



Minizinc Handbook

Release 2.2.3

Peter J. Stuckey, Kim Marriott, Guido Tack

Oct 31, 2018

Contents

1 Overview	3
1.1 Introduction	5
1.2 Installation	7
1.3 First steps with MiniZinc	11
2 A MiniZinc Tutorial	19
2.1 Basic Modelling in MiniZinc	21
2.2 More Complex Models	35
2.3 Predicates and Functions	61
2.4 Option Types	81
2.5 Search	85
2.6 Effective Modelling Practices in MiniZinc	95
2.7 Boolean Satisfiability Modelling in MiniZinc	107
2.8 FlatZinc and Flattening	117
3 User Manual	133
3.1 The MiniZinc Command Line Tool	135
3.2 The MiniZinc IDE	145
3.3 Globalizer	157
3.4 FindMUS	165
3.5 Using MiniZinc in Jupyter Notebooks	173
4 Reference Manual	177
4.1 Specification of MiniZinc	179
4.2 The MiniZinc library	235

4.3 Interfacing Solvers to Flatzinc 337

Index 357

MiniZinc is a free and open-source constraint modeling language. You can use MiniZinc to model constraint satisfaction and optimization problems in a high-level, solver-independent way, taking advantage of a large library of pre-defined constraints.

This handbook consists of four parts: [Section 1](#) covers installation and basic steps; [Section 2](#) is a tutorial-style introduction into modelling with MiniZinc; [Section 3](#) is a user manual for the individual tools in the MiniZinc tool chain; and [Section 4](#) is a reference to the language.

This documentation is licensed under a [Creative Commons Attribution-NoDerivatives 4.0 International License](#)⁶. This means that you are free to copy and redistribute the material in any medium or format for any purpose, even commercially. However, you must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. If you remix, transform, or build upon the material, you may **not** distribute the modified material.

⁶ <http://creativecommons.org/licenses/by-nd/4.0/>

Part 1

Overview

CHAPTER 1.1

Introduction

MiniZinc is a language for specifying constrained optimization and decision problems over integers and real numbers. A MiniZinc model does not dictate *how* to solve the problem - the MiniZinc compiler can translate it into different forms suitable for a wide range of *solvers*, such as Constraint Programming (CP), Mixed Integer Linear Programming (MIP) or Boolean Satisfiability (SAT) solvers.

The MiniZinc language lets users write models in a way that is close to a mathematical formulation of the problem, using familiar notation such as existential and universal quantifiers, sums over index sets, or logical connectives like implications and if-then-else statements. Furthermore, MiniZinc supports defining predicates and functions that let users structure their models (similar to procedures and functions in regular programming languages).

MiniZinc models are usually *parametric*, i.e., they describe a whole *class* of problems rather than an individual problem instance. That way, a model of, say, a vehicle routing problem could be reused to generate weekly plans, by instantiating it with the updated customer demands for the upcoming week.

MiniZinc is designed to interface easily to different backend solvers. It does this by transforming an input MiniZinc model and data file into a FlatZinc model. FlatZinc models consist of variable declarations 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 makes use of a library of function and predicate definitions for the particular target solver, which allows the MiniZinc compiler to produce specialised FlatZinc that only contains the types of variables and constraints that are supported by the target. In particular, MiniZinc allows the specification of *global constraints* by *decomposition*. Furthermore, *annotations* of the model let the user fine tune the behaviour of the solver, independent of the declarative meaning of the model.

1.1.1 Structure

This documentation consists of four parts. *The first part* (page 5) includes this introduction and then describes how to download and install MiniZinc and how to make your first steps using the MiniZinc IDE and the command line tools. *The second part* (page 21) is a tutorial

introduction to modelling with MiniZinc, from basic syntax and simple modelling techniques to more advanced topics. It also explains how MiniZinc is compiled to FlatZinc. *The third part* (page 135) is a user manual for the tools that make up the MiniZinc tool chain. Finally, *The fourth part* (page 179) contains the reference documentation for MiniZinc, including a definition of the MiniZinc language, documentation on how to interface a solver to FlatZinc, and an annotated list of all predicates and functions in the MiniZinc standard library.

1.1.2 How to Read This

If you are new to MiniZinc, follow the installation instructions and the introduction to the IDE and then work your way through the tutorial. Most example code can be downloaded, but it is sometimes more useful to type it in yourself to get the language into your muscle memory! If you need help, visit the MiniZinc web site at <http://www.minizinc.org> where you find a discussion forum.

Some of the code examples are shown in boxes like the one below. If a code box has a heading, it usually lists the name of a file that can be downloaded from <http://minizinc.org/doc-latest/en/downloads/index.html>.

Listing 1.1.1: A code example (`dummy.mzn`)

```
% Just an example
var int: x;
solve satisfy;
```

Throughout the documentation, some concepts are defined slightly more formally in special sections like this one.

More details

These sections can be skipped over if you just want to work through the tutorial for the first time, but they contain important information for any serious MiniZinc user!

Finally, if you find a mistake in this documentation, please report it through our GitHub issue tracker.

CHAPTER 1.2

Installation

A complete installation of the MiniZinc system comprises the MiniZinc *compiler tool chain*, one or more *solvers*, and (optionally) the *MiniZinc IDE*. We provide fully self-contained binary packages for all major operating systems that contain all of these components. Alternatively, it is possible to compile all components from source code.

1.2.1 Binary Packages

The easiest way to get a full, working MiniZinc system is to use the **bundled binary packages**, available from <http://www.minizinc.org/software.html>.

The bundled binary packages contain the compiler and IDE, as well as the following solvers: Gecode, Chuffed, COIN-OR CBC, and a Gurobi interface (the Gurobi library itself is not included). For backwards compatibility with older versions of MiniZinc, the packages also contain the now deprecated G12 suite of solvers (G12 fd, G12 lazy, G12 MIP).

1.2.1.1 Microsoft Windows

To install the bundled binary packages, simply download the installer, double-click to execute it, and follow the prompts. **Note:** you should select the 64 bit version of the installer if your Windows is a 64 bit operating system, otherwise pick the 32 bit version.

After installation is complete, you can find the MiniZinc IDE installed as a Windows application. The file extensions .mzn, .dzn and .fzn are linked to the IDE, so double-clicking any MiniZinc file should open it in the IDE.

If you want to use MiniZinc from a command prompt, you need to add the installation directory to the PATH environment variable. In a Windows command prompt you could use the following command:

```
C:\>setx PATH "%PATH%;C:\Program Files\MiniZinc 2.2.3 (bundled)\"
```

1.2.1.2 Linux

The MiniZinc bundled binary distribution for Linux is provided as an archive that contains everything that is needed to run MiniZinc. It was compiled on a Ubuntu 16.04 LTS system, but it bundles all required libraries except for the system C and C++ libraries (so it should be compatible with any Linux distribution that uses the same C and C++ libraries as Ubuntu 16.04). **Note:** you should select the 64 bit version of the installer if your Linux is a 64 bit operating system, otherwise pick the 32 bit version.

After downloading, uncompress the archive, for example in your home directory or any other location where you want to install it:

```
$ tar xf MiniZincIDE-2.2.3-bundle-linux-x86_64.tgz
```

This will unpack MiniZinc into a directory that is called the same as the archive file (without the .tgz). You can run the MiniZinc IDE or any of the command line tools directly from that directory, or add it to your PATH environment variable for easier access. **Note:** the MiniZinc IDE needs to be started using the `MiniZincIDE.sh` script, which sets up a number of paths that are required by the IDE.

1.2.1.3 Apple macOS

The macOS bundled binary distribution works with any version of OS X starting from 10.9. After downloading the disk image (.dmg) file, double click it if it doesn't open automatically. You will see an icon for the MiniZinc IDE that you can drag into your Applications folder (or anywhere else you want to install MiniZinc).

In order to use the MiniZinc tools from a terminal, you need to add the path to the MiniZinc installation to the PATH environment variable. If you installed the MiniZinc IDE in the standard Applications folder, the following command will add the correct path:

```
$ export PATH=/Applications/MiniZincIDE.app/Contents/Resources:$PATH
```

1.2.2 Compilation from Source Code

All components of MiniZinc are free and open source software, and compilation should be straightforward if you have all the necessary build tools installed. However, third-party components, in particular the different solvers, may be more difficult to install correctly, and we cannot provide any support for these components.

The source code for MiniZinc can be downloaded from its GitHub repository at <https://github.com/MiniZinc/libminizinc>. The source code for the MiniZinc IDE is available from <https://github.com/MiniZinc/MiniZincIDE>.

You will also need to install additional solvers to use with MiniZinc. To get started, try Gecode (<http://www.gecode.org>) or Chuffed (<https://github.com/chuffed/chuffed>). We don't cover installation instructions for these solvers here.

1.2.2.1 Microsoft Windows

Required development tools:

- CMake, version 3.0.0 or later (<http://cmake.org>)
- Microsoft Visual C++ 2013 or later (e.g. the Community Edition available from <https://www.visualstudio.com/de/downloads/>)
- Optional, only needed for MiniZinc IDE: Qt toolkit, version 5.4 or later (<http://qt.io>)

Compiling MiniZinc: Unpack the source code (or clone the git repository). Open a command prompt and change into the source code directory. The following sequence of commands will build a 64 bit version of the MiniZinc compiler tool chain (you may need to adapt the `cmake` command to fit your version of Visual Studio):

```
mkdir build
cd build
cmake -G"Visual Studio 14 2015 Win64" -DCMAKE_INSTALL_PREFIX="C:/Program_
Files/MiniZinc" ..
cmake --build . --config Release --target install
```

This will install MiniZinc in the usual Program Files location. You can change where it gets installed by modifying the `CMAKE_INSTALL_PREFIX`.

Compiling the MiniZinc IDE: Unpack the source code (or clone the git repository). Open a Visual Studio command prompt that matches the version of the Qt libraries installed on your system. Change into the source code directory for the MiniZinc IDE. Then use the following commands to compile:

```
mkdir build
cd build
qmake ..\MinizincIDE
nmake
```

1.2.2.2 Linux

Required development tools:

- CMake, version 3.0.0 or later
- A recent C++ compiler (g++ or clang)
- Optional, only needed for MiniZinc IDE: Qt toolkit, version 5.4 or later (<http://qt.io>)

Compiling MiniZinc: Unpack the source code (or clone the git repository). Open a terminal and change into the source code directory. The following sequence of commands will build the MiniZinc compiler tool chain:

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
```

```
cmake --build .
```

Compiling the MiniZinc IDE: Unpack the source code (or clone the git repository). Open a terminal and change into the source code directory for the MiniZinc IDE. Then use the following commands to compile:

```
mkdir build  
cd build  
qmake ../MinizincIDE  
make
```

1.2.2.3 Apple macOS

Required development tools:

- CMake, version 3.0.0 or later (from <http://cmake.org> or e.g. through homebrew)
- The Xcode developer tools
- Optional, only needed for MiniZinc IDE: Qt toolkit, version 5.4 or later (<http://qt.io>)

Compiling MiniZinc: Unpack the source code (or clone the git repository). Open a terminal and change into the source code directory. The following sequence of commands will build the MiniZinc compiler tool chain:

```
mkdir build  
cd build  
cmake -DCMAKE_BUILD_TYPE=Release ..  
cmake --build .
```

Compiling the MiniZinc IDE: Unpack the source code (or clone the git repository). Open a terminal and change into the source code directory for the MiniZinc IDE. Then use the following commands to compile:

```
mkdir build  
cd build  
qmake ../MinizincIDE  
make
```

1.2.3 Adding Third-party Solvers

Third party solvers for MiniZinc typically consist of two parts: a solver *executable*, and a solver-specific MiniZinc *library*. MiniZinc must be aware of the location of both the executable and the library in order to compile and run a model with that solver. Each solver therefore needs to provide a *configuration file* in a location where the MiniZinc toolchain can find it.

The easiest way to add a solver to the MiniZinc system is via the MiniZinc IDE. This is explained in Section 3.2.5.2. You can also add configuration files manually, as explained in Section 4.3.5.

CHAPTER 1.3

First steps with MiniZinc

We recommend using the bundled binary distribution of MiniZinc introduced in [Section 1.2](#). It contains the MiniZinc IDE, the MiniZinc compiler, and several pre-configured solvers so you can get started straight away.

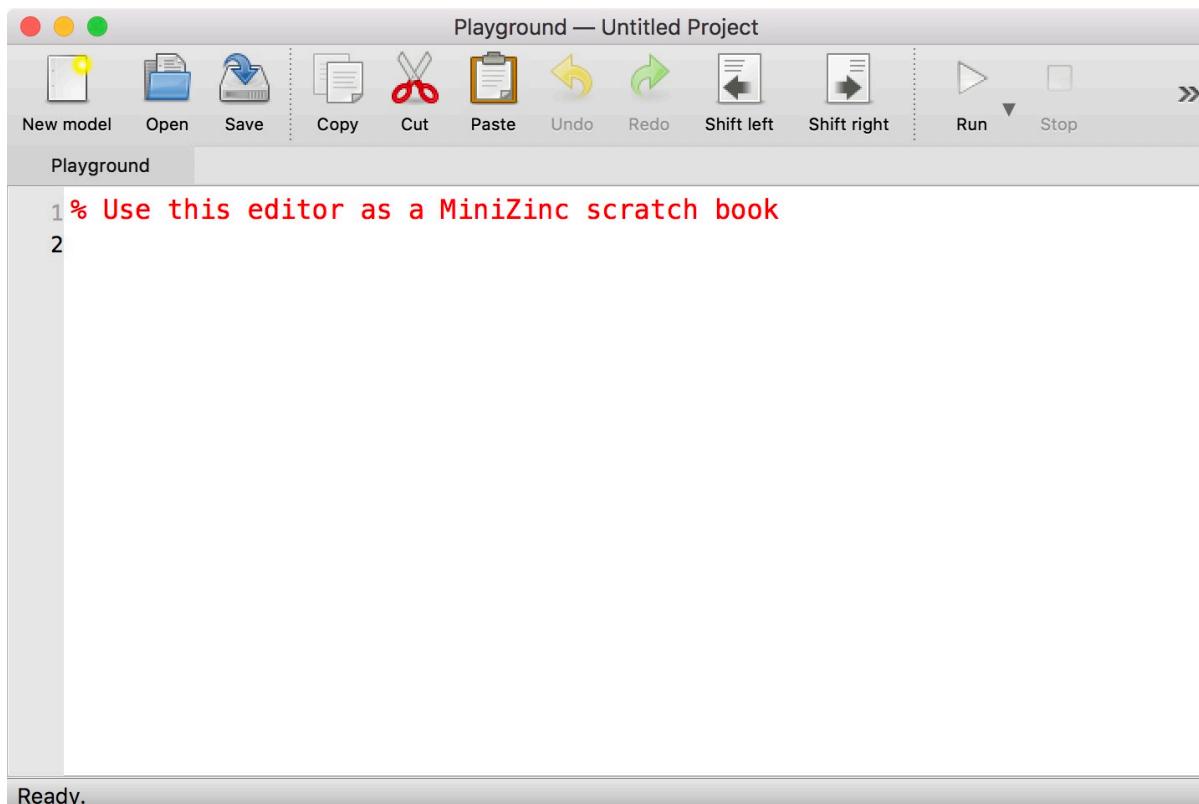
This section introduces the MiniZinc IDE and the command line tool `minizinc` using some very basic examples. This should be enough to get you started with the MiniZinc tutorial in [Section 2](#) (in fact, you only need to be able to use one of the two, for instance just stick to the IDE if you are not comfortable with command line tools, or just use the `minizinc` command if you suffer from fear of mice).

1.3.1 The MiniZinc IDE

The MiniZinc IDE provides a simple interface to most of MiniZinc's functionality. It lets you edit model and data files, solve them with any of the solvers supported by MiniZinc, run debugging and profiling tools, and submit solutions to online courses (such as the MiniZinc Coursera courses).

When you open the MiniZinc IDE for the first time, it will ask you whether you want to be notified when an update is available. If you installed the IDE from sources, it may next ask you to locate your installation of the MiniZinc compiler. Please refer to [Section 3.2.5](#) for more details on this.

The IDE will then greet you with the *MiniZinc Playground*, a window that will look like this:

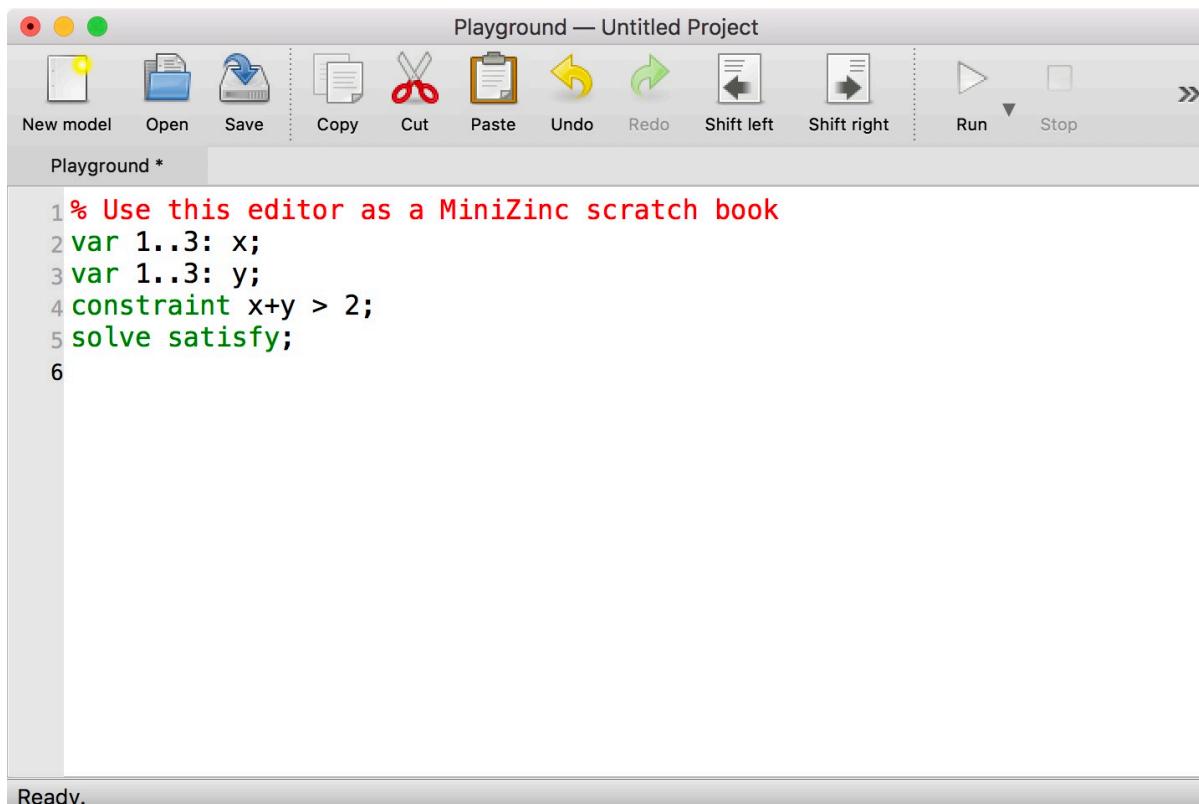


The screenshot shows the MiniZinc IDE's playground window titled "Playground — Untitled Project". The toolbar at the top includes standard file operations (New model, Open, Save), text editing (Copy, Cut, Paste, Undo, Redo), navigation (Shift left, Shift right), and execution controls (Run, Stop). The main editor area displays the following text:

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

In the bottom status bar, it says "Ready."

You can start writing your first MiniZinc model! Let's try something very simple:

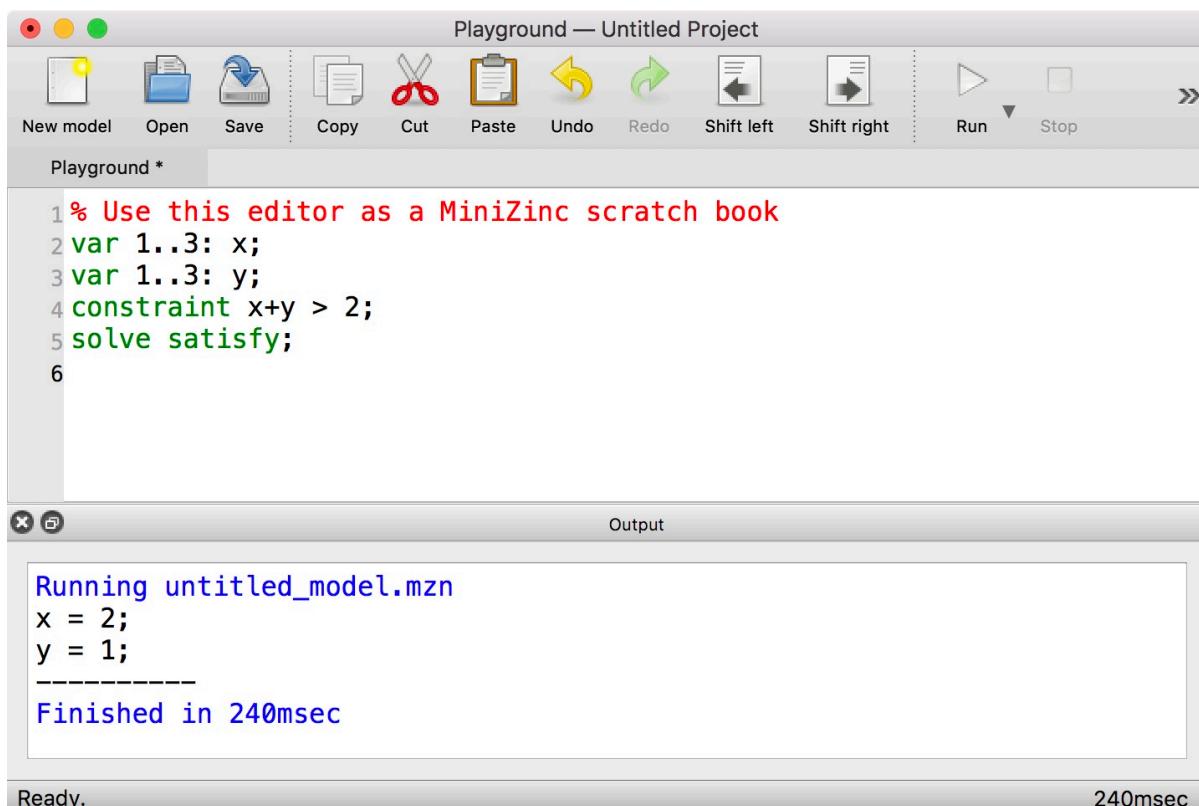


The screenshot shows the MiniZinc IDE's playground window titled "Playground — Untitled Project". The toolbar and editor area are identical to the previous screenshot. The main editor area displays the following text:

```
1 % Use this editor as a MiniZinc scratch book
2 var 1..3: x;
3 var 1..3: y;
4 constraint x+y > 2;
5 solve satisfy;
6
```

In the bottom status bar, it says "Ready."

In order to solve the model, you click on the *Run* button in the toolbar, or use the keyboard shortcut *Ctrl+R* (or *command+R* on macOS):



The screenshot shows the MiniZinc IDE interface. The toolbar at the top includes icons for New model, Open, Save, Copy, Cut, Paste, Undo, Redo, Shift left, Shift right, Run, and Stop. The title bar says "Playground — Untitled Project". The main editor window titled "Playground *" contains the following MiniZinc code:

```

1 % Use this editor as a MiniZinc scratch book
2 var 1..3: x;
3 var 1..3: y;
4 constraint x+y > 2;
5 solve satisfy;
6

```

The output window below shows the results of running the model:

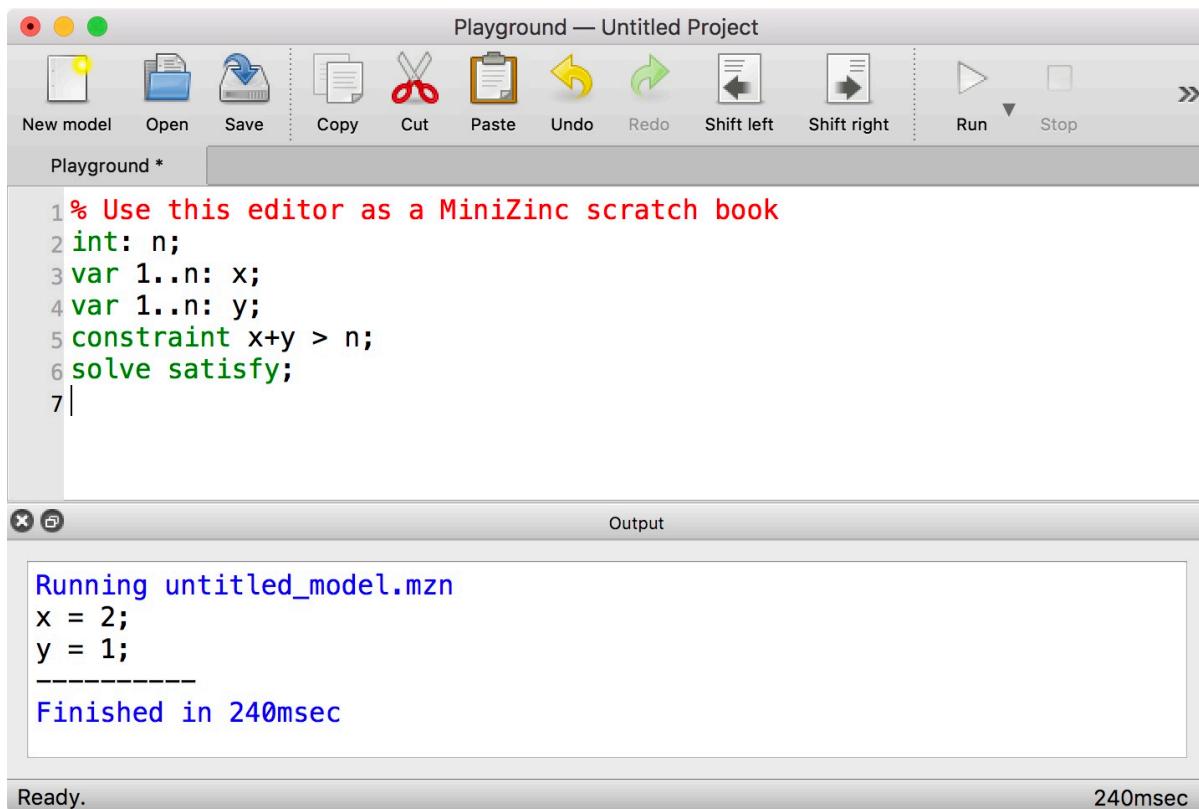
```

Running untitled_model.mzn
x = 2;
y = 1;
-----
Finished in 240msec

```

At the bottom, a status bar indicates "Ready." and "240msec".

As you can see, an output window pops up that displays a solution to the problem you entered. Let us now try a model that requires some additional data.



The screenshot shows the MiniZinc IDE interface. The toolbar and title bar are identical to the previous screenshot. The main editor window titled "Playground *" contains the following MiniZinc code:

```

1 % Use this editor as a MiniZinc scratch book
2 int: n;
3 var 1..n: x;
4 var 1..n: y;
5 constraint x+y > n;
6 solve satisfy;
7

```

The output window below shows the results of running the model:

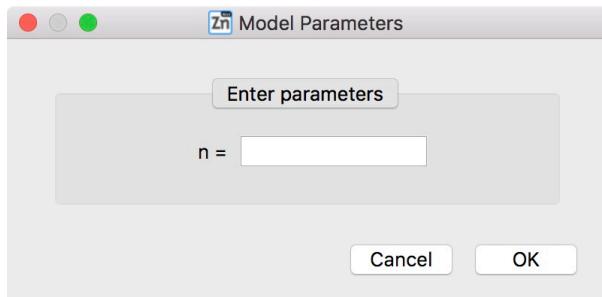
```

Running untitled_model.mzn
x = 2;
y = 1;
-----
Finished in 240msec

```

At the bottom, a status bar indicates "Ready." and "240msec".

When you run this model, the IDE will ask you to enter a value for the parameter n :



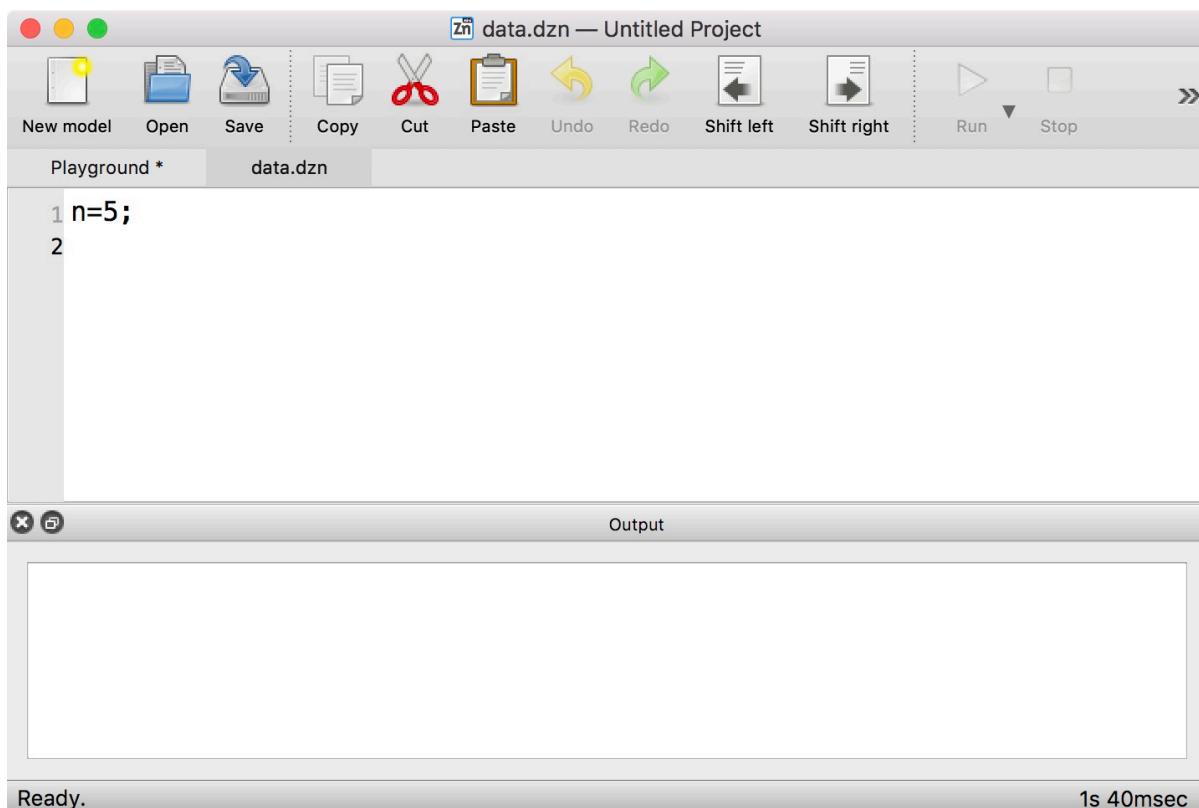
After entering, for example, the value 4 and clicking *Ok*, the solver will execute the model for $n=4$:

```
% Use this editor as a MiniZinc scratch book
int: n;
var 1..n: x;
var 1..n: y;
constraint x+y > n;
solve satisfy;
```

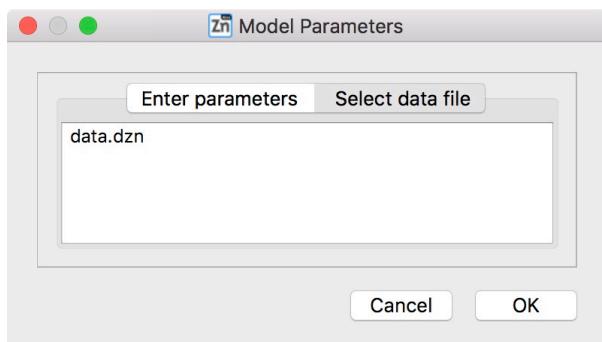
Running `untitled_model.mzn`, additional arguments `n=4`;
`x = 4;`
`y = 1;`

`Finished in 100msec`

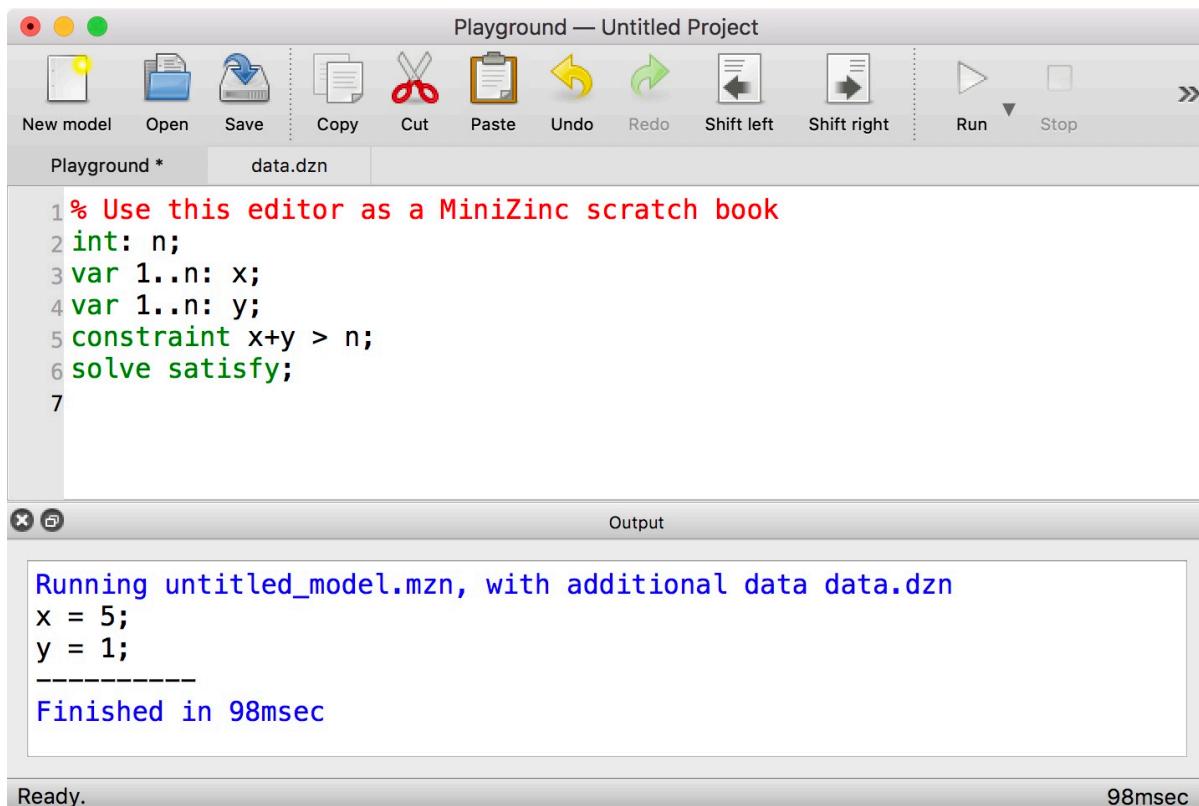
Alternatively, data can also come from a file. Let's create a new file with the data and save it as `data.dzn`:



When you now go back to the *Playground* tab and click *Run*, the IDE will give you the option to select a data file:



Click on the `data.dzn` entry, then on *Ok*, and the model will be run with the given data file:



Of course you can save your model to a file, and load it from a file, and the editor supports the usual functionality.

If you want to know more about the MiniZinc IDE, continue reading from [Section 3.2](#).

1.3.2 The MiniZinc command line tool

The MiniZinc command line tool, `minizinc`, combines the functionality of the MiniZinc compiler, different solver interfaces, and the MiniZinc output processor. After installing MiniZinc from the bundled binary distribution, you may have to set up your PATH in order to use the command line tool (see [Section 1.2](#)).

Let's assume we have a file `model.mzn` with the following contents:

```

var 1..3: x;
var 1..3: y;
constraint x+y > 3;
solve satisfy;

```

You can simply invoke `minizinc` on that file to solve the model and produce some output:

```

$ minizinc model.mzn
x = 3;
y = 1;
-----
$ 

```

If you have a model that requires a data file (like the one we used in the IDE example above), you pass both files to `minizinc`:

```
$ minizinc model.mzn data.dzn
x = 5;
y = 1;
-----
$
```

The `minizinc` tool supports numerous command line options. One of the most useful options is `-a`, which switches between *one solution* mode and *all solutions* mode. For example, for the first model above, it would result in the following output:

```
$ minizinc -a model.mzn
x = 3;
y = 1;
-----
x = 2;
y = 2;
-----
x = 3;
y = 2;
-----
x = 1;
y = 3;
-----
x = 2;
y = 3;
-----
x = 3;
y = 3;
-----
=====
$
```

To learn more about the `minizinc` command, explore the output of `minizinc --help` or continue reading in [Section 3.1](#).

Part 2

A MiniZinc Tutorial

CHAPTER 2.1

Basic Modelling in MiniZinc

In this section we introduce the basic structure of a MiniZinc model using two simple examples.

2.1.1 Our First Example

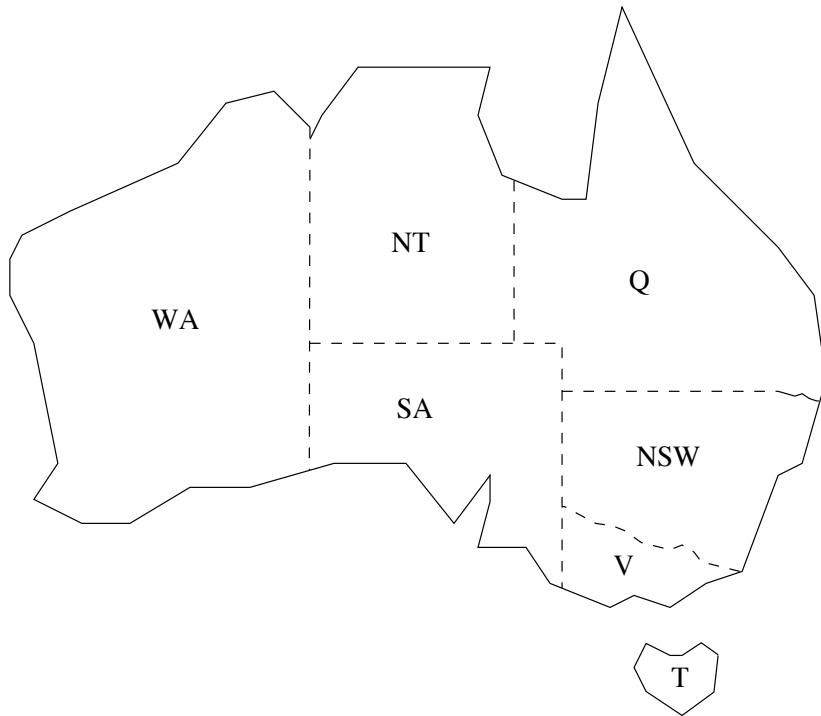


Fig. 2.1.1: Australian states

As our first example, imagine that we wish to colour a map of Australia as shown in Fig. 2.1.1. It is made up of seven different states and territories each of which must be given a colour so that adjacent regions have different colours.

Listing 2.1.1: A MiniZinc model aust.mzn for colouring the states and territories in Australia

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

We can model this problem very easily in MiniZinc. The model is shown in Listing 2.1.1. The first line in the model is a comment. A comment starts with a % which indicates that the rest of the line is a comment. MiniZinc also has C-style block comments, which start with /* and end with */.

The next part of the model declares the variables in the model. The line

```
int: nc = 3;
```

specifies a parameter in the problem which is the number of colours to be used. Parameters are similar to (constant) variables in most programming languages. They must be declared and given a type. In this case the type is `int`. They are given a value by an assignment. MiniZinc allows the assignment to be included as part of the declaration (as in the line above) or to be a separate assignment statement. Thus the following is equivalent to the single line above

```
int: nc;
nc = 3;
```

Unlike variables in many programming languages a parameter can only be given a *single* value, in that sense they are named constants. It is an error for a parameter to occur in more than one assignment.

The basic parameter types are integers (`int`), floating point numbers (`float`), Booleans (`bool`) and strings (`string`). Arrays and sets are also supported.

MiniZinc models can also contain another kind of variable called a *decision variable*. Decision

variables are variables in the sense of mathematical or logical variables. Unlike parameters and variables in a standard programming language, the modeller does not need to give them a value. Rather the value of a decision variable is unknown and it is only when the MiniZinc model is executed that the solving system determines if the decision variable can be assigned a value that satisfies the constraints in the model and if so what this is.

In our example 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*. This can be given as part of the variable declaration and 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, or sets. Also supported are arrays whose elements are decision variables. In our MiniZinc model we use integers to model the different colours. Thus each of our decision variables is declared to have the domain `1..nc` which is an integer range expression indicating the set $\{1, 2, \dots, nc\}$ using the `var` declaration. The type of the values is integer so all of the variables in the model are integer decision variables.

Identifiers

Identifiers, which are used to name parameters and variables, are sequences of lower and uppercase alphabetic characters, digits and the underscore `_` character. They must start with an alphabetic character. Thus `myName_2` is a valid identifier. MiniZinc *keywords* are not allowed to be used as identifier names, they are listed in [Identifiers](#) (page 183). Neither are MiniZinc *operators* allowed to be used as identifier names; they are listed in [Operators](#) (page 196).

MiniZinc carefully distinguishes between the two kinds of model variables: parameters and decision variables. The kinds of expressions that can be constructed using decision variables are more restricted than those that can be built from parameters. However, in any place that a decision variable can be used, so can a parameter of the same type.

Integer Variable Declarations

An integer parameter variable is declared as either:

```
int : <var-name>
<l> .. <u> : <var-name>
```

where `<l>` and `<u>` are fixed integer expressions.

An integer decision variable is declared as either:

```
var int : <var-name>
var <l>..<u> : <var-name>
```

where `<l>` and `<u>` are fixed integer expressions.

Formally the distinction between parameters and decision variables is called the *instantiation* of the variable. The combination of variable instantiation and type is called a *type-inst*. As you start to use MiniZinc you will undoubtedly see examples of *type-inst* errors.

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 this case 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.

Relational Operators

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

The next line in the model:

```
solve satisfy;
```

indicates the kind of problem it is. 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. This tells MiniZinc what to print when the model has been run and a solution is found.

Output and Strings

An output statement is followed by a *list* of strings. These are typically either string literals which are written between double quotes and use a C like notation for special characters, or an expression of the form `show(e)` where `e` is a MiniZinc expression. In the example `\n` represents the newline character and `\t` a tab.

There are also formatted varieties of `show` for numbers: `show_int(n, X)` outputs the value of integer `X` in at least $|n|$ characters, right justified if $n > 0$ and left justified otherwise; `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 `:math'd'` characters after the decimal point.

String literals must fit on a single line. Longer string literals can be split across multiple lines using the string concatenation operator `++`. For example, the string literal

```
"Invalid datafile: Amount of flour is non-negative"
```

is equivalent to the string literal expression

```
"Invalid datafile: " ++
"Amount of flour is non-negative"
```

MiniZinc supports interpolated strings. Expressions can be embedded directly in string literals, where a sub string of the form `"\(\e)"` is replaced by the result of `show(e)`. For example `"t=\(t)\n"` produces the same string as `"t=" ++ show(t) ++ "\n"`.

A model can contain multiple output statements. In that case, all outputs are concatenated in the order they appear in the model.

We can evaluate our model by clicking the *Run* button in the MiniZinc IDE, or by typing

```
$ minizinc --solver gecode aust.mzn
```

where `aust.mzn` is the name of the file containing our MiniZinc model. We must use the file extension `.mzn` to indicate a MiniZinc model. The command `minizinc` with the option `--solver`

gecode uses the Gecode finite domain solver to evaluate our model. If you use the MiniZinc binary distribution, this solver is in fact the default, so you can just run `minizinc aust.mzn` instead.

When we run this we obtain the result:

```
wa=3    nt=2    sa=1
q=3    nsw=2    v=3
t=1
-----
```

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

2.1.2 An Arithmetic Optimisation Example

Our second example is motivated by the need to bake some cakes for a fete at our local school. We know how to make two sorts of cakes (WARNING: please don't use these recipes at home). 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. The question is how many of each sort of cake should we bake for the fete to maximise the profit. A possible MiniZinc model is shown in Listing 2.1.2.

Listing 2.1.2: Model for determining how many banana and chocolate cakes to bake for the school fete (`cakes.mzn`)

```
% 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 = \$(b)\n",
        "no. of chocolate cakes = \$(c)\n"];
```

The first new feature is the use of *arithmetic expressions*.

Integer Arithmetic Operators

MiniZinc provides the standard integer arithmetic operators. Addition (+), subtraction (-), multiplication (*), integer division (`div`) and integer modulus (`mod`). It also provides + and - as unary operators.

Integer modulus is defined to give a result $a \bmod b$ that has the same sign as the dividend a . Integer division is defined so that $a = b(a \text{ div } b) + (a \bmod b)$.

MiniZinc provides standard integer functions for absolute value (`abs`) and power function (`pow`). For example `abs(-4)` and `pow(2,5)` evaluate to 4 and 32 respectively.

The syntax for arithmetic literals is reasonably standard. Integer literals can be decimal, hexadecimal or octal. For instance 0, 5, 123, 0x1b7, 0o777.

The second new feature shown in the example is optimisation. The line

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

specifies that 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 keyword `maximize` by `minimize` to specify a minimisation problem.

When we run this we obtain the result:

```
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.

2.1.3 Datafiles and Assertions

A drawback of this model is that if we wish to solve a similar problem the next time we need to bake cakes for the school (which is often) we need to modify the constraints in the model to reflect the ingredients that we have in the pantry. If we want to reuse the model then we would be better off to make the amount of each ingredient a parameter of the model and then set their values at the top of the model.

Even better would be to set the value of these parameters in a separate *data file*. MiniZinc (like most other modelling languages) allows the use of data files to set the value of parameters declared in the original model. This allows the same model to be easily used with different data by running it with different data files.

Data files must have the file extension `.dzn` to indicate a MiniZinc data file and a model can be run with any number of data files (though a variable/parameter can only be assigned a value in one file).

Listing 2.1.3: Data-independent model for determining how many banana and chocolate cakes to bake for the school fete (cakes2.mzn)

```
% 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

constraint assert(flour >= 0, "Invalid datafile: " ++
                  "Amount of flour should be non-negative");
constraint assert(banana >= 0, "Invalid datafile: " ++
                  "Amount of banana should be non-negative");
constraint assert(sugar >= 0, "Invalid datafile: " ++
                  "Amount of sugar should be non-negative");
constraint assert(butter >= 0, "Invalid datafile: " ++
                  "Amount of butter should be non-negative");
constraint assert(cocoa >= 0, "Invalid datafile: " ++
                  "Amount of cocoa should be non-negative");

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

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

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

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

Our new model is shown in Listing 2.1.3. We can run it using the command

where the data file `pantry.dzn` is defined in Listing 2.1.4. This gives the same result as `cakes.mzn`. The output from running the command

```
$ minizinc cakes2.mzn pantry2.dzn
```

with an alternate data set defined in Listing 2.1.5 is

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

If we remove the output statement from `cakes.mzn` then MiniZinc will use a default output. In this case the resulting output will be

```
b = 3;
c = 8;
-----
=====
```

Default Output

A MiniZinc model with no output will output a line for each decision variable with its value, unless it is assigned an expression on its declaration. Note how the output is in the form of a correct datafile.

Listing 2.1.4: Example data file for `cakes2.mzn` (`pantry.dzn`)

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

Listing 2.1.5: Example data file for `cakes2.mzn` (`pantry2.dzn`)

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

Small data files can be entered without directly creating a `.dzn` file, using the command line flag `-D string`, where `string` is the contents of the data file. For example the command

```
$ minizinc cakes2.mzn -D \
"flour=4000;banana=6;sugar=2000;butter=500;cocoa=500;"
```

will give identical results to

```
$ minizinc cakes2.mzn pantry.dzn
```

Data files can only contain assignment statements for decision variables and parameters in the model(s) for which they are intended.

Defensive programming suggests that we should check that the values in the data file are reasonable. For our example it is sensible 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 for checking parameter values. The form is `assert(B, S)`. The Boolean expression B is evaluated and if it is false execution aborts and the string expression S is evaluated and printed as an error message. To check and generate an appropriate error message if the amount of flour is negative we can simply add the line

```
constraint assert(flour >= 0, "Amount of flour is non-negative");
```

to our model. Notice that the `assert` expression is a Boolean expression and so is regarded as a type of constraint. We can add similar lines to check that the quantity of the other ingredients is non-negative.

2.1.4 Real Number Solving

MiniZinc also supports “real number” constraint solving using floating point variables and constraints. Consider a problem of taking out a short loan for one year to be repaid in 4 quarterly instalments. A model for this is shown in Listing 2.1.6. It uses a simple interest calculation to calculate the balance after each quarter.

Listing 2.1.6: Model for determining relationships between a 1 year loan repaying every quarter (loan.mzn)

```
% variables
var float: R;           % quarterly repayment
var float: P;           % principal initially borrowed
var 0.0 .. 10.0: 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 owing 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 [
  "Borrowing ", show_float(0, 2, P), " at ", show(I*100.0),
  "% interest, and repaying ", show_float(0, 2, R),
  "\nper quarter for 1 year leaves ", show_float(0, 2, B4), " owing\n"
];
```

Note that we declare a float variable f similar to an integer variable using the keyword `float`

instead of `int`.

We can use the same model to answer a number of different questions. The first question is: if I borrow \$1000 at 4% and repay \$260 per quarter, how much do I end up owing? This question is encoded by the data file `loan1.dzn`.

Since we wish to use real number variables and constraint we need to use a solver that supports this type of problem. While Gecode (the default solver in the MiniZinc bundled binary distribution) does support floating point variables, a mixed integer linear programming (MIP) solver may be better suited to this particular type of problem. The MiniZinc distribution contains such a solver. We can invoke it by selecting OSICBC from the solver menu in the IDE (the triangle below the *Run* button), or on the command line using the command `minizinc --solver osicbc`:

```
$ minizinc --solver osicbc loan.mzn loan1.dzn
```

The output is

```
Borrowing 1000.00 at 4.0% interest, and repaying 260.00  
per quarter for 1 year leaves 65.78 owing
```

The second question is if I want to borrow \$1000 at 4% and owe nothing at the end, how much do I need to repay? This question is encoded by the data file `loan2.dzn`. The output from running the command

```
$ minizinc --solver osicbc loan.mzn loan2.dzn
```

is

```
Borrowing 1000.00 at 4.0% interest, and repaying 275.49  
per quarter for 1 year leaves 0.00 owing
```

The third question is if I can repay \$250 a quarter, how much can I borrow at 4% to end up owing nothing? This question is encoded by the data file `loan3.dzn`. The output from running the command

```
$ minizinc --solver osicbc loan.mzn loan3.dzn
```

is

```
Borrowing 907.47 at 4.0% interest, and repaying 250.00  
per quarter for 1 year leaves 0.00 owing
```

Listing 2.1.7: Example data file for loan.mzn (loan1.dzn)

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

Listing 2.1.8: Example data file for loan.mzn (loan2.dzn)

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

Listing 2.1.9: Example data file for loan.mzn (loan3.dzn)

```
I = 0.04;
R = 250.0;
B4 = 0.0;
```

Float Arithmetic Operators

MiniZinc provides the standard floating point arithmetic operators: addition (+), subtraction (-), multiplication (*) and floating point division (/). It also provides + and - as unary operators.

MiniZinc can automatically coerce integers to floating point numbers. But to make the coercion explicit, the built-in function `int2float` can be used. Note that one consequence of the automatic coercion is that an expression `a / b` is always considered a floating point division. If you need an integer division, make sure to use the `div` operator!

MiniZinc provides in addition the following floating point functions: 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`), arc-sine (`asin`), arc-cosine (`acos`), arctangent (`atan`), hyperbolic sine (`sinh`), hyperbolic cosine (`cosh`), hyperbolic tangent (`tanh`), hyperbolic arcsine (`asinh`), hyperbolic arccosine (`acosh`), hyperbolic arctangent (`atanh`), and power (`pow`) which is the only binary function, the rest are unary.

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

2.1.5 Basic structure of a model

We are now in a position to summarise the basic structure of a MiniZinc model. It consists of multiple *items* each of which has a semicolon ; at its end. Items can occur in any order. For example, identifiers need not be declared before they are used.

There are 8 kinds of items.

- Include items allow the contents of another file to be inserted into the model. They have the form:

```
include <filename>;
```

where `<filename>` is a string literal. They allow large models to be split into smaller sub-models and also the inclusion of constraints defined in library files. We shall see an example in [Listing 2.2.4](#).

- Variable declarations declare new variables. Such variables are global variables and can be referred to from anywhere in the model. Variables come in two kinds. Parameters which are assigned a fixed value in the model or in a data file and decision variables whose value is found only when the model is solved. We say that parameters are fixed and decision variables unfixed. The variable can be optionally assigned a value as part of the declaration. The form is:

```
<type inst expr>: <variable> [ = ] <expression>;
```

The `<type-inst expr>` gives the instantiation and type of the variable. These are one of the more complex aspects of MiniZinc. Instantiations are declared using `par` for parameters and `var` for decision variables. If there is no explicit instantiation declaration then the variable is a parameter. The type can be a base type, an integer or float range or an array or a set. The base types are `float`, `int`, `string`, `bool`, `ann` of which only `float`, `int` and `bool` can be used for decision variables. The base type `ann` is an annotation – we shall discuss annotations in [Search](#) (page 85). Integer range expressions can be used instead of the type `int`. Similarly float range expressions can be used instead of type `float`. These are typically used to give the domain of a decision variable but can also be used to restrict the range of a parameter. Another use of variable declarations is to define enumerated types, which we discuss in [Enumerated Types](#) (page 48).

- Assignment items assign a value to a variable. They have the form:

```
<variable> = <expression>;
```

Values can be assigned to decision variables in which case the assignment is equivalent to writing `constraint <variable> = <expression>`.

- Constraint items form the heart of the model. They have the form:

```
constraint <Boolean expression>;
```

We have already seen examples of simple constraints using arithmetic comparison and the built-in `assert` operator. In the next section we shall see examples of more complex constraints.

- Solve items specify exactly what kind of solution is being looked for. As we have seen they have one of three forms:

```
solve satisfy;
solve maximize <arithmetic expression>;
solve minimize <arithmetic expression>;
```

A model is required to have at most one solve item. If its omitted it is treated as [solve satisfy](#).

- Output items are for nicely presenting the results of the model execution. They have the form:

```
output [ <string expression>, ..., <string expression> ];
```

If there is no output item, MiniZinc will by default print out the values of all the decision variables which are not optionally assigned a value in the format of assignment items.

- Enumerated type declarations. We discuss these in [Arrays and Sets](#) (page 35) and [Enumerated Types](#) (page 48).
- Predicate, function and test items are for defining new constraints, functions and Boolean tests. We discuss these in [Predicates and Functions](#) (page 61).
- The annotation item is used to define a new annotation. We discuss these in [Search](#) (page 85).

CHAPTER 2.2

More Complex Models

In the last section we introduced the basic structure of a MiniZinc model. In this section we introduce the array and set data structures, enumerated types and more complex constraints.

2.2.1 Arrays and Sets

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.

Consider a simple finite element model for modelling temperatures on a rectangular sheet of metal. We approximate the temperatures across the sheet by breaking the sheet into a finite number of elements in a two-dimensional matrix. A model is shown in Listing 2.2.1. It declares the width w and height h of the finite element model. The declaration

```
set of int: HEIGHT = 0..h;
set of int: CHEIGHT = 1..h-1;
set of int: WIDTH = 0..w;
set of int: CWIDTH = 1..w-1;
array[HEIGHT,WIDTH] of var float: t; % temperature at point (i,j)
```

declares four fixed sets of integers describing the dimensions of the finite element model: `HEIGHT` is the whole height of the model, while `CHEIGHT` is the centre of the height omitting the top and bottom, `WIDTH` is the whole width of the model, while `CWIDTH` is the centre of the width omitting the left and rightsides, Finally a two dimensional array of float variables `t` with rows numbered 0 to h (`HEIGHT`) and columns 0 to w (`WIDTH`), to represent the temperatures at each point in the metal plate. We can access the element of the array in the i^{th} row and j^{th} column using an expression `t[i,j]`.

Laplace's equation states that when the plate reaches a steady state the temperature at each internal point is the average of its orthogonal neighbours. The constraint

```
% Laplace equation: each internal temp. is average of its neighbours
constraint forall(i in CHEIGHT, j in CWIDTH)(
    4.0*t[i,j] = t[i-1,j] + t[i,j-1] + t[i+1,j] + t[i,j+1]);
```

ensures that each internal point (i, j) is the average of its four orthogonal neighbours. The constraints

```
% edge constraints
constraint forall(i in CHEIGHT)(t[i,0] = left);
constraint forall(i in CHEIGHT)(t[i,w] = right);
constraint forall(j in CWIDTH)(t[0,j] = top);
constraint forall(j in CWIDTH)(t[h,j] = bottom);
```

restrict the temperatures on each edge to be equal, and gives these temperatures names: `left`, `right`, `top` and `bottom`. While the constraints

```
% corner constraints
constraint t[0,0]=0.0;
constraint t[0,w]=0.0;
constraint t[h,0]=0.0;
constraint t[h,w]=0.0;
```

ensure that the corners (which are irrelevant) are set to 0.0. We can determine the temperatures in a plate broken into 5×5 elements with left, right and bottom temperature 0 and top temperature 100 with the model shown in Listing 2.2.1.

Listing 2.2.1: Finite element plate model for determining steady state temperatures (`laplace.mzn`).

```
int: w = 4;
int: h = 4;

% arraydec
set of int: HEIGHT = 0..h;
set of int: CHEIGHT = 1..h-1;
set of int: WIDTH = 0..w;
set of int: CWIDTH = 1..w-1;
array[HEIGHT,WIDTH] of var float: t; % temperature at point (i,j)
var float: left; % left edge temperature
var float: right; % right edge temperature
var float: top; % top edge temperature
var float: bottom; % bottom edge temperature

% equation
% Laplace equation: each internal temp. is average of its neighbours
constraint forall(i in CHEIGHT, j in CWIDTH)(
    4.0*t[i,j] = t[i-1,j] + t[i,j-1] + t[i+1,j] + t[i,j+1]);
% sides
% edge constraints
```

```

constraint forall(i in CHEIGHT)(t[i,0] = left);
constraint forall(i in CHEIGHT)(t[i,w] = right);
constraint forall(j in CWIDTH)(t[0,j] = top);
constraint forall(j in CWIDTH)(t[h,j] = bottom);
% corners
% corner constraints
constraint t[0,0]=0.0;
constraint t[0,w]=0.0;
constraint t[h,0]=0.0;
constraint t[h,w]=0.0;
left = 0.0;
right = 0.0;
top = 100.0;
bottom = 0.0;

solve satisfy;

output [ show_float(6, 2, t[i,j]) ++
        if j == h then "\n" else " " endif |
        i in HEIGHT, j in WIDTH
];

```

Running the command

```
$ minizinc --solver osicbc laplace.mzn
```

gives the output

```

-0.00 100.00 100.00 100.00 -0.00
-0.00 42.86 52.68 42.86 -0.00
-0.00 18.75 25.00 18.75 -0.00
-0.00 7.14 9.82 7.14 -0.00
-0.00 -0.00 -0.00 -0.00 -0.00
-----
```

Sets

Set variables are declared with a declaration of the form

```
set of <type-inst> : <var-name> ;
```

where sets of integers, enums (see later), floats or Booleans are allowed. The only type allowed for decision variable sets are variable sets of integers or enums. Set literals are of the form

```
{ <expr-1>, ..., <expr-n> }
```

or are range expressions over either integers, enums or floats of the form

```
<expr-1> .. <expr-2>
```

The standard set operations are provided: element membership ([in](#)), (non-strict) subset relationship ([subset](#)), (non-strict) superset relationship ([superset](#)), union ([union](#)), intersection ([intersect](#)), set difference ([diff](#)), symmetric set difference ([syndiff](#)) and the number of elements in the set ([card](#)).

As we have seen set variables and set literals (including ranges) can be used as an implicit type in variable declarations in which case the variable has the type of the elements in the set and the variable is implicitly constrained to be a member of the set.

Our cake baking problem is an example of a very simple kind of production planning problem. In this kind of problem we wish to determine how much of each kind of product to make to maximise the profit where manufacturing a product consumes varying amounts of some fixed resources. We can generalise the MiniZinc model in [Listing 2.1.3](#) to handle this kind of problem with a model that is generic in the kinds of resources and products. The model is shown in [Listing 2.2.2](#) and a sample data file (for the cake baking example) is shown in [Listing 2.2.3](#).

Listing 2.2.2: Model for simple production planning (prod-planning.mzn).

```
% Products to be produced
enum Products;
% profit per unit for each product
array[Products] of int: profit;
% Resources to be used
enum Resources;
% 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;
constraint assert(forall (r in Resources, p in Products)
    (consumption[p,r] >= 0), "Error: negative consumption");

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

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

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

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

output [ "\(p) = \$(produce[p]);\n" | p in Products ] ++
    [ "\(r) = \$(used[r]);\n" | r in Resources ];
```

Listing 2.2.3: Example data file for the simple production planning problem (prod-planning-data.dzn).

```
% Data file for simple production planning model
Products = { BananaCake, ChocolateCake };
profit = [400, 450]; % in cents

Resources = { Flour, Banana, Sugar, Butter, Cocoa };
capacity = [4000, 6, 2000, 500, 500];

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

The new feature in this model is the use of enumerated types. These allow us to treat the choice of resources and products as parameters to the model. The first item in the model

```
enum Products;
```

declares `Products` as an *unknown* set of products.

Enumerated Types

Enumerated types, which we shall refer to as enums, are declared with a declaration of the form

```
enum <var-name> ;
```

An enumerated type is defined by an assignment of the form

```
enum <var-name> = { <var-name-1>, ..., <var-name-n> } ;
```

where `<var-name-1>`, ..., `<var-name-n>` are the elements of the enumerated type, with name `<var-name>`. Each of the elements of the enumerated type is also effectively declared by this definition as a new constant of that type. The declaration and definition can be combined into one line as usual.

The second item declares an array of integers:

```
array[Products] of int: profit;
```

The index set of the array `profit` is `Products`. This means that only elements of the set `Products` can be used to index the array.

The elements of an enumerated type of n elements act very similar to the integers $1 \dots n$. They can be compared, they are ordered, by the order they appear in the enumerated type definition, they can be iterated over, they can appear as indices of arrays, in fact they can appear anywhere an integer can appear.

In the example data file we have initialized the array using a list of integers

```
Products = { BananaCake, ChocolateCake };
profit = [400, 450];
```

meaning the profit for a banana cake is 400, while for a chocolate cake it is 450. Internally `BananaCake` will be treated like the integer 1, while `ChocolateCake` will be treated like the integer 2. While MiniZinc does not provide an explicit list type, one-dimensional arrays with an index set `1..n` behave like lists, and we will sometimes refer to them as lists.

In a similar fashion, in the next two items we declare a set of resources `Resources`, and an array `capacity` which gives the amount of each resource that is available.

More interestingly, the item

```
array[Products, Resources] of int: consumption;
```

declares a 2-D array `consumption`. The value of `consumption[p, r]` is the amount of resource `r` required to produce one unit of product `p`. Note that the first index is the row and the second is the column.

The data file contains an example initialization of a 2-D array:

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

Notice how the delimiter | is used to separate rows.

Arrays

Thus, MiniZinc provides one- and multi-dimensional arrays which are declared using the type:

```
array [ <index-set-1>, ..., <index-set-n> ] of <type-inst>
```

MiniZinc requires that the array declaration contains the index set of each dimension and that the index set is either an integer range, a set variable initialised to an integer range, or an enumeration type. Arrays can contain any of the base types: integers, enums, Booleans, floats or strings. These can be fixed or unfixed except for strings which can only be parameters. Arrays can also contain sets but they cannot contain arrays.

One-dimensional array literals are of form

```
[<expr-1>, ..., <expr-n>]
```

while two-dimensional array literals are of form

```
[| <expr-1-1>, ..., <expr-1-n> |
   ...
   | <expr-m-1>, ..., <expr-m-n> |]
```

where the array has m rows and n columns.

The family of built-in functions `array1d`, `array2d`, etc, can be used to initialise an array of any dimension from a list (or more exactly a one-dimensional array). The call:

```
array<n>d(<index-set-1>, ..., <index-set-n>, <list>)
```

returns an n dimensional array with index sets given by the first n arguments and the last argument contains the elements of the array. For instance, `array2d(1..3, 1..2, [1, 2, 3, 4, 5, 6])` is equivalent to `[|1, 2 |3, 4 |5, 6|]`.

Array elements are accessed in the usual way: `a[i, j]` gives the element in the *ith* row and *jth* column.

The concatenation operator ++ can be used to concatenate two one-dimensional arrays together. The result is a list, i.e. a one-dimensional array whose elements are indexed from 1. For instance `[4000, 6] ++ [2000, 500, 500]` evaluates to `[4000, 6, 2000, 500, 500]`. The built-in function `length` returns the length of a one-dimensional array.

The next item in the model defines the parameter `mproducts`. This is set to an upper-bound on the number of products of any type that can be produced. This is quite a complex example of nested array comprehensions and aggregation operators. We shall introduce these before we try to understand this item and the rest of the model.

First, MiniZinc provides list comprehensions similar to those provided in many functional programming languages, or Python. For example, the list comprehension

`[i + j | i, j in 1..3 where j < i]` evaluates to `[1 + 2, 1 + 3, 2 + 3]` which is `[3, 4, 5]`. Of course `[3, 4, 5]` is simply an array with index set `1..3`.

MiniZinc also provides set comprehensions which have a similar syntax: for instance, `{i + j | i, j in 1..3 where j < i}` evaluates to the set `{3, 4, 5}`.

List and Set Comprehensions

The generic form of a list comprehension is

```
[ <expr> | <generator-exp> ]
```

The expression `<expr>` specifies how to construct elements in the output list from the elements generated by `<generator-exp>`. The generator `<generator-exp>` consists of a comma separated sequence of generator expressions optionally followed by a Boolean expression. The two forms are

```
<generator>
<generator> where <bool-exp>
```

The optional `<bool-exp>` in the second form acts as a filter on the generator expression: only elements satisfying the Boolean expression are used to construct elements in the output list. A generator `<generator>` has the form

```
<identifier>, ..., <identifier> in <array-exp>
```

Each identifier is an *iterator* which takes the values of the array expression in turn, with the last identifier varying most rapidly.

The generators of a list comprehension and `<bool-exp>` usually do not involve decision variables. If they do involve decision variables then the list produced is a list of `var opt <T>` where `<T>` is the type of the `<expr>`. See the discussion of option types in [Option Types](#) (page 81) for more details.

Set comprehensions are almost identical to list comprehensions: the only difference is the use of `{` and `}` to enclose the expression rather than `[` and `]`. The elements generated by a set comprehension must be fixed, i.e. free of decision variables. Similarly the generators and optional `<bool-exp>` for set comprehensions must be fixed.

Second, MiniZinc provides a number of built-in functions that take a one-dimensional array and aggregate the elements. Probably the most useful of these is `:mzn:forall`. This takes an array of Boolean expressions (that is, constraints) and returns a single Boolean expression which is the logical conjunction of the Boolean expressions in the array.

For example, consider the expression

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

where `a` is an arithmetic array with index set `1..3`. This constrains the elements in `a` to be pairwise 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] != a[2] /\ a[1] != a[3] /\ a[2] != a[3]`.

Aggregation functions

The *aggregation functions* for arithmetic arrays are: `sum` which adds the elements, `product` which multiplies them together, and `min` and `max` which respectively return the least and greatest element in the array. When applied to an empty array, `min` and `max` give a run-time error, `sum` returns 0 and `product` returns 1.

MiniZinc provides four aggregation functions for arrays containing Boolean expressions. As we have seen, the first of these, `forall`, returns a single constraint which is the logical conjunction of the constraints. The second function, `exists`, returns the logical disjunction of the constraints. Thus, `forall` enforces that all constraints in the array hold, while `exists` ensures that at least one of the constraints holds. The third function, `xorall`, ensures that an odd number of constraints hold. The fourth function, `iffall`, ensures that an even number of constraints holds.

The third, and final, piece in the puzzle is that MiniZinc allows a special syntax for aggregation functions when used with an array comprehension. Instead of writing

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

the modeller can instead write the more mathematical looking

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

The two expressions are completely equivalent: the modeller is free to use whichever they feel looks most natural.

Generator call expressions

A *generator call expression* has form

```
<agg-func> ( <generator-exp> ) ( <exp> )
```

The round brackets around the generator expression `<generator-exp>` and the constructor expression `<exp>` are not optional: they must be there. This is equivalent to writing

```
<agg-func> ( [ <exp> | <generator-exp> ] )
```

The aggregation function `<agg-func>` is any MiniZinc function expecting a single array as argument.

We are now in a position to understand the rest of the simple production planning model shown in Listing 2.2.2. For the moment ignore the item defining `mproducts`. The item afterwards:

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

defines a one-dimensional array `produce` of decision variables. The value of `produce[p]` will be set to the amount of product `p` in the optimal solution. The next item

```
array[Resources] of var 0..max(capacity): used;
```

defines a set of auxiliary variables that record how much of each resource is used. The next two constraints

```
constraint forall (r in Resources)
    (used[r] = sum (p in Products) (consumption[p, r] * produce[p]));
constraint forall (r in Resources)(used[r] <= capacity[r] );
```

compute in `used[r]` the total consumption of the resource `r` and ensure it is less than the available amount. Finally, the item

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

indicates that this is a maximisation problem and that the objective to be maximised is the total profit.

We now return to the definition of `mproducts`. For each product `p` the expression

```
(min (r in Resources where consumption[p,r] > 0)
    (capacity[r] div consumption[p,r]))
```

determines the maximum amount of `p` that can be produced taking into account the amount of each resource `r` and how much of `r` is required to produce the product. Notice the use of the filter `where consumption[p,r] > 0` to ensure that only resources required to make the product are considered so as to avoid a division by zero error. Thus, the complete expression

```
int: mproducts = max (p in Products)
    (min (r in Resources where consumption[p,r] > 0)
        (capacity[r] div consumption[p,r]));
```

computes the maximum amount of *any* product that can be produced, and so this can be used as an upper bound on the domain of the decision variables in `produce`.

Finally notice the output item is more complex, and uses list comprehensions to create an understandable output. Running

```
$ minizinc --solver gecode prod-planning.mzn prod-planning-data.dzn
```

results in the output

```
BananaCake = 2;
ChocolateCake = 2;
Flour = 900;
Banana = 4;
Sugar = 450;
Butter = 500;
Cocoa = 150;
-----
=====
```

2.2.2 Global Constraints

MiniZinc includes a library of global constraints which can also be used to define models. An example is the `alldifferent` constraint which requires all the variables appearing in its argument to be pairwise different.

Listing 2.2.4: Model for the cryptarithmetic problem SEND+MORE=MONEY
(`send-more-money.mzn`)

```
include "alldifferent.mzn";

var 1..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: R;
var 0..9: Y;

constraint      1000 * S + 100 * E + 10 * N + D
                + 1000 * M + 100 * O + 10 * R + E
                = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;

constraint alldifferent([S,E,N,D,M,O,R,Y]);

solve satisfy;

output ["  \$(S)\$(E)\$(N)\$(D)\n",
        "+  \$(M)\$(O)\$(R)\$(E)\n",
        "= \$(M)\$(O)\$(N)\$(E)\$(Y)\n"];
```

The SEND+MORE=MONEY problem requires assigning a different digit to each letter so that the arithmetic constraint holds. The model shown in Listing 2.2.4 uses the constraint expression `alldifferent([S,E,N,D,M,O,R,Y])` to ensure that each letter takes a different digit value. The global constraint is made available in the model using include item

```
include "alldifferent.mzn";
```

which makes the global constraint `alldifferent` usable by the model. One could replace this line by

```
include "globals.mzn";
```

which includes all globals.

A list of all the global constraints defined for MiniZinc is included in the release documentation. See [Global Constraints](#) (page 61) for a description of some important global constraints.

2.2.3 Conditional Expressions

MiniZinc provides a conditional *if-then-else-endif* expression. An example of its use is

```
int: r = if y != 0 then x div y else 0 endif;
```

which sets *r* to *x* divided by *y* unless *y* is zero in which case it sets it to zero.

Conditional expressions

The form of a conditional expression is

```
if <bool-exp> then <exp-1> else <exp-2> endif
```

It is a true expression rather than a control flow statement and so can be used in other expressions. It evaluates to *<exp-1>* if the Boolean expression *<bool-exp>* is true and *<exp-2>* otherwise. The type of the conditional expression is that of *<exp-1>* and *<exp-2>* which must have the same type.

If the *<bool-exp>* contains decision variables, then the type-inst of the expression is *var <T>* where *<T>* is the type of *<exp-1>* and *<exp-2>* even if both expressions are fixed.

