini1*, *inik*, *fin1*, *fink* son enteros, *tipo* es el tipo de las variables contenidas y *nombre* el nombre de la variable.
Ejemplos:

- array[0..w, 0..h] of var float: t;
- array [1..n] of var 1..n: t;
- array [1..3] of 1..n: s = [1,2,3];
- array [1..3,1..2] of par 0..500: u = [| 1,2, | 4,5, | 6,7 |];

2.2 Listas intensionales

MiniZinc permite escribir arrays mediante listas intensionales. La sintaxis general es:

Listas intensionales: [expr | generator-exp]

la expresión *expr* construye los valores de la lista a partir de los valores generados por . A su vez, es una lista de generadores individuales opcionalmente seguidos por la palabra clave *where* y una condición. La condición se utiliza como filtro para descartar algunos valores generados. Por último los generadores son de la forma

```
 id1,..., idk in array-exp
```

Un ejemplo de lista intensional.

```
[a[i] != a[j] | i,j in 1..3 where i < j]
```

que se evalúa a:

```
[ a[1] != a[2],a[1] != a[3], a[2] != a[3] ]
```

.

2.3.- Funciones sobre arrays

Los arrays en MiniZinc admiten las siguientes funciones:

- *sum*, *product*: suma (respectivamente multiplica) todos los elementos del array
- *min*, *max*: elemento mínimo (respectivamente máximo)
- *length*: longitud del array.

2.4.- Bucles y funciones de agregación

La instrucción *forall* es la que nos permite manejar los arrays.

forall(nombrevar1, nombrevark *in* ini..fin *where* cond)(instrucción);

la parte *where* es opcional. Ejemplo:

- constraint forall (i,j in 1..3 where i < j) (a[i] != a[j]);

Una segunda versión de *forall* tiene como entrada un array de expresiones lógicas (es decir de constraints) y las convierte en una única expresión lógica correspondiente a la conjunción de las expresiones lógicas en el array.
Por ejemplo en

```
forall( [a[i] != a[j] | i,j in 1..3 where i < j])
```

primero la lista intensional se evalúa a

```
[ a[1] != a[2],a[1] != a[3], a[2] != a[3] ]
```

, y al aplicar la función *forall*

```
 a[1] != a[2] ^ a[1] != a[3] ^ a[2] != a[3]
```

en realidad las dos sintaxis de *forall* son equivalentes. El ejemplo anterior se puede escribir también como:

```
forall (i,j in 1..3 where i < j) (a[i] != a[j])
```

forall es solo una de las 4 funciones de agregación que include MiniZinc. Las otras 3 son:

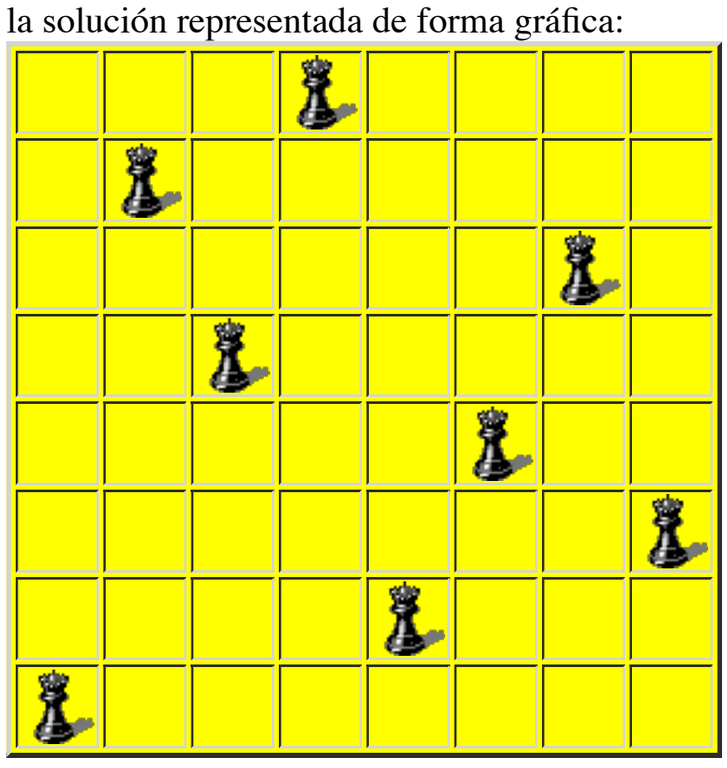
- *exists*: hace la disyunción del array de expresiones lógicas.
- *odd*: asegura que se cumplirá una cantidad impar de las expresiones lógicas.
- *iffall*: asegura que se cumplirá una cantidad par de las expresiones lógicas.

