# Topic 2: Basic Modelling [1]
## (Version of 23rd October 2023)

Pierre Flener and Jean-Noël Monette

Optimisation Group
Department of Information Technology
Uppsala University
Sweden

Course 1DL442:
Combinatorial Optimisation and Constraint Programming,
whose part 1 is Course 1DL451:
Modelling for Combinatorial Optimisation

[1] Many thanks to Guido Tack for feedback

# Outline

# Outline

# MiniZinc Model

A MiniZinc model may comprise the following items:

- Parameter declarations
- Decision variable declarations
- Predicate and function definitions
- Constraints
- Objective
- Output

# Types for Parameters

MiniZinc is strongly typed. Some parameter types are:

- `int`: integer
- `bool`: Boolean
- `enum`: enumeration
- `float`: floating-point number
- `string`: string of characters
- `set of` $\tau$: set of elements of type $\tau$
- `array[`$\rho$`] of` $\tau$: possibly multidimensional array of elements of type $\tau$; each index range in $\rho$ is an enumeration or an integer range $\alpha$`..`$\beta$

## Example

The parameter declaration `int: n` declares an integer parameter called `n`. One can also write `par int: n` in order to emphasise that `n` is a parameter.

# Types for Decision Variables

Decision variables are implicitly existentially quantified: the objective is to find feasible (and optimal) values in their finite domains. Some variable types are:

- `int`: integer
- `bool`: Boolean
- `enum`: enumeration
- `float`: floating-point number (do not use in this course)
- `set of enum` and `set of int`: set

A possibly multidimensional `array` can be declared to have variables of any variable type, but it is itself not a variable.

**Example**

The variable declaration `var int: n` declares a decision variable of domain `int` and identifier `n`.

Tight domains for variables might accelerate the solving: see the next slides.

# Literals

The following literals (or: constants) can be used:

- Boolean: `true` and `false`
- Integers: in decimal, hexadecimal, or octal format
- Sets: between curly braces, for example `{1,3,5}`, or as integer ranges, for example `10..30`
- 1d arrays: between square brackets, say `[6,3,1,7]`
- 2d arrays: A vertical bar `|` is used before the first row, between rows, and after the last row; for example `[|11,12,13|21,22,23|]`
- For higher-dimensional arrays, see slide 11

Careful: The indices of arrays start from `1` by default.

# Declarations of Parameters and Decision Variables

```
1 int: n = 4;
2 par int: p;
3 p = 10;
4 var 0..23: hour;
5 set of int: Primes = {2,3,5,7,11,13};
6 var set of Primes: Taken;
7 var int: nbr = card(Taken);
```

- A parameter must be instantiated, *once*, to a literal. The instantiation of a parameter can be separate from its declaration: either in the model (see `p` in lines 2 & 3 above), or in a datafile, or at the command line, or in the IDE.
- The domain of a decision variable can be tightened by replacing its type by a smaller finite set of values of that type:
  - `hour` must take an integer value from `0` to `23` inclusive
  - `Taken` must be a subset of `{2,3,5,7,11,13}`
  - `nbr` is equality-constrained at its declaration:
    its domain is inferred to be `0..6` (see slide 39 for more)

# Array and Set Comprehensions

An array or set can be built by a comprehension, using the notation $[\sigma \mid \gamma]$ or $\{\sigma \mid \gamma\}$, where expression $\sigma$ is evaluated for each element produced by the generator $\gamma$: a generator introduces one or more identifiers with values drawn from finite integer sets, optionally under a `where` test.

## Examples

```
[i * 2 | i in 1..8]
      evaluates to [2,4,6,8,10,12,14,16]
[i * j | i,j in 1..3 where i<j]  % both i and j are in 1..3
      evaluates to [2,3,6]
[i + 2 * j | i in 1..3, j in 1..4]
      evaluates to [3,5,7,9,4,6,8,10,5,7,9,11]
{i + 2 * j | i in 1..3, j in 1..4}
      evaluates to {3,4,5,6,7,8,9,10,11}
Sudoku[row,..]  % slicing
      is syntactic sugar for [Sudoku[row,col] | col in 1..9]
```

# Indexing: Syntactic Sugar

For example,

```
sum(i,j in 1..n where i<j)(X[i] * X[j])
```

is syntactic sugar for

```
sum([X[i] * X[j] | i,j in 1..n where i<j])
```

This works for any function or predicate that takes an array as sole argument. In particular:

```
forall(i in 1..n)(Z[i] = X[i] + Y[i]);
```

is syntactic sugar for

```
forall([Z[i] = X[i] + Y[i] | i in 1..n]);
```

where the `forall(array[int] of var bool: B)` constraint holds if and only if (iff) all the expressions in the Boolean array `B` hold: it generalises the 2-ary logical-and connective (`/\`).