Listing 2.2.5: Model for generalized Sudoku problem (sudoku.mzn)

```
include "alldifferent.mzn";

int: S;
int: N = S * S;
int: digs = ceil(log(10.0,int2float(N))); % digits for output

set of int: PuzzleRange = 1..N;
set of int: SubSquareRange = 1..S;

array[1..N,1..N] of 0..N: start; %% initial board 0 = empty
array[1..N,1..N] of var PuzzleRange: puzzle;

% fill initial board
constraint forall(i,j in PuzzleRange)(
    if start[i,j] > 0 then puzzle[i,j] = start[i,j] else true endif);

% All different in rows
constraint forall (i in PuzzleRange) (
    alldifferent( [ puzzle[i,j] | j in PuzzleRange ] ) );

% All different in columns.
constraint forall (j in PuzzleRange) (
    alldifferent( [ puzzle[i,j] | i in PuzzleRange ] ) );

% All different in sub-squares:
constraint
    forall (a, o in SubSquareRange)(
```

```

alldifferent( [ puzzle[(a-1)*S + a1, (o-1)*S + o1] |
    a1, o1 in SubSquareRange ] ) );

solve satisfy;

output [ show_int(digs,puzzle[i,j]) ++
    " " ++
    if j mod S == 0 then " " else "" endif ++
    if j == N then
        if i != N then
            if i mod S == 0 then "\n\n" else "\n" endif
        else "" endif else "" endif
    | i,j in PuzzleRange ] ++
    ["\n"];

```

Listing 2.2.6: Example data file for generalised Sudoku problem (sudoku.dzn)

```

S=3;
start=[|
0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 6, 8, 4, 0, 1, 0, 7, 0|
0, 0, 0, 0, 8, 5, 0, 3, 0|
0, 2, 6, 8, 0, 9, 0, 4, 0|
0, 0, 7, 0, 0, 0, 9, 0, 0|
0, 5, 0, 1, 0, 6, 3, 2, 0|
0, 4, 0, 6, 1, 0, 0, 0, 0|
0, 3, 0, 2, 0, 7, 6, 9, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0|];

```

6	8	4		1		7		
			8	5		3		
2	6	8		9		4		
	7				9			
5		1		6	3	2		
4		6	1					
3		2		7	6	9		

Fig. 2.2.1: The problem represented by data file sudoku.dzn

Conditional expressions are very useful in building complex models, or complex output. Consider the model of Sudoku problems shown in Listing 2.2.5. The initial board positions are given by the `start` parameter where 0 represents an empty board position. This is converted to constraints on the decision variables `puzzle` using the conditional expression

```
constraint forall(i,j in PuzzleRange)(  
    if start[i,j] > 0 then puzzle[i,j] = start[i,j] else true endif );
```

Conditional expressions are also very useful for defining complex output. In the Sudoku model of Listing 2.2.5 the expression

```
if j mod S == 0 then " " else "" endif
```

inserts an extra space between groups of size S . The output expression also uses conditional expressions to add blank lines after each S lines. The resulting output is highly readable.

The remaining constraints ensure that the numbers appearing in each row and column and $S \times S$ subsquare are all different.

One can use MiniZinc to search for all solutions to a satisfaction problem (`solve satisfy`) by using the flag `-a` or `--all-solutions`. Running

```
$ minizinc --solver gecode --all-solutions sudoku.mzn sudoku.dzn
```

results in

```
5 9 3 7 6 2 8 1 4  
2 6 8 4 3 1 5 7 9  
7 1 4 9 8 5 2 3 6  
  
3 2 6 8 5 9 1 4 7  
1 8 7 3 2 4 9 6 5  
4 5 9 1 7 6 3 2 8  
  
9 4 2 6 1 8 7 5 3  
8 3 5 2 4 7 6 9 1  
6 7 1 5 9 3 4 8 2  
-----  
=====
```

The line ===== is output when the system has output all possible solutions, here verifying that there is exactly one.

2.2.4 Enumerated Types

Enumerated types allows us to build models that depend on a set of objects which are part of the data, or are named in the model, and hence make models easier to understand and debug. We have introduce enumerated types or enums briefly, in this subsection we will explore how we can use them more fully, and show some of the built in functions for dealing with enumerated types.

Let's revisit the problem of coloring the graph of Australia from *Basic Modelling in MiniZinc* (page 21).

Listing 2.2.7: Model for coloring Australia using enumerated types (aust-enum.mzn).

```
enum Color;
var Color: wa;
var Color: nt;
var Color: sa;
var Color: q;
var Color: nsw;
var Color: v;
var Color: t;
constraint wa != nt /\ wa != sa /\ nt != sa /\ nt != q /\ sa != q;
constraint sa != nsw /\ sa != v /\ q != nsw /\ nsw != v;
solve satisfy;
```

The model shown in Listing 2.2.7 declares an enumerated type `Color` which must be defined in the data file. Each of the state variables is declared to take a value from this enumerated type. Running this program using

```
$ minizinc --solver gecode -D"Color = { red, yellow, blue };" aust-enum.mzn
```

might result in output

```
wa = blue;
nt = yellow;
sa = red;
q = blue;
nsw = yellow;
v = blue;
t = red;
```

Enumerated Type Variable Declarations

An enumerated type parameter is declared as either:

```
<enum-name> : <var-name>
<l>..<u> : <var-name>
```

where `<enum-name>` is the name of a enumerated type, and `<l>` and `<u>` are fixed enumerated type expressions of the same enumerated type.

An enumerated type decision variable is declared as either:

```
var <enum-name> : <var-name>
var <l>..<u> : <var-name>
```

where `<enum-name>` is the name of a enumerated type, and `<l>` and `<u>` are fixed enumerated type expressions of the same enumerated type.

A key behaviour of enumerated types is that they are automatically coerced to integers when they are used in a position expecting an integer. For example, this allows us to use global constraints

defined on integers, such as

```
global_cardinality_low_up([wa,nt,sa,q,nsw,v,t],
                           [red,yellow,blue],[2,2,2],[2,2,3]);
```

This requires at least two states to be colored each color and three to be colored blue.

Enumerated Type Operations

There are a number of built in operations on enumerated types:

- `enum_next(X,x)`: returns the next value in after `x` in the enumerated type `X`. This is a partial function, if `x` is the last value in the enumerated type `X` then the function returns \perp causing the Boolean expression containing the expression to evaluate to false.
- `enum_prev(X,x)`: returns the previous value before `x` in the enumerated type `X`. Similarly `enum_prev` is a partial function.
- `to_enum(X,i)`: maps an integer expression `i` to an enumerated type value in type `X` or evaluates to \perp if `i` is less than or equal to 0 or greater than the number of elements in `X`.

Note also that a number of standard functions are applicable to enumerated types:

- `card(X)`: returns the cardinality of an enumerated type `X`.
- `min(X)`: returns the minimum element of of an enumerated type `X`.
- `max(X)`: returns the maximum element of of an enumerated type `X`.

2.2.5 Complex 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. Imagine a scheduling problem in which we have two tasks that cannot overlap in time. If `s1` and `s2` are the corresponding start times and `d1` and `d2` are the corresponding durations we can express this as:

```
constraint s1 + d1 <= s2  \/ s2 + d2 <= s1;
```

which ensures that the tasks do not overlap.

Booleans

Boolean expressions in MiniZinc can be written using a standard mathematical syntax. The Boolean literals are `true` and `false` and the Boolean operators are conjunction, i.e. `and` (`/\`), disjunction, i.e. `or` (`\vee`), only-if (`<->`), implies (`->`), if-and-only-if (`<->`) and negation (`not`). Booleans can be automatically coerced to integers, but to make this coercion explicit the built-in function `bool2int` can be used: it returns 1 if its argument is true and 0 otherwise.

Listing 2.2.8: Model for job-shop scheduling problems (jobshop.mzn).

```

enum JOB;
enum TASK;
TASK: last = max(TASK);
array [JOB,TASK] of int: d;                                % task durations
int: total = sum(i in JOB, j in TASK)(d[i,j]);% total duration
int: digs = ceil(log(10.0,int2float(total))); % digits for output
array [JOB,TASK] of var 0..total: s;                      % start times
var 0..total: end;                                         % total end time

constraint %% ensure the tasks occur in sequence
forall(i in JOB) (
    forall(j in TASK where j < last)
        (s[i,j] + d[i,j] <= s[i,enum_next(TASK,j)]) /\ 
        s[i,last] + d[i,last] <= end
);

constraint %% ensure no overlap of tasks
forall(j in TASK) (
    forall(i,k in JOB where i < k) (
        s[i,j] + d[i,j] <= s[k,j] \/
        s[k,j] + d[k,j] <= s[i,j]
    )
);

solve minimize end;

output ["end = \n"] ++
[ show_int(digs,s[i,j]) ++ " " ++
  if j == last then "\n" else "" endif |
  i in JOB, j in TASK ];

```

Listing 2.2.9: Data for job-shop scheduling problems (jdata.dzn).

```
JOB = anon_enum(5);
TASK = anon_enum(5);
d = [| 1, 4, 5, 3, 6
      | 3, 2, 7, 1, 2
      | 4, 4, 4, 4, 4
      | 1, 1, 1, 6, 8
      | 7, 3, 2, 2, 1 |];
```

The job shop scheduling model given in Listing 2.2.8 gives a realistic example of the use of this disjunctive modelling capability. In job shop scheduling we have a set of jobs, each consisting of a sequence of tasks on separate machines: so task $[i, j]$ is the task in the i^{th} job performed on the j^{th} machine. Each sequence of tasks must be completed in order, and no two tasks on the same machine can overlap in time. Even small instances of this problem can be quite challenging to find optimal solutions.

The command

```
$ minizinc --solver gecode --all-solutions jobshop.mzn jdata.dzn
```

solves a small job shop scheduling problem, and illustrates the behaviour of `--all-solutions` for optimisation problems. Here the solver outputs each better solutions as it finds it, rather than all possible optimal solutions. The output from this command is:

```
end = 39
5 9 13 22 30
6 13 18 25 36
0 4 8 12 16
4 8 12 16 22
9 16 25 27 38
-----
end = 37
4 8 12 17 20
5 13 18 26 34
0 4 8 12 16
8 12 17 20 26
9 16 25 27 36
-----
end = 34
0 1 5 10 13
6 10 15 23 31
2 6 11 19 27
1 5 10 13 19
9 16 22 24 33
-----
end = 30
5 9 13 18 21
6 13 18 25 27
```

```

1 5 9 13 17
0 1 2 3 9
9 16 25 27 29
-----
=====
```

indicating an optimal solution with end time 30 is finally found, and proved optimal. We can generate all *optimal solutions* by adding a constraint that `end = 30` and changing the solve item to `solve satisfy` and then executing

```
$ minizinc --solver gecode --all-solutions jobshop.mzn jobshop.dzn
```

For this problem there are 3,444,375 optimal solutions.

Listing 2.2.10: Model for the stable marriage problem (`stable-marriage.mzn`).

```

int: n;

enum Men = anon_enum(n);
enum Women = anon_enum(n);

array[Women, Men] of int: rankWomen;
array[Men, Women] of int: rankMen;

array[Men] of var Women: wife;
array[Women] of var Men: husband;

% assignment
constraint forall (m in Men) (husband[wife[m]] = m);
constraint forall (w in Women) (wife[husband[w]] = w);
% ranking
constraint forall (m in Men, o in Women) (
    rankMen[m,o] < rankMen[m,wife[m]] ->
    rankWomen[o,husband[o]] < rankWomen[o,m] );

constraint forall (w in Women, o in Men) (
    rankWomen[w,o] < rankWomen[w,husband[w]] ->
    rankMen[o,wife[o]] < rankMen[o,w] );
solve satisfy;

output ["wives= \n(wife)\nhusbands= \n(husband)\n"];
```

Listing 2.2.11: Example data for the stable marriage problem (`stable-marriage.dzn`).

```

n = 5;
rankWomen =
[| 1, 2, 4, 3, 5,
 | 3, 5, 1, 2, 4,
 | 5, 4, 2, 1, 3,
```

```

| 1, 3, 5, 4, 2,
| 4, 2, 3, 5, 1 |];
rankMen =
[| 5, 1, 2, 4, 3,
| 4, 1, 3, 2, 5,
| 5, 3, 2, 4, 1,
| 1, 5, 4, 3, 2,
| 4, 3, 2, 1, 5 |];

```

Another powerful modelling feature in MiniZinc is that decision variables can be used for array access. As an example, consider the (old-fashioned) *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 so that all marriages are *stable* in the sense that:

- whenever m prefers another women o to his wife w , o prefers her husband to m , and
- whenever w prefers another man o to her husband m , o prefers his wife to w .

This can be elegantly modelled in MiniZinc. The model and sample data is shown in [Listing 2.2.10](#) and [Listing 2.2.11](#).

The first three items in the model declare the number of men/women and the set of men and women. Here we introduce the use of *anonymous enumerated types*. Both Men and Women are sets of size n , but we do not wish to mix them up so we use an anonymous enumerated type. This allows MiniZinc to detect modelling errors where we use Men for Women or vice versa.

The matrices rankWomen and rankMen, respectively, give the women's ranking of the men and the men's ranking of the women. Thus, the entry `rankWomen[w,m]` gives the ranking by woman w of man m . The lower the number in the ranking, the more the man or women is preferred.

There are two arrays of decision variables: `wife` and `husband`. These, respectively, contain the wife of each man and the husband of each women.

The first two constraints

```

constraint forall (m in Men) (husband[wife[m]] = m);
constraint forall (w in Women) (wife[husband[w]] = w);

```

ensure that the assignment of husbands and wives is consistent: w is the wife of m implies m is the husband of w and vice versa. Notice how in `husband[wife[m]]` the index expression `wife[m]` is a decision variable, not a parameter.

The next two constraints are a direct encoding of the stability condition:

```

constraint forall (m in Men, o in Women) (
    rankMen[m,o] < rankMen[m,wife[m]] ->
    rankWomen[o,husband[o]] < rankWomen[o,m] );
constraint forall (w in Women, o in Men) (
    rankWomen[w,o] < rankWomen[w,husband[w]] ->
    rankMen[o,wife[o]] < rankMen[o,w] );

```

This natural modelling of the stable marriage problem is made possible by the ability to use decision variables as array indices and to construct constraints using the standard Boolean connectives. The alert reader may be wondering at this stage, what happens if the array index variable takes a value that is outside the index set of the array. MiniZinc treats this as failure: an array access `a[e]` implicitly adds the constraint `e in index_set(a)` to the closest surrounding Boolean context where `index_set(a)` gives the index set of `a`.

Anonymous Enumerated Types

An *anonymous enumerated type* is of the form `anon_enum(<n>)` where `<n>` is a fixed integer expression defining the size of the enumerated type.

An anonymous enumerated type is just like any other enumerated type except that we have no names for its elements. When printed out, they are given unique names based on the enumerated type name.

Thus for example, consider the variable declarations

```
array[1..2] of int: a= [2,3];
var 0..2: x;
var 2..3: y;
```

The constraint `a[x] = y` will succeed with $x = 1 \wedge y = 2$ and $x = 2 \wedge y = 3$. And the constraint `not a[x] = y` will succeed with $x = 0 \wedge y = 2$, $x = 0 \wedge y = 3$, $x = 1 \wedge y = 3$ and $x = 2 \wedge y = 2$.

In the case of invalid array accesses by a parameter, the formal semantics of MiniZinc treats this as failure so as to ensure that the treatment of parameters and decision variables is consistent, but a warning is issued since it is almost always an error.

Listing 2.2.12: Model solving the magic series problem (`magic-series.mzn`).

```
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;

output [ "s = \\"(s);\n" ] ;
```

The coercion function `bool2int` can be called with any Boolean expression. This allows the MiniZinc modeller to use so called *higher order constraints*. As a simple example consider the *magic series problem*: find a list of numbers $s = [s_0, \dots, s_{n-1}]$ such that s_i is the number of occurrences of i in s . An example is $s = [1, 2, 1, 0]$.

A MiniZinc model for this problem is shown in Listing 2.2.12. The use of `bool2int` allows us to sum up the number of times the constraint `s[j]=i` is satisfied. Executing the command

```
$ minizinc --solver gecode --all-solutions magic-series.mzn -D "n=4;"
```

leads to the output

```
s = [1, 2, 1, 0];
-----
s = [2, 0, 2, 0];
-----
=====
```

indicating exactly two solutions to the problem.

Note that MiniZinc will automatically coerce Booleans to integers and integers to floats when required. We could replace the the constraint item in Listing 2.2.12 with

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

and get identical results, since the Boolean expression $s[j] = i$ will be automatically coerced to an integer, effectively by the MiniZinc system automatically adding the missing `bool2int`.

Coercion

In MiniZinc one can *coerce* a Boolean value to an integer value using the `bool2int` function. Similarly one can coerce an integer value to a float value using `int2float`. The instantiation of the coerced value is the same as the argument, e.g. `par bool` is coerced to `par int`, while `var bool` is coerced to `var int`.

MiniZinc automatically coerces Boolean expressions to integer expressions and integer expressions to float expressions, by inserting `bool2int` and `int2float` in the model appropriately. Note that it will also coerce Booleans to floats using two steps.

2.2.6 Set Constraints

Another powerful modelling feature of MiniZinc is that it allows sets containing integers to be decision variables: this means that when the model is evaluated the solver will find which elements are in the set.

As a simple example, consider the *0/1 knapsack problem*. This is a restricted form of the knapsack problem in which we can either choose to place the item in the knapsack or not. Each item has a weight and a profit and we want to find which choice of items leads to the maximum profit subject to the knapsack not being too full.

It is natural to model this in MiniZinc with a single decision variable: `var set of ITEM: knapsack` where `ITEM` is the set of possible items. If the arrays `weight[i]` and `profit[i]` respectively give the weight and profit of item `i`, and the maximum weight the knapsack can carry is given by `capacity` then a naural model is given in Listing 2.2.13.

Listing 2.2.13: Model for the 0/1 knapsack problem (`knapsack.mzn`).

```
enum ITEM;
int: capacity;

array[ITEM] of int: profits;
```

```

array[ITEM] of int: weights;

var set of ITEM: knapsack;

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

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

output ["knapsack = \$(knapsack)\n"];

```

Notice that the `var` keyword comes before the `set` declaration indicating that the set itself is the decision variable. This contrasts with an array in which the `var` keyword qualifies the elements in the array rather than the array itself since the basic structure of the array is fixed, i.e. its index set.

Listing 2.2.14: Model for the social golfers problems (`social-golfers.mzn`).

```

include "partition_set.mzn";
int: weeks;      set of int: WEEK = 1..weeks;
int: groups;    set of int: GROUP = 1..groups;
int: size;      set of int: SIZE = 1..size;
int: ngolfers = groups*size;
set of int: GOLFER = 1..ngolfers;

array[WEEK, GROUP] of var set of GOLFER: Sched;

% constraints
constraint
  forall (i in WEEK, j in GROUP) (
    card(Sched[i,j]) = size
    /\ forall (k in j+1..groups) (
      Sched[i,j] intersect Sched[i,k] = {}
    )
  ) /\
  forall (i in WEEK) (
    partition_set([Sched[i,j] | j in GROUP], GOLFER)
  ) /\
  forall (i in 1..weeks-1, j in i+1..weeks) (
    forall (x,y in GROUP) (
      card(Sched[i,x] intersect Sched[j,y]) <= 1
    )
  );
% symmetry
constraint
  % Fix the first week %
  forall (i in GROUP, j in SIZE) (
    ((i-1)*size + j) in Sched[1,i]
  ) /\
  % Fix first group of second week %
  forall (i in SIZE) (

```

```

((i-1)*size + 1) in Sched[2,1]
) /\%
% Fix first 'size' players
forall (w in 2..weeks, p in SIZE) (
    p in Sched[w,p]
);

solve satisfy;

output [ show(Sched[i,j]) ++ " " ++
        if j == groups then "\n" else "" endif |
        i in WEEK, j in GROUP ];

```

As a more complex example of set constraint consider the social golfers problem shown in Listing 2.2.14. The aim is to schedule a golf tournament over weeks using groups \times size golfers. Each week we have to schedule groups different groups each of size size. No two pairs of golfers should ever play in two groups.

The variables in the model are sets of golfers $Sched[i, j]$ for the i^{th} week and j^{th} group.

The constraints shown in lines 11-32 first enforces an ordering on the first set in each week to remove symmetry in swapping weeks. Next they enforce an ordering on the sets in each week, and make each set have a cardinality of size. They then ensure that each week is a partition of the set of golfers using the global constraint `partition_set`. Finally the last constraint ensures that no two players play in two groups together (since the cardinality of the intersection of any two groups is at most 1).

There are also symmetry breaking initialisation constraints shown in lines 34-46: the first week is fixed to have all players in order; the second week is made up of the first players of each of the first groups in the first week; finally the model forces the first size players to appear in their corresponding group number for the remaining weeks.

Executing the command

```
$ minizinc --solver gecode social-golfers.mzn social-golfers.dzn
```

where the data file defines a problem with 4 weeks, with 4 groups of size 3 leads to the output

```

1..3 4..6 7..9 10..12
{1,4,7} {2,9,12} {3,5,10} {6,8,11}
{1,6,9} {2,8,10} {3,4,11} {5,7,12}
{1,5,8} {2,7,11} {3,6,12} {4,9,10}
-----
```

Notice hows sets which are ranges may be output in range format.

2.2.7 Putting it all together

We finish this section with a complex example illustrating most of the features introduced in this chapter including enumerated types, complex constraints, global constraints, and complex

output.

Listing 2.2.15: Planning wedding seating using enumerated types (wedding.mzn).

```

enum Guests = { bride, groom, bestman, bridesmaid, bob, carol,
    ted, alice, ron, rona, ed, clara};
set of int: Seats = 1..12;
set of int: Hatreds = 1..5;
array[Hatreds] of Guests: h1 = [groom, carol, ed, bride, ted];
array[Hatreds] of Guests: h2 = [clara, bestman, ted, alice, ron];
set of Guests: Males = {groom, bestman, bob, ted, ron,ed};
set of Guests: Females = {bride,bridesmaid,carol,alice,rona,clara};

array[Guests] of var Seats: pos; % seat of guest
array[Hatreds] of var Seats: p1; % seat of guest 1 in hatred
array[Hatreds] of var Seats: p2; % seat of guest 2 in hatred
array[Hatreds] of var 0..1: sameside; % seats of hatred on same side
array[Hatreds] of var Seats: cost; % penalty of hatred

include "alldifferent.mzn";
constraint alldifferent(pos);
constraint forall(g in Males)( pos[g] mod 2 == 1 );
constraint forall(g in Females)( pos[g] mod 2 == 0 );
constraint not (pos[ed] in {1,6,7,12});
constraint abs(pos[bride] - pos[groom]) <= 1 /\ 
    (pos[bride] <= 6 <-> pos[groom] <= 6);
constraint forall(h in Hatreds)(
    p1[h] = pos[h1[h]] /\ 
    p2[h] = pos[h2[h]] /\ 
    sameside[h] = bool2int(p1[h] <= 6 <-> p2[h] <= 6) /\ 
    cost[h] = sameside[h] * abs(p1[h] - p2[h]) +
    (1 - sameside[h]) * (abs(13 - p1[h] - p2[h]) + 1) );

solve maximize sum(h in Hatreds)(cost[h]);

output [ show(g)++" " | s in Seats, g in Guests where fix(pos[g]) == s]
++ ["\n"];

```

The model of Listing 2.2.15 arranges seats at the wedding table. The table has 12 numbered seats in order around the table, 6 on each side. Males must sit in odd numbered seats, and females in even. Ed cannot sit at the end of the table because of a phobia, and the bride and groom must sit next to each other. The aim is to maximize the distance between known hatreds. The distance between seats is the difference in seat number if on the same side, otherwise its the distance to the opposite seat + 1.

Note that in the output statement we consider each seat s and search for a guest g who is assigned to that seat. We make use of the built in function `fix` which checks if a decision variable is fixed and returns its fixed value, and otherwise aborts. This is always safe to use in output statements, since by the time the output statement is run all decision variables should be fixed.

Running

```
$ minizinc --solver gecode wedding.mzn
```

Results in the output

```
ted bride groom rona ed carol ron alice bob bridesmaid bestman clara
-----
=====
```

The resulting table placement is illustrated in Fig. 2.2.2 where the lines indicate hatreds. The total distance is 22.

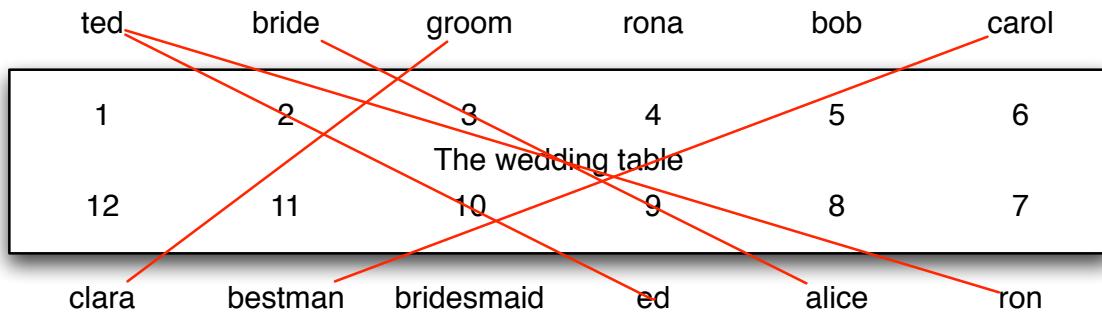


Fig. 2.2.2: Seating arrangement at the wedding table

Fix

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.

CHAPTER 2.3

Predicates and Functions

Predicates in MiniZinc allow us to capture complex constraints of our model in a succinct way. Predicates in MiniZinc are used to model with both predefined global constraints, and to capture and define new complex constraints by the modeller. Functions are used in MiniZinc to capture common structures of models. Indeed a predicate is just a function with output type `var bool`.

2.3.1 Global Constraints

There are many global constraints defined in MiniZinc for use in modelling. The definitive list is to be found in the documentation for the release, as the list is slowly growing. Below we discuss some of the most important global constraints.

2.3.1.1 Alldifferent

The `alldifferent` constraint takes an array of variables and constrains them to take different values. A use of the `alldifferent` has the form

```
alldifferent(array[int] of var int: x)
```

The argument is an array of integer variables.

The `alldifferent` constraint is one of the most studied and used global constraints in constraint programming. It is used to define assignment subproblems, and efficient global propagators for `alldifferent` exist. The models `send-more-money.mzn` ([Listing 2.2.4](#)) and `sudoku.mzn` ([Listing 2.2.5](#)) are examples of models using `alldifferent`.

2.3.1.2 Cumulative

The `cumulative` constraint is used for describing cumulative resource usage.

```
cumulative(array[int] of var int: s, array[int] of var int: d,
           array[int] of var int: r, var int: b)
```

It requires that a set of tasks given by start times s , durations d , and resource requirements r , never require more than a global resource bound b at any one time.

Listing 2.3.1: Model for moving furniture using cumulative (moving.mzn).

```
include "cumulative.mzn";

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

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

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

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

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

solve minimize end;

output [ "start = \nstart\nend = \nend\n"];
```

Listing 2.3.2: Data for moving furniture using cumulative (moving.dzn).

```
OBJECTS = { piano, fridge, doublebed, singlebed,
            wardrobe, chair1, chair2, table };

duration = [60, 45, 30, 30, 20, 15, 15, 15];
handlers = [3, 2, 2, 1, 2, 1, 1, 2];
trolleys = [2, 1, 2, 2, 2, 0, 0, 1];

available_time = 180;
available_handlers = 4;
available_trolleys = 3;
```

The model in Listing 2.3.1 finds a schedule for moving furniture so that each piece of furniture has enough handlers (people) and enough trolleys available during the move. The available time, handlers and trolleys are given, and the data gives for each object the move duration, the number of handlers and the number of trolleys required. Using the data shown in ex-movingd, the command

```
$ minizinc moving.mzn moving.dzn
```

may result in the output

```
start = [0, 60, 60, 90, 120, 0, 15, 105]
end = 140
-----
=====
```

[Fig. 2.3.1](#) and [Fig. 2.3.2](#) show the requirements for handlers and trolleys at each time in the move for this solution.

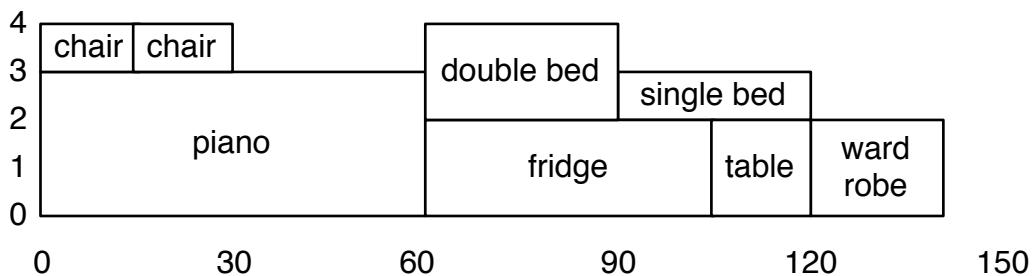


Fig. 2.3.1: Histogram of usage of handlers in the move.

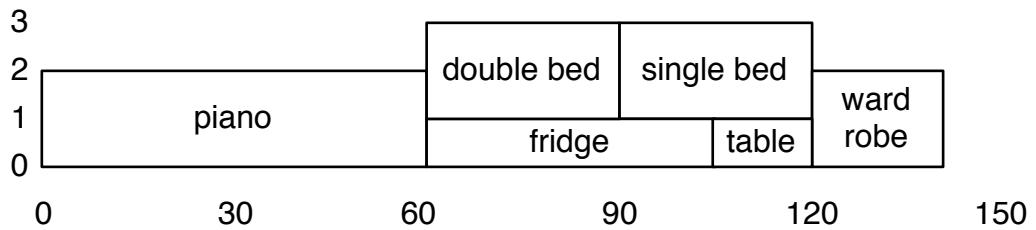


Fig. 2.3.2: Histogram of usage of trolleys in the move.

2.3.1.3 Table

The `table` constraint enforces that a tuple of variables takes a value from a set of tuples. Since there are no tuples in MiniZinc this is encoded using arrays. The usage of `table` has one of the forms

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

depending on whether the tuples are Boolean or integer. The constraint enforces $x \in t$ where we consider x and each row in t to be a tuple, and t to be a set of tuples.

Listing 2.3.3: Model for meal planning using table constraint (meal.mzn).

```
% Planning a balanced meal
include "table.mzn";
int: min_energy;
int: min_protein;
int: max_salt;
int: max_fat;
set of FOOD: desserts;
set of FOOD: mains;
set of FOOD: sides;
enum FEATURE = { name, energy, protein, salt, fat, cost};
enum FOOD;
array[FOOD,FEATURE] of int: dd; % food database

array[FEATURE] of var int: main;
array[FEATURE] of var int: side;
array[FEATURE] of var int: dessert;
var int: budget;

constraint main[name] in mains;
constraint side[name] in sides;
constraint dessert[name] in desserts;
constraint table(main, dd);
constraint table(side, dd);
constraint table(dessert, dd);
constraint main[energy] + side[energy] + dessert[energy] >=min_energy;
constraint main[protein]+side[protein]+dessert[protein] >=min_protein;
constraint main[salt] + side[salt] + dessert[salt] <= max_salt;
constraint main[fat] + side[fat] + dessert[fat] <= max_fat;
constraint budget = main[cost] + side[cost] + dessert[cost];

solve minimize budget;

output ["main = ",show(to_enum(FOOD,main[name])),",
        ", side = ",show(to_enum(FOOD,side[name])),",
        ", dessert = ",show(to_enum(FOOD,dessert[name]))],
        ", cost = ",show(budget), "\n"];
```

Listing 2.3.4: Data for meal planning defining the table used (meal.dzn).

```
FOODS = { icecream, banana, chocolatecake, lasagna,
          steak, rice, chips, brocolli, beans} ;

dd = [| icecream,      1200,   50,   10,  120,   400      % icecream
       | banana,        800,  120,    5,   20,   120      % banana
       | chocolatecake, 2500,   400,   20,  100,   600      % chocolate cake
       | lasagna,       3000,   200,   100,  250,   450      % lasagna
       | steak,         1800,   800,   50,  100,  1200      % steak
```

```

| rice,           1200, 50, 5, 20, 100    % rice
| chips,          2000, 50, 200, 200, 250   % chips
| brocolli,       700, 100, 10, 10, 125    % brocolli
| beans,          1900, 250, 60, 90, 150  []]; % beans

min_energy = 3300;
min_protein = 500;
max_salt = 180;
max_fat = 320;
desserts = { icecream, banana, chocolotecake };
mains = { lasagna, steak, rice };
sides = { chips, brocolli, beans };

```

The model in Listing 2.3.3 searches for balanced meals. Each meal item has a name (encoded as an integer), a kilojoule count, protein in grams, salt in milligrams, and fat in grams, as well as cost in cents. The relationship between these items is encoded using a `table` constraint. The model searches for a minimal cost meal which has a minimum kilojoule count `min_energy`, a minimum amount of protein `min_protein`, maximum amount of salt `max_salt` and fat `max_fat`.

2.3.1.4 Regular

The `regular` constraint is used to enforce that a sequence of variables takes a value defined by a finite automaton. The usage of `regular` has the form

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

It constrains that the sequence of values in array `x` (which must all be in the range `1..S`) is accepted by the DFA of `Q` states with input `1..S` and transition function `d` (which maps `<1..Q, 1..S>` to `0..Q`) and initial state `q0` (which must be in `1..Q`) and accepting states `F` (which all must be in `1..Q`). State 0 is reserved to be an always failing state.

Consider a nurse rostering problem. Each nurse is scheduled for each day as either: (d) on day shift, (n) on night shift, or (o) off. In each four day period a nurse must have at least one day off, and no nurse can be scheduled for 3 night shifts in a row. This can be encoded using the incomplete DFA shown in Fig. 2.3.3. We can encode this DFA as having start state 1, final states 1..6, and transition function

	d	n	o
1	2	3	1
2	4	4	1
3	4	5	1
4	6	6	1
5	6	0	1
6	0	0	1

Note that state 0 in the table indicates an error state. The model shown in Listing 2.3.5 finds a schedule for `num_nurses` nurses over `num_days` days, where we require `req_day` nurses on day

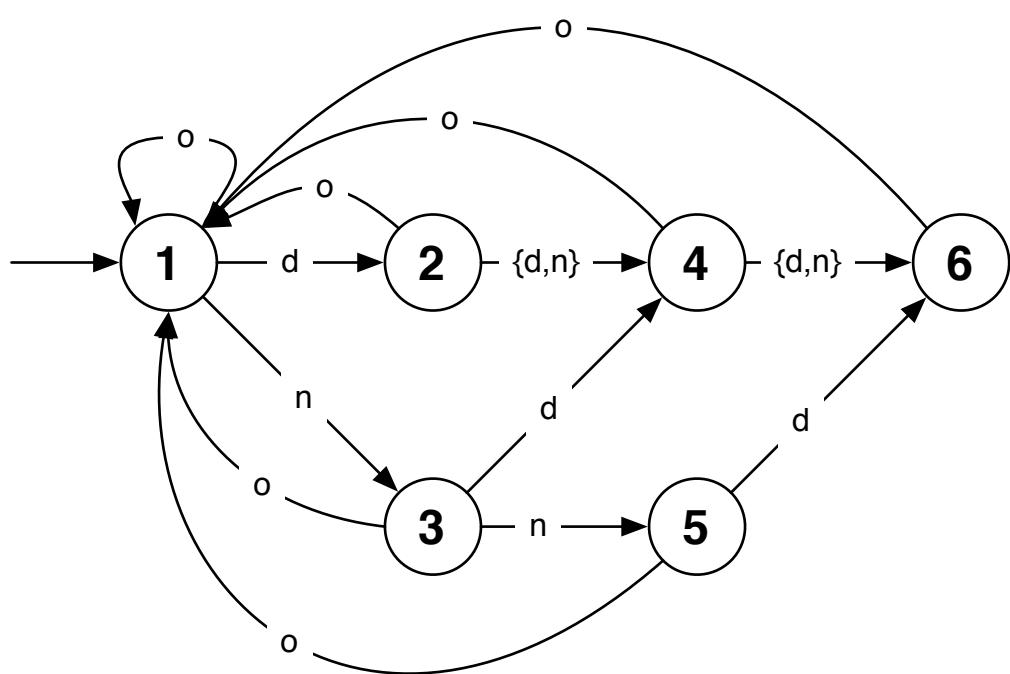


Fig. 2.3.3: A DFA determining correct rosters.

shift each day, and `req_night` nurses on night shift, and that each nurse takes at least `min_night` night shifts.

Listing 2.3.5: Model for nurse rostering using regular constraint (`nurse.mzn`)

```
% Simple nurse rostering
include "regular.mzn";
enum NURSE;
enum DAY;
int: req_day;
int: req_night;
int: min_night;

enum SHIFT = { d, n, o };
int: S = card(SHIFT);

int: Q = 6; int: q0 = 1; set of int: STATE = 1..Q;
array[STATE,SHIFT] of int: t =
  [| 2, 3, 1    % state 1
   | 4, 4, 1    % state 2
   | 4, 5, 1    % state 3
   | 6, 6, 1    % state 4
   | 6, 0, 1    % state 5
   | 0, 0, 1|]; % state 6

array[NURSE,DAY] of var SHIFT: roster;

constraint forall(j in DAY)(
    sum(i in NURSE)(roster[i,j] == d) == req_day /\ 
    sum(i in NURSE)(roster[i,j] == n) == req_night
);
constraint forall(i in NURSE)(
    regular([roster[i,j] | j in DAY], Q, S, t, q0, STATE) /\ 
    sum(j in DAY)(roster[i,j] == n) >= min_night
);

solve satisfy;

output [ show(roster[i,j]) ++ if j==card(DAY) then "\n" else " " endif
  | i in NURSE, j in DAY ];
```

Running the command

```
$ minizinc nurse.mzn nurse.dzn
```

finds a 10 day schedule for 7 nurses, requiring 3 on each day shift and 2 on each night shift, with a minimum 2 night shifts per nurse. A possible output is

```

d o n o n o d n o o
d o d n n o d d n o
o d d n o n d n o n
o d d d o n n o n n
d d n o d d n o d d
n n o d d d o d d d
n n o d d d o d d d
-----

```

There is an alternate form of the regular constraint `regular_nfa` which specifies the regular expression using an NFA (without ϵ arcs). This constraint has the form

```

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

```

It constrains that the sequence of values in array `x` (which must all be in the range `1..S`) is accepted by the NFA of `Q` states with input `1..S` and transition function `d` (which maps $<1..Q, 1..S>$ to subsets of $1..Q$) and initial state `q0` (which must be in $1..Q$) and accepting states `F` (which all must be in $1..Q$). There is no need for a failing state `0`, since the transition function can map to an empty set of states.

2.3.2 Defining Predicates

One of the most powerful modelling features of MiniZinc is the ability for the modeller to define their own high-level constraints. This allows them to abstract and modularise their model. It also allows re-use of constraints in different models and allows the development of application specific libraries defining the standard constraints and types.

Listing 2.3.6: Model for job shop scheduling using predicates (`jobshop2.mzn`)

```

int: jobs;                                     % no of jobs
set of int: JOB = 1..jobs;
int: tasks;                                     % no of tasks per job
set of int: TASK = 1..tasks;
array [JOB,TASK] of int: d;                    % task durations
int: total = sum(i in JOB, j in TASK)(d[i,j]); % total duration
int: digs = ceil(log(10.0,total));              % digits for output
array [JOB,TASK] of var 0..total: s;           % start times
var 0..total: end;                             % total end time

% nooverlap
predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
    s1 + d1 <= s2 /\ s2 + d2 <= s1;

constraint %% ensure the tasks occur in sequence
    forall(i in JOB) (
        forall(j in 1..tasks-1)
            (s[i,j] + d[i,j] <= s[i,j+1]) /\
```

```

        s[i,tasks] + d[i,tasks] <= end
    );

constraint % ensure no overlap of tasks
    forall(j in TASK) (
        forall(i,k in JOB where i < k) (
            no_overlap(s[i,j], d[i,j], s[k,j], d[k,j])
        )
    );
}

solve minimize end;

output [ "end = \n" ] ++
[ show_int(digs,s[i,j]) ++ " " ++
  if j == tasks then "\n" else "" endif |
  i in JOB, j in TASK ];

```

We start with a simple example, revisiting the job shop scheduling problem from the previous section. The model is shown in Listing 2.3.6. The item of interest is the `predicate` item:

```

predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
    s1 + d1 <= s2 \/ s2 + d2 <= s1;

```

This defines a new constraint that enforces that a task with start time s_1 and duration d_1 does not overlap with a task with start time s_2 and duration d_2 . This can now be used inside the model anywhere any other Boolean expression (involving decision variables) can be used.

As well as predicates the modeller can define new constraints that only involve parameters. These are useful to write fixed tests for a conditional expression. These are defined using the keyword `test`. For example

```

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

```

Predicate definitions

Predicates are defined by a statement of the form

```
predicate <pred-name> ( <arg-def>, ..., <arg-def> ) = <bool-exp>
```

The `<pred-name>` must be a valid MiniZinc identifier, and each `<arg-def>` is a valid MiniZinc type declaration.

One relaxation of argument definitions is that the index types for arrays can be unbounded, written `int`.

```
test <pred-name> ( <arg-def>, ..., <arg-def> ) = <bool-exp>
```

The `<bool-exp>` of the body must be fixed.

We also introduce a new form of the `assert` command for use in predicates.

```
assert ( <bool-exp>, <string-exp>, <exp> )
```

The type of the `assert` expression is the same as the type of the last argument. The `assert` expression checks whether the first argument is false, and if so prints the second argument string. If the first argument is true it returns the third argument.

Note that assert expressions are lazy in the third argument, that is if the first argument is false they will not be evaluated. Hence, they can be used for checking:

```
predicate lookup(array[int] of var int:x, int: i, var int: y) =
    assert(i in index_set(x), "index out of range in lookup"
        y = x[i]
    );
```

This code will not evaluate `x[i]` if `i` is out of the range of the array `x`.

2.3.3 Defining Functions

Functions are defined in MiniZinc similarly to predicates, but with a more general return type.

The function below defines the row in a Sudoku matrix of the $a1^{th}$ row of the a^{th} of subsquares.

```
function int: posn(int: a, int: a1) = (a-1) * S + a1;
```

With this definition we can replace the last constraint in the Sudoku problem shown in Listing 2.2.5 by

```
constraint forall(a, o in SubSquareRange)(
    alldifferent([ puzzle [ posn(a,a0), posn(o,o1) ] |
        a1,o1 in SubSquareRange ] ) );
```

Functions are useful for encoding complex expressions that are used frequently in the model. For example, imagine placing the numbers 1 to n in different positions in an $n \times n$ grid such that

the Manhattan distance between any two numbers i and j is greater than the maximum of the two numbers minus 1. The aim is to minimize the total of the Manhattan distances between the pairs. The Manhattan distance function can be expressed as:

```
function var int: manhattan(var int: x1, var int: y1,
                           var int: x2, var int: y2) =
    abs(x1 - x2) + abs(y1 - y2);
```

The complete model is shown in Listing 2.3.7.

Listing 2.3.7: Model for a number placement problem illustrating the use of functions
(manhattan.mzn).

```
int: n;
set of int: NUM = 1..n;

array[NUM] of var NUM: x;
array[NUM] of var NUM: y;
array[NUM,NUM] of var 0..2*n-2: dist =
    array2d(NUM,NUM, [
        if i < j then manhattan(x[i],y[i],x[j],y[j]) else 0 endif
        | i,j in NUM ]);

% manf
function var int: manhattan(var int: x1, var int: y1,
                           var int: x2, var int: y2) =
    abs(x1 - x2) + abs(y1 - y2);

constraint forall(i,j in NUM where i < j)
    (dist[i,j] >= max(i,j)-1);

var int: obj = sum(i,j in NUM where i < j)(dist[i,j]);
solve minimize obj;

% simply to display result
include "alldifferent_except_0.mzn";
array[NUM,NUM] of var 0..n: grid;
constraint forall(i in NUM)(grid[x[i],y[i]] = i);
constraint alldifferent_except_0([grid[i,j] | i,j in NUM]);

output ["obj = \obj;\n"] ++
    [ if fix(grid[i,j]) > 0 then show(grid[i,j]) else "." endif
    ++ if j = n then "\n" else "" endif
    | i,j in NUM];
```

Function definitions

Functions are defined by a statement of the form

```
function <ret-type> : <func-name> ( <arg-def>, ..., <arg-def> ) = <exp>
```

The `<func-name>` must be a valid MiniZinc identifier, and each `<arg-def>` is a valid MiniZinc type declaration. The `<ret-type>` is the return type of the function which must be the type of `<exp>`. Arguments have the same restrictions as in predicate definitions.

Functions in MiniZinc can have any return type, not just fixed return types. Functions are useful for defining and documenting complex expressions that are used multiple times in a model.

2.3.4 Reflection Functions

To help write generic tests and predicates, various reflection functions return information about array index sets, var set domains and decision variable ranges. Those for index sets are `index_set(<1-D array>)`, `index_set_1of2(<2-D array>)`, `index_set_2of2(<2-D array>)`, and so on for higher dimensional arrays.

A better model of the job shop conjoins all the non-overlap constraints for a single machine into a single disjunctive constraint. An advantage of this approach is that while we may initially model this simply as a conjunction of non-overlap constraints, if the underlying solver has a better approach to solving disjunctive constraints we can use that instead, with minimal changes to our model. The model is shown in Listing 2.3.8.

Listing 2.3.8: Model for job shop scheduling using disjunctive predicate
(`jobshop3.mzn`).

```
include "disjunctive.mzn";

int: jobs;                                     % no of jobs
set of int: JOB = 1..jobs;
int: tasks;                                     % no of tasks per job
set of int: TASK = 1..tasks;
array [JOB,TASK] of int: d;                    % task durations
int: total = sum(i in JOB, j in TASK)(d[i,j]); % total duration
int: digs = ceil(log(10.0,total));              % digits for output
array [JOB,TASK] of var 0..total: s;           % start times
var 0..total: end;                             % total end time

constraint %% ensure the tasks occur in sequence
  forall(i in JOB) (
    forall(j in 1..tasks-1)
      (s[i,j] + d[i,j] <= s[i,j+1]) /\ 
      s[i,tasks] + d[i,tasks] <= end
  );

constraint %% ensure no overlap of tasks
  forall(j in TASK) (
```

```

    disjunctive([s[i,j] | i in JOB], [d[i,j] | i in JOB])
);

solve minimize end;

output ["end = \n(end)\n"] ++
[ show_int(digs,s[i,j]) ++ " " ++
  if j == tasks then "\n" else "" endif |
  i in JOB, j in TASK ];

```

The `disjunctive` constraint takes an array of start times for each task and an array of their durations and makes sure that only one task is active at any one time. We define the disjunctive constraint as a predicate with signature

```
predicate disjunctive(array[int] of var int:s, array[int] of int:d);
```

We can use the disjunctive constraint to define the non-overlap of tasks as shown in Listing 2.3.8. We assume a definition for the `disjunctive` predicate is given by the file `disjunctive.mzn` which is included in the model. If the underlying system supports `disjunctive` directly, it will include a file `disjunctive.mzn` in its `globals` directory (with contents just the signature definition above). If the system we are using does not support `disjunctive` directly we can give our own definition by creating the file `disjunctive.mzn`. The simplest implementation simply makes use of the `no_overlap` predicate defined above. A better implementation is to make use of a global `cumulative` constraint assuming it is supported by the underlying solver. Listing 2.3.9 shows an implementation of `disjunctive`. Note how we use the `index_set` reflection function to (a) check that the arguments to `disjunctive` make sense, and (b) construct the array of resource utilisations of the appropriate size for `cumulative`. Note also that we use a ternary version of `assert` here.

Listing 2.3.9: Defining a disjunctive predicate using cumulative (`disjunctive.mzn`).

```

include "cumulative.mzn";

predicate disjunctive(array[int] of var int:s, array[int] of int:d) =
  assert(index_set(s) == index_set(d), "disjunctive: " ++
    "first and second arguments must have the same index set",
    cumulative(s, d, [ 1 | i in index_set(s) ], 1)
);

```

2.3.5 Local Variables

It is often useful to introduce *local variables* in a predicate, function or test. The `let` expression allows you to do so. It can be used to introduce both decision variables and parameters, but parameters must be initialised. For example:

```

var s..e: x;
let {int: l = s div 2; int: u = e div 2; var l .. u: y;} in x = 2*y

```

introduces parameters `l` and `u` and variable `y`. While most useful in predicate, function and test definitions, `let` expressions can also be used in other expressions, for example for eliminating common subexpressions:

```
constraint let { var int: s = x1 + x2 + x3 + x4 } in
    l <= s /\ s <= u;
```

Local variables can be used anywhere and can be quite useful for simplifying complex expressions. Listing 2.3.10 gives a revised version of the wedding model, using local variables to define the objective function, rather than adding lots of variables to the model explicitly.

Listing 2.3.10: Using local variables to define a complex objective function (wedding2.mzn).

```
enum Guests = { bride, groom, bestman, bridesmaid, bob, carol,
    ted, alice, ron, rona, ed, clara};
set of int: Seats = 1..12;
set of int: Hatreds = 1..5;
array[Hatreds] of Guests: h1 = [groom, carol, ed, bride, ted];
array[Hatreds] of Guests: h2 = [clara, bestman, ted, alice, ron];
set of Guests: Males = {groom, bestman, bob, ted, ron,ed};
set of Guests: Females = {bride,bridesmaid,carol,alice,rona,clara};

array[Guests] of var Seats: pos; % seat of guest

include "alldifferent.mzn";
constraint alldifferent(pos);

constraint forall(g in Males)( pos[g] mod 2 == 1 );
constraint forall(g in Females)( pos[g] mod 2 == 0 );

constraint not (pos[ed] in {1,6,7,12});
constraint abs(pos;bride] - pos[groom]) <= 1 /\ 
    (pos[bride] <= 6 <-> pos[groom] <= 6);

solve maximize sum(h in Hatreds)(
    let { var Seats: p1 = pos[h1[h]];
        var Seats: p2 = pos[h2[h]];
        var 0..1: same = bool2int(p1 <= 6 <-> p2 <= 6); } in
    same * abs(p1 - p2) + (1-same) * (abs(13 - p1 - p2) + 1));

output [ show(g)++" " | s in Seats, g in Guests where fix(pos[g]) == s]
++ ["\n"];
```

2.3.6 Context

One limitation is that predicates and functions containing decision variables that are not initialised in the declaration cannot be used inside a negative context. The following is illegal:

```

predicate even(var int:x) =
    let { var int: y } in x = 2 * y;

constraint not even(z);

```

The reason for this is that solvers only solve existentially constrained problems, and if we introduce a local variable in a negative context, then the variable is *universally quantified* and hence out of scope of the underlying solvers. For example the $\neg \text{even}(z)$ is equivalent to $\neg \exists y. z = 2y$ which is equivalent to $\forall y. z \neq 2y$.

If local variables are given values, then they can be used in negative contexts. The following is legal

```

predicate even(var int:x) =
    let { var int: y = x div 2; } in x = 2 * y;

constraint not even(z);

```

Note that the meaning of `even` is correct, since if x is even then $x = 2 * (x \text{ div } 2)$. Note that for this definition $\neg \text{even}(z)$ is equivalent to $\neg \exists y. y = z \text{ div } 2 \wedge z = 2y$ which is equivalent to $\exists y. y = z \text{ div } 2 \wedge \neg z \neq 2y$, because y is functionally defined by z .

Every expression in MiniZinc appears in one of the four *contexts*: root, positive, negative, or mixed. The context of a non-Boolean expression is simply the context of its nearest enclosing Boolean expression. The one exception is that the objective expression appears in a root context (since it has no enclosing Boolean expression).

For the purposes of defining contexts we assume implication expressions $e1 \rightarrow e2$ are rewritten equivalently as `not e1 \vee e2`, and similarly $e1 <- e2$ is rewritten as `e1 \vee not e2`.

The context for a Boolean expression is given by:

root root context is the context for any expression `e` appearing as the argument of `constraint` or as an assignment item, or appearing as a sub expression `e1` or `e2` in an expression `e1 /\ e2` occurring in a root context.

Root context Boolean expressions must hold in any model of the problem.

positive positive context is the context for any expression appearing as a sub expression `e1` or `e2` in an expression `e1 \vee e2` occurring in a root or positive context, appearing as a sub expression `e1` or `e2` in an expression `e1 /\ e2` occurring in a positive context, or appearing as a sub expression `e` in an expression `not e` appearing in a negative context.

Positive context Boolean expressions need not hold in a model, but making them hold will only increase the possibility that the enclosing constraint holds. A positive context expression has an even number of negations in the path from the enclosing root context to the expression.

negative negative context is the context for any expression appearing as a sub expression `e1` or `e2` in an expression `e1 \vee e2` or `e1 /\ e2` occurring in a negative context, or appearing as a sub expression `e` in an expression `not e` appearing in a positive context.

Negative context Boolean expressions need not hold in a model, but making them false will increase the possibility that the enclosing constraint holds. A negative context expres-

sion has an odd number of negations in the path from the enclosing root context to the expression.

mixed mixed context is the context for any Boolean expression appearing as a subexpression `e1 or e2 in e1 <-> e2, e1 = e2, or bool2int(e)`.

Mixed context expression are effectively both positive and negative. This can be seen from the fact that `e1 <-> e2` is equivalent to `(e1 /\ e2) \vee (not e1 /\ not e2)` and `x = bool2int(e)` is equivalent to `(e /\ x=1) \vee (not e /\ x=0)`.

Consider the code fragment

```
constraint x > 0 /\ (i <= 4 -> x + bool2int(x > i) = 5);
```

then `x > 0` is in the root context, `i <= 4` is in a negative context, `x + bool2int(x > i) = 5` is in a positive context, and `x > i` is in a mixed context.

2.3.7 Local Constraints

Let expressions can also be used to include local constraints, usually to constrain the behaviour of local variables. For example, consider defining a square root function making use of only multiplication:

```
function var float: mysqrt(var float:x) =
    let { var float: y;
          constraint y >= 0;
          constraint x = y * y; } in y;
```

The local constraints ensure `y` takes the correct value; which is then returned by the function.

Local constraints can be used in any let expression, though the most common usage is in defining functions.

Let expressions

Local variables can be introduced in any expression with a *let expression* of the form:

```
let { <dec>; ... <dec> ; } in <exp>
```

The declarations `<dec>` can be declarations of decision variables and parameters (which must be initialised) or constraint items. No declaration can make use of a newly declared variable before it is introduced.

Note that local variables and constraints cannot occur in tests. Local variables cannot occur in predicates or functions that appear in a negative or mixed context, unless the variable is defined by an expression.

2.3.8 Domain Reflection Functions

Other important reflection functions are those that allow us to access the domains of variables. The expression `lb(x)` returns a value that is lower than or equal to any value that `x` may take in a solution of the problem. Usually it will just be the declared lower bound of `x`. If `x` is declared as a non-finite type, e.g. simply `var int` then it is an error. Similarly the expression `dom(x)` returns a (non-strict) superset of the possible values of `x` in any solution of the problem. Again it is usually the declared values, and again if it is not declared as finite then there is an error.

Listing 2.3.11: Using reflection predicates (`reflection.mzn`).

```
var -10..10: x;
constraint x in 0..4;
int: y = lb(x);
set of int: D = dom(x);
solve satisfy;
output ["y = ", show(y), "\nD = ", show(D), "\n"];
```

For example, the model show in Listing 2.3.11 may output

```
y = -10
D = -10..10
-----
```

or

```
y = 0
D = {0, 1, 2, 3, 4}
-----
```

or any answer with $-10 \leq y \leq 0$ and $\{0, \dots, 4\} \subseteq D \subseteq \{-10, \dots, 10\}$.

Variable domain reflection expressions should be used in a manner where they are correct for any safe approximations, but note this is not checked! For example the additional code

```
var -10..10: z;
constraint z <= y;
```

is not a safe usage of the domain information. Since using the tighter (correct) approximation leads to more solutions than the weaker initial approximation.

Domain reflection

There are reflection functions to interrogate the possible values of expressions containing variables:

- `dom(<exp>)` returns a safe approximation to the possible values of the expression.
- `lb(<exp>)` returns a safe approximation to the lower bound value of the expression.
- `ub(<exp>)` returns a safe approximation to the upper bound value of the expression.

The expressions for `lb` and `ub` can only be of types `int`, `bool`, `float` or `set of int`. For `dom` the type cannot be `float`. If one of the variables appearing in `<exp>` has a non-finite declared type (e.g. `var int` or `var float`) then an error can occur.

There are also versions that work directly on arrays of expressions (with similar restrictions):

- `dom_array(<array-exp>)`: returns a safe approximation to the union of all possible values of the expressions appearing in the array.
- `lb_array(<array-exp>)` returns a safe approximation to the lower bound of all expressions appearing in the array.
- `ub_array(<array-exp>)` returns a safe approximation to the upper bound of all expressions appearing in the array.

The combinations of predicates, local variables and domain reflection allows the definition of complex global constraints by decomposition. We can define the time based decomposition of the `cumulative` constraint using the code shown in [Listing 2.3.12](#).

Listing 2.3.12: Defining a cumulative predicate by decomposition (`cumulative.mzn`).

```
%-----%
% Requires that a set of tasks given by start times 's',
% durations 'd', and resource requirements 'r',
% never require more than a global
% resource bound 'b' at any one time.
% Assumptions:
% - forall i, d[i] >= 0 and r[i] >= 0
%-----%
predicate cumulative(array[int] of var int: s,
                     array[int] of var int: d,
                     array[int] of var int: r, var int: b) =
  assert(index_set(s) == index_set(d) /\ 
         index_set(s) == index_set(r),
        "cumulative: the array arguments must have identical index sets",
  assert(lb_array(d) >= 0 /\ lb_array(r) >= 0,
        "cumulative: durations and resource usages must be non-negative",
  let {
    set of int: times =
      lb_array(s) ..
      max([ ub(s[i]) + ub(d[i]) | i in index_set(s) ])
  }
  in
    forall( t in times ) (
      b >= sum( i in index_set(s) ) (
        bool2int( s[i] <= t /\ t < s[i] + d[i] ) * r[i]
      )
    )

```

```

    )
);

```

The decomposition uses `lb` and `ub` to determine the set of times `times` over which tasks could range. It then asserts for each time `t` in `times` that the sum of resources for the active tasks at time `t` is less than the bound `b`.

2.3.9 Scope

It is worth briefly mentioning the scope of declarations in MiniZinc. MiniZinc has a single namespace, so all variables appearing in declarations are visible in every expression in the model. MiniZinc introduces locally scoped variables in a number of ways:

- as iterator variables in comprehension expressions
- using `let` expressions
- as predicate and function arguments

Any local scoped variable overshadows the outer scoped variables of the same name.

Listing 2.3.13: A model for illustrating scopes of variables (`scope.mzn`).

```

int: x = 3;
int: y = 4;
predicate smallx(var int:y) = -x <= y /\ y <= x;
predicate p(int: u, var bool: y) =
  exists(x in 1..u)(y /\ smallx(x));
constraint p(x,false);
solve satisfy;

```

For example, in the model shown in Listing 2.3.13 the `x` in `-x <= y` is the global `x`, the `x` in `smallx(x)` is the iterator `x in 1..u`, while the `y` in the disjunction is the second argument of the predicate.

CHAPTER 2.4

Option Types

Option types are a powerful abstraction that allows for concise modelling. An option type decision variable represents a decision that has another possibility \top , represented in MiniZinc as <> indicating the variable is *absent*. Option type decisions are useful for modelling problems where a decision is not meaningful unless other decisions are made first.

2.4.1 Declaring and Using Option Types

Option type Variables

An option type variable is declared as:

```
var opt <type> : <var-name>;
```

where `<type>` is one of `int`, `float` or `bool` or a fixed range expression. Option type variables can be parameters but this is rarely useful.

An option type variable can take the additional value <> indicating *absent*.

Three builtin functions are provided for option type variables: `absent(v)` returns true iff option type variable `v` takes the value <> , `occurs(v)` returns true iff option type variable `v` does *not* take the value <> , and `deopt(v)` returns the normal value of `v` or fails if it takes the value <> .

The most common use of option types is for optional tasks in scheduling. In the flexible job shop scheduling problem we have n tasks to perform on k machines, and the time to complete each task on each machine may be different. The aim is to minimize the completion time of all tasks. A model using option types to encode the problem is given in Listing 2.4.1. We model the problem using $n \times k$ optional tasks representing the possibility of each task run on each machine. We require that start time of the task and its duration spans the optional tasks that make it up, and require only one actually runs using the `alternative` global constraint. We require that at most one task runs on any machine using the `disjunctive` global constraint extended to optional

tasks. Finally we constrain that at most k tasks run at any time, a redundant constraint that holds on the actual (not optional) tasks.

Listing 2.4.1: Model for flexible job shop scheduling using option types
(flexible-js.mzn).

```

int: horizon;                                % time horizon
set of int: Time = 0..horizon;
enum Task;
enum Machine;

array[Task,Machine] of int: d; % duration on each machine
int: maxd = max([ d[t,m] | t in Task, m in Machine ]);
int: mind = min([ d[t,m] | t in Task, m in Machine ]);

array[Task] of var Time: S;                  % start time
array[Task] of var mind..maxd: D;           % duration
array[Task,Machine] of var opt Time: O; % optional task start

constraint forall(t in Task)(alternative(S[t],D[t],
                                         [0[t,m]|m in Machine],[d[t,m]|m in Machine]));
constraint forall(m in Machine)
    (disjunctive([0[t,m]|t in Task],[d[t,m]|t in Task]));
constraint cumulative(S,D,[1|i in Task],k);

solve minimize max(t in Task)(S[t] + D[t]);

```

2.4.2 Hidden Option Types

Option type variable arise implicitly when list comprehensions are constructed with iteration over variable sets, or where the expressions in `where` clauses are not fixed.

For example the model fragment

```

var set of 1..n: x;
constraint sum(i in x)(i) <= limit;

```

is syntactic sugar for

```

var set of 1..n: x;
constraint sum(i in 1..n)(if i in x then i else 0) <= limit;

```

The `sum` builtin function actually operates on a list of type-inst `var opt int`. Since the `<0` acts as the identity 0 for `+` this gives the expected results.

Similarly the model fragment

```
array[1..n] of var int: x;
constraint forall(i in 1..n where x[i] >= 0)(x[i] <= limit);
```

is syntactic sugar for

```
array[1..n] of var int: x;
constraint forall(i in 1..n)(if x[i] >= 0 then x[i] <= limit else <> endif);
```

Again the `forall` function actually operates on a list of type-inst `var opt bool`. Since `<>` acts as identity true for \wedge this gives the expected results.

The hidden uses can lead to unexpected behaviour though so care is warranted. Consider

```
var set of 1..9: x;
constraint card(x) <= 4;
constraint length([ i | i in x]) > 5;
solve satisfy;
```

which would appear to be unsatisfiable. It returns $x = \{1, 2, 3, 4\}$ as example answer. This is correct since the second constraint is equivalent to

```
constraint length([ if i in x then i else <> endif | i in 1..9 ]) > 5;
```

and the length of the list of optional integers is always 9 so the constraint always holds!

One can avoid hidden option types by not constructing iteration over variables sets or using unfixed `where` clauses. For example the above two examples could be rewritten without option types as

```
var set of 1..n: x;
constraint sum(i in 1..n)(bool2int(i in x)*i) <= limit;
```

and

```
array[1..n] of var int: x;
constraint forall(i in 1..n)(x[i] >= 0 -> x[i] <= limit);
```


CHAPTER 2.5

Search

By default in MiniZinc there is no declaration of how we want to search for solutions. This leaves the search completely up to the underlying solver. But sometimes, particularly for combinatorial integer problems, we may want to specify how the search should be undertaken. This requires us to communicate to the solver a search strategy. Note that the search strategy is *not* really part of the model. Indeed it is not required that each solver implements all possible search strategies. MiniZinc uses a consistent approach to communicating extra information to the constraint solver using *annotations*.

2.5.1 Finite Domain Search

Search in a finite domain solver involves examining the remaining possible values of variables and choosing to constrain some variables further. The search then adds a new constraint that restricts the remaining values of the variable (in effect guessing where the solution might lie), and then applies propagation to determine what other values are still possible in solutions. In order to guarantee completeness, the search leaves another choice which is the negation of the new constraint. The search ends either when the finite domain solver detects that all constraints are satisfied, and hence a solution has been found, or that the constraints are unsatisfiable. When unsatisfiability is detected the search must proceed down a different set of choices. Typically finite domain solvers use depth first search where they undo the last choice made and then try to make a new choice.

Listing 2.5.1: Model for n-queens (nqueens.mzn).

```
int: n;
array [1..n] of var 1..n: q; % queen i is in row q[i]

include "alldifferent.mzn";

constraint alldifferent(q); % distinct rows
constraint alldifferent([ q[i] + i | i in 1..n]); % distinct diagonals
```

```

constraint alldifferent([ q[i] - i | i in 1..n]); % upwards+downwards

% search
solve :: int_search(q, first_fail, indomain_min, complete)
    satisfy;
output [ if fix(q[j]) == i then "Q" else "." endif ++
        if j == n then "\n" else "" endif | i,j in 1..n]

```

A simple example of a finite domain problem is the n queens problem which requires that we place n queens on an $n \times n$ chessboard so that none can attack another. The variable $q[i]$ records in which row the queen in column i is placed. The `alldifferent` constraints ensure that no two queens are on the same row, or diagonal. A typical (partial) search tree for $n = 9$ is illustrated in Fig. 2.5.1. We first set $q[1] = 1$, this removes values from the domains of other variables, so that e.g. $q[2]$ cannot take the values 1 or 2. We then set $q[2] = 3$, this further removes values from the domains of other variables. We set $q[3] = 5$ (its earliest possible value). The state of the chess board after these three decisions is shown in Fig. 2.5.2 where the queens indicate the position of the queens fixed already and the stars indicate positions where we cannot place a queen since it would be able to take an already placed queen.

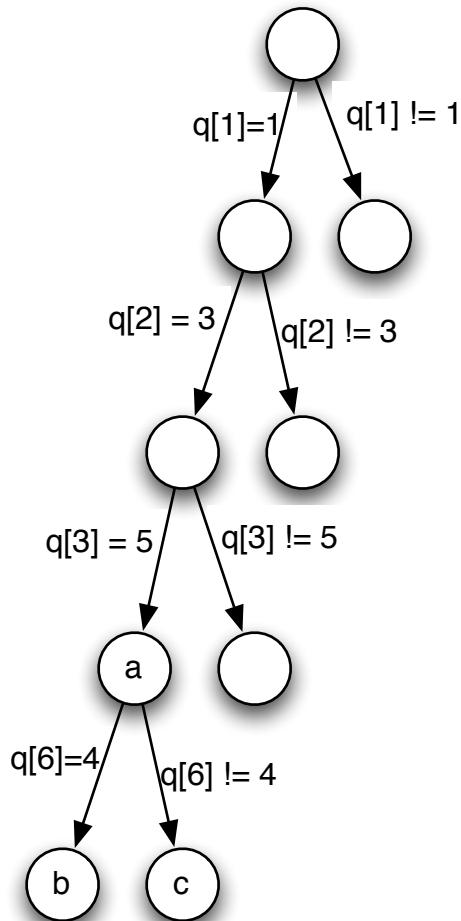


Fig. 2.5.1: Partial search trees for 9 queens

A search strategy determines which choices to make. The decisions we have made so far follow

	$q[1]$	$q[2]$	$q[3]$	$q[4]$	$q[5]$	$q[6]$	$q[7]$	$q[8]$	$q[9]$
1									
2									
3									
4									
5									
6									
7									
8									
9									

Fig. 2.5.2: The state after the addition of $q[1] = 1$, $q[2] = 4$, $q[3] = 5$

	q[1]	q[2]	q[3]	q[4]	q[5]	q[6]	q[7]	q[8]	q[9]
1									
2									
3									
4									
5									
6									
7									
8									
9									

Fig. 2.5.3: The initial propagation on adding further $q[6] = 4$

the simple strategy of picking the first variable which is not fixed yet, and try to set it to its least possible value. Following this strategy the next decision would be $q[4] = 7$. An alternate strategy for variable selection is to choose the variable whose current set of possible values (*domain*) is smallest. Under this so called *first-fail* variable selection strategy the next decision would be $q[6] = 4$. If we make this decision, then initially propagation removes the additional values shown in Fig. 2.5.3. But this leaves only one value for $q[8]$, $q[8] = 7$, so this is forced, but then this leaves only one possible value for $q[7]$ and $q[9]$, that is 2. Hence a constraint must be violated. We have detected unsatisfiability, and the solver must backtrack undoing the last decision $q[6] = 4$ and adding its negation $q[6] \neq 4$ (leading us to state (c) in the tree in Fig. 2.5.1) which forces $q[6] = 8$. This removes some values from the domain and then we again reinvoke the search strategy to decide what to do.

Many finite domain searches are defined in this way: choose a variable to constrain further, and then choose how to constrain it further.

2.5.2 Search Annotations

Search annotations in MiniZinc specify how to search in order to find a solution to the problem. The annotation is attached to the solve item, after the keyword `solve`. The search annotation

```
solve :: int_search(q, first_fail, indomain_min, complete)
        satisfy;
```

appears on the solve item. Annotations are attached to parts of the model using the connector `:::`. This search annotation means that we should search by selecting from the array of integer variables `q`, the variable with the smallest current domain (this is the `first_fail` rule), and try setting it to its smallest possible value (`indomain_min` value selection), looking across the entire search tree (`complete` search).

Basic search annotations

There are three basic search annotations corresponding to different basic variable types:

- `int_search(<variables>, <varchoice>, <constrainchoice>, <strategy>)`
where `<variables>` is a one dimensional array of `var int`, `<varchoice>` is a variable choice annotation discussed below, `<constrainchoice>` is a choice of how to constrain a variable, discussed below, and `<strategy>` is a search strategy which we will assume for now is complete.
- `bool_search(<variables>, <varchoice>, <constrainchoice>, <strategy>)`
where `<variables>` is a one dimensional array of `var bool` and the rest are as above.
- `set_search(<variables>, <varchoice>, <constrainchoice>, <strategy>)`
where `<variables>` is a one dimensional array of `var set of int` and the rest are as above.
- `float_search(<variables>, <precision>, <varchoice>, <constrainchoice>, <strategy>)`
where `<variables>` is a one dimensional array of `var float`, `<precision>` is a fixed float specifying the ϵ below which two float values are considered equal, and the rest are as above.

Example variable choice annotations are:

- `input_order`: choose in order from the array
- `first_fail`: choose the variable with the smallest domain size, and
- `smallest`: choose the variable with the smallest value in its domain.
- `dom_w_deg`: choose the variable with the smallest value of domain size divided by weighted degree, which is the number of times it has been in a constraint that caused failure earlier in the search.

Example ways to constrain a variable are:

- `indomain_min`: assign the variable its smallest domain value,
- `indomain_median`: assign the variable its median domain value,
- `indomain_random`: assign the variable a random value from its domain, and
- `indomain_split` bisect the variables domain excluding the upper half.

The `<strategy>` is almost always `complete` for complete search. For a complete list of variable and constraint choice annotations see the FlatZinc specification in the MiniZinc reference documentation.

We can construct more complex search strategies using search constructor annotations. There is only one such annotation at present:

```
seq_search([ <search-ann>, ..., <search-ann> ])
```

The sequential search constructor first undertakes the search given by the first annotation in its list, when all variables in this annotation are fixed it undertakes the second search annotation, etc. until all search annotations are complete.

Consider the jobshop scheduling model shown in Listing 2.3.8. We could replace the solve item with

```
solve :: seq_search([
    int_search(s, smallest, indomain_min, complete),
    int_search([end], input_order, indomain_min, complete)])
minimize end
```

which tries to set start times `s` by choosing the job that can start earliest and setting it to that time. When all start times are complete the end time `end` may not be fixed. Hence we set it to its minimal possible value.

2.5.3 Annotations

Annotations are a first class object in MiniZinc. We can declare new annotations in a model, and declare and assign to annotation variables.

Annotations

Annotations have a type `ann`. You can declare an annotation parameter (with optional assignment):

```
ann : <ident>;
ann : <ident> = <ann-expr> ;
```

and assign to an annotation variable just as any other parameter.

Expressions, variable declarations, and `solve` items can all be annotated using the `:::` operator.

We can declare a new annotation using the `annotation` item:

```
annotation <annotation-name> ( <arg-def>, ..., <arg-def> ) ;
```

Listing 2.5.2: Annotated model for n-queens (`nqueens-ann.mzn`).

```
annotation bitdomain(int:nwords);

include "alldifferent.mzn";

int: n;
array [1..n] of var 1..n: q :: bitdomain(n div 32 + 1);

constraint alldifferent(q) :: domain;
constraint alldifferent([ q[i] + i | i in 1..n]) :: domain;
constraint alldifferent([ q[i] - i | i in 1..n]) :: domain;

ann: search_ann;

solve :: search_ann satisfy;

output [ if fix(q[j]) == i then "Q" else "." endif ++
        if j == n then "\n" else "" endif | i,j in 1..n]
```

The program in Listing 2.5.2 illustrates the use of annotation declarations, annotations and annotation variables. We declare a new annotation `bitdomain` which is meant to tell the solver that variables domains should be represented via bit arrays of size `nwords`. The annotation is attached to the declarations of the variables `q`. Each of the `alldifferent` constraints is annotated with the built in annotation `domain` which instructs the solver to use the domain propagating

version of `alldifferent` if it has one. An annotation variable `search_ann` is declared and used to define the search strategy. We can give the value to the search strategy in a separate data file.

Example search annotations might be the following (where we imagine each line is in a separate data file)

```
search_ann = int_search(q, input_order, indomain_min, complete);
search_ann = int_search(q, input_order, indomain_median, complete);
search_ann = int_search(q, first_fail, indomain_min, complete);
search_ann = int_search(q, first_fail, indomain_median, complete);
search_ann = int_search(q, input_order, indomain_random, complete);
```

The first just tries the queens in order setting them to the minimum value, the second tries the queens variables in order, but sets them to their median value, the third tries the queen variable with smallest domain and sets it to the minimum value, and the final strategy tries the queens variable with smallest domain setting it to its median value.