2.5 Ejemplo

```
% reinas

int:n=8;

array [1..n] of var 1..n: t;


% constraints

%fila
constraint forall (i in 1..n-1)(forall(j in i+1..n)(t[i]!=t[j]));

% dos diagonales
constraint forall (i in 1..n-1)(forall(j in i+1..n)(t[i]!=t[j]+(j-i)));
constraint forall (i in 1..n-1)(forall(j in i+1..n)(t[i]!=t[j]-(j-i)));

solve satisfy;

output[ show(t[i])++" " | i in 1..n];
```

```
[~/docencia/1213/pr]$ minizinc reinas.mzn
4 2 7 3 6 8 5 1
```

la solución representada de forma gráfica:



3.- Conjuntos

3.1.- Declaración

Declaración: set of tipo:nombre;

El tipo puede ser *int*, *float* o *bool*
Los **literales** pueden tomar dos formas:

- {elemnto1, ..., elementon }
- valIni..valFin

En el segundo caso tenemos un subrango donde *valIni*..*valFin*. También se admiten *conjuntos intensionales* definidos de forma análoga a las listas intensionales.

3.2.- Operaciones sobre conjuntos

Las operaciones sobre conjuntos que permite MiniZinc: pertenencia (*in*), subconjunto, superconjunto (*subset*, *superset*), unión (*union*), intersección (*intersect*), diferencia de conjuntos (*diff*), diferencia simétrica (*symdiff*) y cardinalidad (*card*).
Ejemplo sencillo: encontrar dos conjuntos con valores enteros entre 1 y 7, tales que su unión sea un conjunto con 4 valores y su intersección el conjunto {4,5,7}:

```
% conjuntos
int:n=4;
int:rango=7;

%conjuntos
var set of 1..rango: s1;
var set of 1..rango: s2;

constraint s1 intersect s2={4,5,7};
constraint card (s1 union s2)=4;

solve satisfy;

output ["s1: "++show(s1)++"\n", "s2: "++show(s2) ];
```

3.3.- Conjuntos y arrays

Una de las utilidades de los conjuntos es servir para representar rangos de arrays:

```
int:n=10;

set of int:s = 1..n;
array [s] of var 1..1000:u;
array [s] of var 1..1000:v;

constraint forall(i in s)(u[i]>i*v[i] /\ v[i]>i);

solve satisfy;

output [ "u["++show(i)++"]= "++show(u[i])++
         " v["++show(i)++"]= "++show(v[i])++"\n" | i in s]
```

que da la salida:

```
D:\minizinc>minizinc arraypru.mzn
u[1]= 3 v[1]= 2
u[2]= 7 v[2]= 3
u[3]= 13 v[3]= 4
u[4]= 21 v[4]= 5
u[5]= 31 v[5]= 6
u[6]= 43 v[6]= 7
u[7]= 57 v[7]= 8
u[8]= 73 v[8]= 9
u[9]= 91 v[9]= 10
u[10]= 111 v[10]= 11
----------
```


Error común: intentar definir un array mediante un conjunto que no está construido mediante rangos.

Esto se ve en el ejemplo:

```
int:n=10;

set of int:s = {3,4,5};
array [s] of var 1..1000:u;
array [s] of var 1..1000:v;
...
```

da el error:

```
D:\minizinc>minizinc arrayset2.mzn
arrayset2.mzn:5:
  type-inst error: MiniZinc does not permit array indices without fixed ranges
arrayset2.mzn:6:
```

3.4.-Ejemplo

Un ejemplo más práctico y complicado: generalizamos el problema de la fundición para producción de productos en general. Queremos elaborar productos a elegir entre *nproducts* productos, donde el nombre del producto i-ésimo es *name[i]*. Tenemos que la ganancia obtenida obtenida por unidad del producto *i* es *profit[i]*.

Por otro lado tenemos *nresources* recursos, los recursos se utilizan para producir los productos. Los nombres de los recursos están en la tabla *rname* y la cantidad de cada recurso en la tabla *capacity*. Por último la tabla *consumption* indica la cantidad de recursos que se precisa en la elaboración de cada unidad de producto.

```
% Number of different products

int: nproducts;
set of int: Products = 1..nproducts;

%profit per unit for each product
array[Products] of int: profit;
array[Products] of string: pname;

%Number of resources
int: nresources;
set of int: Resources = 1..nresources;

%amount of each resource available
array[Resources] of int: capacity;
array[Resources] of string: rname;

%units of each resource required to produce 1 unit of product
array[Products, Resources] of int: consumption;

constraint assert(forall (r in Resources, p in Products) (consumption[p,r] >= 0), "Error: negative consumption");

% bound on number of Products
int: mproducts = max (p in Products)
                 (min (r in Resources where consumption[p,r] > 0)
                      (capacity[r] div consumption[p,r]));

% Variables: how much should we make of each product
array[Products] of var 0..mproducts: produce;
array[Resources] of var 0..max(capacity): used;

% Production cannot use more than the available Resources:
constraint forall (r in Resources) ( used[r] = sum (p in Products)(consumption[p, r] * produce[p]) /\ used[r] <= capacity[r]);

% Maximize profit
solve maximize sum (p in Products) (profit[p]*produce[p]);

output [ show(pname[p]) ++ " = " ++ show(produce[p]) ++ ";\n" | p in Products ] ++ [ show(rname[r]) ++ " = " ++ show(used[r]) ++ ";\n" | r in
```