# Array Manipulation

- Changing the number of dimensions and their index ranges, provided the numbers of elements match:

  `array1d(5..10,[|2,7|3,7|4,9|])` casts a 2D array into a 1D array

  `array2d(1..2,1..3,[2,7,3,7,4,9])` casts a 1D array into 2D

  and so on, until `array6d`.

  Try and keep your index ranges starting from `1`:

  - It is easier to read a model under this usual convention.
  - Subtle errors might occur otherwise.

- Concatenation: for example, `[1,2] ++ [3,4]`.

# Subtyping

A parameter can be used wherever a decision variable is expected. This extends to arrays: for example, a predicate or function expecting an argument of type `array[int] of var int` can be passed an argument of type `array[int] of int`.

The type `bool` is a subtype of the type `int`. One can coerce from `bool` to `int` using the `bool2int` function, defined by `bool2int(true) = 1` and `bool2int(false) = 0`. This coercion is automatic when needed.

In mathematics, one uses the Iverson bracket for this purpose: we define $[\phi] = 1$ if and only if formula $\phi$ is true, and $[\phi] = 0$ otherwise.

# Option Variables

An option variable is a decision variable that can also take the special value `<>` indicating the absence of a value for the decision variable.

A decision variable is declared optional with the keyword `opt`.

For example, `var opt 1..4: x` declares a decision variable `x` of domain `{1,2,3,4,<>}`.

Do not use *explicit* option variables in this course.
However, one can see them:

- In the documentation:
  for example, `var int` is a subtype of `var opt int`.
- In error messages, due to *implicit* option variables being made explicit while flattening, but things getting too complex:
  see the symptomatic example at slide 21.

# Constraints

A constraint is the keyword `constraint` followed by a Boolean expression that must be true in every solution.

## Examples

```
constraint x < y;
constraint sum(X) = 0 /\ all_different(X);
```

Constraints separated by a semi-colon (`;`) are implicitly connected by the 2-ary logical-and connective (`/\`).

What does `constraint x = x + 1` mean?
MiniZinc is declarative and has no destructive assignment: this equality constraint is not satisfied by any value for `x`.

MiniZinc tolerates the syntax `x == y + 1` for `x = y + 1`, but note that MiniZinc is syntax for mathematics and logic, where `==` does not exist!

# Objective

The `solve` item gives the objective of the problem:

- `solve satisfy;`
  The objective is to solve a satisfaction problem.
- `solve minimize x;`
  The objective is to minimise the value of decision variable `x`.
- `solve maximize x + y;`
  The objective is to maximise the value of the objective function `x + y`.

MiniZinc does not support multi-objective optimisation yet:
multiple objective functions must either be aggregated into a weighted sum,
or be handled outside a MiniZinc model.

# Output

The `output` item prescribes what to print upon finding a solution: the keyword `output` is followed by an array of strings.

```
output [show(x div n)];
```

The function `show` returns a string with the value of its variable expression.

```
output ["Solution: "] ++ [if X[i] > 0 then
  show(X[i])++", " else " , " endif | i in 1..n];
```

The operator `++` concatenates two strings or two arrays.

The string `"\(X[i]), "` equals `show(X[i])++", "`. There is `show2d`.

The search strategy of the CP backend Gecode depends on the decision variables mentioned in the `output` statement.

# Operators and Functions

- Booleans: `not`, `/\`, `\/`, `<->`, `->`, `<-`, `xor`, `forall`, `exists`, `xorall`, `iffall`, `clause`, `bool2int`, ...
  Beware of arbitrarily nested logical quantifications, such as `forall(...exists(...forall(...)))`!
- Integers: `+`, `-`, `*`, `div` (note that `/` is for `float`), `mod`, `abs`, `pow`, `min`, `max`, `sum`, `product`, `=` (or `==` if you have to), `<`, `<=`, `=>`, `>`, `!=`, ...
  Beware of `div`, `mod`, and `pow` on decision variables!
- Sets: `..`, `in`, `card`, `subset`, `superset`, `union`, `array_union`, `intersect`, `array_intersect`, `diff`, `symdiff`, `set2array`, ...
- Strings: `++`, `concat`, `join`, ...
- Arrays: `length`, `index_set`, `index_set_1of2`, `index_set_2of2`, ..., `index_set_6of6`, `array1d`, `array2d`, ..., `array6d`, ...