Different search strategies can make a significant difference in how easy it is to find solutions. A small comparison of the number of failures made to find the first solution of the n-queens problems using the 5 different search strategies is shown in the table below (where — means more than 100,000 failures). Clearly the right search strategy can make a significant difference, and variables selection is more important than value selection, except that for this problem random value selection is very powerful.

n	input-min	input-median	ff-min	ff-median	input-random
10	22	2	5	0	6
15	191	4	4	12	39
20	20511	32	27	16	2
25	2212	345	51	25	2
30	—	137	22	66	9
35	—	1722	52	12	12
40	—	—	16	44	2
45	—	—	41	18	6

2.5.4 Restart

Any kind of depth first search for solving optimization problems suffers from the problem that wrong decisions made at the top of the search tree can take an exponential amount of search to undo. One common way to ameliorate this problem is to restart the search from the top thus having a chance to make different decisions.

MiniZinc includes annotations to control restart behaviour. These annotations, like other search annotations, are attached to the solve item of the model.

Restart search annotations

The different restart annotations control how frequently a restart occurs. Restarts occur when a limit in nodes is reached, where search returns to the top of the search tree and begins again. The possibilities are

- `restart_constant(<scale>)` where `<scale>` is an integer defining after how many nodes to restart.
- `restart_linear(<scale>)` where `<scale>` is an integer defining the initial number of nodes before the first restart. The second restart gets twice as many nodes, the third gets three times, etc.
- `restart_geometric(<base>, <scale>)` where `<base>` is a float and `<scale>` is an integer. The k th restart has a node limit of $<scale> * <base>^k$.
- `restart_luby(<scale>)` where `:mzndef: '<scale>'` is an integer. The k th restart gets $<scale> * L[k]$ where `:mzn'L[k]` is the k th number in the Luby sequence. The Luby sequence looks like 1 1 2 1 1 2 4 1 1 2 1 1 2 4 8 ..., that is it repeats two copies of the sequence ending in 2^{i-1} before adding the number 2^{i+1} .
- `restart_none` dont apply any restart (useful for setting a `ann` parameter that controls restart).

Solvers behaviour where two or more restart annotations are used is undefined.

Restart search is much more robust in finding solutions, since it can avoid getting stuck in a non-productive area of the search. Note that restart search does not make much sense if the underlying search strategy does not do something different the next time it starts at the top. For example the search annotation

```
solve :: int_search(q, input_order, indomain_min, complete);
      :: restart_linear(1000)
      satisfy
```

does not very much sense since the underlying search is deterministic and each restart will just redo the same search as the previous search. Some solvers record the parts of the search tree that have already been searched and avoid them. This will mean deterministic restarts will simply effectively continue the search from the previous position. This gives no benefit to restarts, whose aim is to change decisions high in the search tree.

The simplest way to ensure that something is different in each restart is to use some randomization, either in variable choice or value choice. Alternatively some variable selection strategies make use of information gathered from earlier search and hence will give different behaviour, for example `dom_w_deg`.

To see the effectiveness of restart lets examine the n-queens problem again with the underlying search strategy

```
int_search(q, first_fail, indomain_random, complete);
```

with one of four restart strategies

```
r1 = restart_constant(100);
r2 = restart_linear(100);
r3 = restart_geometric(1.5, 100);
r4 = restart_luby(100);
```

```
r5 = restart_none;
```

n	constant	linear	geometric	luby	none
10	35	35	35	35	14
15	36	36	36	36	22
20	15	15	15	16	
25	2212	345	51	25	
30	—	137	22	66	
35	—	1722	52	12	
40	148	148	194	148	15
100	183	183	183	183	103
500	1480	1480	1480	1480	1434
1000	994	994	994	994	994

THE CURRENT EXPERIMENT IS USELESS!

2.5.5 Warm Starts

In many cases when solving an optimization or satisfaction problem we may have solved a previous version of the problem which is very similar. In this case it can be advantageous to use the previous solution found when searching for solution to the new problem.

The warm start annotations are attached to the solve item, just like other search annotations.

Warm start search annotations

The different restart annotations control how frequently a restart occurs. Restarts occur when a limit in nodes is reached, where search returns to the top of the search tree and begins again. The possibilities are

- `warm_start(<vars>, <vals>)` where `<vars>` is a one dimensional array of integer variables, and `<vals>` is a one dimensional array of integer of the same length giving the warm start values for each integer variable in `<vars>`.
- `warm_start(<vars>, <vals>)` where `<vars>` is a one dimensional array of float variables, and `<vals>` is a one dimensional array of floats of the same length giving the warm start values for each float variable in `<vars>`.
- `warm_start(<vars>, <vals>)` where `<vars>` is a one dimensional array of Boolean variables, and `<vals>` is a one dimensional array of Booleans of the same length giving the warm start values for each Boolean variable in `<vars>`.
- `warm_start(<vars>, <vals>)` where `<vars>` is a one dimensional array of set variables, and `<vals>` is a one dimensional array of sets of integers of the same length giving the warm start values for each set variable in `<vars>`.

The warm start annotation is used by the solver as part of value selection. If the selected variable `v[i]` has in its current domain the warm start value `w[i]` then this is the value selected for the variable. If not the solver uses the existing value selection rule applicable to that variable.

ADD AN EXAMPLE OF THEIR USE (jobshop???)

CHAPTER 2.6

Effective Modelling Practices in MiniZinc

There are almost always multiple ways to model the same problem, some of which generate models which are efficient to solve, and some of which are not. In general it is very hard to tell a priori which models are the most efficient for solving a particular problem, and indeed it may critically depend on the underlying solver used, and search strategy. In this chapter we concentrate on modelling practices that avoid inefficiency in generating models and generated models.

2.6.1 Variable Bounds

Finite domain propagation engines, which are the principle type of solver targeted by MiniZinc, are more effective the tighter the bounds on the variables involved. They can also behave badly with problems which have subexpressions that take large integer values, since they may implicitly limit the size of integer variables.

Listing 2.6.1: A model with unbounded variables (`grocery.mzn`).

```
var int: item1;
var int: item2;
var int: item3;
var int: item4;

constraint item1 + item2 + item3 + item4 == 711;
constraint item1 * item2 * item3 * item4 == 711 * 100 * 100 * 100;

constraint      0 < item1 /\ item1 <= item2
               /\ item2 <= item3 /\ item3 <= item4;

solve satisfy;

output [":", show(item1), ", ", show(item2), ", ", show(item3), ", ",
```

```
show(item4, "}\n"];
```

The grocery problem shown in Listing 2.6.1 finds 4 items whose prices in dollars add up to 7.11 and multiply up to 7.11. The variables are declared unbounded. Running

```
$ minizinc --solver chuffed grocery.mzn
```

yields

```
=====UNSATISFIABLE=====
```

This is because the intermediate expressions in the multiplication are also `var int` and are given default bounds in the solver $-1,000,000 \dots 1,000,000$, and these ranges are too small to hold the values that the intermediate expressions may need to take.

Modifying the model so that the items are declared with tight bounds

```
var 1..711: item1;
var 1..711: item2;
var 1..711: item3;
var 1..711: item4;
```

results in a better model, since now MiniZinc can infer bounds on the intermediate expressions and use these rather than the default bounds. With this modification, executing the model gives

```
{120,125,150,316}
```

```
-----
```

Note however that even the improved model may be too difficult for some solvers. Running

```
$ minizinc --solver g12lazy grocery.mzn
```

does not return an answer, since the solver builds a huge representation for the intermediate product variables.

Bounding variables

Always try to use bounded variables in models. When using `let` declarations to introduce new variables, always try to define them with correct and tight bounds. This will make your model more efficient, and avoid the possibility of unexpected overflows. One exception is when you introduce a new variable which is immediately defined as equal to an expression. Usually MiniZinc will be able to infer effective bounds from the expression.

2.6.2 Effective Generators

Imagine we want to count the number of triangles (K_3 subgraphs) appearing in a graph. Suppose the graph is defined by an adjacency matrix: `adj[i, j]` is true if nodes `i` and `j` are adjacent.

We might write

```
int: count = sum ([ 1 | i,j,k in NODES where i < j /\ j < k
                  /\ adj[i,j] /\ adj[i,k] /\ adj[j,k]]);
```

which is certainly correct, but it examines all triples of nodes. If the graph is sparse we can do better by realising that some tests can be applied as soon as we select *i* and *j*.

```
int: count = sum ([ 1 | i,j in NODES where i < j /\ adj[i,j],
                   k in NODES where j < k /\ adj[i,k] /\ adj[j,k]]);
```

You can use the builtilin `trace` function to help determine what is happening inside generators.

Tracing

The function `trace(s,e)` prints the string *s* before evaluating the expression *e* and returning its value. It can be used in any context.

For example, we can see how many times the test is performed in the inner loop for both versions of the calculation.

```
int:count=sum([ 1 | i,j,k in NODES where
               trace("+" , i<j /\ j<k /\ adj[i,j] /\ adj[i,k] /\ adj[j,k]) ]);
adj = [| false, true, true, false
       | true, false, true, false
       | true, true, false, true
       | false, false, true, false |];
constraint trace("\n",true);
solve satisfy;
```

Produces the output:

```
+++++
-----
```

indicating the inner loop is evaluated 64 times while

```
int: count = sum ([ 1 | i,j in NODES where i < j /\ adj[i,j],
                   k in NODES where trace("+" , j < k /\ adj[i,k] /\_
                   adj[j,k])]);
```

Produces the output:

```
+++++
-----
```

indicating the inner loop is evaluated 16 times.

Note that you can use the dependent strings in `trace` to understand what is happening during model creation.

```
int: count = sum( i,j in NODES where i < j /\ adj[i,j])(
    sum([trace("("++show(i)++, "++show(j)++", "++show(k)++")", 1) |
        k in NODES where j < k /\ adj[i,k] /\ adj[j,k]]));
```

will print out each of triangles that is found in the calculation. It produces the output

```
(1,2,3)
-----
```

We have to admit that we cheated a bit here: In certain circumstances, the MiniZinc compiler is in fact able to re-order the arguments in a `where` clause automatically, so that they are evaluated as early as possible. In this case, adding the `trace` function in fact *prevented* this optimisation. In general, it is however a good idea to help the compiler get it right, by splitting the `where` clauses and placing them as close to the generators as possible.

2.6.3 Redundant Constraints

The form of a model will affect how well the constraint solver can solve it. In many cases adding constraints which are redundant, i.e. are logically implied by the existing model, may improve the search for solutions by making more information available to the solver earlier.

Consider the magic series problem from [Complex Constraints](#) (page 50). Running this for `n = 16` as follows:

```
$ minizinc --solver gecode --all-solutions --statistics magic-series.mzn -D
→"n=16;"
```

might result in output

```
s = [12, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0];
-----
=====
```

and the statistics showing 89 failures required.

We can add redundant constraints to the model. Since each number in the sequence counts the number of occurrences of a number we know that they sum up to `n`. Similarly we know that the sum of `s[i] * i` must also add up to `n` because the sequence is magic. Adding these constraints gives the model in [Listing 2.6.2](#).

Listing 2.6.2: Model solving the magic series problem with redundant constraints (`magic-series2.mzn`).

```
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)))); 
% redundant
constraint sum(i in 0..n-1)(s[i]) = n;
constraint sum(i in 0..n-1)(s[i] * i) = n;
solve satisfy;

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

```

Running the same problem as before

```
$ minizinc --solver gecode --all-solutions --statistics magic-series2.mzn -D
→"n=16;"
```

results in the same output, but with statistics showing just 14 failures explored. The redundant constraints have allowed the solver to prune the search much earlier.

2.6.4 Modelling Choices

There are many ways to model the same problem in MiniZinc, although some may be more natural than others. Different models may have very different efficiency of solving, and worse yet, different models may be better or worse for different solving backends. There are however some guidelines for usually producing better models:

Choosing between models

The better model is likely to have some of the following features

- smaller number of variables, or at least those that are not functionally defined by other variables
- smaller domain sizes of variables
- more succinct, or direct, definition of the constraints of the model
- uses global constraints as much as possible

In reality all this has to be tempered by how effective the search is for the model. Usually the effectiveness of search is hard to judge except by experimentation.

Consider the problem of finding permutations of n numbers from 1 to n such that the differences between adjacent numbers also form a permutation of numbers 1 to $n - 1$. Note that the u variables are functionally defined by the x variables so the raw search space is n^n . The obvious way to model this problem is shown in Listing 2.6.3.

Listing 2.6.3: A natural model for the all interval series problem prob007 in CSplib (allinterval.mzn).

```

include "alldifferent.mzn";

int: n;
```

```

array[1..n] of var 1..n: x;      % sequence of numbers
array[1..n-1] of var 1..n-1: u;  % sequence of differences

constraint alldifferent(x);
constraint alldifferent(u);
constraint forall(i in 1..n-1)(u[i] = abs(x[i+1] - x[i]));

solve :: int_search(x, first_fail, indomain_min, complete)
    satisfy;
output ["x = \\\(x);\n"];

```

In this model the array `x` represents the permutation of the `n` numbers and the constraints are naturally represented using `alldifferent`.

Running the model

```
$ minizinc --solver gecode --all-solutions --statistics allinterval.mzn -D
→ "n=10;"
```

finds all solutions in 16077 nodes and 71ms

An alternate model uses array `y` where `y[i]` gives the position of the number `i` in the sequence. We also model the positions of the differences using variables `v`. `v[i]` is the position in the sequence where the absolute difference `i` occurs. If the values of `y[i]` and `y[j]` differ by one where `j > i`, meaning the positions are adjacent, then `v[j-i]` is constrained to be the earliest of these positions. We can add two redundant constraints to this model: since we know that a difference of `n-1` must result, we know that the positions of 1 and `n` must be adjacent (`abs(y[1] - y[n]) = 1`), which also tell us that the position of difference `n-1` is the earlier of `y[1]` and `y[n]`, i.e. `v[n-1] = min(y[1], y[n])`. With this we can model the problem as shown in Listing 2.6.4. The output statement recreates the original sequence `x` from the array of positions `y`.

Listing 2.6.4: An inverse model for the all interval series problem prob007 in CSplib (allinterval2.mzn).

```

include "alldifferent.mzn";

int: n;

array[1..n] of var 1..n: y;  % position of each number
array[1..n-1] of var 1..n-1: v; % position of difference i

constraint alldifferent(y);
constraint alldifferent(v);
constraint forall(i,j in 1..n where i < j)(
    (y[i] - y[j] = 1 -> v[j-i] = y[j]) /\ 
    (y[j] - y[i] = 1 -> v[j-i] = y[i])
);

constraint abs(y[1] - y[n]) = 1 /\ v[n-1] = min(y[1], y[n]);

```

```

solve :: int_search(y, first_fail, indomain_min, complete)
    satisfy;

array[1..n] of 1..n: x :: output_only
= [ sum(i in 1..n)(i*(fix(y[j]) = i) | j in 1..n ];
output [ "x = \\\(x);\n" ]

```

The inverse model has the same size as the original model, in terms of number of variables and domain sizes. But the inverse model has a much more indirect way of modelling the relationship between the `y` and `v` variables as opposed to the relationship between `x` and `u` variables. Hence we might expect the original model to be better.

The command

```
$ minizinc --solver gecode --all-solutions --statistics allinterval2.mzn -D
→"n=10;"
```

finds all the solutions in 98343 nodes and 640 ms. So the more direct modelling of the constraints is clearly paying off.

Note that the inverse model prints out the answers using the same `x` view of the solution. The way this is managed is using `output_only` annotations. The array `x` is defined as a fixed array and annotated as `output_only`. This means it will only be evaluated, and can only be used in output statements. Once a solution for `y` is discovered the value of `x` is calculated during output processing, and hence can be displayed in the output.

Output_only annotation

The `output_only` annotation can be applied to variable definitions. The variable defined must not be a `var` type, it can only be `par`. The variable must also have a right hand side definition giving its value. This right hand side definition can make use of `fix` functions to access the values of decision variables, since it is evaluated at solution processing time

2.6.5 Multiple Modelling and Channels

When we have two models for the same problem it may be useful to use both models together by tying the variables in the two models together, since each can give different information to the solver.

Listing 2.6.5: A dual model for the all interval series problem prob007 in CSPlib (allinterval3.mzn).

```

include "inverse.mzn";

int: n;

array[1..n] of var 1..n: x; % sequence of numbers
array[1..n-1] of var 1..n-1: u; % sequence of differences

```

```

constraint forall(i in 1..n-1)(u[i] = abs(x[i+1] - x[i]));

array[1..n] of var 1..n: y; % position of each number
array[1..n-1] of var 1..n-1: v; % position of difference i

constraint inverse(x,y);
constraint inverse(u,v);

constraint abs(y[1] - y[n]) = 1 /\ v[n-1] = min(y[1], y[n]);

solve :: int_search(y, first_fail, indomain_min, complete)
    satisfy;

output ["x = ", show(x), "\n"];

```

[Listing 2.6.5](#) gives a dual model combining features of `allinterval.mzn` and `allinterval2.mzn`. The beginning of the model is taken from `allinterval.mzn`. We then introduce the `y` and `v` variables from `allinterval2.mzn`. We tie the variables together using the global `inverse` constraint: `inverse(x,y)` holds if `y` is the inverse function of `x` (and vice versa), that is $x[i] = j \leftrightarrow y[j] = i$. A definition is shown in [Listing 2.6.6](#). The model does not include the constraints relating the `y` and `v` variables, they are redundant (and indeed propagation redundant) so they do not add information for a propagation solver. The `alldifferent` constraints are also missing since they are made redundant (and propagation redundant) by the inverse constraints. The only constraints are the relationships of the `x` and `u` variables and the redundant constraints on `y` and `v`.

[Listing 2.6.6](#): A definition of the `inverse` global constraint (`inverse.mzn`).

```

predicate inverse(array[int] of var int: f,
                  array[int] of var int: invf) =
    forall(j in index_set(inv)) (inv[j] in index_set(f)) /\ 
    forall(i in index_set(f)) (
        f[i] in index_set(inv) /\ 
        forall(j in index_set(inv)) (j == f[i] <-> i == inv[j])
    );

```

One of the benefits of the dual model is that there is more scope for defining different search strategies. Running the dual model,

```
$ minizinc --solver g12fd --all-solutions --statistics allinterval3.mzn -D
  "n=10;"
```

which uses the search strategy of the inverse model, labelling the `y` variables, finds all solutions in 1714 choice points and 0.5s. Note that running the same model with labelling on the `x` variables requires 13142 choice points and 1.5s.

2.6.6 Symmetry

Symmetry is very common in constraint satisfaction and optimisation problems. To illustrate this, let us look again at the n-queens problem from Listing 2.5.1. The top left chess board in Fig. 2.6.1 shows a solution to the 8-queens problems (labeled “original”). The remaining chess boards show seven symmetric variants of the same solution: rotated by 90, 180 and 270 degrees, and flipped vertically.

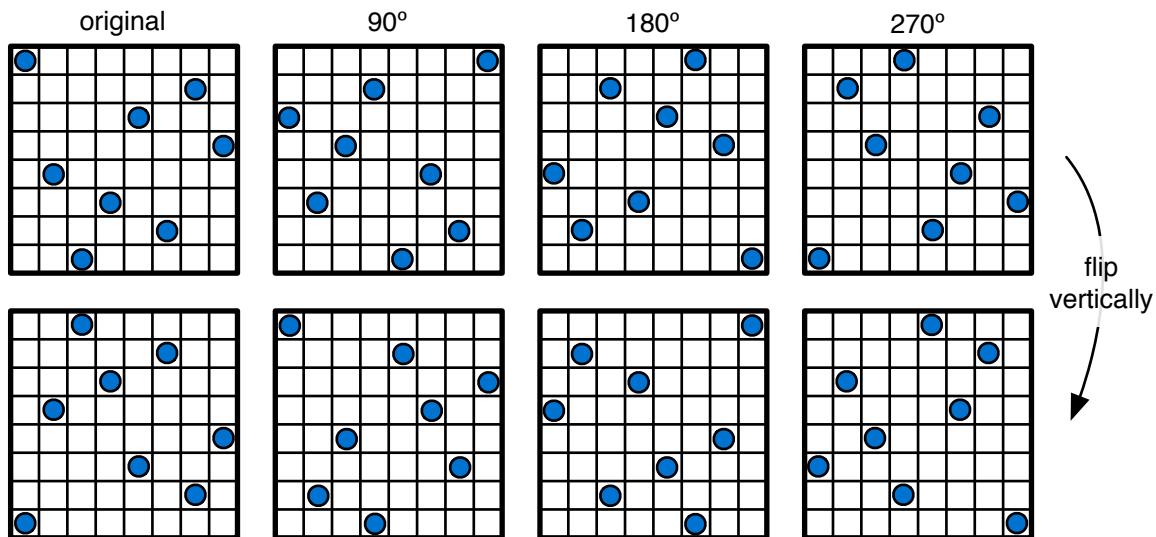


Fig. 2.6.1: Symmetric variants of an 8-queens solution

If we wanted to enumerate *all* solutions to the 8-queens problem, we could obviously save the solver some work by only enumerating *non-symmetric* solutions, and then generating the symmetric variants ourselves. This is one reason why we want to get rid of symmetry in constraint models. The other, much more important reason, is that the solver may also **explore symmetric variants of non-solution states!**

For example, a typical constraint solver may try to place the queen in column 1 into row 1 (which is fine), and then try to put the column 2 queen into row 3, which, at first sight, does not violate any of the constraints. However, this configuration cannot be completed to a full solution (which the solver finds out after a little search). Fig. 2.6.2 shows this configuration on the top left chess board. Now nothing prevents the solver from trying, e.g., the second configuration from the left in the bottom row of Fig. 2.6.2, where the queen in column 1 is still in row 1, and the queen in column 3 is placed in row 2. Therefore, even when only searching for a single solution, the solver may explore many symmetric states that it has already seen and proven unsatisfiable before!

2.6.6.1 Static Symmetry Breaking

The modelling technique for dealing with symmetry is called *symmetry breaking*, and in its simplest form, involves adding constraints to the model that rule out all symmetric variants of a (partial) assignment to the variables except one. These constraints are called *static symmetry breaking constraints*.

The basic idea behind symmetry breaking is to impose an *order*. For example, to rule out any vertical flips of the chess board, we could simply add the constraint that the queen in the first column must be in the top half of the board:

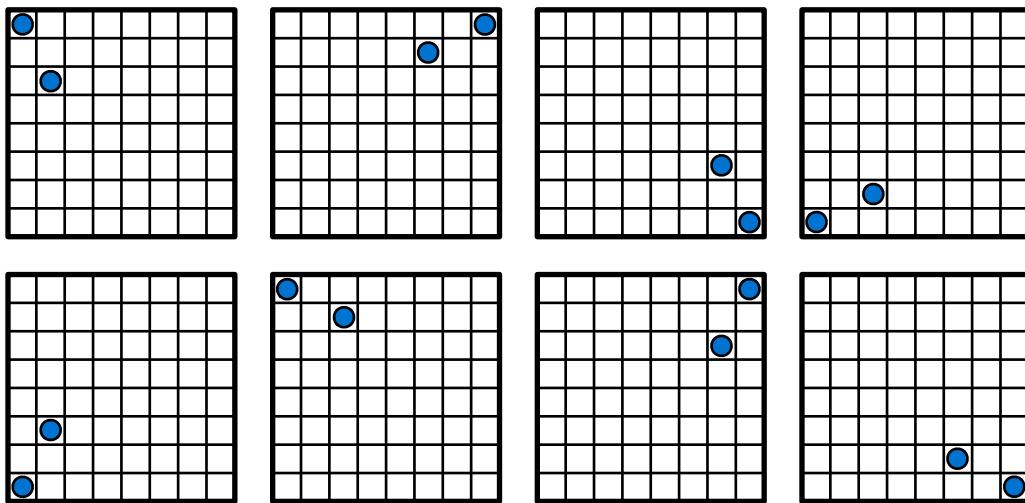


Fig. 2.6.2: Symmetric variants of an 8-queens unsatisfiable partial assignment

```
constraint q[1] <= n div 2;
```

Convince yourself that this would remove exactly half of the symmetric variants in Fig. 2.6.1. In order to remove *all* symmetry, we need to work a bit harder.

Whenever we can express all symmetries as permutations of the array of variables, a set of *lexicographic ordering constraints* can be used to break all symmetry. For example, if the array of variables is called x , and reversing the array is a symmetry of the problem, then the following constraint will break that symmetry:

```
constraint lex_lesseq(x, reverse(x));
```

How about two-dimensional arrays? Lexicographic ordering works just the same, we only have to coerce the arrays into one dimension. For example, the following breaks the symmetry of flipping the array along one of the diagonals (note the swapped indices in the second comprehension):

```
array[1..n,1..n] of var int: x;
constraint lex_lesseq([ x[i,j] | i,j in 1..n ], [ x[j,i] | i,j in 1..n ]);
```

The great thing about using lexicographic ordering constraints is that we can add multiple ones (to break several symmetries simultaneously), without them interfering with each other, as long as we keep the order in the first argument the same.

For the n-queens problem, unfortunately this technique does not immediately apply, because some of its symmetries cannot be described as permutations of the q array. The trick to overcome this is to express the n-queens problem in terms of Boolean variables that model, for each field of the board, whether it contains a queen or not. Now all the symmetries can be modeled as permutations of this array. Since the main constraints of the n-queens problem are much easier to express with the integer q array, we simply use both models together and add channeling constraints between them, as explained in [Multiple Modelling and Channels](#) (page 101).

The full model, with added Boolean variables, channeling constraints and symmetry breaking

constraints is shown in Listing 2.6.7. We can conduct a little experiment to check whether it successfully breaks all the symmetry. Try running the model with increasing values for n , e.g. from 1 to 10, counting the number of solutions (e.g., by using the `-s` flag with the Gecode solver, or selecting “Print all solutions” as well as “Statistics for solving” in the IDE). You should get the following sequence of numbers of solutions: 1, 0, 0, 1, 2, 1, 6, 12, 46, 92. To verify the sequence, you can search for it in the *On-Line Encyclopedia of Integer Sequences* (<http://oeis.org>).

Listing 2.6.7: Partial model for n-queens with symmetry breaking (full model: `nqueens_sym.mzn`).

```
% Map each position i,j to a Boolean telling us whether there is a queen at
% i,j
array[1..n,1..n] of var bool: qb;

% Channeling constraint
constraint forall (i,j in 1..n) ( qb[i,j] <-> (q[i]=j) );

% Lexicographic symmetry breaking constraints
constraint
    lex_lesseq(array1d(qb), [ qb[j,i] | i,j in 1..n ])
    /\ lex_lesseq(array1d(qb), [ qb[i,j] | i in reverse(1..n), j in 1..n ])
    /\ lex_lesseq(array1d(qb), [ qb[j,i] | i in 1..n, j in reverse(1..n) ])
    /\ lex_lesseq(array1d(qb), [ qb[i,j] | i in 1..n, j in reverse(1..n) ])
    /\ lex_lesseq(array1d(qb), [ qb[j,i] | i in reverse(1..n), j in 1..n ])
    /\ lex_lesseq(array1d(qb), [ qb[i,j] | i,j in reverse(1..n) ])
    /\ lex_lesseq(array1d(qb), [ qb[j,i] | i,j in reverse(1..n) ])
;
```

2.6.6.2 Other Examples of Symmetry

Many other problems have inherent symmetries, and breaking these can often make a significant difference in solving performance. Here is a list of some common cases:

- Bin packing: when trying to pack items into bins, any two bins that have the same capacity are symmetric.
- Graph colouring: When trying to assign colours to nodes in a graph such that adjacent nodes must have different colours, we typically model colours as integer numbers. However, any permutation of colours is again a valid graph colouring.
- Vehicle routing: if the task is to assign customers to certain vehicles, any two vehicles with the same capacity may be symmetric (this is similar to the bin packing example).
- Rostering/time tabling: two staff members with the same skill set may be interchangeable, just like two rooms with the same capacity or technical equipment.

CHAPTER 2.7

Boolean Satisfiability Modelling in MiniZinc

MiniZinc can be used to model Boolean satisfiability problems where the variables are restricted to be Boolean (`bool`). MiniZinc can be used with efficient Boolean satisfiability solvers to solve the resulting models efficiently.

2.7.1 Modelling Integers

Many times although we wish to use a Boolean satisfiability solver we may need to model some integer parts of our problem.

There are three common ways of modelling an integer variables I in the range $0 \dots m$ where $m = 2^k - 1$ using Boolean variables.

- Binary: I is represented by k binary variables i_0, \dots, i_{k-1} where $I = 2^{k-1}i_{k-1} + 2^{k-2}i_{k-2} + \dots + 2i_1 + i_0$. This can be represented in MiniZinc as

```
array[0..k-1]  of var bool: i;
var 0..pow(2,k)-1: I = sum(j in 0..k-1)(bool2int(i[j])*pow(2,j));
```

- Unary: where I is represented by m binary variables i_1, \dots, i_m and $i = \sum_{j=1}^m \text{bool2int}(i_j)$. Since there is massive redundancy in the unary representation we usually require that $i_j \rightarrow i_{j-1}, 1 < j \leq m$. This can be represented in MiniZinc as

```
array[1..m]  of var bool: i;
constraint forall(j in 2..m)(i[j] -> i[j-1]);
var 0..m: I = sum(j in 1..m)(bool2int(i[j]));
```

- Value: where I is represented by $m + 1$ binary variables i_0, \dots, i_m where $i = k \Leftrightarrow i_k$, and at most one of i_0, \dots, i_m is true. This can be represented in MiniZinc as

```

array[0..m]  of var bool: i;
constraint sum(j in 0..m)(bool2int(i[j])) == 1;
var 0..m: I;
constraint foall(j in 0..m)(I == j <-> i[j]);

```

There are advantages and disadvantages to each representation. It depends on what operations on integers are required in the model as to which is preferable.

2.7.2 Modelling Disequality

Let us consider modelling a latin squares problem. A latin square is an $n \times n$ grid of numbers from $1..n$ such that each number appears exactly once in every row and column. An integer model for latin squares is shown in Listing 2.7.1.

Listing 2.7.1: Integer model for Latin Squares (latin.mzn).

```

int: n; % size of latin square
array[1..n,1..n] of var 1..n: a;

include "alldifferent.mzn";
constraint forall(i in 1..n)(
    alldifferent(j in 1..n)(a[i,j]) /\ 
    alldifferent(j in 1..n)(a[j,i])
);
solve satisfy;
output [ show(a[i,j]) ++ if j == n then "\n" else " " endif |
    i in 1..n, j in 1..n ];

```

The only constraint on the integers is in fact disequality, which is encoded in the `alldifferent` constraint. The value representation is the best way of representing disequality. A Boolean only model for latin squares is shown in Listing 2.7.2. Note each integer array element $a[i,j]$ is replaced by an array of Booleans. We use the `exactlyone` predicate to encode that each value is used exactly once in every row and every column, as well as to encode that exactly one of the Booleans corresponding to integer array element $a[i,j]$ is true.

Listing 2.7.2: Boolean model for Latin Squares (latinbool.mzn).

```

int: n; % size of latin square
array[1..n,1..n,1..n] of var bool: a;

predicate atmostone(array[int] of var bool:x) =
    forall(i,j in index_set(x) where i < j)(
        (not x[i] \/\ not x[j]));
predicate exactlyone(array[int] of var bool:x) =
    atmostone(x) /\ exists(x);

constraint forall(i,j in 1..n)(
    exactlyone(k in 1..n)(a[i,j,k]) /\ 

```

```

        exactlyone(k in 1..n)(a[i,k,j]) /\ 
        exactlyone(k in 1..n)(a[k,i,j])
    );
solve satisfy;
output [ if fix(a[i,j,k]) then
            show(k) ++ if j == n then "\n" else " " endif
        else "" endif | i,j,k in 1..n ];

```

2.7.3 Modelling Cardinality

Let us consider modelling the Light Up puzzle. The puzzle consists of a rectangular grid of squares which are blank, or filled. Every filled square may contain a number from 1 to 4, or may have no number. The aim is to place lights in the blank squares so that

- Each blank square is “illuminated”, that is can see a light through an uninterrupted line of blank squares
- No two lights can see each other
- The number of lights adjacent to a numbered filled square is exactly the number in the filled square.

An example of a Light Up puzzle is shown in Fig. 2.7.1 with its solution in Fig. 2.7.2.

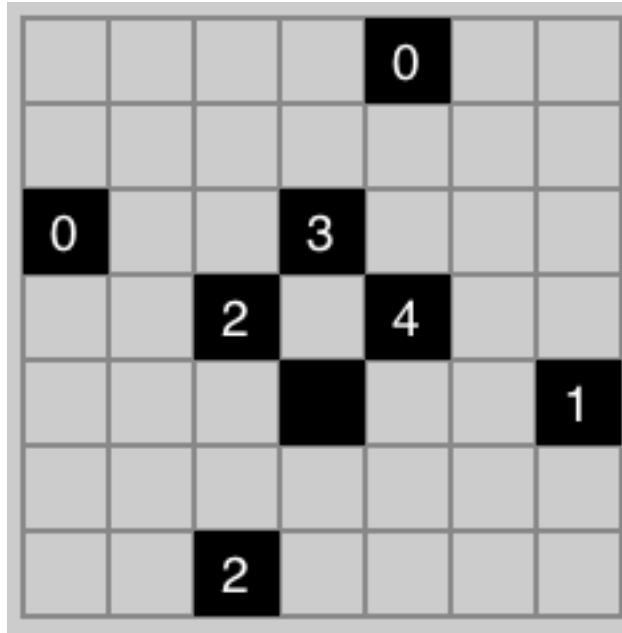


Fig. 2.7.1: An example of a Light Up puzzle

It is natural to model this problem using Boolean variables to determine which squares contain a light and which do not, but there is some integer arithmetic to consider for the filled squares.

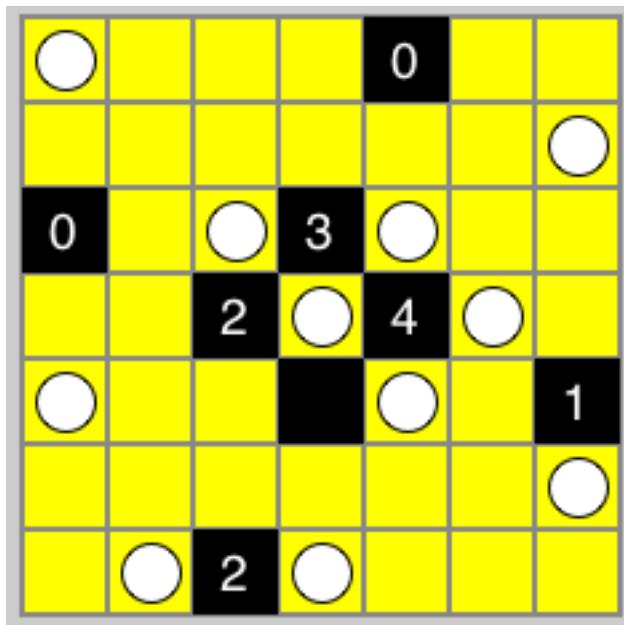


Fig. 2.7.2: The completed solution of the Light Up puzzle

Listing 2.7.3: SAT Model for the Light Up puzzle (lightup.mzn).

```

int: h; set of int: H = 1..h; % board height
int: w; set of int: W = 1..w; % board width
array[H,W] of -1..5: b;           % board
int: E = -1;                      % empty square
set of int: N = 0..4;              % filled and numbered square
int: F = 5;                        % filled unnumbered square

% position (i1,j1) is visible to (i2,j2)
test visible(int: i1, int: j1, int: i2, int: j2) =
  ((i1 == i2) /\ forall(j in min(j1,j2)..max(j1,j2))(b[i1,j] == E))
  \vee ((j1 == j2) /\ forall(i in min(i1,i2)..max(i1,i2))(b[i,j1] == E));

array[H,W] of var bool: l; % is there a light

% filled squares have no lights
constraint forall(i in H, j in W, where b[i,j] != E)(l[i,j] == false);
% lights next to filled numbered square agree
include "boolsum.mzn";
constraint forall(i in H, j in W where b[i,j] in N)(
  bool_sum_eq([ l[i1,j1] | i1 in i-1..i+1, j1 in j-1..j+1 where
    abs(i1 - i) + abs(j1 - j) == 1 /\
    i1 in H /\ j1 in W ], b[i,j]));
% each blank square is illuminated
constraint forall(i in H, j in W where b[i,j] == E)(
  exists(j1 in W where visible(i,j,i,j1))(l[i,j1]) \/
  exists(i1 in H where visible(i,j,i1,j))(l[i1,j]))
);
% no two lights see each other

```

```

constraint forall(i1,i2 in H, j1,j2 in W where
    (i1 != i2 \vee j1 != j2) /\ b[i1,j1] == E
    /\ b[i2,j2] == E /\ visible(i1,j1,i2,j2))(

    not l[i1,j1] \vee not l[i2,j2]
);

solve satisfy;
output [ if b[i,j] != E then show(b[i,j])
    else if fix(l[i,j]) then "L" else "." endif
    endif ++ if j == w then "\n" else " " endif |
i in H, j in W];

```

A model for the problem is given in Listing 2.7.3. A data file for the problem shown in Fig. 2.7.1 is shown in Listing 2.7.4.

Listing 2.7.4: Datafile for the Light Up puzzle instance shown in Fig. 2.7.1.

```

h = 7;
w = 7;
b = [| -1,-1,-1,-1, 0,-1,-1
      | -1,-1,-1,-1,-1,-1,-1
      |  0,-1,-1, 3,-1,-1,-1
      | -1,-1, 2,-1, 4,-1,-1
      | -1,-1,-1, 5,-1,-1, 1
      | -1,-1,-1,-1,-1,-1,-1
      |  1,-1, 2,-1,-1,-1,-1 |];

```

The model makes use of a Boolean sum predicate

```
predicate bool_sum_eq(array[int] of var bool:x, int:s);
```

which requires that the sum of an array of Boolean equals some fixed integer. There are a number of ways of modelling such *cardinality* constraints using Booleans.

- Adder networks: we can use a network of adders to build a binary Boolean representation of the sum of the Booleans
- Sorting networks: we can use a sorting network to sort the array of Booleans to create a unary representation of the sum of the Booleans
- Binary decision diagrams: we can create a binary decision diagram (BDD) that encodes the cardinality constraint.

Listing 2.7.5: Cardinality constraints by binary adder networks (bboolsum.mzn).

```
% the sum of booleans x = s
predicate bool_sum_eq(array[int] of var bool:x, int:s) =
    let { int: c = length(x) } in
    if s < 0 then false
    elseif s == 0 then
        forall(i in 1..c)(x[i] == false)
```

```

elseif s < c then
    let { % cp = number of bits required for representing 0..c
        int: cp = floor(log2(int2float(c))),
        % z is sum of x in binary
        array[0..cp] of var bool:z } in
    binary_sum(x, z) /\
    % z == s
    forall(i in 0..cp)(z[i] == ((s div pow(2,i)) mod 2 == 1))
elseif s == c then
    forall(i in 1..c)(x[i] == true)
else false endif;

include "binarysum.mzn";

```

Listing 2.7.6: Code for building binary addition networks (binarysum.mzn).

```

% the sum of bits x = s in binary.
%           s[0], s[1], ..., s[k] where 2^k >= length(x) > 2^(k-1)
predicate binary_sum(array[int] of var bool:x,
                     array[int] of var bool:s)=
let { int: l = length(x) } in
if l == 1 then s[0] = x[1]
elseif l == 2 then
    s[0] = (x[1] xor x[2]) /\ s[1] = (x[1] /\ x[2])
else let { int: ll = (l div 2),
           array[1..ll] of var bool: f = [ x[i] | i in 1..ll ],
           array[1..ll] of var bool: t = [x[i]| i in ll+1..2*ll],
           var bool: b = if ll*2 == l then false else x[1] endif,
           int: cp = floor(log2(int2float(ll))),
           array[0..cp] of var bool: fs,
           array[0..cp] of var bool: ts } in
    binary_sum(f, fs) /\ binary_sum(t, ts) /\
    binary_add(fs, ts, b, s)
endif;

% add two binary numbers x, and y and carry in bit ci to get binary s
predicate binary_add(array[int] of var bool: x,
                     array[int] of var bool: y,
                     var bool: ci,
                     array[int] of var bool: s) =
let { int:l = length(x),
      int:n = length(s), } in
assert(l == length(y),
      "length of binary_add input args must be same",
      assert(n == 1 \vee n == l+1, "length of binary_add output " ++
             "must be equal or one more than inputs",
      let { array[0..1] of var bool: c } in
      full_adder(x[0], y[0], ci, s[0], c[0]) /\
      forall(i in 1..l)(full_adder(x[i], y[i], c[i-1], s[i], c[i])) /\
      if n > 1 then s[n] = c[1] else c[1] == false endif));

```

```

predicate full_adder(var bool: x, var bool: y, var bool: ci,
                     var bool: s, var bool: co) =
    let { var bool: xy = x xor y } in
    s = (xy xor ci) /\ co = ((x /\ y) \vee (ci /\ xy));

```

We can implement `bool_sum_eq` using binary adder networks using the code shown in Listing 2.7.5. The predicate `binary_sum` defined in Listing 2.7.6 creates a binary representation of the sum of `x` by splitting the list into two, summing up each half to create a binary representation and then summing these two binary numbers using `binary_add`. If the list `x` is odd the last bit is saved to use as a carry in to the binary addition.

Listing 2.7.7: Cardinality constraints by sorting networks (uboolsum.mzn).

```

% the sum of booleans x = s
predicate bool_sum_eq(array[int] of var bool:x, int:s) =
    let { int: c = length(x) } in
    if s < 0 then false
    elseif s == 0 then forall(i in 1..c)(x[i] == false)
    elseif s < c then
        let { % cp = nearest power of 2 >= c
            int: cp = pow(2,ceil(log2(int2float(c)))), 
            array[1..cp] of var bool:y, % y is padded version of x
            array[1..cp] of var bool:z } in
            forall(i in 1..c)(y[i] == x[i]) /\ 
            forall(i in c+1..cp)(y[i] == false) /\ 
            oesort(y, z) /\ z[s] == true /\ z[s+1] == false
        elseif s == c then forall(i in 1..c)(x[i] == true)
        else false endif;
    include "oesort.mzn";

```

Listing 2.7.8: Odd-even merge sorting networks (oesort.mzn).

```

%% odd-even sort
%% y is the sorted version of x, all trues before falses
predicate oesort(array[int] of var bool:x, array[int] of var bool:y)=
    let { int: c = card(index_set(x)) } in
    if c == 1 then x[1] == y[1]
    elseif c == 2 then comparator(x[1],x[2],y[1],y[2])
    else
        let {
            array[1..c div 2] of var bool:xf = [x[i] | i in 1..c div 2],
            array[1..c div 2] of var bool:xl = [x[i] | i in c div 2 +1..c],
            array[1..c div 2] of var bool:tf,
            array[1..c div 2] of var bool:tl } in
            oesort(xf,tf) /\ oesort(xl,tl) /\ oemerge(tf ++ tl, y)
        endif;

```

```

%% odd-even merge
%% y is the sorted version of x, all trues before falses
%% assumes first half of x is sorted, and second half of x
predicate oemerge(array[int] of var bool:x, array[int] of var bool:y)=
    let { int: c = card(index_set(x)) } in
    if c == 1 then x[1] == y[1]
    elseif c == 2 then comparator(x[1],x[2],y[1],y[2])
    else
        let { array[1..c div 2] of var bool:xo =
            [ x[i] | i in 1..c where i mod 2 == 1 ],
            array[1..c div 2] of var bool:xe =
            [ x[i] | i in 1..c where i mod 2 == 0 ],
            array[1..c div 2] of var bool:to,
            array[1..c div 2] of var bool:te } in
            oemerge(xo,to) /\ oemerge(xe,te) /\
            y[1] = to[1] /\
            forall(i in 1..c div 2 -1)(
                comparator(te[i],to[i+1],y[2*i],y[2*i+1])) /\
            y[c] = te[c div 2]
        endif);
    endif);

% comparator o1 = max(i1,i2), o2 = min(i1,i2)
predicate comparator(var bool:i1,var bool:i2,var bool:o1,var bool:o2)=
    (o1 = (i1 \vee i2)) /\ (o2 = (i1 /\ i2));

```

We can implement `bool_sum_eq` using unary sorting networks using the code shown in Listing 2.7.7. The cardinality constraint is defined by expanding the input `x` to have length a power of 2, and sorting the resulting bits using an odd-even merge sorting network. The odd-even merge sorter shown in `ex-oesort` works recursively by splitting the input list in 2, sorting each list and merging the two sorted lists.

Listing 2.7.9: Cardinality constraints by binary decision diagrams (`bddsum.mzn`).

```

% the sum of booleans x = s
predicate bool_sum_eq(array[int] of var bool:x, int:s) =
    let { int: c = length(x),
          array[1..c] of var bool: y = [x[i] | i in index_set(x)] }
    } in
    rec_bool_sum_eq(y, 1, s);

predicate rec_bool_sum_eq(array[int] of var bool:x, int: f, int:s) =
    let { int: c = length(x) } in
    if s < 0 then false
    elseif s == 0 then
        forall(i in f..c)(x[i] == false)
    elseif s < c - f + 1 then
        (x[f] == true /\ rec_bool_sum_eq(x,f+1,s-1)) \/
        (x[f] == false /\ rec_bool_sum_eq(x,f+1,s))
    elseif s == c - f + 1 then
        forall(i in f..c)(x[i] == true)

```

```
else false endif;
```

We can implement `bool_sum_eq` using binary decision diagrams using the code shown in `ex:bddsum`. The cardinality constraint is broken into two cases: either the first element `x[1]` is true, and the sum of the remaining bits is `s-1`, or `x[1]` is false and the sum of the remaining bits is `s`. For efficiency this relies on common subexpression elimination to avoid creating many equivalent constraints.

CHAPTER 2.8

FlatZinc and Flattening

Constraint solvers do not directly support MiniZinc models, rather in order to run a MiniZinc model, it is translated into a simple subset of MiniZinc called FlatZinc. FlatZinc reflects the fact that most constraint solvers only solve satisfaction problems of the form $\exists c_1 \wedge \dots \wedge c_m$ or optimization problems of the form minimize z subject to $c_1 \wedge \dots \wedge c_m$, where c_i are primitive constraints and z is an integer or float expression in a restricted form.

The `minizinc` tool includes the MiniZinc *compiler*, which takes a MiniZinc model and data files and creates a flattened FlatZinc model which is equivalent to the MiniZinc model with the given data, and that appears in the restricted form discussed above. Normally the construction of a FlatZinc model which is sent to a solver is hidden from the user but you can view the result of flattening a model `model.mzn` with its data `data.dzn` as follows:

```
minizinc -c model.mzn data.dzn
```

which creates a FlatZinc model called `model.fzn`.

In this chapter we explore the process of translation from MiniZinc to FlatZinc.

2.8.1 Flattening Expressions

The restrictions of the underlying solver mean that complex expressions in MiniZinc need to be *flattened* to only use conjunctions of primitive constraints which do not themselves contain structured terms.

Consider the following model for ensuring that two circles in a rectangular box do not overlap:

Listing 2.8.1: Modelling non overlap of two circles (`cnonoverlap.mzn`).

```
float: width;           % width of rectangle to hold circles
float: height;          % height of rectangle to hold circles
float: r1;
```

```

var r1..width-r1: x1; % (x1,y1) is center of circle of radius r1
var r1..height-r1: y1;
float: r2;
var r2..width-r2: x2; % (x2,y2) is center of circle of radius r2
var r2..height-r2: y2;
    % centers are at least r1 + r2 apart
constraint (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) >= (r1+r2)*(r1+r2);
solve satisfy;

```

2.8.1.1 Simplification and Evaluation

Given the data file

```

width = 10.0;
height = 8.0;
r1 = 2.0;
r2 = 3.0;

```

the translation to FlatZinc first simplifies the model by replacing all the parameters by their values, and evaluating any fixed expression. After this simplification the values of parameters are not longer needed. An exception to this is large arrays of parametric values. If they are used more than once, then the parameter is retained to avoid duplicating the large expression.

After simplification the variable and parameter declarations parts of the model of Listing 2.8.1 become

```

var 2.0 .. 8.0: x1;
var 2.0 .. 6.0: y1;
var 3.0 .. 7.0: x2;
var 3.0 .. 5.0: y2;

```

2.8.1.2 Defining Subexpressions

Now no constraint solver directly handles complex constraint expressions like the one in Listing 2.8.1. Instead, each subexpression in the expression is named, and we create a constraint to construct the value of each expression. Let's examine the subexpressions of the constraint expression. $(x_1 - x_2)$ is a subexpression, if we name it `FLOAT01` we can define it as `constraint FLOAT01 = x1 - x2;` Notice that this expression occurs twice in the model. We only need to construct the value once, we can then reuse it. This is called *common subexpression elimination*. The subexpression $(x_1 - x_2) * (x_1 - x_2)$ can be named `FLOAT02` and we can define it as `constraint FLOAT02 = FLOAT01 * FLOAT01;` We can similarly name `constraint FLOAT03 = y1 - y2;` and `constraint FLOAT04 = FLOAT03 * FLOAT03;` and finally `constraint FLOAT05 = FLOAT02 * FLOAT04;`. The inequality constraint itself becomes `constraint FLOAT05 >= 25.0;` since $(r_1+r_2) * (r_1+r_2)$ is calculated as `25.0`. The flattened constraint is hence

```

constraint FLOAT01 = x1 - x2;
constraint FLOAT02 = FLOAT01 * FLOAT01;
constraint FLOAT03 = y1 - y2;
constraint FLOAT04 = FLOAT03 * FLOAT03;
constraint FLOAT05 = FLOAT02 * FLOAT04;
constraint FLOAT05 >= 25.0

```

2.8.1.3 FlatZinc constraint form

Flattening as its final step converts the form of the constraint to a standard FlatZinc form which is always $p(a_1, \dots, a_n)$ where p is the name of the primitive constraint and a_1, \dots, a_n are the arguments. FlatZinc tries to use a minimum of different constraint forms so for example the constraint `FLOAT01 = x1 - x2` is first rewritten to `FLOAT01 + x2 = x1` and then output using the `float_plus` primitive constraint. The resulting constraint form is as follows:

```

constraint float_plus(FLOAT01, x2, x1);
constraint float_plus(FLOAT03, y2, y1);
constraint float_plus(FLOAT02, FLOAT04, FLOAT05);
constraint float_times(FLOAT01, FLOAT01, FLOAT02);
constraint float_times(FLOAT03, FLOAT03, FLOAT04);

```

2.8.1.4 Bounds analysis

We are still missing one thing, the declarations for the introduced variables `FLOAT01, ..., FLOAT05`. While these could just be declared as `var float`, in order to make the solver's task easier MiniZinc tries to determine upper and lower bounds on newly introduced variables, by a simple bounds analysis. For example since `FLOAT01 = x1 - x2` and $2.0 \leq x1 \leq 8.0$ and $3.0 \leq x2 \leq 7.0$ then we can see that $-5.0 \leq \text{FLOAT01} \leq 5.0$. Given this information we can see that $-25.0 \leq \text{FLOAT02} \leq 25.0$ (although note that if we recognized that the multiplication was in fact a squaring we could give the much more accurate bounds $0.0 \leq \text{FLOAT02} \leq 25.0$).

The alert reader may have noticed a discrepancy between the flattened form of the constraints in *Defining Subexpressions* (page 118) and *FlatZinc constraint form* (page 119). In the latter there is no inequality constraint. Since unary inequalities can be fully represented by the bounds of a variable, the inequality forces the lower bound of `FLOAT05` to be `25.0` and is then redundant. The final flattened form of the model of Listing 2.8.1 is:

```

% Variables
var 2.0 .. 8.0: x1;
var 2.0 .. 6.0: y1;
var 3.0 .. 7.0: x2;
var 3.0 .. 5.0: y2;
%
var -5.0..5.0:  FLOAT01;
var -25.0..25.0: FLOAT02;
var -3.0..3.0:   FLOAT03;
var -9.0..9.0:   FLOAT04;

```

```

var 25.0..34.0: FLOAT05;
% Constraints
constraint float_plus(FLOAT01, x2, x1);
constraint float_plus(FLOAT03, y2, y1);
constraint float_plus(FLOAT02, FLOAT04, FLOAT05);
constraint float_times(FLOAT01, FLOAT01, FLOAT02);
constraint float_times(FLOAT03, FLOAT03, FLOAT04);
%
solve satisfy;

```

2.8.1.5 Objectives

MiniZinc flattens minimization or maximization objectives just like constraints. The objective expression is flattened and a variable is created for it, just as for other expressions. In the FlatZinc output the solve item is always on a single variable. See *Let Expressions* (page 129) for an example.

2.8.2 Linear Expressions

One of the most important form of constraints, widely used for modelling, are linear constraints of the form

$$\begin{array}{c} = \\ a_1x_1 + \dots + a_nx_n \leq a_0 \\ < \end{array}$$

where a_i are integer or floating point constants, and x_i are integer or floating point variables. They are highly expressive, and are the only class of constraint supported by (integer) linear programming constraint solvers. The translator from MiniZinc to FlatZinc tries to create linear constraints, rather than break up linear constraints into many subexpressions.

Listing 2.8.2: A MiniZinc model to illustrate linear constraint flattening (`linear.mzn`).

```

int:      d = -1;
var 0..10: x;
var -3..6: y;
var 3..8: z;
constraint 3*x - y + x * z <= 19 + d * (x + y + z) - 4*d;
solve satisfy;

```

Consider the model shown in Listing 2.8.2. Rather than create variables for all the subexpressions $3 * x$, $3 * x - y$, $x * z$, $3 * x - y + x * z$, $x + y + z$, $d * (x + y + z)$, $19 + d * (x + y + z)$, and $19 + d * (x + y + z) - 4 * d$ translation will attempt to create a large linear constraint which captures as much as possible of the constraint in a single FlatZinc constraint.

Flattening creates linear expressions as a single unit rather than building intermediate variables for each subexpression. It also simplifies the linear expression created. Extracting the linear expression from the constraints leads to

```
var 0..80: INT01;
constraint 4*x + z + INT01 <= 23;
constraint INT01 = x * z;
```

Notice how the *nonlinear expression* $x \times z$ is extracted as a new subexpression and given a name, while the remaining terms are collected together so that each variable appears exactly once (and indeed variable y whose terms cancel is eliminated).

Finally each constraint is written to FlatZinc form obtaining:

```
var 0..80: INT01;
constraint int_lin_le([1,4,1],[INT01,x,z],23);
constraint int_times(x,z,INT01);
```

2.8.3 Unrolling Expressions

Most models require creating a number of constraints which is dependent on the input data. MiniZinc supports these models with array types, list and set comprehensions, and aggregation functions.

Consider the following aggregate expression from the production scheduling example of Listing 2.2.2.

```
int: mproducts = max (p in Products)
    (min (r in Resources where consumption[p,r] > 0)
        (capacity[r] div consumption[p,r]));
```

Since this uses generator call syntax we can rewrite it to equivalent form which is processed by the compiler:

```
int: mproducts = max([ min [ capacity[r] div consumption[p,r]
    | r in Resources where consumption[p,r] > 0]
    | p in Products]);
```

Given the data

```
nproducts = 2;
nresources = 5;
capacity = [4000, 6, 2000, 500, 500];
consumption= [| 250, 2, 75, 100, 0,
    | 200, 0, 150, 150, 75 |];
```

this first builds the array for $p = 1$

```
[ capacity[r] div consumption[p,r]
    | r in 1..5 where consumption[p,r] > 0]
```

which is [16, 3, 26, 5] and then calculates the minimum as 3. It then builds the same array for $p = 2$ which is [20, 13, 3, 6] and calculates the minimum as 3. It then constructs the array [3, 3] and calculates the maximum as 3. There is no representation of `mproducts` in the output FlatZinc, this evaluation is simply used to replace `mproducts` by the calculated value 3.

The most common form of aggregate expression in a constraint model is `forall`. Forall expressions are unrolled into multiple constraints.

Consider the MiniZinc fragment

```
array[1..8] of var 0..9: v = [S,E,N,D,M,O,R,Y];
constraint forall(i,j in 1..8 where i < j)(v[i] != v[j])
```

which arises from the SEND-MORE-MONEY example of Listing 2.2.4 using a default decomposition for `alldifferent`. The `forall` expression creates a constraint for each i, j pair which meet the requirement $i < j$, thus creating

```
constraint v[1] != v[2]; % S != E
constraint v[1] != v[3]; % S != N
...
constraint v[1] != v[8]; % S != Y
constraint v[2] != v[3]; % E != N
...
constraint v[7] != v[8]; % R != Y
```

In FlatZinc form this is

```
constraint int_neq(S,E);
constraint int_neq(S,N);
...
constraint int_neq(S,Y);
constraint int_neq(E,N);
...
constraint int_neq(R,Y);
```

Notice how the temporary array variables `v[i]` are replaced by the original variables in the output FlatZinc.

2.8.4 Arrays

One dimensional arrays in MiniZinc can have arbitrary indices as long as they are contiguous integers. In FlatZinc all arrays are indexed from 1..1 where 1 is the length of the array. This means that array lookups need to be translated to the FlatZinc view of indices.

Consider the following MiniZinc model for balancing a seesaw of length $2 * 12$, with a child of weight `cw` kg using exactly `m` 1kg weights.

```
int: cw;                                % child weight
int: 12;                                 % half seesaw length
```

```

int: m;                                % number of 1kg weight
array[-12..12] of var 0..max(m,cw): w; % weight at each point
var -12..12: p;                         % position of child
constraint sum(i in -12..12)(i * w[i]) = 0; % balance
constraint sum(i in -12..12)(w[i]) = m + cw; % all weights used
constraint w[p] = cw;                   % child is at position p
solve satisfy;

```

Given $cw = 2$, $12 = 2$, and $m = 3$ the unrolling produces the constraints

```

array[-2..2] of var 0..3: w;
var -2..2: p
constraint -2*w[-2] + -1*w[-1] + 0*w[0] + 1*w[1] + 2*w[2] = 0;
constraint w[-2] + w[-1] + w[0] + w[1] + w[2] = 5;
constraint w[p] = 2;

```

But FlatZinc insists that the w array starts at index 1. This means we need to rewrite all the array accesses to use the new index value. For fixed array lookups this is easy, for variable array lookups we may need to create a new variable. The result for the equations above is

```

array[1..5] of var 0..3: w;
var -2..2: p
var 1..5: INT01;
constraint -2*w[1] + -1*w[2] + 0*w[3] + 1*w[4] + 2*w[5] = 0;
constraint w[1] + w[2] + w[3] + w[4] + w[5] = 5;
constraint w[INT01] = 2;
constraint INT01 = p + 3;

```

Finally we rewrite the constraints into FlatZinc form. Note how the variable array index lookup is mapped to `array_var_int_element`.

```

array[1..5] of var 0..3: w;
var -2..2: p
var 1..5: INT01;
constraint int_lin_eq([-2, 1, -1, -2], [w[1], w[2], w[4], w[5]], 0);
constraint int_lin_eq([1, 1, 1, 1, 1], [w[1], w[2], w[3], w[4], w[5]], 5);
constraint array_var_int_element(INT01, w, 2);
constraint int_lin_eq([-1, 1], [INT01, p], -3);

```

Multidimensional arrays are supported by MiniZinc, but only single dimension arrays are supported by FlatZinc (at present). This means that multidimensional arrays must be mapped to single dimension arrays, and multidimensional array access must be mapped to single dimension array access.

Consider the Laplace equation constraints defined for a finite element plate model in Listing 2.2.1:

```

set of int: HEIGHT = 0..h;
set of int: CHEIGHT = 1..h-1;
set of int: WIDTH = 0..w;
set of int: CWIDTH = 1..w-1;
array[HEIGHT,WIDTH] of var float: t; % temperature at point (i,j)
var float: left; % left edge temperature
var float: right; % right edge temperature
var float: top; % top edge temperature
var float: bottom; % bottom edge temperature

% equation
% Laplace equation: each internal temp. is average of its neighbours
constraint forall(i in CHEIGHT, j in CWIDTH)(
    4.0*t[i,j] = t[i-1,j] + t[i,j-1] + t[i+1,j] + t[i,j+1]);

```

Assuming $w = 4$ and $h = 4$ this creates the constraints

```

array[0..4,0..4] of var float: t; % temperature at point (i,j)
constraint 4.0*t[1,1] = t[0,1] + t[1,0] + t[2,1] + t[1,2];
constraint 4.0*t[1,2] = t[0,2] + t[1,1] + t[2,2] + t[1,3];
constraint 4.0*t[1,3] = t[0,3] + t[1,2] + t[2,3] + t[1,4];
constraint 4.0*t[2,1] = t[1,1] + t[2,0] + t[3,1] + t[2,2];
constraint 4.0*t[2,2] = t[1,2] + t[2,1] + t[3,2] + t[2,3];
constraint 4.0*t[2,3] = t[1,3] + t[2,2] + t[3,3] + t[2,4];
constraint 4.0*t[3,1] = t[2,1] + t[3,0] + t[4,1] + t[3,2];
constraint 4.0*t[3,2] = t[2,2] + t[3,1] + t[4,2] + t[3,3];
constraint 4.0*t[3,3] = t[2,3] + t[3,2] + t[4,3] + t[3,4];

```

The 2 dimensional array of 25 elements is converted to a one dimensional array and the indices are changed accordingly: so index $[i, j]$ becomes $[i * 5 + j + 1]$.

```

array [1..25] of var float: t;
constraint 4.0*t[7] = t[2] + t[6] + t[12] + t[8];
constraint 4.0*t[8] = t[3] + t[7] + t[13] + t[9];
constraint 4.0*t[9] = t[4] + t[8] + t[14] + t[10];
constraint 4.0*t[12] = t[7] + t[11] + t[17] + t[13];
constraint 4.0*t[13] = t[8] + t[12] + t[18] + t[14];
constraint 4.0*t[14] = t[9] + t[13] + t[19] + t[15];
constraint 4.0*t[17] = t[12] + t[16] + t[22] + t[18];
constraint 4.0*t[18] = t[13] + t[17] + t[23] + t[19];
constraint 4.0*t[19] = t[14] + t[18] + t[24] + t[20];

```

2.8.5 Reification

FlatZinc models involve only variables and parameter declarations and a series of primitive constraints. Hence when we model in MiniZinc with Boolean connectives other than conjunction, something has to be done. The core approach to handling complex formulae that use connec-

tives other than conjunction is by *reification*. Reifying a constraint c creates a new constraint equivalent to $b \leftrightarrow c$ where the Boolean variable b is true if the constraint holds and false if it doesn't hold.

Once we have the capability to *reify* constraints the treatment of complex formulae is not different from arithmetic expressions. We create a name for each subexpression and a flat constraint to constrain the name to take the value of its subexpression.

Consider the following constraint expression that occurs in the jobshop scheduling example of Listing 2.2.8.

```
constraint % ensure no overlap of tasks
  forall(j in 1..tasks) (
    forall(i,k in 1..jobs where i < k) (
      s[i,j] + d[i,j] <= s[k,j] \/
      s[k,j] + d[k,j] <= s[i,j]
    ) );
  
```

Given the data file

```
jobs = 2;
tasks = 3;
d = [| 5, 3, 4 | 2, 6, 3 |]
```

then the unrolling creates

```
constraint s[1,1] + 5 <= s[2,1] \\/ s[2,1] + 2 <= s[1,1];
constraint s[1,2] + 3 <= s[2,2] \\/ s[2,2] + 6 <= s[1,2];
constraint s[1,3] + 4 <= s[2,3] \\/ s[2,3] + 3 <= s[1,3];
```

Reification of the constraints that appear in the disjunction creates new Boolean variables to define the values of each expression.

```
array[1..2,1..3] of var 0..23: s;
constraint BOOL01 <-> s[1,1] + 5 <= s[2,1];
constraint BOOL02 <-> s[2,1] + 2 <= s[1,1];
constraint BOOL03 <-> s[1,2] + 3 <= s[2,2];
constraint BOOL04 <-> s[2,2] + 6 <= s[1,2];
constraint BOOL05 <-> s[1,3] + 4 <= s[2,3];
constraint BOOL06 <-> s[2,3] + 3 <= s[1,3];
constraint BOOL01 \/\ BOOL02;
constraint BOOL03 \/\ BOOL04;
constraint BOOL05 \/\ BOOL06;
```

Each primitive constraint can now be mapped to the FlatZinc form. Note how the two dimensional array s is mapped to a one dimensional form.

```
array[1..6] of var 0..23: s;
constraint int_lin_le_reif([1, -1], [s[1], s[4]], -5, BOOL01);
```

```

constraint int_lin_le_reif([-1, 1], [s[1], s[4]], -2, BOOL02);
constraint int_lin_le_reif([1, -1], [s[2], s[5]], -3, BOOL03);
constraint int_lin_le_reif([-1, 1], [s[2], s[5]], -6, BOOL04);
constraint int_lin_le_reif([1, -1], [s[3], s[6]], -4, BOOL05);
constraint int_lin_le_reif([-1, 1], [s[3], s[6]], -3, BOOL06);
constraint array_bool_or([BOOL01, BOOL02], true);
constraint array_bool_or([BOOL03, BOOL04], true);
constraint array_bool_or([BOOL05, BOOL06], true);

```

The `int_lin_le_reif` is the reified form of the linear constraint `int_lin_le`.

Most FlatZinc primitive constraints $p(\bar{x})$ have a reified form $p_reif(\bar{x}, b)$ which takes an additional final argument b and defines the constraint $b \leftrightarrow p(\bar{x})$. FlatZinc primitive constraints which define functional relationships, like `int_plus` and `int_minus`, do not need to support reification. Instead, the equality with the result of the function is reified.

Another important use of reification arises when we use the coercion function `bool2int` (either explicitly, or implicitly by using a Boolean expression as an integer expression). Flattening creates a Boolean variable to hold the value of the Boolean expression argument, as well as an integer variable (restricted to `0..1`) to hold this value.

Consider the magic series problem of Listing 2.2.12.

```

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))));
```

Given $n = 2$ the unrolling creates

```

constraint s[0] = bool2int(s[0] = 0) + bool2int(s[1] = 0);
constraint s[1] = bool2int(s[0] = 1) + bool2int(s[1] = 1);
```

and flattening creates

```

constraint BOOL01 <-> s[0] = 0;
constraint BOOL03 <-> s[1] = 0;
constraint BOOL05 <-> s[0] = 1;
constraint BOOL07 <-> s[1] = 1;
constraint INT02 = bool2int(BOOL01);
constraint INT04 = bool2int(BOOL03);
constraint INT06 = bool2int(BOOL05);
constraint INT08 = bool2int(BOOL07);
constraint s[0] = INT02 + INT04;
constraint s[1] = INT06 + INT08;
```

The final FlatZinc form is

```

var bool: BOOL01;
var bool: BOOL03;
var bool: BOOL05;
var bool: BOOL07;
var 0..1: INT02;
var 0..1: INT04;
var 0..1: INT06;
var 0..1: INT08;
array [1..2] of var 0..2: s;
constraint int_eq_reif(s[1], 0, BOOL01);
constraint int_eq_reif(s[2], 0, BOOL03);
constraint int_eq_reif(s[1], 1, BOOL05);
constraint int_eq_reif(s[2], 1, BOOL07);
constraint bool2int(BOOL01, INT02);
constraint bool2int(BOOL03, INT04);
constraint bool2int(BOOL05, INT06);
constraint bool2int(BOOL07, INT08);
constraint int_lin_eq([-1, -1, 1], [INT02, INT04, s[1]], 0);
constraint int_lin_eq([-1, -1, 1], [INT06, INT08, s[2]], 0);
solve satisfy;

```

2.8.6 Predicates

An important factor in the support for MiniZinc by many different solvers is that global constraints (and indeed FlatZinc constraints) can be specialized for the particular solver.

Each solver will either specify a predicate without a definition, or with a definition. For example a solver that has a builtin global `alldifferent` predicate, will include the definition

```
predicate alldifferent(array[int] of var int:x);
```

in its globals library, while a solver using the default decomposition will have the definition

```
predicate alldifferent(array[int] of var int:x) =
  forall(i,j in index_set(x) where i < j)(x[i] != x[j]);
```

Predicate calls $p(\bar{t})$ are flattened by first constructing variables v_i for each argument terms t_i . If the predicate has no definition we simply use a call to the predicate with the constructed arguments: $p(\bar{v})$. If the predicate has a definition $p(\bar{x}) = \phi(\bar{x})$ then we replace the predicate call $p(\bar{t})$ with the body of the predicate with the formal arguments replaced by the argument variables, that is $\phi(\bar{v})$. Note that if a predicate call $p(\bar{t})$ appears in a reified position and it has no definition, we check for the existence of a reified version of the predicate $p_reif(\bar{x}, b)$ in which case we use that.

Consider the `alldifferent` constraint in the SEND-MORE-MONEY example of Listing 2.2.4

```
constraint alldifferent([S,E,N,D,M,O,R,Y]);
```

If the solver has a builtin `alldifferent` we simply construct a new variable for the argument, and replace it in the call.

```
array[1..8] of var 0..9: v = [S,E,N,D,M,O,R,Y];
constraint alldifferent(v);
```

Notice that bounds analysis attempts to find tight bounds on the new array variable. The reason for constructing the array argument is if we use the same array twice the FlatZinc solver does not have to construct it twice. In this case since it is not used twice a later stage of the translation will replace `v` by its definition.

What if the solver uses the default definition of `alldifferent`? Then the variable `v` is defined as usual, and the predicate call is replaced by a renamed copy where `v` replaces the formal argument `x`. The resulting code is

```
array[1..8] of var 0..9: v = [S,E,N,D,M,O,R,Y];
constraint forall(i,j in 1..8 where i < j)(v[i] != v[j])
```

which we examined in *Unrolling Expressions* (page 121).

Consider the following constraint, where `alldifferent` appears in a reified position.

```
constraint alldifferent([A,B,C]) \vee alldifferent([B,C,D]);
```

If the solver has a reified form of `alldifferent` this will be flattend to

```
constraint alldifferent_reif([A,B,C],BOOL01);
constraint alldifferent_reif([B,C,D],BOOL02);
constraint array_bool_or([BOOL01,BOOL02],true);
```

Using the default decomposition, the predicate replacement will first create

```
array[1..3] of var int: v1 = [A,B,C];
array[1..3] of var int: v2 = [B,C,D];
constraint forall(i,j in 1..3 where i<j)(v1[i] != v1[j]) \vee
    forall(i,j in 1..3 where i<j)(v2[i] != v2[j]);
```

which will eventually be flattened to the FlatZinc form

```
constraint int_neq_reif(A,B,BOOL01);
constraint int_neq_reif(A,C,BOOL02);
constraint int_neq_reif(B,C,BOOL03);
constraint array_bool_and([BOOL01,BOOL02,BOOL03],BOOL04);
constraint int_neq_reif(B,D,BOOL05);
constraint int_neq_reif(C,D,BOOL06);
constraint array_bool_and([BOOL03,BOOL05,BOOL06],BOOL07);
constraint array_bool_or([BOOL04,BOOL07],true);
```

Note how common subexpression elimination reuses the reified inequality $B \neq C$ (although there is a better translation which lifts the common constraint to the top level conjunction).

2.8.7 Let Expressions

Let expressions are a powerful facility of MiniZinc to introduce new variables. This is useful for creating common sub expressions, and for defining local variables for predicates. During flattening let expressions are translated to variable and constraint declarations. The relational semantics of MiniZinc means that these constraints must appear as if conjoined in the first enclosing Boolean expression.

A key feature of let expressions is that each time they are used they create new variables.

Consider the flattening of the code

```
constraint even(u) \/ even(v);
predicate even(var int: x) =
    let { var int: y } in x = 2 * y;
```

First the predicate calls are replaced by their definition.

```
constraint (let { var int: y} in u = 2 * y) \/
    (let { var int: y} in v = 2 * y);
```

Next let variables are renamed apart

```
constraint (let { var int: y1} in u = 2 * y1) \/
    (let { var int: y2} in v = 2 * y2);
```

Finally variable declarations are extracted to the top level

```
var int: y1;
var int: y2;
constraint u = 2 * y1 \/ v = 2 * y2;
```

Once the let expression is removed we can flatten as usual.

Remember that let expressions can define values for newly introduced variables (and indeed must do so for parameters). These implicitly define constraints that must also be flattened.

Consider the complex objective function for wedding seating problem of Listing 2.3.10.

```
solve maximize sum(h in Hatreds)(
    let { var Seats: p1 = pos[h1[h]],
          var Seats: p2 = pos[h2[h]],
          var 0..1: same = bool2int(p1 <= 6 <-> p2 <= 6) } in
    same * abs(p1 - p2) + (1-same) * (abs(13 - p1 - p2) + 1));
```

For conciseness we assume only the first two Hatreds, so

```
set of int: Hatreds = 1..2;
array[Hatreds] of Guests: h1 = [groom, carol];
array[Hatreds] of Guests: h2 = [clara, bestman];
```

The first step of flattening is to unroll the `sum` expression, giving (we keep the guest names and parameter Seats for clarity only, in reality they would be replaced by their definition):

```
solve maximize
  (let { var Seats: p1 = pos[groom],
         var Seats: p2 = pos[clara],
         var 0..1: same = bool2int(p1 <= 6 <-> p2 <= 6) } in
    same * abs(p1 - p2) + (1-same) * (abs(13 - p1 - p2) + 1))
  +
  (let { var Seats: p1 = pos[carol],
         var Seats: p2 = pos[bestman],
         var 0..1: same = bool2int(p1 <= 6 <-> p2 <= 6) } in
    same * abs(p1 - p2) + (1-same) * (abs(13 - p1 - p2) + 1));
```

Next each new variable in a let expression is renamed to be distinct

```
solve maximize
  (let { var Seats: p11 = pos[groom],
         var Seats: p21 = pos[clara],
         var 0..1: same1 = bool2int(p11 <= 6 <-> p21 <= 6) } in
    same1 * abs(p11 - p21) + (1-same1) * (abs(13 - p11 - p21) + 1))
  +
  (let { var Seats: p12 = pos[carol],
         var Seats: p22 = pos[bestman],
         var 0..1: same2 = bool2int(p12 <= 6 <-> p22 <= 6) } in
    same2 * abs(p12 - p22) + (1-same2) * (abs(13 - p12 - p22) + 1));
```

Variables in the let expression are extracted to the top level and defining constraints are extracted to the correct level (which in this case is also the top level).

```
var Seats: p11;
var Seats: p21;
var 0..1: same1;
constraint p12 = pos[clara];
constraint p11 = pos[groom];
constraint same1 = bool2int(p11 <= 6 <-> p21 <= 6);
var Seats p12;
var Seats p22;
var 0..1: same2;
constraint p12 = pos[carol];
constraint p22 = pos[bestman];
constraint same2 = bool2int(p12 <= 6 <-> p22 <= 6) } in
solve maximize
  same1 * abs(p11 - p21) + (1-same1) * (abs(13 - p11 - p21) + 1))
```

```
+  
same2 * abs(p12 - p22) + (1-same2) * (abs(13 - p12 - p22) + 1));
```

Now we have constructed equivalent MiniZinc code without the use of let expressions and the flattening can proceed as usual.

