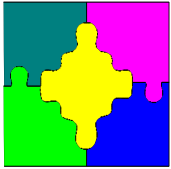


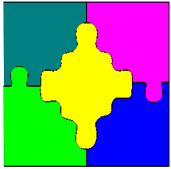
Modelling Constrained Optimization Problems

How can we formally describe a
constrained optimization problem in order
to solve it



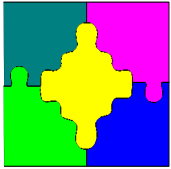
Overview

- Different approaches to modelling constrained optimization problems
- Basic modelling with MiniZinc
 - Structure of a model
 - Variables
 - Expressions
 - Constraints
- Advanced modelling with MiniZinc



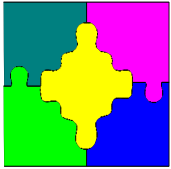
Modelling Problems

- As *constraint programmers* we need to create a
 - *conceptual model*: abstract the real-world problem to create a constraint problem which adequately *models* the problem and yet can be solved
 - *design model*: create a *program* which solves this constraint problem
- Typically this is an iterative process, requiring experimentation with
 - different techniques
 - different models
 - development of problem-specific heuristics
- This is a lot to do from scratch, what software can we use to help with this?



Main Approaches to Computer Modelling of Constraint Problems

- There are five *generic* approaches
 - *Traditional* language with constraint-solving library
 - *Object-oriented* language with high-level constraint solving library
 - *Constraint programming* language
 - *Mathematical modelling language*
 - *Embedded domain specific language*
- These vary in
 - how *high-level* they are, i.e. closeness to the application vs closeness to the computer architecture
 - how *expressive* they are
- In principle, they can all be used with different constraint-solving techniques but specific tools typically support only one or two techniques



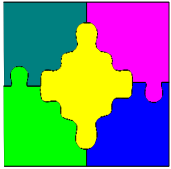
Comparative Example

- The *problem*:
 - A toy manufacturer must determine how many bicycles, B , and tricycles, T , to make in a 40 hr week given that
 - the factory can produce 200 bicycles per hour or 140 tricycles
 - the profit for a bicycle is \$25 and for a tricycle it is \$30
 - no more than 6,000 bicycles and 4,000 tricycles can be sold in a week
- The *model*:

Maximise $25B + 30T$

Subject to

$$(1/200)B + (1/140)T \leq 40 \wedge$$
$$0 \leq B \leq 6000 \wedge 0 \leq T \leq 4000$$



Approach 1: Traditional language with constraint-solving library

Using C and the CPLEX Callable Library we have

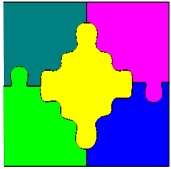
```
#include "cplex.h"
char probname[]="toy_problem";
int numcols=2; /*number of variables*/
int numrows=1; /*number of constraints*/
int objsense= -1; /*maximisation problem*/
double obj[] = [25,30]; /*objective function*/
double rhs[] = [40]; /*RHS constant of constraints*/
char sense[] = ['L']; /*type of constraint operator*/
int matbeg[] = [0,1]; /*constraint coefficients stored as a sparse array by */
int matcnt[] = [1,1]; /* column. */
int matind[] = [0,0];
double matval[] = [1.0/200.0,1.0/140.0];
double lb[] = [0,0]; /* variable lower bounds*/
double ub[] = [6000,4000]; /* variable upper bounds*/
double rngval[] = []; /* range constraints */
```

Maximise $25B + 30T$

Subject to

$(1/200)B + (1/140)T \leq 40 \wedge$

$0 \leq B \leq 6000 \wedge 0 \leq T \leq 4000$



Approach 1: Traditional language with constraint-solving library

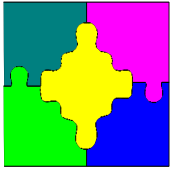
```
int solstat; /* solution status */  
double v[2]; /*variable values*/  
double slack[1]; /*slack variable*/
```

```
double objval; /*objective value*/  
double pi[1]; /*dual variables*/  
double dj[2]; /* reduced costs*/
```

```
/* CPLEX library stuff */  
CPXENVptr env = NULL;  
CPXLPptr lp = NULL;  
int status;
```

```
/* initialise environment*/  
env = CPXopenCPLEXdevelop(&status);  
if (env==NULL) then return(1);
```

```
/* create problem */  
lp = CPXcreateprob(env, &status, probname);  
if (lp==NULL) then return(1);  
status = CPXcopylp(env,lp, numvols, numrows, objsen, obj, rhs, sense,  
                  matbeg, matcnt, matind, matval, lb, ub, rngval);  
if (status) then return(1);
```