# Predicates and Functions

MiniZinc offers a large collection of predefined predicates and functions in order to enable a high level at which models can be formulated. See Topic 3: Constraint Predicates.

Each predefined constrained function is defined by the corresponding constraint predicate, possibly upon introducing a new decision variable.

**Example**

`count(X,v)>m` is defined by `count(X,v,c) /\ c>m` with `var int: c`.

It is also possible for modellers to define their own functions and predicates, as discussed at slide 25.

# Reification

Reification enables the reasoning about the truth of a constraint or a Boolean expression.

## Example

```
constraint x < y;
```

requires that `x` be smaller than `y`.

```
constraint b <-> x < y;
```

requires that the Boolean variable `b` take the value `true` iff `x` is smaller than `y`: the constraint `x < y` is said to be reified, and `b` is called its reification.

Reification is a powerful mechanism that enables:

- higher-level modelling;
- easier implementation of the logical connectives.

The expression `bool2int(`$\phi$`)`, for a Boolean expression $\phi$, denotes the integer `1` if $\phi$ is true, and `0` if $\phi$ is false.

## Example (Cardinality constraint)

Constrain one or two of three constraints $\gamma_1$, $\gamma_2$, $\gamma_3$ to hold:

`bool2int(`$\gamma_1$`) + bool2int(`$\gamma_2$`) + bool2int(`$\gamma_3$`) in {1,2}`

As `bool2int` coercion is automatic, one can actually write:

$\gamma_1$ `+` $\gamma_2$ `+` $\gamma_3$ `in {1,2}`

However, as a coding convention, we recommend to write:

`(`$\gamma_1$`) + (`$\gamma_2$`) + (`$\gamma_3$`) in {1,2}`

thereby mimicking the Iversion bracket (see slide 12).

Reification (implicit via `bool2int` and `(...)`) has pitfalls:

- Inference and relaxation might be poor: slow solving.
- Not all constraints can be reified in MiniZinc, such as some of those in Topic 3: Constraint Predicates.

A conditional expression can be formulated as follows:

- Conditional: `if` $\theta$ `then` $\phi_1$ `else` $\phi_2$ `endif`
- Comprehension: `[i | i in` $\sigma$ `where` $\theta$`]`

The expressions $\phi_1$ and $\phi_2$ must have the same type.

The test $\theta$ after `if` or `where` may have variables, but this can be a source of unexpected behaviour (Section 2.4.3), inefficiency, or impossible flattening!

## Example

```
enum I; set of int: T; array[I] of var T: X;
array[I] of var T: Y=[X[i]|i in I where X[i]>0]; constraint sum(Y)<7;
```

This yields an error message with `var opt` (see slide 13) as the indices of `Y` cannot be determined when flattening and cannot just be set to `I`.

But the following works:

```
constraint sum([X[i] | i in I where X[i]>0]) < 7;
```

and so does the use of implicit reification, possibly better:

```
constraint sum([(X[i]>0) * X[i] | i in I]) < 7;
```

## Example (Soft Constraints: Alignment Photo Problem + )

An enumeration `Students` of students want to line up for a class photo.

Consider:

```
array[_] of record(Students: who, Students: whom          ): Wish;
```

The wish `w` in `Wish` denotes that student `w.who` wants to be next to student `w.whom` on the photo.
Maximise the number of granted wishes.

Let decision variable `Pos[s]` denote the position in `1..card(Students)` of student `s` on the photo.

The array `Pos` must form a permutation of the positions:

```
constraint all_different(Pos);
```

The objective, formulated using implicit reification, is:

```
solve maximize sum(w in Wish)
  (         (abs(Pos[w.who] - Pos[w.whom]) = 1));
```

UPPSALA UNIVERSITET



## Example (Soft Constraints: Weighted Photo Alignment Problem + )

An enumeration `Students` of students want to line up for a class photo.

Consider:

```
array[_] of record(Students: who, Students: whom, int: bid): Wish;
```

The wish `w` in `Wish` denotes that student `w.who` wants to pay `w.bid` in order to be next to student `w.whom` on the photo.
Maximise the weighted number of granted wishes.

Let decision variable `Pos[s]` denote the position in `1..card(Students)` of student `s` on the photo.

The array `Pos` must form a permutation of the positions:

```
constraint all_different(Pos);
```

The objective, formulated using implicit reification, is:

```
solve maximize sum(w in Wish)
  (           (abs(Pos[w.who] - Pos[w.whom]) = 1));
```

## Example (Soft Constraints: Weighted Photo Alignment Problem + )

An enumeration `Students` of students want to line up for a class photo.