As an illustration of let expressions that do not appear at the top level consider the following model

```
var 0..9: x;  
constraint x >= 1 -> let { var 2..9: y = x - 1 } in  
    y + (let { var int: z = x * y } in z * z) < 14;
```

We extract the variable definitions to the top level and the constraints to the first enclosing Boolean context, which here is the right hand side of the implication.

```
var 0..9: x;  
var 2..9: y;  
var int: z;  
constraint x >= 1 -> (y = x - 1 /\ z = x * y /\ y + z * z < 14);
```

Note that if we know that the equation defining a variable definition cannot fail we can extract it to the top level. This will usually make solving substantially faster.

For the example above the constraint $y = x - 1$ can fail since the domain of y is not big enough for all possible values of $x - 1$. But the constraint $z = x * y$ cannot (indeed bounds analysis will give z bounds big enough to hold all possible values of $x * y$). A better flattening will give

```
var 0..9: x;  
var 2..9: y;  
var int: z;  
constraint z = x * y;  
constraint x >= 1 -> (y = x - 1 /\ y + z * z < 14);
```

Currently the MiniZinc compiler does this by always defining the declared bounds of an introduced variable to be big enough for its defining equation to always hold and then adding bounds constraints in the correct context for the let expression. On the example above this results in

```
var 0..9: x;  
var -1..8: y;  
var -9..72: z;  
constraint y = x - 1;  
constraint z = x * y;  
constraint x >= 1 -> (y >= 2 /\ y + z * z < 14);
```

This translation leads to more efficient solving since the possibly complex calculation of the let variable is not reified.

Another reason for this approach is that it also works when introduced variables appear in negative contexts (as long as they have a definition). Consider the following example similar to the

previous one

```
var 0..9: x;
constraint (let { var 2..9: y = x - 1 } in
            y + (let { var int: z = x * y } in z * z) > 14) -> x >= 5;
```

The let expressions appear in a negated context, but each introduced variable is defined. The flattened code is

```
var 0..9: x;
var -1..8: y;
var -9..72: z;
constraint y = x - 1;
constraint z = x * y;
constraint (y >= 2 /\ y + z * z > 14) -> x >= 5;
```

Note the analog to the simple approach to let elimination does not give a correct translation:

```
var 0..9: x;
var 2..9: y;
var int: z;
constraint (y = x - 1 /\ z = x * y /\ y + z * z > 14) -> x >= 5;
```

gives answers for all possible values of x , whereas the original constraint removes the possibility that $x = 4$.

The treatment of *constraint items* in let expressions is analogous to defined variables. One can think of a constraint as equivalent to defining a new Boolean variable. The definitions of the new Boolean variables are extracted to the top level, and the Boolean remains in the correct context.

```
constraint z > 1 -> let { var int: y,
                           constraint (x >= 0) -> y = x,
                           constraint (x < 0) -> y = -x
                         } in y * (y - 2) >= z;
```

is treated like

```
constraint z > 1 -> let { var int: y,
                           var bool: b1 = ((x >= 0) -> y = x),
                           var bool: b2 = ((x < 0) -> y = -x),
                           constraint b1 /\ b2
                         } in y * (y - 2) >= z;
```

and flattens to

```
constraint b1 = ((x >= 0) -> y = x);
constraint b2 = ((x < 0) -> y = -x);
constraint z > 1 -> (b1 /\ b2 /\ y * (y - 2) >= z);
```

Part 3

User Manual

CHAPTER 3.1

The MiniZinc Command Line Tool

The core of the MiniZinc constraint modelling system is the `minizinc` tool. You can use it directly from the command line, through the MiniZinc IDE, or through a programmatic interface (API). This chapter summarises the options accepted by the tool, and explains how it interacts with target solvers.

3.1.1 Basic Usage

The `minizinc` tool performs three basic functions: it *compiles* a MiniZinc model (plus instance data), it *runs* an external solver, and it *translates solver output* into the form specified in the model. Most users would use all three functions at the same time. For example, let us assume that we want to solve the following simple problem, given as two files (`model.mzn` and `data.dzn`):

```
int: n;
array[1..n] of var 1..2*n: x;
include "alldifferent.mzn";
constraint alldifferent(x);
solve maximize sum(x);
output ["The resulting values are \$(x).\n"];
```

```
n = 5;
```

To run the model file `model.mzn` with data file `data.dzn` using the Gecode solver, you can use the following command line:

```
$ minizinc --solver Gecode model.mzn data.dzn
```

This would result in the output

```
The resulting values are [10, 9, 8, 7, 6].
```

```
-----  
=====
```

However, each of the three functions can also be accessed individually. For example, to compile the same model and data, use the `-c` option:

```
$ minizinc -c --solver Gecode model.mzn data.dzn
```

This will result in two new files, `model.fzn` and `model.ozn`, being output in the same directory as `model.mzn`. You could then run a target solver directly on the `model.fzn` file, or use `minizinc`:

```
$ minizinc --solver Gecode model.fzn
```

You will see that the solver produces output in a standardised form, but not the output prescribed by the `output` item in the model:

```
x = array1d(1..5 ,[10, 9, 8, 7, 6]);
```

```
-----  
=====
```

The translation from this output to the form specified in the model is encoded in the `model.ozn` file. You can use `minizinc` to execute the `.ozn` file. In this mode, it reads a stream of solutions from standard input, so we need to pipe the solver output into `minizinc`:

```
$ minizinc --solver Gecode model.fzn | minizinc --ozn-file model.ozn
```

These are the most basic command line options that you need in order to compile and run models and translate their output. The next section lists all available command line options in detail. Section 4.3.5 explains how new solvers can be added to the system.

3.1.2 Adding Solvers

Solvers that support MiniZinc typically consist of two parts: a solver *executable*, which can be run on the FlatZinc output of the MiniZinc compiler, and a *solver library*, which consists of a set of MiniZinc files that define the constraints that the solver supports natively. This section deals with making existing solvers available to the MiniZinc tool chain. For information on how to add FlatZinc support to a solver, refer to Section 4.3.

3.1.2.1 Configuration files

In order for MiniZinc to be able to find both the solver library and the executable, the solver needs to be described in a *solver configuration file* (see Section 4.3.5 for details). If the solver you want to install comes with a configuration file (which has the file extension `.msc` for MiniZinc Solver Configuration), it has to be in one of the following locations:

- In the `minizinc/solvers/` directory of the MiniZinc installation. If you install MiniZinc from the binary distribution, this directory can be found at `/usr/share/minizinc/solvers` on Linux systems, inside the MiniZincIDE application on macOS system, and in the `Program Files\MiniZinc IDE (bundled)` folder on Windows.
- In the directory `$HOME/.minizinc/solvers` on Linux and macOS systems, and the Application Data directory on Windows systems.
- In any directory listed on the `MZN_SOLVER_PATH` environment variable (directories are separated by : on Linux and macOS, and by ; on Windows systems).
- In any directory listed in the `mzn_solver_path` option of the global or user-specific configuration file (see [Section 3.1.4](#))
- Alternatively, you can use the MiniZinc IDE to create solver configuration files, see [Section 3.2.5.2](#) for details.

After adding a solver, it will be listed in the output of the `minizinc --solvers` command.

3.1.2.2 Configuration for MIP solvers

Some solvers require additional configuration flags before they can be used. For example, the binary bundle of MiniZinc comes with interfaces to the CPLEX and Gurobi Mixed Integer Programming solvers. However, due to licensing restrictions, the solvers themselves are not part of the bundled release. Depending on where CPLEX or Gurobi is installed on your system, MiniZinc may be able to find the solvers automatically, or it may require an additional option to point it to the shared library.

In case the libraries cannot be found automatically, you can use one of the following:

- CPLEX: Specify the location of the shared library using the `--cplex-dll` command line option. On Windows, the library is called `cplexXXXX.dll` and typically found in same directory as the `cplex` executable. On Linux it is `libcplexXXX.so`, and on macOS `libcplexXXXX.jnilib`, where XXX and XXXX stand for the version number.
- Gurobi: The command line option for Gurobi is `--gurobi-dll`. On Windows, the library is called `gurobiXX.dll` (in the same directory as the `gurobi` executable), and on Linux and macOS is it `libgurobiXX.so` (in the `lib` directory of your Gurobi installation).
- You can define these paths as defaults in your user configuration file, see [Section 3.1.4](#).

3.1.3 Options

You can get a list of all the options supported by the `minizinc` tool using the `--help` flag.

3.1.3.1 General options

These options control the general behaviour of the `minizinc` tool.

- `--help, -h`
Print a help message.
- `--version`
Print version information.

```
--solvers
    Print list of available solvers.

--solver <id>, --solver <solver configuration file>.msc
    Select solver to use. The first form of the command selects one of the solvers known to MiniZinc (that appear in the list of the --solvers command). You can select a solver by name, id, or tag, and add a specific version. For example, to select a mixed-integer programming solver, identified by the mip tag, you can use --solver mip. To select a specific version of Gurobi (in case you have two versions installed), use --solver Gurobi@7.5.2. Instead of the name you can also use the solver's identifier, e.g. --solver org.gecode.gecode.

The second form of the command selects the solver from the given configuration file (see Section 4.3.5).

--help <id>
    Print help for a particular solver. The scheme for selecting a solver is the same as for the --solver option.

-v, -l, --verbose
    Print progress/log statements (for both compilation and solver). Note that some solvers may log to stdout.

--verbose-compilation
    Print progress/log statements for compilation only.

-s, --statistics
    Print statistics (for both compilation and solving).

--compiler-statistics
    Print statistics for compilation.

-c, --compile
    Compile only (do not run solver).

--config-dirs
    Output configuration directories.

--solvers-json
    Print configurations of available solvers as a JSON array.
```

3.1.3.2 Solving options

Each solver may support specific command line options for controlling its behaviour. These can be queried using the --help <id> flag, where <id> is the name or identifier of a particular solver. Most solvers will support some or all of the following options.

```
-a, --all-solutions
    Report all solutions in the case of satisfaction problems, or print intermediate solutions of increasing quality in the case of optimisation problems.

-n <i>, --num-solutions <i>
    Stop after reporting i solutions (only used with satisfaction problems).

-f, --free-search
    Instructs the solver to conduct a “free search”, i.e., ignore any search annotations. The solver is not required to ignore the annotations, but it is allowed to do so.
```

```
--solver-statistics
    Print statistics during and/or after the search for solutions.

--verbose-solving
    Print log messages (verbose solving) to the standard error stream.

-p <i>, --parallel <i>
    Run with i parallel threads (for multi-threaded solvers).

-r <i>, --random-seed <i>
    Use i as the random seed (for any random number generators the solver may be using).
```

3.1.3.3 Flattener input options

These options control aspects of the MiniZinc compiler.

```
--ignore-stdlib
    Ignore the standard libraries stdlib.mzn and builtins.mzn

--instance-check-only
    Check the model instance (including data) for errors, but do not convert to FlatZinc.

-e, --model-check-only
    Check the model (without requiring data) for errors, but do not convert to FlatZinc.

--model-interface-only
    Only extract parameters and output variables.

--model-types-only
    Only output variable (enum) type information.

--no-optimize
    Do not optimize the FlatZinc

-d <file>, --data <file>
    File named <file> contains data used by the model.

-D <data>, --cmdline-data <data>
    Include the given data assignment in the model.

--stdlib-dir <dir>
    Path to MiniZinc standard library directory

-G --globals-dir --mzn-globals-dir <dir>
    Search for included globals in <stdlib>/<dir>.

- --input-from-stdin
    Read problem from standard input

-I --search-dir
    Additionally search for included files in <dir>.

-D "fMIPdomains=false"
    No domain unification for MIP

--MIPDMaxIntvEE <n>
    Max integer domain subinterval length to enforce equality encoding, default 0
```

```
--MIPDMaxDensEE <n>
    Max domain cardinality to N subintervals ratio to enforce equality encoding, default 0,
    either condition triggers

--only-range-domains
    When no MIPdomains: all domains contiguous, holes replaced by inequalities

--allow-multiple-assignments
    Allow multiple assignments to the same variable (e.g. in dzn)

--compile-solution-checker <file>.mzc.mzn
    Compile solution checker model.
```

Flattener two-pass options

Two-pass compilation means that the MiniZinc compiler will first compile the model in order to collect some global information about it, which it can then use in a second pass to improve the resulting FlatZinc. For some combinations of model and target solver, this can lead to substantial improvements in solving time. However, the additional time spent on the first compilation pass does not always pay off.

```
--two-pass
    Flatten twice to make better flattening decisions for the target

--use-gecode
    Perform root-node-propagation with Gecode (adds --two-pass)

--shave
    Probe bounds of all variables at the root node (adds --use-gecode)

--sac
    Probe values of all variables at the root node (adds --use-gecode)

--pre-passes <n>
    Number of times to apply shave/sac pass (0 = fixed-point, 1 = default)

-0<n>
    Two-pass optimisation levels:
        -O0: Disable optimize (-no-optimize) -O1: Single pass (default) -O2: Same as: --two-pass
        -O3: Same as: --use-gecode -O4: Same as: --shave -O5: Same as: --sac
```

Flattener output options

These options control how the MiniZinc compiler produces the resulting FlatZinc output. If you run the solver directly through the `minizinc` command or the MiniZinc IDE, you do not need to use any of these options.

```
--no-output-ozn, -0-
    Do not output ozn file

--output-base <name>
    Base name for output files

--fzn <file>, --output-fzn-to-file <file>
    Filename for generated FlatZinc output
```

```

-0, --ozn, --output-ozn-to-file <file>
    Filename for model output specification (-O- for none)

--keep-paths
    Don't remove path annotations from FlatZinc

--output-paths
    Output a symbol table (.paths file)

--output-paths-to-file <file>
    Output a symbol table (.paths file) to <file>

--output-to-stdout, --output-fzn-to-stdout
    Print generated FlatZinc to standard output

--output-ozn-to-stdout
    Print model output specification to standard output

--output-paths-to-stdout
    Output symbol table to standard output

--output-mode <item|dzn|json>
    Create output according to output item (default), or output compatible with dzn or json
    format

--output-objective
    Print value of objective function in dzn or json output

-Werror
    Turn warnings into errors

```

3.1.3.4 Solution output options

These options control how solutions are output. Some of these options only apply if `minizinc` is used to translate a stream of solutions coming from a solver into readable output (using a `.ozn` file generated by the compiler).

```

--ozn-file <file>
    Read output specification from ozn file.

-o <file>, --output-to-file <file>
    Filename for generated output.

-i <n>, --ignore-lines <n>, --ignore-leading-lines <n>
    Ignore the first <n> lines in the FlatZinc solution stream.

--soln-sep <s>, --soln-separator <s>, --solution-separator <s>
    Specify the string printed after each solution (as a separate line). The default is to use the
    same as FlatZinc, “-----”.

--soln-comma <s>, --solution-comma <s>
    Specify the string used to separate solutions. The default is the empty string.

--unsat-msg (--unsatisfiable-msg)
    Specify status message for unsatisfiable problems (default: "=====UNSATISFIABLE=====") 

--unbounded-msg
    Specify status message for unbounded problems (default: "=====UNBOUNDED=====")

```

```
--unsatorunbnd-msg
    Specify status message for unsatisfiable or unbounded problems (default:
    "=====UNSATOrUNBOUNDED=====")

--unknown-msg
    Specify status message if search finished before determining status (default:
    "=====UNKNOWN=====")

--error-msg
    Specify status message if search resulted in an error (default: "=====ERROR=====") 

--search-complete-msg <msg>
    Specify status message if when search exhausted the entire search space (default:
    "=====") 

--non-unique
    Allow duplicate solutions.

-c, --canonicalize
    Canonicalize the output solution stream (i.e., buffer and sort).

--output-non-canonical <file>
    Non-buffered solution output file in case of canonicalization.

--output-raw <file>
    File to dump the solver's raw output (not for hard-linked solvers)

--no-output-comments
    Do not print comments in the FlatZinc solution stream.

--output-time
    Print timing information in the FlatZinc solution stream.

--no-flush-output
    Don't flush output stream after every line.
```

3.1.4 User Configuration Files

The `minizinc` tool reads a system-wide and a user-specific configuration file to determine default paths, solvers and solver options. The files are called `Preferences.json`, and you can find out the locations for your platform using the option `--config-dirs`:

```
$ minizinc --config-dirs
{
  "globalConfigFile" : "/Applications/MiniZincIDE.app/Contents/Resources/
  ↪share/minizinc/Preferences.json",
  "userConfigFile" : "/Users/Joe/.minizinc/Preferences.json",
  "userSolverConfigDir" : "/Users/Joe/.minizinc/solvers",
  "mznStdlibDir" : "/Applications/MiniZincIDE.app/Contents/Resources/share/
  ↪minizinc"
}
```

The configuration files are simple JSON files that can contain the following configuration options:

- `mzn_solver_path` (list of strings): Additional directories to scan for solver configuration files.
- `mzn_lib_dir` (string): Location of the MiniZinc standard library.
- `tagDefaults` (list of lists of strings): Each entry maps a tag to the default solver for that tag. For example, `[["cp", "org.chuffed.chuffed"], ["mip", "org.minizinc.gurobi"]]` would declare that whenever a solver with tag "cp" is requested, Chuffed should be used, and for the "mip" tag, Gurobi is the default. The empty tag ("") can

be used to define the system-wide default solver (i.e., the solver that is chosen when running `minizinc` without the `--solver` argument). - `solverDefaults` (list of lists of strings): Each entry consists of a list of three strings: a solver identifier, a command line option, and a value for that command line option. For example, `[["org.minizinc.gurobi", "--gurobi-dll", "/Library/gurobi752/mac64/lib/libgurobi75.so"]]` would specify the Gurobi shared library to use (on a macOS system with Gurobi 7.5.2). For command line options that don't take a value, you have to specify an empty string, e.g. `[["org.minizinc.gurobi", "--uniform-search", ""]]`.

Here is a sample configuration file:

```
{
  "mzn_solver_path": ["/usr/share/choco"],
  "tagDefaults": [[{"cp": "org.choco-solver.choco"}, {"mip": "org.minizinc.cplex"}], [{"": "org.gecode.gecode"}]],
  "solverDefaults": [{"cplex": "org.minizinc.cplex", "cplex-dll": "/opt/CPLEX_Studio128/cplex/bin/x86-64_sles10_4.1/libcplex128.so"}]
}
```

Configuration values in the user-specific configuration file override the global values, except for solver default arguments, which are only overridden if the name of the option is the same, and otherwise get added to the command line.

Note: Due to current limitations in MiniZinc's JSON parser, we use lists of strings rather than objects for the default mappings. This may change in a future release, but the current syntax will remain valid. The location of the global configuration is currently the `share/minizinc` directory of the MiniZinc installation. This may change in future versions to be more consistent with file system standards (e.g., to use `/etc` on Linux and `/Library/Preferences` on macOS).

CHAPTER 3.2

The MiniZinc IDE

The MiniZinc IDE lets you edit and run MiniZinc models. It requires a working installation of the MiniZinc tool chain (which is included when you download the *bundled version* of the IDE). For installation instructions, see [Section 1.2](#). This document assumes that you have installed the IDE from the bundled binary distribution, which means that it will already be configured correctly when you start it for the first time. The configuration options are described in [Section 3.2.5](#).

[Section 1.3](#) contains an introduction to the absolute basics of working with the IDE. This document goes into a bit more detail.

3.2.1 Editing files

The basic editor provides the usual functionality of a simple text editor. You can edit MiniZinc models (file extension `.mzn`) and data files (`.dzn`). When you first open the MiniZinc IDE, you are presented with an empty *Playground* editor, which lets you quickly try out simple MiniZinc models (it does not have to be saved to a file before running a solver on it).

Each file will be opened in a separate tab. To switch between files, click on the tab, select the file from the *Window* menu, or use the *Previous tab/Next tab* options from the *View* menu.

When saving a file, make sure that you select the correct file type (represented by the file extension). Models should be saved as `.mzn` files, while data files should have a `.dzn` extension. This is important because the IDE will only let you invoke the solver on model files.

3.2.1.1 Editing functions

The *Edit* menu contains the usual functions for editing text files, such as undo/redo, copy/cut/paste, and find/replace. It also allows you to jump to a particular line number (*Go to line*), and to shift the currently selected text (or the current line if nothing is selected) right or left by two spaces. The *(Un)comment* option will turn the current selection (or current line) into comments, or remove the comment symbols if it is already commented.

3.2.1.2 Fonts and dark mode

You can select the font and font size in the *View* menu. We recommend to use a fixed-width font (the IDE should pick such a font by default).

The *View* menu also lets you activate “dark mode”, which switches the colour scheme to a dark background.

3.2.2 Configuring and Running a Solver

The MiniZinc IDE automatically detects which solvers are available to MiniZinc. You can select the solver to use from the solver selection drop-down menu next to the *Run* icon in the tool bar:



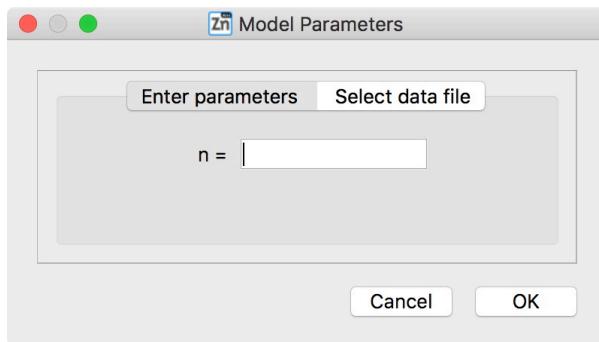
3.2.2.1 Running a model

MiniZinc models can be compiled and run by clicking the *Run* icon, selecting *Run* from the *MiniZinc* menu, or using the keyboard shortcut **Ctrl+R** (**Cmd+R** on macOS). The IDE will use the currently selected solver for compiling and running the model.

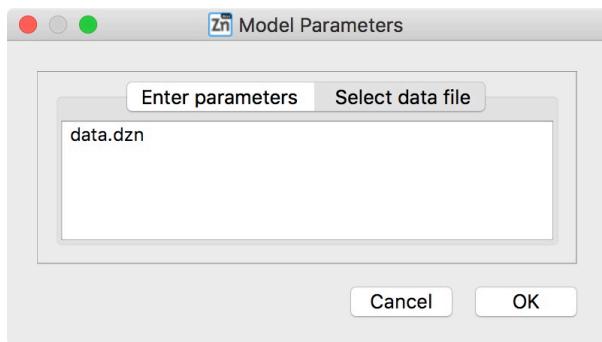
Running a model will open the *Output* window, usually located at the bottom of the IDE’s main window. MiniZinc displays progress messages as well as any output received from the solver there. If compilation resulted in an error message, clicking on the error will jump to the corresponding location in the model or data file.

The current run can be aborted by clicking the *Stop* icon, selecting *Stop* from the *MiniZinc* menu, or using the keyboard shortcut **Ctrl+E** (**Cmd+E** on macOS).

If the selected model requires some parameters to be set before it can be run, the MiniZinc IDE will open a parameter dialog. It has two tabs. The left tab lets you enter the parameters manually:



The second tab lets you select one or several of the data files that are currently open:



3.2.2.2 Solver configurations

Selecting one of the built-in solvers from the drop-down menu activates its default configuration. In order to change the solver’s behaviour, open the solver configuration editor by clicking on the icon in the tool bar, selecting *Show configuration editor* from the *MiniZinc/Solver configurations* menu, or using the keyboard shortcut `Ctrl+Shift+C` (`Cmd+Shift+C` on macOS).

Fig. 3.2.1 shows the configuration window. The first section (marked with a 1 in a red circle) contains a drop-down menu to select the *solver configuration*. In this case, a built-in configuration for the OSI-CBC solver was selected. You can make this configuration the default (the MiniZinc IDE will remember this setting), you can reset all values to the defaults, and you can make a clone of the configuration. Cloning a configuration is useful if you want to be able to quickly switch between different sets of options.

Note that any changes to the built-in configurations will be lost when you close the IDE. Any changes to a cloned configuration are saved as part of the *project* (see Section 3.2.3).

The *Solving* section below contains a number of general options. First of all, it shows the concrete solver used in this configuration. Below that, you can set a time limit, after which the execution will be stopped. The built-in configurations all use the “default behaviour” (marked with a 2), which is to print all intermediate solutions for optimisation problems, and stop after the first found solution for satisfaction problems. To change this, you can select *User-defined behavior* instead (marked with a 3).

The next section, *Compiler options* (marked with a 4), controls different aspects of the compilation from MiniZinc to FlatZinc for the selected solver. The first two checkboxes control verbosity and statistics output of the compiler. The drop-down below controls the optimisation level of the compiler, i.e., how much effort it should spend on creating the best possible FlatZinc representation for the given model. The two input fields below allow you to specify additional data (passed into the compilation as if it was part of a `.dzn` file) and additional command line options for the compiler.

The *Solver options* section (marked with a 5) contains configuration options for the selected solver. Only options that are supported by the solver will be available, others will be grayed out (e.g., the selected solver in Fig. 3.2.1 does not support setting a random seed, or free search).

Finally, the *Output options* section gives you control over the output behaviour. The first tick box enables you to clear the *Output* window automatically every time you run a model. The second option inserts timing information into the stream of solutions. The third check box (*Check solutions*) is described in Section 3.2.2.3 below. The *Compress solution output* option is useful for problems that produce a lot (read: thousands) of solutions, which can slow down and clutter the output window. The compression works by printing just the number of solutions

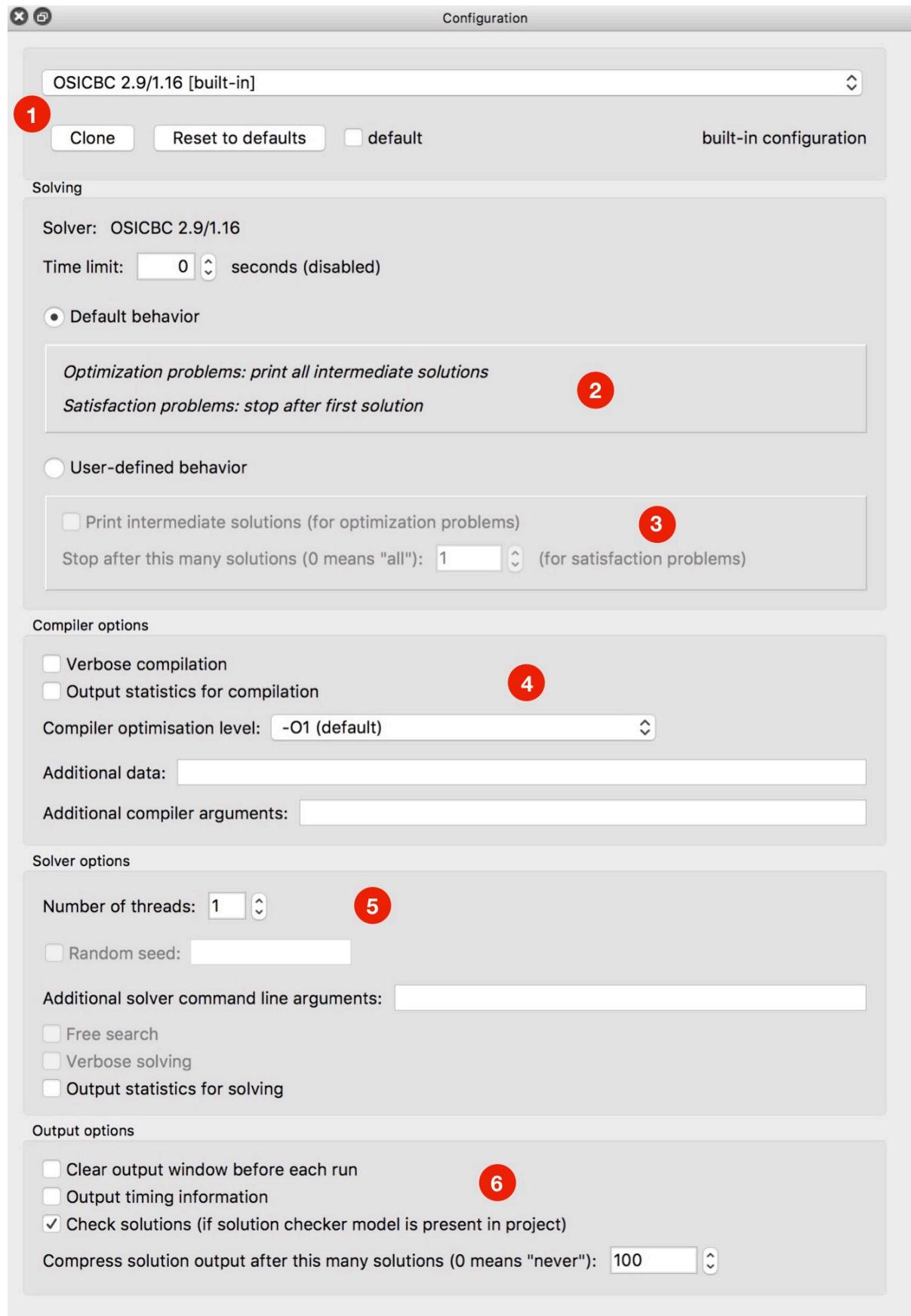


Fig. 3.2.1: The solver configuration window

rather than the solutions themselves. For example, the following model would produce 1000 solutions when run with *User-defined behavior* and solution limit set to 0:

```
var 1..1000: x;
solve satisfy;
```

When running with compression set to 100, MiniZinc will output the first 100 solutions, and then a sequence of output like this:

```
[ 100 more solutions ]
[ 200 more solutions ]
[ 400 more solutions ]
[ 199 more solutions ]
x = 1000;
-----
=====
```

The number of solutions captured by one of the ... more solutions lines is doubled each time, in order to keep the overall output low. The last solution produced by the solver will always be printed (since, in the case of optimisation problems, the last solution is the best one found).

3.2.2.3 Automatic Solution Checking

MiniZinc can automatically run the output of a model through a *solution checker*, another MiniZinc model that verifies that the solution satisfies a given set of rules. This can be useful for teaching constraint modelling, if the solution checker is given to students. Another use case is to use a simple checker while working on a more complex model, to ensure that the complex model still meets the specification.

The default behaviour of the MiniZinc IDE is to run a solution checker if one is present. For a model abc.mzn, a solution checker must be called abc.mzc or abc.mzc.mzn. If a checker is present, the *Run* icon will turn into a *Run + check* icon instead. The output of the solution checker is displayed together with the normal solution output in the *Output* window.

You can disable solution checkers by deselecting the *Check solutions* option in the solver configuration window.

3.2.2.4 Compiling a model

It can sometimes be useful to look at the FlatZinc code generated by the MiniZinc compiler for a particular model. You can use the *Compile* option from the *MiniZinc* menu to compile the model without solving. The generated FlatZinc will be opened in a new tab. You can edit and save the FlatZinc to a file, and run it directly (without recompiling).

3.2.3 Working With Projects

Each main window of the MiniZinc IDE corresponds to a *project*, a collection of files and settings that belong together. A project can be saved to and loaded from a file.

You can open a new project by selecting the *New project* option from the *File* menu, or using the **Ctrl+Shift+N** keyboard shortcut (**Cmd+Shift+N** on macOS).

All the files that belong to the current project are shown in the *Project explorer* (see Fig. 3.2.2), which can be opened using the tool bar icon, or using the *Show project explorer* option in the *View* menu. The project explorer lets you run the model in the currently active tab with any of the data files by right-clicking on a `.dzn` file and selecting *Run model with this data*. Right-clicking any file presents a number of options: opening it, removing it from the project, renaming it, running it, and adding new files to the project.

A saved project contains the following pieces of information:

- The names of all files in the project. These are stored as relative paths (relative to the project file).
- Which files were open in tabs (and in which order) at the time the project was saved.
- The active solver configuration.
- The state of any cloned solver configuration.

The following will *not* be saved as part of the project:

- The contents of the *Output* window.
- The state of the built-in solver configurations.

3.2.4 Submitting Solutions to Online Courses

The MiniZinc IDE has built-in support for submitting solutions and models to online courses, including the Coursera courses that introduce modelling in MiniZinc:

- Basic Modeling for Discrete Optimization⁷
- Advanced Modeling for Discrete Optimization⁸

The submission system is controlled by a file called `_mooc`, which is typically part of the projects that you can download for workshops and assignments. When a project contains this file, a new submission icon will appear in the tool bar, together with an option in the *MiniZinc* menu.

Clicking the icon (or selecting the menu option) opens the submission dialog (see Fig. 3.2.3). It lets you select the problems that you would like to run on your machine, after which the solutions will be sent to the online course auto-grading system. Some projects may also contain model submissions, which are not run on your machine, but are evaluated by the online auto-grader on new data that was not available to you for testing.

You will have to enter the assignment-specific login details. By clicking the *Run and submit* button, you start the solving process. When it finishes, the MiniZinc IDE will upload the solutions to the auto-grading platform.

⁷ <https://www.coursera.org/learn/basic-modeling>

⁸ <https://www.coursera.org/learn/advanced-modeling>

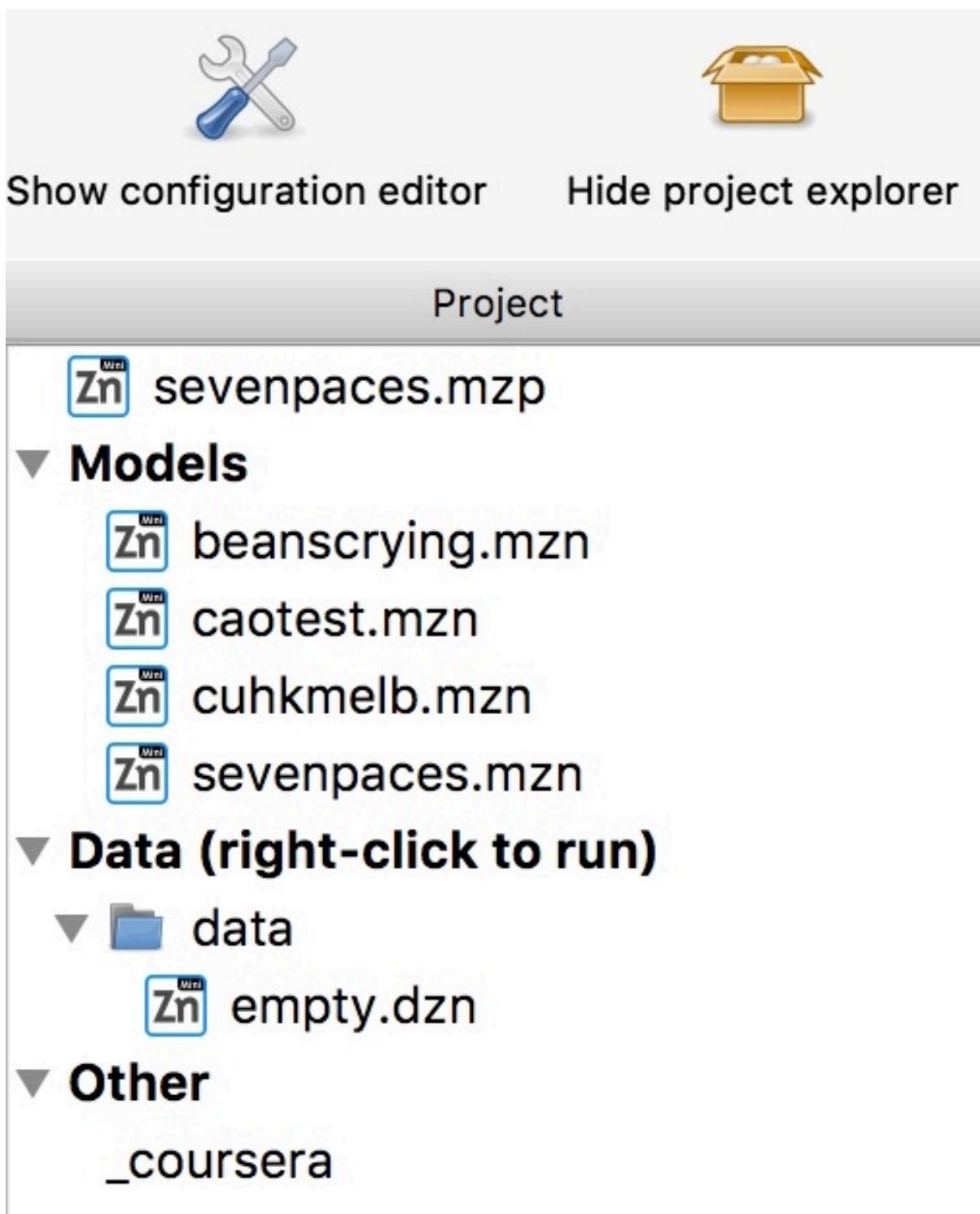


Fig. 3.2.2: Project explorer

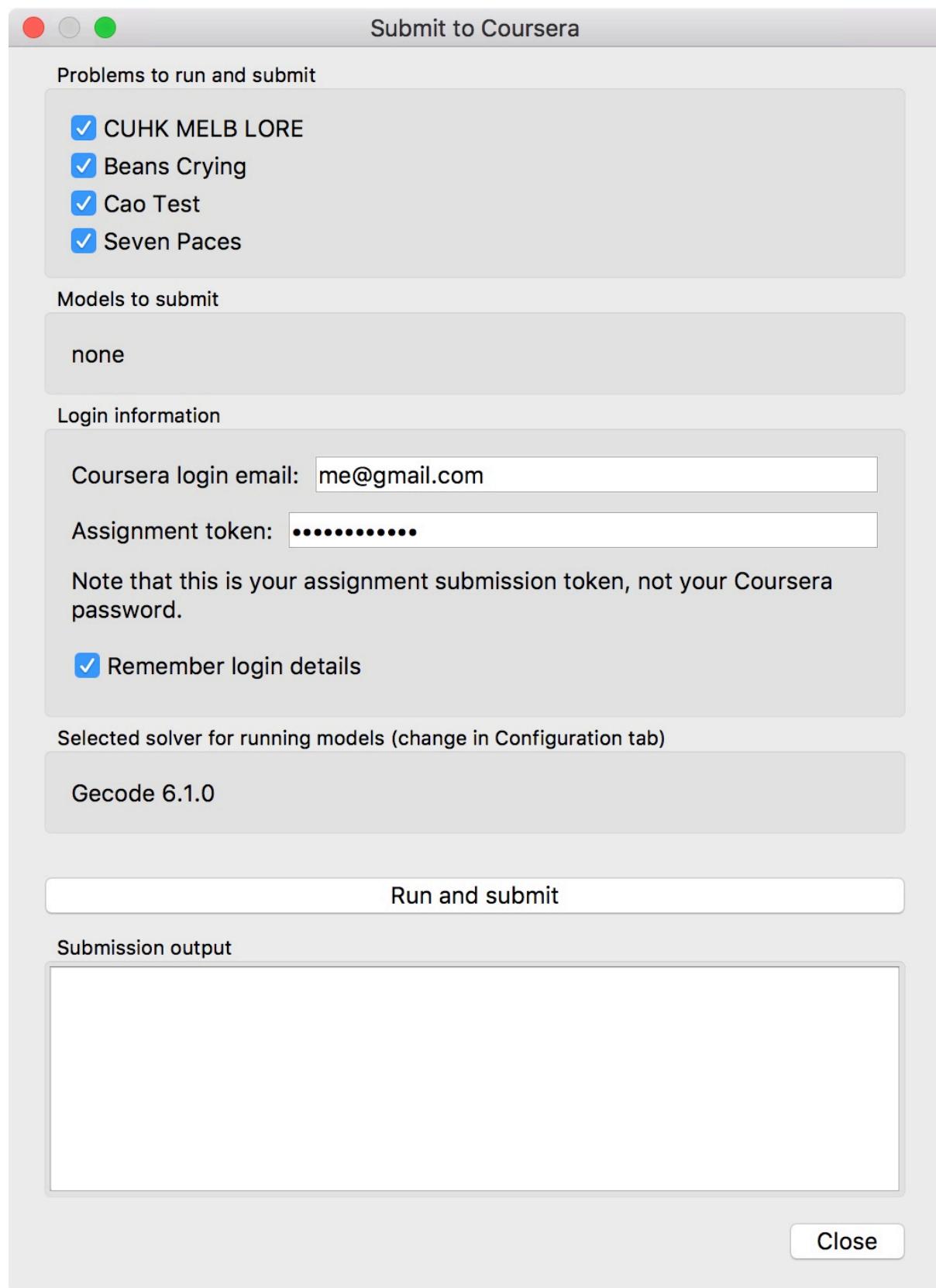


Fig. 3.2.3: Submitting to Coursera

3.2.5 Configuration Options

The MiniZinc IDE can be configured through the *Preferences* dialog in the *MiniZinc* menu (on Windows and Linux) or the *MiniZincIDE* menu (on macOS), as shown in Fig. ??.

3.2.5.1 Locating the MiniZinc installation

The most important configuration option is the path to the `minizinc` executable. In the example in Fig. ??, this field has been left empty, in which case `minizinc` is assumed to be on the standard search path (usually the `PATH` environment variable). Typically, in a bundled binary installation of MiniZinc, this field can therefore be left empty.

If you installed MiniZinc from sources, or want to switch between different versions of the compiler, you can add the path to the directory that contains the `minizinc` executable here. You can select a directory from a file dialog using the *Select* button, or enter it manually. Clicking the *Check* button will check that `minizinc` can in fact be run, and has the right version. The version of `minizinc` that was found is displayed below the path input field. Fig. ?? below shows an example where MiniZinc is located at `/home/me/minizinc-2.2.0/bin`.

You can have the MiniZinc IDE check once a day whether a new version of MiniZinc is available.

3.2.5.2 Adding Third-Party Solvers

The *Solver* section of the configuration dialog can be used to inspect the solvers that are currently available to MiniZinc, and to add new solvers to the system.

Configuring existing solvers

You can use the configuration dialog to set up defaults for the installed solvers. In the current version of the MiniZinc IDE, this is limited to configuring the CPLEX and Gurobi backends. The bundled binary version of MiniZinc comes with support for loading CPLEX and Gurobi as *plugins*, i.e., MiniZinc does not ship with the code for these solvers but can load them dynamically if they are installed.

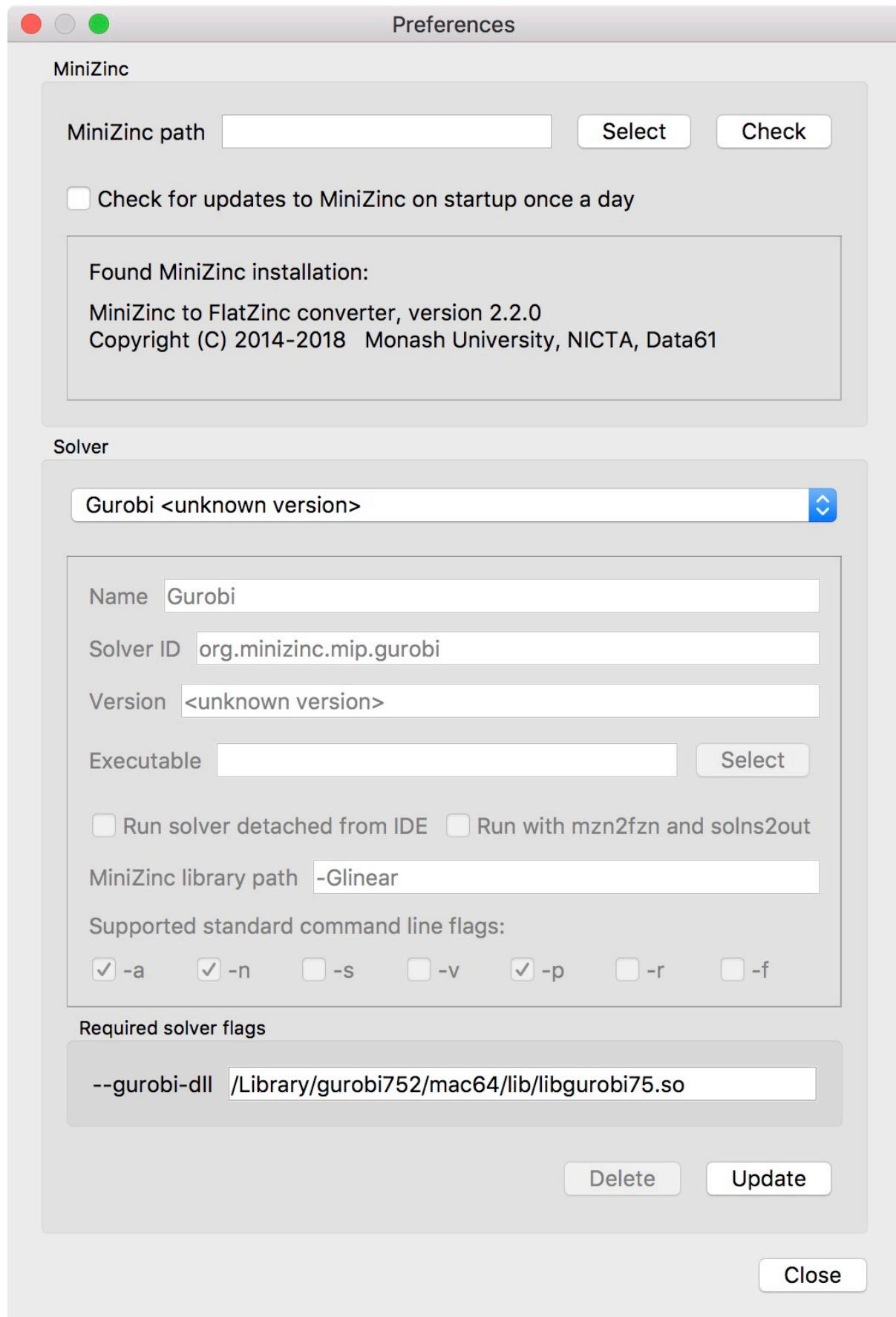
For example, Fig. ?? shows a potential configuration for Gurobi. On Windows, the library is called `gurobiXX.dll` (in the same directory as the `gurobi` executable), and on Linux and macOS is it `libgurobiXX.so` (in the `lib` directory of your Gurobi installation), where `XX` stands for the version number of Gurobi.

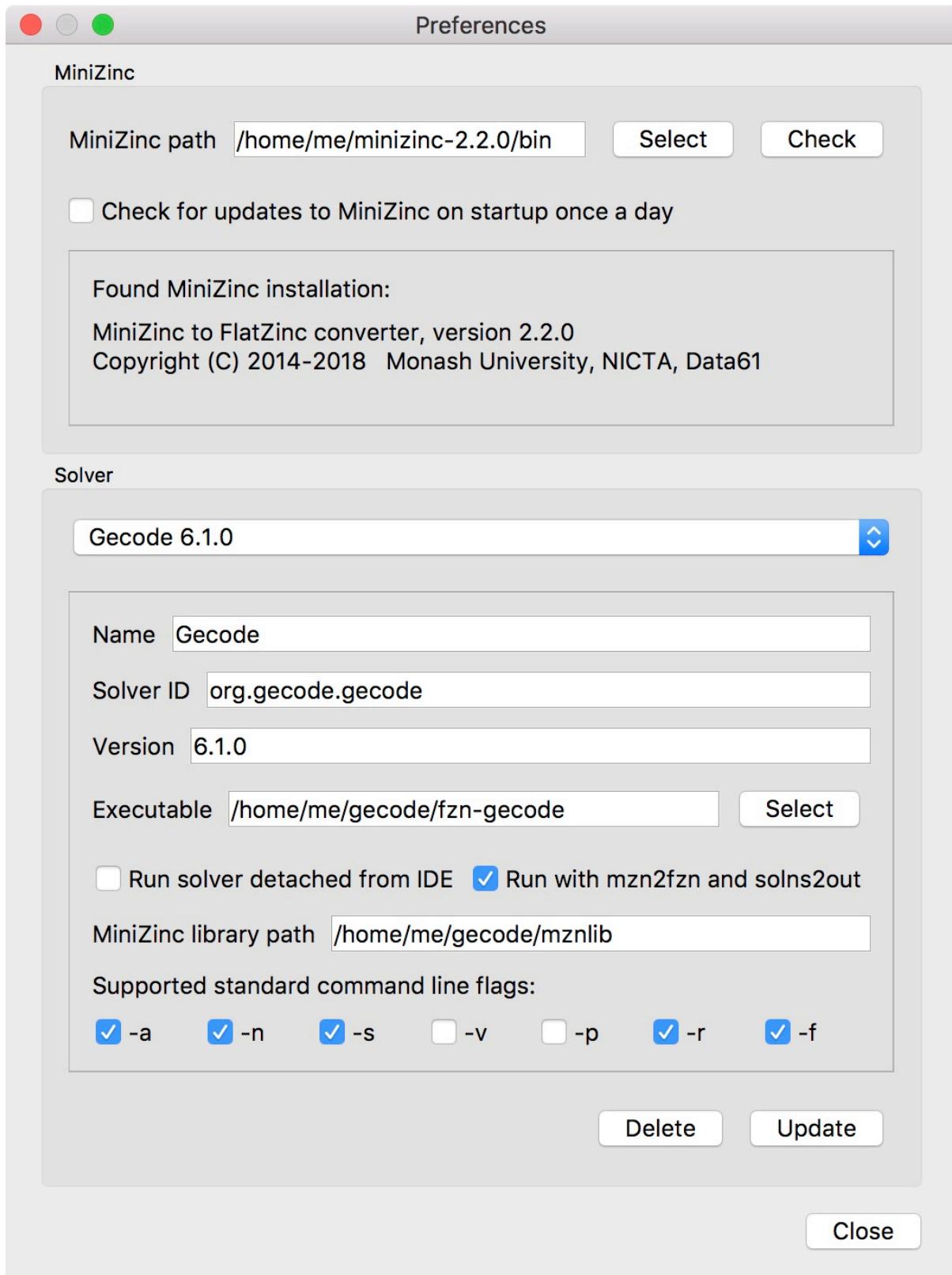
If you select the CPLEX solver, a similar option appears (`--cplex-dll`). On Windows, the CPLEX library is called `cplexXXXX.dll` and typically found in same directory as the `cplex` executable. On Linux it is `libcplexXXX.so`, and on macOS `libcplexXXXX.jnilib`, where `XXX` and `XXXX` stand for the version number of CPLEX.

Adding new solvers

The example in Fig. ?? shows a potential configuration for Gecode, which was installed in `/home/me/gecode`.

Each solver needs to be given





- a name;
- a unique identifier (usually in reverse domain name notation);
- a version string; and
- the executable that can run FlatZinc.

In addition, you can specify the location of a solver-specific MiniZinc library (see Section 4.3.3 for details). If you leave this field empty, the MiniZinc standard library will be used. The path entered into this field should be an absolute path in the file system, without extra quoting, and without any command line arguments (such as -I).

Most solvers will require compilation and output processing, since they only deal with FlatZinc files. For these solvers, the *Run with mzn2fzn and solns2out* option must be selected. For solvers that can deal with MiniZinc models natively, this option can be deselected.

Some solvers open an independent application with its own graphical user interface. One such example is the *Gecode (Gist)* solver that comes with the bundled version of the IDE. For these solvers, select the *Run solver detached from IDE* option, so that the IDE does not wait for solver output.

Finally, you can select which command line flags are supported by the solver. This controls which options will be available in the solver configuration window.

Solver configurations that are edited or created through the IDE are saved in a configuration file in a standard location. These solvers are therefore available the next time the IDE is started, as well as through the `minizinc` command line tool.

CHAPTER 3.3

Globalizer

Globalizer¹ analyses a model and set of data to provide suggestions of global constraints that can replace or enhance a user's model.

3.3.1 Basic Usage

To use Globalizer simply execute it on a model and set of data files:

```
minizinc --solver org.minizinc.globalizer model.mzn data-1.dzn data-2.dzn
```

Note: Globalizer may also be executed on a non-parameterised model with no data files.

The following demonstrates the basic usage of Globalizer. Below, we see a simple model for the car sequencing problem², cars.mzn.

Listing 3.3.1: Model for the car sequencing problem (cars.mzn).

```
% cars.mzn
include "globals.mzn";

int: n_cars; int: n_options; int: n_classes;

set of int: steps = 1..n_cars;
set of int: options = 1..n_options;
set of int: classes = 1..n_classes;

array [options] of int: max_per_block;
array [options] of int: block_size;
array [classes] of int: cars_in_class;
array [classes, options] of 0..1: need;
```

¹ Leo, K. et al., “Globalizing Constraint Models”, 2013.

² <http://www.csplib.org/Problems/prob001/>

```
% The class of car being started at each step.
array [steps] of var classes: class;

% Which options are required by the car started at each step.
array [steps, options] of var 0..1: used;

% Block p must be used at step s if the class of the car to be
% produced at step s needs it.
constraint forall (s in steps, p in options) (used[s, p]=need[class[s], p]);

% For each option p with block size b and maximum limit m, no consecutive
% sequence of b cars contains more than m that require option p.
constraint
  forall (p in options, i in 1..(n_cars - (block_size[p] - 1))) (
    sum (j in 0..(block_size[p] - 1)) (used[i + j, p])
    <= max_per_block[p]);

% Require that the right number of cars in each class are produced.
constraint forall (c in classes) (count(class, c, cars_in_class[c]));

solve satisfy; % Find any solution.
```

And here we have a data file `cars_data.dzn`.

Listing 3.3.2: Data for the car sequencing problem (`cars_data.dzn`).

```
% cars_data.dzn
n_cars = 10;
n_options = 5;
n_classes = 6;
max_per_block = [1, 2, 1, 2, 1];
block_size = [2, 3, 3, 5, 5];
cars_in_class = [1, 1, 2, 2, 2, 2];

need = array2d(1..6, 1..5, [
  1, 0, 1, 1, 0,
  0, 0, 0, 1, 0,
  0, 1, 0, 0, 1,
  0, 1, 0, 1, 0,
  1, 0, 1, 0, 0,
  1, 1, 0, 0, 0
]);
```

Executing Globalizer on this model and data file we get the following output:

```
% minizinc --solver globalizer cars.mzn cars_data.dzn
cars.mzn|33|12|33|69 [ ] gcc(class,cars_in_class)
cars.mzn|33|35|33|68 [ ] count(class,c,cars_in_class[c])
```

```
cars.mzn|28|27|28|65;cars.mzn|29|9|30|32 [ ] sliding_sum(0,max_per_block[p],
→block_size[p],used[_, p])
```

Each line of output is comprised of three elements.

1. Expression locations (separated by semicolon ‘;’) that the constraint might be able to replace.
2. (Between square brackets ‘[‘]’) the context under which the constraint might replace the expressions of 1.
3. The replacement constraint.

From the example above we see that a `gcc` constraint and a `sliding_sum` constraint can replace some constraints in the model. Taking the `sliding_sum` case, we see that it replaces the expression `cars.mzn|28|27|28|65` which corresponds to `i` in `1..(n_cars - (block_size[p] - 1))` and `cars.mzn|29|27|28|65` which corresponds to the `<=` expression including the sum. The original constraint can be replaced with:

```
constraint forall (p in options) (
    sliding_sum(0, max_per_block[p], block_size[p], used[..,p]));
```

3.3.2 Caveats

MiniZinc syntax support. Globalizer was implemented with support for a subset of an early version of the MiniZinc 2.0 language. As a result there are some limitations. First, Globalizer does not support set variables or enum types. The array slicing syntax supported in MiniZinc was not decided upon when Globalizer was implemented so it uses an incompatible syntax. Globalizer uses `_` instead of `...`. This can be seen above in the translation of `used[_,p]` to `used[..,p]`.

New constraint. A special two argument version of the `global_cardinality` constraint called `gcc` has been added which is not in the standard library. It is defined as follows:

```
predicate gcc(array[int] of var int: x, array[int] of var int: counts) =
    global_cardinality(x,
        [ i | i in index_set(counts) ],
        array1d(counts));
```

3.3.3 Supported Constraints

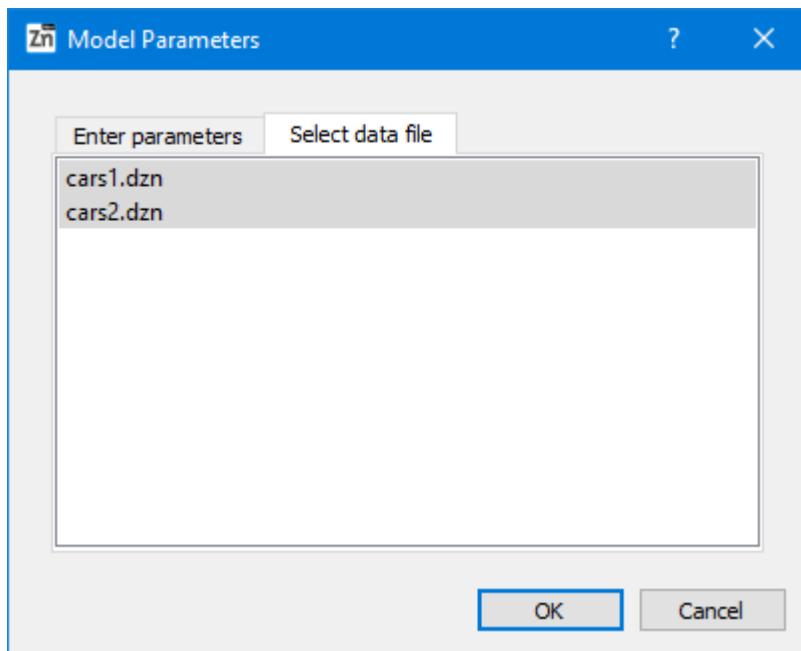
Globalizer currently supports detection of the following constraints:

alldifferent	alldifferent_except_0	all_equal_int
bin_packing	bin_packing_capa	bin_packing_load
binaries_represent_int	binaries_represent_int_3A	binaries_represent_int_3B
binaries_represent_int_3C	channel	channelACB

count_geq	cumulative_assert	decreasing
gcc	global_cardinality	inverse
lex_lesseq_int_checking	lex2_checking	maximum_int_checking
minimum_int_checking	member	nvalue
strict_lex2_checking	subcircuit_checking	true
atleast	atmost	bin_packing_load_ub
circuit_checking	count	diffn
distribute	element	increasing
lex_less_int_checking	sliding_sum	sort_checking
unary	value_precede_checking	

3.3.4 Using Globalizer in the MiniZinc IDE

To use the Globalizer in the MiniZinc IDE, open a model, and several data files. Select Globalizer from the solver configuration dropdown menu. Click the solve button (play symbol). If you did not select any data file, a dialog should pop up that allows you to select data files. To select multiple data files hold the **Ctrl** or **Cmd** key while clicking on each data file.



Click run. While processing your model and data files the MiniZinc IDE will display a progress bar on the toolbar.

The screenshot shows the MiniZinc IDE interface. The main window displays a MiniZinc model named 'cars.mzn' with the following content:

```

15 % The class of car being started at each step.
16 array [steps] of var classes: class;
17
18 % Which options are required by the car started at each step.
19 array [steps, options] of var 0..1: used;
20
21 % Block p must be used at step s if the class of the car to be
22 % produced at step s needs it.
23 constraint forall (s in steps, p in options) (used[s, p]=need[class[s], p]);
24
25 % For each option p with block size b and maximum limit m, no consecutive
26 % sequence of b cars contains more than m that require option p.
27 constraint
28   forall (p in options, i in 1..(n_cars - (block_size[p] - 1))) (
29     sum (j in 0..(block_size[p] - 1)) (used[i + j, p])
30       <= max_per_block[p]);
31
32 % Require that the right number of cars in each class are produced.
33 constraint forall (c in classes) (count(class, c, cars_in_class[c]));
34
35 solve satisfy; % Find any solution.

```

The 'Output' pane below the code editor lists discovered global constraints:

- [gcc\(class,\[1, 1, 2, 2, 2, 2\]\)](#)
- [gcc\(class,cars_in_class\)](#)
- [count\(class,c,cars_in_class\[c\]\)](#)
- [sliding_sum\(0,max_per_block\[p\],block_size\[p\],used\[, p\]\)](#)
% NUMCALLS: 610

At the bottom of the output pane, there is a status bar showing 'Ready.', a timer '1m 25s', and a zoom level '100%'.

Any discovered global constraints will be listed in the output pane of the IDE. Clicking on one of these constraints will highlight in the model the expressions that might be replaceable or strengthened by the constraint.

3.3.5 How it works

A summary of the algorithm is presented here. For a more detailed exploration of the approach see the Globalizing Constraint Models paper¹.

- **Normalize.** In this step the model is transformed to make analysis easier.
 - Conjunctions of constraints are broken into individual constraints. For example: $C1 \wedge C2;$ becomes $C1;$ $C2;$
 - Conjunctions under `forall` constraints are broken into individual `forall` constraints. For example: `forall(...) (C1 \wedge C2);` becomes `forall(...) (C1); forall(...) (C2)`
- **Generate submodels.** In this step all subsets containing 1 or 2 constraints (this can be configured using `--numConstraints` argument) are enumerated.

- **Instantiate and unroll into groups.** Each submodel is instantiated with the provided data files. These instantiated submodels are further used to produce more submodels by unrolling loops. For example, a constraint such as `forall(i in 1..n) (c(i));` will be used to produce the constraints: `c(1)` and `c(n)` for each instantiation of `n` in the data files. All instantiated submodels are then grouped together.
- **Process groups.** For each submodel in a group a set of 30 random solutions are found. (configurable using the `--randomSolutions` argument). A template model with all of these random solutions is created.

The different variables (including arrays and array accesses) and parameters used by a submodel are collected into a set of potential arguments along with the constant 0 and a special blank symbol representing an argument that can be inferred based on others used.

The list of constraints above is filtered based on the arguments available. For example, the `alldifferent` constraint will be removed from the list of candidates if the submodel does not reference any arrays of variables.

Finally the set of constraints are filtered by adding them to the template and solving it. If the template is satisfiable, the constraint accepts all of the random solutions.

30 sample solutions for each candidate constraint are generated. (configurable using the `--sampleSolutions` argument). The candidate is then ranked based on how many of these solutions are also solutions to the original submodel. If its score is less than some threshold the candidate is removed from the set of candidates for this group and will not be tested on later submodels.

- **Report.** The remaining candidates are presented to the user.

3.3.6 Performance tips

Globalizing constraint models can be a time consuming task. If

- **Use small or relatively easy instances.** Globalizer solves many subproblems while processing your model and data files. Using easier instances can speed up the process considerably.
- **Disable the initial pass.** As discussed above, Globalizer performs two passes. The first pass tries to detect alternate viewpoints that can be added to your model. If you are confident that this will not be useful we recommend disabling this first pass using the `--no-initial-pass` argument.
- **Narrow search using filters.** Globalizer attempts to match a large selection of global constraints to subproblems in your model. If you wish to check if only a few specific constraints are present you can focus Globalizer using the `--constraintFilter` or `-f` arguments followed by a comma separated list of strings. Only global constraints where one of the strings is a substring will be included in the search.
- **Disable implies check.** The implication check can also be disabled to improve performance. This results in less accurate results but may still help a user to understand the structure of their model.
- **Free-search.** To improve accuracy and to avoid false positives, subproblems solved by Globalizer are solved using a random heuristic and restarts. If the subproblems are particularly difficult to solve a random heuristic may be prohibitive and the solver may not

be able to produce enough samples within the solving timelimit. In these circumstances a user can either increase the solver timeout using the `-t` argument followed by the number of milliseconds a solver should be given to find samples. Alternatively the `--free-search` argument can be used to force Globalizer to use the solver's free search to find samples. This has the downside of reducing the diversity of samples but allows enough samples to be found to allow suggested globals to be found.

3.3.7 Limitations / Future work

- Globalizer supports only a subset of the MiniZinc language and as such cannot be executed on all MiniZinc models.
- There are some relatively cheap syntactic approaches that should be performed before Globalization that currently is not implemented. For example, there are several common formulations of an `alldifferent` constraint that can be detected syntactically. This would be much cheaper than using Globalizer.

CHAPTER 3.4

FindMUS

FindMUS¹ lists unsatisfiable subsets of constraints in your MiniZinc model. These subsets, called Minimal Unsatisfiable Subsets can help you to find faults in your unsatisfiable constraint model. FindMUS uses the hierarchical structure provided by the model to guide its search.

3.4.1 Basic Usage

To use FindMUS on the command line simply execute it on a model and set of data files by typing:

```
minizinc --solver findMUS model.mzn data-1.dzn
```

Note: FindMUS requires a fully instantiated constraint model.

3.4.1.1 Commandline arguments

The FindMUS tool supports the following arguments:

Driver options

The driver creates the main solver and sub-solver and requests MUSes from FindMUS's enumeration algorithm HierMUS.

-n <count> Stop after the n th MUS is found (Default: 1)

-a Find all MUSes

--timeout <s> Stop search after s seconds (Default: 1800)

Enumeration options

The enumeration algorithm (HierMUS) explores the constraint hierarchy provided in the user's model, proposes potential MUSes to the sub-solver. The default algorithm is internally referred

¹ Leo, K. et al., "Debugging Unsatisfiable Constraint Models", 2017.

to as stackMUS. This algorithm can be replaced with either MARCO or ReMUS which have their own strengths. Our implementation of these algorithms in turn utilizes a simple linear deletion based ‘shrink’ method. This can be replaced with the binary-splitting QuickXplain which can be much quicker.

```
--marco Use the MARCO3 algorithm as sub-enumerator  
--remus Use the ReMUS4 algorithm as sub-enumerator (not compatible with Hierarchical Search)  
--qx Use QuickXplain5 for shrink step of MARCO or ReMUS  
--depth mzn,fzn,<n> How deep in the tree should search explore. (Default: 1)  
    mzn expands the search as far as the point when the compiler leaves the MiniZinc  
    model.  
    fzn expands search as far as the FlatZinc constraints.  
    <n> expand search to the n level of the hierarchy.
```

Subsolver options

FindMUS can be used in conjunction with any FlatZinc solver. These options mimic the `minizinc` arguments `--solver` and `--fzn-flags`. The behavior of these arguments is likely to change in later versions of the tool.

```
--solver <s> Use solver s for SAT checking. (Default: “gecode”)  
--solver-flags <f> Pass flags f to sub-solver. (Default: empty)  
--solver-timelimit <ms> Set hard time limit for solver in milliseconds. (Default: 1100)  
--soft-defines Consider functional constraints as part of MUSes
```

Filtering options

FindMUS can include or exclude constraints from its search based on the expression and constraint name annotations as well as properties of their paths (for example, line numbers). These filters are currently based on substrings but in future may support text spans and regular expressions.

```
--named-only Only consider constraints annotated with string annotations  
--filter-named <names>; --filter-named-exclude <names>; Include/exclude  
constraints with names that match sep separated names  
--filter-path <paths>; --filter-path-exclude <paths>; Include/exclude based  
on paths  
--filter-sep <sep>; Separator used for named and path filters
```

Structure options

The structure coming from a user’s model can significantly impact the performance of a MUS enumeration algorithm. Here we allow the structure to be generalized in various ways and extra structure can be injected in the form of binarization of the tree.

```
--structure flat,gen,normal,mix
```

³ Liffiton, M. H. et al., “Fast, Flexible MUS Enumeration”, 2016.

⁴ Bendík, J. et al., “Recursive Online Enumeration of All Minimal Unsatisfiable Subsets”, 2018.

⁵ Junker, U. et al., “QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems”, 2004.

Alters initial structure: (Default: normal)

- flat Remove all structure
- gen Remove instance specific structure
- normal No change
- mix Apply gen before normal

--binarize normal,leaves,all

Add additional structure: (Default: normal)

- normal No change
- leaves Introduce structure at the leaves
- all Introduce structure throughout tree

Verbosity options

--verbose-{map,enum,subsolve} <n> Set verbosity level for different components
 --verbose Set verbosity level of all components to 1

Misc options

--dump-dot <dot> Write tree in GraphViz format to file <dot>

3.4.1.2 Example

The following demonstrates the basic usage of FindMUS on a simple example. Below, we see a model for the latin squares puzzle² with some incompatible symmetry breaking constraints added.

Listing 3.4.1: Faulty model for Latin Squares (latin_squares.mzn).

```
% latin_squares.mzn
include "globals.mzn";

int: n = 3;
set of int: N = 1..n;
array[N, N] of var N: X;

constraint :: "ADRows"
  forall (i in N)
    (alldifferent(row(X, i)) :: "AD(row \((i))");
constraint :: "ADCols"
  forall (j in N)
    (alldifferent(col(X, j)) :: "AD(col \((j))");
constraint :: "LLRows"
  forall (i in 1..n-1)
    (lex_less(row(X, i), row(X, i+1)) :: "LL(rows \((i)) \((i+1))");
constraint :: "LGcols"
```

² https://en.wikipedia.org/wiki/Latin_square

```

    forall (j in 1..n-1)
        (lex_greater(col(X, j), col(X, j+1)) :: "LG(cols \j \j+1)");

solve satisfy;

output [ show2d(X) ];

```

Here we have used the new constraint and expression annotations added in MiniZinc 2.2.0. Note that these annotations are not necessary for FindMUS to work but may help with interpreting the output. The first two constraints: ADRows and ADCols define the alldifferent constraints on the respective rows and columns of the latin square. The next two constraints LLRows and LGCols post lex constraints that order the rows to be increasing and the columns to be increasing. Certain combinations of these constraints are going to be in conflict.

Executing the command `minizinc --solver findMUS -a latin_squares.mzn` returns the following output. Note that the `-a` argument requests all MUSes that can be found with the default settings (more detail below).

```

FznSubProblem: hard cons: 36 soft cons: 26 leaves: 26 branches: 21
↳ Built tree in 0.03100 seconds.
SubsetMap: nleaves: 4 nbranches: 1
MUS: 0 1 2 21 22 3 32 33 4 43 44 5 54 55 6 7 8
Brief: exists;@{LG(cols 1 2@LGCols)}:() exists;@{LG(cols 2 3@LGCols)}:()
↳ exists;@{LL(rows 1 2)@LLRows}:() exists;@{LL(rows 2 3)@LLRows}:() int_lin_
↳ le;@{LG(cols 1 2@LGCols)}:() int_lin_le;@{LG(cols 2 3@LGCols)}:() int_lin_le;
↳ @{LL(rows 1 2)@LLRows}:() int_lin_le;@{LL(rows 2 3)@LLRows}:() int_lin_ne;@
↳ {AD(row 1)@ADRows}:() int_lin_ne;@{AD(row 1)@ADRows}:() int_lin_ne;@
↳ {AD(row 1)@ADRows}:() int_lin_ne;@{AD(row 2)@ADRows}:() int_lin_ne;@
↳ {AD(row 2)@ADRows}:() int_lin_ne;@{AD(row 2)@ADRows}:() int_lin_ne;@
↳ {AD(row 3)@ADRows}:() int_lin_ne;@{AD(row 3)@ADRows}:() int_lin_ne;@
↳ {AD(row 3)@ADRows}:()

Traces:
latin_squares.mzn|9|5|10|51|ca|forall
latin_squares.mzn|16|5|17|68|ca|forall
latin_squares.mzn|19|5|20|70|ca|forall
=====
```

```

MUS: 10 11 12 13 14 15 16 17 21 22 32 33 43 44 54 55 9
Brief: exists;@{LG(cols 1 2@LGCols)}:() exists;@{LG(cols 2 3@LGCols)}:()
↳ exists;@{LL(rows 1 2)@LLRows}:() exists;@{LL(rows 2 3)@LLRows}:() int_lin_
↳ le;@{LG(cols 1 2@LGCols)}:() int_lin_le;@{LG(cols 2 3@LGCols)}:() int_lin_le;
↳ @{LL(rows 1 2)@LLRows}:() int_lin_le;@{LL(rows 2 3)@LLRows}:() int_lin_ne;@
↳ {AD(col 1)@ADCols}:() int_lin_ne;@{AD(col 1)@ADCols}:() int_lin_ne;@
↳ {AD(col 1)@ADCols}:() int_lin_ne;@{AD(col 2)@ADCols}:() int_lin_ne;@
↳ {AD(col 2)@ADCols}:() int_lin_ne;@{AD(col 2)@ADCols}:() int_lin_ne;@
↳ {AD(col 3)@ADCols}:() int_lin_ne;@{AD(col 3)@ADCols}:() int_lin_ne;@
↳ {AD(col 3)@ADCols}:()
```

Traces:

```

latin_squares.mzn|16|5|17|68|ca|forall
latin_squares.mzn|12|5|13|51|ca|forall
latin_squares.mzn|19|5|20|70|ca|forall
=====
Total Time: 0.24700      nmuses: 2      map: 10 sat: 6 total: 16
=====UNKNOWN=====

```

The first two lines, starting with `FznSubProblem:` and `SubsetMap` provide some useful information for debugging the `findMUS` tool. Next we have the list of MUSes separated by a series of equals = signs. Each MUS is described with three sections:

1. MUS: lists the indices of FlatZinc constraints involved in this MUS.
2. Brief: lists the FlatZinc constraint name, the expression name, and the constraint name for each involved FlatZinc constraint.
3. Traces: lists the MiniZinc paths corresponding to the constraints of the MUS. Each path typically contains a list of path elements separated by semi-colons ;. Each element includes a file path, a start line, start column, end line and end column denoting a span of text from the listed file. And finally, a description of the element. In the examples above all paths point to calls to a forall on different lines of the model. (ca|forall)

The final line of output lists the `Total Time`, the number of MUSes found, and some statistics about the number of times the internal map solver `map` was called, and the number of times the subproblem solver was called `sat`.

