MiniZinc Handbook

发行版本 *2.9.7*

Peter J. Stuckey, Kim Marriott, Guido Tack

2026 年 04 月 30 日

Contents

1 Overview **3**

1.1 Introduction . . . 5
1.1.1 Structure . . . 6
1.1.2 How to Read This . . . 6

1.2 Installation . . . 7
1.2.1 Microsoft Windows . . . 7
1.2.2 Linux . . . 8
1.2.2.1 Snap . . . 8
1.2.2.2 AppImage . . . 8
1.2.2.3 Archive . . . 9
1.2.3 Apple macOS . . . 9
1.2.4 Adding Third-party Solvers . . . 9

1.3 First steps with MiniZinc . . . 11
1.3.1 The MiniZinc IDE . . . 11
1.3.2 The MiniZinc command line tool . . . 16

2 Minizinc 指南 **19**

2.1 MiniZinc 基本模型 . . . 21
2.1.1 第一个实例 . . . 21
2.1.2 算术优化实例 . . . 25
2.1.3 数据文件和谓词 . . . 26
2.1.4 实数求解 . . . 28
2.1.5 模型的基本结构 . . . 30

2.2 更多复杂模型 . . . 33
2.2.1 数组和集合 . . . 33
2.2.2 全局约束 . . . 41
2.2.3 条件表达式 . . . 42
2.2.4 枚举类型 . . . 45
2.2.5 复杂约束 . . . 46
2.2.6 集合约束 . . . 51
2.2.7 汇总 . . . 53

2.3 谓词和函数 . . . 57
2.3.1 全局约束 . . . 57
2.3.1.1 Alldifferent . . . 57
2.3.1.2 Cumulative . . . 57
2.3.1.3 Table . . . 59
2.3.1.4 Regular . . . 60
2.3.2 定义谓词 . . . 63
2.3.3 定义函数 . . . 65
2.3.4 反射函数 . . . 66
2.3.5 局部变量 . . . 67
2.3.6 语境 . . . 68
2.3.7 局部约束 . . . 69
2.3.8 定义域反射函数 . . . 70
2.3.9 作用域 . . . 72

2.4 选项类型 . . . 73
2.4.1 声明和使用选项类型 . . . 73
2.4.2 隐藏选项类型 . . . 74

2.5 Tuple and record types . . . 77
2.5.1 Declaring and using tuples and records . . . 77
2.5.2 Using type-inst synonyms . . . 78
2.5.3 Types with both var and par members . . . 79
2.5.4 Example: rectangle packing using records . . . 80

2.6 搜索 . . . 83
2.6.1 有限域搜索 . . . 83
2.6.2 搜索注解 . . . 87
2.6.3 注解 . . . 88

2.7 MiniZinc 中的有效建模实践 . . . 91
2.7.1 变量界限 . . . 91
2.7.2 有效的生成元 . . . 92
2.7.3 冗余约束 . . . 94
2.7.4 模型选择 . . . 95
2.7.5 多重建模和连通 . . . 96
2.7.6 对称 . . . 97
2.7.6.1 静态的对称性破缺 . . . 98
2.7.6.2 其他对称的例子 . . . 99

2.8 在 MiniZinc 中对布尔可满足性问题建模 . . . 101
2.8.1 整型建模 . . . 101
2.8.2 非等式建模 . . . 102
2.8.3 势约束建模 . . . 103

2.9 FlatZinc 和展平 . . . 109
2.9.1 展平表达式 . . . 109
2.9.1.1 简化和求值 . . . 110
2.9.1.2 定义子表达式 . . . 110
2.9.1.3 FlatZinc 约束形式 . . . 110
2.9.1.4 边界分析 . . . 111
2.9.1.5 目标函数 . . . 111

2.9.2 线性表达式 . . . 111
2.9.3 展开表达式 . . . 112
2.9.4 数组 . . . 113
2.9.5 具体化 . . . 115
2.9.6 谓词 . . . 118
2.9.7 Let 表达式 . . . 119

3 User Manual **123**

3.1 The MiniZinc Command Line Tool . . . 125
3.1.1 Basic Usage . . . 125
3.1.2 Adding Solvers . . . 126
3.1.2.1 Configuration files . . . 126
3.1.2.2 Configuration for MIP solvers . . . 127
3.1.3 Options . . . 127
3.1.3.1 General options . . . 127
3.1.3.2 Solving options . . . 128
3.1.3.3 Flattener input options . . . 129
3.1.3.4 Solution output options . . . 132
3.1.4 User Configuration Files . . . 133
3.1.5 Command-line Parameter Files . . . 134

3.2 The MiniZinc IDE . . . 137
3.2.1 Editing files . . . 137
3.2.1.1 Editing functions . . . 138
3.2.1.2 Fonts and dark mode . . . 138
3.2.2 Configuring and Running a Solver . . . 138
3.2.2.1 Running a model . . . 138
3.2.2.2 Solver configurations . . . 139
3.2.2.3 Automatic Solution Checking . . . 141
3.2.2.4 Compiling a model . . . 142
3.2.2.5 Profiling a model . . . 142
3.2.3 Working With Projects . . . 143
3.2.4 Submitting Solutions to Online Courses . . . 143
3.2.5 Configuration Options . . . 144
3.2.5.1 Locating the MiniZinc installation . . . 144
3.2.5.2 Adding Third-Party Solvers . . . 147
3.2.5.3 Editing options . . . 150
3.2.5.4 Output options . . . 151

3.3 Visualising solutions in the MiniZinc IDE . . . 153
3.3.1 Visualisation library . . . 153
3.3.2 Pre-defined visualisations . . . 153
3.3.3 Custom visualisations . . . 154
3.3.3.1 Example visualisation . . . 156
3.3.3.2 MiniZincIDE JavaScript API . . . 160

3.4 CP-Profiler . . . 163
3.4.1 Using the profiler . . . 163
3.4.2 Traditional Tree Visualisation . . . 164
3.4.3 Alternative Search Tree Visualisations . . . 167

3.4.4 Similar Subtree Analysis . . . 171
3.4.5 Execution Comparison by Merging . . . 172
3.4.6 Replaying Search . . . 173
3.4.7 Nogood Analysis . . . 174
3.4.8 The protocol (high level) . . . 175
3.4.9 The protocol (low level) . . . 176

3.5 Solving Technologies and Solver Backends . . . 179
3.5.1 Constraint Programming Solvers . . . 179
3.5.1.1 Gecode . . . 179
3.5.1.2 Chuffed . . . 180
3.5.1.3 OR-Tools . . . 180
3.5.2 Mixed-Integer Programming Solvers . . . 180
3.5.2.1 MIP-Aware Modeling (But Mostly Useful for All Backends) . . . 180
3.5.2.2 Useful Flattening Parameters . . . 181
3.5.2.3 Some Solver Options and Changed Default Values . . . 182
3.5.2.4 Subtour Elimination Constraints . . . 183
3.5.2.5 Unified Domains (MIPdomains) . . . 183
3.5.2.6 Indicator Constraints . . . 183
3.5.2.7 Quadratic Constraints and Objectives (MIQCP) . . . 183
3.5.2.8 Pools of User Cuts and Lazy Constraints . . . 183
3.5.2.9 Warm Starts . . . 184
3.5.3 Non-Linear Solvers via NL File Format . . . 184

3.6 Automatic Solution Checking, Model Validation, and Benchmarking . . . 185
3.6.1 Autonomous Automatic Solution Checking and Benchmarking . . . 185
3.6.2 Model Validation: Automatic Solution Checking with a Checker Model 186
3.6.2.1 Basic checker models . . . 187
3.6.2.2 Detailed feedback . . . 188
3.6.2.3 Instance data in checker models . . . 190
3.6.2.4 Checking optimisation problems . . . 190
3.6.2.5 Hidden variables . . . 191

3.7 Globalizer . . . 195
3.7.1 Basic Usage . . . 195
3.7.2 Caveats . . . 197
3.7.3 Supported Constraints . . . 197
3.7.4 Using Globalizer in the MiniZinc IDE . . . 198
3.7.5 How it works . . . 199
3.7.6 Performance tips . . . 200
3.7.7 Limitations / Future work . . . 201

3.8 FindMUS . . . 203
3.8.1 Basic Usage . . . 203
3.8.1.1 Commandline arguments . . . 203
3.8.1.2 Example . . . 206
3.8.2 Using FindMUS in the MiniZinc IDE . . . 209
3.8.3 How it works . . . 212
3.8.4 Performance tips . . . 212
3.8.5 Limitations / Future work . . . 212

3.9 Mzn-Analyse . . . 215

3.9.1 Basic Usage . . . 215
3.9.1.1 Passes: read/write models & json . . . 215
3.9.1.2 Passes: Manipulate Models . . . 216
3.9.1.3 Passes: Special Passes . . . 216
3.9.2 Examples . . . 217
3.9.3 Limitations / Future work . . . 218

3.10 Using MiniZinc in Jupyter Notebooks . . . 219
3.10.1 Installation . . . 219
3.10.2 Basic usage . . . 219

3.11 Python Interface . . . 223

3.12 JavaScript Interface . . . 225

3.13 Installation from Source Code . . . 227

4 Reference Manual 229

4.1 Specification of MiniZinc . . . 231
4.1.1 Introduction . . . 231
4.1.1.1 Original authors. . . . 232
4.1.2 Notation . . . 232
4.1.3 Overview of a Model . . . 233
4.1.3.1 Evaluation Phases . . . 233
4.1.3.2 Run-time Outcomes . . . 234
4.1.3.3 Output . . . 234
4.1.4 Syntax Overview . . . 235
4.1.4.1 Character Set . . . 235
4.1.4.2 Comments . . . 235
4.1.4.3 Identifiers . . . 235
4.1.5 High-level Model Structure . . . 236
4.1.5.1 Model Instance Files . . . 237
4.1.5.2 Namespaces . . . 237
4.1.5.3 Scopes . . . 237
4.1.6 Types and Type-insts . . . 238
4.1.6.1 Properties of Types . . . 238
4.1.6.2 Instantiations . . . 238
4.1.6.3 Type-insts . . . 239
4.1.6.4 Type-inst expression overview . . . 239
4.1.6.5 Built-in Scalar Types and Type-insts . . . 240
4.1.6.6 Built-in Compound Types and Type-insts . . . 242
4.1.6.7 User-defined Types and Type-insts . . . 247
4.1.6.8 Constrained Type-insts . . . 247
4.1.7 Expressions . . . 248
4.1.7.1 Expressions Overview . . . 248
4.1.7.2 Operators . . . 250
4.1.7.3 Expression Atoms . . . 252
4.1.8 Items . . . 265
4.1.8.1 Include Items . . . 265
4.1.8.2 Variable Declaration Items . . . 266

4.1.8.3 Enum Items . . . 266
4.1.8.4 Type-inst Synonym Items . . . 269
4.1.8.5 Assignment Items . . . 269
4.1.8.6 Constraint Items . . . 270
4.1.8.7 Solve Items . . . 270
4.1.8.8 Output Items . . . 270
4.1.8.9 Annotation Items . . . 271
4.1.8.10 User-defined Operations . . . 271
4.1.9 Annotations . . . 275
4.1.10 Partiality . . . 276
4.1.10.1 Partial Assignments . . . 276
4.1.10.2 Partial Predicate/Function and Annotation Arguments . . . 277
4.1.10.3 Partial Array Accesses . . . 277
4.1.11 Built-in Operations . . . 278
4.1.11.1 Comparison Operations . . . 278
4.1.11.2 Arithmetic Operations . . . 278
4.1.11.3 Logical Operations . . . 280
4.1.11.4 Set Operations . . . 281
4.1.11.5 Array Operations . . . 282
4.1.11.6 Coercion Operations . . . 283
4.1.11.7 String Operations . . . 283
4.1.11.8 Bound and Domain Operations . . . 284
4.1.11.9 Option Type Operations . . . 285
4.1.11.10 Other Operations . . . 286
4.1.12 Content-types . . . 287
4.1.13 JSON support . . . 287
4.1.14 Full grammar . . . 288
4.1.14.1 Items . . . 288
4.1.14.2 Type-Inst Expressions . . . 290
4.1.14.3 Expressions . . . 291
4.1.14.4 Miscellaneous Elements . . . 294

4.2 The MiniZinc library . . . 295
4.2.1 Standard Library . . . 295
4.2.1.1 Built-in functions and operators . . . 295
4.2.1.2 Annotations . . . 341
4.2.1.3 Option type support . . . 356
4.2.1.4 Array sorting operations . . . 358
4.2.1.5 Language information . . . 359
4.2.1.6 Compiler options . . . 360
4.2.1.7 Conditionals . . . 362
4.2.1.8 Reflection operations . . . 363
4.2.1.9 Assertions and debugging functions . . . 366
4.2.1.10 Random Number Generator builtins . . . 370
4.2.1.11 Structured Output . . . 372
4.2.2 Global constraints . . . 372
4.2.2.1 All-Different and related constraints . . . 372
4.2.2.2 Lexicographic constraints . . . 374
4.2.2.3 Sorting constraints . . . 377
4.2.2.4 Channeling constraints . . . 379
4.2.2.5 Counting constraints . . . 380
4.2.2.6 Array-related constraints . . . 387

4.2.2.7 Set-related constraints . . . 388
4.2.2.8 Mathematical constraints . . . 389
4.2.2.9 Packing constraints . . . 393
4.2.2.10 Scheduling constraints . . . 399
4.2.2.11 Graph constraints . . . 402
4.2.2.12 Extensional constraints (table, regular etc.) . . . 414
4.2.2.13 Machine learning constraints . . . 418
4.2.2.14 Deprecated constraints . . . 419
4.2.3 FlatZinc builtins . . . 420
4.2.3.1 Integer FlatZinc builtins . . . 420
4.2.3.2 Bool FlatZinc builtins . . . 423
4.2.3.3 Set FlatZinc builtins . . . 425
4.2.3.4 Float FlatZinc builtins . . . 428
4.2.3.5 FlatZinc builtins added in MiniZinc 2.0.0. . . . 433
4.2.3.6 FlatZinc builtins added in MiniZinc 2.0.2. . . . 434
4.2.3.7 FlatZinc builtins added in MiniZinc 2.1.0. . . . 435
4.2.3.8 FlatZinc builtins added in MiniZinc 2.1.1. . . . 436
4.2.3.9 FlatZinc builtins added in MiniZinc 2.2.1. . . . 436
4.2.3.10 FlatZinc builtins added in MiniZinc 2.3.3. . . . 436
4.2.3.11 FlatZinc builtins added in MiniZinc 2.5.2. . . . 436
4.2.3.12 FlatZinc builtins added in MiniZinc 2.7.1. . . . 437
4.2.3.13 Deprecated FlatZinc builtins . . . 438
4.2.4 Additional declarations for Gecode . . . 438
4.2.4.1 Additional Gecode search annotations . . . 438
4.2.4.2 Additional Gecode constraints . . . 440
4.2.5 Additional declarations for Chuffed . . . 441
4.2.5.1 Additional Chuffed search annotations . . . 442
4.2.5.2 Other declarations . . . 442
4.2.6 MiniZincIDE tools . . . 443
4.2.6.1 MiniZincIDE solution visualisation tools . . . 443
4.2.7 MiniZinc definitions for the MiniZinc diversity extension. . . . 453
4.2.7.1 Annotations provided to guide the MiniZinc diversity extension. . . 453
4.2.7.2 Distance functions provided in the MiniZinc diversity extension. . 455
4.2.8 Experimental Features . . . 456
4.2.8.1 On Restart . . . 456

4.3 Interfacing Solvers to Flatzinc . . . 459
4.3.1 Specification of FlatZinc . . . 459
4.3.1.1 Comments . . . 460
4.3.1.2 Types . . . 460
4.3.1.3 Values and expressions . . . 462
4.3.1.4 FlatZinc models . . . 462
4.3.1.5 Predicate declarations . . . 463
4.3.1.6 Parameter declarations . . . 463
4.3.1.7 Variable declarations . . . 464
4.3.1.8 Constraints . . . 464
4.3.1.9 Solve item . . . 465
4.3.1.10 Annotations . . . 465
4.3.2 Specification of FlatZinc-JSON . . . 469
4.3.3 Output . . . 473
4.3.3.1 Solution output . . . 473
4.3.3.2 Statistics output . . . 476

4.3.3.3 Error and warning output . . . 477
4.3.4 Solver-specific Libraries . . . 477
4.3.4.1 Standard predicates . . . 478
4.3.4.2 Solver-specific predicates . . . 479
4.3.4.3 Reified and half-reified predicates . . . 479
4.3.5 Command-Line Interface and Standard Options . . . 480
4.3.6 Solver Configuration Files . . . 481
4.3.7 Grammar . . . 484

4.4 Machine-readable JSON output format . . . 487
4.4.1 Solution messages . . . 487
4.4.2 Checker messages . . . 488
4.4.3 Status messages . . . 488
4.4.4 Statistics messages . . . 489
4.4.5 Timestamp messages . . . 489
4.4.6 Comment messages . . . 489
4.4.7 Trace messages . . . 490
4.4.8 Profiling messages . . . 490
4.4.9 Paths messages . . . 490
4.4.10 Error messages . . . 491
4.4.10.1 Locations . . . 491
4.4.10.2 Stack traces . . . 492
4.4.11 Warning messages . . . 492

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: 节 1 covers installation and basic steps; 节 2 is a tutorial-style introduction into modelling with MiniZinc; 节 3 is a user manual for the individual tools in the MiniZinc tool chain; and 节 4 is a reference to the language.