Consider:

```
array[_] of record(Students: who, Students: whom, int: bid): Wish;
```

The wish `w` in `Wish` denotes that student `w.who` wants to pay `w.bid` in order to be next to student `w.whom` on the photo.
Maximise the weighted number of granted wishes.

Let decision variable `Pos[s]` denote the position in `1..card(Students)` of student `s` on the photo.

The array `Pos` must form a permutation of the positions:

```
constraint all_different(Pos);
```

The objective, formulated using implicit reification, is:

```
solve maximize sum(w in Wish)
  (w.bid * (abs(Pos[w.who] - Pos[w.whom]) = 1));
```

UPPSALA UNIVERSITET

**The MiniZinc Language**

Modelling

Set Variables &Constraints

Modelling Checklist

### Example (Sum of unweighted reified constraints)

The expression `sum(i in index_set(X))(X[i] = v)` denotes the number of decision variables of array `X` that are equal to decision variable `v`.

This idiom is very common in constraint-based models. So it has a name:

### Definition (The `count` constraint predicate)

The constraint `count(X,v,c)` holds if and only if decision variable `c` has the number of decision variables of array `X` that are equal to decision variable `v`.

For other predicates, see Topic 3: Constraint Predicates.

### Definition (The `count` constrained function)

The expression `count(X,v)` denotes the number of decision variables of array `X` that are equal to decision variable `v`.

### Example (Unweighted Photo Alignment Problem, revisited)

```
solve maximize count([abs(Pos[w.who] - Pos[w.whom]) | w in Wish], 1);
```

# Predicate and Function Definitions

## Examples

```
function int: double(int: x);
function var int: double(var int: x);
predicate pos(var int: x);
function var bool: neg(var int: x);
```

A predicate is a function denoting a `var bool`:

## Examples

```
function var bool: pos(var int: x);
predicate neg(var int: x);
```

Function and predicate names can be overloaded.

The body of a predicate or function definition is an expression of the same type as the denoted value.

## Examples

```
function int: double(int: x) = 2*x;
function var int: double(var int: x) = 2*x;
predicate pos(var int: x) = x > 0;
function var bool: neg(var int: x) = x < 0;
```

One can use `if ... then ... else ... endif` expressions, predicates and functions, such as `forall` and `exists`, as well as `let` expressions (see the next slide) in the body of a predicate or function definition.

UPPSALA UNIVERSITET

**The MiniZinc Language**

Modelling

Set Variables &Constraints

Modelling Checklist

# Let Expressions

One can introduce local identifiers within a `let` expression and constrain them.

## Examples

```
function int: double(int: x) =
  let { int: y = 2 * x } in y;
function var int: double(var int: x) =
  let { var int: y = 2 * x } in y;
function var int: double(var int: x) =
  let { var int: y;
        constraint y = 2 * x
  } in y;
```

The second and third functions are equivalent:
each use adds a decision variable to the model.

# Constraints in Let Expressions

What is the difference between the following two definitions?

```
predicate posProd(var int: x, var int: y) =
  let { var int: z; constraint z = x * y
  } in z > 0;
predicate posProd(var int: x, var int: y) =
  let { var int: z
  } in z = x * y /\ z > 0;
```

Their behaviour is different in a negative context, such as `not posProd(a,b)`:

- The 1st one then ensures `a * b = z /\ z <= 0`.
- The 2nd one then ensures `a * b != z \/ z <= 0` and leaves `a` and `b` unconstrained.

# Using Predicates and Functions

Advantages of using predicates and functions in a model:

- Software engineering good practice:
  - Reusability
  - Readability
  - Modularity
- The model might be solved more efficiently:
  - Better common-subexpression elimination.
  - The definitions can be technology-specific or solver-specific. If a predefined constraint predicate is a built-in of a solver, then its solver-specific definition is identity!

# Remarks

- The order of model items does not matter.
- One can include other files.
  Example: `include "globals.mzn"`.
- The following functions are useful for debugging:
  - `constraint assert(`$\theta$`,"error message")`
    If the Boolean expression $\theta$ evaluates to `false`,
    then abort with the error message, otherwise denote `true`.
  - `trace("message",` $\phi$`)`
    Print the message and denote the expression $\phi$.
  - …

# Other Modelling Languages

- OPL: https://www.ibm.com/optimization-modeling
- Comet: https://mitpress.mit.edu/books/constraint-based-local-search
- Essence and Essence′: https://constraintmodelling.org
- Zinc: https://dx.doi.org/10.1007/s10601-008-9041-4
- AIMMS: https://aimms.com
- AMPL: https://ampl.com
- FICO Xpress Insight: https://www.fico.com/en/products/fico-xpress-optimization
- GAMS: https://gams.com
- SMT-lib: https://smtlib.cs.uiowa.edu
- ...