Interpreting the two MUSes listed here we see that the `lex` constraints from lines 16 and 19 were included in both and only one of the alldifferent constraints from line 9 and 12 are required for the model to be unsatisfiable. The `lex` constraints being involved in every MUS make them a strong candidate for being the source of unsatisfiability in the user's model.

3.4.2 Using FindMUS in the MiniZinc IDE

To use FindMUS in the MiniZinc IDE, upon discovering that a model is unsatisfiable. Select `FindMUS` from the solver configuration dropdown menu and click the solve button (play symbol). By default FindMUS is configured to return a single MUS at a depth of '1'. This should be relatively fast and help locate the relevant constraint items. The following shows the result of running FindMUS with the default options.

The screenshot shows the MiniZinc IDE interface. The main window displays the source code for a Latin squares model named `latin_squares.mzn`. The code defines variables and constraints for a 3x3 Latin square, including global constraints for rows, columns, and diagonals. The solver output window at the bottom shows the execution details, including the MUS (Minimal Unsatisfiable Subformula) found by the solver.

```

% latin_squares.mzn
include "globals.mzn";

int: n = 3;
set of int: N = 1..n;
array[N, N] of var N: X;

constraint :: "ADRows"
  forall (i in N)
    (alldifferent(row(X, i)) :: "AD(row \((i))");

constraint :: "ADCols"
  forall (j in N)
    (alldifferent(col(X, j)) :: "AD(col \((j))");

constraint :: "LLRows"
  forall (i in 1..n-1)
    (lex_less(row(X, i), row(X, i+1)) :: "LL(rows \((i) \((i+1))");

constraint :: "LGCols"
  forall (j in 1..n-1)
    (lex_greater(col(X, j), col(X, j+1)) :: "LG(cols \((j) \((j+1))");

solve satisfy;

output [ show2d(X) ];

```

Output window content:

```

int lin_ne:@{AD(row 2)@ADRows}:()
int lin_ne:@{AD(row 2)@ADRows}:()
int lin_ne:@{AD(row 2)@ADRows}:()
int lin_ne:@{AD(row 3)@ADRows}:()
int lin_ne:@{AD(row 3)@ADRows}:()
int lin_ne:@{AD(row 3)@ADRows}:()

Total Time: 0.19000      nnodes: 1      map: 6   sat: 5   total: 11
Finished in 475msec

```

Bottom status bar: Ready. 475msec

Selecting the returned MUS highlights three top level constraints as shown: `ADRows`, `LLRows` and `LGCols`. To get a more specific MUS we can instruct `FindMUS` to go deeper than the top level constraints by clicking the “Show configuration editor” button in the top right hand corner of the MiniZinc IDE window, and adding `--depth mzn` to the “Additional solver command line arguments” textbox in the “Solver options” section. The following shows a more specific MUS in this model.

The screenshot shows the MiniZinc IDE interface. The main window displays the source code for a MiniZinc model named `latin_squares.mzn`. The code defines a 3x3 Latin square with constraints for rows, columns, and 2x2 blocks. The output pane below shows the generated FlatZinc code, which includes specific constraint names and their parameters, such as `exists;@{LG(cols 1 2@LGCols):(j=1)`.

```

% latin_squares.mzn
include "globals.mzn";

int: n = 3;
set of int: N = 1..n;
array[N, N] of var N: X;

constraint :: "ADRows"
  forall (i in N)
    (alldifferent(row(X, i)) :: "AD(row \$(i))");
constraint :: "ADCols"
  forall (j in N)
    (alldifferent(col(X, j)) :: "AD(col \$(j))");

constraint :: "LLRows"
  forall (i in 1..n-1)
    (lex_less(row(X, i), row(X, i+1)) :: "LL(rows \$(i) \$(i+1))");
constraint :: "LGCols"
  forall (j in 1..n-1)
    (lex_greater(col(X, j), col(X, j+1)) :: "LG(cols \$(j) \$(j+1))");

solve satisfy;

```

Output

```

10 11 12 13 14 21 22 32 33 43 44 9 Brief: exists;@{LG(cols 1 2@LGCols):(j=1)
exists;@{LL(rows 1 2)@LLRows}:(i=1)
exists;@{LL(rows 2 3)@LLRows}:(i=2)
int_lin_le;@{LG(cols 1 2@LGCols):(j=1)
int_lin_le;@{LL(rows 1 2)@LLRows}:(i=1)
int_lin_le;@{LL(rows 2 3)@LLRows}:(i=2)
int_lin_ne;@{AD(col 1)@ADCols}:(j=1)
int_lin_ne;@{AD(col 1)@ADCols}:(j=1)
int_lin_ne;@{AD(col 1)@ADCols}:(j=1)
int_lin_ne;@{AD(col 2)@ADCols}:(j=2)
int_lin_ne;@{AD(col 2)@ADCols}:(j=2)
int_lin_ne;@{AD(col 2)@ADCols}:(j=2)

```

Ready. 1s 70msec

In this case we can see that the output pane lists more specific information about the constraints involved in the MUS. After each listed constraint name we see what any loop variables were assigned to when the constraint was added to the FlatZinc. For example `(j=2)`.

3.4.3 How it works

A simple explanation of the algorithm is presented here. For a more detailed exploration of an earlier version of the approach see the Debugging Unsatisfiable Constraint Models paper¹.

The approach takes the full FlatZinc program and partitions the constraints into groups based on the hierarchy provided in the user's model. To begin with (at depth '1') we search for MUSes in the set of top level constraint items. If we are not at the target depth we recursively select a found MUS, split its constituent constraints into lower level constraints based on the hierarchy and begin another search for MUSes underneath this high-level MUS. If any MUSes are found we know that the high-level MUS is not minimal and so it should not be reported. This process is repeated on any found MUSes until we reach the required depth at which point we will start to report MUSes. If in the recursive search we return to a high-level MUS without finding any sub-MUSes we can report this MUS as a real MUS. This recursive process is referred to as HierMUS. At each stage when we request the enumeration of a set of MUSes underneath a high-level MUS we can use one of several MUS enumeration algorithms. By default we use the internally developed StackMUS. We can also utilize MARCO and ReMUS for this role.

3.4.4 Performance tips

If you are trying to find MUSes in a very large instance it is advised to make use of the filtering tools available. Use the default settings to find a very high-level MUS and then use the `--depth` option to find lower-level, more specific MUSes in conjunction with the `--filter-name` and `--filter-path` options to focus on finding specific sub-MUSes of a found high-level MUS.

3.4.5 Limitations / Future work

There are several features that we aim to include quite soon:

Regular expression based filtering This will allow more complex filtering to be used.

Text span based filtering This will allow a user to simply click-and-drag a selection around the parts of a constraint model they wish to analyse.

Single MUS focus mode This mode would perform the process outlined in the 'Performance tips' section automatically making it easier for users to find detailed MUSes.

CHAPTER 3.5

Using MiniZinc in Jupyter Notebooks

You can use MiniZinc inside a Jupyter / IPython notebook using the `iminizinc` Python module. The module provides a “cell magic” extension that lets you solve MiniZinc models.

The module requires an existing installation of MiniZinc.

3.5.1 Installation

You can install or upgrade this module via pip:

```
pip install -U iminizinc
```

Consult your Python documentation to find out if you need any extra options (e.g. you may want to use the `--user` flag to install only for the current user, or you may want to use virtual environments).

Make sure that the `minizinc` binary are on the PATH environment variable when you start the notebook server. The easiest way to do that is to get the “bundled installation” that includes the MiniZinc IDE and a few solvers, available from GitHub here: <https://github.com/MiniZinc/MiniZincIDE/releases/latest> You then need to change your PATH environment variable to include the MiniZinc installation.

3.5.2 Basic usage

After installing the module, you have to load the extension using `%load_ext iminizinc`. This will enable the cell magic `%minizinc`, which lets you solve MiniZinc models. Here is a simple example:

```
In[1]: %load_ext iminizinc  
  
In[2]: n=8  
  
In[3]: %%minizinc  
  
        include "globals.mzn";  
        int: n;  
        array[1..n] of var 1..n: queens;  
        constraint all_different(queens);  
        constraint all_different([queens[i]+i | i in 1..n]);  
        constraint all_different([queens[i]-i | i in 1..n]);  
        solve satisfy;  
Out[3]: {u'queens': [4, 2, 7, 3, 6, 8, 5, 1]}
```

As you can see, the model binds variables in the environment (in this case, `n`) to MiniZinc parameters, and returns an object with fields for all declared decision variables.

Alternatively, you can bind the decision variables to Python variables:

```
In[1]: %load_ext iminizinc  
  
In[2]: n=8  
  
In[3]: %%minizinc -m bind  
  
        include "globals.mzn";  
        int: n;  
        array[1..n] of var 1..n: queens;  
        constraint all_different(queens);  
        constraint all_different([queens[i]+i | i in 1..n]);  
        constraint all_different([queens[i]-i | i in 1..n]);  
        solve satisfy;  
  
In[4]: queens  
  
Out[4]: [4, 2, 7, 3, 6, 8, 5, 1]
```

If you want to find all solutions of a satisfaction problem, or all intermediate solutions of an optimisation problem, you can use the `-a` flag:

```
In[1]: %load_ext iminizinc  
  
In[2]: n=6  
  
In[3]: %%minizinc -a  
  
        include "globals.mzn";  
        int: n;
```

```
array[1..n] of var 1..n: queens;
constraint all_different(queens);
constraint all_different([queens[i]+i | i in 1..n]);
constraint all_different([queens[i]-i | i in 1..n]);
solve satisfy;

Out[3]: [{u'queens': [5, 3, 1, 6, 4, 2]},
          {u'queens': [4, 1, 5, 2, 6, 3]},
          {u'queens': [3, 6, 2, 5, 1, 4]},
          {u'queens': [2, 4, 6, 1, 3, 5]}]
```

The magic supports a number of additional options, in particular loading MiniZinc models and data from files. Some of these may only work with the development version of MiniZinc (i.e., not the one that comes with the bundled binary releases). You can take a look at the help using

```
In[1]: %%minizinc?
```


Part 4

Reference Manual

CHAPTER 4.1

Specification of MiniZinc

4.1.1 Introduction

This document defines MiniZinc, a language for modelling constraint satisfaction and optimisation problems.

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;
- high-level data structures and data encapsulation (sets, arrays, enumerated types, constrained type-insts);
- extensibility (user-defined functions and predicates);
- reliability (type checking, instantiation checking, assertions);
- solver-independent modelling;
- simple, declarative semantics.

MiniZinc is similar to OPL and moves closer to CLP languages such as ECLiPSe.

This document has the following structure. [Notation](#) (page 180) introduces the syntax notation used throughout the specification. [Overview of a Model](#) (page 181) provides a high-level overview of MiniZinc models. [Syntax Overview](#) (page 183) covers syntax basics. [High-level Model Structure](#) (page 184) covers high-level structure: items, multi-file models, namespaces, and scopes. [Types and Type-insts](#) (page 186) introduces types and type-insts. [Expressions](#) (page 194) covers expressions. [Items](#) (page 208) describes the top-level items in detail. [Annotations](#) (page 216) describes annotations. [Partiality](#) (page 217) describes how partiality is handled in various cases. [Built-in Operations](#) (page 219) describes the language built-ins. [Full grammar](#)

(page 229) gives the MiniZinc grammar. *Content-types* (page 227) defines content-types used in this specification.

This document also provides some explanation of why certain design decisions were made. Such explanations are marked by the word *Rationale* and written in italics, and do not constitute part of the specification as such. *Rationale:* *These explanations are present because they are useful to both the designers and the users of MiniZinc.*

4.1.1.1 Original authors.

The original version of this document was prepared by Nicholas Nethercote, Kim Marriott, Reza Rafeh, Mark Wallace and Maria Garcia de la Banda. MiniZinc is evolving, however, and so is this document.

For a formally published paper on the MiniZinc language and the superset Zinc language, please see:

- N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 529–543. Springer-Verlag, 2007.
- K. Marriott, N. Nethercote, R. Rafeh, P.J. Stuckey, M. Garcia de la Banda, and M. Wallace. The Design of the Zinc Modelling Language. *Constraints*, 13(3):229–267, 2008.

4.1.2 Notation

The basics of the EBNF used in this specification are as follows.

- Non-terminals are written between angle brackets, `<item>`.
- Terminals are written in double quotes, e.g. `"constraint"`. A double quote terminal is written as a sequence of three double quotes: `"""`.
- Optional items are written in square brackets, e.g. `["var"]`.
- Sequences of zero or more items are written with parentheses and a star, e.g. `("," <ident>)*`.
- Sequences of one or more items are written with parentheses and a plus, e.g. `(<msg>)+`.
- Non-empty lists are written with an item, a separator/terminator terminal, and three dots. For example, this:

```
<expr> ", " ...
```

is short for this:

```
<expr> ( "," <expr> )* [ " " ]
```

The final terminal is always optional in non-empty lists.

- Regular expressions are used in some productions, e.g. `[-+]?[0-9]+`.

MiniZinc's grammar is presented piece-by-piece throughout this document. It is also available as a whole in [Full grammar](#) (page 229). The output grammar also includes some details of the use of whitespace. The following conventions are used:

- A newline character or CRLF sequence is written `\n`.

4.1.3 Overview of a Model

Conceptually, a MiniZinc problem specification has two parts.

1. The *model*: the main part of the problem specification, which describes the structure of a particular class of problems.
2. The *data*: the input data for the model, which specifies one particular problem within this class of problems.

The pairing of a model with a particular data set is a *model instance* (sometimes abbreviated to *instance*).

The model and data may be separated, or the data may be “hard-wired” into the model. [Model Instance Files](#) (page 185) specifies how the model and data can be structured within files in a model instance.

There are two broad classes of problems: satisfaction and optimisation. In satisfaction problems all solutions are considered equally good, whereas in optimisation problems the solutions are ordered according to an objective and the aim is to find a solution whose objective is optimal. [Solve Items](#) (page 211) specifies how the class of problem is chosen.

4.1.3.1 Evaluation Phases

A MiniZinc model instance is evaluated in two distinct phases.

1. Instance-time: static checking of the model instance.
2. Run-time: evaluation of the instance (i.e., constraint solving).

The model instance may not compile due to a problem with the model and/or data, detected at instance-time. This could be caused by a syntax error, a type-inst error, the use of an unsupported feature or operation, etc. In this case the outcome of evaluation is a static error; this must be reported prior to run-time. The form of output for static errors is implementation-dependent, although such output should be easily recognisable as a static error.

An implementation may produce warnings during all evaluation phases. For example, an implementation may be able to determine that unsatisfiable constraints exist prior to run-time, and the resulting warning given to the user may be more helpful than if the unsatisfiability is detected at run-time.

An implementation must produce a warning if the objective for an optimisation problem is unbounded.

4.1.3.2 Run-time Outcomes

Assuming there are no static errors, the output from the run-time phase has the following abstract form:

```
<output> ::= <no-solutions> [ <warnings> ] <free-text>
| ( <solution> )* [ <complete> ] [ <warnings> ] <free-text>
```

If a solution occurs in the output then it must be feasible. For optimisation problems, each solution must be strictly better than any preceding solution.

If there are no solutions in the output, the outcome must indicate that there are no solutions.

If the search is complete the output may state this after the solutions. The absence of the completeness message indicates that the search is incomplete.

Any warnings produced during run-time must be summarised after the statement of completeness. In particular, if there were any warnings at all during run-time then the summary must indicate this fact.

The implementation may produce text in any format after the warnings. For example, it may print a summary of benchmarking statistics or resources used.

4.1.3.3 Output

Implementations must be capable of producing output of content type application/x-zinc-output, which is described below and also in [Content-types](#) (page 227). Implementations may also produce output in alternative formats. Any output should conform to the abstract format from the previous section and must have the semantics described there.

Content type application/x-zinc-output extends the syntax from the previous section as follows:

```
<solution> ::= <solution-text> [ \n ] "-----" \n
```

The solution text for each solution must be as described in [Output Items](#) (page 212). A newline must be appended if the solution text does not end with a newline. *Rationale: This allows solutions to be extracted from output without necessarily knowing how the solutions are formatted.* Solutions end with a sequence of ten dashes followed by a newline.

```
<no-solutions> ::= "=====UNSATISFIABLE=====" \n
```

The completeness result is printed on a separate line. *Rationale: The strings are designed to clearly indicate the end of the solutions.*

```
<complete> ::= "===== " \n
```

If the search is complete, a statement corresponding to the outcome is printed. For an outcome of no solutions the statement is that the model instance is unsatisfiable, for an outcome of no more solutions the statement is that the solution set is complete, and for an outcome of no better solutions the statement is that the last solution is optimal. *Rationale: These are the logical implications of a search being complete.*

```
<warnings> ::= ( <message> )+
<message> ::= ( <line> )+
```

If the search is incomplete, one or more messages describing reasons for incompleteness may be printed. Likewise, if any warnings occurred during search they are repeated after the completeness message. Both kinds of message should have lines that start with % so they are recognized as comments by post-processing. *Rationale:* *This allows individual messages to be easily recognised.*

For example, the following may be output for an optimisation problem:

```
=====UNSATISFIABLE=====
% trentin.fzn:4: warning: model inconsistency detected before search.
```

Note that, as in this case, an unbounded objective is not regarded as a source of incompleteness.

4.1.4 Syntax Overview

4.1.4.1 Character Set

The input files to MiniZinc must be encoded as UTF-8.

MiniZinc is case sensitive. There are no places where upper-case or lower-case letters must be used.

MiniZinc has no layout restrictions, i.e., any single piece of whitespace (containing spaces, tabs and newlines) is equivalent to any other.

4.1.4.2 Comments

A % indicates that the rest of the line is a comment. MiniZinc also has block comments, using symbols /* and */ to mark the beginning and end of a comment.

4.1.4.3 Identifiers

Identifiers have the following syntax:

```
<ident> ::= [A-Za-z][A-Za-z0-9_]*           % excluding keywords
          | "'' [^\xa\xd\x0]* "''
```

```
my_name_2
MyName2
'An arbitrary identifier'
```

A number of keywords are reserved and cannot be used as identifiers. The keywords are: `ann`, `annotation`, `any`, `array`, `bool`, `case`, `constraint`, `diff`, `div`, `else`, `elseif`, `endif`, `enum`, `false`,

`float, function, if, in, include, int, intersect, let, list, maximize, minimize, mod, not, of, op, opt, output, par, predicate, record, satisfy, set, solve, string, subset, superset, symdiff, test, then, true, tuple, type, union, var, where, xor.`

A number of identifiers are used for built-ins; see [Built-in Operations](#) (page 219) for details.

4.1.5 High-level Model Structure

A MiniZinc model consists of multiple *items*:

```
<model> ::= [ <item> ";" ... ]
```

Items can occur in any order; identifiers need not be declared before they are used. Items have the following top-level syntax:

```
<item> ::= <include-item>
          | <var-decl-item>
          | <enum-item>
          | <assign-item>
          | <constraint-item>
          | <solve-item>
          | <output-item>
          | <predicate-item>
          | <test-item>
          | <function-item>
          | <annotation-item>

<ti-expr-and-id> ::= <ti-expr> ":" <ident>
```

Include items provide a way of combining multiple files into a single instance. This allows a model to be split into multiple files ([Include Items](#) (page 208)).

Variable declaration items introduce new global variables and possibly bind them to a value ([Variable Declaration Items](#) (page 209)).

Assignment items bind values to global variables ([Assignment Items](#) (page 210)).

Constraint items describe model constraints ([Constraint Items](#) (page 211)).

Solve items are the “starting point” of a model, and specify exactly what kind of solution is being looked for: plain satisfaction, or the minimization/maximization of an expression. Each model must have exactly one solve item ([Solve Items](#) (page 211)).

Output items are used for nicely presenting the result of a model execution ([Output Items](#) (page 212)).

Predicate items, test items (which are just a special type of predicate) and function items introduce new user-defined predicates and functions which can be called in expressions ([User-defined Operations](#) (page 212)). Predicates, functions, and built-in operators are described collectively as *operations*.

Annotation items augment the `ann` type, values of which can specify non-declarative and/or solver-specific information in a model.

4.1.5.1 Model Instance Files

MiniZinc models can be constructed from multiple files using include items (see [Include Items](#) (page 208)). MiniZinc has no module system as such; all the included files are simply concatenated and processed as a whole, exactly as if they had all been part of a single file. *Rationale: We have not found much need for one so far. If bigger models become common and the single global namespace becomes a problem, this should be reconsidered.*

Each model may be paired with one or more data files. Data files are more restricted than model files. They may only contain variable assignments (see [Assignment Items](#) (page 210)).

Data files may not include calls to user-defined operations.

Models do not contain the names of data files; doing so would fix the data file used by the model and defeat the purpose of allowing separate data files. Instead, an implementation must allow one or more data files to be combined with a model for evaluation via a mechanism such as the command-line.

When checking a model with data, all global variables with fixed type-insts must be assigned, unless they are not used (in which case they can be removed from the model without effect).

A data file can only be checked for static errors in conjunction with a model, since the model contains the declarations that include the types of the variables assigned in the data file.

A single data file may be shared between multiple models, so long as the definitions are compatible with all the models.

4.1.5.2 Namespaces

All names declared at the top-level belong to a single namespace. It includes the following names.

1. All global variable names.
2. All function and predicate names, both built-in and user-defined.
3. All enumerated type names and enum case names.
4. All annotation names.

Because multi-file MiniZinc models are composed via concatenation ([Model Instance Files](#) (page 185)), all files share this top-level namespace. Therefore a variable `x` declared in one model file could not be declared with a different type in a different file, for example.

MiniZinc supports overloading of built-in and user-defined operations.

4.1.5.3 Scopes

Within the top-level namespace, there are several kinds of local scope that introduce local names:

- Comprehension expressions ([Set Comprehensions](#) (page 201)).
- Let expressions ([Let Expressions](#) (page 206)).
- Function and predicate argument lists and bodies ([User-defined Operations](#) (page 212)).

The listed sections specify these scopes in more detail. In each case, any names declared in the local scope overshadow identical global names.

4.1.6 Types and Type-insts

MiniZinc provides four scalar built-in types: Booleans, integers, floats, and strings; enumerated types; two compound built-in types: sets and multi-dimensional arrays; and the user extensible annotation type `ann`.

Each type has one or more possible *instantiations*. The instantiation of a variable or value indicates if it is fixed to a known value or not. A pairing of a type and instantiation is called a *type-inst*.

We begin by discussing some properties that apply to every type. We then introduce instantiations in more detail. We then cover each type individually, giving: an overview of the type and its possible instantiations, the syntax for its type-insts, whether it is a finite type (and if so, its domain), whether it is varifiable, the ordering and equality operations, whether its variables must be initialised at instance-time, and whether it can be involved in automatic coercions.

4.1.6.1 Properties of Types

The following list introduces some general properties of MiniZinc types.

- Currently all types are monotypes. In the future we may allow types which are polymorphic in other types and also the associated constraints.
- We distinguish types which are *finite types*. In MiniZinc, finite types include Booleans, enums, types defined via set expression type-insts such as range types (see *Set Expression Type-insts* (page 194)), as well as sets and arrays, composed of finite types. Types that are not finite types are unconstrained integers, unconstrained floats, unconstrained strings, and `ann`. Finite types are relevant to sets (`spec-Sets`) and array indices (`spec-Arrays`). Every finite type has a *domain*, which is a set value that holds all the possible values represented by the type.
- Every first-order type (this excludes `ann`) has a built-in total order and a built-in equality; $>$, $<$, $==/!=$, $!=$, \leq and \geq comparison operators can be applied to any pair of values of the same type. *Rationale:* This facilitates the specification of symmetry breaking and of polymorphic predicates and functions. Note that, as in most languages, using equality on floats or types that contain floats is generally not reliable due to their inexact representation. An implementation may choose to warn about the use of equality with floats or types that contain floats.

4.1.6.2 Instantiations

When a MiniZinc model is evaluated, the value of each variable may initially be unknown. As it runs, each variable's *domain* (the set of values it may take) may be reduced, possibly to a single value.

An *instantiation* (sometimes abbreviated to *inst*) describes how fixed or unfixed a variable is at instance-time. At the most basic level, the instantiation system distinguishes between two kinds of variables:

1. *Parameters*, whose values are fixed at instance-time (usually written just as “fixed”).

2. *Decision variables* (often abbreviated to *variables*), whose values may be completely unfixed at instance-time, but may become fixed at run-time (indeed, the fixing of decision variables is the whole aim of constraint solving).

In MiniZinc decision variables can have the following types: Booleans, integers, floats, and sets of integers, and enums. Arrays and `ann` can contain decision variables.

4.1.6.3 Type-insts

Because each variable has both a type and an inst, they are often combined into a single *type-inst*. Type-insts are primarily what we deal with when writing models, rather than types.

A variable's type-inst *never changes*. This means a decision variable whose value becomes fixed during model evaluation still has its original type-inst (e.g. `var int`), because that was its type-inst at instance-time.

Some type-insts can be automatically coerced to another type-inst. For example, if a `par int` value is used in a context where a `var int` is expected, it is automatically coerced to a `var int`. We write this $\text{par int} \xrightarrow{c} \text{var int}$. Also, any type-inst can be considered coercible to itself. MiniZinc allows coercions between some types as well.

Some type-insts can be *varified*, i.e., made unfixed at the top-level. For example, `par int` is varified to `var int`. We write this $\text{par int} \xrightarrow{v} \text{var int}$.

Type-insts that are varifiable include the type-insts of the types that can be decision variables (Booleans, integers, floats, sets, enumerated types). Varification is relevant to type-inst synonyms and array accesses.

4.1.6.4 Type-inst expression overview

This section partly describes how to write type-insts in MiniZinc models. Further details are given for each type as they are described in the following sections.

A type-inst expression specifies a type-inst. Type-inst expressions may include type-inst constraints. Type-inst expressions appear in variable declarations (*Variable Declaration Items* (page 209)) and user-defined operation items (*User-defined Operations* (page 212))..

Type-inst expressions have this syntax:

```

<ti-expr> ::= <base-ti-expr>

<base-ti-expr> ::= <var-par> <base-ti-expr-tail>

<var-par> ::= "var" | "par" | ε

<base-type> ::= "bool"
              | "int"
              | "float"
              | "string"

<base-ti-expr-tail> ::= <ident>
                         | <base-type>

```

```

| <set-ti-expr-tail>
| <ti-variable-expr-tail>
| <array-ti-expr-tail>
| "ann"
| "opt" <base-ti-expr-tail>
| { <expr> "," ... }
| <num-expr> ".." <num-expr>

```

(The final alternative, for range types, uses the numeric-specific `<num-expr>` non-terminal, defined in *Expressions Overview* (page 194), rather than the `<expr>` non-terminal. If this were not the case, the rule would never match because the `..` operator would always be matched by the first `<expr>`.)

This fully covers the type-inst expressions for scalar types. The compound type-inst expression syntax is covered in more detail in *Built-in Compound Types and Type-insts* (page 190).

The `par` and `var` keywords (or lack of them) determine the instantiation. The `par` annotation can be omitted; the following two type-inst expressions are equivalent:

```

par int
int

```

Rationale: The use of the explicit `var` keyword allows an implementation to check that all parameters are initialised in the model or the instance. It also clearly documents which variables are parameters, and allows more precise type-inst checking.

A type-inst is fixed if it does not contain `var` or `any`, with the exception of `ann`.

Note that several type-inst expressions that are syntactically expressible represent illegal type-insts. For example, although the grammar allows `var` in front of all these base type-inst expression tails, it is a type-inst error to have `var` in the front of a string or array expression.

4.1.6.5 Built-in Scalar Types and Type-insts

Booleans

Overview. Booleans represent truthhood or falsity. *Rationale:* Boolean values are not represented by integers. Booleans can be explicitly converted to integers with the `bool2int` function, which makes the user's intent clear.

Allowed Insts. Booleans can be fixed or unfixed.

Syntax. Fixed Booleans are written `bool` or `par bool`. Unfixed Booleans are written as `var bool`.

Finite? Yes. The domain of a Boolean is `false`, `true`.

Variable? `par bool` \xrightarrow{v} `var bool`, `var bool` \xrightarrow{v} `var bool`.

Ordering. The value `false` is considered smaller than `true`.

Initialisation. A fixed Boolean variable must be initialised at instance-time; an unfixed Boolean variable need not be.

Coercions. `par bool` \xrightarrow{c} `var bool`.

Also Booleans can be automatically coerced to integers; see [Integers](#) (page 189).

Integers

Overview. Integers represent integral numbers. Integer representations are implementation-defined. This means that the representable range of integers is implementation-defined. However, an implementation should abort at run-time if an integer operation overflows.

Allowed Insts. Integers can be fixed or unfixed.

Syntax. Fixed integers are written `int` or `par int`. Unfixed integers are written as `var int`.

Finite? Not unless constrained by a set expression (see [Set Expression Type-insts](#) (page 194)).

Varifiable? $\text{par int} \xrightarrow{v} \text{var int}, \text{var int} \xrightarrow{v} \text{var int}$.

Ordering. The ordering on integers is the standard one.

Initialisation. A fixed integer variable must be initialised at instance-time; an unfixed integer variable need not be.

Coercions. $\text{par int} \xrightarrow{c} \text{var int}, \text{par bool} \xrightarrow{c} \text{par int}, \text{par bool} \xrightarrow{c} \text{var int}, \text{var bool} \xrightarrow{c} \text{var int}$.

Also, integers can be automatically coerced to floats; see [Floats](#) (page 189).

Floats

Overview. Floats represent real numbers. Float representations are implementation-defined. This means that the representable range and precision of floats is implementation-defined. However, an implementation should abort at run-time on exceptional float operations (e.g., those that produce NaN, if using IEEE754 floats).

Allowed Insts. Floats can be fixed or unfixed.

Syntax. Fixed floats are written `float` or `par float`. Unfixed floats are written as `var float`.

Finite? Not unless constrained by a set expression (see [Set Expression Type-insts](#) (page 194)).

Varifiable? $\text{par float} \xrightarrow{v} \text{var float}, \text{var float} \xrightarrow{v} \text{var float}$.

Ordering. The ordering on floats is the standard one.

Initialisation. A fixed float variable must be initialised at instance-time; an unfixed float variable need not be.

Coercions. $\text{par int} \xrightarrow{c} \text{par float}, \text{par int} \xrightarrow{c} \text{var float}, \text{var int} \xrightarrow{c} \text{var float}, \text{par float} \xrightarrow{c} \text{var float}$.

Enumerated Types

Overview. Enumerated types (or *enums* for short) provide a set of named alternatives. Each alternative is identified by its *case name*. Enumerated types, like in many other languages, can be used in the place of integer types to achieve stricter type checking.

Allowed Insts. Enums can be fixed or unfixed.

Syntax. Variables of an enumerated type named `X` are represented by the term `X` or `par X` if fixed, and `var X` if unfixed.

Finite? Yes.

The domain of an enum is the set containing all of its case names.

Varifiable? `par X` \xrightarrow{v} `var X`, `var X` \xrightarrow{v} `var X`.

Ordering. When two enum values with different case names are compared, the value with the case name that is declared first is considered smaller than the value with the case name that is declared second.

Initialisation. A fixed enum variable must be initialised at instance-time; an unfixed enum variable need not be.

Coercions. `par X` \xrightarrow{c} `par int`, `var X` \xrightarrow{c} `var int`.

Strings

Overview. Strings are primitive, i.e., they are not lists of characters.

String expressions are used in assertions, output items and annotations, and string literals are used in include items.

Allowed Insts. Strings must be fixed.

Syntax. Fixed strings are written `string` or `par string`.

Finite? Not unless constrained by a set expression (see *Set Expression Type-insts* (page 194)).

Varifiable? No.

Ordering. Strings are ordered lexicographically using the underlying character codes.

Initialisation. A string variable (which can only be fixed) must be initialised at instance-time.

Coercions. None automatic. However, any non-string value can be manually converted to a string using the built-in `show` function or using string interpolation (see *String Literals and String Interpolation* (page 200)).

4.1.6.6 Built-in Compound Types and Type-insts

Sets

Overview. A set is a collection with no duplicates.

Allowed Insts. The type-inst of a set's elements must be fixed. *Rationale:* This is because current solvers are not powerful enough to handle sets containing decision variables. Sets may contain any type, and may be fixed or unfixed. If a set is unfixed, its elements must be finite, unless it occurs in one of the following contexts:

- the argument of a predicate, function or annotation.
- the declaration of a variable or let local variable with an assigned value.

Syntax. A set base type-inst expression tail has this syntax:

```
<set-ti-expr-tail> ::= "set" "of" <base-type>
```

Some example set type-inst expressions:

```
set of int
var set of bool
```

Finite? Yes, if the set elements are finite types. Otherwise, no.

The domain of a set type that is a finite type is the powerset of the domain of its element type. For example, the domain of `set of 1..2` is `powerset(1..2)`, which is `{}`, `{1}`, `{1,2}`, `{2}`.

Variable? `par set of TI` \xrightarrow{v} `var set of TI`, `var set of TI` \xrightarrow{v} `var set of TI`.

Ordering. The pre-defined ordering on sets is a lexicographic ordering of the *sorted set form*, where `{1,2}` is in sorted set form, for example, but `{2,1}` is not. This means, for instance, `{}` < `{1,3}` < `{2}`.

Initialisation. A fixed set variable must be initialised at instance-time; an unfixed set variable need not be.

Coercions. `par set of TI` \xrightarrow{c} `par set of UI` and `par set of TI` \xrightarrow{c} `var set of UI` and `var set of TI` \xrightarrow{c} `var set of UI`, if `TI` \xrightarrow{c} `UI`.

Arrays

Overview. MiniZinc arrays are maps from fixed integers to values. Values can be of any type. The values can only have base type-insts. Arrays-of-arrays are not allowed. Arrays can be multi-dimensional.

MiniZinc arrays can be declared in two different ways.

- *Explicitly-indexed* arrays have index types in the declaration that are finite types. For example:

```
array[0..3] of int: a1;
array[1..5, 1..10] of var float: a5;
```

For such arrays, the index type specifies exactly the indices that will be in the array - the array's index set is the *domain* of the index type - and if the indices of the value assigned do not match then it is a run-time error.

For example, the following assignments cause run-time errors:

```
a1 = [4,6,4,3,2];    % too many elements
a5 = [];              % too few elements
```

- *Implicitly-indexed* arrays have index types in the declaration that are not finite types. For example:

```
array[int,int] of int: a6;
```

No checking of indices occurs when these variables are assigned.

In MiniZinc all index sets of an array must be contiguous ranges of integers, or enumerated types. The expression used for initialisation of an array must have matching index sets. An array expression with an enum index set can be assigned to an array declared with an integer index set, but not the other way around. The exception are array literals, which can be assigned to arrays declared with enum index sets.

For example:

```
enum X = {A,B,C};  
enum Y = {D,E,F};  
array[X] of int: x = array1d(X, [5,6,7]); % correct  
array[Y] of int: y = x; % index set mismatch: Y != X  
array[int] of int: z = x; % correct: assign X index set to  
  ↳ int  
array[X] of int: x2 = [10,11,12]; % correct: automatic coercion for  
  ↳ array literals
```

The initialisation of an array can be done in a separate assignment statement, which may be present in the model or a separate data file.

Arrays can be accessed. See [Array Access Expressions](#) (page 203) for details.

Allowed Insts. An array's size must be fixed. Its indices must also have fixed type-insts. Its elements may be fixed or unfixed.

Syntax. An array base type-inst expression tail has this syntax:

```
<array-ti-expr-tail> ::= "array" [ <ti-expr> "," ... ] "of" <ti-expr>  
  | "list" "of" <ti-expr>
```

Some example array type-inst expressions:

```
array[1..10] of int  
list of var int
```

Note that `list of <T>` is just syntactic sugar for `array[int] of <T>`. *Rationale: Integer-indexed arrays of this form are very common, and so worthy of special support to make things easier for modellers. Implementing it using syntactic sugar avoids adding an extra type to the language, which keeps things simple for implementers.*

Because arrays must be fixed-size it is a type-inst error to precede an array type-inst expression with `var`.

Finite? Yes, if the index types and element type are all finite types. Otherwise, no.

The domain of an array type that is a finite array is the set of all distinct arrays whose index set equals the domain of the index type and whose elements are of the array element type.

Variable? No.

Ordering. Arrays are ordered lexicographically, taking absence of a value for a given key to be before any value for that key. For example, `[1, 1]` is less than `[1, 2]`, which is less than `[1, 2, 3]` and `array1d(2..4,[0, 0, 0])` is less than `[1, 2, 3]`.

Initialisation. An explicitly-indexed array variable must be initialised at instance-time only if its elements must be initialised at instance time. An implicitly-indexed array variable must be initialised at instance-time so that its length and index set is known.

Coercions. $\text{array}[\text{TI0}] \text{ of } \text{TI} \xrightarrow{c} \text{array}[\text{UI0}] \text{ of } \text{UI}$ if $\text{TI0} \xrightarrow{c} \text{UI0}$ and $\text{TI} \xrightarrow{c} \text{UI}$.

Option Types

Overview. Option types defined using the `opt` type constructor, define types that may or may not be there. They are similar to Maybe types of Haskell implicitly adding a new value <> to the type.

Allowed Insts. The argument of an option type must be one of the base types `bool`, `int` or `float`.

Syntax. The option type is written `opt <T>` where `<T>` is one of the three base types, or one of their constrained instances.

Finite? Yes if the underlying type is finite, otherwise no.

Varifiable? Yes.

Ordering. <> is always less than any other value in the type. But beware that overloading of operators like $<$ is different for option types.

Initialisation. An `opt` type variable does not need to be initialised at instance-time. An uninitialised `opt` type variable is automatically initialised to <> .

Coercions. $\text{TI} \xrightarrow{c} \text{opt UI}$ if $\text{TI} \xrightarrow{c} \text{UI}$.

The Annotation Type

Overview. The annotation type, `ann`, can be used to represent arbitrary term structures. It is augmented by annotation items ([Annotation Items](#) (page 212)).

Allowed Insts. `ann` is always considered unfixed, because it may contain unfixed elements. It cannot be preceded by `var`.

Syntax. The annotation type is written `ann`.

Finite? No.

Varifiable? No.

Ordering. N/A. Annotation types do not have an ordering defined on them.

Initialisation. An `ann` variable must be initialised at instance-time.

Coercions. None.

4.1.6.7 Constrained Type-insts

One powerful feature of MiniZinc is *constrained type-insts*. A constrained type-inst is a restricted version of a *base* type-inst, i.e., a type-inst with fewer values in its domain.

Set Expression Type-insts

Three kinds of expressions can be used in type-insts.

1. Integer ranges: e.g. `1..3`.
2. Set literals: e.g. `var {1,3,5}`.
3. Identifiers: the name of a set parameter (which can be global, let-local, the argument of a predicate or function, or a generator value) can serve as a type-inst.

In each case the base type is that of the set's elements, and the values within the set serve as the domain. For example, whereas a variable with type-inst `var int` can take any integer value, a variable with type-inst `var 1..3` can only take the value 1, 2 or 3.

All set expression type-insts are finite types. Their domain is equal to the set itself.

Float Range Type-insts

Float ranges can be used as type-insts, e.g. `1.0 .. 3.0`. These are treated similarly to integer range type-insts, although `1.0 .. 3.0` is not a valid expression whereas `1 .. 3` is.

Float ranges are not finite types.

4.1.7 Expressions

4.1.7.1 Expressions Overview

Expressions represent values. They occur in various kinds of items. They have the following syntax:

```
<expr> ::= <expr-atom> <expr-binop-tail>

<expr-atom> ::= <expr-atom-head> <expr-atom-tail> <annotations>

<expr-binop-tail> ::= "[" <bin-op> <expr> "]"

<expr-atom-head> ::= <builtin-un-op> <expr-atom>
| "(" <expr> ")"
| <ident-or-quoted-op>
| "_"
| <bool-literal>
| <int-literal>
| <float-literal>
| <string-literal>
| <set-literal>
| <set-comp>
| <array-literal>
| <array-literal-2d>
| <array-comp>
| <ann-literal>
```

```

| <if-then-else-expr>
| <let-expr>
| <call-expr>
| <gen-call-expr>

<expr-atom-tail> ::= ε
                     | <array-access-tail> <expr-atom-tail>

```

Expressions can be composed from sub-expressions combined with operators. All operators (binary and unary) are described in [Operators](#) (page 196), including the precedences of the binary operators. All unary operators bind more tightly than all binary operators.

Expressions can have one or more annotations. Annotations bind more tightly than unary and binary operator applications, but less tightly than access operations and non-operator applications. In some cases this binding is non-intuitive. For example, in the first three of the following lines, the annotation `a` binds to the identifier expression `x` rather than the operator application. However, the fourth line features a non-operator application (due to the single quotes around the `not`) and so the annotation binds to the whole application.

```

not x::a;
not (x)::a;
not(x)::a;
'not'(x)::a;

```

[Annotations](#) (page 216) has more on annotations.

Expressions can be contained within parentheses.

The array access operations all bind more tightly than unary and binary operators and annotations. They are described in more detail in [Array Access Expressions](#) (page 203).

The remaining kinds of expression atoms (from `<ident>` to `<gen-call-expr>`) are described in [Identifier Expressions and Quoted Operator Expressions](#) (page 199) to [Generator Call Expressions](#) (page 208).

We also distinguish syntactically valid numeric expressions. This allows range types to be parsed correctly.

```

<num-expr> ::= <num-expr-atom> <num-expr-binop-tail>

<num-expr-atom> ::= <num-expr-atom-head> <expr-atom-tail> <annotations>

<num-expr-binop-tail> ::= "[" <num-bin-op> <num-expr> "]"

<num-expr-atom-head> ::= <builtin-num-un-op> <num-expr-atom>
                         | "(" <num-expr> ")"
                         | <ident-or-quoted-op>
                         | <int-literal>
                         | <float-literal>
                         | <if-then-else-expr>
                         | <let-expr>
                         | <call-expr>

```

```
| <gen-call-expr>
```

4.1.7.2 Operators

Operators are functions that are distinguished by their syntax in one or two ways. First, some of them contain non-alphanumeric characters that normal functions do not (e.g. `+`). Second, their application is written in a manner different to normal functions.

We distinguish between binary operators, which can be applied in an infix manner (e.g. `3 + 4`), and unary operators, which can be applied in a prefix manner without parentheses (e.g. `not x`). We also distinguish between built-in operators and user-defined operators. The syntax is the following:

```
<builtin-op> ::= <builtin-bin-op> | <builtin-un-op>

<bin-op> ::= <builtin-bin-op> | '<ident>'

<builtin-bin-op> ::= "<->" | "->" | "<->" | "\/" | "xor" | "/\\" | "<" | ">" |
→ "<=" | ">=" | "==" | "=" | "!="
→ "in" | "subset" | "superset" | "union" | "diff" |
→ "symdiff"
→ ".." | "intersect" | "++" | <builtin-num-bin-op>

<builtin-un-op> ::= "not" | <builtin-num-un-op>
```

Again, we syntactically distinguish numeric operators.

```
<num-bin-op> ::= <builtin-num-bin-op> | '<ident>'

<builtin-num-bin-op> ::= "+" | "-" | "*" | "/" | "div" | "mod"

<builtin-num-un-op> ::= "+" | "-"
```

Some operators can be written using their unicode symbols, which are listed in Table 4.1.7.2 (recall that MiniZinc input is UTF-8).

Table 1.1: Unicode equivalents of binary operators

Operator	Unicode symbol	UTF-8 code
<code><-></code>	\leftrightarrow	E2 86 94
<code>-></code>	\rightarrow	E2 86 92
<code><-</code>	\leftarrow	E2 86 90
<code>not</code>	\neg	C2 AC
<code>\vee</code>	\vee	E2 88 A8
<code>\wedge</code>	\wedge	E2 88 A7
<code>!=</code>	\neq	E2 89 A0
<code><=</code>	\leq	E2 89 A4
<code>>=</code>	\geq	E2 89 A5
<code>in</code>	\in	E2 88 88
<code>subset</code>	\subseteq	E2 8A 86
<code>superset</code>	\supseteq	E2 8A 87
<code>union</code>	\cup	E2 88 AA
<code>intersect</code>	\cap	E2 88 A9

The binary operators are listed in Table 4.1.7.2. A lower precedence number means tighter binding; for example, `1+2*3` is parsed as `1+(2*3)` because `*` binds tighter than `+`. Associativity indicates how chains of operators with equal precedences are handled; for example, `1+2+3` is parsed as `(1+2)+3` because `+` is left-associative, `a++b++c` is parsed as `a++(b++c)` because `++` is right-associative, and `1<x<2` is a syntax error because `<` is non-associative.

Table 1.2: Binary infix operators

Symbol(s)	Assoc.	Prec.
<code><-></code>	left	1200
<code>-></code>	left	1100
<code><-</code>	left	1100
<code>\/</code>	left	1000
<code>xor</code>	left	1000
<code>/\</code>	left	900
<code><</code>	none	800
<code>></code>	none	800
<code><=</code>	none	800
<code>>=</code>	none	800
<code>==,</code>		
<code>=</code>	none	800
<code>!=</code>	none	800
<code>in</code>	none	700
<code>subset</code>	none	700
<code>superset</code>	none	700
<code>union</code>	left	600
<code>diff</code>	left	600
<code>symdiff</code>	left	600
<code>..</code>	none	500
<code>+</code>	left	400
<code>-</code>	left	400
<code>*</code>	left	300
<code>div</code>	left	300
<code>mod</code>	left	300
<code>/</code>	left	300
<code>intersect</code>	left	300
<code>++</code>	right	200
<code>`<ident>`</code>	left	100

A user-defined binary operator is created by backquoting a normal identifier, for example:

```
A `min2` B
```

This is a static error if the identifier is not the name of a binary function or predicate.

The unary operators are: `+`, `-` and `not`. User-defined unary operators are not possible.

As [Identifiers](#) (page 183) explains, any built-in operator can be used as a normal function identifier by quoting it, e.g: `'+'(3, 4)` is equivalent to `3 + 4`.

The meaning of each operator is given in [Built-in Operations](#) (page 219).

4.1.7.3 Expression Atoms

Identifier Expressions and Quoted Operator Expressions

Identifier expressions and quoted operator expressions have the following syntax:

```
<ident-or-quoted-op> ::= <ident>
| '<builtin-op>'
```

Examples of identifiers were given in [Identifiers](#) (page 183). The following are examples of quoted operators:

```
'+'
'union'
```

In quoted operators, whitespace is not permitted between either quote and the operator. [Operators](#) (page 196) lists MiniZinc's built-in operators.

Syntactically, any identifier or quoted operator can serve as an expression. However, in a valid model any identifier or quoted operator serving as an expression must be the name of a variable.

Anonymous Decision Variables

There is a special identifier, `_`, that represents an unfixed, anonymous decision variable. It can take on any type that can be a decision variable. It is particularly useful for initialising decision variables within compound types. For example, in the following array the first and third elements are fixed to 1 and 3 respectively and the second and fourth elements are unfixed:

```
array[1..4] of var int: xs = [1, _, 3, _];
```

Any expression that does not contain `_` and does not involve decision variables is fixed.

Boolean Literals

Boolean literals have this syntax:

```
<bool-literal> ::= "false" | "true"
```

Integer and Float Literals

There are three forms of integer literals - decimal, hexadecimal, and octal - with these respective forms:

```
<int-literal> ::= [0-9]+
| 0x[0-9A-Fa-f]+
| 0o[0-7]+
```

For example: `0`, `005`, `123`, `0x1b7`, `0o777`; but not `-1`.

Float literals have the following form:

```
<float-literal> ::= [0-9]+.[0-9]+  
                  | [0-9]+.[0-9]+[Ee][-+]?[0-9]+  
                  | [0-9]+[Ee][-+]?[0-9]+
```

For example: `1.05`, `1.3e-5`, `1.3e+5`; but not `1.`, `.5`, `1.e5`, `.1e5`, `-1.0`, `-1E05`. A `-` symbol preceding an integer or float literal is parsed as a unary minus (regardless of intervening whitespace), not as part of the literal. This is because it is not possible in general to distinguish a `-` for a negative integer or float literal from a binary minus when lexing.

String Literals and String Interpolation

String literals are written as in C:

```
<string-contents> ::= ([^\n\] | \[^n\])*  
  
<string-literal> ::= """ <string-contents> """  
                     | """ <string-contents> "\(" <string-interpolate-tail>  
  
<string-interpolate-tail> ::= <expr> ")" <string-contents> """  
                           | <expr> ")" <string-contents> "\(" _  
                           ↴<string-interpolate-tail>
```

This includes C-style escape sequences, such as `\"` for double quotes, `\\\` for backslash, and `\n` for newline.

For example: `"Hello, world!\n"`.

String literals must fit on a single line.

Long string literals can be split across multiple lines using string concatenation. For example:

```
string: s = "This is a string literal "  
       ++ "split across two lines.";
```

A string expression can contain an arbitrary MiniZinc expression, which will be converted to a string similar to the builtin `show` function and inserted into the string.

For example:

```
var set of 1..10: q;  
solve satisfy;  
output [show("The value of q is \$(q), and it has \$(card(q)) elements.")];
```

Set Literals

Set literals have this syntax:

```
<set-literal> ::= "{" [ <expr> "," ... ] "}"
```

For example:

```
{ 1, 3, 5 }
{ }
{ 1, 2.0 }
```

The type-insts of all elements in a literal set must be the same, or coercible to the same type-inst (as in the last example above, where the integer 1 will be coerced to a float).

Set Comprehensions

Set comprehensions have this syntax:

```
<set-comp> ::= "{" <expr> " | " <comp-tail> "}"
<comp-tail> ::= <generator> [ "where" <expr> ] " , " ...
<generator> ::= <ident> " , " ... "in" <expr>
```

For example (with the literal equivalent on the right):

```
{ 2*i | i in 1..5 }      % { 2, 4, 6, 8, 10 }
{ 1 | i in 1..5 }      % { 1 }  (no duplicates in sets)
```

The expression before the | is the *head expression*. The expression after the in is a *generator expression*. Generators can be restricted by a *where-expression*. For example:

```
{ i | i in 1..10 where (i mod 2 = 0) }      % { 2, 4, 6, 8, 10 }
```

When multiple generators are present, the right-most generator acts as the inner-most one. For example:

```
{ 3*i+j | i in 0..2, j in {0, 1} }      % { 0, 1, 3, 4, 6, 7 }
```

The scope of local generator variables is given by the following rules:

- They are visible within the head expression (before the |).
- They are visible within the where-expression of their own generator.
- They are visible within generator expressions and where-expressions in any subsequent generators.

The last of these rules means that the following set comprehension is allowed:

```
{ i+j | i in 1..3, j in 1..i } % { 1+1, 2+1, 2+2, 3+1, 3+2, 3+3 }
```

Multiple where-expressions are allowed, as in the following example:

```
[f(i, j) | i in A1 where p(i), j in A2 where q(i,j)]
```

A generator expression must be an array or a fixed set.

Rationale: For set comprehensions, set generators would suffice, but for array comprehensions, array generators are required for full expressivity (e.g., to provide control over the order of the elements in the resulting array). Set comprehensions have array generators for consistency with array comprehensions, which makes implementations simpler.

The where-expression (if present) must be Boolean. It can be var, in which case the type of the comprehension is lifted to an optional type.

Array Literals

Array literals have this syntax:

```
<array-literal> ::= "[" [ <expr> "," ... ] "]"
```

For example:

```
[1, 2, 3, 4]
[]
[1, _]
```

In a array literal all elements must have the same type-inst, or be coercible to the same type-inst (as in the last example above, where the fixed integer 1 will be coerced to a var int).

The indices of a array literal are implicitly 1..n, where n is the length of the literal.

2d Array Literals

Simple 2d array literals have this syntax:

```
<array-literal-2d> ::= "[" | " [ (<expr> "," ...) "|" ... ] "|" ]"
```

For example:

```
[| 1, 2, 3
| 4, 5, 6
| 7, 8, 9 |]      % array[1..3, 1..3]
[| x, y, z |]    % array[1..1, 1..3]
[| 1 | _ | _ |]  % array[1..3, 1..1]
```

In a 2d array literal, every sub-array must have the same length.

In a 2d array literal all elements must have the same type-inst, or be coercible to the same type-inst (as in the last example above, where the fixed integer 1 will be coerced to a `var int`).

The indices of a 2d array literal are implicitly $(1, 1) \dots (m, n)$, where m and n are determined by the shape of the literal.

Array Comprehensions

Array comprehensions have this syntax:

```
<array-comp> ::= "[" <expr> "|" <comp-tail> "]"
```

For example (with the literal equivalents on the right):

```
[2*i | i in 1..5] % [2, 4, 6, 8, 10]
```

Array comprehensions have more flexible type and inst requirements than set comprehensions (see [Set Comprehensions](#) (page 201)).

Array comprehensions are allowed over a variable set with finite type, the result is an array of optional type, with length equal to the cardinality of the upper bound of the variable set. For example:

```
var set of 1..5: x;
array[int] of var opt int: y = [i * i | i in x];
```

The length of array will be 5.

Array comprehensions are allowed where the where-expression is a `var bool`. Again the resulting array is of optional type, and of length equal to that given by the generator expressions. For example:

```
var int x;
array[int] of var opt int: y = [i | i in 1..10 where i != x];
```

The length of the array will be 10.

The indices of an evaluated simple array comprehension are implicitly $1 \dots n$, where n is the length of the evaluated comprehension.

Array Access Expressions

Array elements are accessed using square brackets after an expression:

```
<array-access-tail> ::= "[" <expr> "," ... "]"
```

For example:

```
int: x = a1[1];
```

If all the indices used in an array access are fixed, the type-inst of the result is the same as the element type-inst. However, if any indices are not fixed, the type-inst of the result is the varified element type-inst. For example, if we have:

```
array[1..2] of int: a2 = [1, 2];
var int: i;
```

then the type-inst of `a2[i]` is `var int`. If the element type-inst is not varifiable, such an access causes a static error.

Multi dimensional arrays are accessed using comma separated indices.

```
array[1..3,1..3] of int: a3;
int: y = a3[1, 2];
```

Indices must match the index set type of the array. For example, an array declared with an enum index set can only be accessed using indices from that enum.

```
enum X = {A,B,C};
array[X] of int: a4 = [1,2,3];
int: y = a4[1];           % wrong index type
int: z = a4[B];          % correct
```

Array Slice Expressions

Arrays can be *sliced* in order to extract individual rows, columns or blocks. The syntax is that of an array access expression (see above), but where one or more of the expressions inside the square brackets are set-valued.

For example, the following extracts row 2 from a two-dimensional array:

```
array[1..n,4..8] of int: x;
array[int] of int: row_2_of_x = x[2,4..8];
```

Note that the resulting array `row_2_of_x` will have index set `4..8`.

A short-hand for all indices of a particular dimension is to use just dots:

```
array[1..n,4..8] of int: x;
array[int] of int: row_2_of_x = x[2,..];
```

You can also restrict the index set by giving a sub-set of the original index set as the slice:

```
array[1..n,4..8] of int: x;
array[int] of int: row_2_of_x = x[2,5..6];
```

The resulting array `row_2_of_x` will now have length 2 and index set 5..6.

The dots notation also allows for partial bounds, for example:

```
array[1..n, 4..8] of int: x;
array[int] of int: row_2_of_x = x[2,..6];
```

The resulting array will have length 3 and index set 4..6. Of course 6.. would also be allowed and result in an array with index set 6..8.

Annotation Literals

Literals of the `ann` type have this syntax:

```
<ann-literal> ::= <ident> [ "(" <expr> "," ... ")" ]
```

For example:

```
foo
cons(1, cons(2, cons(3, nil)))
```

There is no way to inspect or deconstruct annotation literals in a MiniZinc model; they are intended to be inspected only by an implementation, e.g., to direct compilation.

If-then-else Expressions

MiniZinc provides if-then-else expressions, which provide selection from two alternatives based on a condition. They have this syntax:

```
<if-then-else-expr> ::= "if" <expr> "then" <expr> [ "elseif" <expr> "then"_
↳<expr> ]* "else" <expr> "endif"
```

For example:

```
if x <= y then x else y endif
if x < 0 then -1 elseif x > 0 then 1 else 0 endif
```

The presence of the `endif` avoids possible ambiguity when an if-then-else expression is part of a larger expression.

The type-inst of the `if` expression must be `par bool` or `var bool`. The `then` and `else` expressions must have the same type-inst, or be coercible to the same type-inst, which is also the type-inst of the whole expression.

If the `if` expression is `var bool` then the type-inst of the `then` and `else` expressions must be varifiable.

If the `if` expression is `par bool` then evaluation of if-then-else expressions is lazy: the condition is evaluated, and then only one of the `then` and `else` branches are evaluated, depending on whether the condition succeeded or failed. This is not the case if it is `var bool`.

Let Expressions

Let expressions provide a way of introducing local names for one or more expressions and local constraints that can be used within another expression. They are particularly useful in user-defined operations.

Let expressions have this syntax:

```
<let-expr> ::= "let" "{" <let-item> ";" ... "}" "in" <expr>
<let-item> ::= <var-decl-item>
            | <constraint-item>
```

For example:

```
let { int: x = 3; int: y = 4; } in x + y;
let { var int: x;
      constraint x >= y /\ x >= -y /\ (x = y \/\ x = -y); }
in x
```

The scope of a let local variable covers:

- The type-inst and initialisation expressions of any subsequent variables within the let expression (but not the variable's own initialisation expression).
- The expression after the `in`, which is parsed as greedily as possible.

A variable can only be declared once in a let expression.

Thus in the following examples the first is acceptable but the rest are not:

```
let { int: x = 3; int: y = x; } in x + y; % ok
let { int: y = x; int: x = 3; } in x + y; % x not visible in y's defn.
let { int: x = x; } in x;                  % x not visible in x's defn.
let { int: x = 3; int: x = 4; } in x;       % x declared twice
```

The initialiser for a let local variable can be omitted only if the variable is a decision variable. For example:

```
let { var int: x; } in ...;    % ok
let {     int: x; } in ...;    % illegal
```

The type-inst of the entire let expression is the type-inst of the expression after the `in` keyword.

There is a complication involving let expressions in negative contexts. A let expression occurs in a negative context if it occurs in an expression of the form `not X`, `X <-> Y` or in the sub-expression `X` in `X -> Y` or `Y <- X`, or in a subexpression `bool2int(X)`.

If a let expression is used in a negative context, then any let-local decision variables must be defined only in terms of non-local variables and parameters. This is because local variables are implicitly existentially quantified, and if the let expression occurred in a negative context then the local variables would be effectively universally quantified which is not supported by MiniZinc.

Constraints in let expressions float to the nearest enclosing Boolean context. For example

```
constraint b -> x + let { var 0..2: y; constraint y != -1;} in y >= 4;
```

is equivalent to

```
var 0..2: y;
constraint b -> (x + y >= 4 /\ y != 1);
```

For backwards compatibility with older versions of MiniZinc, items inside the let can also be separated by commas instead of semicolons.

Call Expressions

Call expressions are used to call predicates and functions.

Call expressions have this syntax:

```
<call-expr> ::= <ident-or-quoted-op> [ "(" <expr> "," ... ")" ]
```

For example:

```
x = min(3, 5);
```

The type-insts of the expressions passed as arguments must match the argument types of the called predicate/function. The return type of the predicate/function must also be appropriate for the calling context.

Note that a call to a function or predicate with no arguments is syntactically indistinguishable from the use of a variable, and so must be determined during type-inst checking.

Evaluation of the arguments in call expressions is strict: all arguments are evaluated before the call itself is evaluated. Note that this includes Boolean operations such as `/\`, `\/`, `->` and `<-` which could be lazy in one argument. The one exception is `assert`, which is lazy in its third argument (*Other Operations* (page 226)).

Rationale: Boolean operations are strict because: (a) this minimises exceptional cases; (b) in an expression like `A -> B` where `A` is not fixed and `B` causes an abort, the appropriate behaviour is unclear if laziness is present; and (c) if a user needs laziness, an if-then-else can be used.

The order of argument evaluation is not specified. *Rationale:* Because MiniZinc is declarative, there is no obvious need to specify an evaluation order, and leaving it unspecified gives implementors some freedom.

Generator Call Expressions

MiniZinc has special syntax for certain kinds of call expressions which makes models much more readable.

Generator call expressions have this syntax:

```
<gen-call-expr> ::= <ident-or-quoted-op> "(" <comp-tail> ")" "(" <expr> ")"
```

A generator call expression $P(Gs)(E)$ is equivalent to the call expression $P([E \mid Gs])$. For example, the expression:

```
forall(i,j in Domain where i<j)
  (noattack(i, j, queens[i], queens[j]));
```

(in a model specifying the N-queens problem) is equivalent to:

```
forall( [ noattack(i, j, queens[i], queens[j])
  | i,j in Domain where i<j ] );
```

The parentheses around the latter expression are mandatory; this avoids possible confusion when the generator call expression is part of a larger expression.

The identifier must be the name of a unary predicate or function that takes an array argument.

The generators and where-expression (if present) have the same requirements as those in array comprehensions ([Array Comprehensions](#) (page 203)).

4.1.8 Items

This section describes the top-level program items.

4.1.8.1 Include Items

Include items allow a model to be split across multiple files. They have this syntax:

```
<include-item> ::= "include" <string-literal>
```

For example:

```
include "foo.mzn";
```

includes the file `foo.mzn`.

Include items are particularly useful for accessing libraries or breaking up large models into small pieces. They are not, as [Model Instance Files](#) (page 185) explains, used for specifying data files.

If the given name is not a complete path then the file is searched for in an implementation-defined set of directories. The search directories must be able to be altered with a command line option.

4.1.8.2 Variable Declaration Items

Variable declarations have this syntax:

```
<var-decl-item> ::= <ti-expr-and-id> <annotations> [ "=" <expr> ]
```

For example:

```
int: A = 10;
```

It is a type-inst error if a variable is declared and/or defined more than once in a model.

A variable whose declaration does not include an assignment can be initialised by a separate assignment item ([Assignment Items](#) (page 210)). For example, the above item can be separated into the following two items:

```
int: A;
...
A = 10;
```

All variables that contain a parameter component must be defined at instance-time.

Variables can have one or more annotations. [Annotations](#) (page 216) has more on annotations.

4.1.8.3 Enum Items

Enumerated type items have this syntax:

```
<enum-item> ::= "enum" <ident> <annotations> [ "=" <enum-cases> ]
<enum-cases> ::= "{" <ident> " , " ... " }"
```

An example of an enum:

```
enum country = {Australia, Canada, China, England, USA};
```

Each alternative is called an *enum case*. The identifier used to name each case (e.g. *Australia*) is called the *enum case name*.

Because enum case names all reside in the top-level namespace ([Namespaces](#) (page 185)), case names in different enums must be distinct.

An enum can be declared but not defined, in which case it must be defined elsewhere within the model, or in a data file. For example, a model file could contain this:

```
enum Workers;  
enum Shifts;
```

and the data file could contain this:

```
Workers = { welder, driller, stamper };  
Shifts = { idle, day, night };
```

Sometimes it is useful to be able to refer to one of the enum case names within the model. This can be achieved by using a variable. The model would read:

```
enum Shifts;  
Shifts: idle; % Variable representing the idle constant.
```

and the data file:

```
enum Shifts = { idle_const, day, night };  
idle = idle_const; % Assignment to the variable.
```

Although the constant `idle_const` cannot be mentioned in the model, the variable `idle` can be. All enums must be defined at instance-time.

Enum items can be annotated. [Annotations](#) (page 216) has more details on annotations.

Each case name can be coerced automatically to the integer corresponding to its index in the type.

```
int: oz = Australia; % oz = 1
```

For each enumerated type `T`, the following functions exist:

```
% Return next greater enum value of x in enum type X  
function T: enum_next(set of T: X, T: x);  
function var T: enum_next(set of T: X, var T: x);  
  
% Return next smaller enum value of x in enum type X  
function T: enum_prev(set of T: X, T: x);  
function var T: enum_prev(set of T: X, var T: x);  
  
% Convert x to enum type X  
function T: to_enum(set of T: X, int: x);  
function var T: to_enum(set of T: X, var int: x);
```

4.1.8.4 Assignment Items

Assignments have this syntax:

```
<assign-item> ::= <ident> "=" <expr>
```

For example:

```
A = 10;
```

4.1.8.5 Constraint Items

Constraint items form the heart of a model. Any solutions found for a model will satisfy all of its constraints.

Constraint items have this syntax:

```
<constraint-item> ::= "constraint" <string-annotation> <expr>
```

For example:

```
constraint a*x < b;
```

The expression in a constraint item must have type-inst `par bool` or `var bool`; note however that constraints with fixed expressions are not very useful.

4.1.8.6 Solve Items

Every model must have exactly one or no solve item. Solve items have the following syntax:

```
<solve-item> ::= "solve" <annotations> "satisfy"
               | "solve" <annotations> "minimize" <expr>
               | "solve" <annotations> "maximize" <expr>
```

Example solve items:

```
solve satisfy;
solve maximize a*x + y - 3*z;
```

The solve item determines whether the model represents a constraint satisfaction problem or an optimisation problem. If there is no solve item, the model is assumed to be a satisfaction problem. For optimisation problems, the given expression is the one to be minimized/maximized.

The expression in a minimize/maximize solve item can have integer or float type.

Rationale: This is possible because all type-insts have a defined order. Note that having an expression with a fixed type-inst in a solve item is not very useful as it means that the model requires no optimisation.

Solve items can be annotated. [Annotations](#) (page 216) has more details on annotations.

4.1.8.7 Output Items

Output items are used to present the solutions of a model instance. They have the following syntax:

```
<output-item> ::= "output" <expr>
```

For example:

```
output ["The value of x is ", show(x), "!\\n"];
```

The expression must have type-inst `array[int] of par string`. It can be composed using the built-in operator `++` and the built-in functions `show`, `show_int`, and `show_float` (*Built-in Operations* (page 219)), as well as string interpolations (*String Literals and String Interpolation* (page 200)). The output is the concatenation of the elements of the array. If multiple output items exist, the output is the concatenation of all of their outputs, in the order in which they appear in the model.

If no output item is present, the implementation should print all the global variables and their values in a readable format.

4.1.8.8 Annotation Items

Annotation items are used to augment the `ann` type. They have the following syntax:

```
<annotation-item> ::= "annotation" <ident> <params>
```

For example:

```
annotation solver(int: kind);
```

It is a type-inst error if an annotation is declared and/or defined more than once in a model.

The use of annotations is described in *Annotations* (page 216).

4.1.8.9 User-defined Operations

MiniZinc models can contain user-defined operations. They have this syntax:

```
<predicate-item> ::= "predicate" <operation-item-tail>  
  
<test-item> ::= "test" <operation-item-tail>  
  
<function-item> ::= "function" <ti-expr> ":" <operation-item-tail>  
  
<operation-item-tail> ::= <ident> <params> <annotations> [ "=" <expr> ]
```

```
<params> ::= [ ( <ti-expr-and-id> ", " ... ) ]
```

The type-inst expressions can include type-inst variables in the function and predicate declaration.

For example, predicate even checks that its argument is an even number.

```
predicate even(var int: x) =
  x mod 2 = 0;
```

A predicate supported natively by the target solver can be declared as follows:

```
predicate alldifferent(array [int] of var int: xs);
```

Predicate declarations that are natively supported in MiniZinc are restricted to using FlatZinc types (for instance, multi-dimensional and non-1-based arrays are forbidden). .. % pjs{need to fix this if we allow 2d arrays in FlatZinc!}

Declarations for user-defined operations can be annotated. *Annotations* (page 216) has more details on annotations.

Basic Properties

The term “predicate” is generally used to refer to both test items and predicate items. When the two kinds must be distinguished, the terms “test item” and “predicate item” can be used.

The return type-inst of a test item is implicitly `par bool`. The return type-inst of a predicate item is implicitly `var bool`.

Predicates and functions are allowed to be recursive. Termination of a recursive function call depends solely on its fixed arguments, i.e., recursive functions and predicates cannot be used to define recursively constrained variables. .. % Rationale{This ensures that the satisfiability of models is decidable.}

Predicates and functions introduce their own local names, being those of the formal arguments. The scope of these names covers the predicate/function body. Argument names cannot be repeated within a predicate/function declaration.

Ad-hoc polymorphism

MiniZinc supports ad-hoc polymorphism via overloading. Functions and predicates (both built-in and user-defined) can be overloaded. A name can be overloaded as both a function and a predicate.

It is a type-inst error if a single version of an overloaded operation with a particular type-inst signature is defined more than once in a model. For example:

```
predicate p(1..5: x);
predicate p(1..5: x) = false;      % ok:      first definition
predicate p(1..5: x) = true;       % error:   repeated definition
```

The combination of overloading and coercions can cause problems. Two overloadings of an operation are said to *overlap* if they could match the same arguments. For example, the following overloading of `p` overlap, as they both match the call `p(3)`.

```
predicate p(par int: x);
predicate p(var int: x);
```

However, the following two predicates do not overlap because they cannot match the same argument:

```
predicate q(int:      x);
predicate q(set of int: x);
```

We avoid two potential overloading problems by placing some restrictions on overlapping overloading of operations.

1. The first problem is ambiguity. Different placement of coercions in operation arguments may allow different choices for the overloaded function. For instance, if a MiniZinc function `f` is overloaded like this:

```
function int: f(int: x, float: y) = 0;
function int: f(float: x, int: y) = 1;
```

then `f(3,3)` could be either 0 or 1 depending on coercion/overloading choices.

To avoid this problem, any overlapping overloading of an operation must be semantically equivalent with respect to coercion. For example, the two overloading of the predicate `p` above must have bodies that are semantically equivalent with respect to overloading.

Currently, this requirement is not checked and the modeller must satisfy it manually. In the future, we may require the sharing of bodies among different versions of overloaded operations, which would provide automatic satisfaction of this requirement.

2. The second problem is that certain combinations of overloading could require a MiniZinc implementation to perform combinatorial search in order to explore different choices of coercions and overloading. For example, if function `g` is overloaded like this:

```
function float: g(int: t1, float: t2) = t2;
function int : g(float: t1, int: t2) = t1;
```

then how the overloading of `g(3,4)` is resolved depends upon its context:

```
float: s = g(3,4);
int: t = g(3,4);
```

In the definition of `s` the first overloaded definition must be used while in the definition of `t` the second must be used.

To avoid this problem, all overlapping overloading of an operation must be closed under intersection of their input type-insts. That is, if overloaded versions have input type-inst

(S_1, \dots, S_n) and (T_1, \dots, T_n) then there must be another overloaded version with input type-inst (R_1, \dots, R_n) where each R_i is the greatest lower bound (*glb*) of S_i and T_i .

Also, all overlapping overloading of an operation must be monotonic. That is, if there are overloaded versions with input type-insts (S_1, \dots, S_n) and (T_1, \dots, T_n) and output type-inst S and T , respectively, then $S_i \preceq T_i$ for all i , implies $S \preceq T$. At call sites, the matching overloading that is lowest on the type-inst lattice is always used.

For `g` above, the type-inst intersection (or *glb*) of `(int, float)` and `(float, int)` is `(int, int)`. Thus, the overloaded versions are not closed under intersection and the user needs to provide another overloading for `g` with input type-inst `(int, int)`. The natural definition is:

```
function int: g(int: t1, int: t2) = t1;
```

Once `g` has been augmented with the third overloading, it satisfies the monotonicity requirement because the output type-inst of the third overloading is `int` which is less than the output type-inst of the original overloading.

Monotonicity and closure under type-inst conjunction ensure that whenever an overloaded function or predicate is reached during type-inst checking, there is always a unique and safe “minimal” version to choose, and so the complexity of type-inst checking remains linear. Thus in our example `g(3,4)` is always resolved by choosing the new overloaded definition.

Local Variables

Local variables in operation bodies are introduced using let expressions. For example, the predicate `have_common_divisor` takes two integer values and checks whether they have a common divisor greater than one:

```
predicate have_common_divisor(int: A, int: B) =
  let {
    var 2..min(A,B): C;
  } in
  A mod C = 0 /\ B mod C = 0;
```

However, as *Let Expressions* (page 206) explained, because `C` is not defined, this predicate cannot be called in a negative context. The following is a version that could be called in a negative context:

```
predicate have_common_divisor(int: A, int: B) =
  exists(C in 2..min(A,B))
  (A mod C = 0 /\ B mod C = 0);
```

4.1.9 Annotations

Annotations allow a modeller to specify non-declarative and solver-specific information that is beyond the core language. Annotations do not change the meaning of a model, however, only how it is solved.

Annotations can be attached to variables (on their declarations), constraints, expressions, type-inst synonyms, enum items, solve items and on user defined operations. They have the following syntax:

```
<annotations> ::= [ ":" <annotation> ]*
<annotation> ::= <expr-atom-head> <expr-atom-tail>
<string-annotation> ::= ":" <string-literal>
```

For example:

```
int: x::foo;
x = (3 + 4)::bar("a", 9)::baz("b");
solve :: blah(4)
minimize x;
```

The types of the argument expressions must match the argument types of the declared annotation. Like user-defined predicates and functions, annotations can be overloaded.

Annotation signatures can contain type-inst variables.