This documentation is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International License[5]. 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.

[5] 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 125) is a user manual for the tools that make up the MiniZinc tool chain. Finally, *The fourth part* (page 231) 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.

列表 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 at https://github.com/MiniZinc/libminizinc/issues.

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*.

The easiest way to get a full, working MiniZinc system is to use the **bundled binary packages**, available from https://www.minizinc.org/downloads/.

The bundled binary packages contain the compiler and IDE, as well as the following solvers: Gecode, Chuffed, COIN-OR CBC, HiGHS, and OR-Tools CP-SAT. Interfaces to Gurobi, CPLEX, SCIP and XPress are also included and can be used if the corresponding solvers are installed.

Alternatively, it is possible to compile all components from source code, as described in *Installation from Source Code* (page 227).

1.2.1 Microsoft Windows

To install the bundled binary packages, simply download the installer, double-click to execute it, and follow the prompts. **Note:** 32 bit Windows installations are no longer supported in the bundled binary packages.

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 2.9.7 (bundled)\"
```

1.2.2 Linux

The MiniZinc bundled binary distribution for Linux is provided using three different distribution methods: *Snap*, *AppImage*, and an *archive* containing all necessary files. The capabilities of each package is the same, but it depends on your linux distribution which method is the easiest to use. **Note:** linux bundles are compiled to be compatible with many linux distributions using a *x86_64* architecture, but some dependencies on low level libraries cannot be avoided. Make use of the MiniZinc issue trackers if you encounter any problems.

1.2.2.1 Snap

The MiniZinc Snap is the easiest way to install the MiniZinc bundle if Snap (https://snapcraft.io/) is available on your system. The installation will provide the `minizinc` command within terminal environments and will add an application entry for the MiniZinc IDE. The IDE can be started from the command line using `minizinc.ide`. The following command will commence the installation the MiniZinc snap:

```
$ snap install minizinc --classic
```

Note: the MiniZinc snap requires the *classic* permission model to access solvers that are installed somewhere else on your system.

An advantage of using the MiniZinc snap is that it will be automatically updated whenever a new version of the snap is available. Using snap channels it is also possible to automatically follow the developmental versions of MiniZinc. More information about the MiniZinc snap can be found in the Snap Store.

1.2.2.2 AppImage

The MiniZinc AppImage provides a way of installing the MiniZinc bundles without any added requirements. AppImages (https://appimage.org/) provide self-contained versions of applications with all their required dependencies. To use the MiniZinc AppImage you simply download the AppImage, make it executable, and run:

```
$ chmod +x MiniZincIDE-2.9.7-x86_64.AppImage
$ ./MiniZincIDE-2.9.7-x86_64.AppImage
```

AppImages have no standard integration with the desktop environment. A script has been added to the MiniZinc AppImage to integrate the application into both the terminal and the desktop environment. The one requirement for this script is that the AppImage *CANNOT be moved* after the install script has been run. A suggestion would be to store the AppImages in `~/Applications/` or `/opt/Applications/`. The following commands move the AppImage to the second location and install `minizinc` and `MiniZincIDE` to `/usr/local/bin` and will add a desktop entry and its corresponding icon to `$HOME/.local/share/{applications,icons}`:

```
$ mv ./MiniZincIDE-2.9.7-x86_64.AppImage /opt/Applications/
$ /opt/Applications/MiniZincIDE-2.9.7-x86_64.AppImage install
```

Note: The installation locations can be changed using the environmental variables `$BIN_LOCATION` and `$DESKTOP_LOCATION`. The default behaviour can be achieved using the following command:

```
$ BIN_LOCATION=/usr/local/bin DESKTOP_LOCATION=$HOME/.local/share ./MiniZincIDE-2.
↪9.7-x86_64.AppImage install
```

1.2.2.3 Archive

The MiniZinc archive bundle is straightforward, but doesn't offer any automatic integrations with the desktop environment. After downloading, uncompress the archive, for example in your home directory or any other location where you want to install it:

```
$ tar xf MiniZincIDE-2.9.7-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 start the MiniZinc IDE using the `MiniZincIDE.sh` shell script, which sets up a number of paths and environment variables.

In order to use the command line tools, after unpacking, make the following changes to some environment variables (assuming a bash-like shell):

```
$ export PATH=MiniZincIDE-2.9.7-bundle-linux-x86_64/bin:$PATH
$ export LD_LIBRARY_PATH=MiniZincIDE-2.9.7-bundle-linux-x86_64/lib:$LD_LIBRARY_
↪PATH
$ export QT_PLUGIN_PATH=MiniZincIDE-2.9.7-bundle-linux-x86_64/plugins:$QT_PLUGIN_
↪PATH
```