Approach 1: Traditional language with constraint-solving library

```
/* solve problem & get solution*/
```

```
status = CPXprimopt(env,lp);
```

```
if (status) then return(1);
```

```
status = CPXsolution(env, lp, &solstat, &objval, x, pi, slack, dj);
```

```
if (status) then return(1);
```

```
/* write answer to answer to screen */
```

```
printf("Solution value = %f\n\n", objval)'
```

```
...
```

```
/* terminate */
```

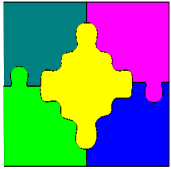
```
status = CPXfreeprob(env, &lp);
```

```
if (status) then return(1);
```

```
status = CPXcloseCPLEX(&env);
```

```
if (status) then return(1);
```

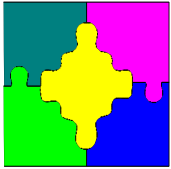
```
return 0;
```

Approach 2: OO language with HL constraint-solving library

- Using hypothetical nice C++ CPLEX interface
Maximise $25B + 30T$
Subject to
 $(1/200)B + (1/140)T \leq 40 \wedge$
 $0 \leq B \leq 6000 \wedge 0 \leq T \leq 4000$

```
#include <floatsolver.h>
CPLEXFloatSolver s();
Float B(s,0.0,6000.0), T(s,0.0,4000.0);
s.AddConstraint( (1.0/200.0)*B + (1.0/140.0)*T
    =< 40.0);
s.MaxObjective(25.0*B+30.0*T);
s.Solve();
cout << "B =" << B.val() << "T=" << T.val();
```
- Note that this makes use of C++ overloading



Approach 3: Constraint Programming Language

- Using ECLiPSe

```
:- lib(eplex).
```

```
:- eplex_instance(prob).
```

```
main(B, T) :-
```

```
    prob: ([B] $:: 0.0..6000.0),
```

```
    prob: ([T] $:: 0.0..4000.0),
```

```
    prob: ((1.0/200.0)*B+(1.0/140.0)*T $=< 40.0),
```

```
    prob: eplex_solver_setup(min(25*B+30*T)),
```

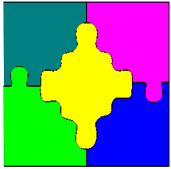
```
    prob: eplex_solve(_Cost).
```

Maximise $25B + 30T$

Subject to

$(1/200)B + (1/140)T \leq 40 \wedge$

$0 \leq B \leq 6000 \wedge 0 \leq T \leq 4000$



Approach 3: Constraint Programming Language

- Using ECLiPSe
:- lib(eplex).

main(B, T) :-

[B] \$:: 0.0..6000.0,

[T] \$:: 0.0..4000.0,

(1.0/200.0)*B+(1.0/140.0)*T \$=< 40.0,

eplex_solver_setup(min(25*B+30*T),

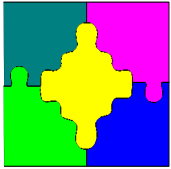
eplex_solve(_Cost).

Maximise $25B + 30T$

Subject to

$(1/200)B + (1/140)T \leq 40 \wedge$

$0 \leq B \leq 6000 \wedge 0 \leq T \leq 4000$



Approach 4: Mathematical Modelling Language

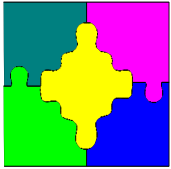
Maximise $25B + 30T$

Subject to

$(1/200)B + (1/140)T \leq 40 \wedge$

$0 \leq B \leq 6000 \wedge 0 \leq T \leq 4000$

- Using AMPL
var B; var T;
maximize profit: 25*B+30*T;
subject to: (1/200)*B+(1/140)*T <= 40;
subject to: 0 <= B <= 6000;
subject to: 0 <= T <= 4000;
solve;



Approach 5: Embedded DSL

Maximise $25B + 30T$

Subject to

$(1/200)B + (1/140)T \leq 40 \wedge$

$0 \leq B \leq 6000 \wedge 0 \leq T \leq 4000$

- NumberJack in Python

def smodel:

```
    B = Variable(0,6000)
```

```
    T = Variable(0,4000)
```

```
    model = Model(
```