The order and nesting of annotations do not matter. For the expression case it can be helpful to view the annotation connector `::` as an overloaded operator:

```
ann: ':':'(any $T: e, ann: a);           % associative
ann: ':':'(ann:     a, ann: b);           % associative + commutative
```

Both operators are associative, the second is commutative. This means that the following expressions are all equivalent:

```
e :: a :: b
e :: b :: a
(e :: a) :: b
(e :: b) :: a
e :: (a :: b)
e :: (b :: a)
```

This property also applies to annotations on solve items and variable declaration items. *Rationale:* *This property make things simple, as it allows all nested combinations of annotations to be treated as if they are flat, thus avoiding the need to determine what is the meaning of an annotated annotation. It also makes the MiniZinc abstract syntax tree simpler by avoiding the need to represent nesting.*

Annotations have to be values of the `ann` type or string literals. The latter are used for *naming* constraints and expressions, for example

```
constraint :: "first constraint" alldifferent(x);
constraint :: "second constraint" alldifferent(y);
constraint forall (i in 1..n) (my_constraint(x[i],y[i])):: "constraint \$(i)";
```

Note that constraint items can *only* be annotated with string literals.

Rationale: *Allowing arbitrary annotations on constraint items makes the grammar ambiguous, and seems unnecessary since we can just as well annotate the constraint expression.*

4.1.10 Partiality

The presence of constrained type-insts in MiniZinc means that various operations are potentially *partial*, i.e., not clearly defined for all possible inputs. For example, what happens if a function expecting a positive argument is passed a negative argument? What happens if a variable is assigned a value that does not satisfy its type-inst constraints? What happens if an array index is out of bounds? This section describes what happens in all these cases.

In general, cases involving fixed values that do not satisfy constraints lead to run-time aborts.
Rationale: *Our experience shows that if a fixed value fails a constraint, it is almost certainly due to a programming error. Furthermore, these cases are easy for an implementation to check.*

But cases involving unfixed values vary, as we will see. *Rationale:* *The best thing to do for unfixed values varies from case to case. Also, it is difficult to check constraints on unfixed values, particularly because during search a decision variable might become fixed and then backtracking will cause this value to be reverted, in which case aborting is a bad idea.*

4.1.10.1 Partial Assignments

The first operation involving partiality is assignment. There are four distinct cases for assignments.

- A value assigned to a fixed, constrained global variable is checked at run-time; if the assigned value does not satisfy its constraints, it is a run-time error. In other words, this:

```
1..5: x = 3;
```

is equivalent to this:

```
int: x = 3;
constraint assert(x in 1..5,
    "assignment to global parameter 'x' failed")
```

- A value assigned to an unfixed, constrained global variable makes the assignment act like a constraint; if the assigned value does not satisfy the variable's constraints, it causes a run-time model failure. In other words, this:

```
var 1..5: x = 3;
```

is equivalent to this:

```
var int: x = 3;
constraint x in 1..5;
```

Rationale: This behaviour is easy to understand and easy to implement.

- A value assigned to a fixed, constrained let-local variable is checked at run-time; if the assigned value does not satisfy its constraints, it is a run-time error. In other words, this:

```
let { 1..5: x = 3; } in x+1
```

is equivalent to this:

```
let { int: x = 3; } in
  assert(x in 1..5,
         "assignment to let parameter 'x' failed", x+1)
```

- A value assigned to an unfixed, constrained let-local variable makes the assignment act like a constraint; if the constraint fails at run-time, the failure “bubbles up” to the nearest enclosing Boolean scope, where it is interpreted as false.

Rationale: This behaviour is consistent with assignments to global variables.

Note that in cases where a value is partly fixed and partly unfixed, e.g., some arrays, the different elements are checked according to the different cases, and fixed elements are checked before unfixed elements. For example:

```
u = [ let { var 1..5: x = 6 } in x, let { par 1..5: y = 6; } in y ];
```

This causes a run-time abort, because the second, fixed element is checked before the first, unfixed element. This ordering is true for the cases in the following sections as well. *Rationale:* This ensures that failures cannot mask aborts, which seems desirable.

4.1.10.2 Partial Predicate/Function and Annotation Arguments

The second kind of operation involving partiality is calls and annotations.

The semantics is similar to assignments: fixed arguments that fail their constraints will cause aborts, and unfixed arguments that fail their constraints will cause failure, which bubbles up to the nearest enclosing Boolean scope.

4.1.10.3 Partial Array Accesses

The third kind of operation involving partiality is array access. There are two distinct cases.

- A fixed value used as an array index is checked at run-time; if the index value is not in the index set of the array, it is a run-time error.
- An unfixed value used as an array index makes the access act like a constraint; if the access fails at run-time, the failure “bubbles up” to the nearest enclosing Boolean scope, where it is interpreted as false. For example:

```
array[1..3] of int: a = [1,2,3];
var int: i;
constraint (a[i] + 3) > 10 \vee i = 99;
```

Here the array access fails, so the failure bubbles up to the disjunction, and `i` is constrained to be 99. *Rationale: Unlike predicate/function calls, modellers in practice sometimes do use array accesses that can fail. In such cases, the “bubbling up” behaviour is a reasonable one.*

4.1.11 Built-in Operations

This appendix lists built-in operators, functions and predicates. They may be implemented as true built-ins, or in libraries that are automatically imported for all models. Many of them are overloaded.

Operator names are written within single quotes when used in type signatures, e.g. `bool: '\<'(bool, bool)`.

We use the syntax `TI: f(TI1, ..., TIn)` to represent an operation named `f` that takes arguments with type-insts `TI, ..., TIn` and returns a value with type-inst `TI`. This is slightly more compact than the usual MiniZinc syntax, in that it omits argument names.

4.1.11.1 Comparison Operations

Less than. Other comparisons are similar: greater than (`>`), less than or equal (`<=`), greater than or equal (`>=`), equality (`==`, `=`), and disequality (`!=`).

```
bool: '<'(      $T,      $T)
var bool: '<'(var $T, var $T)
```

4.1.11.2 Arithmetic Operations

Addition. Other numeric operations are similar: subtraction (`-`), and multiplication (`*`).

```
int: '+'(      int,      int)
var int: '+'(var int,  var int)
float: '+'(    float,    float)
var float: '+'(var float, var float)
```

Unary minus. Unary plus (`+`) is similar.

```

int: '-'(    int)
var int: '-'(var int)
float: '-'(   float)
var float: '-'(var float)

```

Integer and floating-point division and modulo.

```

int: 'div'(    int,      int)
var int: 'div'(var int,  var int)
int: 'mod'(    int,      int)
var int: 'mod'(var int,  var int)
float: '/' (   float,    float)
var float: '/' (var float, var float)

```

The result of the modulo operation, if non-zero, always has the same sign as its first operand. The integer division and modulo operations are connected by the following identity:

$$x = (x \text{ div } y) * y + (x \text{ mod } y)$$

Some illustrative examples:

7 div 4 = 1	7 mod 4 = 3
-7 div 4 = -1	-7 mod 4 = -3
7 div -4 = -1	7 mod -4 = 3
-7 div -4 = 1	-7 mod -4 = -3

Sum multiple numbers. Product ([product](#)) is similar. Note that the sum of an empty array is 0, and the product of an empty array is 1.

```

int: sum(array[$T] of int )
var int: sum(array[$T] of var int )
float: sum(array[$T] of float)
var float: sum(array[$T] of var float)

```

Minimum of two values; maximum ([max](#)) is similar.

```
any $T: min(any $T, any $T )
```

Minimum of an array of values; maximum ([max](#)) is similar. Aborts if the array is empty.

```
any $U: min(array[$T] of any $U)
```

Minimum of a fixed set; maximum ([max](#)) is similar. Aborts if the set is empty.

```
$T: min(set of $T)
```

Absolute value of a number.

```
int: abs(    int)
var int: abs(var int)
float: abs(    float)
var float: abs(var float)
```

Square root of a float. Aborts if argument is negative.

```
float: sqrt(    float)
var float: sqrt(var float)
```

Power operator. E.g. `pow(2, 5)` gives 32.

```
int: pow(int,      int)
float: pow(float,   float)
```

Natural exponent.

```
float: exp(float)
var float: exp(var float)
```

Natural logarithm. Logarithm to base 10 (`log10`) and logarithm to base 2 (`log2`) are similar.

```
float: ln(float)
var float: ln(var float)
```

General logarithm; the first argument is the base.

```
float: log(float, float)
```

Sine. Cosine (`cos`), tangent (`tan`), inverse sine (`asin`), inverse cosine (`acos`), inverse tangent (`atan`), hyperbolic sine (`sinh`), hyperbolic cosine (`cosh`), hyperbolic tangent (`tanh`), inverse hyperbolic sine (`asinh`), inverse hyperbolic cosine (`acosh`) and inverse hyperbolic tangent (`atanh`) are similar.

```
float: sin(float)
var float: sin(var float)
```

4.1.11.3 Logical Operations

Conjunction. Other logical operations are similar: disjunction (`\vee`) reverse implication (`<-`), forward implication (`->`), bi-implication (`<->`), exclusive disjunction (`xor`), logical negation (`not`).

Note that the implication operators are not written using `=>`, `<=` and `<=>` as is the case in some languages. This allows `<=` to instead represent “less than or equal”.

```
bool: '/\'(    bool,    bool)
var bool: '/\'(var bool, var bool)
```

Universal quantification. Existential quantification (`exists`) is similar. Note that, when applied to an empty list, `forall` returns true, and `exists` returns false.

```
bool: forall(array[$T]  of    bool)
var bool: forall(array[$T]  of var bool)
```

N-ary exclusive disjunction. N-ary bi-implication (`iffall`) is similar, with `true` instead of `false`.

```
bool: xorall(array[$T]  of    bool: bs) = foldl('xor', false, bs)
var bool: xorall(array[$T]  of var bool: bs) = foldl('xor', false, bs)
```

4.1.11.4 Set Operations

Set membership.

```
bool: 'in'(    $T,      set of $T )
var bool: 'in'(var int,  var set of int)
```

Non-strict subset. Non-strict superset (superset) is similar.

```
bool: 'subset'(    set of $T ,      set of $T )
var bool: 'subset'(var set of int, var set of int)
```

Set union. Other set operations are similar: intersection (`intersect`), difference (`diff`), symmetric difference (`symdiff`).

```
set of $T: 'union'(    set of $T,      set of $T )
var set of int: 'union'(var set of int, var set of int)
```

Set range. If the first argument is larger than the second (e.g. `1..0`), it returns the empty set.

```
set of int: '..'(int, int)
```

Cardinality of a set.

```
int: card(    set of $T)
var int: card(var set of int)
```

Union of an array of sets. Intersection of multiple sets (`array_intersect`) is similar.

```
set of $U: array_union(array[$T] of set of $U)
var set of int: array_union(array[$T] of var set of int)
```

4.1.11.5 Array Operations

Length of an array.

```
int: length(array[$T] of any $U)
```

List concatenation. Returns the list (integer-indexed array) containing all elements of the first argument followed by all elements of the second argument, with elements occurring in the same order as in the arguments. The resulting indices are in the range $1..n$, where n is the sum of the lengths of the arguments. *Rationale:* This allows list-like arrays to be concatenated naturally and avoids problems with overlapping indices. The resulting indices are consistent with those of implicitly indexed array literals. Note that '`++`' also performs string concatenation.

```
array[int] of any $T: '+'(array[int] of any $T, array[int] of any $T)
```

Index sets of arrays. If the argument is a literal, returns $1..n$ where n is the (sub-)array length. Otherwise, returns the declared or inferred index set. This list is only partial, it extends in the obvious way, for arrays of higher dimensions.

```
set of $T: index_set (array[$T] of any $V)
set of $T: index_set_1of2(array[$T, $U] of any $V)
set of $U: index_set_2of2(array[$T, $U] of any $V)
...
```

Replace the indices of the array given by the last argument with the Cartesian product of the sets given by the previous arguments. Similar versions exist for arrays up to 6 dimensions.

```
array[$T1] of any $V: array1d(set of $T1, array[$U] of any $V)
array[$T1, $T2] of any $V:
    array2d(set of $T1, set of $T2, array[$U] of any $V)
array[$T1, $T2, $T3] of any $V:
    array3d(set of $T1, set of $T2, set of $T3, array[$U] of any $V)
```

4.1.11.6 Coercion Operations

Round a float towards $+\infty$, $-\infty$, and the nearest integer, respectively.

```
int: ceil (float)
int: floor(float)
int: round(float)
```

Explicit casts from one type-inst to another.

```
int:          bool2int(  bool)
var int:      bool2int(var bool)
float:        int2float(   int)
var float:    int2float(var int)
array[int] of $T: set2array(set of $T)
```

4.1.11.7 String Operations

To-string conversion. Converts any value to a string for output purposes. The exact form of the resulting string is implementation-dependent.

```
string: show(any $T)
```

Formatted to-string conversion for integers. Converts the integer given by the second argument into a string right justified by the number of characters given by the first argument, or left justified if that argument is negative. If the second argument is not fixed, the form of the string is implementation-dependent.

```
string: show_int(int, var int);
```

Formatted to-string conversion for floats. Converts the float given by the third argument into a string right justified by the number of characters given by the first argument, or left justified if that argument is negative. The number of digits to appear after the decimal point is given by the second argument. It is a run-time error for the second argument to be negative. If the third argument is not fixed, the form of the string is implementation-dependent.

```
string: show_float(int, int, var float)
```

String concatenation. Note that '++' also performs array concatenation.

```
string: '++'(string, string)
```

Concatenate an array of strings. Equivalent to folding '++' over the array, but may be implemented more efficiently.

```
string: concat(array[$T] of string)
```

Concatenate an array of strings, putting a separator between adjacent strings. Returns the empty string if the array is empty.

```
string: join(string, array[$T] of string)
```

4.1.11.8 Bound and Domain Operations

The bound operations `lb` and `ub` return fixed, correct lower/upper bounds to the expression. For numeric types, they return a lower/upper bound value, e.g. the lowest/highest value the expression can take. For set types, they return a subset/superset, e.g. the intersection/union of all possible values of the set expression.

The bound operations abort on expressions that have no corresponding finite bound. For example, this would be the case for a variable declared without bounds in an implementation that does not assign default bounds. (Set expressions always have a finite lower bound of course, namely {}, the empty set.)

Numeric lower/upper bound:

```
int: lb(var int)
float: lb(var float)
int: ub(var int)
float: ub(var float)
```

Set lower/upper bound:

```
set of int: lb(var set of int)
set of int: ub(var set of int)
```

Versions of the bound operations that operate on arrays are also available, they return a safe lower bound or upper bound for all members of the array - they abort if the array is empty:

```
int: lb_array(array[$T] of var int)
float: lb_array(array[$T] of var float)
set of int: lb_array(array[$T] of var set of int)
int: ub_array(array[$T] of var int)
float: ub_array(array[$T] of var float)
set of int: ub_array(array[$T] of var set of int)
```

Integer domain:

```
set of int: dom(var int)
```

The domain operation `dom` returns a fixed superset of the possible values of the expression.

Integer array domain, returns a superset of all possible values that may appear in the array - this aborts if the array is empty:

```
set of int: dom_array(array[$T] of var int)
```

Domain size for integers:

```
int: dom_size(var int)
```

The domain size operation `dom_size` is equivalent to `card(dom(x))`.

Note that these operations can produce different results depending on when they are evaluated and what form the argument takes. For example, consider the numeric lower bound operation.

- If the argument is a fixed expression, the result is the argument's value.
- If the argument is a decision variable, then the result depends on the context.
 - If the implementation can determine a lower bound for the variable, the result is that lower bound. The lower bound may be from the variable's declaration, or higher than that due to preprocessing, or lower than that if an implementation-defined lower bound is applied (e.g. if the variable was declared with no lower bound, but the implementation imposes a lowest possible bound).
 - If the implementation cannot determine a lower bound for the variable, the operation aborts.
- If the argument is any other kind of unfixed expression, the lower bound depends on the bounds of unfixed subexpressions and the connecting operators.

4.1.11.9 Option Type Operations

The option type value (\top) is written

```
opt $T: '<>';
```

One can determine if an option type variable actually occurs or not using `occurs` and `absent`

```
par bool: occurs(par opt $T);  
var bool: occurs(var opt $T);  
par bool: absent(par opt $T);  
var bool: absent(var opt $T);
```

One can return the non-optional value of an option type variable using the function `deopt`

```
par $T: deopt{par opt $T};  
var $T: deopt(var opt $T);
```

4.1.11.10 Other Operations

Check a Boolean expression is true, and abort if not, printing the second argument as the error message. The first one returns the third argument, and is particularly useful for sanity-checking arguments to predicates and functions; importantly, its third argument is lazy, i.e. it is only evaluated if the condition succeeds. The second one returns true and is useful for global sanity-checks (e.g. of instance data) in constraint items.

```
any $T: assert(bool, string, any $T)  
par bool: assert(bool, string)
```

Abort evaluation, printing the given string.

```
any $T: abort(string)
```

Return true. As a side-effect, an implementation may print the first argument.

```
bool: trace(string)
```

Return the second argument. As a side-effect, an implementation may print the first argument.

```
any $T: trace(string, any $T)
```

Check if the argument's value is fixed at this point in evaluation. If not, abort; if so, return its value. This is most useful in output items when decision variables should be fixed: it allows them to be used in places where a fixed value is needed, such as if-then-else conditions.

```
$T: fix(any $T)
```

As above, but return false if the argument's value is not fixed.

```
par bool: is_fixed(any $T)
```

4.1.12 Content-types

The content-type application/x-zinc-output defines a text output format for Zinc. The format extends the abstract syntax and semantics given in *Run-time Outcomes* (page 181), and is discussed in detail in *Output* (page 182).

The full syntax is as follows:

```
% Output
<output> ::= <no-solutions> [ <warnings> ] <free-text>
           | ( <solution> )* [ <complete> ] [ <warnings> ] <free-text>

% Solutions
<solution> ::= <solution-text> [ \n ] "-----" \n

% Unsatisfiable
<no-solutions> ::= "=====UNSATISFIABLE=====" \n

% Complete
<complete> ::= "===== " \n

% Messages
<warnings> ::= ( <message> )+
```

```
<message> ::= ( <line> )+
<line>    ::= "%" [^\n]* \n
```

The solution text for each solution must be as described in [Output Items](#) (page 212). A newline must be appended if the solution text does not end with a newline.

4.1.13 JSON support

MiniZinc can support reading input parameters and providing output formatted as JSON objects. A JSON input file needs to have the following structure:

- Consist of a single top-level object
- The members of the object (the key-value pairs) represent model parameters
- Each member key must be a valid MiniZinc identifier (and it supplies the value for the corresponding parameter of the model)
- Each member value can be one of the following:
 - A string (assigned to a MiniZinc string parameter)
 - A number (assigned to a MiniZinc int or float parameter)
 - The values true or false (assigned to a MiniZinc bool parameter)
 - An array of values. Arrays of arrays are supported only if all inner arrays are of the same length, so that they can be mapped to multi-dimensional MiniZinc arrays.
 - A set of values encoded as an object with a single member with key "set" and a list of values (the elements of the set).

This is an example of a JSON parameter file using all of the above features:

```
{
  "n" : 3,
  "distances" : [ [1,2,3],
                  [4,5,6]],
  "patterns"  : [ {"set" : [1,3,5]}, {"set" : [2,4,6]} ]
}
```

The first parameter declares a simple integer n. The distances parameter is a two-dimensional array; note that all inner arrays must be of the same size in order to map to a (rectangular) MiniZinc two-dimensional array. The third parameter is an array of sets of integers.

Note: The JSON input and output currently does not support enumerated types. This will be added in a future release.

4.1.14 Full grammar

4.1.14.1 Items

```
% A MiniZinc model
<model> ::= [ <item> ";" ... ]

% Items
<item> ::= <include-item>
          | <var-decl-item>
          | <enum-item>
          | <assign-item>
          | <constraint-item>
          | <solve-item>
          | <output-item>
          | <predicate-item>
          | <test-item>
          | <function-item>
          | <annotation-item>

<ti-expr-and-id> ::= <ti-expr> ":" <ident>

% Include items
<include-item> ::= "include" <string-literal>

% Variable declaration items
<var-decl-item> ::= <ti-expr-and-id> <annotations> [ "=" <expr> ]

% Enum items
<enum-item> ::= "enum" <ident> <annotations> [ "=" <enum-cases> ]

<enum-cases> ::= "{" <ident> "," ... "}"

% Assign items
<assign-item> ::= <ident> "=" <expr>

% Constraint items
<constraint-item> ::= "constraint" <string-annotation> <expr>

% Solve item
<solve-item> ::= "solve" <annotations> "satisfy"
                | "solve" <annotations> "minimize" <expr>
                | "solve" <annotations> "maximize" <expr>

% Output items
<output-item> ::= "output" <expr>

% Annotation items
<annotation-item> ::= "annotation" <ident> <params>
```

```
% Predicate, test and function items
<predicate-item> ::= "predicate" <operation-item-tail>

<test-item> ::= "test" <operation-item-tail>

<function-item> ::= "function" <ti-expr> ":" <operation-item-tail>

<operation-item-tail> ::= <ident> <params> <annotations> [ "=" <expr> ]

<params> ::= [ ( <ti-expr-and-id> "," ... ) ]
```

4.1.14.2 Type-Inst Expressions

```
<ti-expr> ::= <base-ti-expr>

<base-ti-expr> ::= <var-par> <base-ti-expr-tail>

<var-par> ::= "var" | "par" | ε

<base-type> ::= "bool"
              | "int"
              | "float"
              | "string"

<base-ti-expr-tail> ::= <ident>
                         | <base-type>
                         | <set-ti-expr-tail>
                         | <ti-variable-expr-tail>
                         | <array-ti-expr-tail>
                         | "ann"
                         | "opt" <base-ti-expr-tail>
                         | { <expr> "," ... }
                         | <num-expr> ".." <num-expr>

% Type-inst variables
<ti-variable-expr-tail> ::= $[A-Za-z][A-Za-z0-9_]*

% Set type-inst expressions
<set-ti-expr-tail> ::= "set" "of" <base-type>

% Array type-inst expressions
<array-ti-expr-tail> ::= "array" [ <ti-expr> "," ... ] "of" <ti-expr>
                           | "list" "of" <ti-expr>
```

4.1.14.3 Expressions

```

<expr> ::= <expr-atom> <expr-binop-tail>

<expr-atom> ::= <expr-atom-head> <expr-atom-tail> <annotations>

<expr-binop-tail> ::= "[" <bin-op> <expr> "]"

<expr-atom-head> ::= <builtin-un-op> <expr-atom>
                     | "(" <expr> ")"
                     | <ident-or-quoted-op>
                     | "_"
                     | <bool-literal>
                     | <int-literal>
                     | <float-literal>
                     | <string-literal>
                     | <set-literal>
                     | <set-comp>
                     | <array-literal>
                     | <array-literal-2d>
                     | <array-comp>
                     | <ann-literal>
                     | <if-then-else-expr>
                     | <let-expr>
                     | <call-expr>
                     | <gen-call-expr>

<expr-atom-tail> ::= ε
                     | <array-access-tail> <expr-atom-tail>

% Numeric expressions

<num-expr> ::= <num-expr-atom> <num-expr-binop-tail>

<num-expr-atom> ::= <num-expr-atom-head> <expr-atom-tail> <annotations>

<num-expr-binop-tail> ::= "[" <num-bin-op> <num-expr> "]"

<num-expr-atom-head> ::= <builtin-num-un-op> <num-expr-atom>
                     | "(" <num-expr> ")"
                     | <ident-or-quoted-op>
                     | <int-literal>
                     | <float-literal>
                     | <if-then-else-expr>
                     | <let-expr>
                     | <call-expr>
                     | <gen-call-expr>

% Built-in operators

<builtin-op> ::= <builtin-bin-op> | <builtin-un-op>

```

```

<bin-op> ::= <builtin-bin-op> | '<ident>'

<builtin-bin-op> ::= "<->" | "->" | "<->" | "\/" | "xor" | "/\\" | "<" | ">" |
    ↵ "<=" | ">=" | "==" | "=" | "!="
    | "in" | "subset" | "superset" | "union" | "diff" |
    ↵ "symdiff"
    | ".." | "intersect" | "++" | <builtin-num-bin-op>

<builtin-un-op> ::= "not" | <builtin-num-un-op>

% Built-in numeric operators
<num-bin-op> ::= <builtin-num-bin-op> | '<ident>'

<builtin-num-bin-op> ::= "+" | "-" | "*" | "/" | "div" | "mod"

<builtin-num-un-op> ::= "+" | "-"

% Boolean literals
<bool-literal> ::= "false" | "true"

% Integer literals
<int-literal> ::= [0-9]+
    | 0x[0-9A-Fa-f]+
    | 0o[0-7]+

% Float literals
<float-literal> ::= [0-9]+.[0-9]+
    | [0-9]+.[0-9]+[Ee][-+]?[0-9]+
    | [0-9]+[Ee][-+]?[0-9]+

% String literals
<string-contents> ::= ([^\n\] | \[^\\n\\])*
    | ""<string-contents>"""

<string-literal> ::= """<string-contents>"""
    | """<string-contents> "\\(" <string-interpolate-tail>

<string-interpolate-tail> ::= <expr> ")"<string-contents>"""
    | <expr> ")"<string-contents> "\\("
    ↵ <string-interpolate-tail>

% Set literals
<set-literal> ::= "{" [ <expr> "," ... ] "}"

% Set comprehensions
<set-comp> ::= "{" <expr> " | " <comp-tail> "}"

<comp-tail> ::= <generator> [ "where" <expr> ] "," ...
    | "in" <expr>

<generator> ::= <ident> "," ... "in" <expr>

```

```
% Array literals
<array-literal> ::= "[" [ <expr> "," ... ] "]"

% 2D Array literals
<array-literal-2d> ::= "[" [ (<expr> "," ...) "|" ... ] "]"

% Array comprehensions
<array-comp> ::= "[" <expr> " | " <comp-tail> "]"

% Array access
<array-access-tail> ::= "[" <expr> "," ... "]"

% Annotation literals
<ann-literal> ::= <ident> [ "(" <expr> "," ... ")" ]

% If-then-else expressions
<if-then-else-expr> ::= "if" <expr> "then" <expr> [ "elseif" <expr> "then" ...
    ↪<expr> ]* "else" <expr> "endif"

% Call expressions
<call-expr> ::= <ident-or-quoted-op> [ "(" <expr> "," ... ")" ]

% Let expressions
<let-expr> ::= "let" "{" <let-item> ";" ... "}" "in" <expr>

<let-item> ::= <var-decl-item>
            | <constraint-item>

% Generator call expressions
<gen-call-expr> ::= <ident-or-quoted-op> "(" <comp-tail> ")" "(" <expr> ")"
```

4.1.14.4 Miscellaneous Elements

```
% Identifiers
<ident> ::= [A-Za-z][A-Za-z0-9_]* | '['^\xa\xd\x0]*'

% Identifiers and quoted operators
<ident-or-quoted-op> ::= <ident>
                        | '<builtin-op>'

% Annotations
<annotations> ::= [ ":" <annotation> ]*

<annotation> ::= <expr-atom-head> <expr-atom-tail>

<string-annotation> ::= ":" <string-literal>
```


CHAPTER 4.2

The MiniZinc library

4.2.1 Global constraints

These constraints represent high-level modelling abstractions, for which many solvers implement special, efficient inference algorithms.

4.2.1.1 All-Different and related constraints

```
predicate all_different(array [$X] of var int: x)
```

Constrain the array of integers x to be all different.

```
predicate all_different(array [$X] of var set of int: x)
```

Constrain the array of sets of integers x to be all different.

```
predicate all_disjoint(array [int] of var set of int: S)
```

Constrain the array of sets of integers S to be pairwise disjoint.

```
predicate all_equal(array [$X] of var int: x)
```

Constrain the array of integers x to be all equal

```
predicate all_equal(array [$X] of var set of int: x)
```

Constrain the array of sets of integers x to be all equal

```
predicate alldifferent_except_0(array [int] of var int: vs)
```

Constrain the array of integers vs to be all different except those elements that are assigned the value 0.

```
predicate nvalue(var int: n, array [int] of var int: x)
```

Requires that the number of distinct values in x is n .

```
function var int: nvalue(array [int] of var int: x)
```

Returns the number of distinct values in x .

```
predicate symmetric_all_different(array [int] of var int: x)
```

Requires the array of integers x to be all different, and for all i , $x[i]=j \rightarrow x[j]=i$.

4.2.1.2 Lexicographic constraints

```
predicate lex2(array [int,int] of var int: x)
```

Require adjacent rows and adjacent columns in the array x to be lexicographically ordered. Adjacent rows and adjacent columns may be equal.

```
predicate lex_greater(array [int] of var bool: x,
                     array [int] of var bool: y)
```

Requires that the array x is strictly lexicographically greater than array y . Compares them from first to last element, regardless of indices.

```
predicate lex_greater(array [int] of var int: x,
                     array [int] of var int: y)
```

Requires that the array x is strictly lexicographically greater than array y . Compares them from first to last element, regardless of indices.

```
predicate lex_greater(array [int] of var float: x,
                     array [int] of var float: y)
```

Requires that the array x is strictly lexicographically greater than array y . Compares them from first to last element, regardless of indices.

```
predicate lex_greater(array [int] of var set of int: x,
                     array [int] of var set of int: y)
```

Requires that the array x is strictly lexicographically greater than array y . Compares them from first to last element, regardless of indices.

```
predicate lex_greatereq(array [int] of var bool: x,
                       array [int] of var bool: y)
```

Requires that the array x is lexicographically greater than or equal to array y . Compares them from first to last element, regardless of indices.

```
predicate lex_greatereq(array [int] of var int: x,
                       array [int] of var int: y)
```

Requires that the array x is lexicographically greater than or equal to array y . Compares them from first to last element, regardless of indices.

```
predicate lex_greatereq(array [int] of var float: x,
                       array [int] of var float: y)
```

Requires that the array x is lexicographically greater than or equal to array y . Compares them from first to last element, regardless of indices.

```
predicate lex_greatereq(array [int] of var set of int: x,
                       array [int] of var set of int: y)
```

Requires that the array x is lexicographically greater than or equal to array y . Compares them from first to last element, regardless of indices.

```
predicate lex_less(array [int] of var bool: x,
                  array [int] of var bool: y)
```

Requires that the array x is strictly lexicographically less than array y . Compares them from first to last element, regardless of indices.

```
predicate lex_less(array [int] of var int: x,
                  array [int] of var int: y)
```

Requires that the array x is strictly lexicographically less than array y . Compares them from first to last element, regardless of indices.

```
predicate lex_less(array [int] of var float: x,
                  array [int] of var float: y)
```

Requires that the array x is strictly lexicographically less than array y . Compares them from first to last element, regardless of indices.

```
predicate lex_less(array [int] of var set of int: x,
                  array [int] of var set of int: y)
```

Requires that the array x is strictly lexicographically less than array y . Compares them from first to last element, regardless of indices.

```
predicate lex_lesseq(array [int] of var bool: x,
                     array [int] of var bool: y)
```

Requires that the array x is lexicographically less than or equal to array y . Compares them from first to last element, regardless of indices.

```
predicate lex_lesseq(array [int] of var float: x,
                     array [int] of var float: y)
```

Requires that the array x is lexicographically less than or equal to array y . Compares them from first to last element, regardless of indices.

```
predicate lex_lesseq(array [int] of var int: x,
                     array [int] of var int: y)
```

Requires that the array x is lexicographically less than or equal to array y . Compares them from first to last element, regardless of indices.

```
predicate lex_lesseq(array [int] of var set of int: x,
                     array [int] of var set of int: y)
```

Requires that the array x is lexicographically less than or equal to array y . Compares them from first to last element, regardless of indices.

```
predicate strict_lex2(array [int,int] of var int: x)
```

Require adjacent rows and adjacent columns in the array x to be lexicographically ordered. Adjacent rows and adjacent columns cannot be equal.

```
predicate value_precede(int: s, int: t, array [int] of var int: x)
```

Requires that s precede t in the array x .

Precedence means that if any element of x is equal to t , then another element of x with a lower index is equal to s .

```
predicate value_precede(int: s,
                      int: t,
                      array [int] of var set of int: x)
```

Requires that s precede t in the array x .

Precedence means that if an element of x contains t but not s , then another element of x with lower index contains s but not t .

```
predicate value_precede_chain(array [int] of int: c,
                             array [int] of var int: x)
```

Requires that $c[i]$ precedes $c[i+1]$ in the array x .

Precedence means that if any element of x is equal to $c[i+1]$, then another element of x with a lower index is equal to $c[i]$.

```
predicate value_precede_chain(array [int] of int: c,
                             array [int] of var set of int: x)
```

Requires that $c[i]$ precedes $c[i+1]$ in the array x .

Precedence means that if an element of x contains $c[i+1]$ but not $c[i]$, then another element of x with lower index contains $c[i]$ but not $c[i+1]$.

4.2.1.3 Sorting constraints

```
function array [int] of var int: arg_sort(array [int] of var int: x)
```

Returns the permutation p which causes x to be in sorted order hence $x[p[i]] \leq x[p[i+1]]$.

The permutation is the stable sort hence $x[p[i]] = x[p[i+1]] \rightarrow p[i] < p[i+1]$.

```
function array [int] of var int: arg_sort(array [int] of var float: x)
```

Returns the permutation p which causes x to be in sorted order hence $x[p[i]] \leq x[p[i+1]]$.

The permutation is the stable sort hence $x[p[i]] = x[p[i+1]] \rightarrow p[i] < p[i+1]$.

```
predicate arg_sort(array [int] of var int: x,
                  array [int] of var int: p)
```

Constrains p to be the permutation which causes x to be in sorted order hence $x[p[i]] \leq x[p[i+1]]$.

The permutation is the stable sort hence $x[p[i]] = x[p[i+1]] \rightarrow p[i] < p[i+1]$.

```
predicate arg_sort(array [int] of var float: x,
                  array [int] of var int: p)
```

Constrains p to be the permutation which causes x to be in sorted order hence $x[p[i]] \leq x[p[i+1]]$.

The permutation is the stable sort hence $x[p[i]] = x[p[i+1]] \rightarrow p[i] < p[i+1]$.

```
predicate decreasing(array [int] of var bool: x)
```

Requires that the array x is in decreasing order (duplicates are allowed).

```
predicate decreasing(array [int] of var float: x)
```

Requires that the array x is in decreasing order (duplicates are allowed).

```
predicate decreasing(array [int] of var int: x)
```

Requires that the array x is in decreasing order (duplicates are allowed).

```
predicate decreasing(array [int] of var set of int: x)
```

Requires that the array x is in decreasing order (duplicates are allowed).

```
predicate increasing(array [int] of var bool: x)
```

Requires that the array x is in increasing order (duplicates are allowed).

```
predicate increasing(array [int] of var float: x)
```

Requires that the array x is in increasing order (duplicates are allowed).

```
predicate increasing(array [int] of var int: x)
```

Requires that the array x is in increasing order (duplicates are allowed).

```
predicate increasing(array [int] of var set of int: x)
```

Requires that the array x is in increasing order (duplicates are allowed).

```
predicate sort(array [int] of var int: x, array [int] of var int: y)
```

Requires that the multiset of values in x are the same as the multiset of values in y but y is in sorted order.

```
function array [int] of var int: sort(array [int] of var int: x)
```

Return a multiset of values that is the same as the multiset of values in x but in sorted order.

4.2.1.4 Channeling constraints

```
predicate int_set_channel(array [int] of var int: x,
                           array [int] of var set of int: y)
```

Requires that array of int variables x and array of set variables y are related such that ($x[i] = j$) if and only if ($i \in y[j]$).

```
predicate inverse(array [int] of var int: f,
                  array [int] of var int: invf)
```

Constrains two arrays of int variables, f and invf , to represent inverse functions. All the values in each array must be within the index set of the other array.

```
function array [int] of var int: inverse(array [int] of var int: f)
```

Given a function f represented as an array, return the inverse function.

```
predicate inverse_set(array [int] of var set of int: f,
                      array [int] of var set of int: invf)
```

Constrains two arrays of set of int variables, f and invf , so that $a \in f[i]$ iff $i \in invf[a]$. All the values in each array's sets must be within the index set of the other array.

```
predicate link_set_to_booleans(var set of int: s,
                               array [int] of var bool: b)
```

Constrain the array of Booleans b to be a representation of the set s : i in s if and only if $b[i]$.

The index set of b must be a superset of the possible values of s .

4.2.1.5 Counting constraints

```
predicate among(var int: n, array [int] of var int: x, set of int: v)
```

Requires exactly n variables in x to take one of the values in v .

```
function var int: among(array [int] of var int: x, set of int: v)
```

Returns the number of variables in x that take one of the values in v .

```
predicate at_least(int: n, array [int] of var int: x, int: v)
```

Requires at least n variables in x to take the value v .

```
predicate at_least(int: n,
                  array [int] of var set of int: x,
                  set of int: v)
```

Requires at least n variables in x to take the value v .

```
predicate at_most(int: n, array [int] of var int: x, int: v)
```

Requires at most n variables in x to take the value v .

```
predicate at_most(int: n,
                  array [int] of var set of int: x,
                  set of int: v)
```

Requires at most n variables in x to take the value v .

```
predicate at_most1(array [int] of var set of int: s)
```

Requires that each pair of sets in s overlap in at most one element.

```
predicate count(array [int] of var int: x, var int: y, var int: c)
```

Constrains c to be the number of occurrences of y in x .

```
function var int: count(array [int] of var int: x, var int: y)
```

Returns the number of occurrences of y in x .

```
predicate count_eq(array [int] of var int: x, var int: y, var int: c)
```

Constrains c to be the number of occurrences of y in x .

```
predicate count_geq(array [int] of var int: x, var int: y, var int: c)
```

Constrains c to be greater than or equal to the number of occurrences of y in x .

```
predicate count_gt(array [int] of var int: x, var int: y, var int: c)
```

Constrains c to be strictly greater than the number of occurrences of y in x .

```
predicate count_leq(array [int] of var int: x, var int: y, var int: c)
```

Constrains c to be less than or equal to the number of occurrences of y in x .

```
predicate count_lt(array [int] of var int: x, var int: y, var int: c)
```

Constrains c to be strictly less than the number of occurrences of y in x .

```
predicate count_neq(array [int] of var int: x, var int: y, var int: c)
```

Constrains c to be not equal to the number of occurrences of y in x .

```
predicate distribute(array [int] of var int: card,
                    array [int] of var int: value,
                    array [int] of var int: base)
```

Requires that card [i] is the number of occurrences of value [i] in base . The values in value need not be distinct.

```
function array [int] of var int: distribute(array [int] of var int: value,
                                            array [int] of var int: base)
```

Returns an array of the number of occurrences of value [i] in base . The values in value need not be distinct.

```
predicate exactly(int: n, array [int] of var int: x, int: v)
```

Requires exactly n variables in x to take the value v .

```
predicate exactly(int: n,
                 array [int] of var set of int: x,
                 set of int: v)
```

Requires exactly n variables in x to take the value v .

```
predicate global_cardinality(array [int] of var int: x,
                            array [int] of int: cover,
                            array [int] of var int: counts)
```

Requires that the number of occurrences of cover [i] in x is counts [i].

```
function array [int] of var int: global_cardinality(array [int] of var int:_
    ↪x,
                                array [int] of int:_ 
    ↪cover)
```

Returns the number of occurrences of cover [i] in x .

```
predicate global_cardinality_closed(array [int] of var int: x,
                                    array [int] of int: cover,
                                    array [int] of var int: counts)
```

Requires that the number of occurrences of i in x is counts [i].

The elements of x must take their values from cover .

```
function array [int] of var int: global_cardinality_closed(array [int] of_
    ↪var int: x,
                                array [int] of_
    ↪int: cover)
```

Returns an array with number of occurrences of i in x .

The elements of x must take their values from cover .

```
predicate global_cardinality_low_up(array [int] of var int: x,
                                    array [int] of int: cover,
                                    array [int] of int: lbound,
                                    array [int] of int: ubound)
```

Requires that for all i , the value cover [i] appears at least lbound [i] and at most ubound [i] times in the array x .

```
predicate global_cardinality_low_up_closed(array [int] of var int: x,
                                            array [int] of int: cover,
                                            array [int] of int: lbound,
                                            array [int] of int: ubound)
```

Requires that for all i , the value cover [i] appears at least lbound [i] and at most ubound [i] times in the array x .

The elements of x must take their values from cover .

4.2.1.6 Packing constraints

```

predicate bin_packing(int: c,
                      array [int] of var int: bin,
                      array [int] of int: w)

```

Requires that each item i with weight $w[i]$, be put into $bin[i]$ such that the sum of the weights of the items in each bin does not exceed the capacity c .

Assumptions:

- $\forall i, w[i] \geq 0$
- $c \geq 0$

```

predicate bin_packing_capa(array [int] of int: c,
                           array [int] of var int: bin,
                           array [int] of int: w)

```

Requires that each item i with weight $w[i]$, be put into $bin[i]$ such that the sum of the weights of the items in each bin b does not exceed the capacity $c[b]$.

Assumptions:

- $\forall i, w[i] \geq 0$
- $\forall b, c[b] \geq 0$

```

predicate bin_packing_load(array [int] of var int: load,
                           array [int] of var int: bin,
                           array [int] of int: w)

```

Requires that each item i with weight $w[i]$, be put into $bin[i]$ such that the sum of the weights of the items in each bin b is equal to $load[b]$.

Assumptions:

- $\forall i, w[i] \geq 0$

```

function array [int] of var int: bin_packing_load(array [int] of var int:_
→bin,
                                                 array [int] of int: w)

```

Returns the load of each bin resulting from packing each item i with weight $w[i]$ into $bin[i]$, where the load is defined as the sum of the weights of the items in each bin.

Assumptions:

- $\forall i, w[i] \geq 0$

```

predicate diffn(array [int] of var int: x,
                array [int] of var int: y,
                array [int] of var int: dx,
                array [int] of var int: dy)

```

Constrains rectangles i , given by their origins ($x[i], y[i]$) and sizes ($dx[i], dy[i]$), to be non-overlapping. Zero-width rectangles can still not overlap with any other rectangle.

```
predicate diffn_k(array [int,int] of var int: box_posn,
                  array [int,int] of var int: box_size)
```

Constrains k -dimensional boxes to be non-overlapping. For each box i and dimension j , $\text{box_posn}[i, j]$ is the base position of the box in dimension j , and $\text{box_size}[i, j]$ is the size in that dimension. Boxes whose size is 0 in any dimension still cannot overlap with any other box.

```
predicate diffn_nonstrict(array [int] of var int: x,
                          array [int] of var int: y,
                          array [int] of var int: dx,
                          array [int] of var int: dy)
```

Constrains rectangles i , given by their origins ($x[i], y[i]$) and sizes ($dx[i], dy[i]$), to be non-overlapping. Zero-width rectangles can be packed anywhere.

```
predicate diffn_nonstrict_k(array [int,int] of var int: box_posn,
                            array [int,int] of var int: box_size)
```

Constrains k -dimensional boxes to be non-overlapping. For each box i and dimension j , $\text{box_posn}[i, j]$ is the base position of the box in dimension j , and $\text{box_size}[i, j]$ is the size in that dimension. Boxes whose size is 0 in at least one dimension can be packed anywhere.

```
predicate geost(int: k,
                array [int,int] of int: rect_size,
                array [int,int] of int: rect_offset,
                array [int] of set of int: shape,
                array [int,int] of var int: x,
                array [int] of var int: kind)
```

A global non-overlap constraint for k dimensional objects. It enforces that no two objects overlap.

Parameters:

- k : the number of dimensions
- rect_size : the size of each box in k dimensions
- rect_offset : the offset of each box from the base position in k dimensions
- shape : the set of rectangles defining the i -th shape. Assumption: Each pair of boxes in a shape must not overlap.
- x : the base position of each object. $x[i, j]$ is the position of object i in dimension j .
- kind : the shape used by each object.

```
predicate geost_bb(int: k,
                   array [int,int] of int: rect_size,
                   array [int,int] of int: rect_offset,
                   array [int] of set of int: shape,
```

```
array [int,int] of var int: x,
array [int] of var int: kind,
array [int] of var int: l,
array [int] of var int: u)
```

A global non-overlap constraint for k dimensional objects. It enforces that no two objects overlap, and that all objects fit within a global k dimensional bounding box.

Parameters:

- k: the number of dimensions
- rect_size: the size of each box in k dimensions
- rect_offset: the offset of each box from the base position in k dimensions
- shape: the set of rectangles defining the i -th shape. Assumption: Each pair of boxes in a shape must not overlap.
- x: the base position of each object. $x [i , j]$ is the position of object i in dimension j .
- kind: the shape used by each object.
- l: is an array of lower bounds, $l [i]$ is the minimum bounding box for all objects in dimension i .
- u: is an array of upper bounds, $u [i]$ is the maximum bounding box for all objects in dimension i .

```
predicate geost_smallest_bb(int: k,
                           array [int,int] of int: rect_size,
                           array [int,int] of int: rect_offset,
                           array [int] of set of int: shape,
                           array [int,int] of var int: x,
                           array [int] of var int: kind,
                           array [int] of var int: l,
                           array [int] of var int: u)
```

A global non-overlap constraint for k dimensional objects. It enforces that no two objects overlap, and that all objects fit within a global k dimensional bounding box. In addition, it enforces that the bounding box is the smallest one containing all objects, i.e., each of the $2k$ boundaries is touched by at least by one object.

Parameters:

- k: the number of dimensions
- rect_size: the size of each box in k dimensions
- rect_offset: the offset of each box from the base position in k dimensions
- shape: the set of rectangles defining the i -th shape. Assumption: Each pair of boxes in a shape must not overlap.
- x: the base position of each object. $x [i , j]$ is the position of object i in dimension j .
- kind: the shape used by each object.

- l : is an array of lower bounds, $l[i]$ is the minimum bounding box for all objects in dimension i .
- u : is an array of upper bounds, $u[i]$ is the maximum bounding box for all objects in dimension i .

```
predicate knapsack(array [int] of int: w,
                    array [int] of int: p,
                    array [int] of var int: x,
                    var int: W,
                    var int: P)
```

Requires that items are packed in a knapsack with certain weight and profit restrictions.

Assumptions:

- Weights w and profits p must be non-negative
- w , p and x must have the same index sets

Parameters:

- w : weight of each type of item
- p : profit of each type of item
- x : number of items of each type that are packed
- W : sum of sizes of all items in the knapsack
- P : sum of profits of all items in the knapsack

4.2.1.7 Scheduling constraints

```
predicate alternative(var opt int: s0,
                     var int: d0,
                     array [int] of var opt int: s,
                     array [int] of var int: d)
```

Alternative constraint for optional tasks. Task (s_0, d_0) spans the optional tasks ($s[i], d[i]$) in the array arguments and at most one can occur

```
predicate cumulative(array [int] of var int: s,
                     array [int] of var int: d,
                     array [int] of var int: r,
                     var int: b)
```

Requires that a set of tasks given by start times s , durations d , and resource requirements r , never require more than a global resource bound b at any one time.

Assumptions:

- $\text{forall } i, d[i] \geq 0 \text{ and } r[i] \geq 0$

```
predicate cumulative(array [int] of var opt int: s,
                     array [int] of var int: d,
                     array [int] of var int: r,
                     var int: b)
```

Requires that a set of tasks given by start times s , durations d , and resource requirements r , never require more than a global resource bound b at any one time. Start times are optional variables, so that absent tasks do not need to be scheduled.

Assumptions: - forall i , $d[i] \geq 0$ and $r[i] \geq 0$

```
predicate disjunctive(array [int] of var int: s,
                      array [int] of var int: d)
```

Requires that a set of tasks given by start times s and durations d do not overlap in time. Tasks with duration 0 can be scheduled at any time, even in the middle of other tasks.

Assumptions:

- forall i , $d[i] \geq 0$

```
predicate disjunctive(array [int] of var opt int: s,
                      array [int] of var int: d)
```

Requires that a set of tasks given by start times s and durations d do not overlap in time. Tasks with duration 0 can be scheduled at any time, even in the middle of other tasks. Start times are optional variables, so that absent tasks do not need to be scheduled.

Assumptions:

- forall i , $d[i] \geq 0$

```
predicate disjunctive_strict(array [int] of var int: s,
                            array [int] of var int: d)
```

Requires that a set of tasks given by start times s and durations d do not overlap in time. Tasks with duration 0 CANNOT be scheduled at any time, but only when no other task is running.

Assumptions:

- forall i , $d[i] \geq 0$

```
predicate disjunctive_strict(array [int] of var opt int: s,
                            array [int] of var int: d)
```

Requires that a set of tasks given by start times s and durations d do not overlap in time. Tasks with duration 0 CANNOT be scheduled at any time, but only when no other task is running. Start times are optional variables, so that absent tasks do not need to be scheduled.

Assumptions:

- forall i , $d[i] \geq 0$

```
predicate span(var opt int: s0,
              var int: d0,
              array [int] of var opt int: s,
              array [int] of var int: d)
```

Span constraint for optional tasks. Task (s_0, d_0) spans the optional tasks ($s[i], d[i]$) in the array arguments.

4.2.1.8 Extensional constraints (table, regular etc.)

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

The sequence of values in array x (which must all be in the range $1..S$) is accepted by the DFA of Q states with input $1..S$ and transition function d (which maps $(1..Q, 1..S) \rightarrow 0..Q$) and initial state q_0 (which must be in $1..Q$) and accepting states F (which all must be in $1..Q$). We reserve state 0 to be an always failing state.

```
predicate regular(array [int] of var int: x, string: r)
```

The sequence of values in array x is accepted by the regular expression r . This constraint generates its DFA equivalent.

Regular expressions can use the following syntax:

- Selection:
 - Concatenation: “12 34”, 12 followed by 34. (Characters are assumed to be the part of the same number unless split by syntax or whitespace.)
 - Union: “7|11”, a 7 or 11.
 - Groups: “7(6|8)”, a 7 followed by a 6 or an 8.
 - Wildcard: “.”, any value within the domain.
 - Classes: “[3-6 7]”, a 3,4,5,6, or 7.
 - Negated classes: “[^ 3 5]”, any value within the domain except for a 3 or a 5.
- Quantifiers:
 - Asterisk: “12*”, 0 or more times a 12.
 - Question mark: “5?”, 0 or 1 times a 5. (optional)
 - Plus sign: “42+”, 1 or more time a 42.
 - Exact: “1{3}”, exactly 3 times a 1.

- At least: “9{5,}”, 5 or more times a 9.
- Between: “7{3,5}”, at least 3 times, but at most 5 times a 7.

Members of enumerated types can be used in place of any integer (e.g., “A B”, A followed by B). Enumerated identifiers still use whitespace for concatenation.

```
predicate regular_nfa(array [int] of var int: x,
                      int: Q,
                      int: S,
                      array [int,int] of set of int: d,
                      int: q0,
                      set of int: F)
```

The sequence of values in array x (which must all be in the range 1.. S) is accepted by the NFA of Q states with input 1.. S and transition function d (which maps (1.. Q, 1.. S) -> set of 1.. Q)) and initial state q0 (which must be in 1.. Q) and accepting states F (which all must be in 1.. Q).

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

Represents the constraint x in t where we consider each row in t to be a tuple and t as a set of tuples.

```
predicate table(array [int] of var int: x, array [int,int] of int: t)
```

Represents the constraint x in t where we consider each row in t to be a tuple and t as a set of tuples.

4.2.1.9 Other declarations

```
function var int: arg_max(array [int] of var int: x)
```

Returns the index of the maximum value in the array x . When breaking ties the least index is returned.

```
function var int: arg_max(array [int] of var float: x)
```

Returns the index of the maximum value in the array x . When breaking ties the least index is returned.

```
function var int: arg_min(array [int] of var int: x)
```

Returns the index of the minimum value in the array x . When breaking ties the least index is returned.

```
function var int: arg_min(array [int] of var float: x)
```

Returns the index of the minimum value in the array x . When breaking ties the least index is returned.

```
predicate circuit(array [int] of var int: x)
```

Constrains the elements of x to define a circuit where $x[i] = j$ means that j is the successor of i .

```
predicate disjoint(var set of int: s1, var set of int: s2)
```

Requires that sets $s1$ and $s2$ do not intersect.

```
predicate maximum(var int: m, array [int] of var int: x)
```

Constrains m to be the maximum of the values in x .

Assumptions: $|x| > 0$.

```
predicate maximum(var float: m, array [int] of var float: x)
```

Constrains m to be the maximum of the values in x .

Assumptions: $|x| > 0$.

```
predicate maximum_arg(array [int] of var int: x, var int: i)
```

Constrain i to be the index of the maximum value in the array x . When breaking ties the least index is returned.

Assumption: $|x| > 0$

```
predicate maximum_arg(array [int] of var float: x, var int: i)
```

Constrain i to be the index of the maximum value in the array x . When breaking ties the least index is returned.

Assumption: $|x| > 0$

```
predicate member(array [int] of var bool: x, var bool: y)
```

Requires that y occurs in the array x .

```
predicate member(array [int] of var float: x, var float: y)
```

Requires that y occurs in the array x .

```
predicate member(array [int] of var int: x, var int: y)
```

Requires that y occurs in the array x .

```
predicate member(array [int] of var set of int: x, var set of int: y)
```

Requires that y occurs in the array x .

```
predicate member(var set of int: x, var int: y)
```

Requires that y occurs in the set x .

```
predicate minimum(var float: m, array [int] of var float: x)
```

Constrains m to be the minimum of the values in x .

Assumptions: $|x| > 0$.

```
predicate minimum(var int: m, array [int] of var int: x)
```

Constrains m to be the minimum of the values in x .

Assumptions: $|x| > 0$.

```
predicate minimum_arg(array [int] of var int: x, var int: i)
```

Constrain i to be the index of the minimum value in the array x . When breaking ties the least index is returned.

Assumption: $|x| > 0$

```
predicate minimum_arg(array [int] of var float: x, var int: i)
```

Constrain i to be the index of the minimum value in the array x . When breaking ties the least index is returned.

Assumption: $|x| > 0$

```
predicate network_flow(array [int,1..2] of int: arc,
                      array [int] of int: balance,
                      array [int] of var int: flow)
```

Defines a network flow constraint.

Parameters:

- arc: a directed arc of the flow network. Arc i connects node arc [i ,1] to node arc [i ,2].
- balance: the difference between input and output flow for each node.
- flow: the flow going through each arc.

```
predicate network_flow_cost(array [int,1..2] of int: arc,
                           array [int] of int: balance,
                           array [int] of int: weight,
                           array [int] of var int: flow,
                           var int: cost)
```

Defines a network flow constraint with cost.

Parameters:

- arc: a directed arc of the flow network. Arc i connects node arc [i ,1] to node arc [i ,2].
- balance: the difference between input and output flow for each node.
- weight: the unit cost of the flow through the arc.
- flow: the flow going through each arc.
- cost: the overall cost of the flow.

```
predicate partition_set(array [int] of var set of int: S,
                      set of int: universe)
```

Constrains the sets in array S to partition the universe .

```
predicate range(array [int] of var int: x,
               var set of int: s,
               var set of int: t)
```

Requires that the image of function x (represented as an array) on set of values s is t . ub(s) must be a subset of index_set(x) otherwise an assertion failure will occur.

```
function var set of int: range(array [int] of var int: x,
                               var set of int: s)
```

Returns the image of function x (represented as an array) on set of values s . ub(s) must be a subset of index_set(x) otherwise an assertion failure will occur.

```
predicate roots(array [int] of var int: x,
               var set of int: s,
               var set of int: t)
```

Requires that x [i] in t for all i in s

```
function var set of int: roots(array [int] of var int: x,
                                var set of int: t)
```

Returns s such that $x[i]$ in t for all i in s

```
predicate sliding_sum(int: low,
                      int: up,
                      int: seq,
                      array [int] of var int: vs)
```

Requires that in each subsequence $vs[i], \dots, vs[i + seq - 1]$ the sum of the values belongs to the interval [low, up].

```
predicate subcircuit(array [int] of var int: x)
```

Constrains the elements of x to define a subcircuit where $x[i] = j$ means that j is the successor of i and $x[i] = i$ means that i is not in the circuit.

```
predicate sum_pred(var int: i,
                  array [int] of set of int: sets,
                  array [int] of int: cs,
                  var int: s)
```

Requires that the sum of $cs[i_1]..cs[i_N]$ equals s , where $i_1 .. i_N$ are the elements of the i th set in sets .

Nb: not called ‘sum’ as in the constraints catalog because ‘sum’ is a MiniZinc built-in function.

4.2.2 Annotations

These annotations control evaluation and solving behaviour.

4.2.2.1 General annotations

Parameters

```
annotation add_to_output
```

Declare that the annotated variable should be added to the output of the model. This annotation only has an effect when the model does not have an output item.

```
annotation is_defined_var
```

Declare the annotated variable as being functionally defined. This annotation is introduced into FlatZinc code by the compiler.

```
annotation is_reverse_map
```

Declare that the annotated expression is used to map an expression back from FlatZinc to MiniZinc.

```
annotation maybe_partial
```

Declare that expression may have undefined result (to avoid warnings)

```
annotation mzn_break_here
```

With debug build of mzn2fzn, call MiniZinc::mzn_break_here when flattening this expression to make debugging easier. This annotation is ignored by the release build.

```
annotation mzn_check_var
```

Declare that the annotated variable is required for checking solutions.

```
annotation mzn_rhs_from_assignment
```

Used internally by the compiler

```
annotation output_only
```

Declare that the annotated variable should be only used for output. This annotation can be used to define variables that are required for solution checkers, or that are necessary for the output item. The annotated variable must be par.

```
annotation output_var
```

Declare that the annotated variable should be printed by the solver. This annotation is introduced into FlatZinc code by the compiler.

```
annotation promise_total
```

Declare function as total, i.e. it does not put any constraints on its arguments.

```
annotation var_is_introduced
```

Declare a variable as being introduced by the compiler.

Functions and Predicates

```
annotation constraint_name(string: s)
```

Used to attach a name s to a constraint and its decomposition. String annotations on constraint keywords are re-written as constraint_name annotations

```
annotation defines_var(var $t: c)
```

Declare variable: c as being functionally defined by the annotated constraint. This annotation is introduced into FlatZinc code by the compiler.

```
annotation doc_comment(string: s)
```

Document the function or variable declaration item with the string s .

```
annotation expression_name(string: s)
```

Used to attach a name s to an expression, this should also propagate to any sub-expressions or decomposition of the annotated expression. String annotations on expressions are re-written as expression_name annotations

```
annotation mzn_check_enum_var(set of int)
```

Declare that the annotated variable is required for checking solutions and has an enum type.

```
annotation mzn_constraint_name(string)
```

Declare a name for the annotated constraint.

```
annotation mzn_expression_name(string)
```

Declare a name for the annotated expression.

```
annotation mzn_path(string: s)
```

Representation of the call-stack when the annotated item was introduced, as a string s . Can be used to uniquely identify variables and constraints across different compilations of a model that may have different names. This annotations is introduced into FlatZinc code by the compiler and is retained if –keep-paths argument is used.

```
annotation output_array(array [$u] of set of int: a)
```

Declare that the annotated array should be printed by the solver with the given index sets `a`. This annotation is introduced into FlatZinc code by the compiler.

4.2.2.2 Propagation strength annotations

```
annotation bounds
```

Annotate a constraint to use bounds propagation

```
annotation domain
```

Annotate a constraint to use domain propagation

4.2.2.3 Search annotations

Variable selection annotations

```
annotation anti_first_fail
```

Choose the variable with the largest domain

```
annotation dom_w_deg
```

Choose the variable with largest domain, divided by the number of attached constraints weighted by how often they have caused failure

```
annotation first_fail
```

Choose the variable with the smallest domain

```
annotation impact
```

Choose the variable with the highest impact so far during the search

```
annotation input_order
```

Search variables in the given order

```
annotation largest
```

Choose the variable with the largest value in its domain

```
annotation max_regret
```

Choose the variable with largest difference between the two smallest values in its domain

```
annotation most_constrained
```

Choose the variable with the smallest domain, breaking ties using the number of attached constraints

```
annotation occurrence
```

Choose the variable with the largest number of attached constraints

```
annotation smallest
```

Choose the variable with the smallest value in its domain

Value choice annotations

```
annotation indomain
```

Assign values in ascending order

```
annotation indomain_interval
```

If the domain consists of several contiguous intervals, reduce the domain to the first interval. Otherwise bisect the domain.

```
annotation indomain_max
```

Assign the largest value in the domain

```
annotation indomain_median
```

Assign the middle value in the domain

```
annotation indomain_middle
```

Assign the value in the domain closest to the mean of its current bounds

```
annotation indomain_min
```

Assign the smallest value in the domain

```
annotation indomain_random
```

Assign a random value from the domain

```
annotation indomain_reverse_split
```

Bisect the domain, excluding the lower half first

```
annotation indomain_split
```

Bisect the domain, excluding the upper half first

```
annotation indomain_split_random
```

Bisect the domain, randomly selecting which half to exclude first

```
annotation outdomain_max
```

Exclude the largest value from the domain

```
annotation outdomain_median
```

Exclude the middle value from the domain

```
annotation outdomain_min
```

Exclude the smallest value from the domain

```
annotation outdomain_random
```

Exclude a random value from the domain

Exploration strategy annotations

```
annotation complete
```

Perform a complete search

Restart annotations

Parameters

```
annotation restart_none
```

Do not restart

Functions and Predicates

```
annotation restart_constant(int: scale)
```

Restart after constant number of nodes scale

```
annotation restart_geometric(float: base, int: scale)
```

Restart with geometric sequence with parameters base and scale

```
annotation restart_linear(int: scale)
```

Restart with linear sequence scaled by scale

```
annotation restart_luby(int: scale)
```

Restart with Luby sequence scaled by scale

Other declarations

```
annotation bool_search(array [$X] of var bool: x,
                      ann: select,
                      ann: choice,
                      ann: explore)
```

Specify search on variables `x` , with variable selection strategy `select` , value choice strategy `choice` , and exploration strategy `explore` . If `x` is a multi-dimensional array, it is coerced to one-dimensional in row-major order (as with the `array1d` function).

```
annotation bool_search(array [$X] of var bool: x,
                      ann: select,
                      ann: choice)
```

Specify search on variables x , with variable selection strategy `select` , and value choice strategy `choice` . If x is a multi-dimensional array, it is coerced to one-dimensional in row-major order (as with the `array1d` function).

```
annotation float_search(array [$X] of var float: x,
                        float: prec,
                        ann: select,
                        ann: choice,
                        ann: explore)
```

Specify search on variables x , with precision `prec` , variable selection strategy `select` , value choice strategy `choice` , and exploration strategy `explore` . If x is a multi-dimensional array, it is coerced to one-dimensional in row-major order (as with the `array1d` function).

```
annotation float_search(array [$X] of var float: x,
                        float: prec,
                        ann: select,
                        ann: choice)
```

Specify search on variables x , with precision `prec` , variable selection strategy `select` , and value choice strategy `choice` . If x is a multi-dimensional array, it is coerced to one-dimensional in row-major order (as with the `array1d` function).

```
annotation int_search(array [$X] of var int: x,
                      ann: select,
                      ann: choice,
                      ann: explore)
```

Specify search on variables x , with variable selection strategy `select` , value choice strategy `choice` , and exploration strategy `explore` . If x is a multi-dimensional array, it is coerced to one-dimensional in row-major order (as with the `array1d` function).

```
annotation int_search(array [$X] of var int: x,
                      ann: select,
                      ann: choice)
```

Specify search on variables x , with variable selection strategy `select` , and value choice strategy `choice` . If x is a multi-dimensional array, it is coerced to one-dimensional in row-major order (as with the `array1d` function).

```
annotation seq_search(array [int] of ann: s)
```

Sequentially perform the searches specified in array s

```
annotation set_search(array [$X] of var set of int: x,
                      ann: select,
                      ann: choice,
```

```
ann: explore)
```

Specify search on variables x , with variable selection strategy `select`, value choice strategy `choice`, and exploration strategy `explore`. If x is a multi-dimensional array, it is coerced to one-dimensional in row-major order (as with the `array1d` function).

```
annotation set_search(array [$X] of var set of int: x,
                     ann: select,
                     ann: choice)
```

Specify search on variables x , with variable selection strategy `select`, and value choice strategy `choice`. If x is a multi-dimensional array, it is coerced to one-dimensional in row-major order (as with the `array1d` function).

4.2.2.4 Warm start annotations

To be put on the `solve` item, similar to search annotations. A variable can be mentioned several times and in different annotations but only one of the values is taken

Warm start annotations with optional values

The value arrays can contain $<>$ elements (absent values). The following decompositions eliminate those elements because FlatZinc 1.6 does not support optionals.

```
annotation warm_start(array [int] of var bool: x,
                      array [int] of opt bool: v)
```

Specify warm start values v for an array of booleans x

```
annotation warm_start(array [int] of var int: x,
                      array [int] of opt int: v)
```

Specify warm start values v for an array of integers x

```
annotation warm_start(array [int] of var float: x,
                      array [int] of opt float: v)
```

Specify warm start values v for an array of floats x

```
annotation warm_start(array [int] of var set of int: x,
                      array [int] of opt set of int: v)
```

Specify warm start values v for an array of sets x

Other declarations

```
annotation warm_start(array [int] of var bool: x,
                      array [int] of bool: v)
```

Specify warm start values v for an array of booleans x

```
annotation warm_start(array [int] of var int: x, array [int] of int: v)
```

Specify warm start values v for an array of integers x

```
annotation warm_start(array [int] of var float: x,
                      array [int] of float: v)
```

Specify warm start values v for an array of floats x

```
annotation warm_start(array [int] of var set of int: x,
                      array [int] of set of int: v)
```

Specify warm start values v for an array of sets x

```
annotation warm_start_array(array [int] of ann: w)
```

Specify an array w of warm_start annotations or other warm_start_array annotations. Can be useful to keep the annotation order in FlatZinc for manual updating.

Note: if you have search annotations as well, put warm_starts into seq_search in order to have precedence between both, which may matter.

4.2.3 Option type support

These functions and predicates implement the standard library for working with option types. Note that option type support is still incomplete.

4.2.3.1 Option type support for Booleans

```
predicate 'not'(var opt bool: x)
```

Usage: not x

True iff x is absent or false

```
predicate absent(var opt bool: x)
```

True iff x is absent

```
predicate absent(var opt int: x)
```

True iff x is absent

```
predicate bool_eq(var opt bool: b0, var opt bool: b1)
```

True iff both b0 and b1 are absent or both are present and have the same value.

```
predicate bool_eq(var opt bool: b0, var bool: b1)
```

True iff b0 occurs and is equal to b1

```
predicate bool_eq(var bool: b0, var opt bool: b1)
```

True iff b1 occurs and is equal to b0

```
predicate deopt(var opt bool: x)
```

Return value of x (assumes that x is not absent)

```
function var int: deopt(var opt int: x)
```

Return value of x (assumes that x is not absent)

```
function var opt bool: element(var opt int: idx,
                                array [int] of var bool: x)
```

Return absent if idx is absent, otherwise return x [idx]

```
function var opt bool: element(var opt int: idx1,
                               var opt int: idx2,
                               array [int,int] of var bool: x)
```

Return absent if idx1 or idx2 is absent, otherwise return x [idx1 , idx2]

```
function var opt bool: element(var int: idx,
                               array [int] of var opt bool: x)
```

Return x [idx]

```
function var opt bool: element(var int: idx1,
                                var int: idx2,
                                array [int,int] of var opt bool: x)
```

Return $x [idx1 , idx2]$

```
function var opt bool: element(var opt int: idx,
                                array [int] of var opt bool: x)
```

Return absent if idx is absent, otherwise return $x [idx]$

```
function var opt bool: element(var opt int: idx1,
                                var opt int: idx2,
                                array [int,int] of var opt bool: x)
```

Return absent if $idx1$ or $idx2$ is absent, otherwise return $x [idx1 , idx2]$

```
predicate exists(array [int] of var opt bool: x)
```

True iff for at least one i , $x [i]$ occurs and is true

```
predicate forall(array [int] of var opt bool: x)
```

True iff for any i , $x [i]$ is absent or true

```
predicate occurs(var opt bool: x)
```

True iff x is not absent

4.2.3.2 Option type support for integers

```
predicate '<'(var opt int: x, var opt int: y)
```

Usage: $x < y$

Weak comparison: true iff either x or y is absent, or both occur and the value of x is less than the value of y .

```
test '<'(opt int: x, opt int: y)
```

Usage: $x < y$

Weak comparison: true iff either x or y is absent, or both occur and the value of x is less than the value of y .

```
predicate '<='(var opt int: x, var opt int: y)
```

Usage: $x \leq y$

Weak comparison: true iff either x or y is absent, or both occur and the value of x is less than or equal to the value of y .

```
test '<='(opt int: x, opt int: y)
```

Usage: $x \leq y$

Weak comparison: true iff either x or y is absent, or both occur and the value of x is less than or equal to the value of y .

```
predicate '>'(var opt int: x, var opt int: y)
```

Usage: $x > y$

Weak comparison: true iff either x or y is absent, or both occur and the value of x is greater than the value of y .

```
test '>'(opt int: x, opt int: y)
```

Usage: $x > y$

Weak comparison: true iff either x or y is absent, or both occur and the value of x is greater than the value of y .

```
predicate '>='(var opt int: x, var opt int: y)
```

Usage: $x \geq y$

Weak comparison: true iff either x or y is absent, or both occur and the value of x is greater than or equal to the value of y .

```
test '>='(opt int: x, opt int: y)
```

Usage: $x \geq y$

Weak comparison: true iff either x or y is absent, or both occur and the value of x is greater than or equal to the value of y .

```
function var opt int: bool2int(var opt bool: x)
```

Return optional 0/1 integer that is absent iff x is absent, and 1 iff x occurs and is true.

```
function var opt int: element(var opt int: idx,  
                             array [int] of var int: x)
```

Return absent if idx is absent, otherwise return x [idx]

```
function var opt int: element(var opt int: idx1,  
                             var opt int: idx2,  
                             array [int,int] of var int: x)
```

Return absent if idx1 or idx2 is absent, otherwise return x [idx1 , idx2]

```
function var opt int: element(var int: idx,  
                             array [int] of var opt int: x)
```

Return x [idx]

```
function var opt int: element(var int: idx1,  
                             var int: idx2,  
                             array [int,int] of var opt int: x)
```

Return x [idx1 , idx2]

```
function var opt int: element(var opt int: idx,  
                             array [int] of var opt int: x)
```

Return absent if idx is absent, otherwise return x [idx]

```
function var opt int: element(var opt int: idx1,  
                             var opt int: idx2,  
                             array [int,int] of var opt int: x)
```

Return absent if idx1 or idx2 is absent, otherwise return x [idx1 , idx2]

```
predicate int_eq(var opt int: x, var opt int: y)
```

True iff both x and y are absent or both are present and have the same value.

```
predicate int_ne(var opt int: x, var opt int: y)
```

True iff only one of x and y is absent or both are present and have different values.

```
function var opt int: max(array [int] of var opt int: x)
```

Return maximum of elements in x that are not absent, or absent if all elements in x are absent.

```
function var opt int: min(array [int] of var opt int: x)
```

Return minimum of elements in x that are not absent, or absent if all elements in x are absent.

```
predicate occurs(var opt int: x)
```

True iff x is not absent

```
function var int: product(array [int] of var opt int: x)
```

Return product of non-absent elements of x .

```
function var int: sum(array [int] of var opt int: x)
```

Return sum of non-absent elements of x .

```
function var opt int: ~*(var opt int: x, var opt int: y)
```

Weak multiplication. Return product of x and y if both are present, otherwise return absent.

```
function var opt int: ~+(var opt int: x, var opt int: y)
```

Weak addition. Return sum of x and y if both are present, otherwise return absent.

```
function var opt int: ~-(var opt int: x, var opt int: y)
```

Weak subtraction. Return difference of x and y if both are present, otherwise return absent.

```
predicate ~=(var opt int: x, var opt int: y)
```

Weak equality. True if either x or y are absent, or present and equal.

4.2.3.3 Other declarations

```
test absent(var $T: x)
```

Test if x is absent (always returns false)

```
test absent(opt $T: x)
```

Test if x is absent

```
function $T: deopt(opt $T: x)
```

Return value of `x` if `x` is not absent. Aborts when evaluated on absent value.

```
function var $T: deopt(var $T: x)
```

Return value `x` unchanged (since `x` is guaranteed to be non-optional).

```
test occurs(var $T: x)
```

Test if `x` is not absent (always returns true)

```
test occurs(opt $T: x)
```

Test if `x` is not absent

4.2.4 Compiler options

4.2.4.1 Parameters

```
opt bool: mzn_opt_only_range_domains
```

Whether to only generate domains that are contiguous ranges

```
opt int: mzn_min_version_required
```

If defined, this can be used to check that the MiniZinc compiler supports all the features used in the model.

4.2.4.2 Functions and Predicates

```
test mzn_check_only_range_domains()
```

Check whether to only generate domains that are contiguous ranges

4.2.5 Builtins

These functions and predicates define built-in operations of the MiniZinc language.