1.2.3 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
```

1.2.4 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 节 3.2.5.2. You can also add configuration files manually, as explained in 节 4.3.6.

CHAPTER 1.3

First steps with MiniZinc

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

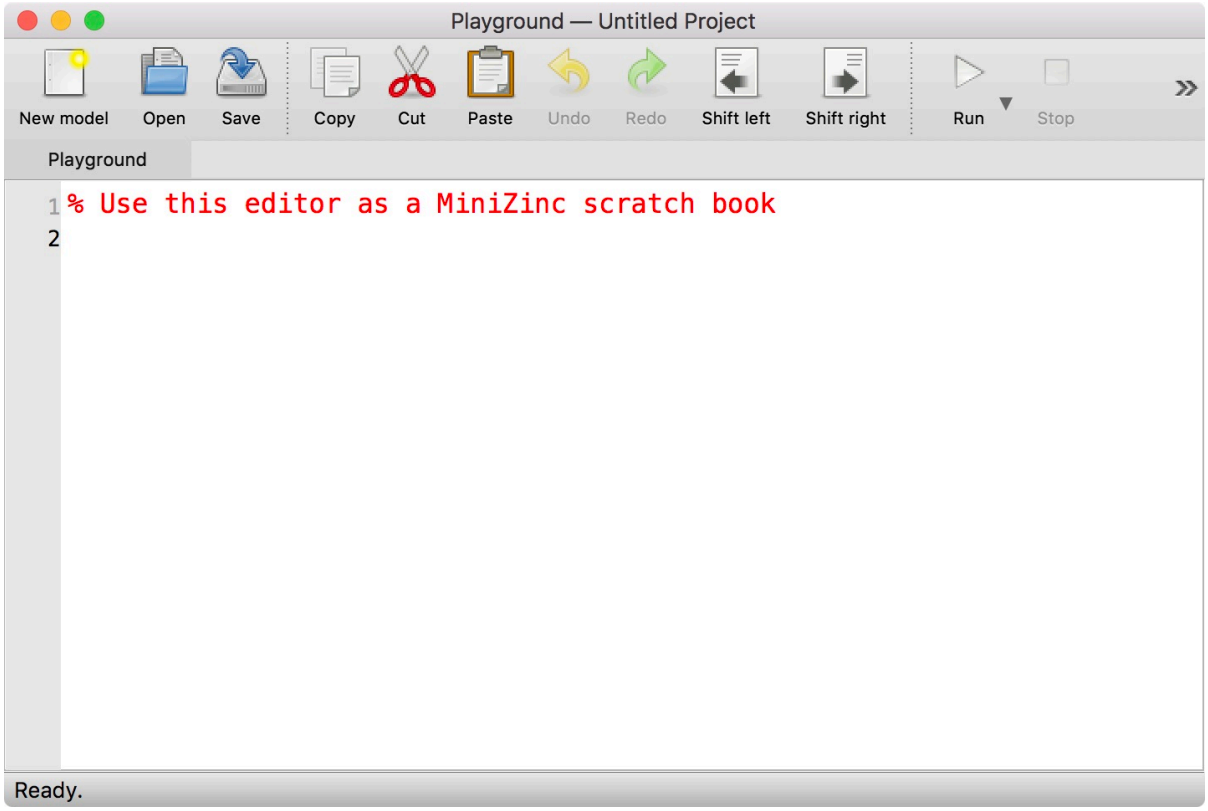
This section introduces the MiniZinc IDE and the command line tool `minizinc` using some very basic examples. This should be enough to get you started with the MiniZinc tutorial in 节 2 (in fact, you only need to be able to use one of the two, for instance just stick to the IDE if you are not comfortable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

1.3.1 The MiniZinc IDE

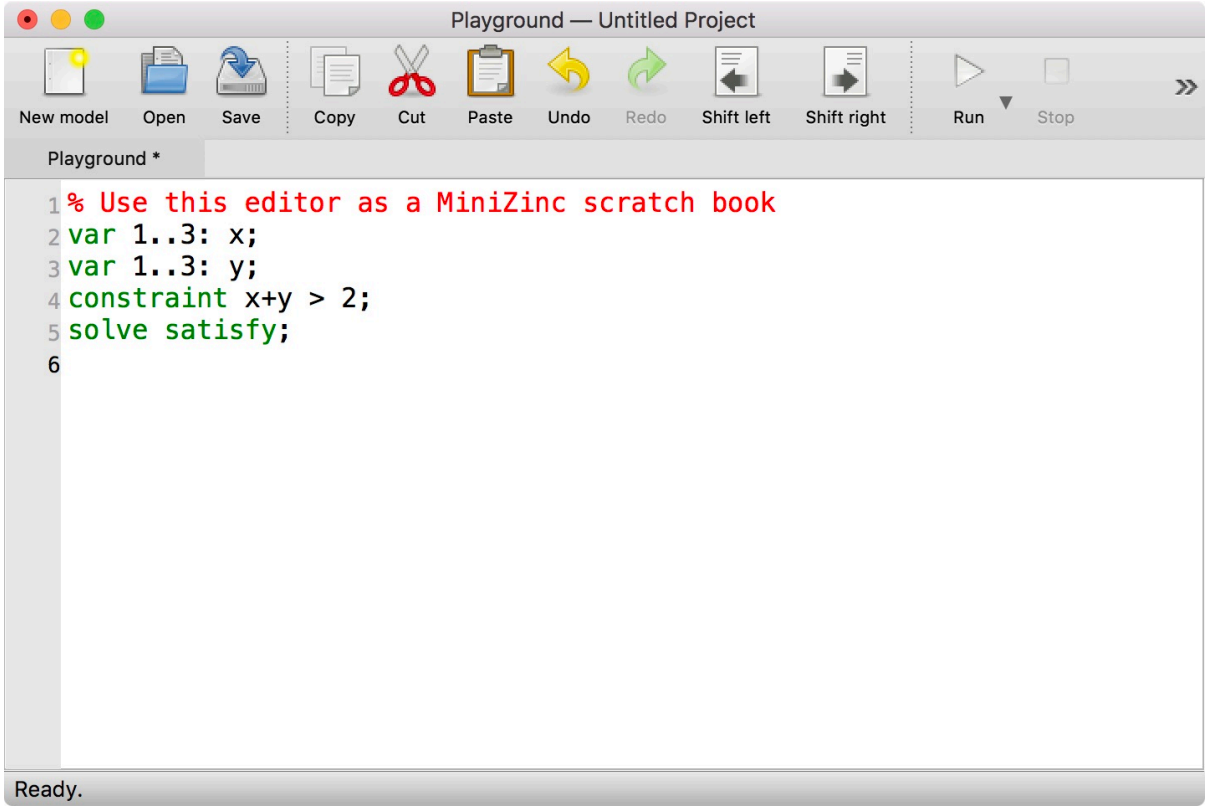
The MiniZinc IDE provides a simple interface to most of MiniZinc's functionality. It lets you edit model and data files, solve them with any of the solvers supported by MiniZinc, run debugging and profiling tools, and submit solutions to online courses (such as the MiniZinc Coursera courses).

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 节 3.2.5 for more details on this.

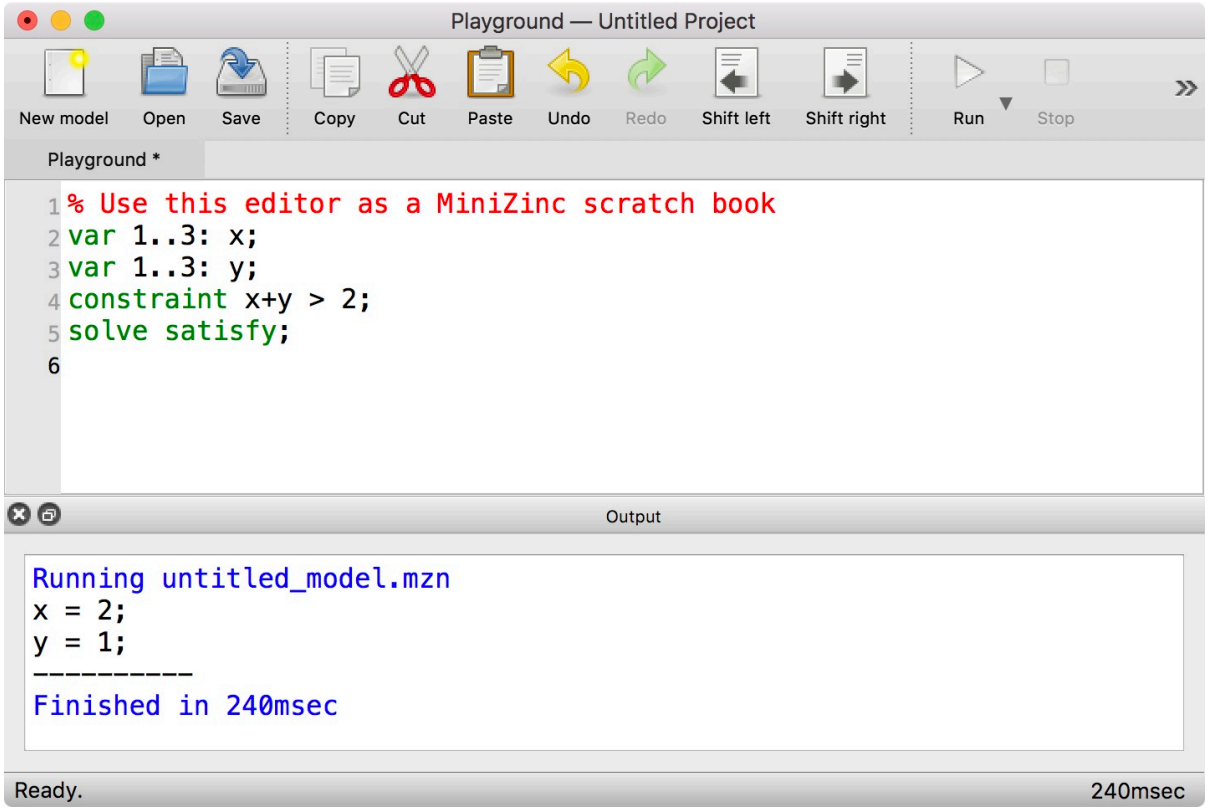
The IDE will then greet you with the *MiniZinc Playground*, a window that will look like this:



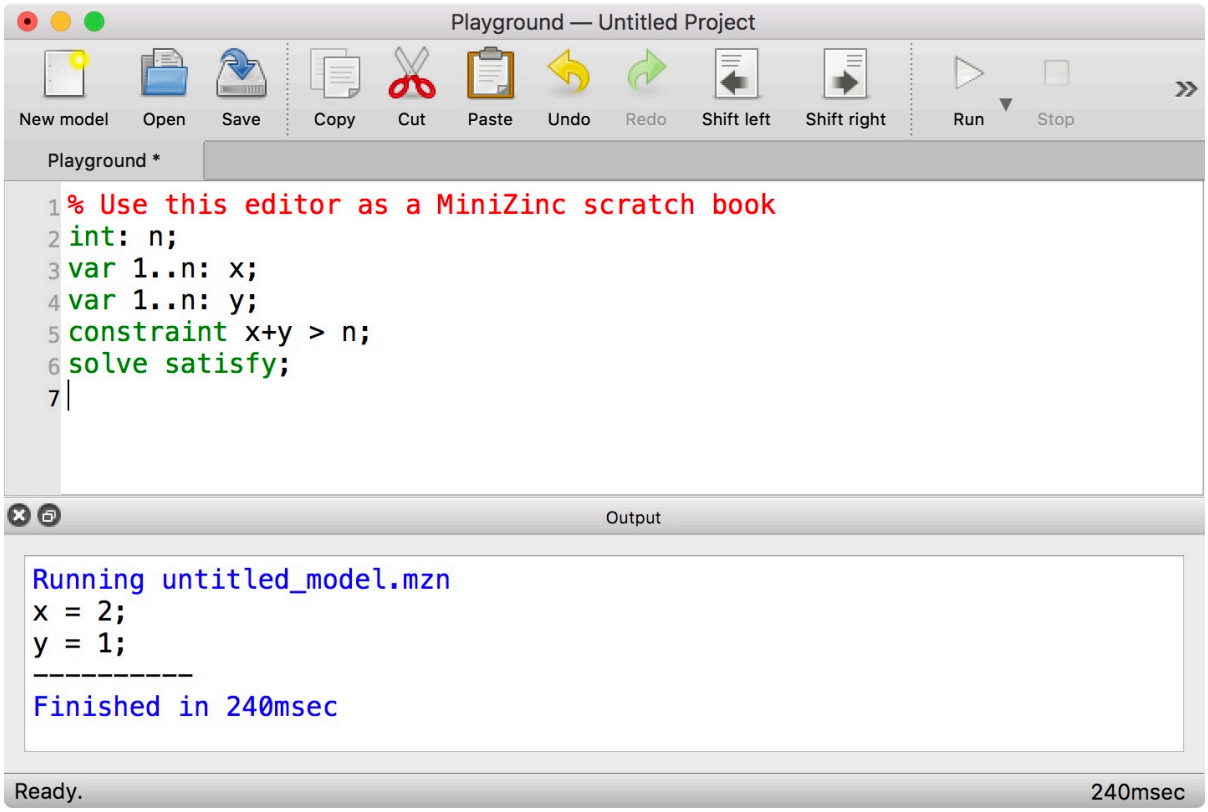
You can start writing your first MiniZinc model! Let's try something very simple:



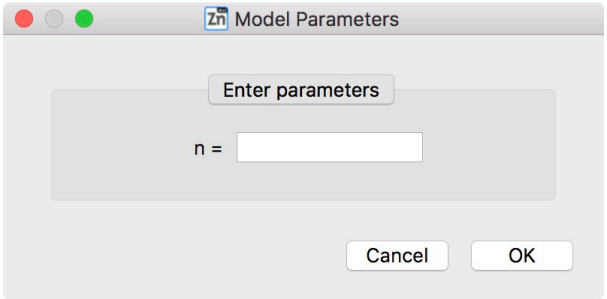
In order to solve the model, you click on the *Run* button in the toolbar, or use the keyboard shortcut *Ctrl+R* (or *command+R* on macOS):



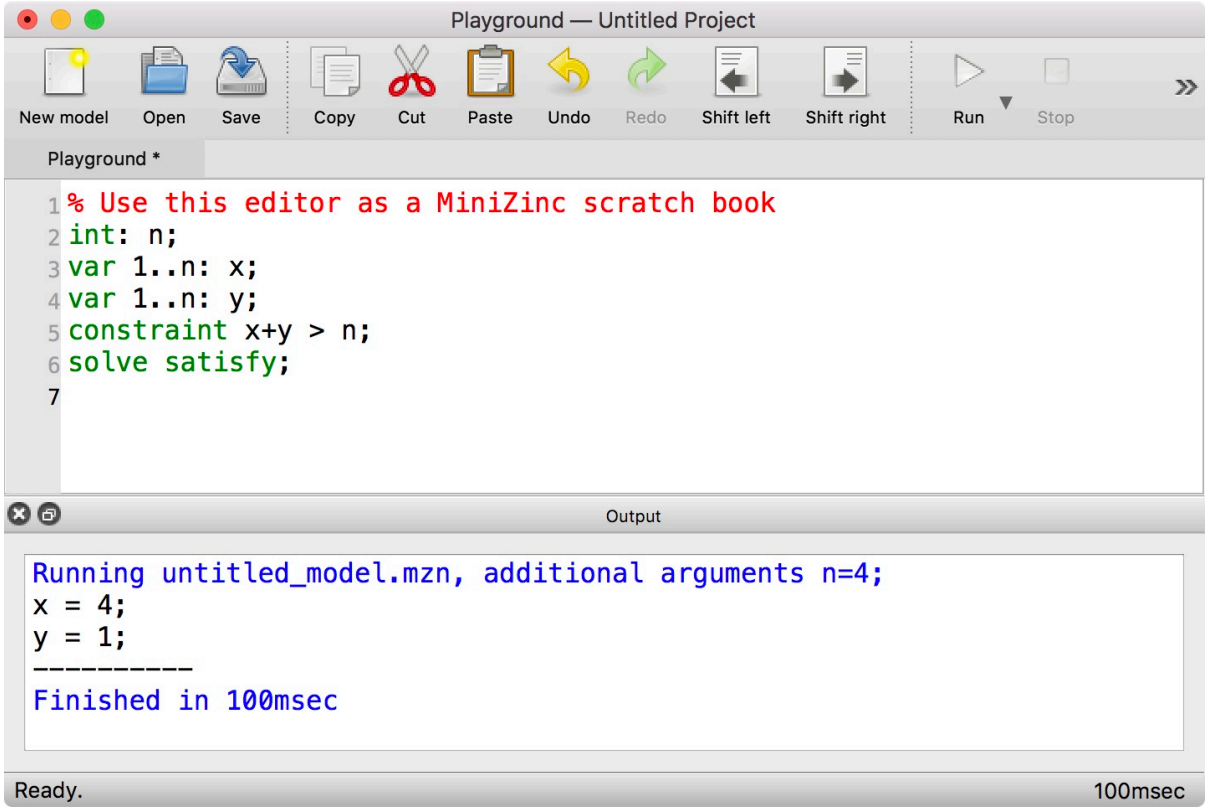
As you can see, an output window pops up that displays a solution to the problem you entered. Let us now try a model that requires some additional data.



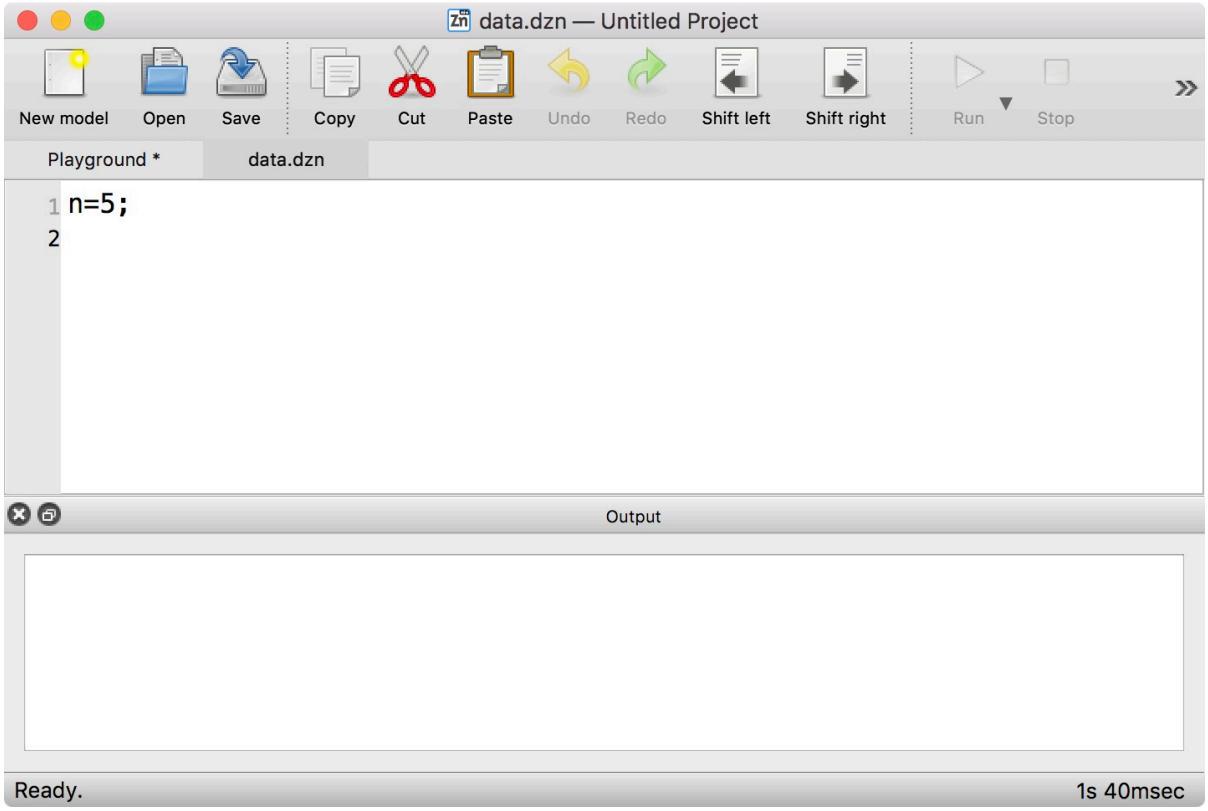
When you run this model, the IDE will ask you to enter a value for the parameter *n*:



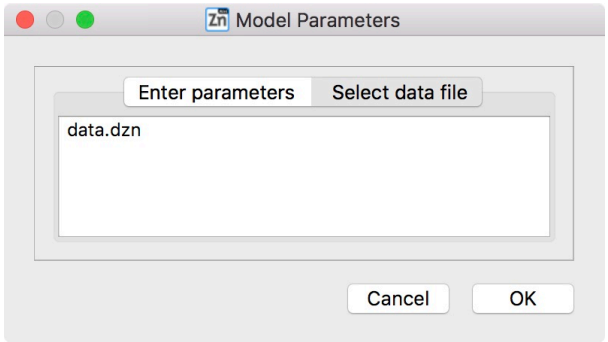
After entering, for example, the value 4 and clicking *Ok*, the solver will execute the model for *n=4*:



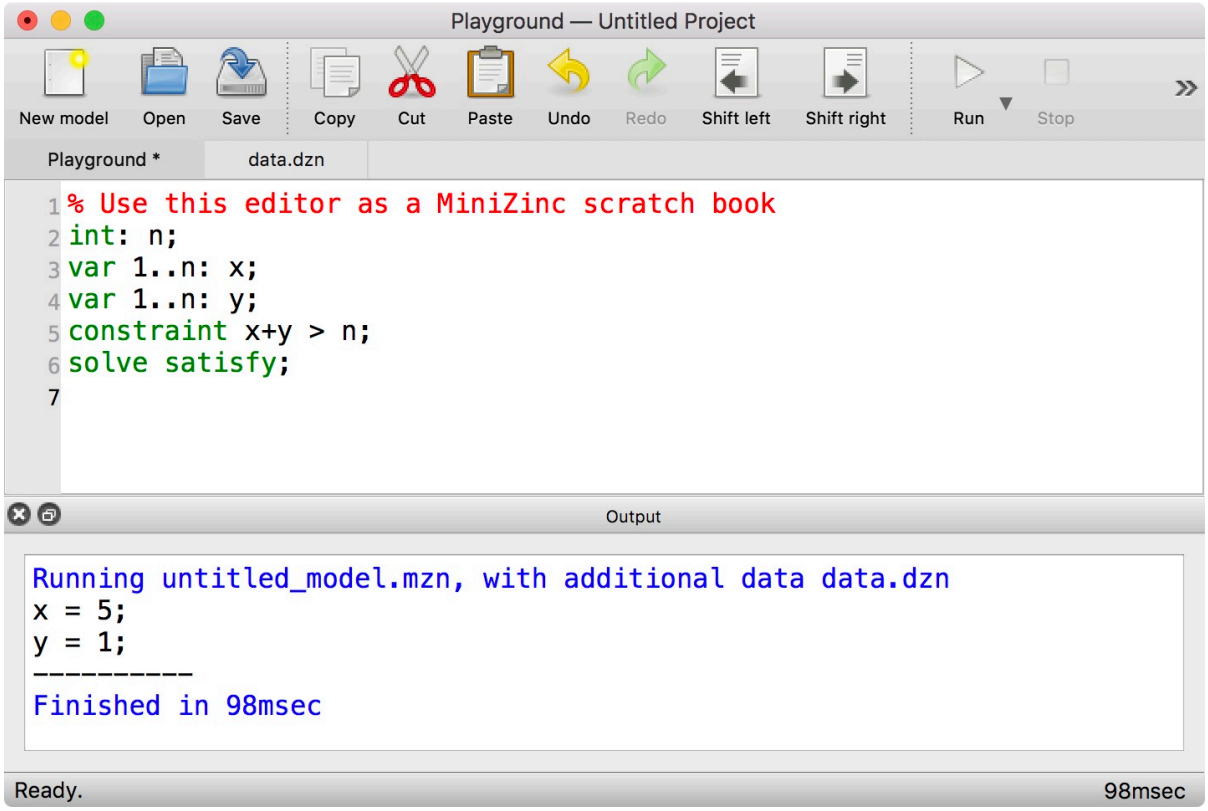
Alternatively, data can also come from a file. Let's create a new file with the data and save it as `data.dzn`:



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 节 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 节 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 节 3.1.

Part 2

Minizinc 指南

CHAPTER 2.1

MiniZinc 基本模型

在此节中，我们利用两个简单的例子来介绍一个 MiniZinc 模型的基本结构。

2.1.1 第一个实例

图 1.1: 澳大利亚各州

作为我们的第一个例子，假设我们要去给 图 1.1 中的澳大利亚地图涂色。它包含了七个不同的州和地区，而每一块都要被涂一个颜色来保证相邻的区域有不同的颜色。

列表 1.1: 一个用来给澳大利亚的州和地区涂色的 MiniZinc 模型 aust.mzn

```
% 用nc个颜色来涂澳大利亚
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"];
```

我们可以很容易的用 MiniZinc 给此问题建模。此模型在 列表 1.1 中给出。

模型中的第一行是注释。注释开始于 % 来表明此行剩下的部分是注释。MiniZinc 同时也含有 C 语言风格的由 /* 开始和 */ 结束的块注释。

模型中的下一部分声明了模型中的变量。此行

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

定义了一个问题的 参数来代表可用的颜色个数。在很多编程语言中，参数和变量是类似的。它们必须被声明并且指定一个类型 type。在此例中，类型是 int。通过 赋值，它们被设了值。MiniZinc 允许变量声明时被赋值（就像上面那一行）或者单独给出一个赋值语句。因此下面的表示跟上面的只一行表示是相等的

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

和很多编程语言中的变量不一样的是，这里的参数只可以被赋 唯一的值。如果一个参数出现在了多于一个的赋值中，就会出现错误。

基本的 参数类型包括 整型 (int)，浮点型 (float)，布尔型 (bool) 以及 字符串型 (string)。同时也支持数组和集合。

MiniZinc 模型同时也可能包含另一种类型的变量 决策变量。决策变量是数学的或者逻辑的变量。和一般的编程语言中的参数和变量不同，建模者不需要给决策变量一个值。而是在开始时，一个决策变量的值是不知道的。只有当 MiniZinc 模型被执行的时候，求解系统才来决定决策变量是否可以被赋值从而满足模型中的约束。若满足，则被赋值。

在我们的模型例子中，我们给每一个区域一个 决策变量 wa, nt, sa, q, nsw, v 和 t。它们代表了会被用来填充区域的（未知）颜色。

对于每一个决策变量，我们需要给出变量可能的取值集合。这个被称为变量的 定义域。定义域部分可以在 变量声明的时候同时给出，这时决策变量的 类型就会从定义域中的数值的类型推断出。

MiniZinc 中的决策变量的类型可以为布尔型，整型，浮点型或者集合。同时也可以是元素为决策变量的数组。在我们的 MiniZinc 模型例子中，我们使用整型去给不用的颜色建模。通过使用 `var` 声明，我们的每一个决策变量被声明为定义域为一个整数类型的范围表示 `1..nc` ，来表明集合 $\{1, 2, \ldots, nc\}$ 。所有数值的类型为整型，所以模型中的所有的变量是整型决策变量。

标识符

用来命名参数和变量的标识符是一列由大小写字母，数字以及下划线 _ 字符组成的字符串。它们必须开始于一个字母字符。因此 `myName_2` 是一个有效的标识符。MiniZinc（和 Zinc）的 关键字不允许被用为标识符名字。它们在*Identifiers* (page 235) 中被列出。所有的 MiniZinc 操作符都不能被用做标识符名字。它们在*Operators* (page 250) 中被列出。

MiniZinc 仔细地区别了以下两种模型变量：参数和决策变量。利用决策变量创建的表达式类型比利用参数可以创建的表达式类型更局限。但是，在任何可以用决策变量的地方，同类型的参数变量也可以被应用。

整型变量声明

一个 整型参数变量可以被声明为以下两种方式：

```
int : <变量名>
<l> .. <u> : <变量名>
```

`<l>` 和 `<u>` 是固定的整型表达式。
一个整型决策变量被声明为以下两种方式：

```
var int : <变量名>
var <l>..<u> : <变量名>
```

`<l>` 和 `<u>` 是固定的整型表达式。

参数和决策变量形式上的区别在于对变量的 实例化。变量的实例化和类型的结合被叫为 类型-实例化。既然你已经开始使用 MiniZinc，毫无疑问的你会看到很多 类型-实例化的错误例子。

模型的下一部分是 约束。它们详细说明了决策变量想要组成一个模型的有效解必须要满足的布尔型表达式。在这个例子中我们有一些决策变量之间的不等式。它们规定如果两个区域是相邻的，则它们必须有不同的颜色。

关系操作符

MiniZinc 提供了以下关系操作符 关系操作符：
相等 (= or ==), 不等 (!=), 小于 (<), 大于 (>), 小于等于 (<=), and 和大于等于 (>=).

模型中的下一行：