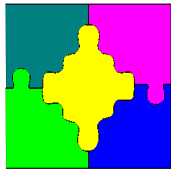
```
        Minimise( 25*B + 30 * T ),
```

```
        [(1/200) * B + (1/140) * T <= 40]
```

```
    )
```

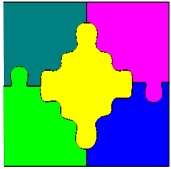
```
solver = smodel.model.load('Mistral')
```

```
solver.solve()
```



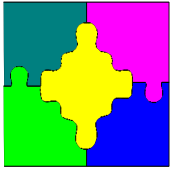
(Relatively) Recent Examples

	<i>CLP</i>	<i>C++ toolkit</i>	<i>Modelling language</i>	
<i>LP/ILP</i>	<i>ECLiPSe</i>	Mistral	OPL	MiniZinc
<i>BT search + propagation</i>		ILOG Solver		
<i>Local search</i>		Localizer ++	Comet	



MiniZinc

- **MiniZinc** is a new modelling language being developed by NICTA with Univ of Melb/Monash.
- Depending on the kind of model it can be solved with constraint programming or with MIP techniques.
- It is a subset of the more powerful modelling language Zinc — first public release 2010.



A First MiniZinc Model

Maximise $25B + 30T$

Subject to

$(1/200)B + (1/140)T \leq 40 \wedge$

$0 \leq B \leq 6000 \wedge 0 \leq T \leq 4000$

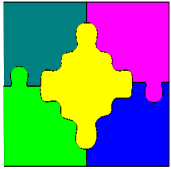
```
var 0.0..6000.0: B;
```

```
var 0.0..4000.0: T;
```

```
constraint (1.0/200.0)*B+(1.0/140.0)*T <= 40.0;
```

```
solve maximize 25.0*B + 30.0*T;
```

```
output ["B = ", show(B), "T = ", show(T), "\n"];
```

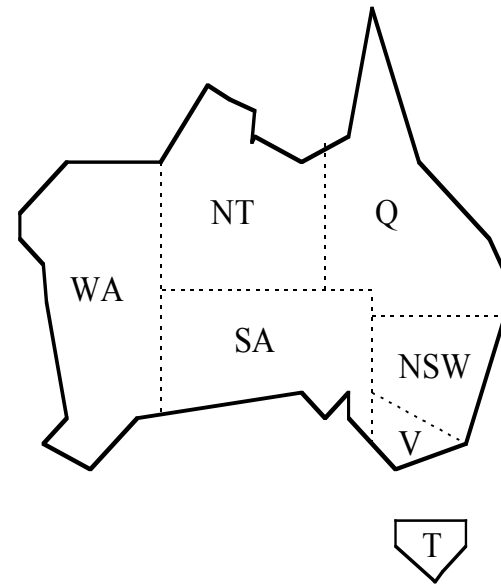



A Second MiniZinc Model

```
% Colouring Australia using  
int: nc = 3;
```

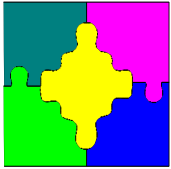
```
var 1..nc: wa;   var 1..nc: nt;  
var 1..nc: sa;   var 1..nc: q;  
var 1..nc: nsw; var 1..nc: v;  
var 1..nc: t;
```

```
constraint wa != nt;  
constraint wa != sa;  
constraint nt != sa;  
constraint nt != q;  
constraint sa != q;  
constraint sa != nsw;  
constraint sa != v;  
constraint q != nsw;  
constraint nsw != v;
```



```
solve satisfy;
```

```
output ["wa=", show(wa), "\t nt=",  
        show(nt), "\t sa=", show(sa), "\n",  
        "q=", show(q), "\t nsw=", show  
        (nsw), "\t v=", show(v), "\n",  
        "t=", show(t), "\n"];
```



A Second MiniZinc Model

- We can run our MiniZinc model as follows

```
$ mzn aust.mzn
```

- This results in

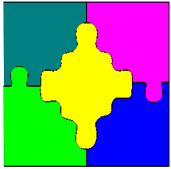
```
wa=1      nt=3      sa=2
```

```
q=1      nsw=3     v=1
```

```
t=1
```

```
-----
```

- MiniZinc models must end in .mzn
- There is also an eclipse IDE for MiniZinc



Parameters

In MiniZinc there are two kinds of variables:

Parameters-These are like variables in a standard programming language. They must be assigned a value (but only one).

They are declared with a type (or a range/set).