4.2.5.1 Comparison Builtins

These builtins implement comparison operations.

```
test '!='($T: x, $T: y)
```

Usage: `x != y`

Return if `x` is not equal to `y`

```
predicate '!='(var $T: x, var $T: y)
```

Usage: `x != y`

Return if `x` is not equal to `y`

```
test '!='(array [$U] of $T: x, array [$U] of $T: y)
```

Usage: `x != y`

Return if array `x` is not equal to array `y`

```
predicate '!='(array [$U] of var $T: x, array [$U] of var $T: y)
```

Usage: `x != y`

Return if array `x` is not equal to array `y`

```
test '<'($T: x, $T: y)
```

Usage: `x < y`

Return if `x` is less than `y`

```
predicate '<'(var $T: x, var $T: y)
```

Usage: `x < y`

Return if `x` is less than `y`

```
test '<'(array [$U] of $T: x, array [$U] of $T: y)
```

Usage: `x < y`

Return if array `x` is lexicographically smaller than array `y`

```
predicate '<'(array [\$U] of var \$T: x, array [\$U] of var \$T: y)
```

Usage: $x < y$

Return if array x is lexicographically smaller than array y

```
test '<='(\$T: x, \$T: y)
```

Usage: $x \leq y$

Return if x is less than or equal to y

```
predicate '<='(var \$T: x, var \$T: y)
```

Usage: $x \leq y$

Return if x is less than or equal to y

```
test '<='(array [\$U] of \$T: x, array [\$U] of \$T: y)
```

Usage: $x \leq y$

Return if array x is lexicographically smaller than or equal to array y

```
predicate '<='(array [\$U] of var \$T: x, array [\$U] of var \$T: y)
```

Usage: $x \leq y$

Return if array x is lexicographically smaller than or equal to array y

```
test '='(\$T: x, \$T: y)
```

Usage: $x = y$

Return if x is equal to y

```
test '='(opt \$T: x, opt \$T: y)
```

Usage: $x = y$

Return if x is equal to y

```
predicate '='(var \$T: x, var \$T: y)
```

Usage: $x = y$

Return if x is equal to y

```
predicate '='(var opt $T: x, var opt $T: y)
```

Usage: `x = y`

Return if `x` is equal to `y`

```
test '='(array [$T] of int: x, array [$T] of int: y)
```

Usage: `x = y`

Return if array `x` is equal to array `y`

```
predicate '='(array [$T] of var int: x, array [$T] of var int: y)
```

Usage: `x = y`

Return if array `x` is equal to array `y`

```
test '='(array [$T] of bool: x, array [$T] of bool: y)
```

Usage: `x = y`

Return if array `x` is equal to array `y`

```
predicate '='(array [$T] of var bool: x, array [$T] of var bool: y)
```

Usage: `x = y`

Return if array `x` is equal to array `y`

```
test '='(array [$T] of set of int: x, array [$T] of set of int: y)
```

Usage: `x = y`

Return if array `x` is equal to array `y`

```
predicate '='(array [$T] of var set of int: x,
            array [$T] of var set of int: y)
```

Usage: `x = y`

Return if array `x` is equal to array `y`

```
test '='(array [$T] of float: x, array [$T] of float: y)
```

Usage: `x = y`

Return if array `x` is equal to array `y`

```
predicate '='(array [$T] of var float: x, array [$T] of var float: y)
```

Usage: $x = y$

Return if array x is equal to array y

```
test '>'($T: x, $T: y)
```

Usage: $x > y$

Return if x is greater than y

```
predicate '>'(var $T: x, var $T: y)
```

Usage: $x > y$

Return if x is greater than y

```
test '>'(array [$U] of $T: x, array [$U] of $T: y)
```

Usage: $x > y$

Return if array x is lexicographically greater than array y

```
predicate '>'(array [$U] of var $T: x, array [$U] of var $T: y)
```

Usage: $x > y$

Return if array x is lexicographically greater than array y

```
test '>='($T: x, $T: y)
```

Usage: $x \geq y$

Return if x is greater than or equal to y

```
predicate '>='(var $T: x, var $T: y)
```

Usage: $x \geq y$

Return if x is greater than or equal to y

```
test '>='(array [$U] of $T: x, array [$U] of $T: y)
```

Usage: $x \geq y$

Return if array x is lexicographically greater than or equal to array y

4.2.5.2 Arithmetic Builtins

These builtins implement arithmetic operations.

```
function int: '*'(int: x, int: y)
```

Usage: $x * y$

Return $x * y$

```
function var int: '*'(var int: x, var int: y)
```

Usage: $x * y$

Return $x * y$

```
function float: '*'(float: x, float: y)
```

Usage: $x * y$

Return $x * y$

```
function var float: '*'(var float: x, var float: y)
```

Usage: $x * y$

Return $x * y$

```
function int: '+'(int: x, int: y)
```

Usage: $x + y$

Return $x + y$

```
function var int: '+'(var int: x, var int: y)
```

Usage: $x + y$

Return $x + y$

```
function float: '+'(float: x, float: y)
```

Usage: $x + y$

Return $x + y$

```
function var float: '+'(var float: x, var float: y)
```

Usage: $x + y$

Return $x + y$

```
function int: '-'(int: x, int: y)
```

Usage: $x - y$

Return $x - y$

```
function var int: '-'(var int: x, var int: y)
```

Usage: $x - y$

Return $x - y$

```
function float: '-'(float: x, float: y)
```

Usage: $x - y$

Return $x - y$

```
function var float: '-'(var float: x, var float: y)
```

Usage: $x - y$

Return $x - y$

```
function int: '-'(int: x)
```

Usage: $-x$

Return negative x

```
function var int: '-'(var int: x)
```

Usage: $-x$

Return negative x

```
function float: '-'(float: x)
```

Usage: $-x$

Return negative x

```
function var float: '-'(var float: x)
```

Usage: `- x`

Return negative `x`

```
function float: '/'(float: x, float: y)
```

Usage: `x / y`

Return result of floating point division `x / y`

```
function var float: '/'(var float: x, var float: y)
```

Usage: `x / y`

Return result of floating point division `x / y`

```
function int: '^'(int: x, int: y)
```

Usage: `x ^ y`

Return `x ^ y`

```
function var int: '^'(var int: x, var int: y)
```

Usage: `x ^ y`

Return `x ^ y`

```
function float: '^'(float: x, float: y)
```

Usage: `x ^ y`

Return `x ^ y`

```
function var float: '^'(var float: x, var float: y)
```

Usage: `x ^ y`

Return `x ^ y`

```
function int: 'div'(int: x, int: y)
```

Usage: `x div y`

Return result of integer division `x / y`

```
function var int: 'div'(var int: x, var int: y)
```

Usage: `x div y`

Return result of integer division x / y

```
function int: 'mod'(int: x, int: y)
```

Usage: `x mod y`

Return remainder of integer division $x \% y$

```
function var int: 'mod'(var int: x, var int: y)
```

Usage: `x mod y`

Return remainder of integer division $x \% y$

```
function int: abs(int: x)
```

Return absolute value of x

```
function var int: abs(var int: x)
```

Return absolute value of x

```
function float: abs(float: x)
```

Return absolute value of x

```
function var float: abs(var float: x)
```

Return absolute value of x

```
function $$E: arg_max(array [$$E] of int: x)
```

Return index of maximum of elements in array x

```
function $$E: arg_max(array [$$E] of float: x)
```

Return index of maximum of elements in array x

```
function $$E: arg_min(array [$$E] of int: x)
```

Return index of minimum of elements in array x

```
function $$E: arg_min(array [$$E] of float: x)
```

Return index of minimum of elements in array x

```
function $T: max($T: x, $T: y)
```

Return maximum of x and y

```
function $T: max(array [$U] of $T: x)
```

Return maximum of elements in array x

```
function $$E: max(set of $$E: x)
```

Return maximum of elements in set x

```
function var int: max(var int: x, var int: y)
```

Return maximum of x and y

```
function var int: max(array [$U] of var int: x)
```

Return maximum of elements in array x

```
function var float: max(var float: x, var float: y)
```

Return maximum of x and y

```
function var float: max(array [$U] of var float: x)
```

Return maximum of elements in array x

```
function $T: min($T: x, $T: y)
```

Return minimum of x and y

```
function $T: min(array [$U] of $T: x)
```

Return minimum of elements in array x

```
function $$E: min(set of $$E: x)
```

Return minimum of elements in set x

```
function var int: min(var int: x, var int: y)
```

Return minimum of x and y

```
function var int: min(array [$U] of var int: x)
```

Return minimum of elements in array x

```
function var float: min(var float: x, var float: y)
```

Return minimum of x and y

```
function var float: min(array [$U] of var float: x)
```

Return minimum of elements in array x

```
function int: pow(int: x, int: y)
```

Return x^y

```
function var int: pow(var int: x, var int: y)
```

Return x^y

```
function float: pow(float: x, float: y)
```

Return x^y

```
function var float: pow(var float: x, var float: y)
```

Return x^y

```
function int: product(array [$T] of int: x)
```

Return product of elements in array x

```
function var int: product(array [$T] of var int: x)
```

Return product of elements in array x

```
function float: product(array [$T] of float: x)
```

Return product of elements in array x

```
function var float: product(array [$T] of var float: x)
```

Return product of elements in array x

```
function float: sqrt(float: x)
```

Return \sqrt{x}

```
function var float: sqrt(var float: x)
```

Return \sqrt{x}

```
function int: sum(array [$T] of int: x)
```

Return sum of elements in array x

```
function var int: sum(array [$T] of var int: x)
```

Return sum of elements in array x

```
function float: sum(array [$T] of float: x)
```

Return sum of elements in array x

```
function var float: sum(array [$T] of var float: x)
```

Return sum of elements in array x

4.2.5.3 Exponential and logarithmic builtins

These builtins implement exponential and logarithmic functions.

```
function float: exp(float: x)
```

Return e^x

```
function var float: exp(var float: x)
```

Return e^x

```
function float: ln(float: x)
```

Return $\ln x$

```
function var float: ln(var float: x)
```

Return $\ln x$

```
function float: log(float: x, float: y)
```

Return $\log_x y$

```
function float: log10(float: x)
```

Return $\log_{10} x$

```
function var float: log10(var float: x)
```

Return $\log_{10} x$

```
function float: log2(float: x)
```

Return $\log_2 x$

```
function var float: log2(var float: x)
```

Return $\log_2 x$

4.2.5.4 Trigonometric functions

These builtins implement the standard trigonometric functions.

```
function float: acos(float: x)
```

Return $\arccos x$

```
function var float: acos(var float: x)
```

Return $\arccos x$

```
function float: acosh(float: x)
```

Return $\text{acosh } x$

```
function var float: acosh(var float: x)
```

Return $\text{acosh } x$

```
function float: asin(float: x)
```

Return $\arcsin x$

```
function var float: asin(var float: x)
```

Return $\arcsin x$

```
function float: asinh(float: x)
```

Return $\text{asinh } x$

```
function var float: asinh(var float: x)
```

Return $\text{asinh } x$

```
function float: atan(float: x)
```

Return $\arctan x$

```
function var float: atan(var float: x)
```

Return $\arctan x$

```
function float: atanh(float: x)
```

Return $\text{atanh } x$

```
function var float: atanh(var float: x)
```

Return $\text{atanh } x$

```
function float: cos(float: x)
```

Return $\cos x$

```
function var float: cos(var float: x)
```

Return $\cos x$

```
function float: cosh(float: x)
```

Return $\cosh x$

```
function var float: cosh(var float: x)
```

Return $\cosh x$

```
function float: sin(float: x)
```

Return $\sin x$

```
function var float: sin(var float: x)
```

Return $\sin x$

```
function float: sinh(float: x)
```

Return $\sinh x$

```
function var float: sinh(var float: x)
```

Return $\sinh x$

```
function float: tan(float: x)
```

Return $\tan x$

```
function var float: tan(var float: x)
```

Return $\tan x$

```
function float: tanh(float: x)
```

Return $\tanh x$

```
function var float: tanh(var float: x)
```

Return $\tanh x$

4.2.5.5 Logical operations

Logical operations are the standard operators of Boolean logic.

```
test ' $\rightarrow$ '(bool: x, bool: y)
```

Usage: `x -> y`

Return truth value of `x` implies `y`

```
predicate '->'(var bool: x, var bool: y)
```

Usage: `x -> y`

Return truth value of `x` implies `y`

```
test '/\'(bool: x, bool: y)
```

Usage: `x /\ y`

Return truth value of `x ∧ y`

```
predicate '/\'(var bool: x, var bool: y)
```

Usage: `x /\ y`

Return truth value of `x ∧ y`

```
test '<-'(bool: x, bool: y)
```

Usage: `x <- y`

Return truth value of `y` implies `x`

```
predicate '<-'(var bool: x, var bool: y)
```

Usage: `x <- y`

Return truth value of `y` implies `x`

```
test '<->'(bool: x, bool: y)
```

Usage: `x <-> y`

Return truth value of `x` if-and-only-if `y`

```
predicate '<->'(var bool: x, var bool: y)
```

Usage: `x <-> y`

Return truth value of `x` if-and-only-if `y`

```
test '\/'(bool: x, bool: y)
```

Usage: $x \vee y$

Return truth value of $x \vee y$

```
predicate '\vee'(var bool: x, var bool: y)
```

Usage: $x \vee y$

Return truth value of $x \vee y$

```
test 'not'(bool: x)
```

Usage: $\text{not } x$

Return truth value of the negation of x

```
predicate 'not'(var bool: x)
```

Usage: $\text{not } x$

Return truth value of the negation of x

```
test 'xor'(bool: x, bool: y)
```

Usage: $x \oplus y$

Return truth value of $x \oplus y$

```
predicate 'xor'(var bool: x, var bool: y)
```

Usage: $x \oplus y$

Return truth value of $x \oplus y$

```
predicate clause(array [\$T] of var bool: x, array [\$T] of var bool: y)
```

Return truth value of $(\bigvee_i x[i]) \vee (\bigvee_j \neg y[j])$

```
predicate clause(array [\$T] of bool: x, array [\$T] of bool: y)
```

Return truth value of $(\bigvee_i x[i]) \vee (\bigvee_j \neg y[j])$

```
test exists(array [\$T] of bool: x)
```

Return truth value of $\bigvee_i x[i]$

```
predicate exists(array [\$T] of var bool: x)
```

Return truth value of $\bigvee_i \mathbf{x}[i]$

```
test forall(array [\$T] of bool: x)
```

Return truth value of $\bigwedge_i \mathbf{x}[i]$

```
predicate forall(array [\$T] of var bool: x)
```

Return truth value of $\bigwedge_i \mathbf{x}[i]$

```
test iffall(array [\$T] of bool: x)
```

Return truth value of true $\oplus (\bigoplus_i \mathbf{x}[i])$

```
predicate iffall(array [\$T] of var bool: x)
```

Return truth value of true $\oplus (\bigoplus_i \mathbf{x}[i])$

```
test xorall(array [\$T] of bool: x)
```

Return truth value of $\bigoplus_i \mathbf{x}[i]$

```
predicate xorall(array [\$T] of var bool: x)
```

Return truth value of $\bigoplus_i \mathbf{x}[i]$

4.2.5.6 Set operations

These functions implement the basic operations on sets.

```
function set of $$E: '...'($$E: a, $$E: b)
```

Usage: a .. b

Return the set {a, ..., b}

```
function set of float: '...' (float: a, float: b)
```

Usage: a .. b

Return the set {a, ..., b}

```
function set of $T: 'diff'(set of $T: x, set of $T: y)
```

Usage: `x diff y`

Return the set difference of sets `x − y`

```
function var set of $$T: 'diff'(var set of $$T: x, var set of $$T: y)
```

Usage: `x diff y`

Return the set difference of sets `x − y`

```
test 'in'(int: x, set of int: y)
```

Usage: `x in y`

Test if `x` is an element of the set `y`

```
predicate 'in'(var int: x, var set of int: y)
```

Usage: `x in y`

`x` is an element of the set `y`

```
test 'in'(float: x, set of float: y)
```

Usage: `x in y`

Test if `x` is an element of the set `y`

```
predicate 'in'(var float: x, set of float: y)
```

Usage: `x in y`

Test if `x` is an element of the set `y`

```
function set of $T: 'intersect'(set of $T: x, set of $T: y)
```

Usage: `x intersect y`

Return the intersection of sets `x` and `y`

```
function var set of $$T: 'intersect'(var set of $$T: x,
                                         var set of $$T: y)
```

Usage: `x intersect y`

Return the intersection of sets `x` and `y`

```
test 'subset'(set of $T: x, set of $T: y)
```

Usage: `x subset y`

Test if `x` is a subset of `y`

```
predicate 'subset'(var set of int: x, var set of int: y)
```

Usage: `x subset y`

`x` is a subset of `y`

```
test 'superset'(set of $T: x, set of $T: y)
```

Usage: `x superset y`

Test if `x` is a superset of `y`

```
predicate 'superset'(var set of int: x, var set of int: y)
```

Usage: `x superset y`

`x` is a superset of `y`

```
function set of $T: 'symdiff'(set of $T: x, set of $T: y)
```

Usage: `x symdiff y`

Return the symmetric set difference of sets `x` and `y`

```
function var set of $$T: 'symdiff'(var set of $$T: x,
                                         var set of $$T: y)
```

Usage: `x symdiff y`

Return the symmetric set difference of sets `x` and `y`

```
function set of $T: 'union'(set of $T: x, set of $T: y)
```

Usage: `x union y`

Return the union of sets `x` and `y`

```
function var set of $$T: 'union'(var set of $$T: x, var set of $$T: y)
```

Usage: `x union y`

Return the union of sets `x` and `y`

```
function set of $U: array_intersect(array [$T] of set of $U: x)
```

Return the intersection of the sets in array x

```
function var set of int: array_intersect(array [int] of var set of int: x)
```

Return the intersection of the sets in array x

```
function set of $U: array_union(array [$T] of set of $U: x)
```

Return the union of the sets in array x

```
function var set of int: array_union(array [int] of var set of int: x)
```

Return the union of the sets in array x

```
function int: card(set of $T: x)
```

Return the cardinality of the set x

```
function var int: card(var set of int: x)
```

Return the cardinality of the set x

```
function var $$E: max(var set of $$E: s)
```

Return the maximum of the set s

```
function var $$E: min(var set of $$E: s)
```

Return the minimum of the set s

4.2.5.7 Array operations

These functions implement the basic operations on arrays.

```
function array [int] of $T: '+'(array [int] of $T: x,
                                array [int] of $T: y)
```

Usage: x ++ y

Return the concatenation of arrays x and y

```
function array [int] of opt $T: '+'(array [int] of opt $T: x,
                                         array [int] of opt $T: y)
```

Usage: `x ++ y`

Return the concatenation of arrays `x` and `y`

```
function array [int] of var $T: '+'(array [int] of var $T: x,
                                         array [int] of var $T: y)
```

Usage: `x ++ y`

Return the concatenation of arrays `x` and `y`

```
function array [int] of var opt $T: '+'(array [int] of var opt $T: x,
                                         array [int] of var opt $T: y)
```

Usage: `x ++ y`

Return the concatenation of arrays `x` and `y`

```
function array [int] of $V: array1d(array [$U] of $V: x)
```

Return array `x` coerced to index set `1..length(x)`. Coercions are performed by considering the array `x` in row-major order.

```
function array [int] of opt $V: array1d(array [$U] of opt $V: x)
```

Return array `x` coerced to index set `1..length(x)`. Coercions are performed by considering the array `x` in row-major order.

```
function array [int] of var $V: array1d(array [$U] of var $V: x)
```

Return array `x` coerced to index set `1..length(x)`. Coercions are performed by considering the array `x` in row-major order.

```
function array [int] of var opt $V: array1d(array [$U] of var opt $V: x)
```

Return array `x` coerced to index set `1..length(x)`. Coercions are performed by considering the array `x` in row-major order.

```
function array [$$E] of $V: array1d(set of $$E: S, array [$U] of $V: x)
```

Return array `x` coerced to one-dimensional array with index set `S`. Coercions are performed by considering the array `x` in row-major order.

```
function array [$$E] of opt $V: array1d(set of $$E: S,
                                         array [$U] of opt $V: x)
```

Return array x coerced to one-dimensional array with index set S . Coercions are performed by considering the array x in row-major order.

```
function array [$$E] of var $V: array1d(set of $$E: S,
                                         array [$U] of var $V: x)
```

Return array x coerced to one-dimensional array with index set S . Coercions are performed by considering the array x in row-major order.

```
function array [$$E] of var opt $V: array1d(set of $$E: S,
                                              array [$U] of var opt $V: x)
```

Return array x coerced to one-dimensional array with index set S . Coercions are performed by considering the array x in row-major order.

```
function array [$$E,$$F] of $V: array2d(set of $$E: S1,
                                         set of $$F: S2,
                                         array [$U] of $V: x)
```

Return array x coerced to two-dimensional array with index sets $S1$ and $S2$. Coercions are performed by considering the array x in row-major order.

```
function array [$$E,$$F] of opt $V: array2d(set of $$E: S1,
                                              set of $$F: S2,
                                              array [$U] of opt $V: x)
```

Return array x coerced to two-dimensional array with index sets $S1$ and $S2$. Coercions are performed by considering the array x in row-major order.

```
function array [$$E,$$F] of var $V: array2d(set of $$E: S1,
                                              set of $$F: S2,
                                              array [$U] of var $V: x)
```

Return array x coerced to two-dimensional array with index sets $S1$ and $S2$. Coercions are performed by considering the array x in row-major order.

```
function array [$$E,$$F] of var opt $V: array2d(set of $$E: S1,
                                              set of $$F: S2,
                                              array [$U] of var opt $V: x)
```

Return array x coerced to two-dimensional array with index sets $S1$ and $S2$. Coercions are performed by considering the array x in row-major order.

```
function array [ $$E, $$F, $$G] of $V: array3d(set of $$E: S1,
                                              set of $$F: S2,
                                              set of $$G: S3,
                                              array [$U] of $V: x)
```

Return array x coerced to three-dimensional array with index sets S_1 , S_2 and S_3 . Coercions are performed by considering the array x in row-major order.

```
function array [ $$E, $$F, $$G] of opt $V: array3d(set of $$E: S1,
                                                 set of $$F: S2,
                                                 set of $$G: S3,
                                                 array [$U] of opt $V: x)
```

Return array x coerced to three-dimensional array with index sets S_1 , S_2 and S_3 . Coercions are performed by considering the array x in row-major order.

```
function array [ $$E, $$F, $$G] of var $V: array3d(set of $$E: S1,
                                                 set of $$F: S2,
                                                 set of $$G: S3,
                                                 array [$U] of var $V: x)
```

Return array x coerced to three-dimensional array with index sets S_1 , S_2 and S_3 . Coercions are performed by considering the array x in row-major order.

```
function array [ $$E, $$F, $$G] of var opt $V: array3d(set of $$E: S1,
                                                       set of $$F: S2,
                                                       set of $$G: S3,
                                                       array [$U] of var opt
                                                       ↵$V: x)
```

Return array x coerced to three-dimensional array with index sets S_1 , S_2 and S_3 . Coercions are performed by considering the array x in row-major order.

```
function array [ $$E, $$F, $$G, $$H] of $V: array4d(set of $$E: S1,
                                                       set of $$F: S2,
                                                       set of $$G: S3,
                                                       set of $$H: S4,
                                                       array [$U] of $V: x)
```

Return array x coerced to 4-dimensional array with index sets S_1 , S_2 , S_3 and S_4 . Coercions are performed by considering the array x in row-major order.

```
function array [ $$E, $$F, $$G, $$H] of opt $V: array4d(set of $$E: S1,
                                                       set of $$F: S2,
                                                       set of $$G: S3,
                                                       set of $$H: S4,
```

```
array [$U] of opt $V: x)
```

Return array x coerced to 4-dimensional array with index sets S_1, S_2, S_3 and S_4 . Coercions are performed by considering the array x in row-major order.

```
function array [$$E, $$F, $$G, $$H] of var $V: array4d(set of $$E: S1,
                                                    set of $$F: S2,
                                                    set of $$G: S3,
                                                    set of $$H: S4,
                                                    array [$U] of var $V: x)
```

Return array x coerced to 4-dimensional array with index sets S_1, S_2, S_3 and S_4 . Coercions are performed by considering the array x in row-major order.

```
function array [$$E, $$F, $$G, $$H] of var opt $V: array4d(set of $$E: S1,
                                                       set of $$F: S2,
                                                       set of $$G: S3,
                                                       set of $$H: S4,
                                                       array [$U] of var_
→opt $V: x)
```

Return array x coerced to 4-dimensional array with index sets S_1, S_2, S_3 and S_4 . Coercions are performed by considering the array x in row-major order.

```
function array [$$E, $$F, $$G, $$H, $$I] of $V: array5d(set of $$E: S1,
                                                       set of $$F: S2,
                                                       set of $$G: S3,
                                                       set of $$H: S4,
                                                       set of $$I: S5,
                                                       array [$U] of $V: x)
```

Return array x coerced to 5-dimensional array with index sets S_1, S_2, S_3, S_4 and S_5 . Coercions are performed by considering the array x in row-major order.

```
function array [$$E, $$F, $$G, $$H, $$I] of opt $V: array5d(set of $$E: S1,
                                                       set of $$F: S2,
                                                       set of $$G: S3,
                                                       set of $$H: S4,
                                                       set of $$I: S5,
                                                       array [$U] of opt
→$V: x)
```

Return array x coerced to 5-dimensional array with index sets S_1, S_2, S_3, S_4 and S_5 . Coercions are performed by considering the array x in row-major order.

```
function array [$$E,$$F,$$G,$$H,$$I] of var $V: array5d(set of $$E: S1,
                                                    set of $$F: S2,
                                                    set of $$G: S3,
                                                    set of $$H: S4,
                                                    set of $$I: S5,
                                                    array [$U] of var
→$V: x)
```

Return array x coerced to 5-dimensional array with index sets S_1, S_2, S_3, S_4 and S_5 . Coercions are performed by considering the array x in row-major order.

```
function array [$$E,$$F,$$G,$$H,$$I] of var opt $V: array5d(set of $$E: S1,
                                                       set of $$F: S2,
                                                       set of $$G: S3,
                                                       set of $$H: S4,
                                                       set of $$I: S5,
                                                       array [$U] of_
→var opt $V: x)
```

Return array x coerced to 5-dimensional array with index sets S_1, S_2, S_3, S_4 and S_5 . Coercions are performed by considering the array x in row-major order.

```
function array [$$E,$$F,$$G,$$H,$$I,$$J] of $V: array6d(set of $$E: S1,
                                                       set of $$F: S2,
                                                       set of $$G: S3,
                                                       set of $$H: S4,
                                                       set of $$I: S5,
                                                       set of $$J: S6,
                                                       array [$U] of $V: x)
```

Return array x coerced to 6-dimensional array with index sets S_1, S_2, S_3, S_4, S_5 and S_6 . Coercions are performed by considering the array x in row-major order.

```
function array [$$E,$$F,$$G,$$H,$$I,$$J] of opt $V: array6d(set of $$E: S1,
                                                       set of $$F: S2,
                                                       set of $$G: S3,
                                                       set of $$H: S4,
                                                       set of $$I: S5,
                                                       set of $$J: S6,
                                                       array [$U] of_
→opt $V: x)
```

Return array x coerced to 6-dimensional array with index sets S_1, S_2, S_3, S_4, S_5 and S_6 . Coercions are performed by considering the array x in row-major order.

```
function array [$$E,$$F,$$G,$$H,$$I,$$J] of var $V: array6d(set of $$E: S1,
                                                       set of $$F: S2,
```

```

set of $$G: S3,
set of $$H: S4,
set of $$I: S5,
set of $$J: S6,
array [\$U] of_
→var $V: x)

```

Return array `x` coerced to 6-dimensional array with index sets `S1` , `S2` , `S3` , `S4` , `S5` and `S6` . Coercions are perfomed by considering the array `x` in row-major order.

```

function array [$$E,$$F,$$G,$$H,$$I,$$J] of var opt $V: array6d(set of $$E:_  

→S1,  

                               set of $$F:_  

→S2,  

                               set of $$G:_  

→S3,  

                               set of $$H:_  

→S4,  

                               set of $$I:_  

→S5,  

                               set of $$J:_  

→S6,  

                               array [\$U]_  

→of var opt $V: x)

```

Return array `x` coerced to 6-dimensional array with index sets `S1` , `S2` , `S3` , `S4` , `S5` and `S6` . Coercions are perfomed by considering the array `x` in row-major order.

```

function array [$T] of $V: arrayXd(array [$T] of var opt $X: x,  

                                         array [\$U] of $V: y)

```

Return array `y` coerced to array with same number of dimensions and same index sets as array `x` . Coercions are perfomed by considering the array `y` in row-major order.

```

function array [$T] of opt $V: arrayXd(array [$T] of var opt $X: x,  

                                         array [\$U] of opt $V: y)

```

Return array `y` coerced to array with same number of dimensions and same index sets as array `x` . Coercions are perfomed by considering the array `y` in row-major order.

```

function array [$T] of var $V: arrayXd(array [$T] of var opt $X: x,  

                                         array [\$U] of var $V: y)

```

Return array `y` coerced to array with same number of dimensions and same index sets as array `x` . Coercions are perfomed by considering the array `y` in row-major order.

```
function array [\$T] of var opt \$V: arrayXd(array [\$T] of var opt \$X: x,
                                              array [\$U] of var opt \$V: y)
```

Return array y coerced to array with same number of dimensions and same index sets as array x . Coercions are performed by considering the array y in row-major order.

```
function array [$$E] of \$T: col(array [$$E,int] of \$T: x, int: c)
```

Return column c of array x

```
function array [$$E] of opt \$T: col(array [$$E,int] of opt \$T: x,
                                         int: c)
```

Return column c of array x

```
function array [$$E] of var \$T: col(array [$$E,int] of var \$T: x,
                                         int: c)
```

Return column c of array x

```
function array [$$E] of var opt \$T: col(array [$$E,int] of var opt \$T: x,
                                         int: c)
```

Return column c of array x

```
test has_element(\$T: e, array [int] of \$T: x)
```

Test if e is an element of array x

```
test has_element(\$T: e, array [int] of opt \$T: x)
```

Test if e is an element of array x

```
predicate has_element(\$T: e, array [$$E] of var opt \$T: x)
```

Test if e is an element of array x

```
test has_index(int: i, array [int] of var opt \$T: x)
```

Test if i is in the index set of x

```
function set of $$E: index_set(array [$$E] of var opt \$U: x)
```

Return index set of one-dimensional array x

```
function set of $$E: index_set_1of2(array [$$E,int] of var opt $U: x)
```

Return index set of first dimension of two-dimensional array x

```
function set of $$E: index_set_1of3(array [$$E,int,int] of var opt $U: x)
```

Return index set of first dimension of 3-dimensional array x

```
function set of $$E: index_set_1of4(array [$$E,int,int,int] of var opt $U: x)
```

Return index set of first dimension of 4-dimensional array x

```
function set of $$E: index_set_1of5(array [$$E,int,int,int,int] of var opt  
→ $U: x)
```

Return index set of first dimension of 5-dimensional array x

```
function set of $$E: index_set_1of6(array [$$E,int,int,int,int,int] of var  
→ opt $U: x)
```

Return index set of first dimension of 6-dimensional array x

```
function set of $$E: index_set_2of2(array [int,$$E] of var opt $U: x)
```

Return index set of second dimension of two-dimensional array x

```
function set of $$E: index_set_2of3(array [int,$$E,int] of var opt $U: x)
```

Return index set of second dimension of 3-dimensional array x

```
function set of $$E: index_set_2of4(array [int,$$E,int,int] of var opt $U: x)
```

Return index set of second dimension of 4-dimensional array x

```
function set of $$E: index_set_2of5(array [int,$$E,int,int,int] of var opt  
→ $U: x)
```

Return index set of second dimension of 5-dimensional array x

```
function set of $$E: index_set_2of6(array [int,$$E,int,int,int,int] of var  
→ opt $U: x)
```

Return index set of second dimension of 6-dimensional array x

```
function set of $$E: index_set_3of3(array [int,int,$$E] of var opt $U: x)
```

Return index set of third dimension of 3-dimensional array x

```
function set of $$E: index_set_3of4(array [int,int,$$E,int] of var opt $U: x)
```

Return index set of third dimension of 4-dimensional array x

```
function set of $$E: index_set_3of5(array [int,int,$$E,int,int] of var opt  
→ $U: x)
```

Return index set of third dimension of 5-dimensional array x

```
function set of $$E: index_set_3of6(array [int,int,$$E,int,int,int] of var  
→ opt $U: x)
```

Return index set of third dimension of 6-dimensional array x

```
function set of $$E: index_set_4of4(array [int,int,int,$$E] of var opt $U: x)
```

Return index set of fourth dimension of 4-dimensional array x

```
function set of $$E: index_set_4of5(array [int,int,int,$$E,int] of var opt  
→ $U: x)
```

Return index set of fourth dimension of 5-dimensional array x

```
function set of $$E: index_set_4of6(array [int,int,int,$$E,int,int] of var  
→ opt $U: x)
```

Return index set of fourth dimension of 6-dimensional array x

```
function set of $$E: index_set_5of5(array [int,int,int,int,$$E] of var opt  
→ $U: x)
```

Return index set of fifth dimension of 5-dimensional array x

```
function set of $$E: index_set_5of6(array [int,int,int,int,$$E,int] of var  
→ opt $U: x)
```

Return index set of fifth dimension of 6-dimensional array x

```
function set of $$E: index_set_6of6(array [int,int,int,int,int,$$E] of var_
→opt $U: x)
```

Return index set of sixth dimension of 6-dimensional array x

```
test index_sets_agree(array [$T] of var opt $U: x,
                      array [$T] of var opt $W: y)
```

Test if x and y have the same index sets

```
function int: length(array [$T] of var opt $U: x)
```

Return the length of array x

Note that the length is defined as the number of elements in the array, regardless of its dimensionality.

```
function array [$$E] of $T: reverse(array [$$E] of $T: x)
```

Return the array x in reverse order

The resulting array has the same index set as x .

```
function array [$$E] of opt $T: reverse(array [$$E] of opt $T: x)
```

Return the array x in reverse order

The resulting array has the same index set as x .

```
function array [$$E] of var $T: reverse(array [$$E] of var $T: x)
```

Return the array x in reverse order

The resulting array has the same index set as x .

```
function array [$$E] of var opt $T: reverse(array [$$E] of var opt $T: x)
```

Return the array x in reverse order

The resulting array has the same index set as x .

```
function array [$$E] of $T: row(array [int,$$E] of $T: x, int: r)
```

Return row r of array x

```
function array [$$E] of opt $T: row(array [int,$$E] of opt $T: x,
                                         int: r)
```

Return row r of array x

```
function array [$$E] of var $T: row(array [int,$$E] of var $T: x,
                                         int: r)
```

Return row r of array x

```
function array [$$E] of var opt $T: row(array [int,$$E] of var opt $T: x,
                                         int: r)
```

Return row r of array x

```
function array [int] of $T: slice_1d(array [$$E] of $T: x,
                                         array [int] of set of int: s,
                                         set of int: dims1)
```

Return slice of array x specified by sets s , coerced to new 1d array with index set dims1

```
function array [int] of opt $T: slice_1d(array [$$E] of opt $T: x,
                                         array [int] of set of int: s,
                                         set of int: dims1)
```

Return slice of array x specified by sets s , coerced to new 1d array with index set dims1

```
function array [int] of var $T: slice_1d(array [$$E] of var $T: x,
                                         array [int] of set of int: s,
                                         set of int: dims1)
```

Return slice of array x specified by sets s , coerced to new 1d array with index set dims1

```
function array [int] of var opt $T: slice_1d(array [$$E] of var opt $T: x,
                                         array [int] of set of int: s,
                                         set of int: dims1)
```

Return slice of array x specified by sets s , coerced to new 1d array with index set dims1

```
function array [int,int] of $T: slice_2d(array [$$E] of $T: x,
                                         array [int] of set of int: s,
                                         set of int: dims1,
                                         set of int: dims2)
```

Return slice of array x specified by sets s , coerced to new 2d array with index sets dims1 and dims2

```
function array [int,int] of opt $T: slice_2d(array [$E] of opt $T: x,
                                              array [int] of set of int: s,
                                              set of int: dims1,
                                              set of int: dims2)
```

Return slice of array x specified by sets s , coerced to new 2d array with index sets dims1 and dims2

```
function array [int,int] of var $T: slice_2d(array [$E] of var $T: x,
                                              array [int] of set of int: s,
                                              set of int: dims1,
                                              set of int: dims2)
```

Return slice of array x specified by sets s , coerced to new 2d array with index sets dims1 and dims2

```
function array [int,int] of var opt $T: slice_2d(array [$E] of var opt $T: x,
                                                 array [int] of set of int:_  
→s,  
                                              set of int: dims1,  
                                              set of int: dims2)
```

Return slice of array x specified by sets s , coerced to new 2d array with index sets dims1 and dims2

```
function array [int,int,int] of $T: slice_3d(array [$E] of $T: x,
                                              array [int] of set of int: s,
                                              set of int: dims1,
                                              set of int: dims2,
                                              set of int: dims3)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets dims1 , dims2 and dims3

```
function array [int,int,int] of opt $T: slice_3d(array [$E] of opt $T: x,
                                                 array [int] of set of int:_  
→s,  
                                              set of int: dims1,  
                                              set of int: dims2,  
                                              set of int: dims3)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets dims1 , dims2 and dims3

```
function array [int,int,int] of var $T: slice_3d(array [$E] of var $T: x,
                                                array [int] of set of int:_
                                                ↵s,
                                                set of int: dims1,
                                                set of int: dims2,
                                                set of int: dims3)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets dims_1 , dims_2 and dims_3

```
function array [int,int,int] of var opt $T: slice_3d(array [$E] of var opt
                                                     ↵$T: x,
                                                     array [int] of set of_
                                                     ↵int: s,
                                                     set of int: dims1,
                                                     set of int: dims2,
                                                     set of int: dims3)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets dims_1 , dims_2 and dims_3

```
function array [int,int,int] of $T: slice_4d(array [$E] of $T: x,
                                              array [int] of set of int: s,
                                              set of int: dims1,
                                              set of int: dims2,
                                              set of int: dims3,
                                              set of int: dims4)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets dims_1 , dims_2 , dims_3 , dims_4

```
function array [int,int,int] of opt $T: slice_4d(array [$E] of opt $T: x,
                                                 array [int] of set of int:_
                                                 ↵s,
                                                 set of int: dims1,
                                                 set of int: dims2,
                                                 set of int: dims3,
                                                 set of int: dims4)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets dims_1 , dims_2 , dims_3 , dims_4

```
function array [int,int,int] of var $T: slice_4d(array [$E] of var $T: x,
                                                array [int] of set of int:_
                                                ↵s,
                                                set of int: dims1,
                                                set of int: dims2,
                                                set of int: dims3,
```

```
set of int: dims4)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets dims1 , dims2 , dims3 , dims4

```
function array [int,int,int] of var opt $T: slice_4d(array [$E] of var opt
→$T: x,
→array [int] of set of_
→int: s,
→set of int: dims1,
→set of int: dims2,
→set of int: dims3,
→set of int: dims4)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets dims1 , dims2 , dims3 , dims4

```
function array [int,int,int] of $T: slice_5d(array [$E] of $T: x,
→array [int] of set of int: s,
→set of int: dims1,
→set of int: dims2,
→set of int: dims3,
→set of int: dims4,
→set of int: dims5)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets dims1 , dims2 , dims3 , dims4 , dims5

```
function array [int,int,int] of opt $T: slice_5d(array [$E] of opt $T: x,
→array [int] of set of int:_→s,
→set of int: dims1,
→set of int: dims2,
→set of int: dims3,
→set of int: dims4,
→set of int: dims5)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets dims1 , dims2 , dims3 , dims4 , dims5

```
function array [int,int,int] of var $T: slice_5d(array [$E] of var $T: x,
→array [int] of set of int:_→s,
→set of int: dims1,
→set of int: dims2,
→set of int: dims3,
→set of int: dims4,
```

```
set of int: dims5)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets $\text{dims1}, \text{dims2}, \text{dims3}, \text{dims4}, \text{dims5}$

```
function array [int,int,int] of var opt $T: slice_5d(array [$E] of var opt
    ↵$T: x,
                                array [int] of set of_
    ↵int: s,
                                set of int: dims1,
                                set of int: dims2,
                                set of int: dims3,
                                set of int: dims4,
                                set of int: dims5)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets $\text{dims1}, \text{dims2}, \text{dims3}, \text{dims4}, \text{dims5}$

```
function array [int,int,int] of $T: slice_6d(array [$E] of $T: x,
                                                array [int] of set of int: s,
                                                set of int: dims1,
                                                set of int: dims2,
                                                set of int: dims3,
                                                set of int: dims4,
                                                set of int: dims5,
                                                set of int: dims6)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets $\text{dims1}, \text{dims2}, \text{dims3}, \text{dims4}, \text{dims5}, \text{dims6}$

```
function array [int,int,int] of opt $T: slice_6d(array [$E] of opt $T: x,
                                                array [int] of set of int:_
    ↵s,
                                set of int: dims1,
                                set of int: dims2,
                                set of int: dims3,
                                set of int: dims4,
                                set of int: dims5,
                                set of int: dims6)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets $\text{dims1}, \text{dims2}, \text{dims3}, \text{dims4}, \text{dims5}, \text{dims6}$

```
function array [int,int,int] of var $T: slice_6d(array [$E] of var $T: x,
                                                array [int] of set of int:_
    ↵s,
                                set of int: dims1,
```

```
set of int: dims2,
set of int: dims3,
set of int: dims4,
set of int: dims5,
set of int: dims6)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets $\text{dims1}, \text{dims2}$, $\text{dims3}, \text{dims4}, \text{dims5}, \text{dims6}$

```
function array [int,int,int] of var opt $T: slice_6d(array [$$E] of var opt
→$T: x,
array [int] of set of_
→int: s,
set of int: dims1,
set of int: dims2,
set of int: dims3,
set of int: dims4,
set of int: dims5,
set of int: dims6)
```

Return slice of array x specified by sets s , coerced to new 3d array with index sets $\text{dims1}, \text{dims2}$, $\text{dims3}, \text{dims4}, \text{dims5}, \text{dims6}$

4.2.5.8 Array sorting operations

```
function array [int] of $$E: arg_sort(array [$$E] of int: x)
```

Returns the permutation p which causes x to be in sorted order hence $x[p[i]] \leq x[p[i+1]]$.

The permutation is the stable sort hence $x[p[i]] = x[p[i+1]] \rightarrow p[i] < p[i+1]$.

```
function array [int] of $$E: arg_sort(array [$$E] of float: x)
```

Returns the permutation p which causes x to be in sorted order hence $x[p[i]] \leq x[p[i+1]]$.

The permutation is the stable sort hence $x[p[i]] = x[p[i+1]] \rightarrow p[i] < p[i+1]$.

```
function array [$$E] of int: sort(array [$$E] of int: x)
```

Return values from array x sorted in non-decreasing order

```
function array [$$E] of float: sort(array [$$E] of float: x)
```

Return values from array x sorted in non-decreasing order

```
function array [$$E] of bool: sort(array [$$E] of bool: x)
```

Return values from array x sorted in non-decreasing order

```
function array [$$E] of var opt $T: sort_by(array [$$E] of var opt $T: x,
                                             array [$$E] of int: y)
```

Return array x sorted by the values in y in non-decreasing order

The sort is stable, i.e. if $y[i] = y[j]$ with $i < j$, then $x[i]$ will appear in the output before $x[j]$.

```
function array [$$E] of var $T: sort_by(array [$$E] of var $T: x,
                                         array [$$E] of int: y)
```

Return array x sorted by the values in y in non-decreasing order

The sort is stable, i.e. if $y[i] = y[j]$ with $i < j$, then $x[i]$ will appear in the output before $x[j]$.

```
function array [$$E] of $T: sort_by(array [$$E] of $T: x,
                                     array [$$E] of int: y)
```

Return array x sorted by the values in y in non-decreasing order

The sort is stable, i.e. if $y[i] = y[j]$ with $i < j$, then $x[i]$ will appear in the output before $x[j]$.

```
function array [$$E] of var opt $T: sort_by(array [$$E] of var opt $T: x,
                                              array [$$E] of float: y)
```

Return array x sorted by the values in y in non-decreasing order

The sort is stable, i.e. if $y[i] = y[j]$ with $i < j$, then $x[i]$ will appear in the output before $x[j]$.

```
function array [$$E] of var $T: sort_by(array [$$E] of var $T: x,
                                         array [$$E] of float: y)
```

Return array x sorted by the values in y in non-decreasing order

The sort is stable, i.e. if $y[i] = y[j]$ with $i < j$, then $x[i]$ will appear in the output before $x[j]$.

```
function array [$$E] of $T: sort_by(array [$$E] of $T: x,
                                     array [$$E] of float: y)
```

Return array x sorted by the values in y in non-decreasing order

The sort is stable, i.e. if $y[i] = y[j]$ with $i < j$, then $x[i]$ will appear in the output before $x[j]$.

4.2.5.9 Coercions

These functions implement coercions, or channeling, between different types.

```
function float: bool2float(bool: b)
```

Return Boolean b coerced to a float

```
function array [\$T] of float: bool2float(array [\$T] of bool: x)
```

Return array of Booleans x coerced to an array of floats

```
function array [\$T] of var float: bool2float(array [\$T] of var bool: x)
```

Return array of Booleans x coerced to an array of floats

```
function var float: bool2float(var bool: b)
```

Return Boolean b coerced to a float

```
function int: bool2int(bool: b)
```

Return Boolean b coerced to an integer

```
function var int: bool2int(var bool: b)
```

Return Boolean b coerced to an integer

```
function array [ $$E ] of var int: bool2int(array [ $$E ] of var bool: b)
```

Return array of Booleans b coerced to an array of integers

```
function array [ \$T ] of int: bool2int(array [ \$T ] of bool: x)
```

Return array of Booleans x coerced to an array of integers

```
function array [ \$T ] of set of int: bool2int(array [ \$T ] of set of bool: x)
```

Return array of sets of Booleans x coerced to an array of sets of integers

```
function array [\$T] of var int: bool2int(array [\$T] of var bool: x)
```

Return array of Booleans x coerced to an array of integers

```
function array [\$T] of var opt int: bool2int(array [\$T] of var opt bool: x)
```

Return array of Booleans x coerced to an array of integers

```
function int: ceil(float: x)
```

Return $\lceil x \rceil$

```
function int: floor(float: x)
```

Return $\lfloor x \rfloor$

```
function float: int2float(int: x)
```

Return integer x coerced to a float

```
function var float: int2float(var int: x)
```

Return integer x coerced to a float

```
function array [\$T] of float: int2float(array [\$T] of int: x)
```

Return array of integers x coerced to an array of floats

```
function array [\$T] of var float: int2float(array [\$T] of var int: x)
```

Return array of integers x coerced to an array of floats

```
function int: round(float: x)
```

Return x rounded to nearest integer

```
function array [int] of $$E: set2array(set of $$E: x)
```

Return a set of integers x coerced to an array of integers

4.2.5.10 String operations

These functions implement operations on strings.

```
function string: '++'(string: s1, string: s2)
```

Usage: `s1 ++ s2`

Return concatenation of `s1` and `s2`

```
function string: concat(array [$T] of string: s)
```

Return concatenation of strings in array `s`

```
function string: file_path()
```

Return path of file where this function is called

```
function string: format(var opt $T: x)
```

Convert `x` into a string

```
function string: format(var opt set of $T: x)
```

Convert `x` into a string

```
function string: format(array [$U] of var opt $T: x)
```

Convert `x` into a string

```
function string: format(int: w, int: p, var opt $T: x)
```

Formatted to-string conversion

Converts the value `x` into a string right justified by the number of characters given by `w`, or left justified if `w` is negative.

The maximum length of the string representation of `x` is given by `p`, or the maximum number of digits after the decimal point for floating point numbers. It is a run-time error for `p` to be negative.

```
function string: format(int: w, int: p, var opt set of $T: x)
```

Formatted to-string conversion

Converts the value `x` into a string right justified by the number of characters given by `w`, or left justified if `w` is negative.

The maximum length of the string representation of x is given by p . It is a run-time error for p to be negative.

```
function string: format(int: w, int: p, array [$U] of var opt $T: x)
```

Formatted to-string conversion

Converts the value x into a string right justified by the number of characters given by w , or left justified if w is negative.

The maximum length of the string representation of x is given by p . It is a run-time error for p to be negative.

```
function string: format(int: w, var opt $T: x)
```

Formatted to-string conversion

Converts the value x into a string right justified by the number of characters given by w , or left justified if w is negative.

```
function string: format(int: w, var opt set of $T: x)
```

Formatted to-string conversion

Converts the value x into a string right justified by the number of characters given by w , or left justified if w is negative.

```
function string: format(int: w, array [$U] of var opt $T: x)
```

Formatted to-string conversion

Converts the value x into a string right justified by the number of characters given by w , or left justified if w is negative.

```
function string: format_justify_string(int: w, string: x)
```

String justification

Returns the string x right justified by the number of characters given by w , or left justified if w is negative.

```
function string: join(string: d, array [$T] of string: s)
```

Join string in array s using delimiter d

```
function array [int] of string: outputJSON()
```

Return array for output of all variables in JSON format

```
function array [int] of string: outputJSONParameters()
```

Return array for output of all parameters in JSON format

```
function string: show(var opt set of $T: x)
```

Convert x into a string

```
function string: show(var opt $T: x)
```

Convert x into a string

```
function string: show(array [$U] of var opt $T: x)
```

Convert x into a string

```
function string: show2d(array [int,int] of var opt $T: x)
```

Convert two-dimensional array x into a string

```
function string: show3d(array [int,int,int] of var opt $T: x)
```

Convert three-dimensional array x into a string

```
function string: showJSON(var opt $T: x)
```

Convert x into JSON string

```
function string: showJSON(array [$U] of var opt $T: x)
```

Convert x into JSON string

```
function string: show_float(int: w, int: p, var float: x)
```

Formatted to-string conversion for floats.

Converts the float x into a string right justified by the number of characters given by w , or left justified if w is negative. The number of digits to appear after the decimal point is given by p . It is a run-time error for p to be negative.

```
function string: show_int(int: w, var int: x)
```

Formatted to-string conversion for integers

Converts the integer x into a string right justified by the number of characters given by w , or left justified if w is negative.

```
function int: string_length(string: s)
```

Return length of s

4.2.5.11 Reflection operations

These functions return information about declared or inferred variable bounds and domains.

```
function set of int: dom(var int: x)
```

Return domain of x

```
function set of int: dom_array(array [$T] of var int: x)
```

Return union of all domains of the elements in array x

```
function set of int: dom_bounds_array(array [$T] of var int: x)
```

Return approximation of union of all domains of the elements in array x

```
function int: dom_size(var int: x)
```

Return cardinality of the domain of x

```
function $T: fix(var opt $T: x)
```

Check if the value of x is fixed at this point in evaluation. If it is fixed, return its value, otherwise abort.

```
function array [$U] of $T: fix(array [$U] of var opt $T: x)
```

Check if the value of every element of the array x is fixed at this point in evaluation. If all are fixed, return an array of their values, otherwise abort.

```
test has_bounds(var int: x)
```

Test if variable x has declared, finite bounds

```
test has_bounds(var float: x)
```

Test if variable x has declared, finite bounds

```
test has_ub_set(var set of int: x)
```

Test if variable x has a declared, finite upper bound

```
test is_fixed(var opt $T: x)
```

Test if x is fixed

```
test is_fixed(array [$U] of var opt $T: x)
```

Test if every element of array x is fixed

```
function int: lb(var int: x)
```

Return lower bound of x

```
function int: lb(var opt int: x)
```

Return lower bound of x

```
function float: lb(var float: x)
```

Return lower bound of x

```
function set of int: lb(var set of int: x)
```

Return lower bound of x

```
function array [$U] of int: lb(array [$U] of var int: x)
```

Return array of lower bounds of the elements in array x

```
function array [$U] of float: lb(array [$U] of var float: x)
```

Return array of lower bounds of the elements in array x

```
function array [$U] of set of int: lb(array [$U] of var set of int: x)
```

Return array of lower bounds of the elements in array x

```
function int: lb_array(array [$U] of var opt int: x)
```

Return minimum of all lower bounds of the elements in array x

```
function float: lb_array(array [$U] of var float: x)
```

Return minimum of all lower bounds of the elements in array x

```
function set of int: lb_array(array [$U] of var set of int: x)
```

Return intersection of all lower bounds of the elements in array x

```
function int: ub(var int: x)
```

Return upper bound of x

```
function int: ub(var opt int: x)
```

Return upper bound of x

```
function float: ub(var float: x)
```

Return upper bound of x

```
function set of int: ub(var set of int: x)
```

Return upper bound of x

```
function array [$U] of int: ub(array [$U] of var int: x)
```

Return array of upper bounds of the elements in array x

```
function array [$U] of float: ub(array [$U] of var float: x)
```

Return array of upper bounds of the elements in array x

```
function array [$U] of set of int: ub(array [$U] of var set of int: x)
```

Return array of upper bounds of the elements in array x

```
function int: ub_array(array [$U] of var opt int: x)
```

Return maximum of all upper bounds of the elements in array x

```
function float: ub_array(array [$U] of var float: x)
```

Return maximum of all upper bounds of the elements in array x

```
function set of int: ub_array(array [$U] of var set of int: x)
```

Return union of all upper bounds of the elements in array x

4.2.5.12 Assertions and debugging functions

These functions help debug models and check that input data conforms to the expectations.

```
test abort(string: msg)
```

Abort evaluation and print message msg .

```
function $T: assert(bool: b, string: msg, $T: x)
```

If b is true, return x , otherwise abort with message msg .

```
function var $T: assert(bool: b, string: msg, var $T: x)
```

If b is true, return x , otherwise abort with message msg .

```
function var opt $T: assert(bool: b, string: msg, var opt $T: x)
```

If b is true, return x , otherwise abort with message msg .

```
function array [$U] of $T: assert(bool: b,
                                string: msg,
                                array [$U] of $T: x)
```

If b is true, return x , otherwise abort with message msg .

```
function array [$U] of var $T: assert(bool: b,
                                         string: msg,
                                         array [$U] of var $T: x)
```

If b is true, return x , otherwise abort with message msg .

```
function array [$U] of var opt $T: assert(bool: b,
                                         string: msg,
                                         array [$U] of var opt $T: x)
```

If b is true, return x , otherwise abort with message msg .

```
test assert(bool: b, string: msg)
```

If b is true, return true, otherwise abort with message msg .

```
function $T: trace(string: msg, $T: x)
```

Return x , and print message msg .

```
function var $T: trace(string: msg, var $T: x)
```

Return x , and print message msg .

```
function var opt $T: trace(string: msg, var opt $T: x)
```

Return x , and print message msg .

```
test trace(string: msg)
```

Return true, and print message msg .

```
function $T: trace_stdout(string: msg, $T: x)
```

Return x , and print message msg .

```
function var $T: trace_stdout(string: msg, var $T: x)
```

Return x , and print message msg .

```
function var opt $T: trace_stdout(string: msg, var opt $T: x)
```

Return x , and print message msg .

```
test trace_stdout(string: msg)
```

Return true, and print message msg .

4.2.5.13 Functions for enums

```
function $$E: enum_next(set of $$E: e, $$E: x)
```

Return next greater enum value of x in enum e

```
function var $$E: enum_next(set of $$E: e, var $$E: x)
```

Return next greater enum value of x in enum e

```
function $$E: enum_prev(set of $$E: e, $$E: x)
```

Return next smaller enum value of x in enum e

```
function var $$E: enum_prev(set of $$E: e, var $$E: x)
```

Return next smaller enum value of x in enum e

```
function $$E: to_enum(set of $$E: X, int: x)
```

Convert x to enum type X

```
function var $$E: to_enum(set of $$E: X, var int: x)
```

Convert x to enum type X

```
function array [$$U] of $$E: to_enum(set of $$E: X,
                                         array [$$U] of int: x)
```

Convert x to enum type X

```
function array [$$U] of var $$E: to_enum(set of $$E: X,
                                         array [$$U] of var int: x)
```

Convert x to enum type X

```
function set of $$E: to_enum(set of $$E: X, set of int: x)
```

Convert x to enum type X

4.2.5.14 Random Number Generator builtins

These functions implement random number generators from different probability distributions.

```
test bernoulli(float: p)
```

Return a boolean sample from the Bernoulli distribution defined by probability **p**

```
function int: binomial(int: t, float: p)
```

Return a sample from the binomial distribution defined by sample number **t** and probability **p**

```
function float: cauchy(float: mean, float: scale)
```

Return a sample from the cauchy distribution defined by **mean, scale**

```
function float: cauchy(int: mean, float: scale)
```

Return a sample from the cauchy distribution defined by **mean, scale**

```
function float: chisquared(int: n)
```

Return a sample from the chi-squared distribution defined by the degree of freedom **n**

```
function float: chisquared(float: n)
```

Return a sample from the chi-squared distribution defined by the degree of freedom **n**

```
function int: discrete_distribution(array [int] of int: weights)
```

Return a sample from the discrete distribution defined by the array of weights **weights** that assigns a weight to each integer starting from zero

```
function float: exponential(int: lambda)
```

Return a sample from the exponential distribution defined by **lambda**

```
function float: exponential(float: lambda)
```

Return a sample from the exponential distribution defined by **lambda**

```
function float: fdistribution(float: d1, float: d2)
```

Return a sample from the Fisher-Snedecor F-distribution defined by the degrees of freedom **d1, d2**

```
function float: fdistribution(int: d1, int: d2)
```

Return a sample from the Fisher-Snedecor F-distribution defined by the degrees of freedom **d1, d2**

```
function float: gamma(float: alpha, float: beta)
```

Return a sample from the gamma distribution defined by **alpha, beta**

```
function float: gamma(int: alpha, float: beta)
```

Return a sample from the gamma distribution defined by **alpha, beta**

```
function float: lognormal(float: mean, float: std)
```

Return a sample from the lognormal distribution defined by **mean, std**

```
function float: lognormal(int: mean, float: std)
```

Return a sample from the lognormal distribution defined by **mean, std**

```
function float: normal(float: mean, float: std)
```

Return a sample from the normal distribution defined by **mean, std**

```
function float: normal(int: mean, float: std)
```

Return a sample from the normal distribution defined by **mean, std**

```
function int: poisson(float: mean)
```

Return a sample from the poisson distribution defined by **mean**

```
function int: poisson(int: mean)
```

Return a sample from the poisson distribution defined by an integer **mean**

```
function float: tdistribution(float: n)
```

Return a sample from the student's t-distribution defined by the sample size **n**

```
function float: tdistribution(int: n)
```

Return a sample from the student's t-distribution defined by the sample size **n**

```
function float: uniform(float: lowerbound, float: upperbound)
```

Return a sample from the uniform distribution defined by **lowerbound, upperbound**

```
function int: uniform(int: lowerbound, int: upperbound)
```

Return a sample from the uniform distribution defined by **lowerbound, upperbound**

```
function float: weibull(float: shape, float: scale)
```

Return a sample from the Weibull distribution defined by **shape, scale**

```
function float: weibull(int: shape, float: scale)
```

Return a sample from the Weibull distribution defined by **shape, scale**

4.2.5.15 Special constraints

These predicates allow users to mark constraints as e.g. symmetry breaking or redundant, so that solvers can choose to implement them differently.

We cannot easily use annotations for this purpose, since annotations are propagated to all constraints in a decomposition, which may be incorrect for redundant or symmetry breaking constraints in the presence of common subexpression elimination (CSE).

```
predicate implied_constraint(var bool: b)
```

Mark b as an implied constraint (synonym for redundant_constraint)

```
predicate redundant_constraint(var bool: b)
```

Mark b as a redundant constraint

```
predicate symmetry_breaking_constraint(var bool: b)
```

Mark b as a symmetry breaking constraint

4.2.5.16 Language information

These functions return information about the MiniZinc system.

```
function int: mzn_compiler_version()
```

Return MiniZinc version encoded as an integer (major*10000+minor*1000+patch).

```
function string: mzn_version_to_string(int: v)
```

Return string representation of v given an integer major*10000+minor*1000+patch

4.2.6 FlatZinc builtins

These are the standard constraints that need to be supported by FlatZinc solvers (or redefined in the `redefinitions.mzn` file).

4.2.6.1 Integer FlatZinc builtins

```
predicate array_int_element(var int: b,
                           array [int] of int: as,
                           var int: c)
```

Constrains as [b] = c

```
predicate array_int_maximum(var int: m, array [int] of var int: x)
```

Constrains m to be the maximum value of the (non-empty) array x

```
predicate array_int_minimum(var int: m, array [int] of var int: x)
```

Constrains m to be the minimum value of the (non-empty) array x

```
predicate array_var_int_element(var int: b,
                                 array [int] of var int: as,
                                 var int: c)
```

Constrains as [b] = c

```
predicate int_abs(var int: a, var int: b)
```

Constrains b to be the absolute value of a

```
predicate int_div(var int: a, var int: b, var int: c)
```

Constrains a / b = c

```
predicate int_eq(var int: a, var int: b)
```

Constrains a to be equal to b

```
predicate int_eq_reif(var int: a, var int: b, var bool: r)
```

Constrains ($a = b \leftrightarrow r$)

```
predicate int_le(var int: a, var int: b)
```

Constrains a to be less than or equal to b

```
predicate int_le_reif(var int: a, var int: b, var bool: r)
```

Constrains ($a \leq b \leftrightarrow r$)

```
predicate int_lin_eq(array [int] of int: as,
                     array [int] of var int: bs,
                     int: c)
```

Constrains $c = \sum_i \mathbf{as}[i] * \mathbf{bs}[i]$

```
predicate int_lin_eq_reif(array [int] of int: as,
                          array [int] of var int: bs,
                          int: c,
                          var bool: r)
```

Constrains $r \leftrightarrow (c = \sum_i \mathbf{as}[i] * \mathbf{bs}[i])$

```
predicate int_lin_le(array [int] of int: as,
                     array [int] of var int: bs,
                     int: c)
```

Constrains $\sum \mathbf{as}[i] * \mathbf{bs}[i] \leq c$

```
predicate int_lin_le_reif(array [int] of int: as,
                          array [int] of var int: bs,
                          int: c,
                          var bool: r)
```

Constrains $r \leftrightarrow (\sum \mathbf{as}[i] * \mathbf{bs}[i] \leq c)$

```
predicate int_lin_ne(array [int] of int: as,
                     array [int] of var int: bs,
```

```
int: c)
```

Constrains $c \neq \sum_i \mathbf{as}[i] * \mathbf{bs}[i]$

```
predicate int_lin_ne_reif(array [int] of int: as,
                           array [int] of var int: bs,
                           int: c,
                           var bool: r)
```

Constrains $r \leftrightarrow (c \neq \sum_i \mathbf{as}[i] * \mathbf{bs}[i])$

```
predicate int_lt(var int: a, var int: b)
```

Constrains $a < b$

```
predicate int_lt_reif(var int: a, var int: b, var bool: r)
```

Constrains $r \leftrightarrow (a < b)$

```
predicate int_max(var int: a, var int: b, var int: c)
```

Constrains $\max(a, b) = c$

```
predicate int_min(var int: a, var int: b, var int: c)
```

Constrains $\min(a, b) = c$

```
predicate int_mod(var int: a, var int: b, var int: c)
```

Constrains $a \% b = c$

```
predicate int_ne(var int: a, var int: b)
```

Constrains $a \neq b$

```
predicate int_ne_reif(var int: a, var int: b, var bool: r)
```

$r \leftrightarrow (a \neq b)$

```
predicate int_plus(var int: a, var int: b, var int: c)
```

Constrains $a + b = c$

```
predicate int_pow(var int: x, var int: y, var int: z)
```

Constrains $z = x^y$

```
predicate int_pow_fixed(var int: x, int: y, var int: z)
```

Constrains $z = x^y$

```
predicate int_times(var int: a, var int: b, var int: c)
```

Constrains $a * b = c$

4.2.6.2 Bool FlatZinc builtins

```
predicate array_bool_and(array [int] of var bool: as, var bool: r)
```

Constrains $r \leftrightarrow \bigwedge_i \text{as}[i]$

```
predicate array_bool_element(var int: b,
                            array [int] of bool: as,
                            var bool: c)
```

Constrains $\text{as}[\text{b}] = c$

```
predicate array_bool_or(array [int] of var bool: as, var bool: r)
```

Constrains $r \leftrightarrow \bigvee_i \text{as}[i]$

```
predicate array_bool_xor(array [int] of var bool: as)
```

Constrains $r \leftrightarrow \bigoplus_i \text{as}[i]$

```
predicate array_var_bool_element(var int: b,
                                array [int] of var bool: as,
                                var bool: c)
```

Constrains $\text{as}[\text{b}] = c$

```
predicate bool2int(var bool: a, var int: b)
```

Constrains $\mathbf{b} \in \{0, 1\}$ and $\mathbf{a} \leftrightarrow \mathbf{b} = 1$

```
predicate bool_and(var bool: a, var bool: b, var bool: r)
```

Constrains $r \leftrightarrow a \wedge b$

```
predicate bool_clause(array [int] of var bool: as,
                      array [int] of var bool: bs)
```

Constrains $\bigvee_i \mathbf{as}[i] \vee \bigvee_j \neg \mathbf{bs}[j]$

```
predicate bool_eq(var bool: a, var bool: b)
```

Constrains $a = b$

```
predicate bool_eq_reif(var bool: a, var bool: b, var bool: r)
```

Constrains $r \leftrightarrow (a = b)$

```
predicate bool_le(var bool: a, var bool: b)
```

Constrains $a \leq b$

```
predicate bool_le_reif(var bool: a, var bool: b, var bool: r)
```

Constrains $r \leftrightarrow (a \leq b)$

```
predicate bool_lin_eq(array [int] of int: as,
                      array [int] of var bool: bs,
                      var int: c)
```

Constrains $c = \sum_i \mathbf{as}[i] * \mathbf{bs}[i]$

```
predicate bool_lin_le(array [int] of int: as,
                      array [int] of var bool: bs,
                      int: c)
```

Constrains $c \leq \sum_i \mathbf{as}[i] * \mathbf{bs}[i]$

```
predicate bool_lt(var bool: a, var bool: b)
```

Constrains $a < b$

```
predicate bool_lt_reif(var bool: a, var bool: b, var bool: r)
```

Constrains $r \leftrightarrow (a < b)$

```
predicate bool_not(var bool: a, var bool: b)
```

Constrains $a \neq b$

```
predicate bool_or(var bool: a, var bool: b, var bool: r)
```

Constrains $r \leftrightarrow a \vee b$

```
predicate bool_xor(var bool: a, var bool: b, var bool: r)
```

Constrains $r \leftrightarrow a \oplus b$

```
predicate bool_xor(var bool: a, var bool: b)
```

Constrains $a \oplus b$

4.2.6.3 Set FlatZinc builtins

```
predicate array_set_element(var int: b,
                           array [int] of set of int: as,
                           var set of int: c)
```

Constrains $as[b] = c$

```
predicate array_var_set_element(var int: b,
                               array [int] of var set of int: as,
                               var set of int: c)
```

Constrains $as[b] = c$

```
predicate set_card(var set of int: S, var int: x)
```

Constrains $x = |S|$

```
predicate set_diff(var set of int: x,
                  var set of int: y,
                  var set of int: r)
```

Constrains $r = x \setminus y$

```
predicate set_eq(var set of int: x, var set of int: y)
```

Constrains $x = y$

```
predicate set_eq_reif(var set of int: x,  
                      var set of int: y,  
                      var bool: r)
```

Constrains $r \leftrightarrow (x = y)$

```
predicate set_in(var int: x, set of int: S)
```

Constrains $x \in S$

```
predicate set_in(var int: x, var set of int: S)
```

Constrains $x \in S$

```
predicate set_in_reif(var int: x, set of int: S, var bool: r)
```

Constrains $r \leftrightarrow (x \in S)$

```
predicate set_in_reif(var int: x, var set of int: S, var bool: r)
```

Constrains $r \leftrightarrow (x \in S)$

```
predicate set_intersect(var set of int: x,  
                      var set of int: y,  
                      var set of int: r)
```

Constrains $r = x \cap y$

```
predicate set_le(var set of int: x, var set of int: y)
```

Constrains $x \leq y$ (lexicographic order)

```
predicate set_lt(var set of int: x, var set of int: y)
```

Constrains $x < y$ (lexicographic order)

```
predicate set_ne(var set of int: x, var set of int: y)
```

Constrains $x \neq y$

```
predicate set_ne_reif(var set of int: x,  
                      var set of int: y,  
                      var bool: r)
```

Constrains $r \leftrightarrow (x \neq y)$

```
predicate set_subset(var set of int: x, var set of int: y)
```

Constrains $x \subseteq y$

```
predicate set_subset_reif(var set of int: x,
                           var set of int: y,
                           var bool: r)
```

Constrains $r \leftrightarrow (x \subseteq y)$

```
predicate set_superset(var set of int: x, var set of int: y)
```

Constrains $x \supseteq y$

```
predicate set_symdiff(var set of int: x,
                      var set of int: y,
                      var set of int: r)
```

Constrains r to be the symmetric difference of x and y

```
predicate set_union(var set of int: x,
                   var set of int: y,
                   var set of int: r)
```

Constrains $r = x \cup y$

4.2.6.4 Float FlatZinc builtins

```
predicate array_float_element(var int: b,
                             array [int] of float: as,
                             var float: c)
```

Constrains $as[b] = c$

```
predicate array_float_maximum(var int: m, array [int] of var int: x)
```

Constrains m to be the maximum value of the (non-empty) array x

```
predicate array_float_minimum(var int: m, array [int] of var int: x)
```

Constrains m to be the minimum value of the (non-empty) array x

```
predicate array_var_float_element(var int: b,
                                  array [int] of var float: as,
                                  var float: c)
```

Constrains $b = \text{as}[b]$

```
predicate float_abs(var float: a, var float: b)
```

Constrains $b = |\text{a}|$

```
predicate float_acos(var float: a, var float: b)
```

Constrains $b = \text{acos}(a)$

```
predicate float_acosh(var float: a, var float: b)
```

Constrains $b = \text{acosh}(a)$

```
predicate float_asin(var float: a, var float: b)
```

Constrains $b = \text{asin}(a)$

```
predicate float_asinh(var float: a, var float: b)
```

Constrains $b = \text{asinh}(a)$

```
predicate float_atan(var float: a, var float: b)
```

Constrains $b = \text{atan}(a)$

```
predicate float_atanh(var float: a, var float: b)
```

Constrains $b = \text{atanh}(a)$

```
predicate float_cos(var float: a, var float: b)
```

Constrains $b = \text{cos}(a)$

```
predicate float_cosh(var float: a, var float: b)
```

Constrains $b = \text{cosh}(a)$

```
predicate float_div(var float: a, var float: b, var float: c)
```

Constrains $a / b = c$

```
predicate float_dom(var float: x, array [int] of float: as)
```

Constrains the domain of x using the values in as , using each pair of values as $[2^* i - 1]..[2^* i]$ for i in $1..n/2$ as a possible range

```
predicate float_eq(var float: a, var float: b)
```

Constrains $a = b$

```
predicate float_eq_reif(var float: a, var float: b, var bool: r)
```

Constrains $r \leftrightarrow (a = b)$

```
predicate float_exp(var float: a, var float: b)
```

Constrains $b = \exp(a)$

```
predicate float_in(var float: a, float: b, float: c)
```

Constrains $a \in [b, c]$

```
predicate float_in_reif(var float: a, float: b, float: c, var bool: r)
```

Constrains $r \leftrightarrow a \in [b, c]$

```
predicate float_le(var float: a, var float: b)
```

Constrains $a \leq b$

```
predicate float_le_reif(var float: a, var float: b, var bool: r)
```

Constrains $r \leftrightarrow (a \leq b)$

```
predicate float_lin_eq(array [int] of float: as,
                      array [int] of var float: bs,
                      float: c)
```

Constrains $c = \sum_i as[i] * bs[i]$

```
predicate float_lin_eq_reif(array [int] of float: as,
                             array [int] of var float: bs,
                             float: c,
                             var bool: r)
```

Constrains $r \leftrightarrow (\mathbf{c} = \sum_i \mathbf{as}[i] * \mathbf{bs}[i])$

```
predicate float_lin_le(array [int] of float: as,
                      array [int] of var float: bs,
                      float: c)
```

Constrains $\mathbf{c} \leq \sum_i \mathbf{as}[i] * \mathbf{bs}[i]$

```
predicate float_lin_le_reif(array [int] of float: as,
                            array [int] of var float: bs,
                            float: c,
                            var bool: r)
```

Constrains $r \leftrightarrow (\mathbf{c} \leq \sum_i \mathbf{as}[i] * \mathbf{bs}[i])$

```
predicate float_lin_lt(array [int] of float: as,
                      array [int] of var float: bs,
                      float: c)
```

Constrains $\mathbf{c} < \sum_i \mathbf{as}[i] * \mathbf{bs}[i]$

```
predicate float_lin_lt_reif(array [int] of float: as,
                            array [int] of var float: bs,
                            float: c,
                            var bool: r)
```

Constrains $r \leftrightarrow (\mathbf{c} < \sum_i \mathbf{as}[i] * \mathbf{bs}[i])$

```
predicate float_lin_ne(array [int] of float: as,
                      array [int] of var float: bs,
                      float: c)
```

Constrains $\mathbf{c} \neq \sum_i \mathbf{as}[i] * \mathbf{bs}[i]$

```
predicate float_lin_ne_reif(array [int] of float: as,
                            array [int] of var float: bs,
                            float: c,
                            var bool: r)
```

Constrains $r \leftrightarrow (\mathbf{c} \neq \sum_i \mathbf{as}[i] * \mathbf{bs}[i])$

```
predicate float_ln(var float: a, var float: b)
```

Constrains $b = \ln(a)$

```
predicate float_log10(var float: a, var float: b)
```

Constrains $b = \log_{10}(a)$

```
predicate float_log2(var float: a, var float: b)
```

Constrains $b = \log_2(a)$

```
predicate float_lt(var float: a, var float: b)
```

Constrains $a < b$

```
predicate float_lt_reif(var float: a, var float: b, var bool: r)
```

Constrains $r \leftrightarrow (a < b)$

```
predicate float_max(var float: a, var float: b, var float: c)
```

Constrains $\max(a, b) = c$

```
predicate float_min(var float: a, var float: b, var float: c)
```

Constrains $\min(a, b) = c$

```
predicate float_ne(var float: a, var float: b)
```

Constrains $a \neq b$

```
predicate float_ne_reif(var float: a, var float: b, var bool: r)
```

Constrains $r \leftrightarrow (a \neq b)$

```
predicate float_plus(var float: a, var float: b, var float: c)
```

Constrains $a + b = c$

```
predicate float_pow(var float: x, var float: y, var float: z)
```

Constrains $z = x^y$

```
predicate float_sin(var float: a, var float: b)
```

Constrains $b = \sin(a)$

```
predicate float_sinh(var float: a, var float: b)
```

Constrains $b = \sinh(a)$

```
predicate float_sqrt(var float: a, var float: b)
```

Constrains $b = \sqrt{a}$

```
predicate float_tan(var float: a, var float: b)
```

Constrains $b = \tan(a)$

```
predicate float_tanh(var float: a, var float: b)
```

Constrains $b = \tanh(a)$

```
predicate float_times(var float: a, var float: b, var float: c)
```

Constrains $a * b = c$

```
predicate int2float(var int: x, var float: y)
```

Constrains $y = x$

4.2.6.5 FlatZinc builtins added in MiniZinc 2.0.0.

These functions and predicates define built-in operations of the MiniZinc language that have been added in MiniZinc 2.0.0. Solvers that support these natively need to include a file called redefinitions-2.0.mzn in their solver library that redefines these predicates as builtins.