UPPSALA UNIVERSITET



# Outline

# From a Problem to a Model

What is a good model for a constraint problem?

- A model that correctly represents the problem
- A model that is easy to understand and maintain
- A model that is solved efficiently, that is:
  - short solving time to find one, all, or best solution(s)
  - good solution within a limited amount of time
  - small search space (under systematic search)

Food for thought: What is correct, easy, short, good, ...?

# Modelling Issues

Modelling is still more an Art than a Science:

- Choice of the decision variables and their domains
- Choice of the constraint predicates,
  in order to model the objective function, if any, and the constraints
- Optional for CP and LCG:
  - Choice of the consistency for each constraint
  - Choice of the variable selection strategy for search
  - Choice of the value selection strategy for search

  See Topic 8: Inference & Search in CP & LCG.

Make a model correct before making it efficient!

# Choice of the Decision Variables

## Examples (Alphametic Problems)

SEND + MORE = MONEY:

Model without carry variables: 19 of 23 CP nodes are visited:

$$\begin{aligned} & 1000 \cdot (S + M) + 100 \cdot (E + O) + 10 \cdot (N + R) + (D + E) \\ & = 10000 \cdot M + 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y \end{aligned}$$

Model with carry variables: 23 of 29 CP nodes are visited:

$$\begin{aligned} & D + E = 10 \cdot C_1 + Y \;\wedge\; N + R + C_1 = 10 \cdot C_2 + E \\ & \wedge\; E + O + C_2 = 10 \cdot C_3 + N \;\wedge\; S + M + C_3 = 10 \cdot M + O \end{aligned}$$

GERALD + DONALD = ROBERT: The model with carry variables is more effective in CP: only 791 of 869 nodes are visited, rather than 13,795 of 16,651 search nodes for the model without carry variables.

# Choice of the Constraint Predicates

**Example (The `all_different` constraint predicate)**

The constraint `all_different(X)` on an array `X` of size `n` usually leads to faster solving than its definition by a conjunction of $\frac{n \cdot (n-1)}{2}$ disequality constraints:

```
forall(i,j in index_set(X) where i < j)(X[i] != X[j])
```

For more examples, see Topic 3: Constraint Predicates.

# Guidelines: Reveal Problem Structure

- Use few decision variables, and declare tight domains
- Beware of nonlinear and power constraints: `pow`
- Beware of division constraints: `div` and `mod` (avoid `/`, which is for `float`)
- Beware of disjunction and negation: `\/`, `<-`, `->`, `<->`, `not`
- Express the problem concisely (see Topic 3: Constraint Predicates)
- Precompute solutions to a sub-problem into a table (see Topic 3: Constraint Predicates; see Topic 4: Modelling)
- Use implied constraints (see Topic 4: Modelling)
- Use different viewpoints (see Topic 4: Modelling)
- Exploit symmetries (see Topic 5: Symmetry)

Careful: These guidelines of course have their exceptions!

It is important to test empirically several combinations of model, solver, and solving technology.

# Use Few Decision Variables

When appropriate, use a single integer decision variable instead of an array of Boolean decision variables:

## Example

Assume Joe must be assigned to exactly one task in `1..n`:

- Use a single integer decision variable, `var 1..n: joesTask`, denoting *which* task Joe is assigned to.
- Do not use `array[1..n] of var bool:joesTask`, each element `joesTask[t]` denoting *whether* (`true`) or not (`false`) Joe is assigned to task `t`, plus `count(joesTask,true) = 1`.

When appropriate, use a single set decision variable instead of an array of Boolean or integer decision variables: see slides 49 and 51.

# Declare the Decision Variables with Tight Domains

Tight domains for decision variables might accelerate the solving.
Beware of `var int` for non equality-constrained decision variables.

**Example (Use parameters for declaring tight domains)**

If the decision variable `t` denotes a time, then write `var 0..h: t`, where horizon `h` is a parameter, instead of `var int: t`.

**Definition**

A derived parameter is computed from the parameters in the instance data.

**Example (Use derived parameters for declaring tight domains)**

```
int: p; int: c= ceil(pow(p,1/3)); int: s= ceil(sqrt(p));
var 1..c: x; var 1..s: y; var 1..p: z; % no "var int"
constraint x * y * z = p /\ x <= y /\ y <= z;
```

# Beware of Nonlinear and Power Constraints

Constraining the product of two or more decision variables often makes the solving slow. Try and find a linear reformulation.