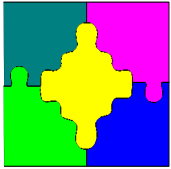
You can use **par** but this is optional

The following are logically equivalent

```
int: i=3;
```

```
par int: i=3;
```

```
int: i;   i=3;
```



Decision Variables

Decision variables-These are like variables in mathematics. They are declared with a type and the **var** keyword. Their value is computed by MiniZinc so that they satisfy the model.

Typically they are declared using a **range** or a **set** rather than a type name

The range or set gives the domain for the variable.

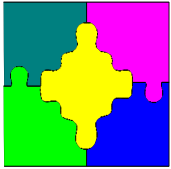
The following are logically equivalent

```
var int: i; constraint i >= 0; constraint i <= 4;
```

```
var 0..4: i;
```

```
var {0,1,2,3,4}: i;
```

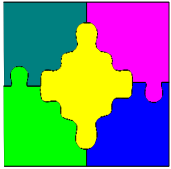
Question: what does this mean **constraint i = i + 1;**



Types

Allowed types for variables are

- Integer `int` or range `1..n` or set of integers
- Floating point number `float` or range `1.0..f` or set of floats
- Boolean `bool`
- Strings `string` (but these cannot be decision variables)
- Arrays
- Sets

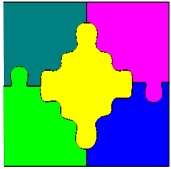


Instantiations

Variables have an **instantiation** which specifies if they are parameters or decision variables.

The type + instantiation is called the **type-inst**.

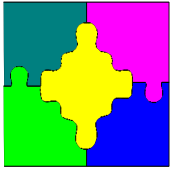
MiniZinc errors are often couched in terms of mismatched type-insts...



Strings

Strings are provided for output

- An output item has form
output <list of strings>;
- String literals are like those in C: enclosed in “ ”
- They cannot extend across more than one line
- Backslash for special characters \n \t etc
- Built in functions are
 - show(v)
 - “house”++”boat” for string concatenation



Arithmetic Expressions

MiniZinc provides the standard arithmetic operations

- Floats: `*` `/` `+` `-`
- Integers: `*` `div` `mod` `+` `-`

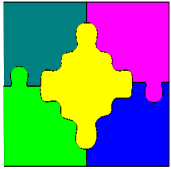
Integer and float literals are like those in C

There is no automatic coercion from integers to floats

The builtin `int2float(intexp)` must be used to explicitly coerce them

The arithmetic relational operators are

`==` `!=` `>` `<` `>=` `<=`



Data files

Here is a simple model about loans:

```
% variables
var float: R; % quarterly repayment
var float: P; % principal initially borrowed
var 0.0 .. 100.0: I; % interest rate
% intermediate variables
var float: B1; % balance after one quarter
var float: B2; % balance after two quarters
var float: B3; % balance after three quarters
var float: B4; % balance at end

constraint B1 = P * (1.0 + I) - R;
constraint B2 = B1 * (1.0 + I) - R;
constraint B3 = B2 * (1.0 + I) - R;
constraint B4 = B3 * (1.0 + I) - R;

solve satisfy;

output ...
```

We want this to be **generic** in the choice of values R, P, I, B1, ..., B4. MiniZinc allows parameters and variables to be initialized in a separate **data file**

- left: borrowing 1000\$ at 4% repaying \$260
- right: borrowing 1000\$ at 4% owing nothing at end

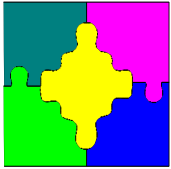
```
I = 0.04;
P = 1000.0;
R = 260.0;

I = 0.04;
P = 1000.0;
B4 = 0.0;
```

We can run a MiniZinc model with a data file as follows

```
$ mzn -b mip loan.mzn loan1.dzn
```

MiniZinc data files must end in .dzn



Basic Structure of a Model

A MiniZinc model is a sequence of items

The order of items does not matter

The kinds of items are

- An inclusion item

`include <filename (which is a string literal)>;`

- An output item

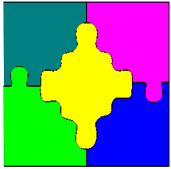
`output <list of string expressions>;`

- A variable declaration

- A variable assignment

- A constraint

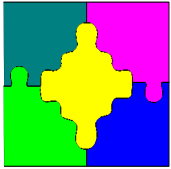
`constraint <Boolean expression>;`



Basic Structure of a Model

The kinds of items (cont.)