Fichero de datos:

```
% Data file for simple production planning model
nproducts = 2; %banana cakes and chocolate cakes
profit = [400, 450]; %in cents
pname = ["banana-cake", "chocolate-cake"];

nresources = 5; %flour, banana, sugar, butter, cocoa
capacity = [4000, 6, 2000, 500, 500];
rname = ["flour","banana","sugar","butter","cocoa"];
consumption= [| 250, 2, 75, 100, 0, | 200, 0, 150, 150, 75 |];
```

Llamada:

```
D:\minizinc>minizinc productos.mzn dataprods.dzn
banana-cake = 2;
chocolate-cake = 2;
flour = 900;
banana = 4;
sugar = 450;
butter = 500;
cocoa = 150;
----------
```

3.5.-Otro ejemplo

Desayuno en Tyfanni's (también conocido como problema de la mochila 0-1).

Nos encontramos en la prestigiosa joyería Tyfanni's. Casualmente el guardia de seguridad a salido a desayunar, y el empleado se ha dejado una vitrina abierta. Nosotros disponemos de un maletín y de pocos escrúpulos. El problema es que el maletín no puede llevar más de 15 kg. Queremos maximizar la cantidad de joyas a sustraer.

```
% num. de items disponibles
int: n;

% rango desde 1 al num. items
set of int: Items = 1..n;

% lo que cuesta cada objeto
array[Items] of int: precio;

% lo que pesa cada objeto
array[Items] of int: peso;

% peso max. que cabe en la maletin
int: peso_max;

%%%% un conjunto de variables de decision
```

```
var set of Items: maletin;

%%%% restricciones
constraint sum([bool2int(i in maletin)*peso[i] | i in Items]) <= peso_max;

%%%% resolver maximizando la ganancia
solve maximize sum([bool2int(i in maletin)*precio[i] | i in Items]);

%%%% mostrar el resultado
output [show(maletin)];
```

3.- Otras instrucciones

Instrucción *if*
Sintaxis: `if expr_bool then expr1 else expr2 endif` Algunos comentarios: • La parte *else* es obligatoria. • En realidad no es una instrucción sino una expresión, es decir se puede escribir: `constraint z > (if x>y then x else y);`

Instrucción Output

Solo puede haber una instrucción *output* por modelo (programa). El parámetro de *output* es una lista de strings, que pueden ser:
• Literales tipo cadena entre comillas dobles (se admiten caracteres de escape como en C: \t para tabulador, \n para fin de línea, etc.). Los literales demasiado largos puedes descomponerse en dos literales concatenados con el operador ++ (por ejemplo para dividirlo en varias líneas).
• Instrucciones *show* para mostrar el valor de una variable de decisión. Hay variantes de *show* que pueden utilizarse para mostrar números formateados:
  ◦ *show int(n,X)*: muestra el valor de X como un entero ocupando como poco n caracteres, justificando con blancos a la derecha si n>0 y a la izquierda si n<0.
  ◦ *show float(n,d,X)*: análogo para números reales, con *d* los carateres que se mostrarán tras el punto decimal.
 	
Para mostrar variables de decisión que no tengan un valor determinado se utiliza la función *fix* que convertirá la variable de decisión en un valor concreto. Se utiliza dentro de un output para evitar errores de tipo.

5.- Predicados, declaraciones locales y reflexión

5.1.-Restricciones globales

Son predicados predefinidos en el sistema que generan restricciones para problemas que surgen a menudo. Para utilizarlas hay que incluir una instrucción *include "predicado.mzn"*, con *predicado* el nombre del predicado. En este tema vamos a mencionar 4:

La lista de todos los predicados globales en MiniZinc puede encontrarse en la página
http://www.g12.csse.unimelb.edu.au/minizinc/downloads/doc-1.5.1/mzn-globals.html

5.1.1.- Predicado *alldifferent*

alldifferent recibe como parámetro un array de variables de decisión. Genera la restricción que indica que todos los valores del array sean diferentes. (ver el ejemplo de los *sudoku* visto en clase).

5.1.2.- Predicado *cumulative*

cumulative se utiliza en problemas de planificación temporal. La idea es que se quieren realizar *n* tareas, y cada una lleva una cantidad de tiempo determinada *ti*. Además cada tarea require la utilización de *ri* unidades de un recurso, del que se sabe que se disponen de un total de *k* unidades. Las tareas pueden hacerse en paralelo, siempre y cuando haya recursos suficientes.
Su sintaxis es:

Sintaxis:
cumulative(array[int] of var int: s, array[int] of var int: t, array[int] of var int: r, var int: k)

donde

- *s*: array de variables de decisión, que representan el comienzo de cada tarea. sobre las que queremos establecer las restricciones. Su longitud es *n*, la cantidad de tareas a realizar.
- *t*: array de longitud *n* con el tiempo que lleva realizar cada tarea.
- *r*: unidades del recurso que consume cada tarea.
- *k*: unidades totales del recurso que se pueden usar.