```
solve satisfy;
```

表明了它是什么类型的问题。在这个例子中，它是一个 满足问题：我们希望给决策变量找到一个值使得约束被满足，但具体是哪一个值却没有所谓。

模型的最后一个部分是 输出语句。它告诉 MiniZinc 当模型被运行并且找到一个解 解的时候，要输出什么。

输出和字符串

一个输出语句跟着一 串字符。它们通常或者是写在双引号之间的字符串常量 字符串常量并且对特殊字符用类似 C 语言的标记法，或者是 `show(e)` 格式的表达式，其中 `e` 是 MiniZinc 表达式。例子中的 `\n` 代表换行符，`\t` 代表制表符。
数字的 `show` 有各种不同方式的表示：`show_int(n,X)` 在至少 $|n|$ 个字符里输出整型 `X` 的值，若 $n > 0$ 则右对齐，否则则左对齐；`show_float(n,d,X)` 在至少 $|n|$ 个字符里输出浮点型 `X` 的值，若 $n > 0$ 则右对齐，否则则左对齐，并且小数点后有 d 个字符。
字符串常量必须在同一行中。长的字符串常量可以利用字符串连接符 `++` 来分成几行。例如，字符串常量

```
"Invalid datafile: Amount of flour must be non-negative"
```

和字符串常量表达式

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

是相等的。
MiniZinc 支持内插字符串 内插字符串。表达式可以直接插入字符串常量中。`"\(e)"` 形式的子字符串会被替代为 `show(e)` 。例如，`"t=\(t)\n"` 产生和 `"t=" ++ show(t) ++ "\n"` 一样的字符串。
一个模型可以包含多个输出语句。在这种情况下，所有输出会根据它们在模型中出现的顺序连接。

我们可以通过点击 MiniZinc IDE 中的 *Run* 按钮，或者输入

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

来评估我们的模型。其中 `aust.mzn` 是包含我们的 MiniZinc 模型的文件名字。我们必须使用文件扩展名 `.mzn` 来表明一个 MiniZinc 模型。带有 `--solver Gecode` 选项的命令 `minizinc` 使用 Gecode 有限域求解器去评估我们的模型。如果你使用的是 MiniZinc 二进制发布，这个求解器实际上是预设的，所以你也可以运行 `minizinc aust.mzn` 。

当我们运行上面的命令后，我们得到如下的结果：

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

10 个破折号 `----------` 这行是自动被 MiniZinc 输出的，用来表明一个解已经被找到。

2.1.2 算术优化实例

我们的第二个例子来自于要为了本地的校园游乐会烤一些蛋糕的需求。我们知道如何制作两种蛋糕。footnote{警告: 请不要在家里使用这些配方制作} 一个香蕉蛋糕的制作需要 250 克自发酵的面粉，2 个捣碎的香蕉，75 克糖和 100 克黄油。一个巧克力蛋糕的制作需要 200 克自发酵的面粉，75 克可可粉，150 克糖和 150 克黄油。一个巧克力蛋糕可以卖 $4.50，一个香蕉蛋糕可以卖 $4.00。我们一共有 4 千克的自发酵面粉，6 个香蕉，2 千克的糖，500 克的黄油和 500 克的可可粉。问题是对每一种类型的蛋糕，我们需要烤多少给游乐会来得到最大的利润。一个可能的 MiniZinc 模型在 列表 1.2 中给出。

列表 1.2: 决定为了校园游乐会要烤多少香蕉和巧克力蛋糕的模型。(cakes.mzn)

```
% 为校园游乐会做蛋糕

var 0..100: b; % 香蕉蛋糕的个数
var 0..100: c; % 巧克力蛋糕的个数

% 面粉
constraint 250*b + 200*c <= 4000;
% 香蕉
constraint 2*b  <= 6;
% 糖
constraint 75*b + 150*c <= 2000;
% 黄油
constraint 100*b + 150*c <= 500;
% 可可粉
constraint 75*c <= 500;

% 最大化我们的利润
solve maximize 400*b + 450*c;

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

第一个新特征是 *arithmetic expressions* 的使用。

整数算术操作符

MiniZinc 提供了标准的整数算术操作符。加 (+), 减 (-), 乘 (*), 整数除 (div) 和整数模 (mod). 同时也提供了一元操作符 + 和 - 。
整数模被定义为输出和被除数 a 一样正负的 a mod b 值。整数除被定义为使得 $a = b$“‘$*$‘ (a div b) + (a mod b) 值。
MiniZinc 提供了标准的整数函数用来取绝对值 (abs) 和幂函数 (pow). 例如 abs(-4) 和 pow(2,5) 分别求得数值 4 和 32 。
算术常量的语法是相当标准的。整数常量可以是十进制，十六进制或者八进制。例如 0, 5, 123, 0x1b7, 0o777 。

例子中的第二个新特征是优化。这行

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

指出我们想找一个可以使 solve 语句中的表达式（我们叫做 目标）最大化的解。这个目标可以为任何类型的算术表达式。我们可以把关键字 maximize 换为 minimize 来表明一个最小化问题。

当我们运行上面这个模型时，我们得到以下的结果：

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

一旦系统证明了一个解是最优解，这一行 ========== 在最优化问题中会自动被输出。

2.1.3 数据文件和谓词

此模型的一个缺点是如果下一次我们希望解决一个相似的问题，即我们需要为学校烤蛋糕（这是经常发生的），我们需要改变模型中的约束来表明食品间拥有的原料数量。如果我们想重新利用此模型，我们最好使得每个原料的数量作为模型的参数，然后在模型的最上层设置它们的值。

更好的办法是在一个单独的 *数据文件* 中设置这些参数的值。MiniZinc（就像很多其他的建模语言一样）允许使用数据文件来设置在原始模型中声明的参数的值。通过运行不同的数据文件，使得同样的模型可以很容易地和不同的数据一起使用。

数据文件的文件扩展名必须是 `.dzn` ，来表明它是一个 MiniZinc 数据文件。一个模型可以被任何多个数据文件运行（但是每个变量/参数在每个文件中只能被赋一个值）

列表 1.3: 独立于数据的用来决定为了校园游乐会要烤多少香蕉和巧克力蛋糕的模型。(cakes2.mzn)

```
% 为校园游乐会做蛋糕（和数据文件一起）

int: flour;  %拥有的面粉克数
int: banana; %拥有的香蕉个数
int: sugar;  %拥有的糖克数
int: butter; %拥有的黄油克数
int: cocoa;  %拥有的可可粉克数

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; % 香蕉蛋糕的个数
var 0..100: c; % 巧克力蛋糕的个数

% 面粉
constraint 250*b + 200*c <= flour;
% 香蕉
constraint 2*b  <= banana;
% 糖
constraint 75*b + 150*c <= sugar;
% 黄油
constraint 100*b + 150*c <= butter;
```

```
% 可可粉
constraint 75*c <= cocoa;

% 最大化我们的利润
solve maximize 400*b + 450*c;

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

我们的新模型在 列表 1.3 中给出。我们可以用下面的命令来运行

数据文件 pantry.dzn 在 列表 1.4 中给出。我们得到和 cakes.mzn 同样的结果。运行下面的命令

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

利用另外一个 列表 1.5 中定义的数据集，我们得到如下结果

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

如果我们从 cakes.mzn 中去掉输出语句，MiniZinc 会使用默认的输出。这种情况下得到的输出是

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

默认输出

一个没有输出语句的 MiniZinc 模型会给每一个决策变量以及它的值一个输出行，除非决策变量已经在声明的时候被赋了一个表达式。注意观察此输出是如何呈现一个正确的数据文件格式的。

列表 1.4: cakes2.mzn 的数据文件例子 (pantry.dzn)

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

列表 1.5: cakes2.mzn 的数据文件例子 (pantry2.dzn)

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

通过使用 命令行标示 `-D` *string* ，小的数据文件可以被直接输入而不是必须要创建一个 `.dzn` 文件，其中 *string* 是数据文件里面的内容。

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

会给出和

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

一模一样的结果。

数据文件只能包含给模型中的决策变量和参数赋值的语句。

防御性编程建议我们应该检查数据文件中的数值是否合理。在我们的例子中，检查所有原料的份量是否是非负的并且若不正确则产生一个运行错误，这是明智的。MiniZinc 提供了一个内置的布尔型操作符 断言用来检查参数值。格式是 `assert(B,S)` 。布尔型表达式 `B` 被检测。若它是假的，运行中断。此时字符串表达式 `S` 作为错误信息被输出。如果我们想当面粉的份量是负值的时候去检测出并且产生合适的错误信息，我们可以直接加入下面的一行

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

到我们的模型中。注意 断言表达式是一个布尔型表达式，所以它被看做是一种类型的约束。我们可以加入类似的行来检测其他原料的份量是否是非负值。

2.1.4 实数求解

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 列表 1.6. It uses a simple interest calculation to calculate the balance after each quarter.

通过使用浮点数求解，MiniZinc 也支持“实数”约束求解。考虑一个要在 4 季度分期偿还的一年短期贷款问题。此问题的一个模型在 列表 1.6 中给出。它使用了一个简单的计算每季度结束后所欠款的利息计算方式。

列表 1.6: 确定一年借款每季度还款关系的模型 (`loan.mzn`)

```
% 变量
var float: R;        % 季度还款
var float: P;        % 初始借贷本金
var 0.0 .. 10.0: I;  % 利率

% 中间变量
var float: B1; % 一个季度后的欠款
var float: B2; % 两个季度后的欠款
var float: B3; % 三个季度后的欠款
var float: B4; % 最后欠款

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"
];
```

注意我们声明了一个浮点型变量 `f`，它和整型变量很类似。只是这里我们使用关键字 `float` 而不是 `int`。

我们可以用同样的模型来回答一系列不同的问题。第一个问题是：如果我以利息 4% 借款 $1000 并且每季度还款 $260，我最终还欠款多少？这个问题在数据文件 `loan1.dzn` 中被编码。

由于我们希望用实数求解，我们需要使用一个可以支持这种问题类型的求解器。Gecode(Minizinc 捆绑二进制发布预设的求解器) 支持浮点型变量，一个混合整数线性求解器可能更加适合这种类型的问题。MiniZinc 发布包含了这样的一个求解器。我们可以通过从求解器 IDE 菜单 (*Run* 按钮下面的三角形) 选择 `COIN-BC` 来使用，或者在命令行中运行命令 `minizinc --solver cbc`：

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

输出是

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

第二个问题是如果我希望用 4% 的利息来借款 $1000 并且在最后的时候一点都不欠款，我需要在每季度还款多少？这个问题在数据文件 `loan2.dzn` 中被编码。运行命令

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

后的输出是

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

第三个问题是如果我可以每个季度返还 $250，我可以用 4% 的利息来借款多少并且在最后的时候一点都不欠款？这个问题在数据文件 `loan3.dzn` 中被编码。运行命令

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

后的输出是

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

列表 1.7: loan.mzn 的数据文件例子 (loan1.dzn)

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

列表 1.8: loan.mzn 的数据文件例子 (loan2.dzn)

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

列表 1.9: loan.mzn 的数据文件例子 (loan3.dzn)

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

浮点算术操作符

MiniZinc 提供了标准的浮点算术操作符: 加 (+), 减 (-), 乘 (*) 和浮点除 (/)。同时也提供了一元操作符 + 和 - 。
MiniZinc 不会自动地强制转换整数为浮点数。内建函数 `int2float` 被用来达到此目的。注意强制转换的一个后果是表达式 `a / b` 总是被认为是一个浮点除。如果你需要一个整数除, 请确定使用 `div` 操作符。
MiniZinc 同时也包含浮点型函数来计算: 绝对值 (abs), 平方根 (sqrt), 自然对数 (ln), 底数为 2 的对数 (log2), 底数为 10 的对数 (log10), e 的幂 (exp), 正弦 (sin), 余弦 (cos), 正切 (tan), 反正弦 (asin), 反余弦 (acos), 反正切 (atan), 双曲正弦 (sinh), 双曲余弦 (cosh), 双曲正切 (tanh), 双曲反正弦 (asinh), 双曲反余弦 (acosh), 双曲反正切 (atanh), 和唯一的二元函数次方 (pow)，其余的都是一元函数。
算术常量的语法是相当标准的。浮点数常量的例子有 `1.05`，`1.3e-5` 和 `1.3E+5` 。

2.1.5 模型的基本结构

我们现在可以去总结 MiniZinc 模型的基本结构了。它由多个项组成，每一个在其最后都有一个分号 ; 。项可以按照任何顺序出现。例如，标识符在被使用之前不需要被声明。

有八种类型的项 items 。

- 引用项允许另外一个文件的内容被插入模型中。它们有以下形式:

  ```
  include <文件名>;
  ```

  其中 <文件名> 是一个字符串常量。它们使得大的模型可以被分为小的子模型以及包含库文件中定义的约束。我们会在 列表 2.4 中看到一个例子。

- 变量声明声明新的变量。这种变量是全局变量，可以在模型中的任何地方被提到。变量有两种。在模型中被赋一个固定值的参数变量以及只有在模型被求解的时候才会被赋值的决策变量。我们称参数是 固定的，决策变量是 不固定的。变量可以选择性地被赋一个值来作为声明的一部分。形式是:

```
<类型-实例化 表达式>: <变量> [ = ] <表达式>;
```

<类型-实例化 表达式> 给了变量的类型和实例化。这些是 MiniZinc 比较复杂的其中一面。用 par 来实例化声明参数，用 var 来实例化声明决策变量。如果没有明确的实例化声明，则变量是一个参数。类型可以为基类型，一个 整数或者浮点数范围，或者一个数组或集合。基类型有 float, int, string, bool, ann 。其中只有 float, int and bool 可以被决策变量使用。基类型 ann 是一个 注解 -- 我们会在搜索 (page 83) 中讨论注解。整数范围表达式可以被用来代替类型 int. 类似的，浮点数范围表达式可以被用来代替类型 float 。这些通常被用来定义一个整型决策变量的定义域，但也可以被用来限制一个整型参数的范围。变量声明的另外一个用处是定义 枚举类型, ---我们会在枚举类型 (page 45) 中讨论。

- 赋值项给一个变量赋一个值。它们有以下形式：

```
<变量> = <表达式>;
```

数值可以被赋给决策变量。在这种情况下，赋值相当于加入 constraint <变量> = <表达式> ;

- 约束项是模型的核心。它们有以下形式：

```
constraint <布尔型表达式>;
```

我们已经看到了使用算术比较的简单约束以及内建函数 assert 操作符。在下一节我们会看到更加复杂的约束例子。

- 求解项详细说明了到底要找哪种类型的解。正如我们看到的，它们有以下三种形式：

```
solve satisfy;
solve maximize <算术表达式>;
solve minimize <算术表达式>;
```

一个模型必须有且只有一个求解项。

- 输出项用来恰当的呈现模型运行后的结果。它们有下面的形式：

```
output [ <字符串表达式>, ..., <字符串表达式> ];
```

如果没有输出项，MiniZinc 会默认输出所有没有被以赋值项的形式赋值的决策变量值。

- 枚举类型声明. 我们会在数组和集合 (page 33) 和枚举类型 (page 45) 中讨论。
- 谓词函数和测试项被用来定义新的约束，函数和布尔测试。我们会在谓词和函数 (page 57) 中讨论。
- 注解项用来定义一个新的注解。我们会在搜索 (page 83) 中讨论。

CHAPTER 2.2

更多复杂模型

在上一节中，我们介绍了 MiniZinc 模型的基本结构。在这一节中，我们介绍数组和集合数据结构，枚举类型，以及更加复杂的约束。

2.2.1 数组和集合

在绝大多数情况下，我们都是有兴趣建一个约束和变量的个数依赖于输入数据的模型。为了达到此目的，我们通常会使用 数组。

考虑一个关于金属矩形板温度的简单有限元素模型。通过把矩形板在 2 维的矩阵上分成有限个的元素，我们近似计算矩形板上的温度。一个模型在 列表 2.1 中给出。它声明了有限元素模型的宽 w 和高 h 。

声明

```
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; % 在点 (i,j) 处的温度
```

声明了四个固定的整型集合来描述有限元素模型的尺寸：HEIGHT 是整个模型的整体高度，而 CHEIGHT 是省略了顶部和底部的中心高度，WIDTH 是模型的整体宽度，而 CWIDTH 是省略了左侧和右侧的中心宽度。最后，声明了一个浮点型变量组成的行编号从 0 到 w ，列编号从 0 到 h 的两维数组 t 用来表示金属板上每一点的温度。我们可以用表达式 t[i,j] 来得到数组中第 i^{th} 行和第 j^{th} 列的元素。

拉普拉斯方程规定当金属板达到一个稳定状态时，每一个内部点的温度是它的正交相邻点的平均值。约束