## Example

The model snippet

```
array[1..n] of var 0..1: X;
array[1..n] of var 0..1: Y;
constraint count([X[i] * Y[i] | i in 1..n], 1) = b;
```

should be reformulated as:

```
array[1..n] of var 0..1: X;
array[1..n] of var 0..1: Y;
constraint count([X[i] + Y[i] | i in 1..n], 2) = b;
```

# Beware of Division Constraints

The use of `div` and `mod` on decision variables often makes the solving slow. Use `table` (see Topic 3: Constraint Predicates) or reformulate.

## Example

The model snippet

```
solve minimize sum(X) div n; % minimise the average
```

over `n` decision variables `X[i]` and parameter `n` should become:

```
solve minimize sum(X);         % minimise the sum
output [show(sum(X) div n)]; % output the average
```

# Beware of Disjunction and Negation

The disjunction of constraints (with `\/`, `xor`, `<-`, `->`, `<->`, `exists`, `xorall`, `if` $\theta$ `then` $\phi$ `else` $\psi$ `endif`) often makes the flat code long and the solving slow. Try and express disjunctive combinations of constraints otherwise.

## Example

The model snippet

```
constraint x = 0 \/ (low <= x /\ x <= up);
```

with parameters `low` and `up`, should be reformulated as:

```
constraint x in {0} union low..up;
```

or, even better in this particular case, as:

```
var {0} union low..up: x;
```

Disjunction or other sources of slow solving may also be introduced by `not`.

## Example

The model snippet

```
constraint x=3 \/ x=5 \/ x=7;
```

flattens into the very inefficient code:

```
var int: x;
var bool: B3; var bool: B5; var bool: B7;
constraint array_bool_or([B3,B5,B7],true);
constraint B3 -> x=3; % int_eq_imp(x,3,B3)
constraint B5 -> x=5; % this is called
constraint B7 -> x=7; %   half-reification
```

It should be reformulated as `constraint x in {3,5,7}` or, even better, as `var {3,5,7}: x`, which both flatten into the latter: a domain is a disjunction of candidate values!

## Example

The model snippet

```
constraint b -> x = 9;  constraint (not b) -> x = 0;
```

can be reformulated as (recall that `bool2int(true)=1`):

```
constraint x = 9 * b;
```

or as (note that array indexing starts by default at 1):

```
constraint x = [0,9][1+b];
```

But beware of such premature fine-tuning of a model!

The following reformulations are clearer and often good enough:

```
constraint x = if b then 9 else 0 endif;
```

and

```
constraint if b then x=9 else x=0 endif;
```

# Express the Problem Concisely

Whenever possible, use a single predefined constraint predicate instead of a long-winded (quantified) formulation of its semantics.

## Example (The `all_different` constraint predicate)

The constraint `all_different(X)` on an array `X` of size `n` usually leads to faster solving than its definition by a conjunction of $\frac{n \cdot (n-1)}{2}$ disequality constraints:

```
forall(i,j in index_set(X) where i < j)(X[i] != X[j])
```

For more examples, see Topic 3: Constraint Predicates.

# Outline

# Motivating Example 1

## Example (Agricultural experiment design, AED)

| | plot1 | plot2 | plot3 | plot4 | plot5 | plot6 | plot7 |
|---|---|---|---|---|---|---|---|
| barley | | | | | | | |
| corn | | | | | | | |
| millet | | | | | | | |
| oats | | | | | | | |
| rye | | | | | | | |
| spelt | | | | | | | |
| wheat | | | | | | | |

**Constraints** to be **satisfied**:

1. Equal growth load: Every plot grows 3 grains.
2. Equal sample size: Every grain is grown in 3 plots.
3. Balance: Every grain pair is grown in 1 common plot.

# Motivating Example 1

## Example (Agricultural experiment design, AED)

| | plot1 | plot2 | plot3 | plot4 | plot5 | plot6 | plot7 |
|---|---|---|---|---|---|---|---|
| barley | ✓ | ✓ | ✓ | – | – | – | – |
| corn | ✓ | – | – | ✓ | ✓ | – | – |
| millet | ✓ | – | – | – | – | ✓ | ✓ |
| oats | – | ✓ | – | ✓ | – | ✓ | – |
| rye | – | ✓ | – | – | ✓ | – | ✓ |
| spelt | – | – | ✓ | ✓ | – | – | ✓ |
| wheat | – | – | ✓ | – | ✓ | ✓ | – |

**Constraints** to be **satisfied**:

1. Equal growth load: Every plot grows 3 grains.
2. Equal sample size: Every grain is grown in 3 plots.
3. Balance: Every grain pair is grown in 1 common plot.