Ejemplo: Planificación de proyectos.
Una empresa tiene que desarrollar 4 aplicaciones. Para ello dispone de 5 programadores, y suponemos que cada programador se dedica a una sola aplicación en cada

momento. Cada aplicación tarda un tiempo determinado en realizarse y requiere una cantidad prefijada de programadores. Se trata de minimizar el tiempo que se requiere para acabar los 4 programas.

El código para resolver el problema en MiniZinc:

```
include "cumulative.mzn";

int:n=4; % total programas

int:k=5; % total de programadores

int:max_tiempo = 100; % limite superior de tiempo

array [1..n] of int: tiempo = [2,4,6,3];
array [1..n] of int: prog =   [3,2,4,2];

%%% variables de decisión
array [1..n] of var 0..max_tiempo: comienzo;

var  0..max_tiempo:  total;

% cumulative
constraint cumulative(comienzo, tiempo, prog, k);

constraint forall(i in 1..n)(comienzo[i]+tiempo[i]<=total);

solve minimize total;

output["Total: "++show(total)++"\n"]++[show(comienzo[i])++" " | i in 1..n];
```

Supongamos que lo que queremos es mostrar el resultado en forma de cronograma. En este caso basta con reformar la instrucción *output*. Para cada programa escribimos una línea de salida con tantos caracteres como indica la variable *total*. El caracteres '*' en las posiciones que corresponden a momentos en los que se está desarrollando el programa y espacios en otro caso. Sería algo como:

```
output[  ..... | i in 1..n, j in 0 .. total-1];
```

(el código ..... no es importante en este momento). Sin embargo este código dará con seguridad un error de compilación. La razón es que los subrangos solo están definidos entre constantes o parámetros (que a su vez tienen un valor constante). Sin embargo, en este bucle el subrango

```
0 .. total-1
```

esta definida en términos de una variable de decisión y esto provoca el error. Para evitarlo tenemos que utilizar la función *fix* que convierte *total* en una variable tipo parámetro dentro la expresión, evitando el error.
El output resultante con el resultado convenientemente formateado y ya sin errores:

```
output["Tiempo total: "++show(total)++"\n" ]++
      [ (if j>=fix(comienzo[i]) /\ j< fix(comienzo[i])+tiempo[i]
         then "|*"
         else "|  " endif) ++
        (if j==fix(total)-1
         then "|\n"
         else "" endif)       | i in 1..n, j in 0 .. fix(total)-1];
```

que muestra el resultado:

```
D:\minizinc>minizinc prog.mzn
Tiempo total: 11
|  |  |  |*|*|  |  |  |  |  |
|*|*|*|*|  |  |  |  |  |  |
|  |  |  |  |  |*|*|*|*|*|*|
|*|*|*|  |  |  |  |  |  |  |
----------
==========
```

Ejemplo: Una mudanza. En el siguiente ejemplo (tomado del tutorial de MiniZinc) se trata de mover muebles. Cada mueble se tarda un tiempo estimado para su traslado, require una cantidad de personas y el uso de varios carritos. Tenemos por tanto 2 recursos, uno el número total de personas y otro el número total de carritos. Se quiere averiguar: cuándo se puede comenzar cada tarea con el objetivo de acabar lo antes posible.

```
include "cumulative.mzn";

int: n; % number of objects;

set of int: OBJECTS = 1..n;

array[OBJECTS] of int: duration; % duration to move
array[OBJECTS] of int: handlers; % number of handlers required
array[OBJECTS] of int: trolleys; % number of trolleys required

int: available_handlers;
int: available_trolleys;
int: available_time;

array[OBJECTS] of var 0..available_time: start;

var 0..available_time: end;

constraint cumulative(start, duration, handlers, available_handlers);
constraint cumulative(start, duration, trolleys, available_trolleys);

constraint forall(o in OBJECTS)(start[o] +duration[o] <= end);

solve minimize end;

output [ "start = ", show(start), "\nend = ", show(end), "\n"];
```

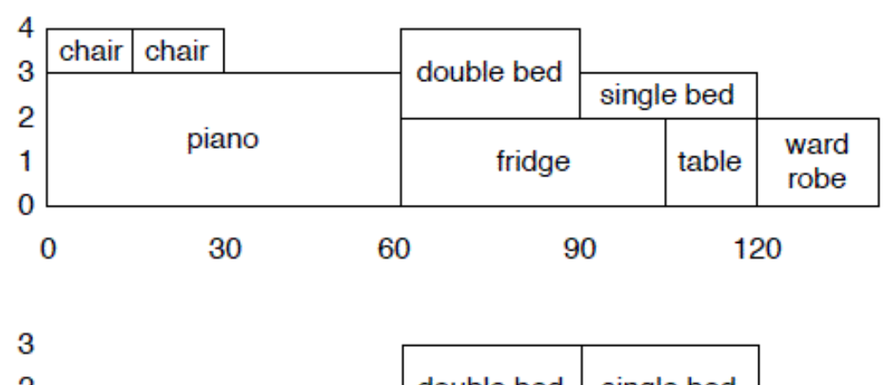
lo que por ejemplo con el siguiente fichero de parámetros:

```
n = 8;
% piano, fridge, double bed, single bed, wardrobe, chair, chair, table
duration = [60, 45, 30, 30, 20, 15, 15, 15];
handlers = [3, 2, 2, 1, 2, 1, 1, 2];
trolleys = [2, 1, 2, 2, 2, 0, 0, 1];
available_time = 180;
available_handlers = 4;
available_trolleys = 3;
```