```
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]);
```

保证了每一个内部点 (i, j) 是它的四个正交相邻点的平均值。约束

```
% 边约束
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);
```

限制了每一个边的温度必须是相等的，并且给了这些温度名字：`left` ，`right` ，`top` 和 `bottom`。而约束

```
% 角约束
constraint t[0,0]=0.0;
constraint t[0,w]=0.0;
constraint t[h,0]=0.0;
constraint t[h,w]=0.0;
```

确保了角的温度（这些是不相干的）被设置为 0.0。我们可以用 列表 2.1 中给出的模型来决定一个被分成 5×5 个元素的金属板的温度。其中左右下侧的温度为 0，上侧的温度为 100。

列表 2.1: 决定稳定状态温度的有限元平板模型 (`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; % 在点 (i,j) 处的温度
var float: left;   % 左侧温度
var float: right;  % 右侧温度
var float: top;    % 顶部温度
var float: bottom; % 底部温度

% 拉普拉斯方程：每一个内部点温度是它相邻点的平均值
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
% 边约束
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);

% 角约束
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 == w then "\n" else " " endif |
         i in HEIGHT, j in WIDTH
];
```

运行命令

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

得到输出

```
  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
----------
```

集合

集合变量用以下方式声明

```
set of <类型-实例化> : <变量名> ;
```

整型，枚举型（参见后面），浮点型和布尔型集合都可以定义。决策变量集合只可以是类型为整型或者枚举型的变量集合。集合常量有以下形式

```
{ <表达式-1>, ..., <表达式-n> }
```

或者是以下形式的整型，枚举型或浮点型 范围表达式

```
<表达式-1> .. <表达式-2>
```

标准的 集合操作符有：元素属于 (`in`), (非严格的) 集合包含 (`subset`), (非严格的) 超集关系 (`superset`), 并集 (`union`), 交集 (`intersect`), 集合差运算 (`diff`), 集合对称差 (`symdiff`) 和集合元素的个数 (`card`).
我们已经看到集合变量和集合常量（包含范围）可以被用来作为变量声明时的隐式类型。在这种情况下变量拥有集合元素中的类型并且被隐式地约束为集合中的一个成员。

我们的烤蛋糕问题是一个非常简单的批量生产计划问题例子。在这类问题中，我们希望去决定每种类型的产品要制造多少来最大化利润。同时制造一个产品会消耗不同数量固定的资源。我们可以扩展 列表 1.3 中的 MiniZinc 模型为一个不限制资源和产品类型的模型去处理这种类型的问题。这个模型在 列表 2.2 中给出。一个（烤蛋糕问题的）数据文件例子在 列表 2.3 中给出。

列表 2.2: 简单批量生产计划模型 (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 ];
```

列表 2.3: 简单批量生产计划模型的数据文件例子 (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 |];
```

这个模型的新特征是只用枚举类型 枚举类型。这使得我们可以把资源和产品的选择作为模型的参数。模型的第一个项

```
enum Products;
```

声明 Products 为 未知的产品集合。

枚举类型

枚举类型，我们称为 enums，用以下方式声明

```
enum <变量名> ;
```

一个枚举类型用以下赋值的方式定义

```
enum <变量名> = { <变量名-1>, ..., <变量名-n> } ;
```

其中 <变量名-1>, ..., <变量名-n> 是名为 <变量名> 的枚举类型中的元素。通过这个定义，枚举类型中的每个元素也被有效地声明为这个类型的一个新的常量。声明和定义可以像往常一样结合为一行。

第二个项声明了一个整型数组：

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

profit 数组的 下标集合是 Products 。理想情况下，这种声明方式表明只有集合 Products 中的元素才能被用来做数组的下标。

有 n 个元素组成的枚举类型中的元素的行为方式和整数 $1 \ldots n$ 的行为方式很像。它们可以被比较，它们可以按照它们出现在枚举类型定义中的顺序被排序，它们可以遍历，它们可以作为数组的下标，实际上，它们可以出现在一个整数可以出现的任何地方。

在数据文件例子中，我们用一列整数来初始化数组

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

意思是香蕉蛋糕的利润是 400，而巧克力蛋糕的利润是 450。在内部，BananaCake 会被看成是像整数 1 一样，而 ChocolateCake 会被看成像整数 2 一样。MiniZinc 虽然不提供明确的列表类型，但用 1..n 为下标集合的一维数组表现起来就像列表。我们有时候也会称它们为列表 lists 。

根据同样的方法，接下来的两项中我们声明了一个资源集合 Resources ，一个表明每种资源可获得量的数组 capacity。

更有趣的是项

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

声明了一个两维数组 consumption 。consumption[p,r] 的值是制造一单位的产品 p 所需要的资源 r 的数量。其中第一个下标是行下标，而第二个下标是列下标。

数据文件包含了一个两维数组的初始化例子:

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

注意分隔符 | 是怎样被用来分隔行的。

数组

因此，MiniZinc 提供一维和多维数组。它们用以下类型来声明：

```
array [ <下标集合-1>, ..., <下标集合-n> ] of <类型-实例化>
```

MiniZinc 要求数组声明要给出每一维的下标集合。下标集合或者是一个整型范围，一个被初始化为整型范围的集合变量，或者是一个 枚举类型。数组可以是所有的基类型：整型，枚举型，布尔型，浮点型或者字符串型。这些可以是固定的或者不固定的，除了字符串型，它只可以是参数。数组也可以作用于集合但是不可以作用于数组。
一维数组常量有以下格式

```
[ <表达式-1>, ..., <表达式-n> ]
```

而 二维数组常量有以下格式

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

其中这个数组有 m 行 n 列。
内建函数 array1d，array2d 等家族可以被用来从一个列表（或者更准确的说是一个一维数组）去实例化任何维度的数组。调用

```
array<n>d(<下标集合-1>, ..., <下标集合-n>, <列表>)
```

返回一个 n 维的数组，它的下标集合在前 n 个参数给出，最后一个参数包含了数组的元素。例如 array2d(1..3, 1..2, [1, 2, 3, 4, 5, 6]) 和 [|1, 2 |3, 4 |5, 6|] 是相等的。
数组元素按照通常的方式获取 获取：a[i,j] 给出第 i^{th} 行第 j^{th} 列的元素。
串联操作符 ++ 可以被用来串联两个一维的数组。结果得到一个列表，即一个元素从 1 索引的一维数组。例如 [4000, 6] ++ [2000, 500, 500] 求得 [4000, 6, 2000, 500, 500] 。内建函数 length 返回一维数组的长度。

模型的下一项定义了参数 mproducts 。它被设为可以生产出的任何类型产品的数量上限。这个确实是一个复杂的内嵌数组推导式和聚合操作符例子。在我们试图理解这些项和剩下的模型之前，我们应该先介绍一下它们。

首先，MiniZinc 提供了在很多函数式编程语言都提供的列表推导式。例如，列表推导式 [i + j | i, j in 1..3 where j < i] 算得 [2 + 1, 3 + 1, 3 + 2] 等同于 [3, 4, 5] 。[3, 4, 5] 只是一个下标集合为 1..3 的数组。

MiniZinc 同时也提供了集合推导式，它有类似的语法：例如 {i + j | i, j in 1..3 where j < i} 计算得到集合 {3, 4, 5} 。

列表和集合推导式

列表推导式的一般格式是

```
[ <表达式> | <生成元表达式> ]
```

<表达式> 指明了如何从 <生成元表达式> 产生的元素输出列表中创建元素。生成元 <generator-exp> 由逗号分开的一列生成元表达式组成，选择性地跟着一个布尔型表达式。两种格式是

```
<生成元>, ..., <生成元>
<生成元>, ..., <生成元> where <布尔表达式>
```

第二种格式中的可选择的 <布尔型表达式> 被用作生成元表达式的过滤器：只有满足布尔型表达式的输出列表中的元素才被用来构建元素。生成元 <generator> 有以下格式

```
<标识符>, ..., <标识符> in <数组表达式>
```

每一个标识符是一个 迭代器，轮流从数值表达式中取值，最后一个标识符变化的最迅速。列表推导式的生成元和 <布尔型表达式> 通常不会涉及决策变量。如果它们确实涉及了决策变量，那么产生的列表是一列 var opt <T> ，其中 T 是 <表达式> 的类型。更多细节，请参考选项类型 (page 73) 中有关选项类型 option types 的论述。
集合推导式几乎和列表推导式一样：唯一的不同是这里使用 { 和 } 括住表达式而不是 [和] 。集合推导式生成的元素必须是固定的 fixed ，即不能是决策变量。类似的，集合推导式的生成元和可选择的 <布尔型表达式> 必须是固定的。

第二，MiniZinc 提供了一系列的可以把一维数组的元素聚合起来的内建函数。它们中最有用的可能是:mzn:forall 。它接收一个布尔型表达式数组（即，约束），返回单个布尔型表达式，它是对数组中的布尔型表达式的逻辑合取。

例如，以下表达式

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

其中 a 是一个下标集合为 1..3 的算术数组。它约束了 a 中的元素是互相不相同的。列表推导式计算得到 [a[1] != a[2], a[1] != a[3], a[2] != a[3]] ，所以 forall 函数返回逻辑合取 a[1] != a[2] /\ a[1] != a[3] /\ a[2] != a[3] 。

聚合函数

算术数组的 聚合函数有: sum 把元素加起来，product 把元素乘起来，和 min 跟 max 各自返回数组中的最小和最大元素。当作用于一个空的数组时，min 和 max 返回一个运行错误，sum 返回 0，product 返回 1。
MiniZinc 为数组提供了包含有布尔型表达式的四个聚合函数。正如我们看到的，它们中的第一个是 forall ，它返回一个等于多个约束的逻辑合取的单个约束。第二个函数，exists ，返回多个约束的逻辑析取。因此 forall 强制数组中的所有约束都满足，而 exists 确保至少有一个约束满足。第三个函数，xorall 确保奇数个约束满足。第四个函数，iffall 确保偶数个约束满足。

第三个，也是难点的最后一个部分是当使用数组推导式时，MiniZinc 允许使用一个特别的聚合函数的语法。建模者不仅仅可以用

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

也可以用一个更加数学的表示

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

两种表达方式是完全相等的：建模者可以自由使用任何一个他们认为更自然的表达方式。

生成表达式

一个 生成表达式有以下格式

```
<聚合函数> ( <生成元表达式> ) ( <表达式> )
```

圆括号内的 <生成元表达式> 以及构造表达式 <表达式> 是非选择性的：它们必须存在。它等同于

```
<聚合函数> ( [ <表达式> | <生成元表达式> ] )
```

<聚合函数> aggregation function 可以是 MiniZinc 的任何由单个数组作为其参数的函数。

接下来我们就来了解 列表 2.2 中的简单批量生产计划模型剩余的部分。现在请暂时忽略定义 `mproducts` 的这部分。接下来的项：

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

定义了一个一维的决策变量数组 `produce` 。`produce[p]` 的值代表了最优解中产品 `p` 的数量。下一项

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

定义了一个辅助变量集合来记录每一种资源的使用量。下面的两个约束

```
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] );
```

使用 `used[r]` 计算资源 `r` 的总体消耗以及保证它是少于可获得的资源 `r` 的量。最后，项

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

表明这是一个最大化问题以及最大化的目标是全部利润。

现在我们回到 `mproducts` 的定义。对每个产品 `p` ，表达式

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

决定了在考虑了每种资源 r 的数量以及制造产品 p 需要的 r 量的情况下，p 可以生产的最大量。注意过滤器 where consumption[p,r] > 0 的使用保证了只有此产品需要的资源才会考虑，因此避免了出现除数为零的错误。所以，完整的表达式

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

计算 任何产品可以被制造的最大量，因此它可以被作为 produce 中的决策变量定义域的上限。

最后，注意输出项是比较复杂的，并且使用了列表推导式 列表推导式去创建一个易于理解的输出。运行

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

输出得到如下结果

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

2.2.2 全局约束

MiniZinc 包含了一个全局约束的库，这些全局约束也可以被用来定义模型。一个例子是 alldifferent 约束，它要求所有参数中的变量都必须是互相不相等的。

列表 2.4: SEND+MORE=MONEY 算式谜题模型 (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"];
```

SEND+MORE=MONEY 问题要求给每一个字母赋不同的数值使得此算术约束满足。列表 2.4 中的模型使用 `alldifferent([S,E,N,D,M,O,R,Y])` 约束表达式来保证每个字母有不同的数字值。在使用了引用项

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

后，此全局约束 `alldifferent` 可以在模型中使用。我们可以用以下代替此行

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

它包含了所有的全局约束。

一系列所有在 MiniZinc 中定义了的全局约束都被包含在了发布的文档中。对一些重要的全局约束的描述，请参见全局约束 (page 57) 。

2.2.3 条件表达式

MiniZinc 提供了一个条件表达式 *if-then-else-endif* 。它的一个使用例子如下

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

若 `y` 不是零，则 `r` 设为 `x` 除以 `y` ，否则则设为零。

条件表达式

条件表达式的格式是

```
if <布尔型表达式> then <表达式-1> else <表达式-2> endif
```

它是一个真表达式而不是一个控制流语句，所以它可以被用于其他表达式中。如果 `<布尔型表达式>` 是真，则它取值 `<表达式-1>` ，否则则是 `<表达式-2>` 。条件表达式的类型是 `<表达式-1>` 和 `<表达式-2>` 的类型，而它们俩必须有相同的类型。
如果 `<布尔型表达式>` 包含决策变量，则表达式的类型-实例化是 `var <T>` ，其中 `<T>` 是 `<表达式-1>` 和 `<表达式-2>` 的类型，就算是在两个表达式都已经固定了的情况下也是如此。

列表 2.5: 广义数独问题的模型 (sudoku.mzn)

```
include "alldifferent.mzn";

int: S;
int: N = S * S;
int: digs = ceil(log(10.0,int2float(N))); % 输出的数字

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

```
array[1..N,1..N] of 0..N: start; %% 板初始0 = 空
array[1..N,1..N] of var PuzzleRange: puzzle;

% 填充初始板
constraint forall(i,j in PuzzleRange)(
    if start[i,j] > 0 then puzzle[i,j] = start[i,j] else true endif );

% 每行中取值各不相同
constraint forall (i in PuzzleRange) (
                   alldifferent( [ puzzle[i,j] | j in PuzzleRange ]) );

% 每列中取值各不相同
constraint forall (j in PuzzleRange) (
                   alldifferent( [ puzzle[i,j] | i in PuzzleRange ]) );

% 每个子方格块中取值各不相同
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"];
```

列表 2.6: 广义数独问题的数据文件例子 (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|];
```

在创建复杂模型或者复杂输出时，条件表达式是非常有用的。我们来看下 列表 2.5 中的数独问题模型。板的初始位置在参数 `start` 中给出，其中 0 代表了一个空的板位置。通过使用以下条件表达式

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

它被转换为对决策变量 `puzzle` 的约束。

	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	

图 2.1: sudoku.dzn 代表的问题。

在定义复杂输出时，条件表达式也很有用。output 在数独模型 列表 2.5 中，表达式

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

在大小为 S 的组群之间插入了一个额外的空格。输出表达式同时也使用条件表达式来在每 S 行后面加入一个空白行。这样得到的输出有很高的可读性。

剩下的约束保证了每行中，每列中以及每 $S \times S$ 子方格块中的值都是互相不相同的。

通过使用标示 -a 或 --all-solutions ，我们可以用 MiniZinc 求解得到一个满足问题 (solve satisfy) 的所有解。运行

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

得到

```
 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
----------
==========
```

当系统输出完所有可能的解之后，此行 ========== 被输出。在这里则表明了此问题只有一个解。

2.2.4 枚举类型

枚举类型允许我们根据一个或者是数据中的一部分，或者在模型中被命名的对象集合来创建模型。这样一来，模型就更容易被理解和调试。我们之前已经简单介绍了枚举类型或者 enums。在这一小分段，我们会探索如何可以全面地使用它们，并且给出一些处理枚举类型的内建函数。

让我们重新回顾一下*MiniZinc 基本模型* (page 21) 中的给澳大利亚涂色问题。

列表 2.7: 使用枚举类型的澳大利亚涂色模型 (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;
```

列表 2.7 中的模型声明了一个枚举类型 `Color` ，而它必须在数据文件中被定义。每一个州变量被声明为从此枚举类型中取一个值。使用以下方式运行这个程序

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

可能会得到输出

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

枚举类型变量声明

一个枚举类型参数变量被声明为以下两种方式：

```
<枚举名> : <变量名>
<l>..<u> : <变量名>
```

其中 <枚举名> 是枚举类型的名字，<l> 和 <u> 是此枚举类型的固定枚举类型表达式。

枚举类型一个重要的行为是，当它们出现的位置所期望的是整数时，它们会自动地强制转换为整数。这样一来，这就允许我们使用定义在整数上的全局变量，例如

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

要求每种颜色至少有两个州涂上并且有三个州被涂了蓝色。

枚举类型操作符

有一系列关于枚举类型的内部操作符：

- `enum_next(X,x)`：返回枚举类型 `X` 中 `x` 后的下一个值。这是一个部份函数，如果 `x` 是枚举类型 `X` 最后一个值，则函数会返回 ⊥ 令包含这个表达式的布尔表达式返回 `false` 。
- `enum_prev(X,x)` `enum_prev(X,x)`：返回枚举类型 `X` 中 `x` 的上一个值。`enum_prev` 同样是一个部份函数。
- `to_enum(X,i)`：映射一个整型表达式 `i` 到一个在 `X` 的枚举类型值，或者如果 `i` 是小于等于 0 或大于:mzn:*X*‘中元素的个数，则返回:*math*:bot‘ 。