- A solve item (a model must have exactly one of these)
 - solve satisfy;
 - solve maximize <arith. expression>;
 - solve minimize <arith. expression>;
 - Predicate and test items
 - Annotation items
-
- Identifiers in MiniZinc start with a letter followed by other letters, underscores or digits
 - In addition, the underscore `_' is the name for an anonymous decision variable



Exercise

We want to bake some cakes for a fete for school.

Banana cake

250g of self-raising flour,
2 mashed bananas,
75g sugar and
100g of butter

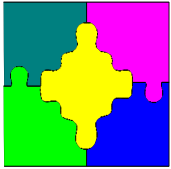
Chocolate cake

200g cups of self-raising flour,
75g of cocoa,
150g sugar and
150g of butter.

(WARNING: please don't use these recipes at home).

We have 4kg self-raising flour, 6 bananas, 2kg of sugar, 500g of butter and 500g of cocoa.

Exercise: Write a MiniZinc model to determine how many of each sort of cake should we make to maximize the profit where a chocolate cake sells for \$4.50 and a banana cake for \$4.00.

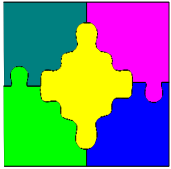


Production Planning Example

A problem with this model is that the recipes and the available ingredients are hard wired into the model.

It is an example of simple kind of **production planning problem** in which we wish to

- determine how much of each kind of product to make to maximize the profit where
- manufacturing a product consumes varying amounts of some fixed resources.
- We can use a generic MiniZinc model to handle this kind of problem.



Example Using Arrays & Sets

```
% Number of different products
int: nproducts;
set of int: products = 1..nproducts;
```

```
%profit per unit for each product
array[products] of int: profit;
```

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

```
%amount of each resource available
array[resources] of int: capacity;
```

```
%units of each resource required to produce 1 unit of
    product
array[products, resources] of int: 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]));
```

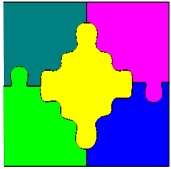
```
% Variables: how much should we make of each product
array[products] of var 0..mproducts: produce;
```

```
% Production cannot use more than the available
    resources:
constraint forall (r in resources) (
    sum (p in products) (consumption[p, r] *
    produce[p]) <= capacity[r]
);
```

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

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

MiniZinc supports **arrays** and **sets**.



Sets

Sets are declared by

set of type

They are only allowed to contain integers, floats or Booleans.

Set expressions:

Set literals are of form $\{e_1, \dots, e_n\}$

Integer or float ranges are also sets

Standard set operators are provided:

in, union, intersect, subset, superset, diff, symdiff

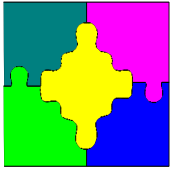
The size of the set is given by `card`

Some examples:

set of int: `products = 1..nproducts;`

`{1,2} union {3,4}`

Set variable names, set literals or ranges can be used as types.



Arrays

An array can be multi-dimensional. It is declared by

array[index_set 1, index_set 2, ...,] of type

The index set of an array needs to be

an integer range or

the name of a set variable that is an integer range.

The elements in an array can be anything except another array

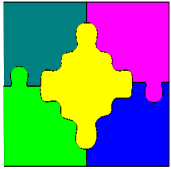
They can be decision variables.

For example

`array[products, resources] of int: consumption;`

`array[products] of var 0..mproducts: produce;`

The built-in function **length** returns the number of elements in a
1-D array



Arrays (Cont.)

1-D arrays are initialized using a list

```
profit = [400, 450];
```

```
capacity = [4000, 6, 2000, 500, 500];
```

2-D array initialization uses a list with ``|`` separating rows

```
consumption= [| 250, 2, 75, 100, 0,  
               | 200, 0, 150, 150, 75 |];
```

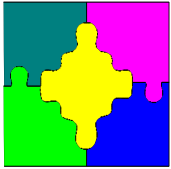
Arrays of *any* dimension ($well \leq 3$) can be initialized from a list using the

`arraynd` family of functions:

```
consumption= array2d(1..2,1..5, [250,2,75,100,0,200,0,150,150,75]);
```

The concatenation operator `++` can be used with 1-D arrays:

```
profit = [400]++[450];
```



Array & Set Comprehensions

MiniZinc provides *comprehensions* (similar to ML)

