

# **AN INTRODUCTION TO MINIZING**

# MINIZINC

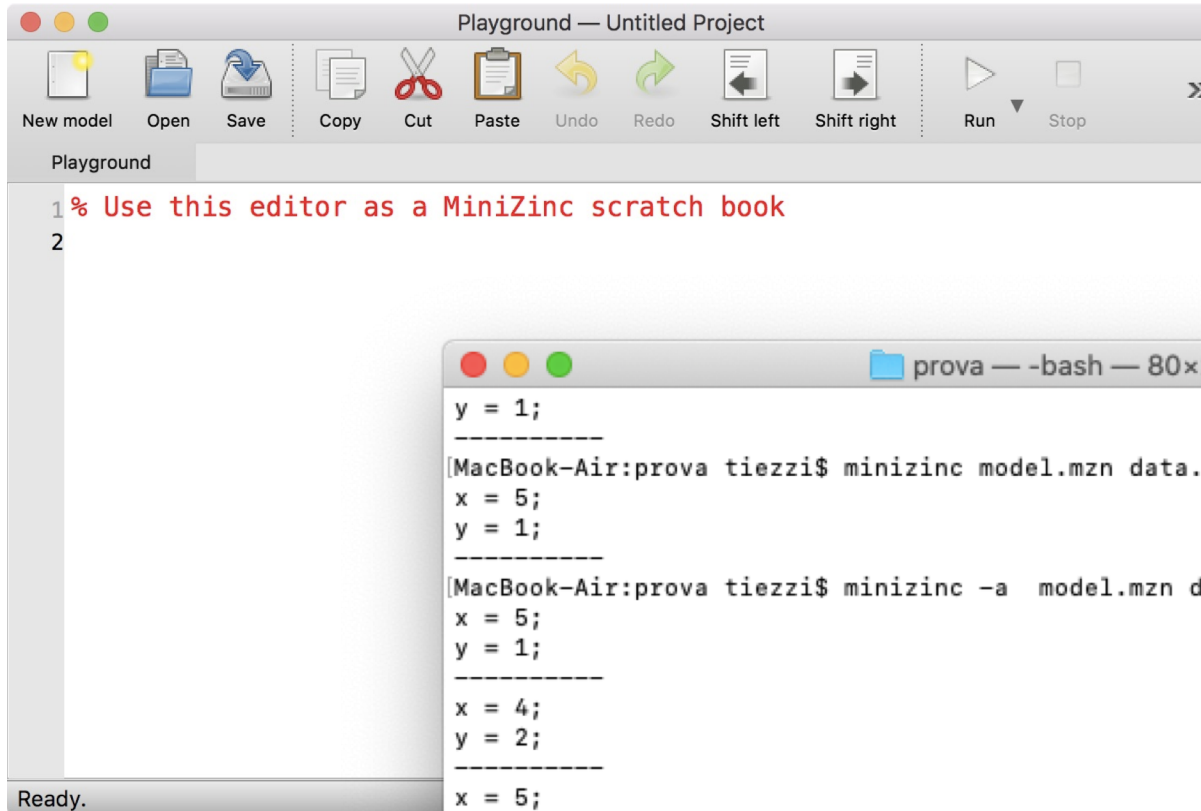
**MiniZinc** is a language designed for specifying constrained optimization and decision problems over **integers** and **real numbers**

A MiniZinc model **does not dictate how to solve the problem** although the model can contain annotations which are used to guide the underlying solver

MiniZinc is designed **to interface easily** to different backend solvers

- An input MiniZinc model and data file is **transformed into** a **FlatZinc** model
- FlatZinc models consist of variable declaration and constraint definitions as well as a definition of the objective function if the problem is an optimization problem
- The translation from MiniZinc to FlatZinc is **specializable** to individual backend solvers