注意，一些标准函数也是可以应用于枚举类型上

- `card(X)`：返回枚举类型 `X` 的势。
- `min(X)`：返回枚举类型 `X` 中最小的元素。
- `max(X)`：返回枚举类型 `X` 中最大的元素。

2.2.5 复杂约束

约束是 MiniZinc 模型的核心。我们已经看到了简单关系表达式，但是约束其实是比这更加强大的。一个约束可以是任何布尔型表达式。想象一个包含两个时间上不能重叠的任务的调度问题。如果 `s1` 和 `s2` 是相对应的起始时间，`d1` 和 `d2` 是相对应的持续时间，我们可以表达约束为：

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

来保证任务之间互相不会重叠。

布尔型

MiniZinc 中的布尔型表达式可以按照标准的数学语法来书写。布尔常量是 `真` 或 `假` ，布尔型操作符有合取，即，与 (/\) ，析取，即，或 (\/) ，必要条件蕴含 (<-) ，充分条件蕴含 (->) ，充分必要蕴含 (<->) 以及非 (`not`)。内建函数 `bool2int` 强制转换布尔型为整型：如果参数为真，它返回 1，否则返回 0。

列表 2.8: 车间作业调度问题模型 (`jobshop.mzn`).

```
enum JOB;
enum TASK;
TASK: last = max(TASK);
array [JOB,TASK] of int: d;                    % 任务持续时间
int: total = sum(i in JOB, j in TASK)(d[i,j]);% 总持续时间
int: digs = ceil(log(10.0,int2float(total))); % 输出的数值
array [JOB,TASK] of var 0..total: s;           % 起始时间
var 0..total: end;                             % 总结束时间


constraint %% 保证任务按照顺序出现
    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 %% 保证任务之间没有重叠
    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 = \(end)\n"] ++
       [ show_int(digs,s[i,j]) ++ " " ++
         if j == last then "\n" else "" endif |
         i in JOB, j in TASK ];
```

列表 2.9: 车间作业调度问题数据 (jdata.dzn).

```
JOB = Job(1..5);
TASK = Task(1..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 |];
```

列表 2.8 中的车间作业调度模型给出了一个使用析取建模功能的现实例子。车间作业调度问题中，我们有一个作业集合，每一个包含一系列的在不同机器上的任务：任务 [i,j] 是在第 i^{th} 个作业中运行在第 j^{th} 个机器上的任务。每列任务必须按照顺序完成，并且运行在同一个机器上的任何两个任务在时间上都不能重叠。就算是对这个问题的小的实例找最优解都会是很有挑战性的。

命令

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

求解了一个小的车间作业调度问题，并且显示了优化问题在 all-solutions 下的表现。在这里，求解器只有当找到一个更好的解时才会输出它，而不是输出所有的可能最优解。这个命令下的（部分）输出是：

```
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
----------
==========
```

表明一个结束时间为 30 的最优解终于被找到，并且被证明为是最优的。通过加一个约束 `end = 30` ，并且把求解项改为 `solve satisfy` ，然后运行

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

我们可以得到所有的 最优解。这个问题有 3,444,375 个最优解。

列表 2.10: 稳定婚姻问题模型 (stable-marriage.mzn).

```
int: n;

enum Men = _(1..n);
enum Women = _(1..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= \(wife)\nhusbands= \(husband)\n"];
```

列表 2.11: 稳定婚姻问题模型的数据文件例子。(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 |];
```

MiniZinc 中的另外一个强大的建模特征是决策变量可以被用来访问数组 array access 。作为一个例子，考虑（老式的）稳定婚姻问题。我们有 n 个（直）女以及 n 个（直）男。每一个男士有一个女士排行榜，女士也是。我们想给每一个女士/男士找一个丈夫/妻子来使得所有的婚姻按以下意义上来说都是 稳定的：

- 每当 m 喜欢另外一个女士 o 多过他的妻子 w 时，o 喜欢她的丈夫多过 m ，以及
- 每当 w 喜欢另外一个男士 o 多过她的丈夫 m 时，o 喜欢他的妻子多过 w。

这个问题可以很优雅地在 MiniZinc 中建模。模型和数据例子在 列表 2.10 和 列表 2.11 中分别被给出。

模型中的前三项声明了男士/女士的数量以及男士和女士的集合。在这里我们介绍 匿名枚举类型的使用。Men 和 Women 都是大小为 n 的集合，但是我们不希望把它们混合到一起，所以我们使用了一个匿名枚举类型。这就允许 MiniZinc 检测到使用 Men 为 Women 或者反之的建模错误。

矩阵 rankWomen 和 rankMen 分别给出了男士们的女士排行以及女士们的男士排行。因此，项 rankWomen[w,m] 给出了女士 texttt{w} 的关于男士 texttt{m} 的排行。在排行中的数目越小，此男士或者女士被选择的倾向越大。

有两个决策变量的数组：wife 和 husband 。这两个分别代表了每个男士的妻子和每个女士的丈夫。

前两个约束

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

确保了丈夫和妻子的分配是一致的：w 是 m 的妻子蕴含了 m 是 w 的丈夫，反之亦然。注意在 husband[wife[m]] 中，下标表达式 wife[m] 是一个决策变量，而不是一个参数。

接下来的两个约束是稳定条件的直接编码：

```
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] );
```

在有了用决策变量作为数组的下标和用标准的布尔型连接符构建约束的功能后，稳定婚姻问题的自然建模才变得可行。敏锐的读者可能会在这时产生疑问，如果数组下标变量取了一个超出

数组下标集合的值，会产生什么情况。MiniZinc 把这种情况看做失败：一个数组访问 `a[e]` 在其周围最近的布尔型语境中隐含地加入了约束 `e in index_set(a)` ，其中 `index_set(a)` 给出了 `a` 的下标集合。

匿名枚举类型

一个 *匿名枚举类型*表达式有格式 `anon_enum(<n>)` ，其中 `<n>` 是一个固定的整型表达式，它定义了枚举类型的大小。
除了其中的元素没有名字，匿名枚举类型和其他的枚举类型一样。当被输出时，它们根据枚举类型的名字被给定独有的名字。

例如，如下的变量声明

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

约束 `a[x] = y` 会在 $x = 1 \wedge y = 2$ 和 $x = 2 \wedge y = 3$ 时得到满足。约束 `not (a[x] = y)` 会在 $x = 0 \wedge y = 2$，$x = 0 \wedge y = 3$, $x = 1 \wedge y = 3$ 和 $x = 2 \wedge y = 2$ 时得到满足。

当参数无效访问数组时，正式的 MiniZinc 语义会把此情况看成失败来确保参数和决策变量的处理方式是一致的，但是会发出警告，因为这种情况下几乎总是会有错误出现。

列表 2.12: 魔术串问题模型 (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" ] ;
```

强制转换函数 `bool2int` 可以被任何布尔型表达式调用。这就使得 MiniZinc 建模者可以使用所谓的 *高价约束*。举个简单的例子，请看 *魔术串问题*：找到一列数字 $s = [s_0, \ldots, s_{n-1}]$ 使得:math:*s_i* 是数字 i 出现在 s 的次数。一个解的例子是 $s = [1, 2, 1, 0]$ 。

这个问题的一个 MiniZinc 模型在 列表 2.12 中给出。列表 2.12 的使用使得我们可以把函数 `s[j]=i` 满足的次数加起来。运行命令

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

得到输出

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

确切地显示出这个问题的两个解。

注意当有需要的时候，MiniZinc 会自动地强制转换布尔型为整型以及整型为浮点型。我们可以把 列表 2.12 中的约束项替换为

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

由于 MiniZinc 系统实际上会自动地加入缺失的 `bool2int`，布尔型表达式 `s[j] = i` 会被自动地强制转换为整型，所以会得到同样的结果。

强制转换

MiniZinc 中，通过使用函数 `bool2int` ，我们可以把一个布尔型数值 emph{强制转换} 为一个整型数值。同样地，通过使用函数 `int2float` ，我们也可以把一个整型数值强制转换为一个浮点型数值。被强制转换的数值的实例化和原数值一样。例如，`par bool` 被强制转换为 `par int` ，而 `var bool` 被强制转换为 `var int` 。
通过适当地在模型中加入 `bool2int` 和 `int2float` ，MiniZinc 会自动地强制转换布尔型表达式为整型表达式，以及整型表达式为浮点型表达式。注意通过两步转换，它也会强制转换布尔型为浮点型。

2.2.6 集合约束

MiniZinc 另外一个强大的建模特征是它允许包含整数的集合是决策变量：这表示当模型被评估时，求解器会查找哪些元素在集合中。

举个简单的例子，*0/1* 背包问题。这个问题是背包问题的局限版本，即我们或者选择把物品放入背包或者不放。每一个物品有一个重量和一个利润，在受限制于背包不能太满的条件下，我们想找到选取哪些物品会得到最大化利润。

很自然地，我们在 MiniZinc 中使用单个的决策变量来建模：`var set of ITEM: knapsack` 其中 `ITEM` 是可放置的物品集合。如果数组 `weight[i]` 和 `profit[i]` 分别给出物品 `i` 的重量和利润，以及背包可以装载的最大重量是 `capacity`，则一个自然的模型在 列表 2.13 中给出。

列表 2.13: 0/1 背包问题模型 (`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"];
```

注意，关键字 `var` 出现在 `set` 声明之前，表明这个集合本身是决策变量。这就和一个 `var` 关键字描述其中元素而不是数组自身的数组形成对比，因为此时数组的基本结构，即它的下标集合，是固定了的。

列表 2.14: 高尔夫联谊问题模型 (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 ];
```

列表 2.15: 高尔夫联谊问题数据 (social-golfers.dzn).

```
weeks = 4;
groups = 4;
size =3 ;
```

我们来看一个更复杂的关于集合约束的例子，列表 2.14 中给出的高尔夫联谊问题。这个问题的

目的是给 `groups` × `size` 个高尔夫手在 `weeks` 时间内安排一个高尔夫联赛。每一周我们需要安排 `groups` 个大小为 `size` 的不同的组。任何一对高尔夫手都不能一起出现于两个组中进行比赛。

模型中的变量是第 i^{th} 周第 j^{th} 组的高尔夫手:mzn:*Sched[i,j]* 组成的集合。

11-32 行中的约束首先对每一周的第一个集合进行一个排序来去除掉周之间可以互相调换的对称。然后它对每一周内的集合进行了一个排序，同时使得每一个集合的势为 `size` 。接下来通过使用全局约束 `partition_set` ，确保了每一周都是对高尔夫手集合的一个划分。最后一个约束确保了任何两个高尔夫手都不会一起在两个组内比赛（因为任何两个组的交集的势最多都是 1）。

我们也有

在 34-46 行中，我们也给出了去对称初始化约束：第一周被固定为所有的高尔夫手都按顺序排列；第二周的第一组被规定为是由第一周的前几组的第一个选手组成；最后，对于剩下的周，模型规定第一个 `size` 内的高尔夫手们出现在他们相对应的组数中。

运行命令

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

其中数据文件定义了一个周数为 4，大小为 3，组数为 4 的问题，得到如下结果

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

注意范围集合是如何以范围格式输出的。

2.2.7 汇总

我们以一个可以阐释这一章介绍的大部分特征的复杂例子来结束这一节，包括枚举类型，复杂约束，全局约束以及复杂输出。

列表 2.16: 使用枚举类型规划婚礼座位 (`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; % 客人的座位
array[Hatreds] of var Seats: p1;  % 互相憎恶的客人1的座位
array[Hatreds] of var Seats: p2;  % 互相憎恶的客人2的座位
array[Hatreds] of var 0..1: sameside;  % 互相憎恶的客人是否坐在同一边
array[Hatreds] of var Seats: cost;  % 互相憎恶的客人的距离

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"];
```

列表 2.16 中的模型安排婚礼桌上的座位。这个桌子有 12 个编码的顺序排列的座位，每边有 6 个。男士必须坐奇数号码的座位，女士坐偶数。Ed 由于恐惧症不能坐在桌子的边缘，新郎和新娘必须坐在彼此旁边。我们的目的是最大化已知的互相憎恶的人之间的距离。如果在同一边，座位之间的距离是座位号码之间的差，否则则是和其对面座位的距离 + 1。

注意在输出语句中我们观察每个座位 `s` 来找一个客人 `g` 分配给此座位。我们利用内建函数 `fix`，它检查一个决策变量是否是固定的以及输出它的固定值，否则的话中断。在输出语句中使用此函数总是安全的，因为当输出语句被运行的时候，所有的决策变量都应该是固定了的。

运行

```
$ minizinc wedding.mzn
```

得到输出

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

最终得到的座位安排在 图 2.2 中给出。其中连线表示互相憎恶，总的距离是 22.

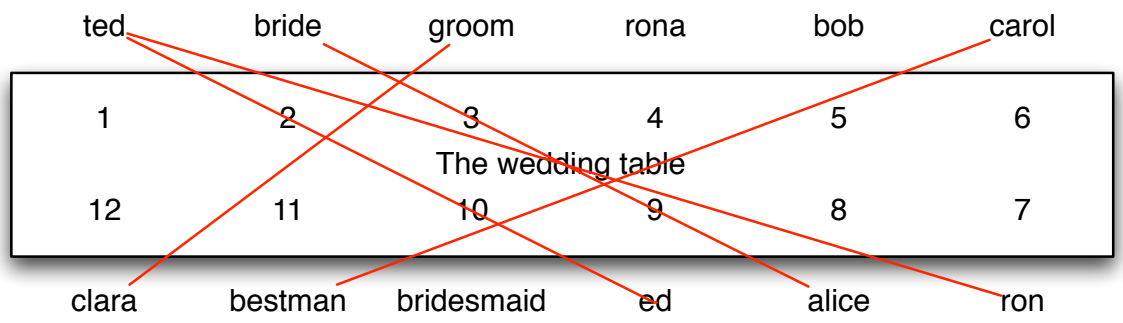


图 2.2: 婚礼桌上座位的安排

固定

输出项中，内建函数 `fix` 检查一个决策变量的值是否固定，然后把决策变量的实例化强制转换为参数。

CHAPTER 2.3

谓词和函数

MiniZinc 中的谓词允许我们用简洁的方法来表达模型中的复杂约束。MiniZinc 中的谓词利用预先定义好的全局约束建模，同时也让建模者获取以及定义新的复杂约束。MiniZinc 中的函数用来捕捉模型中的共同结构。实际上，一个谓词就是一个输出类型为 `var bool` 的函数。

2.3.1 全局约束

MiniZinc 中定义了很多可以在建模中使用的全局约束。由于全局约束的列表一直在慢慢增加，最终确定的列表可以在发布的文档中找到。下面我们讨论一些最重要的全局约束。

2.3.1.1 Alldifferent

约束 `alldifferent` 的输入为一个变量数组，它约束了这些变量取不同的值。

`alldifferent` 的使用有以下格式

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

即，参数是一个整型变量数组。

`alldifferent` 是约束规划中被最多研究以及使用的全局约束之一。它被用来定义分配子问题，人们也给出了 `alldifferent` 的高效全局传播器。`send-more-money.mzn` (列表 2.4) 和 `sudoku.mzn` (列表 2.5) 是使用 `alldifferent` 的模型例子。

2.3.1.2 Cumulative

约束 `cumulative` 被用来描述资源累积使用情况。

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

规定对于一个起始时间为 s ，持续时间为 d 以及资源需求量为 r 的任务集合，在任何时间对资源的需求量都不能超过一个全局资源量界限 b 。

列表 3.1: 使用 cumulative 来建模搬运家具问题的模型 (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 = \(start)\nend = \(end)\n"];
```

列表 3.2: 使用 cumulative 来建模搬运家具问题的数据 (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;
```

列表 3.1 中的模型为搬运家具规划一个行程表使得每一份家具在搬运的过程中都有足够的搬用工和足够的手推车可以使用。允许的时间，可以使用的搬运工以及手推车被给出，每个物体的搬运持续时间，需要的搬运工和手推车的数量等数据也被给出。使用 列表 3.2 中的数据，命令

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

可能会得到如下输出

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

图 3.1 and 图 3.2 给出了这个解中搬运时每个时间点所需要的搬运工和手推车。

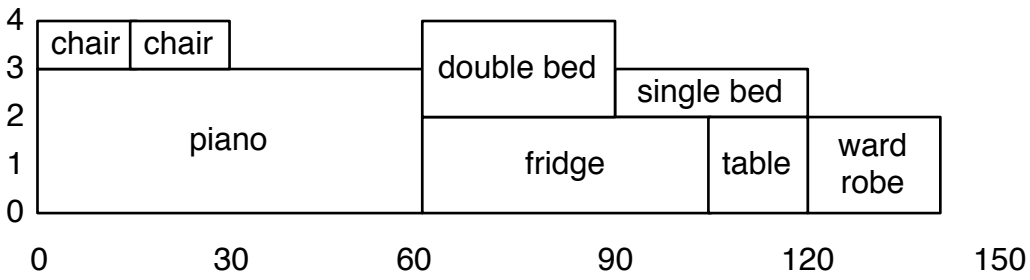


图 3.1: 搬运时搬运工使用量直方图

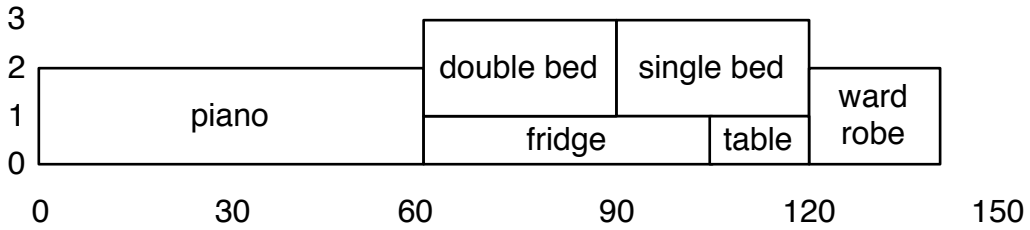


图 3.2: 搬运时手推车使用量直方图

2.3.1.3 Table

约束 `table` 强制变量元组从一个元组集合中取值。由于 MiniZinc 中没有元组，我们用数组来描述它。根据元组是布尔型还是整型，`table` 的使用有以下两种格式

```
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)
```

强制约束了 $x \in t$ ，其中 x 和 t 中的每一行是元组，t 是一个元组集合。

列表 3.3: 使用 `table` 约束来建模食物规划问题的模型 (`meal.mzn`).