In a BIBD, the plots are blocks and the grains are varieties:

## Example (BIBD *integer* model: $\checkmark \rightsquigarrow 1$ and $- \rightsquigarrow 0$)

```
enum Varieties; enum Blocks;
int: blockSize; int: sampleSize; int: balance;
array[Varieties,Blocks] of var 0..1: BIBD; % BIBD[v,b]=1 iff v is in b
solve satisfy;
constraint forall(b in Blocks)    (blockSize  = count(BIBD[..,b], 1));
constraint forall(v in Varieties)(sampleSize = count(BIBD[v,..], 1));
constraint forall(v, w in Varieties where v < w)
  (balance = count([BIBD[v,b]+BIBD[w,b] | b in Blocks], 2));
```

## Example (Instance data for our AED)

```
Varieties = {barley,...,wheat}; Blocks = {plot1,...,plot7};
blockSize = 3; sampleSize = 3; balance = 1;
```

## Example (Idea for another BIBD model)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| barley | {plot1, | plot2, | plot3 | | | | } |
| corn | {plot1, | | | plot4, | plot5 | | } |
| millet | {plot1, | | | | | plot6, | plot7} |
| oats | { | plot2, | | plot4, | | plot6 | } |
| rye | { | plot2, | | | plot5, | | plot7} |
| spelt | { | | plot3, | plot4, | | | plot7} |
| wheat | { | | plot3, | | plot5, | plot6 | } |

**Constraints** to be **satisfied**:

1. Equal growth load: Every plot grows 3 grains.
2. Equal sample size: Every grain is grown in 3 plots.
3. Balance: Every grain pair is grown in 1 common plot.

## Example (BIBD *set* model: a block *set* per variety)

```
-3 enum Varieties; enum Blocks;
-2 int: blockSize; int: sampleSize; int: balance;
-1 array[Varieties] of var set of Blocks: BIBD; % BIBD[v] = blocks for v
 0 solve satisfy;
 1 constraint forall(b in Blocks)
     (blockSize  = sum(v in Varieties)(b in BIBD[v]));
 2 constraint forall(v in Varieties)
     (sampleSize = card(BIBD[v]));
 3 constraint forall(v, w in Varieties where v < w)
     (balance    = card(BIBD[v] intersect BIBD[w]));
```

## Example (Instance data for our AED)

```
-3 Varieties = {barley,...,wheat}; Blocks = {plot1,...,plot7};
-2 blockSize = 3; sampleSize = 3; balance = 1;
```

# Motivating Example 2 [2]

## Example (Hamming code: problem)

The Hamming distance between two same-length strings is the number of positions at which the two strings differ.
Examples: $h(10001, 01001) = 2$ and $h(11010, 11110) = 1$.

ASCII has codewords of $m = 8$ bits for $n = 2^m$ symbols, but the least Hamming distance is $d = 1$: no robustness!

Toward high robustness in data transmission, we want to generate a codeword of $m$ bits for each of the $n$ symbols of an alphabet, such that the Hamming distance between any two codewords is at least some given constant $d$.

[2] Based on material by Christian Schulte

## Example (Hamming code: model and data)

We encode a codeword of $m$ bits as the set of positions of its unit bits, the least significant bit being at position 1.
Example: 10001 is encoded as $\{1, 5\}$, and 01001 is encoded as $\{1, 4\}$.
In general: $b_m \cdots b_1$ is encoded as $\{1 \cdot b_1, \ldots, m \cdot b_m\} \setminus \{0\}$.
So the Hamming distance between two codewords is $u - i$, where $u$ is the size of the union of their encodings and $i$ is the size of the intersection of their encodings, that is the size of the symmetric difference of their encodings:

```
array[1..n] of var set of 1..m: Codeword;
constraint forall(i, j in 1..n where i < j)
  (card(Codeword[i] symdiff Codeword[j]) >= d);
```

## Definition

A set (decision) variable takes a set as value, and has a set of sets as domain. For its domain to be finite, a set variable must be a subset of a given finite set.

Set-constraint predicates exist for the following semantics:

- Cardinality: $|S| = n$
- Membership: $n \in S$
- Equality: $S_1 = S_2$
- Disequality $S_1 \neq S_2$
- Subset: $S_1 \subseteq S_2$
- Union: $S_1 \cup S_2 = S_3$
- Intersection: $S_1 \cap S_2 = S_3$
- Difference: $S_1 \setminus S_2 = S_3$
- Symmetric difference: $(S_1 \cup S_2) \setminus (S_1 \cap S_2) = S_3$
- Order: $S_1 \subseteq S_2 \vee \min((S_1 \setminus S_2) \cup (S_2 \setminus S_1)) \in S_1$
- Strict order: $S_1 \subset S_2 \vee \min((S_1 \setminus S_2) \cup (S_2 \setminus S_1)) \in S_1$

where the $S_i$ are set decision variables and $n$ is an integer decision variable.
Set variables may backfire in M4CO assignments, but may be useful in project.