```
predicate array_float_maximum(var float: m,
                             array [int] of var float: x)
```

Constrains m to be the maximum value in array x .

```
predicate array_float_minimum(var float: m,
                             array [int] of var float: x)
```

Constrains m to be the minimum value in array x .

```
predicate array_int_maximum(var int: m, array [int] of var int: x)
```

Constrains m to be the maximum value in array x .

```
predicate array_int_minimum(var int: m, array [int] of var int: x)
```

Constrains m to be the minimum value in array x .

```
predicate bool_clause_reif(array [int] of var bool: as,
                            array [int] of var bool: bs,
                            var bool: b)
```

Reified clause constraint. Constrains $\mathbf{b} \leftrightarrow \bigvee_i \mathbf{as}[i] \vee \bigvee_j \neg \mathbf{bs}[j]$

4.2.6.6 FlatZinc builtins added in MiniZinc 2.0.2.

These functions and predicates define built-in operations of the MiniZinc language that have been added in MiniZinc 2.0.2. Solvers that support these natively need to include a file called redefinitions-2.0.2.mzn in their solver library that redefines these predicates as builtins.

```
predicate array_var_bool_element_nonshifted(var int: idx,
                                             array [int] of var bool: x,
                                             var bool: c)
```

Element constraint on array with MiniZinc index set, constrains $x[\text{idx}] = c$ This can be overridden in a solver that can perform the index calculation more efficiently than using a MiniZinc decomposition.

```
predicate array_var_float_element_nonshifted(var int: idx,
                                              array [int] of var float: x,
                                              var float: c)
```

Element constraint on array with MiniZinc index set, constrains $x[\text{idx}] = c$ This can be overridden in a solver that can perform the index calculation more efficiently than using a MiniZinc decomposition.

```
predicate array_var_int_element_nonshifted(var int: idx,
                                            array [int] of var int: x,
                                            var int: c)
```

Element constraint on array with MiniZinc index set, constrains $x[\text{idx}] = c$ This can be overridden in a solver that can perform the index calculation more efficiently than using a MiniZinc decomposition.

```
predicate array_var_set_element_nonshifted(var int: idx,
                                         array [int] of var set of int: x,
                                         var set of int: c)
```

Element constraint on array with MiniZinc index set, constrains $x[\text{idx}] = c$. This can be overridden in a solver that can perform the index calculation more efficiently than using a MiniZinc decomposition.

4.2.6.7 FlatZinc builtins added in MiniZinc 2.1.0.

These functions and predicates define built-in operations of the MiniZinc language that have been added in MiniZinc 2.1.0. Solvers that support these natively need to include a file called redefinitions-2.1.0.mzn in their solver library that redefines these predicates as builtins.

```
predicate float_dom(var float: x, array [int] of float: as)
```

Constrains x to take one of the values in as

```
predicate float_in(var float: x, float: a, float: b)
```

Constrains $a \leq x \leq b$

4.2.6.8 FlatZinc builtins added in MiniZinc 2.1.1.

These functions and predicates define built-in operations of the MiniZinc language that have been added in MiniZinc 2.1.1. Solvers that support these natively need to include a file called redefinitions-2.1.1.mzn in their solver library that redefines these predicates as builtins.

```
function var $$E: max(var set of $$E: s)
```

Returns variable constrained to be equal to the maximum of the set s . An alternative implementation can be found in the comments of the source code.

```
function var $$E: min(var set of $$E: s)
```

Returns variable constrained to be equal to the minimum of the set s . An alternative implementation can be found in the comments of the source code.

CHAPTER 4.3

Interfacing Solvers to Flatzinc

This document describes the interface between the MiniZinc system and FlatZinc solvers.

4.3.1 Specification of FlatZinc

This document is the specification of the FlatZinc modelling language. It also includes a definition of the standard command line options a FlatZinc solver should support in order to work with the `minizinc` driver program (and the MiniZinc IDE).

FlatZinc is the target constraint modelling language into which MiniZinc models are translated. It is a very simple solver independent problem specification language, requiring minimal implementation effort to support.

Throughout this document: r_1, r_2 denote float literals; $x_1, x_2, \dots, x_k, i, j, k$ denote int literals; $y_1, y_2, \dots, y_k, y_i$ denote literal array elements.

4.3.1.1 Comments

Comments start with a percent sign % and extend to the end of the line. Comments can appear anywhere in a model.

4.3.1.2 Types

There are three varieties of types in FlatZinc.

- *Parameter* types apply to fixed values that are specified directly in the model.
- *Variable* types apply to values computed by the solver during search. Every parameter type has a corresponding variable type; the variable type being distinguished by a `var` keyword.

- *Annotations* and *strings*: annotations can appear on variable declarations, constraints, and on the solve goal. They provide information about how a variable or constraint should be treated by the solver (e.g., whether a variable should be output as part of the result or whether a particular constraint should be implemented using domain consistency). Strings may appear as arguments to annotations, but nowhere else.

Parameter types

Parameters are fixed quantities explicitly specified in the model (see rule [<par-type>](#) in Section 4.3.6).

Type	Values
bool	true or false
float	float
int	int
set of int	subset of int
array [1.. n] of bool	array of bools
array [1.. n] of float	array of floats
array [1.. n] of int	array of ints
array [1.. n] of set of int	array of sets of ints

A parameter may be used where a variable is expected, but not vice versa.

In predicate declarations the following additional parameter types are allowed.

Type	Values
$r_a \dots r_b$	bounded float
$x_a \dots x_b$	int in range
{ x_a, x_b, \dots, x_k }	int in set
set of $x_a \dots x_b$	subset of int range
set of { x_a, x_b, \dots, x_k }	subset of int set
array [1.. n] of $r_a \dots r_b$	array of floats in range
array [1.. n] of $x_a \dots x_b$	array of ints in range
array [1.. n] of set of $x_a \dots x_b$	array of sets of ints in range
array [1.. n] of set of { x_a, x_b, \dots, x_k }	array of subsets of set

A range $x_a \dots x_b$ denotes a closed interval $\{x | x_a \leq x \leq x_b\}$ (same for float ranges).

An array type appearing in a predicate declaration may use just int instead of 1.. n for the array index range in cases where the array argument can be of any length.

Variable types

Variables are quantities decided by the solver (see rules [<basic-var-type>](#) and [<array-var-type>](#) in Section 4.3.6).

Variable type
var bool
var float
var $r_a \dots r_b$
var int
var $x_a \dots x_b$
var $\{ x_a, x_b, \dots, x_k \}$
var set of $x_a \dots x_b$
var set of $\{ x_a, x_b, \dots, x_k \}$
array [1.. n] of var bool
array [1.. n] of var float
array [1.. n] of var $r_a \dots r_b$
array [1.. n] of var int
array [1.. n] of var $x_a \dots x_b$
array [1.. n] of var set of $x_a \dots x_b$
array [1.. n] of var set of $\{ x_a, x_b, \dots, x_k \}$

In predicate declarations the following additional variable types are allowed.

Variable type
var set of int
array [1.. n] of var set of int

An array type appearing in a predicate declaration may use just int instead of 1.. n for the array index range in cases where the array argument can be of any length.

The string type

String literals and literal arrays of string literals can appear as annotation arguments, but not elsewhere. Strings have the same syntax as in C programs (namely, they are delimited by double quotes and the backslash character is used for escape sequences).

Examples:

```
""                                % The empty string.
"Hello."
"Hello,\nWorld\t\"quoted!\""
    % A string with an embedded newline, tab and
    ↴quotes.
```

4.3.1.3 Values and expressions

(See rule `<expr>` in Section 4.3.6)

Examples of literal values:

Type Literals bool true, false float 2.718, -1.0, 3.0e8 int -42, 0, 69 set of int {}, {2, 3, 5}, 1..10 arrays [], [y_a, \dots, y_k]

where each array element y_i is either: a non-array literal, or the name of a non-array parameter or variable, v . For example:

```
[1, 2, 3]          % Just literals  
[x, y, z]          % x, y, and z are variables or parameters.  
[x, 3]            % Mix of identifiers and literals
```

Section 4.3.6 gives the regular expressions specifying the syntax for float and int literals.

4.3.1.4 FlatZinc models

A FlatZinc model consists of:

1. zero or more external predicate declarations (i.e., a non-standard predicate that is supported directly by the target solver);
2. zero or more parameter declarations;
3. zero or more variable declarations;
4. zero or more constraints;
5. a solve goal

in that order.

FlatZinc uses the UTF-8 character set. Non-ASCII characters can only appear in string literals.

FlatZinc syntax is case sensitive (foo and Foo are different names). Identifiers start with a letter ([A-Za-z]) and are followed by any sequence of letters, digits, or underscores ([A-Za-z0-9_]). Additionally, identifiers of variable or parameter names may start with an underscore. Identifiers that correspond to the names of predicates, predicate parameters and annotations cannot have leading underscores.

The following keywords are reserved and cannot be used as identifiers: annotation, any, array, bool, case, constraint, diff, div, else, elseif, endif, enum, false, float, function, if, in, include, int, intersect, let, list, maximize, minimize, mod, not, of, satisfy, subset, superset, output, par, predicate, record, set, solve, string, symdiff, test, then, true, tuple, union, type, var, where, xor. Note that some of these keywords are not used in FlatZinc. They are reserved because they are keywords in Zinc and MiniZinc.

FlatZinc syntax is insensitive to whitespace.

4.3.1.5 Predicate declarations

(See rule `<predicate-item>` in Section 4.3.6)

Predicates used in the model that are not standard FlatZinc must be declared at the top of a FlatZinc model, before any other lexical items. Predicate declarations take the form

```
<predicate-item> ::= "predicate" <identifier> "(" [ <pred-param-type> :  
        ↴<identifier> "," ... ] ")" ";"
```

Annotations are not permitted anywhere in predicate declarations.

It is illegal to supply more than one predicate declaration for a given <identifier>.

Examples:

```
% m is the median value of {x, y, z}.
%
predicate median_of_3(var int: x, var int: y, var int: z, var int: m);

% all_different([x1, ..., xn]) iff
% for all i, j in 1..n: xi != xj.
%
predicate all_different(array [int] of var int: xs);

% exactly_one([x1, ..., xn]) iff
% there exists an i in 1..n: xi = true
% and for all j in 1..n: j != i -> xj = false.
%
predicate exactly_one(array [int] of var bool: xs);
```

4.3.1.6 Parameter declarations

(See rule param_decl in Section 4.3.6)

Parameters have fixed values and must be assigned values:

```
<par-decl-item> ::= <par-type> ":" <var-par-identifier> "=" <par-expr> ";"
```

where <par-type> is a parameter type, <var-par-identifier> is an identifier, and <par-expr> is a literal value (either a basic integer, float or bool literal, or a set or array of such literals).

Annotations are not permitted anywhere in parameter declarations.

Examples:

```
float: pi = 3.141;
array [1..7] of int: fib = [1, 1, 2, 3, 5, 8, 13];
bool: beer_is_good = true;
```

4.3.1.7 Variable declarations

(See rule var_decl in Section 4.3.6)

Variables have variable types and can be declared with optional assignments. The assignment can fix a variable to a literal value, or create an alias to another variable. Arrays of variables always have an assignment, defining them in terms of an array literal that can contain identifiers of variables or constant literals. Variables may be declared with zero or more annotations.

```

<var-decl-item> ::= <basic-var-type> ":" <var-par-identifier> <annotations>
  ↵ [ "=" <basic-expr> ] ;"
    | <array-var-type> ":" <var-par-identifier> <annotations>
      ↵ "=" <array-literal> ;"

```

where `<basic-var-type>` and `<array-var-type>` are variable types, `<var-par-identifier>` is an identifier, `<annotations>` is a (possibly empty) set of annotations, `<basic-expr>` is an identifier or a literal, and `<array-literal>` is a literal array value.

Examples:

```

var 0..9: digit;
var bool: b;
var set of 1..3: s;
var 0.0..1.0: x;
var int: y :: mip;           % 'mip' annotation: y should be a MIP variable.
array [1..3] of var 1..10: b = [y, 3, digit];

```

4.3.1.8 Constraints

(See rule `<constraint-item>` in Section 4.3.6)

Constraints take the following form and may include zero or more annotations:

```

<constraint-item> ::= "constraint" <identifier> "(" [ <expr> "," ... ] ")" ;
  ↵ <annotations> ;"

```

The arguments expressions (`<expr>`) can be literal values or identifiers.

Examples:

```

constraint int_le(0, x);      % 0 <= x
constraint int_lt(x, y);     % x < y
constraint int_le(y, 10);     % y <= 10
% 'domain': use domain consistency for this constraint:
% 2x + 3y = 10
constraint int_lin_eq([2, 3], [x, y], 10) :: domain;

```

4.3.1.9 Solve item

(See rule `<solve-item>` in Section 4.3.6)

A model finishes with a solve item, taking one of the following forms:

```

<solve-item> ::= "solve" <annotations> "satisfy" ;"
  | "solve" <annotations> "minimize" <basic-expr> ;"

```

```
| "solve" <annotations> "maximize" <basic-expr> ";"
```

The first alternative searches for any satisfying assignment, the second one searches for an assignment minimizing the given expression, and the third one for an assignment maximizing the expression. The `<basic-expr>` can be either a variable identifier or a literal value (if the objective function is constant).

A solution consists of a complete assignment where all variables in the model have been given a fixed value.

Examples:

```
solve satisfy;      % Find any solution using the default strategy.

solve minimize w;  % Find a solution minimizing w, using the default_
                   % strategy.

% First label the variables in xs in the order x[1], x[2], ...
% trying values in ascending order.
solve :: int_search(xs, input_order, indomain_min, complete)
       satisfy;    % Find any solution.

% First use first-fail on these variables, splitting domains
% at each choice point.
solve :: int_search([x, y, z], first_fail, indomain_split, complete)
       maximize x; % Find a solution maximizing x.
```

4.3.1.10 Annotations

Annotations are optional suggestions to the solver concerning how individual variables and constraints should be handled (e.g., a particular solver may have multiple representations for int variables) and how search should proceed. An implementation is free to ignore any annotations it does not recognise, although it should print a warning on the standard error stream if it does so. Annotations are unordered and idempotent: annotations can be reordered and duplicates can be removed without changing the meaning of the annotations.

An annotation is prefixed by `:::`, and either just an identifier or an expression that looks like a predicate call:

```
<annotations> ::= [ ":" <annotation> ]*
<annotation> ::= <identifier>
               | <identifier> "(" <ann-expr> ", " ... ")"
<ann-expr>   ::= <expr>
               | <annotation>
```

The arguments of the second alternative can be any expression or other annotations (without the leading `:::`).

Search annotations

While an implementation is free to ignore any or all annotations in a model, it is recommended that implementations at least recognise the following standard annotations for solve items.

```
seq_search([<searchannotation>, ...])
```

allows more than one search annotation to be specified in a particular order (otherwise annotations can be handled in any order).

A `<searchannotation>` is one of the following:

```
int_search(<vars>, <varchoiceannotation>, <assignmentannotation>,  
          ↴<strategyannotation>)

bool_search(<vars>, <varchoiceannotation>, <assignmentannotation>,  
            ↴<strategyannotation>)

set_search(<vars>, <varchoiceannotation>, <assignmentannotation>,  
           ↴<strategyannotation>)
```

where `<vars>` is the identifier of an array variable or an array literal specifying the variables to be assigned (ints, bools, or sets respectively). Note that these arrays may contain literal values.

`<varchoiceannotation>` specifies how the next variable to be assigned is chosen at each choice point. Possible choices are as follows (it is recommended that implementations support the starred options):

<code>input_order</code>	Choose variables in the order they appear in <code>vars</code> .
<code>first_fail</code>	Choose the variable with the smallest domain.
<code>anti_first_fail</code>	Choose the variable with the largest domain.
<code>smallest</code>	Choose the variable with the smallest value in its domain.
<code>largest</code>	Choose the variable with the largest value in its domain.
<code>occurrence</code>	Choose the variable with the largest number of attached constraints.
<code>most_constraints</code>	Choose the variable with the smallest domain, breaking ties using the number of constraints.
<code>max_regret</code>	Choose the variable with the largest difference between the two smallest values in its domain.
<code>dom_w_deg</code>	Choose the variable with the smallest value of domain size divided by weighted degree, where the weighted degree is the number of times the variables been in a constraint which failed

`<assignmentannotation>` specifies how the chosen variable should be constrained. Possible choices are as follows (it is recommended that implementations support at least the starred options):

indomain_min	★	Assign the smallest value in the variable's domain.
indomain_max	★	Assign the largest value in the variable's domain.
indomain_middle		Assign the value in the variable's domain closest to the mean of its current bounds.
indomain_median		Assign the middle value in the variable's domain.
indomain		Nondeterministically assign values to the variable in ascending order.
indomain_random		Assign a random value from the variable's domain.
indomain_split		Bisect the variable's domain, excluding the upper half first.
indomain_reverse	Bisect	Bisect the variable's domain, excluding the lower half first.
indomain_interval		If the variable's domain consists of several contiguous intervals, reduce the domain to the first interval. Otherwise just split the variable's domain.

Of course, not all assignment strategies make sense for all search annotations (e.g., `bool_search` and `indomain_split`).

Finally, `<strategyannotation>` specifies a search strategy; implementations should at least support complete (i.e., exhaustive search).

Output annotations

Model output is specified through variable annotations. Non-array output variables are annotated with `output_var`. Array output variables are annotated with `output_array([x1 .. x2 , ...])` where $x_1 \dots x_2$, ... are the index set ranges of the original MiniZinc array (which may have had multiple dimensions and/or index sets that do not start at 1). See [Section 4.3.2](#) for details on the output format.

Variable definition annotations

To support solvers capable of exploiting functional relationships, a variable defined as a function of other variables may be annotated thus:

```
var int: x :: is_defined_var;
...
constraint int_plus(y, z, x) :: defines_var(x);
```

(The `defines_var` annotation should appear on exactly one constraint.) This allows a solver to represent `x` internally as a representation of $y+z$ rather than as a separate constrained variable. The `is_defined_var` annotation on the declaration of `x` provides “early warning” to the solver that such an option is available.

Intermediate variables

Intermediate variables introduced during conversion of a MiniZinc model to FlatZinc may be annotated thus:

```
var int: X_INTRODUCED_3 :: var_is_introduced;
```

This information is potentially useful to the solver's search strategy.

Constraint annotations

Annotations can be placed on constraints advising the solver how the constraint should be implemented. Here are some constraint annotations supported by some solvers:

bounds or boundsZ	Use integer bounds propagation.
boundsR	Use real bounds propagation.
boundsD	A tighter version of boundsZ where support for the bounds must exist.
domain	Use domain propagation.
priority(k)	where k is an integer constant indicating propagator priority.

4.3.2 Output

An implementation can produce three types of output: solutions, statistics, and errors.

4.3.2.1 Solution output

An implementation must output values for all and only the variables annotated with `output_var` or `output_array` (`output` annotations must not appear on parameters). Output must be printed to the standard output stream.

For example:

```
var 1..10: x :: output_var;
var 1..10: y;          % y is not output.
% Output zs as a "flat" representation of a 2D array:
array [1..4] of var int: zs :: output_array([1..2, 1..2]);
```

All non-error output must be sent to the standard output stream.

Output must take the following form:

```
<var-par-identifier> = <basic-literal-expr> ;
```

or, for array variables,

```
<var-par-identifier> = array<N>d(<a>..<b>, ..., [<y1>, <y2>, ... <yk>]);
```

where `<N>` is the number of index sets specified in the corresponding `output_array` annotation, `<a>..`, ... are the index set ranges, and `<y1>`, `<y2>`, ... `<yk>` are literals of the element type.

Using this format, the output of a FlatZinc model solution is suitable for input to a MiniZinc model as a data file (this is why parameters are not included in the output).

Implementations must ensure that *all* model variables (not just the output variables) have satisfying assignments before printing a solution.

The output for a solution must be terminated with ten consecutive minus signs on a separate line: -----.

Multiple solutions may be output, one after the other, as search proceeds. How many solutions should be output depends on the mode the solver is run in as controlled by the -a command line flag (see [Section 4.3.4](#)).

If at least one solution has been found and search then terminates having explored the whole search space, then ten consecutive equals signs should be printed on a separate line: =====.

If no solutions have been found and search terminates having explored the whole search space, then =====UNSATISFIABLE===== should be printed on a separate line.

If the objective of an optimization problem is unbounded, then =====UNBOUNDED===== should be printed on a separate line.

If no solutions have been found and search terminates having *not* explored the whole search space, then =====UNKNOWN===== should be printed on a separate line.

Implementations may output further information about the solution(s), or lack thereof, in the form of FlatZinc comments.

Examples:

Asking for a single solution to this model:

```
var 1..3: x :: output_var;
solve satisfy
```

might produce this output:

```
x = 1;
-----
```

Asking for all solutions to this model:

```
array [1..2] of var 1..3: xs :: output_array([1..2]);
constraint int_lt(xs[1], xs[2]);    % x[1] < x[2].
solve satisfy
```

might produce this output:

```
xs = array1d(1..2, [1, 2]);
-----
xs = array1d(1..2, [1, 3]);
-----
xs = array1d(1..2, [2, 3]);
-----
```

```
=====
```

Asking for a single solution to this model:

```
var 1..10: x :: output_var;
solve maximize x;
```

should produce this output:

```
x = 10;
-----
=====
```

The row of equals signs indicates that a complete search was performed and that the last result printed is the optimal solution.

Running a solver on this model with some termination condition (such as a very short time-out):

```
var 1..10: x :: output_var;
solve maximize x;
```

might produce this output:

```
x = 1;
-----
x = 2;
-----
x = 3;
-----
```

Because the output does not finish with =====, search did not finish, hence these results must be interpreted as approximate solutions to the optimization problem.

Asking for a solution to this model:

```
var 1..3: x :: output_var;
var 4..6: y :: output_var;
constraint int_lt(y, x);    % y < x.
solve satisfy;
```

should produce this output:

```
=====UNSATISFIABLE=====
```

indicating that a complete search was performed and no solutions were found (i.e., the problem is unsatisfiable).

4.3.2.2 Statistics output

FlatZinc solvers can output statistics in a standard format so that it can be read by scripts, for example, in order to run experiments and automatically aggregate the results. Statistics should be printed to the standard output stream in the form of FlatZinc comments that follow a specific format. Statistics can be output at any time during the solving, i.e., before the first solution, between solutions, and after the search has finished.

Each value should be output on a line of its own in the following format:

```
%%mzn-stat: <name>=<value>
```

Each block of statistics is terminated by a line of its own with the following format:

```
%%mzn-stat-end
```

The `<name>` describes the kind of statistics gathered, and the `<value>` can be any value of a MiniZinc type. The following names are considered standard statistics:

Name	Type	Explanation
nodes	int	Number of search nodes
failures	int	Number of leaf nodes that were failed
restarts	int	Number of times the solver restarted the search
variables	int	Number of variables
intVariables	int	Number of integer variables created
boolVariables	int	Number of bool variables created
floatVariables	int	Number of float variables created
setVariables	int	Number of set variables created
propagators	int	Number of propagators created
propagations	int	Number of propagator invocations
peakDepth	int	Peak depth of search tree
nogoods	int	Number of nogoods created
backjumps	int	Number of backjumps
peakMem	float	Peak memory (in Mbytes)
initTime	float	Initialisation time (in seconds)
solveTime	float	Solving time (in seconds)

4.3.2.3 Error and warning output

Errors and warnings must be output to the standard error stream. When an error occurs, the implementation should exit with a non-zero exit code, signaling failure.

4.3.3 Solver-specific Libraries

Constraints in FlatZinc can call standard predicates as well as solver-specific predicates. Standard predicates are the ones that the MiniZinc compiler assumes to be present in all solvers.

Without further customisation, the compiler will try to compile the entire model into a set of these standard predicates.

Solvers can use custom predicates and *redefine* standard predicates by supplying a *solver specific library* of predicate declarations. Examples of such libraries can be found in the binary distribution of MiniZinc, inside the share/minizinc/gecode and share/minizinc/chuffed directories.

The solver-specific library needs to be made available to the MiniZinc compiler by specifying its location in the solver's configuration file, see [Section 4.3.5](#).

4.3.3.1 Standard predicates

FlatZinc solvers need to support the predicates listed as FlatZinc builtins in the library reference documentation, see [Section 4.2.6](#).

Any standard predicate that is not supported by a solver needs to be *redefined*. This can be achieved by placing a file called redefinitions.mzn in the solver's MiniZinc library, which can contain alternative definitions of predicates, or define them as unsupported using the abort predicate.

Example for a redefinitions.mzn:

```
% Redefine float_sinh function in terms of exp
predicate float_sinh(var float: a, var float: b) =
    b == (exp(a)-exp(-a))/2.0;

% Mark float_tanh as unsupported
predicate float_tanh(var float: a, var float: b) =
    abort("The builtin float_tanh is not supported by this solver.");
```

The redefinition can use the full MiniZinc language. Note, however, that redefining builtin predicates in terms of MiniZinc expressions can lead to problems if the MiniZinc compiler translates the high-level expression back to the redefined builtin.

The reference documentation ([Section 4.2.6](#)) also contains sections on builtins that were added in later versions of MiniZinc. In order to maintain backwards compatibility with solvers that don't support these, they are organised in redefinition files with a version number attached, such as redefinitions-2.0.mzn. In order to declare support for these builtins, the solver-specific library must contain the corresponding redefinitions file, with the predicates either redefined in terms of other predicates, or declared as supported natively by the solver by providing a predicate declaration without a body.

Example for a redefinitions-2.0.mzn that declares native support for the predicates added in MiniZinc 2.0:

```
predicate bool_clause_reif(array[int] of var bool: as,
                           array[int] of var bool: bs,
                           var bool: b);
predicate array_int_maximum(var int: m, array[int] of var int: x);
predicate array_float_maximum(var float: m, array[int] of var float: x);
predicate array_int_minimum(var int: m, array[int] of var int: x);
predicate array_float_minimum(var float: m, array[int] of var float: x);
```

4.3.3.2 Solver-specific predicates

Many solvers have built-in support for some of the constraints in the MiniZinc standard library. But without declaring which constraints they support, MiniZinc will assume that they don't support any except for the standard FlatZinc builtins mentioned in the section above.

A solver can declare that it supports a non-standard constraint by overriding one of the files of the standard library in its own solver-specific library. For example, assume that a solver supports the `all_different` constraint on integer variables. In the standard library, this constraint is defined in the file `all_different_int.mzn`, with the following implementation:

```
predicate all_different_int(array[int] of var int: x) =
  forall(i,j in index_set(x) where i < j) ( x[i] != x[j] );
```

A solver, let's call it *OptiSolve*, that supports this constraint natively can place a file with the same name, `all_different_int.mzn`, in its library, and redefine it as follows:

```
predicate optisolve_alldifferent(array[int] of var int: x);

predicate all_different_int(array[int] of var int: x) =
  optisolve_alldifferent(x);
```

When a MiniZinc model that contains the `all_different` constraint is now compiled with the *OptiSolve* library, it will contain calls to the newly defined predicate `optisolve_alldifferent`.

4.3.4 Command Line Interface

In order to work with the `minizinc` command line driver, a FlatZinc solver must be an executable (which can include e.g. shell scripts) that can be invoked as follows:

```
$ <executable-name> [options] model.fzn
```

where `<executable-name>` is the name of the executable. Solvers may support the following standard options:

- a
Instructs the solver to report *all* solutions in the case of satisfaction problems, or print *intermediate* solutions of increasing quality in the case of optimisation problems.
- n <i>
Instructs the solver to stop after reporting *i* solutions (only used with satisfaction problems).
- f
Instructs the solver to conduct a “free search”, i.e., ignore any search annotations. The solver is not *required* to ignore the annotations, but it is *allowed* to do so.
- s
Print statistics during and/or after the search for solutions. Statistics should be printed as

FlatZinc comments to the standard output stream. See below for a standard format for statistics.

-v

Print log messages (verbose solving) to the standard error stream. If solvers choose to print to standard output instead, all messages must be valid comments (i.e., start with a % character).

-p <i>

Run with i parallel threads (for multi-threaded solvers).

-r <i>

Use i as the random seed (for any random number generators the solver may be using).

4.3.5 Solver Configuration Files

In order for a solver to be available to MiniZinc, it has to be described in a *solver configuration file*. This is a simple file, in JSON or .dzn format, that contains some basic information such as the solver's name, version, where its library of global constraints can be found, and a path to its executable.

A solver configuration file must have file extension .msc (for MiniZinc Solver Configuration), and can be placed in any of the following locations:

- In the `minizinc/solvers/` directory of the MiniZinc installation. If you install MiniZinc from the binary distribution, this directory can be found at `/usr/share/minizinc/solvers` on Linux systems, inside the MiniZincIDE application on macOS system, and in the `Program Files\\MiniZinc IDE` (bundled) folder on Windows.
- In the directory `$HOME/.minizinc/solvers` on Linux and macOS systems, and the Application Data directory on Windows systems.
- In any directory listed on the `MZN_SOLVER_PATH` environment variable (directories are separated by : on Linux and macOS, and by ; on Windows systems).
- In any directory listed in the `mzn_solver_path` option of the global or user-specific configuration file (see [Section 3.1.4](#))
- Alternatively, you can use the MiniZinc IDE to create solver configuration files, see [Section 3.2.5.2](#) for details.

Solver configuration files must be valid JSON or .dzn files. As a JSON file, it must be an object with certain fields. As a .dzn file, it must consist of assignment items.

For example, a simple solver configuration in JSON format could look like this:

```
{  
  "name" : "My Solver",  
  "version": "1.0",  
  "id": "org.myorg.my_solver",  
  "executable": "fzn-mysolver"  
}
```

The same configuration in .dzn format would look like this:

```

name = "My Solver";
version = "1.0";
id = "org.myorg.my_solver";
executable = "fzn-mysolver";

```

Here is a list of all configuration options recognised by the configuration file parser. Any valid configuration file must at least contain the fields name, version, id, and executable.

- name (string, required): The name of the solver (displayed, together with the version, when you call `minizinc --solvers`, and in the MiniZinc IDE).
- version (string, required): The version of the solver.
- id (string, required): A unique identifier for the solver, “reverse domain name” notation.
- executable (string, required): The executable for this solver that can run FlatZinc files. This can be just a file name (in which case the solver has to be on the current PATH), or an absolute path to the executable, or a relative path (which is interpreted relative to the location of the configuration file).
- mznlib (string, default ""): The solver-specific library of global constraints and redefinitions. This should be the name of a directory (either an absolute path or a relative path, interpreted relative to the location of the configuration file). For solvers whose libraries are installed in the same location as the MiniZinc standard library, this can also take the form -G<solverlib>, e.g., -Ggecode (this is mostly the case for solvers that ship with the MiniZinc binary distribution).
- tags (list of strings, default empty): Each solver can have one or more tags that describe its features in an abstract way. Tags can be used for selecting a solver using the `--solver` option. There is no fixed list of tags, however we recommend using the following tags if they match the solver’s behaviour:
 - “cp”: for Constraint Programming solvers
 - “mip”: for Mixed Integer Programming solvers
 - “float”: for solvers that support float variables
 - “api”: for solvers that use the internal C++ API
- stdFlags (list of strings, default empty): Which of the standard solver command line flags are supported by this solver. The standard flags are `-a`, `-n`, `-s`, `-v`, `-p`, `-r`, `-f`.
- extraFlags (list of list of strings, default empty): Extra command line flags supported by the solver. Each entry must be a list of four strings. The first string is the name of the option (e.g. `--special-algorithm`). The second string is a description that can be used to generate help output (e.g. “which special algorithm to use”). The third string specifies the type of the argument (“int”, “bool”, “float” or “string”). The fourth string is the default value.
- supportsMzn (bool, default false): Whether the solver can run MiniZinc directly (i.e., it implements its own compilation or interpretation of the model).
- supportsFzn (bool, default true): Whether the solver can run FlatZinc. This should be the case for most solvers
- needsSolns2out (bool, default true): Whether the output of the solver needs to be passed through the MiniZinc output processor.

- `needsMznExecutable` (bool, default false): Whether the solver needs to know the location of the MiniZinc executable. If true, it will be passed to the solver using the `mzn-executable` option.
- `needsStdlibDir` (bool, default false): Whether the solver needs to know the location of the MiniZinc standard library directory. If true, it will be passed to the solver using the `stdlib-dir` option.
- `isGUIApplication` (bool, default false): Whether the solver has its own graphical user interface, which means that MiniZinc will detach from the process and not wait for it to finish or to produce any output.

4.3.6 Grammar

This is the full grammar for FlatZinc. It is a proper subset of the MiniZinc grammar (see Section 4.1.14). However, instead of specifying all the cases in the MiniZinc grammar that do *not* apply to FlatZinc, the BNF syntax below contains only the relevant syntactic constructs. It uses the same notation as in Section 4.1.2.

```
% A FlatZinc model
<model> ::= 
  [ <predicate-item> ]*
  [ <par-decl-item> ]*
  [ <var-decl-item> ]*
  [ <constraint-item> ]*
<solve-item>

% Predicate items
<predicate-item> ::= "predicate" <identifier> "(" [ <pred-param-type> :_
  ↪<identifier> "," ... ] ")" ";"

% Identifiers
<identifier> ::= [A-Za-z][A-Za-z0-9_]*

<basic-par-type> ::= "bool"
  | "int"
  | "float"
  | "set of int"

<par-type> ::= <basic-par-type>
  | "array" "[" <index-set> "]" "of" <basic-par-type>

<basic-var-type> ::= "var" "bool"
  | "var" "int"
  | "var" <int-literal> ".." <int-literal>
  | "var" "{" <int-literal> "," ... "}"
  | "var" "float"
  | "var" <float-literal> ".." <float-literal>
  | "var" "set" "of" <int-literal> ".." <int-literal>
  | "var" "set" "of" "{" [ <int-literal> "," ... ] "}"
```

```

<array-var-type> ::= "array" "[" <index-set> "]" "of" <basic-var-type>

<index-set> ::= "1" ".." <int-literal>

<basic-pred-param-type> ::= <basic-par-type>
    | <basic-var-type>
    | <int-literal> ".." <int-literal>
    | <float-literal> ".." <float-literal>
    | "{" <int-literal> "," ... "}"
    | "set" "of" <int-literal> .. <int-literal>
    | "set" "of" "{" [ <int-literal> "," ... ] "}"
    | "var" "set" "of" "int"

<pred-param-type> ::= <basic-pred-param-type>
    | "array" "[" <pred-index-set> "]" "of"_
    ↳<basic-pred-param-type>

<pred-index-set> ::= <index-set>
    | "int"

<basic-literal-expr> ::= <bool-literal>
    | <int-literal>
    | <float-literal>
    | <set-literal>

<basic-expr> ::= <basic-literal-expr>
    | <var-par-identifier>

<expr> ::= <basic-expr>
    | <array-literal>

<par-expr> ::= <basic-literal-expr>
    | <par-array-literal>

<var-par-identifier> ::= [A-Za-z_][A-Za-z0-9_]*

% Boolean literals
<bool-literal> ::= "false"
    | "true"

% Integer literals
<int-literal> ::= [0-9]+
    | 0x[0-9A-Fa-f]+
    | 0o[0-7]+

% Float literals
<float-literal> ::= [0-9]+.[0-9]+
    | [0-9]+.[0-9]+[Ee][-+]?[0-9]+
    | [0-9]+[Ee][-+]?[0-9]+

```

```
% Set literals
<set-literal> ::= "{" [ <int-literal> "," ... ] "}"
  | <int-literal> ".." <int-literal>
  | "{" [ <float-literal> "," ... ] "}"
  | <float-literal> ".." <float-literal>

<array-literal> ::= "[" [ <basic-expr> "," ... ] "]"
<par-array-literal> ::= "[" [ <basic-literal-expr> "," ... ] "]"

% Parameter declarations

<par-decl-item> ::= <par-type> ":" <var-par-identifier> "=" <par-expr> ";"

% Variable declarations

<var-decl-item> ::= <basic-var-type> ":" <var-par-identifier> <annotations>
  ↪[ "=" <basic-expr> ] ";"
  | <array-var-type> ":" <var-par-identifier> <annotations>
  ↪"=" <array-literal> ";"

% Constraint items

<constraint-item> ::= "constraint" <identifier> "(" [ <expr> "," ... ] ")"
  ↪<annotations> ";"

% Solve item

<solve-item> ::= "solve" <annotations> "satisfy" ;"
  | "solve" <annotations> "minimize" <basic-expr> ;"
  | "solve" <annotations> "maximize" <basic-expr> ;"

% Annotations

<annotations> ::= [ ":" <annotation> ]*
<annotation> ::= <identifier>
  | <identifier> "(" <ann-expr> "," ... ")"
<ann-expr> ::= <expr>
  | <annotation>

% End of FlatZinc grammar
```

Index

Symbols

command line option, 140
–compiler-statistics
 command line option, 138
–config-dirs
 command line option, 138
–error-msg
 command line option, 142
–fzn <file>, –output-fzn-to-file <file>
 command line option, 140
–help <id>
 command line option, 138
–help, -h
 command line option, 137
–ignore-stdlib
 command line option, 139
–instance-check-only
 command line option, 139
–keep-paths
 command line option, 141
–model-interface-only
 command line option, 139
–model-types-only
 command line option, 139
–no-flush-output
 command line option, 142
–no-optimize
 command line option, 139
–no-output-comments
 command line option, 142
–no-output-ozn, -O-
 command line option, 140
–non-unique
 command line option, 142
–only-range-domains
 command line option, 140
–output-base <name>
 command line option, 140
–output-mode <item|dzn|json>
 command line option, 141
–output-non-canonical <file>
 command line option, 142
–output-objective
 command line option, 141
–output-ozn-to-stdout
 command line option, 141
–output-paths
 command line option, 141
–output-paths-to-file <file>
 command line option, 141
–output-paths-to-stdout
 command line option, 141

–output-raw <file>
 command line option, 142
–output-time
 command line option, 142
–output-to-stdout, –output-fzn-to-stdout
 command line option, 141
–ozn-file <file>
 command line option, 141
–pre-passes <n>
 command line option, 140
–sac
 command line option, 140
–search-complete-msg <msg>
 command line option, 142
–shave
 command line option, 140
–soln-comma <s>, –solution-comma <s>
 command line option, 141
–soln-sep <s>, –soln-separator <s>, –solution-separator <s>
 command line option, 141
–solver <id>, –solver <solver configuration file>.msc
 command line option, 138
–solver-statistics
 command line option, 138
–solvers
 command line option, 138
–solvers-json
 command line option, 138
–stdlib-dir <dir>
 command line option, 139
–two-pass
 command line option, 140
–unbounded-msg
 command line option, 141
–unknown-msg
 command line option, 142
–unsat-msg (–unsatisfiable-msg)
 command line option, 141
–unsatorunbnd-msg
 command line option, 141
–use-gecode
 command line option, 140
–verbose-compilation
 command line option, 138
–verbose-solving
 command line option, 139
–version
 command line option, 137
–D <data>, –cmdline-data <data>

- command line option, 139
 - D fMIPdomains=false!command line option, 139
 - G -globals-dir -mzn-globals-dir <dir> command line option, 139
 - I -search-dir command line option, 139
 - O <n> command line option, 140
 - O<n> command line option, 140
 - Werror command line option, 141
 - a command line option, 351
 - a, -all-solutions command line option, 138
 - c, -canonicalize command line option, 142
 - c, -compile command line option, 138
 - d <file>, -data <file> command line option, 139
 - e, -model-check-only command line option, 139
 - f command line option, 351
 - f, -free-search command line option, 138
 - i <n>, -ignore-lines <n>, -ignore-leading-lines <n> command line option, 141
 - n <i> command line option, 351
 - n <i>, -num-solutions <i> command line option, 138
 - o <file>, -output-to-file <file> command line option, 141
 - p <i> command line option, 352
 - p <i>, -parallel <i> command line option, 139
 - r <i> command line option, 352
 - r <i>, -random-seed <i> command line option, 139
 - s command line option, 351
 - s, -statistics command line option, 138
 - v command line option, 352
 - v, -l, -verbose command line option, 138
 - =, 24
 - ==, 24
 - ~*
 - Option type support, 269
 - ~+ Option type support, 269
 - ~- Option type support, 269
 - ~= Option type support, 269
 - >, 24
 - >=, 24
 - <, 24
 - <=, 24
- A**
- abort Builtins, 316
 - abs Builtins, 278
 - absent Option type support, 264, 265, 269
 - acos Builtins, 282
 - acosh Builtins, 282
 - add_to_output Annotations, 255
 - aggregation function, 43
 - exists, 43
 - forall, 43
 - iffall, 43
 - max, 43
 - min, 43
 - product, 43
 - sum, 43
 - xorall, 43
 - all_different Global constraints, 235
 - all_disjoint Global constraints, 235
 - all_equal Global constraints, 235
 - alldifferent, 61
 - alldifferent_except_0 Global constraints, 236
 - alternative Global constraints, 248

among
 Global constraints, 241

ann, 91

annotation, 32, 85, 91

Annotations

- add_to_output, 255
- anti_first_fail, 258
- bool_search, 261
- bounds, 258
- complete, 260
- constraint_name, 257
- defines_var, 257
- doc_comment, 257
- dom_w_deg, 258
- domain, 258
- expression_name, 257
- first_fail, 258
- float_search, 262
- impact, 258
- indomain, 259
- indomain_interval, 259
- indomain_max, 259
- indomain_median, 259
- indomain_middle, 259
- indomain_min, 259
- indomain_random, 260
- indomain_reverse_split, 260
- indomain_split, 260
- indomain_split_random, 260
- input_order, 258
- int_search, 262
- is_defined_var, 255
- is_reverse_map, 256
- largest, 258
- max_regret, 258
- maybe_partial, 256
- most_constrained, 259
- mzn_break_here, 256
- mzn_check_enum_var, 257
- mzn_check_var, 256
- mzn_constraint_name, 257
- mzn_expression_name, 257
- mzn_path, 257
- mzn_rhs_from_assignment, 256
- occurrence, 259
- outdomain_max, 260
- outdomain_median, 260
- outdomain_min, 260
- outdomain_random, 260
- output_array, 257
- output_only, 256

 output_var, 256

 promise_total, 256

 restart_constant, 261

 restart_geometric, 261

 restart_linear, 261

 restart_luby, 261

 restart_none, 261

 seq_search, 262

 set_search, 262, 263

 smallest, 259

 var_is_introduced, 256

 warm_start, 263, 264

 warm_start_array, 264

anti_first_fail
 Annotations, 258

arg_max
 Builtins, 278
 Global constraints, 251

arg_min
 Builtins, 278
 Global constraints, 251

arg_sort
 Builtins, 306
 Global constraints, 239

argument, 70, 79

array, 35

- access, 41, 54
- index set, 40
 - unbounded, 70
- literal
 - 1D, 41
 - 2D, 41

array1d
 Builtins, 291, 292

array2d
 Builtins, 292

array3d
 Builtins, 292, 293

array4d
 Builtins, 293, 294

array5d
 Builtins, 294, 295

array6d
 Builtins, 295, 296

array_bool_and
 FlatZinc builtins, 325

array_bool_element
 FlatZinc builtins, 325

array_bool_or
 FlatZinc builtins, 325

array_bool_xor

FlatZinc builtins, 325
array_float_element
 FlatZinc builtins, 329
array_float_maximum
 FlatZinc builtins, 329, 334
array_float_minimum
 FlatZinc builtins, 329, 334
array_int_element
 FlatZinc builtins, 322
array_int_maximum
 FlatZinc builtins, 322, 334
array_int_minimum
 FlatZinc builtins, 322, 335
array_intersect
 Builtins, 289, 290
array_set_element
 FlatZinc builtins, 327
array_union
 Builtins, 290
array_var_bool_element
 FlatZinc builtins, 325
array_var_bool_element_nonshifted
 FlatZinc builtins, 335
array_var_float_element
 FlatZinc builtins, 329
array_var_float_element_nonshifted
 FlatZinc builtins, 335
array_var_int_element
 FlatZinc builtins, 322
array_var_int_element_nonshifted
 FlatZinc builtins, 335
array_var_set_element
 FlatZinc builtins, 327
array_var_set_element_nonshifted
 FlatZinc builtins, 335
arrayXd
 Builtins, 296
asin
 Builtins, 282, 283
asinh
 Builtins, 283
assert, 28
 Builtins, 316, 317
assignment, 22, 75
at_least
 Global constraints, 242
at_most
 Global constraints, 242
at_most1
 Global constraints, 242
atan
Builtins, 283
atanh
 Builtins, 283
B
beroulli
 Builtins, 318
bin_packing
 Global constraints, 244
bin_packing_capa
 Global constraints, 245
bin_packing_load
 Global constraints, 245
binomial
 Builtins, 319
bool2float
 Builtins, 308
bool2int, 50, 55
 Builtins, 308, 309
 FlatZinc builtins, 325
 Option type support, 267
bool_and
 FlatZinc builtins, 325
bool_clause
 FlatZinc builtins, 326
bool_clause_reif
 FlatZinc builtins, 335
bool_eq
 FlatZinc builtins, 326
 Option type support, 265
bool_eq_reif
 FlatZinc builtins, 326
bool_le
 FlatZinc builtins, 326
bool_le_reif
 FlatZinc builtins, 326
bool_lin_eq
 FlatZinc builtins, 326
bool_lin_le
 FlatZinc builtins, 326
bool_lt
 FlatZinc builtins, 326
bool_lt_reif
 FlatZinc builtins, 326
bool_not
 FlatZinc builtins, 326
bool_or
 FlatZinc builtins, 327
bool_search, 90
 Annotations, 261
bool_xor

FlatZinc builtins, 327
Boolean, 22, 50
bounds
 Annotations, 258
Builtins
 '
 '=, 271
 '*', 275
 '+', 275
 '++', 290, 291, 310
 '-', 276
 '->', 284, 285
 '..', 287
 '/', 277
 '/\', 285
 '=', 272, 273
 '^', 277
 '\\', 285
 '>', 274
 '>=', 274
 '<', 271
 '<-', 285
 '<->', 285
 '<=', 272
 'diff', 287, 288
 'div', 277
 'in', 288
 'intersect', 288
 'mod', 278
 'not', 286
 'subset', 288, 289
 'superset', 289
 'symdiff', 289
 'union', 289
 'xor', 286
 abort, 316
 abs, 278
 acos, 282
 acosh, 282
 arg_max, 278
 arg_min, 278
 arg_sort, 306
 array1d, 291, 292
 array2d, 292
 array3d, 292, 293
 array4d, 293, 294
 array5d, 294, 295
 array6d, 295, 296
 array_intersect, 289, 290
 array_union, 290
 arrayXd, 296
 asin, 282, 283
 asinh, 283
 assert, 316, 317
 atan, 283
 atanh, 283
 bernoulli, 318
 binomial, 319
 bool2float, 308
 bool2int, 308, 309
 card, 290
 cauchy, 319
 ceil, 309
 chisquared, 319
 clause, 286
 col, 297
 concat, 310
 cos, 283
 cosh, 283
 discrete_distribution, 319
 dom, 313
 dom_array, 313
 dom_bounds_array, 313
 dom_size, 313
 enum_next, 317, 318
 enum_prev, 318
 exists, 286
 exp, 281
 exponential, 319
 fdistribution, 319
 file_path, 310
 fix, 313
 floor, 309
 forall, 287
 format, 310, 311
 format_justify_string, 311
 gamma, 320
 has_bounds, 313
 has_element, 297
 has_index, 297
 has_ub_set, 314
 iffall, 287
 implied_constraint, 321
 index_set, 297
 index_set_1of2, 297
 index_set_1of3, 298
 index_set_1of4, 298
 index_set_1of5, 298
 index_set_1of6, 298
 index_set_2of2, 298
 index_set_2of3, 298
 index_set_2of4, 298

index_set_2of5, 298
 index_set_2of6, 298
 index_set_3of3, 298
 index_set_3of4, 299
 index_set_3of5, 299
 index_set_3of6, 299
 index_set_4of4, 299
 index_set_4of5, 299
 index_set_4of6, 299
 index_set_5of5, 299
 index_set_5of6, 299
 index_set_6of6, 299
 index_sets_agree, 300
 int2float, 309
 is_fixed, 314
 join, 311
 lb, 314
 lb_array, 314, 315
 length, 300
 ln, 281
 log, 282
 log10, 282
 log2, 282
 lognormal, 320
 max, 279, 290
 min, 279, 280, 290
 mzn_compiler_version, 321
 mzn_version_to_string, 322
 normal, 320
 outputJSON, 311
 outputJSONParameters, 311
 poisson, 320
 pow, 280
 product, 280
 redundant_constraint, 321
 reverse, 300
 round, 309
 row, 300, 301
 set2array, 309
 show, 312
 show2d, 312
 show3d, 312
 show_float, 312
 show_int, 312
 showJSON, 312
 sin, 284
 sinh, 284
 slice_1d, 301
 slice_2d, 301, 302
 slice_3d, 302, 303
 slice_4d, 303, 304
 slice_5d, 304, 305
 slice_6d, 305, 306
 sort, 306
 sort_by, 307
 sqrt, 281
 string_length, 313
 sum, 281
 symmetry_breaking_constraint, 321
 tan, 284
 tanh, 284
 tdistribution, 320
 to_enum, 318
 trace, 317
 trace_stdout, 317
 ub, 315
 ub_array, 315, 316
 uniform, 321
 weibull, 321
 xorall, 287

C

card
 Builtins, 290
 cauchy
 Builtins, 319
 ceil
 Builtins, 309
 chisquared
 Builtins, 319
 circuit
 Global constraints, 252
 clause
 Builtins, 286
 coercion
 automatic, 56
 bool2int, 56
 int2float, 56
 col
 Builtins, 297
 command line option
 -
 -MIPDMaxDensEE <n>, 139
 -MIPDMaxIntvEE <n>, 139
 -allow-multiple-assignments, 140
 -compile-solution-checker
 <file>.mzc.mzn, 140
 -compiler-statistics, 138
 -config-dirs, 138
 -error-msg, 142
 -fzn <file>, -output-fzn-to-file <file>, 140

-help <id>, 138
-help, -h, 137
-ignore-stdlib, 139
-instance-check-only, 139
-keep-paths, 141
-model-interface-only, 139
-model-types-only, 139
-no-flush-output, 142
-no-optimize, 139
-no-output-comments, 142
-no-output-ozn, -O-, 140
-non-unique, 142
-only-range-domains, 140
-output-base <name>, 140
-output-mode <item|dzn|json>, 141
-output-non-canonical <file>, 142
-output-objective, 141
-output-ozn-to-stdout, 141
-output-paths, 141
-output-paths-to-file <file>, 141
-output-paths-to-stdout, 141
-output-raw <file>, 142
-output-time, 142
-output-to-stdout, –output-fzn-to-stdout,
 141
-ozn-file <file>, 141
-pre-passes <n>, 140
-sac, 140
-search-complete-msg <msg>, 142
-shave, 140
-soln-comma <s>, –solution-commma
 <s>, 141
-soln-sep <s>, –soln-separator <s>, –
 solution-separator <s>, 141
-solver <id>, –solver <solver configura-
 tion file>.msc, 138
-solver-statistics, 138
-solvers, 138
-solvers-json, 138
-stdlib-dir <dir>, 139
-two-pass, 140
-unbounded-msg, 141
-unknown-msg, 142
-unsat-msg (-unsatisfiable-msg), 141
-unsatorunbnd-msg, 141
-use-gecode, 140
-verbose-compilation, 138
-verbose-solving, 139
-version, 137
-D <data>, –cmdline-data <data>, 139
-D fMIPdomains=false|hyperpage, 139
-G –globals-dir –mzn-globals-dir <dir>,
 139
-I –search-dir, 139
-O, –ozn, –output-ozn-to-file <file>, 140
-O<n>, 140
-Werror, 141
-a, 351
-a, –all-solutions, 138
-c, –canonicalize, 142
-c, –compile, 138
-d <file>, –data <file>, 139
-e, –model-check-only, 139
-f, 351
-f, –free-search, 138
-i <n>, –ignore-lines <n>, –ignore-
 leading-lines <n>, 141
-n <i>, 351
-n <i>, –num-solutions <i>, 138
-o <file>, –output-to-file <file>, 141
-p <i>, 352
-p <i>, –parallel <i>, 139
-r <i>, 352
-r <i>, –random-seed <i>, 139
-s, 351
-s, –statistics, 138
-v, 352
-v, –l, –verbose, 138

Compiler options

- mzn_check_only_range_domains, 270
- mzn_min_version_required, 270
- mzn_opt_only_range_domains, 270

complete

- Annotations, 260

comprehension, 79

- generator, 42
- list, 44
- set, 42

concat

- Builtins, 310

constraint, 23

- complex, 50
- higher order, 55
- local, 76
- redundant, 98
- set, 56

constraint_name

- Annotations, 257

context, 74

- mixed, 76
- negative, 74, 76

cos

Builtins, 283
cosh
 Builtins, 283
count
 Global constraints, 242
count_eq
 Global constraints, 242
count_geq
 Global constraints, 242
count_gt
 Global constraints, 242
count_leq
 Global constraints, 243
count_lt
 Global constraints, 243
count_neq
 Global constraints, 243
cumulative, 61
 Global constraints, 248

D

data file, 26
 command line, 28
decision variable, *see* variable
decreasing
 Global constraints, 240
defines_var
 Annotations, 257
deopt
 Option type support, 265, 269, 270
DFA, 65
diffn
 Global constraints, 245
diffn_k
 Global constraints, 245
diffn_nonstrict
 Global constraints, 246
diffn_nonstrict_k
 Global constraints, 246
discrete_distribution
 Builtins, 319
disjoint
 Global constraints, 252
disjunctive
 Global constraints, 249
disjunctive_strict
 Global constraints, 249
distribute
 Global constraints, 243
div, 26
doc_comment

Annotations, 257
dom
 Builtins, 313
dom_array
 Builtins, 313
dom_bounds_array
 Builtins, 313
dom_size
 Builtins, 313
dom_w_deg, 90
 Annotations, 258
domain, 23
 Annotations, 258
 reflection, 77, 78

E

element
 Option type support, 265–268
enum_next
 Builtins, 317, 318
enum_prev
 Builtins, 318
enumerated type, 40, 41
enumerated types, 32
exactly
 Global constraints, 243
exists, 43
 Builtins, 286
 Option type support, 266
exp
 Builtins, 281
exponential
 Builtins, 319
expression, 91
 arithmetic, 26
 assert, 70
 Boolean, 50, 69
 conditional, 46
 generator call, 43
 let, 76
expression_name
 Annotations, 257

F

false, 50
fdistribution
 Builtins, 319
file_path
 Builtins, 310
first_fail, 90
 Annotations, 258
fix, 60

Builtins, 313
fixed, 32, 42, 60
FlatZinc builtins
 array_bool_and, 325
 array_bool_element, 325
 array_bool_or, 325
 array_bool_xor, 325
 array_float_element, 329
 array_float_maximum, 329, 334
 array_float_minimum, 329, 334
 array_int_element, 322
 array_int_maximum, 322, 334
 array_int_minimum, 322, 335
 array_set_element, 327
 array_var_bool_element, 325
 array_var_bool_element_nonshifted, 335
 array_var_float_element, 329
 array_var_float_element_nonshifted, 335
 array_var_int_element, 322
 array_var_int_element_nonshifted, 335
 array_var_set_element, 327
 array_var_set_element_nonshifted, 335
bool2int, 325
bool_and, 325
bool_clause, 326
bool_clause_reif, 335
bool_eq, 326
bool_eq_reif, 326
bool_le, 326
bool_le_reif, 326
bool_lin_eq, 326
bool_lin_le, 326
bool_lt, 326
bool_lt_reif, 326
bool_not, 326
bool_or, 327
bool_xor, 327
float_abs, 330
float_acos, 330
float_acosh, 330
float_asin, 330
float_asinh, 330
float_atan, 330
float_atanh, 330
float_cos, 330
float_cosh, 330
float_div, 330
float_dom, 331, 336
float_eq, 331
float_eq_reif, 331
float_exp, 331
float_in, 331, 336
float_in_reif, 331
float_le, 331
float_le_reif, 331
float_lin_eq, 331
float_lin_eq_reif, 331
float_lin_le, 332
float_lin_le_reif, 332
float_lin_lt, 332
float_lin_lt_reif, 332
float_lin_ne, 332
float_lin_ne_reif, 332
float_ln, 332
float_log10, 333
float_log2, 333
float_lt, 333
float_lt_reif, 333
float_max, 333
float_min, 333
float_ne, 333
float_ne_reif, 333
float_plus, 333
float_pow, 333
float_sin, 333
float_sinh, 334
float_sqrt, 334
float_tan, 334
float_tanh, 334
float_times, 334
int2float, 334
int_abs, 322
int_div, 322
int_eq, 322
int_eq_reif, 323
int_le, 323
int_le_reif, 323
int_lin_eq, 323
int_lin_eq_reif, 323
int_lin_le, 323
int_lin_le_reif, 323
int_lin_ne, 323
int_lin_ne_reif, 324
int_lt, 324
int_lt_reif, 324
int_max, 324
int_min, 324
int_mod, 324
int_ne, 324
int_ne_reif, 324
int_plus, 324
int_pow, 324

```

int_pow_fixed, 325
int_times, 325
max, 336
min, 336
set_card, 327
set_diff, 327
set_eq, 327
set_eq_reif, 327
set_in, 328
set_in_reif, 328
set_intersect, 328
set_le, 328
set_lt, 328
set_ne, 328
set_ne_reif, 328
set_subset, 329
set_subset_reif, 329
set_superset, 329
set_symdiff, 329
set_union, 329
float_abs
    FlatZinc builtins, 330
float_acos
    FlatZinc builtins, 330
float_acosh
    FlatZinc builtins, 330
float_asin
    FlatZinc builtins, 330
float_asinh
    FlatZinc builtins, 330
float_atan
    FlatZinc builtins, 330
float_atanh
    FlatZinc builtins, 330
float_cos
    FlatZinc builtins, 330
float_cosh
    FlatZinc builtins, 330
float_div
    FlatZinc builtins, 330
float_dom
    FlatZinc builtins, 331, 336
float_eq
    FlatZinc builtins, 331
float_eq_reif
    FlatZinc builtins, 331
float_exp
    FlatZinc builtins, 331
float_in
    FlatZinc builtins, 331, 336
float_in_reif
    FlatZinc builtins, 331
float_le
    FlatZinc builtins, 331
float_le_reif
    FlatZinc builtins, 331
float_lin_eq
    FlatZinc builtins, 331
float_lin_eq_reif
    FlatZinc builtins, 331
float_lin_le
    FlatZinc builtins, 332
float_lin_le_reif
    FlatZinc builtins, 332
float_lin_lt
    FlatZinc builtins, 332
float_lin_lt_reif
    FlatZinc builtins, 332
float_lin_ne
    FlatZinc builtins, 332
float_lin_ne_reif
    FlatZinc builtins, 332
float_ln
    FlatZinc builtins, 332
float_log10
    FlatZinc builtins, 333
float_log2
    FlatZinc builtins, 333
float_lt
    FlatZinc builtins, 333
float_lt_reif
    FlatZinc builtins, 333
float_max
    FlatZinc builtins, 333
float_min
    FlatZinc builtins, 333
float_ne
    FlatZinc builtins, 333
float_ne_reif
    FlatZinc builtins, 333
float_plus
    FlatZinc builtins, 333
float_pow
    FlatZinc builtins, 333
float_search
    Annotations, 262
float_sin
    FlatZinc builtins, 333
float_sinh
    FlatZinc builtins, 334
float_sqrt
    FlatZinc builtins, 334

```

float_tan
 FlatZinc builtins, 334

float_tanh
 FlatZinc builtins, 334

float_times
 FlatZinc builtins, 334

floor
 Builtins, 309

forall, 42, 43
 Builtins, 287

 Option type support, 266

format
 Builtins, 310, 311

format_justify_string
 Builtins, 311

function, 60, 74
 definition, 70, 72

G

gamma
 Builtins, 320

generator, 96

generator call, 43

geost
 Global constraints, 246

geost_bb
 Global constraints, 246

geost_smallest_bb
 Global constraints, 247

global constraint, 61
 alldifferent, 61
 cumulative, 61
 disjunctive, 73
 regular, 65
 table, 63

Global constraints
 all_different, 235
 all_disjoint, 235
 all_equal, 235
 alldifferent_except_0, 236
 alternative, 248
 among, 241
 arg_max, 251
 arg_min, 251
 arg_sort, 239
 at_least, 242
 at_most, 242
 at_most1, 242
 bin_packing, 244
 bin_packing_capa, 245
 bin_packing_load, 245

circuit, 252

count, 242

count_eq, 242

count_geq, 242

count_gt, 242

count_leq, 243

count_lt, 243

count_neq, 243

cumulative, 248

decreasing, 240

diffn, 245

diffn_k, 245

diffn_nonstrict, 246

diffn_nonstrict_k, 246

disjoint, 252

disjunctive, 249

disjunctive_strict, 249

distribute, 243

exactly, 243

geost, 246

geost_bb, 246

geost_smallest_bb, 247

global_cardinality, 243

global_cardinality_closed, 244

global_cardinality_low_up, 244

global_cardinality_low_up_closed, 244

increasing, 240

int_set_channel, 241

inverse, 241

inverse_set, 241

knapsack, 248

lex2, 236

lex_greater, 236

lex_greatereq, 237

lex_less, 237, 238

lex_lesseq, 238

link_set_to_booleans, 241

maximum, 252

maximum_arg, 252

member, 252, 253

minimum, 253

minimum_arg, 253

network_flow, 253

network_flow_cost, 254

nvalue, 236

partition_set, 254

range, 254

regular, 250

regular_nfa, 251

roots, 254

sliding_sum, 255

sort, 240
 span, 249
 strict_lex2, 238
 subcircuit, 255
 sum_pred, 255
 symmetric_all_different, 236
 table, 251
 value_precede, 238
 value_precede_chain, 239
 global_cardinality
 Global constraints, 243
 global_cardinality_closed
 Global constraints, 244
 global_cardinality_low_up
 Global constraints, 244
 global_cardinality_low_up_closed
 Global constraints, 244

H

has_bounds
 Builtins, 313
 has_element
 Builtins, 297
 has_index
 Builtins, 297
 has_ub_set
 Builtins, 314

I

iffall, 43
 Builtins, 287
 impact
 Annotations, 258
 implied_constraint
 Builtins, 321
 increasing
 Global constraints, 240
 index_set
 Builtins, 297
 index_set_1of2
 Builtins, 297
 index_set_1of3
 Builtins, 298
 index_set_1of4
 Builtins, 298
 index_set_1of5
 Builtins, 298
 index_set_1of6
 Builtins, 298
 index_set_2of2
 Builtins, 298
 index_set_2of3

Builtins, 298
 index_set_2of4
 Builtins, 298
 index_set_2of5
 Builtins, 298
 index_set_2of6
 Builtins, 298
 index_set_3of3
 Builtins, 298
 index_set_3of4
 Builtins, 299
 index_set_3of5
 Builtins, 299
 index_set_3of6
 Builtins, 299
 index_set_4of4
 Builtins, 299
 index_set_4of5
 Builtins, 299
 index_set_4of6
 Builtins, 299
 index_set_5of5
 Builtins, 299
 index_set_5of6
 Builtins, 299
 index_set_6of6
 Builtins, 299
 index_sets_agree
 Builtins, 300
 indomain
 Annotations, 259
 indomain_interval
 Annotations, 259
 indomain_max
 Annotations, 259
 indomain_median, 90
 Annotations, 259
 indomain_middle
 Annotations, 259
 indomain_min, 90
 Annotations, 259
 indomain_random, 90
 Annotations, 260
 indomain_reverse_split
 Annotations, 260
 indomain_split, 90
 Annotations, 260
 indomain_split_random
 Annotations, 260
 input_order, 90
 Annotations, 258

int2float
 Builtins, 309
 FlatZinc builtins, 334

int_abs
 FlatZinc builtins, 322

int_div
 FlatZinc builtins, 322

int_eq
 FlatZinc builtins, 322
 Option type support, 268

int_eq_reif
 FlatZinc builtins, 323

int_le
 FlatZinc builtins, 323

int_le_reif
 FlatZinc builtins, 323

int_lin_eq
 FlatZinc builtins, 323

int_lin_eq_reif
 FlatZinc builtins, 323

int_lin_le
 FlatZinc builtins, 323

int_lin_le_reif
 FlatZinc builtins, 323

int_lin_ne
 FlatZinc builtins, 323

int_lin_ne_reif
 FlatZinc builtins, 324

int_lt
 FlatZinc builtins, 324

int_lt_reif
 FlatZinc builtins, 324

int_max
 FlatZinc builtins, 324

int_min
 FlatZinc builtins, 324

int_mod
 FlatZinc builtins, 324

int_ne
 FlatZinc builtins, 324
 Option type support, 268

int_ne_reif
 FlatZinc builtins, 324

int_plus
 FlatZinc builtins, 324

int_pow
 FlatZinc builtins, 324

int_pow_fixed
 FlatZinc builtins, 325

int_search, 90
 Annotations, 262

int_set_channel
 Global constraints, 241

int_times
 FlatZinc builtins, 325

integer, 22

inverse
 Global constraints, 241

inverse_set
 Global constraints, 241

is_defined_var
 Annotations, 255

is_fixed
 Builtins, 314

is_reverse_map
 Annotations, 256

item, 31
 annotation, 33, 91
 assignment, 32
 constraint, 32
 enum, 33
 include, 31
 output, 33
 predicate, 33
 solve, 32
 variable declaration, 32

J

join
 Builtins, 311

K

knapsack
 Global constraints, 248

L

largest
 Annotations, 258

lb
 Builtins, 314

lb_array
 Builtins, 314, 315

length
 Builtins, 300

let, 73

lex2
 Global constraints, 236

lex_greater
 Global constraints, 236

lex_greatereq
 Global constraints, 237

lex_less
 Global constraints, 237, 238

lex_lesseq
 Global constraints, 238
link_set_to_booleans
 Global constraints, 241
list, 40
ln
 Builtins, 281
log
 Builtins, 282
log10
 Builtins, 282
log2
 Builtins, 282
lognormal
 Builtins, 320

M

max, 43
 Builtins, 279, 290
 FlatZinc builtins, 336
 Option type support, 268
max_regret
 Annotations, 258
maximize, 26
maximum
 Global constraints, 252
maximum_arg
 Global constraints, 252
maybe_partial
 Annotations, 256
member
 Global constraints, 252, 253
min, 43
 Builtins, 279, 280, 290
 FlatZinc builtins, 336
 Option type support, 268
minimize, 26
minimum
 Global constraints, 253
minimum_arg
 Global constraints, 253
minizinc -c, 117
mod, 26
most_constrained
 Annotations, 259
mzn_break_here
 Annotations, 256
mzn_check_enum_var
 Annotations, 257
mzn_check_only_range_domains
 Compiler options, 270

mzn_check_var
 Annotations, 256
mzn_compiler_version
 Builtins, 321
mzn_constraint_name
 Annotations, 257
mzn_expression_name
 Annotations, 257
mzn_min_version_required
 Compiler options, 270
mzn_opt_only_range_domains
 Compiler options, 270
mzn_path
 Annotations, 257
mzn_rhs_from_assignment
 Annotations, 256
mzn_version_to_string
 Builtins, 322

N

network_flow
 Global constraints, 253
network_flow_cost
 Global constraints, 254
NFA, 68
normal
 Builtins, 320
nvalue
 Global constraints, 236

O

objective, 26, 74
occurrence
 Annotations, 259
occurs
 Option type support, 266, 269, 270
operator
 Boolean, 50
 integer, 26
 relational, 24
 set, 38
optimization, 26
option type, 42
 Option type support
 '>', 267
 '>=', 267
 '<', 266
 '<=' , 266, 267
 'not', 264
 '~*', 269
 '~+', 269
 '~-', 269

~=[269](#)
absent, [264](#), [265](#), [269](#)
bool2int, [267](#)
bool_eq, [265](#)
deopt, [265](#), [269](#), [270](#)
element, [265](#)–[268](#)
exists, [266](#)
forall, [266](#)
int_eq, [268](#)
int_ne, [268](#)
max, [268](#)
min, [268](#)
occurs, [266](#), [269](#), [270](#)
product, [269](#)
sum, [269](#)
option types, [81](#)
outdomain_max
 Annotations, [260](#)
outdomain_median
 Annotations, [260](#)
outdomain_min
 Annotations, [260](#)
outdomain_random
 Annotations, [260](#)
output, [24](#), [48](#)
output_array
 Annotations, [257](#)
output_only, [101](#)
 Annotations, [256](#)
output_var
 Annotations, [256](#)
outputJSON
 Builtins, [311](#)
outputJSONParameters
 Builtins, [311](#)

P

parameter, [22](#), [73](#), [91](#)
partition_set
 Global constraints, [254](#)
poisson
 Builtins, [320](#)
pow
 Builtins, [280](#)
predicate, [60](#), [74](#)
 definition, [68](#), [70](#), [73](#)
product, [43](#)
 Builtins, [280](#)
 Option type support, [269](#)
promise_total
 Annotations, [256](#)

R

range, [32](#), [38](#), [65](#)
 float, [32](#)
 Global constraints, [254](#)
 integer, [32](#)
redundant_constraint
 Builtins, [321](#)
regular, [65](#)
 Global constraints, [250](#)
regular_nfa
 Global constraints, [251](#)
reification, [124](#)
restart_constant, [93](#)
 Annotations, [261](#)
restart_geometric, [93](#)
 Annotations, [261](#)
restart_linear, [93](#)
 Annotations, [261](#)
restart_luby, [93](#)
 Annotations, [261](#)
restart_none, [93](#)
 Annotations, [261](#)
reverse
 Builtins, [300](#)
roots
 Global constraints, [254](#)
round
 Builtins, [309](#)
row
 Builtins, [300](#), [301](#)
runtime flag
 –all-solutions, [48](#)
 -a, [48](#)

S

satisfaction, [24](#)
scope, [79](#)
search, [85](#)
 annotation, [89](#)
 constrain choice, [90](#)
 depth first, [85](#)
 finite domain, [85](#)
 restart, [92](#)
 sequential, [90](#)
 variable choice, [90](#)
seq_search, [90](#)
 Annotations, [262](#)
set, [38](#)
set2array
 Builtins, [309](#)
set_card

FlatZinc builtins, 327
set_diff
 FlatZinc builtins, 327
set_eq
 FlatZinc builtins, 327
set_eq_reif
 FlatZinc builtins, 327
set_in
 FlatZinc builtins, 328
set_in_reif
 FlatZinc builtins, 328
set_intersect
 FlatZinc builtins, 328
set_le
 FlatZinc builtins, 328
set_lt
 FlatZinc builtins, 328
set_ne
 FlatZinc builtins, 328
set_ne_reif
 FlatZinc builtins, 328
set_search, 90
 Annotations, 262, 263
set_subset
 FlatZinc builtins, 329
set_subset_reif
 FlatZinc builtins, 329
set_superset
 FlatZinc builtins, 329
set_symdiff
 FlatZinc builtins, 329
set_union
 FlatZinc builtins, 329
show, 24
 Builtins, 312
show2d
 Builtins, 312
show3d
 Builtins, 312
show_float
 Builtins, 312
show_int
 Builtins, 312
showJSON
 Builtins, 312
sin
 Builtins, 284
single enum, 40
sinh
 Builtins, 284
slice_1d

Builtins, 301
slice_2d
 Builtins, 301, 302
slice_3d
 Builtins, 302, 303
slice_4d
 Builtins, 303, 304
slice_5d
 Builtins, 304, 305
slice_6d
 Builtins, 305, 306
sliding_sum
 Global constraints, 255
smallest, 90
 Annotations, 259
solution, 24
 all, 48
 end ‘=====’, 26
 separator ——, 25
solve, 89
sort
 Builtins, 306
 Global constraints, 240
sort_by
 Builtins, 307
span
 Global constraints, 249
sqrt
 Builtins, 281
strict_lex2
 Global constraints, 238
string, 22, 24
 literal, 24
 interpolated, 24
string_length
 Builtins, 313
subcircuit
 Global constraints, 255
sum, 43
 Builtins, 281
 Option type support, 269
sum_pred
 Global constraints, 255
symmetric_all_different
 Global constraints, 236
symmetry
 breaking, 58
symmetry_breaking_constraint
 Builtins, 321

T

table, 63
 Global constraints, 251
tan
 Builtins, 284
tanh
 Builtins, 284
tdistribution
 Builtins, 320
to_enum
 Builtins, 318
trace, 97
 Builtins, 317
trace_stdout
 Builtins, 317
true, 50
type, 22, 23, 70
 enumerated, 39, 48
 anonymous, 55
 non-finite, 78
 parameter, 22
type-inst, 23

U

ub
 Builtins, 315
ub_array
 Builtins, 315, 316
unfixed, 32
uniform
 Builtins, 321

V

value_precede
 Global constraints, 238
value_precede_chain
 Global constraints, 239
var_is_introduced
 Annotations, 256
variable, 22, 73
 bound, 77, 95, 96
 declaration, 23, 91
 enum, 49
 integer, 23
 iterator, 79
 local, 73, 76
 option type, 81

W

warm_start, 94
 Annotations, 263, 264

warm_start_array
 Annotations, 264
weibull
 Builtins, 321

X

xorall, 43
 Builtins, 287