A set comprehension has form

$\{ \text{expr} \mid \text{generator 1, generator 2, ...} \}$

$\{ \text{expr} \mid \text{generator 1, generator 2, ... where bool-expr} \}$

An array comprehension is similar

$[\text{expr} \mid \text{generator 1, generator 2, ...}]$

$[\text{expr} \mid \text{generator 1, generator 2, ... where bool-expr}]$

Some examples

$\{i + j \mid i, j \text{ in } 1..3 \text{ where } j < i\} = \{1 + 2, 1 + 3, 2 + 3\} = \{3, 4, 5\}$

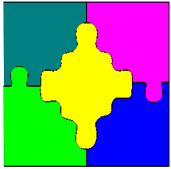
Exercise: What does $b = ?$

set of int: cols = 1..5;

set of int: rows = 1..2;

array [rows,cols] of int: c= [| 250, 2, 75, 100, 0, | 200, 0, 150, 150, 75 |];

$b = \text{array2d}(\text{cols}, \text{rows}, [a[j], i] \mid i \text{ in cols}, j \text{ in rows});$



Iteration

MiniZinc provides a variety of built-in functions for iterating over a list or set:

- Lists of numbers: `sum`, `product`, `min`, `max`
- Lists of constraints: `forall`, `exists`

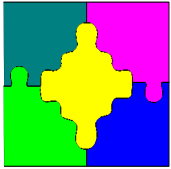
MiniZinc provides a special syntax for calls to these (and other **generator** functions)

For example,

```
forall (i, j in 1..10 where i < j) (a[i] != a[j]);
```

is equivalent to

```
forall ([a[i] != a[j] | i, j in 1..10 where i < j]);
```



Data files

The simple production model is **generic** in the choice of parameter values. MiniZinc allows parameters to be initialized in a separate **data file**

```
% Data file for simple production planning model
```

```
nproducts = 2; %banana cakes and chocolate cakes
```

```
profit = [400, 450]; %in cents
```

```
nresources = 5; %flour, banana, sugar, butter cocoa
```

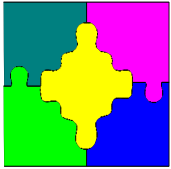
```
capacity = [4000, 6, 2000, 500, 500];
```

```
consumption= [| 250, 2, 75, 100, 0,  
                | 200, 0, 150, 150, 75 |];
```

We can run a MiniZinc model with a data file as follows

```
$ mzn prod.mzn cake.dzn
```

MiniZinc data files must end in **.dzn**



Assertions

Defensive programming requires that we check that the data values are **valid**.

The built-in Boolean function **assert(boolexp,stringexp)** is designed for this.

It returns true if **boolexp** holds, otherwise prints **stringexp** and aborts

Like any other Boolean expression it can be used in a constraint item

For example,

```
int: nresources;
```

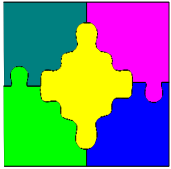
```
constraint assert(nresources > 0, "Error: nresources =< 0");
```

```
array[resources] of int: capacity;
```

```
constraint assert( forall(r in resources)(resources[r] >= 0), "Error: negative capacity");
```

Exercise: Write an expression to ensure consumption is non-negative

```
array[products, resources] of int: consumption;
```



Assertions for Debugging

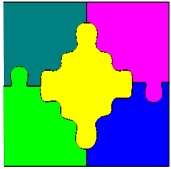
- You can (ab)use assertions to help debug

- ```
int: n = 5;
array[1..n] of var 1..n: a;
array[1..n] of 1..n: b = [3,5,2,3,1];
```

```
constraint forall(j in 1..n, i in b[n-j]..b[n-j])(a[j] < i);
```

- Error message

```
error:
debug.mzn:5
In constraint.
In 'forall' expression.
In comprehension.
j = 5
In comprehension head.
In '..' expression
In array access.
In index argument 1
Index out of range.
```



# Assertions for Debugging

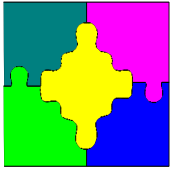
- You can (ab)use assertions to help debug

- ```
int: n = 5;  
array[1..n] of var 1..n: a;  
array[1..n] of 1..n: b = [3,5,2,3,1];
```

```
constraint forall(j in 1..n)(  
    assert(n-j in 1..n, "b[" ++ show(n-j) ++ "]"));
```

- Error message

```
error:  
debug.mzn:6  
In constraint.  
In 'forall' expression.  
In comprehension.  
j = 5  
In comprehension head.  
In 'assert' expression.  
Assertion failure: "b[0]"
```