```
% 规划均衡的膳食
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; % 食物数据库

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"];
```

列表 3.4: 定义 table 的食物规划的数据 (meal.dzn).

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

dd = [| icecream,      1200,  50,  10, 120,  400     % 冰淇淋
      | banana,         800, 120,   5,  20,  120     % 香蕉
      | chocolatecake, 2500, 400,  20, 100,  600     % 巧克力蛋糕
      | lasagna,       3000, 200, 100, 250,  450     % 千层面
      | steak,         1800, 800,  50, 100, 1200     % 牛排
      | rice,          1200,  50,   5,  20,  100     % 米饭
      | chips,         2000,  50, 200, 200,  250     % 薯条
      | brocolli,       700, 100,  10,  10,  125     % 花椰菜
      | beans,         1900, 250,  60,  90,  150 |]; % 黄豆

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

列表 3.3 中的模型寻找均衡的膳食。每一个食物项都有一个名字（用整数表示），卡路里数，蛋白质克数，盐毫克数，脂肪克数以及单位为分的价钱。这些个项之间的关系用一个 table 约束来描述。模型寻找拥有最小花费，最少卡路里数 min_energy ，最少蛋白质量 min_protein ，最大盐分 max_salt 以及脂肪 max_fat 的膳食。

2.3.1.4 Regular

约束 regular 用来约束一系列的变量取有限自动机定义的值。regular 的使用有以下方式

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

它约束了 `x` 中的一列值（它们必须是在范围 range `1..S` 内）被一个有 `Q` 个状态，输入为 `1..S`，转换函数为 `d` （`<1..Q, 1..S>` 映射到 `0..Q`），初始状态为 `q0` （必须在 `1..Q` 中）和接受状态为 `F` （必须在 `:mzn:` `1..Q` 中）的 DFA 接受。状态 0 被保留为总是会失败的状态。

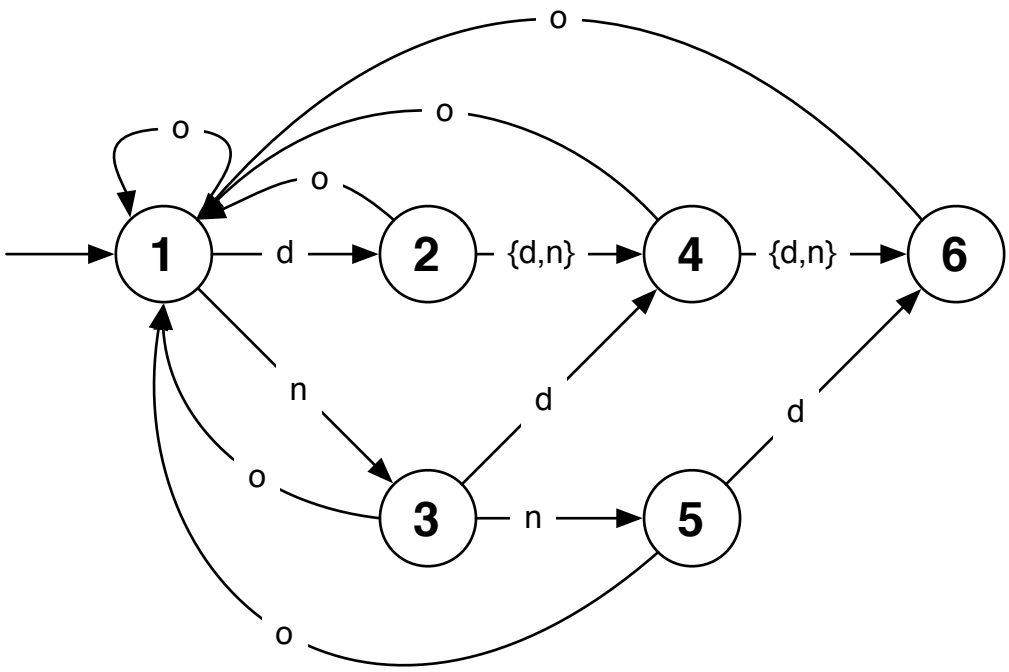


图 3.3: 判定正确排班的 DFA。

我们来看下护士排班问题。每一个护士每一天被安排为以下其中一种：(d) 白班 (n) 夜班或者 (o) 休息。每四天，护士必须有至少一天的休息。每个护士都不可以被安排为连续三天夜班。这个问题可以使用 图 3.3 中的不完全 DFA 来表示。我们可以把这个 DFA 表示为初始状态是 `1`，结束状态是 `1..6`，转换函数为

	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

注意状态表中的状态 0 代表一个错误状态。列表 3.5 中给出的模型为 `num_nurses` 个护士 `num_days` 天寻找一个排班，其中我们要求白天有 `req_day` 个护士值班，晚上有 `req_night` 个护士值班，以及每个护士至少有 `min_night` 个夜班。

列表 3.5: 使用 regular 约束来建模的护士排班问题模型 (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 };

enum STATE = { S1, S2, S3, S4, S5, S6 };

array[STATE,SHIFT] of opt STATE: t =
     [|      d:  n:  o:
      | S1:  S2, S3, S1
      | S2:  S4, S4, S1
      | S3:  S4, S5, S1
      | S4:  S6, S6, S1
      | S5:  S6, <>, S1
      | S6:  <>, <>, S1|];

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], t, S1, STATE) /\
             sum(j in DAY)(roster[i,j] == n) >= min_night
           );

solve satisfy;
```

运行命令

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

找到一个给 7 个护士 10 天的排班，要求白天有 3 个人值班，夜晚有 2 个人值班，以及每个护士最少有 2 个夜班。一个可能的结果是

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

另外一种 regular 约束是 `regular_nfa` 。它使用 NFA（没有 `\epsilon` 弧）来定义 regular 表达

式。此约束有以下格式

```
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)
```

它约束了数组 x 中的数值序列（必须在范围 1..S 中）被含有 Q 个状态，输入为 1..S ，转换函数为 d （映射 <1..Q, 1..S> 到 1..Q 的子集），初始状态为 q0 （必须在范围 1..Q 中）以及接受状态为 F （必须在范围 1..Q 中）的 NFA 接受。在这里，我们没必要再给出失败状态 0，因为转换函数可以映射到一个状态的空集。

2.3.2 定义谓词

MiniZinc 的其中一个最强大的建模特征是建模者可以定义他们自己的高级约束。这就使得他们可以对模型进行抽象化和模块化。也允许了在不同的模型之间重新利用约束以及促使了用来定义标准约束和类型的特殊库应用的发展。

列表 3.6: 使用谓词的车间作业调度问题模型 (jobshop2.mzn)

```
int: jobs;                                      % 作业的数量
set of int: JOB = 1..jobs;
int: tasks;                                     % 每个作业的任务数量
set of int: TASK = 1..tasks;
array [JOB,TASK] of int: d;                     % 任务持续时间
int: total = sum(i in JOB, j in TASK)(d[i,j]);% 总持续时间
int: digs = ceil(log(10.0,total));              % 输出的数值
array [JOB,TASK] of var 0..total: s;            % 起始时间
var 0..total: end;                              % 总结束时间

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

constraint %% 保证任务按照顺序出现
    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 %% 保证任务之间没有重叠
    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 = \(end)\n"] ++
       [ show_int(digs,s[i,j]) ++ " " ++
         if j == tasks then "\n" else "" endif |
         i in JOB, j in TASK ];
```

我们用一个简单的例子开始，回顾下前面章节中的车间作业调度问题。这个模型在 列表 3.6 中

给出。我们感兴趣的项是 谓词 项:

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

它定义了一个新的约束用来约束起始时间为 s1 ，持续时间为 d1 的任务不能和起始时间为 s2 ，持续时间为 d2 的任务重叠。它可以在模型的任何（包含决策变量的）布尔型表达式可以出现的地方使用。

和谓词一样，建模者也可以定义只涉及到参数的新的约束。和谓词不一样的是，它们可以被用在条件表达式的测试中。它们被关键字 test 定义。例如

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

谓词定义

使用以下形式的语句，我们可以定义谓词

```
predicate <谓词名> ( <参数定义>, ..., <参数定义> ) = <布尔表达式>
```

<谓词名> 必须是一个合法的 MiniZinc 标识符，每个 <参数定义> 都是一个合法的 MiniZinc 类型 type 声明。
参数定义的一个松弛是数组的索引类型可以是 没有限制地写为 int 。
类似的，使用以下形式的语句，我们定义测试

```
test <谓词名> ( <参数定义>, ..., <参数定义> ) = <布尔表达式>
```

其中的 <布尔表达式> 必须是固定的。
另外我们介绍一个谓词中使用到的 assert 命令的新形式。

```
assert ( <布尔表达式>, <字符串表达式>, <表达式> )
```

assert 表达式的类型和最后一个参数的类型一样。assert 表达式检测第一个参数是否为假，如果是则输出第二个参数字符串。如果第一个参数是真，则输出第三个参数。

注意 assert 表达式中的第三个参数是延迟的，即如果第一个参数是假，它就不会被评估。所以它可以被用来检查

```
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]
    );
```

此代码在 i 超出数组 x 的范围时不会计算 x[i] 。

2.3.3 定义函数

MiniZinc 中的函数和谓词一样定义，但是它有一个更一般的返回类型。

下面的函数定义了一个数独矩阵中的第 a^{th} 个子方块的第 $a1^{th}$ 行。

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

有了这个定义之后，我们可以把 列表 2.5 中的数独问题的最后一个约束替换为

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

函数对于描述模型中经常用到的复杂表达式非常有用。例如，想象下在 $n \times n$ 的方格的不同位置上放置数字 1 到 n 使得任何两个数字 i 和 j 之间的曼哈顿距离比这两个数字其中最大的值减一还要大。我们的目的是最小化数组对之间的总的曼哈顿距离。曼哈顿距离函数可以表达为：

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

完整的模型在 列表 3.7 中给出。

列表 3.7: 阐释如何使用函数的数字放置问题模型 (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 ];
```

函数定义

函数用以下格式的语句定义

```
function <返回类型> : <函数名> ( <参数定义>, ..., <参数定义> ) = <表达式>
```

<函数名>必须是一个合法的 MiniZinc 标识符。每一个<参数定义>是一个合法的 MiniZinc 类型声明。<返回类型>是函数的返回类型，它必须是<表达式>的类型。参数和谓词定义中的参数有一样的限制。

MiniZinc 中的函数可以有任何返回类型，而不只是固定的返回类型。在定义和记录多次出现在模型中的复杂表达式时，函数是非常有用的。

2.3.4 反射函数

为了方便写出一般性的测试和谓词，各种反射函数会返回数组的下标集合，var 集合的定义域以及决策变量范围的信息。关于下标集合的有以下反射函数 `index_set(<1-D array>)`，`index_set_1of2(<2-D array>)`，`index_set_2of2(<2-D array>)`，以及关于更高维数组的反射函数。

车间作业问题的一个更好的模型是把所有的对于同一个机器上的不重叠约束结合为一个单个的析取约束。这个方法的一个优点是虽然我们只是初始地把它建模成一个 `non-overlap` 约束的连接，但是如果下层的求解器对于解决析取约束有一个更好的方法，在对我们的模型最小改变的情况下，我们可以直接使用它。这个模型在 列表 3.8 中给出。

列表 3.8: 使用 `disjunctive` 谓词的车间作业调度问题模型 (`jobshop3.mzn`).

```
include "disjunctive.mzn";

int: jobs;                                      % 作业的数量
set of int: JOB = 1..jobs;
int: tasks;                                     % 每个作业的任务数量
set of int: TASK = 1..tasks;
array [JOB,TASK] of int: d;                     % 任务持续时间
int: total = sum(i in JOB, j in TASK)(d[i,j]);% 总持续时间
int: digs = ceil(log(10.0,total));              % 输出的数值
array [JOB,TASK] of var 0..total: s;            % 起始时间
var 0..total: end;                              % 总结束时间

constraint %% 保证任务按照顺序出现
    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 %% 保证任务之间没有重叠
    forall(j in TASK) (
        disjunctive([s[i,j] | i in JOB], [d[i,j] | i in JOB])
    );
```

```
solve minimize end;

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

约束 disjunctive 获取每个任务的开始时间数组以及它们的持续时间数组，确保每次只有一个任务是被激活的。我们定义析取约束为一个有以下特征的 谓词

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

在 列表 3.8 中，我们可以用这个析取约束定义任务之间不重叠。我们假设 disjunctive 谓词的定义已经在模型中引用的文件 disjunctive.mzn 中给出。

如果下层的系统直接支持 disjunctive ，则会在它的全局目录下包含一个 disjunctive.mzn 文件（拥有上述特征定义内容）。如果我们使用的系统不直接支持析取，通过创建文件 disjunctive.mzn ，我们可以给出我们自己的定义。最简单的实现是单单使用上面定义的 no_overlap 谓词。一个更好的实现是利用全局约束 cumulative ，假如下层求解器支持它的话。列表 3.9 给出了一个 disjunctive 的实现。注意我们使用 index_set 反射函数来（a）检查 disjunctive 的参数是有意义的，以及（b）构建 cumulative 的合适大小的资源利用数组。另外注意这里我们使用了 assert 的三元组版本。

列表 3.9: 使用 cumulative 来定义一个 disjunctive 谓词 (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 局部变量

在谓词，函数或者测试中，引进 局部变量总是非常有用的。表达式 let 允许你去这样做。它可以被用来引进决策变量 决策变量和 参数，但是参数必须被初始化。例如：

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

引进了参数 l 和 u 以及变量 y 。let 表达式虽然在 谓词，函数和测试定义中最有用，它也可以被用在其他的表达式中。例如，来消除共同的子表达式：

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

局部变量可以被用在任何地方，在简化复杂表达式时也很有用。通过使用局部变量来定义目标 objective 函数而不是显式地加入很多个变量，列表 3.10 给出了稳定婚姻模型的一个改进版本。

列表 3.10: 使用局部变量来定义一个复杂的目标函数 (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; % 客人的座位

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 语境

有一个局限，即含有决策变量并且在声明时没有初始化的谓词和函数不可以被用在一个否定语境下。下面例子是非法的

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

constraint not even(z);
```

原因是求解器只解决存在约束的问题。如果我们在否定语境下引入了一个局部变量，则此变量是 普遍地量化了，因此超出下层求解器的解决范围。例如，$\neg even(z)$ 等价于 $\neg\exists y.z = 2y$ ，然后等价于 $\forall y.z \neq 2y$ 。

如果局部变量被赋了值，则它们可以被用在否定语境中。下面的例子是合法的

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

constraint not even(z);
```

注意，现在 even 的意思是正确的，因为如果 x 是偶数，则 $x = 2*(x \text{ div } 2)$ 。由于 math:y 被 z 功能性定义了，$\neg even(z)$ 等价于 $\neg\exists y.y = z \text{ div } 2 \wedge z = 2y$ ，同时等价于 $\exists y.y = z \text{ div } 2 \wedge \neg z \neq 2y$。