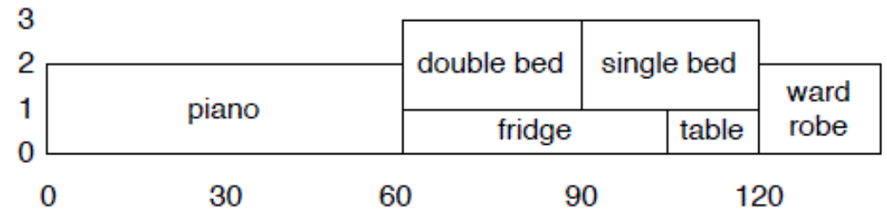
da el resultado:

```
D:\minizinc>minizinc mudanza.mzn mudanza.dzn
start = [50, 0, 110, 20, 0, 45, 30, 125]
end = 140
```

(a)



(b)



que corresponde con el cronograma donde a) son los operarios y b) los carritos:

5.1.3.- Predicado *table*

table trata de asegurar que cada elemento de una tabla toma un valor dentro una tupla de posibilidades. Tiene dos formatos:

Sintaxis:
table(array[int] of var bool: x, array[int, int] of bool: t)
table(array[int] of var int: x, array[int, int] of int: t)

Ejemplo: Queremos comprarnos dos coches. Para elegir tenemos entre unas cuantas posibilidades. Queremos que uno de los coches, que utilizaremos para ir al trabajo, tenga al menos 180 caballos, y que pase de 0 a 140 km/h en menos de 20 segundos. Para el otro coche, que utilizaremos los fines de semana, no tenemos ninguna limitación, pero no podemos gastarnos en total más de 50000 euros.

```
include "table.mzn";

%%% parametros
int: n;
int: numparam = 4;
set of int: COCHES=1..n;
set of int: ATRIBUTOS=1..numparam;

int: maxdinero;
int: mincaballos;
int: maxtiempo;

array[COCHES] of string: nombres;
array[COCHES,ATRIBUTOS] of int: datos;

%%% variables de decision
% coche para ir cada dia al trabajo
array[ATRIBUTOS] of var int:trabajo;

% coche para el fin de semana
array[ATRIBUTOS] of var int:findesemana;

% restricciones
constraint trabajo[4] + findesemana[4] <= maxdinero;
constraint trabajo[2] >= mincaballos;
constraint trabajo[3] < maxtiempo;
constraint trabajo[1] != findesemana[1];
constraint table(trabajo,datos);
constraint table(findesemana,datos);

solve satisfy;

output([show(nombres[fix(trabajo[1])])]++[": "]++
                [show(trabajo[i])++" " | i in 2..numparam]++["\n"]++
       [show(nombres[fix(findesemana[1])])]++[": "]++
                [show(findesemana[i])++" " | i in 2..numparam]
       );
```

junto con el fichero de datos:

```
% num de coches considerados
n = 6;

% total de dinero del que disponemos
maxdinero = 50000;

% minimos rpm requeridos para el coche de trabajo
mincaballos = 180;

% tiempo maximo en pasar de 0-140 (multiplicado x100)
maxtiempo = 2000;

% nombres de los coches
nombres = ["Alfa Romeo 147 GTA",
           "Audi A4 1.8",
```

```
            "Cadillac CTS 2.6",
            "Citroen C4 2.0 Hdi",
            "Ford Focus TDCI",
            "Renault Clio Sport 2.0"];

% datos de los coches

%           indice  caballos  0-140   precio
datos = [| 1,      250,      1171,  31500 % Alfa Romeo 147 GTA
         | 2,      125,      2403,  26800 % Audi A4 1.8
         | 3,      181,      1822,  36800 % Cadillac CTS 2.6
         | 4,      136,      1828,  21000 % Citroen C4 2.0 Hdi
         | 5,      115,      2244,  19000 % Ford Focus TDCI
         | 6,      182,      1450,  17000 % Renault Clio Sport 2.0
         |];
```

Ahora podemos teclear:

```
D:\minizinc>minizinc coches.mzn coches.dzn
Alfa Romeo 147 GTA: 250 1171 31500
Renault Clio Sport 2.0: 182 1450 17000
----------
```