Beware out of range errors in constraints

- You can (ab)use assertions to help debug

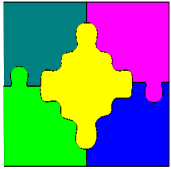
- ```
int: n = 5;
array[1..n] of var 1..n: a;
array[1..n] of 1..n: b = [3,5,2,3,1];
```

```
constraint forall(j in 1..n)(a[j] < b[n-j]);
```

- Error message

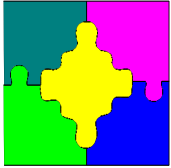
```
error:
debug.mzn:5
In constraint.
In 'forall' expression.
Model inconsistency detected.
```





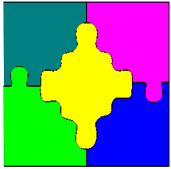
# If-then-else

- MiniZinc provides an  
if <boolexp> then <exp> else <exp> endif  
expression
- For example,  
if  $y \neq 0$  then  $x / y$  else 0 endif
- The Boolean expression is not allowed to contain decision variables, only parameters
- In output items the built-in function **fix** checks that the value of a decision variable is fixed and coerces the instantiation from decision variable to parameter



# Constraints

- Constraints are the core of the MiniZinc model
- We have seen simple relational expressions but constraints can be considerably more powerful than this.
- A constraint is allowed to be any Boolean expression
- The Boolean literals are  
true and false  
and the Boolean operators are  
 $\wedge$     $\vee$     $\leftarrow$     $\rightarrow$     $\leftrightarrow$    not
- Global constraints: alldifferent



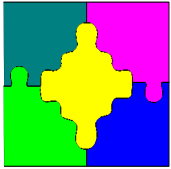
# Complex Constraint Example

Imagine a scheduling problem in which we have a set of tasks that use the same single resource

Let  $\text{start}[i]$  and  $\text{duration}[i]$  give the start time and duration of task  $i$

To ensure that the tasks do not overlap

constraint forall (i,j in tasks where  $i \neq j$ ) (  
     $\text{start}[i] + \text{duration}[i] \leq \text{start}[j] \ \vee$   
     $\text{start}[j] + \text{duration}[j] \leq \text{start}[i]$  );



# Array Constraints

Recall that array access is given by  $a[i]$ .

The index  $i$  is allowed to be an expression involving decision variables in which case it is an implicit constraint on the array.

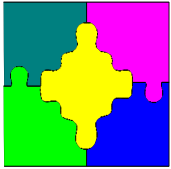
As an example consider the [stable marriage problem](#).

We have  $n$  (straight) women and  $n$  (straight) men.

Each man has a ranked list of women and vice versa

We want to find a husband/wife for each women/man s.t all marriages are stable, i.e.,

- Whenever  $m$  prefers another women  $o$  to his wife  $w$ ,  $o$  prefers her husband to  $m$
- Whenever  $w$  prefers another man  $o$  to her husband  $m$ ,  $o$  prefers his wife to  $m$



# Stable Marriage Problem

```
int: n;
```

```
array[1..n,1..n] of int: rankWomen;
```

```
array[1..n,1..n] of int: rankMen;
```

```
array[1..n] of var 1..n: wife;
```

```
array[1..n] of var 1..n: husband;
```

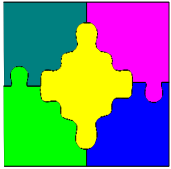
```
constraint forall (m in 1..n) (husband[wife[m]]=m);
```

```
constraint forall (w in 1..n) (wife[husband[w]]=w);
```

**Exercise:** insert stability constraints here...

```
solve satisfy;
```

```
output ["wives= ", show(wife),"\n", "husbands= ", show(husband)];
```



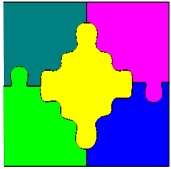
# Higher-order constraints

- The built-in coercion function `bool2int` allows the modeller to use so called `higher order` constraints:
- **Magic series problem**: find a list of numbers  $S = [s_0, \dots, s_{n-1}]$  s.t.  $s_i$  is the number of occurrences of  $i$  in  $S$ .
- A MiniZinc model is

```
int: n;
array[0..n-1] of var 0..n: s;

constraint
 forall(i in 0..n-1) (
 s[i] = sum(j in 0..n-1)(bool2int(s[j]=i));

 solve satisfy;
```



# Set Constraints

- MiniZinc allows sets over integers to be decision variables
- Consider the 0/1 knapsack problem

```
int: n;
```

```
int: capacity;
```

```
array[1..n] of int: profits;
```