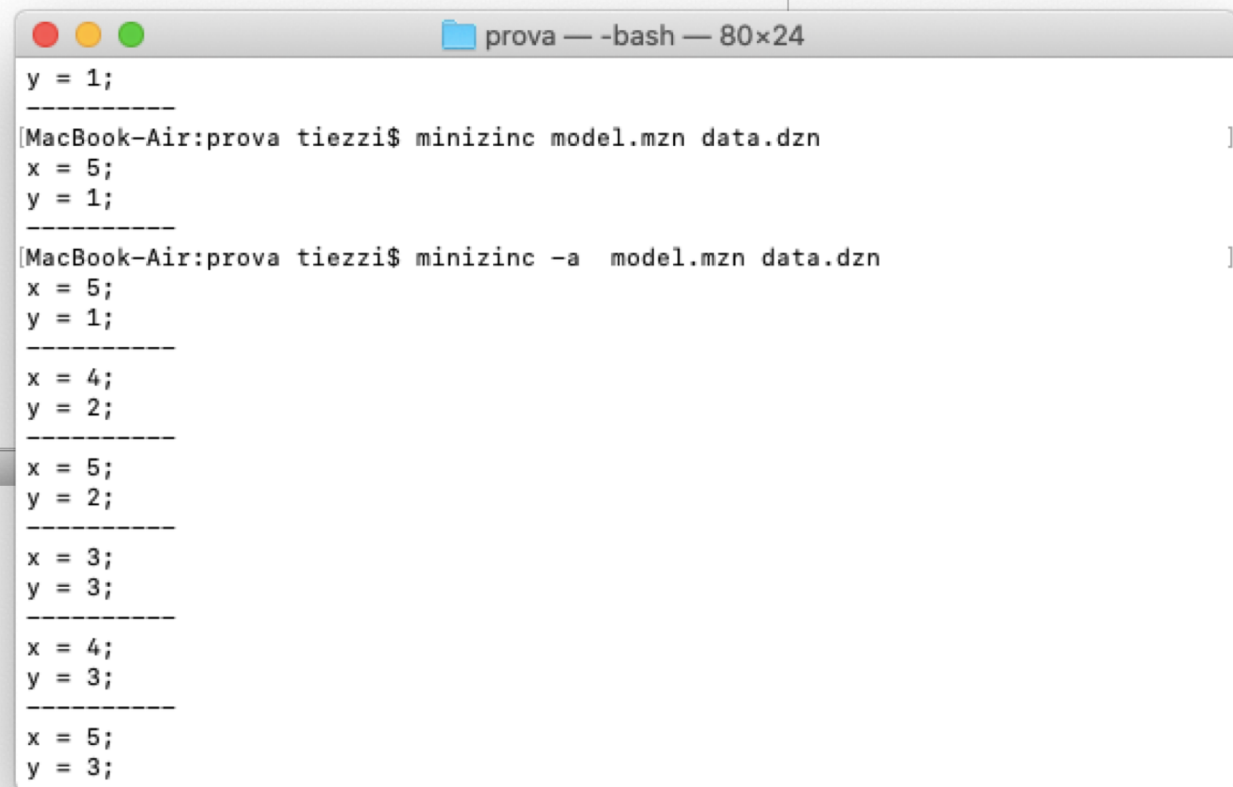
# INTRODUCTORY DEMO



The screenshot shows a code editor window with a toolbar at the top. The toolbar includes icons for 'New model', 'Open', 'Save', 'Copy', 'Cut', 'Paste', 'Undo', 'Redo', 'Shift left', 'Shift right', 'Run', and 'Stop'. The main text area contains two lines of code:

```
1 % Use this editor as a MiniZinc scratch book
2
```

At the bottom left of the window, the text 'Ready.' is displayed.



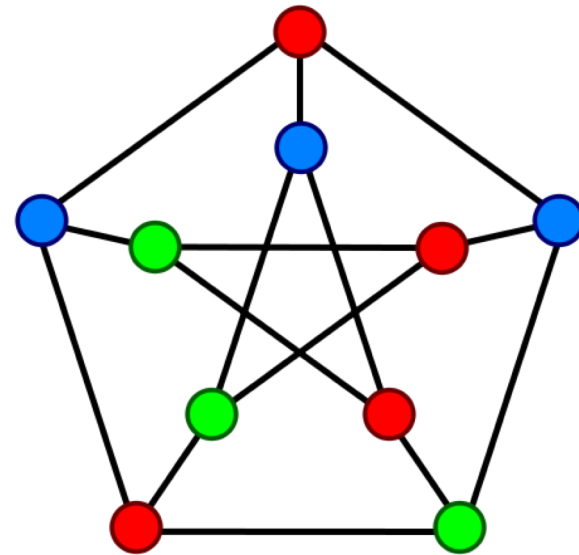
The screenshot shows a terminal window with the following output:

```
y = 1;
-----
[MacBook-Air:prova tiezzi$ minizinc model.mzn data.dzn ]
x = 5;
y = 1;
-----
[MacBook-Air:prova tiezzi$ minizinc -a model.mzn data.dzn ]
x = 5;
y = 1;
-----
x = 4;
y = 2;
-----
x = 5;
y = 2;
-----
x = 3;
y = 3;
-----
x = 4;
y = 3;
-----
x = 5;
y = 3;
```