Parece lógico que queramos saber si hay más posibilidades con estas restricciones.

```
D:\minizinc>minizinc --all-solutions coches.mzn coches.dzn
Alfa Romeo 147 GTA: 250 1171 31500
Renault Clio Sport 2.0: 182 1450 17000
----------
Renault Clio Sport 2.0: 182 1450 17000
Alfa Romeo 147 GTA: 250 1171 31500
----------
Renault Clio Sport 2.0: 182 1450 17000
Audi A4 1.8: 125 2403 26800
----------
Renault Clio Sport 2.0: 182 1450 17000
Citroen C4 2.0 Hdi: 136 1828 21000
----------
Renault Clio Sport 2.0: 182 1450 17000
Ford Focus TDCI: 115 2244 19000
----------
==========
```

5.1.4.- Predicado *regular*

Sintaxis:
regular(array[int] of var int: x, int: Q, int: S, array[int,int] of int: d, int: q0, set of int: F);

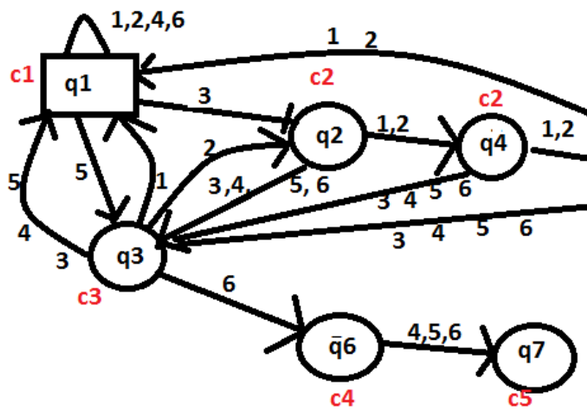
Restringe las variables en el array x (que deben estar definidas en el rango 1..S) a valores tales que formen una secuencia aceptable para el autómata finito determinista definido tal que:

- *Q* es su número de estados.
- Valores de entrada en el rango 1..*S*.
- Función de transición *d* (que establece una correspondencia entre [1..Q]x[1..S] a [0..Q]
- Estado inicial *q0* (valor en 1..Q)
- Estados de aceptación en F *q0* (valores en 1..Q)

Ejemplo: Tenemos un juego en el que la casilla 1 es la de inicio y la casilla 5 es la de llegada. Se juega con un dado con las siguientes reglas:

- Desde la casilla 1, con una tirada de 3 se pasa a la casilla 2, y con una tirada de 5 a la casilla 3. Con el resto de los valores la ficha no se mueve.
- De la casilla 2 se pasa a la 3 con valores 3,4,5,6. Pero si se sacan 3 veces seguidas 1 o 2 se vuelve al comienzo.
- De la casilla 3 se pasa a la 4 con 6, a la 2 con un 2 y a la 1 con el resto de valores.
- De la casilla 4 se pasa a la 5 con 4,5,6. Si sale un valor 1,2,3 el jugador debe abandonar la partida

Queremos obtener una secuencias de números ganadora con 5 tiradas. Para ello representamos las reglas del juego mediante un autómata finito determinista:



Y a partir de este autómata podemos escribir el programa:

```
include "regular.mzn";

int: Q = 7;
int: S = 6;
int: q0 = 1;
set of int: qf = {7};
set of int: ESTADOS = 1..Q;
set of int: DADO = 1..S;
array[ESTADOS,DADO] of int: d =
    [| 1,1,2,1,3,1 % q1
     | 4,4,3,3,3,3 % q2
     | 2,2,1,1,1,6 % q3
     | 5,5,3,3,3,3 % q4
     | 1,1,3,3,3,3 % q5
     | 0,0,0,7,7,7 % q6
     | 0,0,0,0,0,0 % q7
     |];
array[1..5] of var 1..S: s;
```

```
constraint regular(s,Q,S, d,q0,qf);

solve satisfy;

output[show(s)];
```

y lanzar el objetivo:

```
D:\minizinc>minizinc  dados.mzn
```

que devuelve todas una secuencia ganadoras con 5 tiradas.
Supongamos ahora que queremos determinar la tirada más baja que nos permitiría ganar en 5 tiradas.

```
include "regular.mzn";

int: Q = 7;
int: S = 6;
int: q0 = 1;
set of int: qf = {7};
set of int: ESTADOS = 1..Q;
set of int: DADO = 1..S;
array[ESTADOS,DADO] of int: d =
    [| 1,1,2,1,3,1 % q1
     | 4,4,3,3,3,3 % q2
     | 2,2,1,1,1,6 % q3
     | 5,5,3,3,3,3 % q4
     | 1,1,3,3,3,3 % q5
     | 0,0,0,7,7,7 % q6
     | 0,0,0,0,0,0 % q7
     |];
array[1..5] of var 1..S: s;

constraint regular(s,Q,S, d,q0,qf);

solve minimize sum(s);

output[show(s)];
```

con el resultado:

```
D:\minizinc>minizinc dadosmin.mzn
[1, 3, 3, 6, 4]
----------
==========
```

5.2.-Predicados de usuario

Sintaxis: `predicate nombre(tipo1:var1, ..., tipon:varn ) = restricción;`
Algunos comentarios:

- Se puede utilizar en cualquier lugar donde hay una restricción
- No se admiten predicados recursivos.
- También se pueden definir funciones, cuando no hay variables de decisión. En este caso se utiliza la palabra *test*.