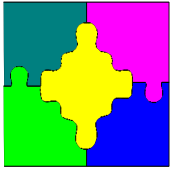
```
array[1..n] of int: weights;
```

```
var set of 1..n: knapsack;
```

```
constraint sum (i in knapsack) (weights[i]) <= capacity;
```

```
solve maximize sum (i in knapsack) (profits[i]) ;
```

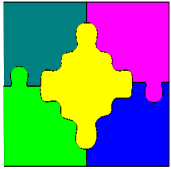
```
output [show(knapsack)];
```



# Set Constraints (Cont.)

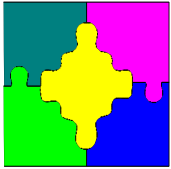
- But this doesn't work—we can't iterate over variable sets
- **Exercise:** Rewrite the example so that it doesn't iterate over a var set





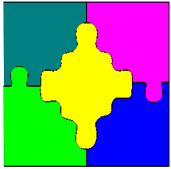
# Enumerated Types

- Enumerated types are useful to name classes of object which we will decide about. In reality they are placeholders for integers
- `enum people = { bob, ted, carol, alice };`
- This can be imitated by
- `set of int: people = 1..4;`  
`int: bob = 1;`  
`int: ted = 2;`  
`int: carol = 3;`  
`int: alice = 4;`  
`array[people] of string: name =`  
`["bob", "ted", "carol", "alice"];`



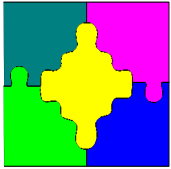
# Using MiniZinc

- Getting MiniZinc
  - <http://www.g12.cs.mu.oz.au/minizinc>
- On-line resources
  - <http://www.g12.csse.unimelb.edu.au/wiki>
- Or use installed version
  - [slippers2.csse.unimelb.edu.au](http://slippers2.csse.unimelb.edu.au)
    - username preferred\_login password (same please change)
  - path to mzn: /Users/Shared/g12/



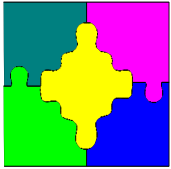
# How does MiniZinc work

- MiniZinc interprets the model and data and spits out a simpler form of model: FlatZinc
  - The tool `mzn2fzn` explicitly does this step.
  - `mzn2fzn file.mzn data.dzn`
    - creates `file.fzn`
  - FlatZinc interpreters run FlatZinc files
    - very simple output (just some variable values)
  - MiniZinc reads the simple output and calculates the complex output



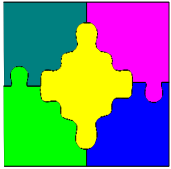
# MiniZinc and Mzn

- MiniZinc is a standalone minizinc interpreter
  - older
  - more stable
  - fixed to FD solver
- Mzn is a script using mzn2fzn/flatzinc
  - Uses mzn2fzn to convert MiniZinc to FlatZinc
  - Runs FlatZinc interpreter
  - Takes output of FlatZinc and pipes to MiniZinc to get output
  - Less stable, links to any FlatZinc solver, supported



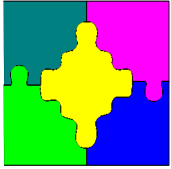
# Summary

- Four main approaches to modelling & solving constraint problems
  - *Traditional* language with constraint-solving library
  - *Object-oriented* language with high-level constraint solving library
  - *Constraint programming* language
  - *Mathematical modelling language*
  - *Embedded domain specific language*
- We have looked at basic modelling with the mathematical modelling language MiniZinc in some detail
  - What is good about MiniZinc?
  - What is bad?
- In the workshop we will use MiniZinc to model some problems—if you have a laptop please bring it along with MiniZinc installed.



# Exercise 1: Magic Square

- A magic square of side  $n$  is an arrangement of the numbers from 1 to  $n*n$  such that each row, column, and major diagonal all sum to the same value.
- Here is a 3×3 magic square:  
2 7 6  
9 5 1  
4 3 8
- **Exercise:** Write a MiniZinc program to generate a magic square for size  $n$



## Exercise 2: Task Allocation

- We have
  - a set of tasks,  $\mathcal{T}$
  - a set of workers,  $\mathcal{W}$
  - a set of tasks for each worker that they are qualified to perform
  - a cost for each worker
- **Exercise:** Write a MiniZinc program to find the set of workers which can complete all tasks and which minimizes the cost