# A FIRST EXAMPLE

We wish to **colour** a map of Australia

**Seven** different states and territories each of which must be given a colour so that **adjacent regions have different colours**



In **graph theory**, graph coloring is a special case of graph labeling; it is an assignment of labels traditionally called “colours” to elements of a graph subject to certain constraints

**Edge** coloring assigns a color to each edge so that no two adjacent edges share the same color

# CODE

AUST ≡

```
% Colouring Australia using nc colours
```

```
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"];
```

# COMMENTS

A comment starts with a '**%**' which indicates that the rest of the line is a comment.

Example:

**% Coloring Australia using nc colors**

MiniZinc has also begin/end comment symbols: **/\*** and **\*/**

# VARIABLE DECLARATIONS

**int: nc = 3;** declares a variable of the model

- the number of colours to be used
- a *parameter* in the problem
- They must be declared and given a *type*. In this case the type is **int**
- They are given a value by an *assignment*, as part of the declaration (as above), or a separate assignment statement

```
int: nc;
```

```
nc = 3;
```

- It is an error for a parameter to occur in more than one assignment (like a **constant** variable in most programming languages)

The basic parameter types are integers (**int**), floating point numbers (**float**), booleans (**bool**) and strings (**string**)

**Arrays** and **sets** are also supported

# DECISION AND PARAMETERS

MiniZinc distinguishes between the two kinds of model variables: **parameters** and **decision** variables

- Expressions that can be constructed using decision variables are more restricted than those that can be built from parameters.
- In any place that a decision variable can be used, so can a parameter of the same type

The distinction between parameters and decision variables concerns the **instantiation** of the variable

- The former is instantiated by **you** (the modeller)
- The second is instantiated by **the solver**



# BACK TO THE EXAMPLE

In our colouring model we associate a **decision variable** with each region, **wa**, **nt**, **sa**, **q**, **nsw**, **v** and **t**, which stands for the (unknown) colour to be used to fill the region

For each decision variable we need to give **the set of possible values** the variable can take. This is called the **variable's domain**

- It can be given as part of the variable declaration
- The type of the decision variable is inferred from the type of the values in the domain

In MiniZinc decision variables can be **booleans**, **integers**, **floating point numbers**, **sets**, or **arrays** whose elements are decision variables

In the example we use integers to model the different colours.

- **1..nc** which is an integer range expression indicating the set  $\{1, 2, \dots, nc\}$

# CONSTRAINTS

The next component of the model are the **constraints**

**These specify the boolean expressions that the decision variables must satisfy to be a valid solution to the model**

**In our running example we have a number of not equal constraints between the decision variables enforcing that if two states are adjacent then they must have different colours**



```
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;
```

MiniZinc provides: **equal** (= or ==), **not equal** (!=), **strictly less than** (<) **strictly greater than** (>), **less than or equal to** (<=), and **greater than or equal to** (>=).

# SOLVE AND OUTPUT

**solve satisfy;** indicates the kind of problem

In this case it is a **satisfaction** problem: we wish to find a value for the decision variables that satisfies the constraints **but we do not care which one**

The final part of the model is the **output** statement

- An output statement is followed by a *list* of strings
  - String literals which are written between double quotes and use a C like notation for special characters
  - expression of the form *show(X)* where *X* is the name of a decision variable or parameter

There are also formatted varieties of *show* for numbers

- *show\_float(n,d,X)* outputs the value of float *X* in at least  $|n|$  characters, right justified if  $n > 0$  and left justified otherwise, with *d* characters after the decimal point

**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"];

# RUN IT!

**Evaluate** our model with RUN button

or the command: `$ minizinc --solver gecode aust.mzn`

- **aust.mzn** is the name of the file which contains the whole model
- **gecode** is one of the solvers in the suite
- **The output is:**