Ejemplo `test even(int:x) = x mod 2 = 0;`

El siguiente ejemplo busca el punto de mayores coordenadas (x,y) que pertenezca a la intersección de dos rectángulos.

```
% rect. representado por x,y,lx,ly
array[1..4] of float: r1 = [20.0, 20.0,10.0,10.0];
array[1..4] of float: r2 = [25.0, 28.0,5.0,5.0];

var float:x;
var float:y;

predicate enElRectangulo(var float:x0, var float:y0, array[1..4] of float: r) =
      x0>=r[1] /\ x0<=r[1]+r[3] /\ y0>=r[2] /\ y0<=r[2]+r[4];

constraint (enElRectangulo(x,y,r1)) /\ (enElRectangulo(x,y,r2));

solve maximize (x+y);

output["x: ",show(x), ". y:",show(y)];
```

5.3.-Declaraciones locales

En cualquier expresión se pueden introducir variables locales, variables que solo existen en esa expresión. Esto se hace o bien para definir subexpresiones que se repiten (es decir sacar factor común) o simplemente para dar mayor claridad al programa. Para ello se utiliza la instrucción *let*:

Sintaxis: `let { var_dec1, . . . var_decn } in expr`
Algunos comentarios:

- Las declaraciones de variables deben corresponder a nombres nuevos.
- Se pueden definir tanto variables de decisión como parámetros, que deben estar inicializados.
- Estas variables solo existen en la expresión.
- Se pueden definir dentro de predicados siempre y cuando el predicado no se utilice en un contexto negativo.

Por ejemplo, en el problema del Sudoku en lugar de:

```
% cuadrados internos diferentes
constraint forall(i,j in 1..C)
        (alldifferent([sudoku[C*(i-1)+i2, C*(j-1)+j2] | i2,j2 in 1..C]));
```

podemos escribir:

```
% cuadrados internos diferentes
constraint forall(i,j in 1..C)
        (alldifferent([ let {int:x = C*(i-1)+i2, int:y= C*(j-1)+j2} in
                        sudoku[x, y] | i2,j2 in 1..C]));
```

```
var 1..3:a;
predicate bmenor(var int:a) =
```

```
 let {var 1..5:b} in b < a;

constraint bmenor(a);

solve satisfy;

output([show(a)]);
```


¿Qué soluciones mostrará MiniZinc al pedirle todas las soluciones para el código anterior?

5.4.-Reflexión

En programación se llama *reflexión* a la capacidad de un sistema para representar y analizar sus propios componentes o los de sistemas similares. En el caso de MiniZinc las posibilidades de reflexión son muy limitadas y se limitan a un conjunto de funciones predefinidas:

- Funciones para determinar el rango válido para un array: index_set(array 1-dim), index_set_1of2(array 2_dim), index_set_2of2(array 2-dim) y así para arrays de cualquier dimensión.
- *lb(x)*: cota inferior de los valores que puede tomar la variable de decisión *x*. Su extensión a arrays es *lb_array(t)* que devuelve una cota inferior para todas las variables de decisión contenidas en el array.
- *ub(x)*: cota superior de los valores que puede tomar la variable de decisión *x*. Su extensión a arrays es *ub_array(t)*.
- *dom(x)*: superconjunto de los posibles valores que puede tomar *x*. Su extensión a arrays is *dom_array*.

Ejemplo: Cota inferior de un array de variables de decisión

```
array [1..10] of var 5..20: t;
array [1..10] of var int  : s;

constraint forall(i in index_set(t))(s[i]>t[i]);

solve satisfy;

output( ["lb_array(s)=",show(lb_array(s))] );
```

Ejemplo: Ordenación de vectores

```
int:n=20;
array [1..n] of int:t = [9,8,7,1,2,3,6,5,4,10,9,8,7,1,2,3,6,5,4,10
                         ];
array [1..n] of var int:s;

predicate compatibles(array[int] of var int:a, array[int] of int:b) =
     assert(index_set(a) = index_set(b), "Error: los dos arrays deben estar definidos en el mismo rango");


predicate mismocardinal(int:x, array[int] of var int:a, array[int] of int:b) =
     let {var int:ca = sum([bool2int(a[i]=x) | i in index_set(b) ]),
          int:cb = sum([bool2int(b[i]=x) | i in index_set(b) ])} in ca=cb;

predicate mismoselementos(array[int] of var int:a, array[int] of int:b) =
          forall(i in b)(mismocardinal(b[i],a,b));

predicate ordenado(array[int] of var int:a) =
            let {set of int:I = index_set(a)} in
             forall(i in I)(if i>min(I) then a[i]>=a[i-1] else true endif);

constraint compatibles(s,t) /\ mismoselementos(s,t) /\ ordenado(s);
solve satisfy;

output([show(s)]);
```

El resultado:

```
D:\minizinc>minizinc sort.mzn
[1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10]
----------
```

6.- Reificación

La reificación en el ámbito de las restricciones consiste en convertir en operaciones aritméticas sobre 0's y 1's operaciones lógicas. No solo es una técnica muy útil para mejorar la eficiencia y en general para modelar problemas de restricciones, en MiniZinc es de uso obligado en algunas situaciones:

- No se pueden utilizar variables de decisión en instrucciones *if*.
- No se pueden utilizar variables de decisión como índices de arrays.