# Beware of Variable Integer Ranges

Reification and set variables may appear while flattening complex expressions:

## Example

```
var 1..5: x;  array[1..7] of var 1..9: X;
constraint forall(i in 1..x)(X[i]<3);
```

flattens into inefficient code, linear in the domain size of `x`:

```
var 1..5: x;  array[1..7] of var 1..9: X;
var set of 1..5: S; % prefix of indices i with X[i]<3
var bool: B2; ...; var bool: B9;
constraint        1 in S; constraint  X[1] < 3;
constraint B2 <-> 2 in S; constraint B2 <-> 2 <= x;
constraint B2 -> B6; constraint B6 -> X[2] < 3;
constraint ...;
constraint B5 <-> 5 in S; constraint B5 <-> 5 <= x;
constraint B5 -> B9; constraint B9 -> X[5] < 3;
```

Avoid ranges $\alpha\,..\,\beta$ where $\alpha$ or $\beta$ (or both) are decision variables.

# Collection Variables

## Definition

- A set decision variable has a set of sets as domain.
- An array decision variable has a set of arrays as domain.
- A string decision variable has a set of strings as domain.

MiniZinc currently has no syntax for the declaration of array variables and string variables, and MiniZinc currently has no constraint predicates for such decision variables (but our proposal at LOPSTR 2016 and CP-AI-OR 2017 for string variables and string constraint predicates is ready for integration).

## Subtle difference with imperative programming and OO programming!

array of variables $=$ variable array $\neq$ array variable

`array[int] of var int: X` $\neq$ `var array[int] of int: Y`

`X[i]` is a variable; `X` itself is not a variable; `Y` would be a variable, if supported

# Outline

# Conventions of all Slides (recommended!)

- Scalar identifiers (`bool`, `enum` items, `int`) start with a lowercase letter.
- Mass identifiers (`array`, `enum`, `set`) start with an uppercase letter.
- Arrays have self-explanatory function identifiers: a given|unknown total function $f\colon X \to Y$ can be modelled as `array[X] of par|var Y: F`.
- Index identifiers are lowercase and mnemonic: memory aid.
- Comments about the *next* line end in "`:`", like line 2 in the example below.

## Example

```
1 int: nQueens; % the given number of queens
2 % Row[c] = the row number of the queen in column c:
3 array[1..nQueens] of var 1..nQueens: Row;
```

Variable `Row[c]` is like $Row(c)$, denoting the function *Row* applied to arg. *c*. The array `Row` is *not* a variable, but an *array of* variables: it has row numbers, but calling it `Rows` would make `Rows[c]` seem to denote a *set* of rows for `c`!

# Ideas for Debugging and Accelerating a Model

- If there are no solutions (or missing solutions) to a known-to-be satisfiable instance, then:
  - Comment away constraints in order to increase the solution set and thereby find unsatisfiable constraints.
  - In the IDE or CLI, choose findMUS as the backend in order to find a minimal unsatisfiable subset (MUS) of the constraints: see Section 3.8 of the MiniZinc Handbook.
- In the IDE, choose “Run > Profile compilation” in order to see per model line the numbers of constraints and decision variables generated by its flattening, and the flattening time: if some of these numbers are extreme, then you probably ran afoul of items of the checklist on the next slide.
- In the IDE, choose “Run > Compile” in order to inspect the flat code.

UPPSALA UNIVERSITET



# Checklist for Designing or Reading a Model

1. Each index of an array occurs in the semantics of the array
2. Each index range of an array either starts from `1` or is `enum`, for clarity
3. Beware of decision variables without tight domains
4. No explicit decision variables of type `opt` $\tau$ are used (in this course)
5. No `sum|forall(i in 1..x)` with a decision variable `x` is used
6. Beware of `where` $\theta$ and `if` $\theta$ with test $\theta$ containing decision variables
7. Beware of explicit (`<->`) and implicit (`(...)`) reification
8. Beware of negation and disjunction: `not`, `\/`, `exists`, `xor`, `xorall`, `if` $\theta$ `then` $\phi$ `else` $\psi$ `endif`, `<-`, `->`, `<->`
9. Beware of arbitrarily nested logical quantifications, such as `forall(...exists(...forall(...)))`
10. Beware of nonlinear, `pow`, `div`, `mod` constraints on decision variables