wa=1

nt=3

sa=2

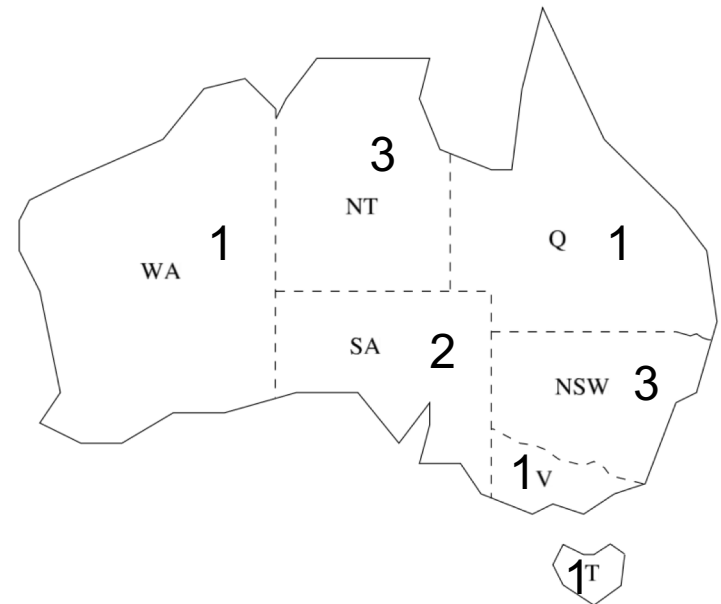
q=1

nsw=3

v=1

t=1

-----



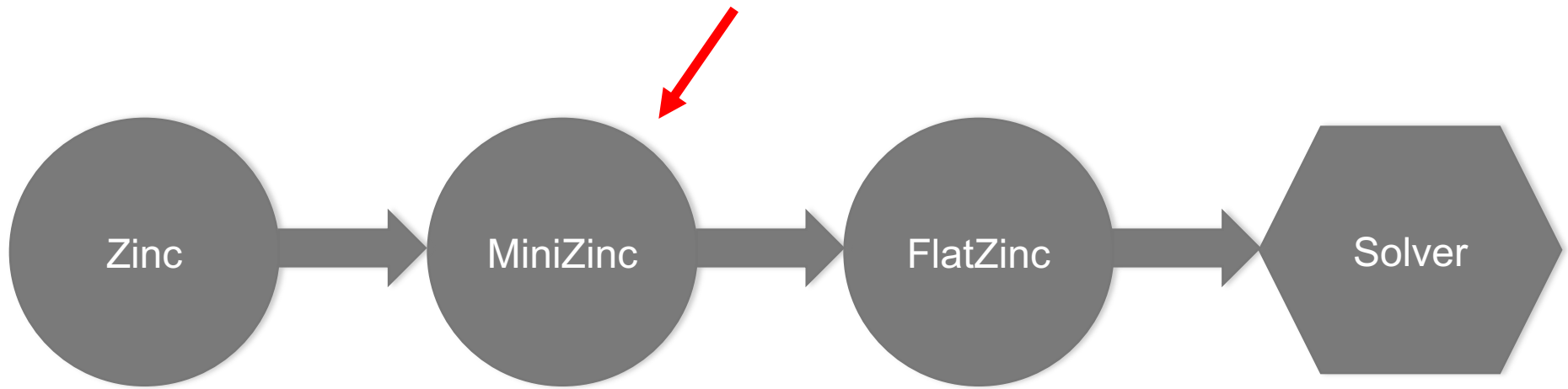
The line of 10 dashes ----- is automatically added by the MiniZinc in the output to indicate a solution has been found

# SOME MORE INFORMATION

**MiniZinc is a high-level, typed, mostly first-order, functional, modelling language. It provides:**

- mathematical notation-like syntax (automatic coercions, overloading, iteration, sets, arrays);
- expressive constraints (finite domain, set, linear arithmetic, integer);
- support for different kinds of problems (satisfaction, explicit optimisation);
- separation of data from model;
- extensibility (user-defined functions and predicates);
- reliability (type checking, instantiation checking, assertions);
- **solver-independent modelling**;
- simple, declarative semantics.

# FROM ZINC TO FLATZINC



FlatZinc is a low-level solver input language that is the target language for MiniZinc. It is designed to be easy to translate into the form required by a solver.

# YOU MODEL WE SOLVE

## FlatZinc Implementations

**Gecode/FlatZinc.** The Gecode generic constraint development environment provides a FlatZinc interface. The source code for the interface stripped of all Gecode-specific code is also available.

**ECLiPSe.** The ECLiPSe Constraint Programming System provides support for evaluating FlatZinc using ECLiPSe's constraint solvers. MiniZinc models can be embedded into ECLiPSe code in order to add user-defined search and I/O facilities to the models.