Ejemplo: Escribir un predicado que *cardinal* que reciba tres parámetros:

- Un array de variables de decisión de tipo entero *a*
- Un entero *x*
- Una variable de decisión de tipo entero *c*

El predicado debe tener éxito si *x* aparece en *a* *c* veces.

Una primera versión:

```
array [1..5] of var int:t;
var int:cuantas;

predicate cardinal(array[int] of var int:s, int:x, var int:c) =
```

```
   c = sum([ if s[i]==x then 1 else 0 endif; | i in index_set(s) ]);

constraint cardinal(t,5,3);

solve satisfy;

output[show(t)];
```

Solución:

```
array [1..5] of var int:t;
var int:cuantas;

%predicate logico_a_entero(var int:a, int:b, ) = (a=b -> true) /\ (a!=b -> );

predicate cardinal(array[int] of var int:s, int:x, var int:c) =
   c = sum([ bool2int(s[i],x) | i in index_set(s) ]);

constraint cardinal(t,5,3);

solve satisfy;

output[show(t)];
```

Ejemplo: Escribir un predicado *primeros* que reciba tres parámetros:

- Un array de variables de decisión de tipo entero *a*
- Un variable de decisión de tipo entero *x*
- Una variable de decisión de tipo entero *c*

El predicado debe tener éxito los primeros *x* valores en *a* contienen el número *c*.

Una primera versión:

```
array [1..5] of var int:t;
var int:cuantas;
var int:que;

predicate primeros(array[int] of var int:s, var int:x, var int:c) =
  forall(i in x)(s[i]=c);

constraint que=4 /\ cuantas=3 /\ primeros(t,cuantas,que);

solve satisfy;

output["t: "++show(t)++"cuantas: "++show(cuantas)++"que: "++show(que)];
```

dará error porque *x* es una variable de decisión y no se admite como índice ni como límite superior del subrango. La solución es hacer que el índice tome todos los valores del rango del array, comprobando la condición solo para los valores que tienen sentido en nuestro problema:

Solución:

```
array [1..5] of var int:t;
var int:cuantas;
var int:que;

predicate primeros(array[int] of var int:s, var int:x, var int:c) =
  x>=0 /\ x<=5 /\ forall(i in 1..5)(i<=x -> s[i]=c);


constraint que=4 /\ cuantas=3 /\ primeros(t,cuantas,que);

solve satisfy;

output["t: "++show(t)++"cuantas: "++show(cuantas)++"que: "++show(que)];
```

6.- Búsqueda

Internamente los resolutores (de dominios finitos) buscan métodos para restringir el valor de las variables. La idea es elegir una variable y una forma de restringir su dominio. Si al hacerlo el modelo queda inconsistente se explora otra posibilidad.
En MiniZinc podemos utilizar *anotaciones* para indicar al resolutor cómo realizar la búsqueda del modelo.
3 tipos de anotaciones:

- int_search(array[int] of var int, varchoice, constrainchoice, strategy)
- bool_search(array[int] of var bool, varchoice, constrainchoice, strategy)
- set search(array[int] of var set, varchoice, constrainchoice, strategy)

La variable *varchoice* puede tomar valores:

- *input_order*:en orden en el array)
- *first_fail*:dominio más pequeño)
- *smallest*:su dominio contiene el menor valor)

La variable *constrainchoice* puede tomar los valores:

- *indomain_min*:menor valor del dominio
- *indomain_max*:máximo valor del dominio
- *indomain_median*:mediana de los valores en el dominio
- *indomain_random*: al azar
- *indomain_split*: considera el domino y lo divide por la mitad, excluyendo la parte superior.

La variable *strategy* toma en la mayoría de los casos el valor *complete*. Un ejemplo típico es el de las n-reinas

```
% reinas
```

```
int:n=40;

array [1..n] of var 1..n: t;

% constraints

%fila
constraint forall (i in 1..n-1)(forall(j in i+1..n)(t[i]!=t[j]));

% dos diagonales
constraint forall (i in 1..n-1)(forall(j in i+1..n)(t[i]!=t[j]+(j-i)));
constraint forall (i in 1..n-1)(forall(j in i+1..n)(t[i]!=t[j]-(j-i)));

% probar la diferencia
%solve satisfy;
solve :: int_search(t, first_fail, indomain_min, complete) satisfy;

output[ show(t[i])++" " | i in 1..n];
```

Apéndice A: Para saber más

- La página principal de MiniZinc: http://www.g12.csse.unimelb.edu.au/minizinc/
- Tutorial de MiniZinc: http://www.g12.csse.unimelb.edu.au/minizinc/downloads/doc-1.6/minizinc-tute.pdf
- Artículo sobre el predicado *cumulative*:Explaining the cumulative propagator

---