**SICStus Prolog.** SICStus (from version 4.0.5) includes a library for evaluating FlatZinc.

**JaCoP.** The JaCoP constraint solver (from version 4.2) has an interface to FlatZinc.

**SCIP.** SCIP, a framework for Constraint Integer Programming, has an interface to FlatZinc.

**Opturion CPX.** Opturion CPX, a Constraint Programming solver with eXplanation system, has an interface to FlatZinc.

**MinisatID.** MinisatID, an implementation of a search algorithm combining techniques from the fields of SAT, SAT Module Theories, Constraint Programming and Answer Set Programming, has an interface to FlatZinc.

# RESOURCES

## MiniZinc 2.0

- Windows, MacOS, Linux, Installation from source code.
- <http://www.minizinc.org/2.0/index.html>

## The MiniZinc IDE is a tool for writing and running MiniZinc models

- Windows, MacOS, Linux, source code.
- <http://www.minizinc.org/ide/index.html>

## MiniZinc 2.0 Specification

- <http://www.minizinc.org/2.0/doc-lib/minizinc-spec.pdf>

## Tutorial

- <http://www.minizinc.org/downloads/doc-latest/minizinc-tute.pdf>

## Global constraints and built-in functions

- <http://www.minizinc.org/2.0/doc-lib/doc.html>



zinc aust.mzn – Untitled Project

New model Open Save Copy Cut Paste Undo Redo Shift left Shift right Run Stop

Configuration aust.mzn

```
1 % Colouring Australia using nc colours
2 int: nc = 3;
3
4 var 1..nc: wa; var 1..nc: nt; var 1..nc: sa; var 1..nc: q;
5 var 1..nc: nsw; var 1..nc: v; var 1..nc: t;
6
7 constraint wa != nt;
8 constraint wa != sa;
9 constraint nt != sa;
10 constraint nt != q;
11 constraint sa != q;
12 constraint sa != nsw;
13 constraint sa != v;
14 constraint q != nsw;
15 constraint nsw != v;
16 solve satisfy;
17
18 output ["wa=", show(wa), "\t nt=", show(nt),
19         "\t sa=", show(sa), "\n", "q=", show(q),
20         "\t nsw=", show(nsw), "\t v=", show(v), "\n",
21         "t=", show(t), "\n"];

```

Output

```
Compiling aust.mzn
Running aust.mzn
wa=1      nt=3      sa=2
q=1      nsw=3     v=1
t=1
-----
Finished in 55msec

```

Ready. 55msec

# **ANOTHER EXAMPLE: BAKING!**

**We know how to make two sorts of cakes for a fete at Unicam.**

**A banana cake which takes 250g of self-raising flour, 2 mashed bananas, 75g sugar and 100g of butter,**

**and**

**a chocolate cake which takes 200g of self-raising flour, 75g of cocoa, 150g sugar and 150g of butter.**

**We can sell a chocolate cake for \$4.50 and a banana cake for \$4.00.**

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

**How many of each sort of cake should we bake for the fete to maximise the profit?**

# BAKING CAKES

```
% Baking cakes for the school fete

var 0..100: b; % no. of banana cakes
var 0..100: c; % no. of chocolate cakes

% flour
constraint 250*b + 200*c <= 4000;
% bananas
constraint 2*b <= 6;
% sugar
constraint 75*b + 150*c <= 2000;
% butter
constraint 100*b + 150*c <= 500;
% cocoa
constraint 75*c <= 500;

% maximize our profit
solve maximize 400*b + 450*c;

output ["no. of banana cakes = ", show(b), "\n",
        "no. of chocolate cakes = ", show(c), "\n"];
```

# INTEGER ARITHMETIC EXPR.

**MiniZinc provides the standard integer arithmetic operators.**

- Addition (+),
- subtraction (-),
- multiplication (\*),
- integer division (div),
- integer modulus (mod),

**MiniZinc provides Integer functions as**

- absolute value ( $\text{abs}(-4) = 4$ ),
- power function ( $\text{pow}(2,5) = 32$ )

**Integer literals can be decimal, hexadecimal or octal.**

- For instance 0, 005, 123, 0x1b7, 0o777

# OPTIMISATION

**solve maximize 400 \* b + 450 \* c;**

- We want to find a solution that maximises the expression in the solve statement called the *objective*
- The objective can be any kind of arithmetic expression
- One can replace the key word *maximize* by *minimize* to specify a minimisation problem

```
no. of banana cakes = 2  
no. of chocolate cakes = 2  
-----  
=====
```

**The line ===== is output automatically for optimisation problems when the system has proved that a solution is optimal**

# DATAFILES

A **drawback** of this model is that, each time we need to modify the amount of ingredients we have, we need to modify the constraints

**Solution:** set the value of these parameters in a separate **data file**, with extension **.dzn** (pantry.dzn)

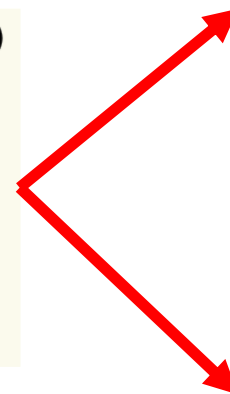
```
% Baking cakes for the school fete (with data file)
```

```
int: flour; %no. grams of flour available  
int: banana; %no. of bananas available  
int: sugar; %no. grams of sugar available  
int: butter; %no. grams of butter available  
int: cocoa; %no. grams of cocoa available
```

```
flour = 4000;  
banana = 6;  
sugar = 2000;  
butter = 500;  
cocoa = 500;
```

```
flour = 8000;  
banana = 11;  
sugar = 3000;  
butter = 1500;  
cocoa = 800;
```

no. of banana cakes = 3  
no. of chocolate cakes = 8



# ASSERTIONS

**Defensive programming** suggests that we should check that the values in the data file are reasonable

In case of our example, to check that the quantity of all ingredients is **non-negative** and generate a run-time error if this is not true

MiniZinc provides a built-in boolean operator

- The form is *assert(b,s)*
- **constraint assert**(flour >= 0.0,"Amount of flour is negative");

# REAL NUMBER SOLVING

MiniZinc also supports “real number” constraint solving using floating point solving

Note that we declare a float variable  $f$  using **var float:  $f$**

- $f$  in a fixed range  $l$  to  $u$  with **var  $l..u$ :  $f$** , where  $l$  and  $u$  are floating point expressions

**Addition (+), subtraction (-), multiplication (\*) and floating point division (/).**

The built-in function **int2float** can be used to coerce integers to floating point numbers

Floating point functions for

- absolute value (**abs**), square root (**sqrt**), natural logarithm (**ln**), logarithm base 2 (**log2**), logarithm base 10 (**log10**), exponentiation of  $e$  (**exp**), sine (**sin**), cosine (**cos**), tangent (**tan**), arcsine (**asin**), arccosine (**acos**), arctangent (**atan**), and unary power (**pow**).

The syntax for arithmetic literals is standard. Example float literals are 1.05, 1.3e-5 and 1.3+E5



# DIFFERENT SOLVERS

Since we wish to use real number solving we need to use a different solver than the finite domain solver `gencode`

A suitable solver would be one that supports mixed integer linear programming (`org.minizinc.mip.osicbc`)

## DEMO:

- Show `loan.mzn` and `loan1.dzn`
- **Question 1:** if I borrow \$1000 at 4% and repay \$260 per quarter, how much do I end up owing?
  - `$ minimzinc --solver org.minizinc.mip.osicbc loan.mzn loan1.dzn`
- **Question 2:** if I want to borrow \$1000 at 4% and owe nothing at the end, how much do I need to repay?
  - `$ minimzinc --solver org.minizinc.mip.osicbc loan.mzn loan2.dzn`
- **Question 3:** if I can repay \$250 a quarter, how much can I borrow at 4% to end up owing nothing?
  - `$ minimzinc --solver org.minizinc.mip.osicbc loan.mzn loan3.dzn`

# ARRAYS

Almost always we are interested in building models where the **number of constraints and variables is dependent on the input data**. In order to do so we will usually use arrays

How to declare a finite number of elements in a 2-dim matrix

- `array[0..w,0..h] of var float: t;`

The index set of an array needs to be a fixed integer range (contiguous), or a fixed set expression whose value is an integer range (contiguous)

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

1-D arrays are initialized using a list

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

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

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

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

# ARRAYS: DEMO

Show code of the Laplace model of the steady state temperature of a sheet of metal.

# ENUMERATED TYPES: DEMO

Show code of the production planning.

This is a generalization of the baking problem with any kinds of resources and products.

This example makes use of built-in functions that take a one-dimensional array and aggregate the elements.

**forall** takes an array of boolean expressions and returns their logical conjunction

**forall( [a[i] != a[j] | i,j in 1..3 where i < j])** constrains the elements in a to be different. The list comprehension evaluates to [ a[1] != a[2], a[1] != a[3], a[2] != a[3] ] and so the forall function returns the logical conjunction  $a[1] \neq a[2] \wedge a[1] \neq a[3] \wedge a[2] \neq a[3]$

It can be written as **forall (i,j in 1..3 where i < j) (a[i] != a[j])**