MiniZinc 中的任意表达式都出现在以下四种 语境中的一种中：根，肯定，否定，或者 混合。非布尔型表达式的语境直接地为包含其最近的布尔型表达式的语境。唯一的例外是目标表达式出现在一个根语境下（由于它没有包含其的布尔型表达式）。

为了方便定义语境，我们把蕴含表达式 `e1 -> e2` 等价地写为 `not e1 \/ e2` ， `e1 <- e2` 等价地写为 `e1 \/ not e2` 。

一个布尔型表达式的语境可能有：

根

根语境是任何作为 `constraint` 的参数或者作为一个赋值项 assignment 出现的表达式 e 的语境，或者作为一个出现在根语境中的 `e1 /\ e2` 的子表达式 `e1` 或 `e2` 的语境。

根语境下的布尔型表达式必须在问题的任何模型中都满足。

肯定

肯定语境是任何作为一个出现在根语境或者肯定语境中的 `e1 \/ e2` 的子表达式 `e1` 或 `e2` 的语境，或者是作为一个出现在肯定语境中的 `e1 /\ e2` 的子表达式 `e1` 或 `e2``的语境，或者是作为一个出现在否定语境中的 :mzn:`not e` 的子表达式 `e` 的语境。

肯定语境下的布尔型表达式不是必须要在模型中满足，但是满足它们会增加包含其的约束被满足的可能性。对于一个肯定语境下的表达式，从包含其的根语境到此表达式有偶数个否定。

否定

否定语境是任何作为一个出现在根语境或者否定语境中的 `e1 \/ e2` 或 `e1 /\ e2` 的子表达式 `e1` 或 `e2` ，或者是作为一个出现在肯定语境中的 `not e` 的子表达式 `e` 的语境。

否定语境下的布尔型表达式不是必须要满足，但是让它们成假会增加包含其的约束被满足的可能性。对于一个否定语境下的表达式，从包含其的根语境到此表达式有奇数个否定。

混合

混合语境是任何作为一个出现在 `e1 <-> e2` , `e1 = e2` 或者 `bool2int(e)` 中的子表达式 `e1` 或 `e2` 的语境。

混合语境下的表达式实际上既是肯定也是否定的。通过以下可以看出：`e1 <-> e2` 等价于 `(e1 /\ e2) \/ (not e1 /\ not e2)` 以及 `x = bool2int(e)` 等价于 `(e /\ x=1) \/ (not e /\ x=0)` 。

观察以下代码段

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

其中 `x > 0` 在根语境中，`i <= 4}` 在否定语境中，`x + bool2int(x > i) = 5` 在肯定语境中，`x > i` 在混合语境中。

2.3.7 局部约束

Let 表达式也可以被用来引入局部约束，通常用来约束局部变量的行为。例如，考虑只利用乘法来定义开根号函数：

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

局部约束确保了 y 取正确的值；而此值则会被函数返回。

局部约束可以在 let 表达式中使用，尽管最普遍的应用是在定义函数时。

Let 表达式

局部变量可以在任何以下格式的 emph{let 表达式} 中引入：

```
let { <声明>; ... <声明> ; } in <表达式>
```

声明 <dec> 可以是决策变量或者参数（此时必须被初始化）或者约束项的声明。任何声明都不能在一个新的声明变量还没有引进时使用它。
注意局部变量和约束不可以出现在测试中。局部变量不可以出现在 否定或者 混合语境下的谓词和函数中，除非这个变量是用表达式定义的。

2.3.8 定义域反射函数

其他重要的反射函数有允许我们对变量定义域进行访问的函数。表达式 lb(x) 返回一个小于等于 x 在一个问题的解中可能取的值的数值。通常它会是 x 声明的下 限。如果 x 被声明为一个非有限类型，例如，只是 var int ，则它是错误的。类似地，表达式 dom(x) 返回一个 x 在问题的任何解中的可能值的（非严格）超集。再次，它通常是声明的值，如果它不是被声明为有限则会出现错误。

列表 3.11: 使用反射谓词 (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"];
```

例如，列表 3.11 中的模型或者输出

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

或

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

或任何满足 $-10 \leq y \leq 0$ 和 $\{0, \ldots, 4\} \subseteq D \subseteq \{-10, \ldots, 10\}$ 的答案。

变量定义域反射表达式应该以在任何安全近似下都正确的的方式使用。但是注意这个是没有被检查的！例如加入额外的代码

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

不是一个定义域信息的正确应用。因为使用更紧密（正确的）近似会比使用更弱的初始近似产生更多的解。

定义域反射

我们有查询包含变量的表达式的可能值的反射函数：

- dom(<表达式>) 返回 <表达式> 所有可能值的安全近似。
- lb(<表达式>) 返回 <表达式> 下限值的安全近似。
- ub(<表达式>) 返回 <表达式> 上限值的安全近似。

lb 和 ub 的表达式必须是 int ， bool ， float 或者 set of int 类型。dom 中表达式的类型不能是 float 。如果 <表达式> 中的一个变量有一个 非有限声明类型（例如，var int 或 var float 类型），则会出现一个错误。

我们也有直接作用于表达式数组的版本（有类似的限制）：

- dom_array(<数组表达式>)：返回数组中出现的表达式的所有可能值的并集的一个安全近似。
- lb_array(<数组表达式>) 返回数组中出现的所有表达式的下限的安全近似。
- ub_array(<数组表达式>) 返回数组中出现的所有表达式的下限的安全近似。

谓词，局部变量和定义域反射的结合使得复杂全局约束通过分解定义变为可能。利用 列表 3.12 中的代码，我们可以定义 cumulative 约束的根据时间的分解。

列表 3.12: 利用分解来定义一个 谓词 (cumulative.mzn).

```
%-----------------------------------------------------------------------%
% 需要给出一个任务集合，其中起始时间为's'，
% 持续时间'd'以及资源需求量'r'，
% 任何时候需求量都不能超过
% 一个全局资源界限'b'。
% 假设：
% - 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]
```

```
                )
            )
        )
    );
```

这个分解利用 `lb` 和 `ub` 来决定任务可以执行的时间范围集合。接下来，它对 `times` 中的每个时间 `times` 都断言在此时间 `t` 激活的所有任务所需要的资源量总和小于界限 `b` 。

2.3.9 作用域

MiniZinc 中声明的作用域值得我们简单地介绍下。MiniZinc 只有一个作用域，所以出现在声明中的所有变量都可以在模型中的每个表达式中可见。用以下几个方式，MiniZinc 引进局部作用域变量：

- 推导式表达式中的 迭代器
- 使用 `let` 表达式
- 谓词和函数中的 参数

任何局部作用域变量都会覆盖同名称的外部作用域变量。

列表 3.13: 阐述变量作用域的模型 (`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;
```

例如，在 列表 3.13 中给出的模型中，`-x <= y` 中的 `x` 是全局 `x` ，`smallx(x)` 中的 `x` 是迭代器 `x in 1..u` ，而析取中的 `y` 是谓词的第二个参数。

CHAPTER 2.4

选项类型

选项类型是一个强大的抽象，使得简洁建模成为可能。一个选项类型决策变量代表了一个有其他可能 ⊤ 的变量，在 MiniZinc 中表达为 <> ，代表了这个变量是 缺失的。选项类型变量在建模一个包含在其他变量没做决定之前不会有意义的变量的问题时是很有用的。

2.4.1 声明和使用选项类型

选项类型变量

一个选项类型变量被声明为：

```
var opt <类型> : <变量名>:
```

其中 <类型> 是 `int` ， `float` 或 `bool` 中的一个，或者是一个固定范围的表达式。选项类型变量可以是参数，但是这个不常用。
一个选项类型变量可以有附加值 <> 表明它是 缺失的。
三个内建函数被提供给选项类型变量：`absent(v)` 只有在选项类型变量 `v` 取值 <> 时返回 `true` ， `occurs(v)` 只有在选项类型变量 `v` 不取值 <> 时返回 `true` ，以及 <> 返回 `v` 的正常值或者当它取值 <> 时返回失败。

选项类型最常被用到的地方是调度中的可选择任务。在灵活的车间作业调度问题中，我们有 `n` 个在 `k` 个机器上执行的任务，其中完成每个机器上每个任务的时间可能是不一样的。我们的目标是最小化所有任务的总完成时间。一个使用选项类型来描述问题的模型在 列表 4.1 中给出。在建模这个问题的时候，我们使用 $n \times k$ 个可选择的任务来代表每个机器上每个任务的可能性。我们使用 `alternative` 全局约束来要求任务的起始时间和它的持续时间跨越了组成它的可选择任务的时间，同时要求只有一个会实际运行。我们使用 `disjunctive` 全局变量在每个机器上最多有一个任务在运行，这里我们延伸到可选择的任务。最后我们约束任何时候最多有 k 个任务在运行，利用一个在实际（不是可选择的）任务上作用的冗余约束。

列表 4.1: 使用选项类型的灵活车间作业调度模型 (flexible-js.mzn).

```
include "globals.mzn";

int: horizon;                    % 时间范围
set of int: Time = 0..horizon;
enum Task;
enum Machine;

array[Task,Machine] of int: d; % 每个机器上的持续时间
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;              % 起始时间
array[Task] of var mind..maxd: D;        % 持续时间
array[Task,Machine] of var opt Time: O; % 可选择的任务起始

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

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

2.4.2 隐藏选项类型

当列表推导式是从在变量集合迭代上创建而来，或者 `where` 从句中的表达式还没有固定时，选项类型变量会隐式地出现。

例如，模型片段

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

是以下的语法糖

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

内建函数 `sum` 实际上在一列类型-实例化 `var opt int` 上操作。由于 <> 在 + 中表现为标识 0，我们会得到期望的结果。

类似地，模型片段

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

是以下的语法糖

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

同样地，函数 forall 实际上在一列类型-实例化 var opt bool 上操作。由于 <> 在 /\ 上表现为标识 true ，我们可以得到期望的结果。

尽管我们已经很小心了，隐式的使用可能会导致意外的行为。观察

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

它本应该是一个不可满足的问题。它返回 x = {1,2,3,4} 作为一个解例子。这个是正确的因为第二个约束等于

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

而可选择整数列表的长度总会是 9，所以这个约束总是会满足。

我们可以通过不在变量集合上创建迭代或者使用不固定的 where 从句来避免隐式的选项类型。例如，上面的两个例子可以不使用选项类型重写为

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

和

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

CHAPTER 2.5

Tuple and record types

In MiniZinc models we often deal with multiple points of data and multiple decisions that all concern the same thing, an "object". There are multiple ways of modelling this. It is common practice in MiniZinc to have different arrays for different types of data points or decisions. These arrays will then share a common index mapping to which object each data point belongs.

In this section we present an alternative, "object-oriented", approach in MiniZinc in the form of tuple and record types. Using these types, different types of data points and decisions can be combined into collections for the modeller's convenience.

2.5.1 Declaring and using tuples and records

Tuple types and literals

A tuple type variable is declared as:

```
<var-par> tuple(<ti-expr>, ...): <var-name>
```

where one or more type instantiations, `<ti-expr>`, are placed between the parentheses to declare the types contained in the tuple variable.
For convenience the `var` keyword can be used to varify all the member types of the tuple (i.e., `var tuple(int, bool)` is the same type as `tuple(var int, var bool)`).
A tuple literal is created using the following syntax:

```
(<expr>, [ <expr>, ... ])
```

For example, `(1, true)` is a tuple literal of type `tuple(int,bool)`. Note that when a tuple contains multiple members, then adding a trailing comma is optional, but when it contains a single member it is **required** to distinguish the literal from a normal parenthesized expression.

Tuples provide a very simple way to create a collection that contains values of different types. In a tuple variable, the values contained in the tuple can be accessed using a number representing the place in the tuple. For example, in the tuple `any: x = (1, true, 2.0)` the first member, `1`, can be retrieved using `x.1`, and the final member, `2.0`, can be retrieved using `x.3`. Note that the MiniZinc compiler will raise a **Type Error**, when an integer is used that is lower than one or higher than the number of members in the tuple.

Although tuple types can be useful, it can often be confusing which member represents what. Record types improve on this by associating a name with each member of the type.

Record types and literals

A record type variable is declared as:

```
<var-par> record(<ti-expr-and-id>, ...): <var-name>
```

where one or more type instantiations with a corresponding identifier, `<ti-expr-and-id>`, are placed between the parentheses to declare the types contained in the record variable.
For convenience the `var` keyword can be used to varify all the member types of the record (i.e., `var record(int: i, bool: b)` is the same type as `record(var int: i, var bool: b)`).
A record literal is created using the following syntax:

```
(<ident>: <expr>[, ... ])
```

For example, `(i: 1, b: true)` is a record literal of type `record(int: i, bool: b)`. Different from tuples, record literals with only a single member do not require a trailing comma.

The syntax for accessing a member of a record is very similar to accessing a member of a tuple. The difference is that instead of a number, we use the name given to the member. For example, given the record `any: x = (i: 1, b: true)`, the integer value `1` that is named by identifier `i` is retrieved using `x.i`. Using identifiers that do not name a member (or any number) will once more result in a **Type Error**.

2.5.2 Using type-inst synonyms

When using records and tuples, writing the types in many places can quickly become tedious and confusing. Additionally, it might often make sense to give a name to such a type-inst to describe its meaning. For this purpose, you can use type-inst synonyms in MiniZinc.

Type-inst synonyms

A type-inst synonym is declared as:

```
type <ident> <annotations> = <ti-expr>;
```

where the identifier `<ident>` can be used instead of the type-inst `<ti-expr>` where required.

For example, in the following MiniZinc fragment we declare two synonyms, `Coord` and `Number`

```
type Coord = var record(int: x, int: y, int: z);
type Number = int;
```

In a model that contains these definitions, we can now declare a variable `array[1..10] of Coord: placement;` or a function `function Number: add(Number: x, Number: y) = x + y;`

Similar to record and tuple types, the `var` keyword can be used before the identifier of a type-inst synonym to varify the type-inst. For instance, given the synonym `type OnOff = bool;` and the variable declaration `var OnOff: choice;`, the type-inst of `choice` would be `var bool`. Different from records and tuple, the reverse is also possible using the `par` keyword. For instance, the given the synonym `type Choice = var bool;` and the variable declaration `par Choice: check = fix(choice);` the type-inst of `check` would be `bool`.

2.5.3 Types with both var and par members

Tuples and records can be used to collect both data and decisions about an object, and it can be natural to mix data and decisions that concern the same object. However, when data and decisions are contained within the same tuple or record type, initialization of these types can be complex. To avoid the usage of anonymous variables or let-expressions and allow the assignment from data files, it is recommended to split data and decisions into separate types.

For example, in a rostering problem it would be natural to combine information about the employees, both data and decisions. It might thus be natural to define the following synonyms and variable declarations.

```
enum EmpId;
type Employee = record(
  string: name,
  array[Timespan] of bool: available,
  set of Capability: capacities,
  array[Timespan] of var Shift: shifts,
  var 0..infinity: hours,
);

array[EmpId] of Employee: employee;
```

However, it is not possible to assign `employees` from a data file. Instead, the data and the

decisions could be split as follows.

```
enum EmpId;
type EmployeeData = record(
  string: name,
  array[Timespan] of bool: available,
  set of Capability: capacities,
);
type EmployeeVar = record(
  array[Timespan] of var Shift: shifts,
  var 0..infinity: hours,
);

array[EmpId] of EmployeeData: employee_data;
array[EmpId] of EmployeeVar: employee_var;
```

Now it is possible to initialize `employee_data` from a data file. Depending on the model, it might be easiest to use these synonyms and variable declarations separately, but it might be useful to combine them again. For instance, when your model contains functions that should operate on the complete record. The `++` operator can be used to easily combine these synonyms and values, as shown in the following fragment.

```
type Employee = EmployeeData ++ EmployeeVar;

array[EmpId] of Employee: employee = [employee_data[id] ++ employee_var[id] | id in␣
↪EmpId];
```

++ operator for tuples and records

The ++ operator can be used to combine expressions and type-inst expressions that both have tuple types or both have record types.

```
<expr> ++ <expr>
<ti-expr> ++ <ti-expr>
```

When both (type-inst) expressions have a tuple type, this operation is very similar to array concatenation. The result of evaluating the expression will be a new tuple that contains all members of left-hand side followed by all the members of the right-hand side.
When both (type-inst) expressions have a record type, then the result of evaluating the expression is a record that merges the fields in both records. Note that if both records contain a field with the same name, then the compiler will reject this as a Type Error.

2.5.4 Example: rectangle packing using records

Consider a packing problem where a collection of rectangular objects have to be packed into a rectangular space. Importantly, none of the rectangles being packed can overlap. To keep the model simple, we don't allow the rectangles to be turned. When we create a model to decide whether the packing is possible, we can use records to help us make

the model easier to read.

The data for the model comes in two parts: the total area of the space and the sizes of the rectangles being packed. We can represent both using a "dimensions" record that includes the width and the height of a rectangle. We therefore define the following type-inst synonym.

```
type Dimensions = record(int: width, int: height);
```

To decide whether the problem is satisfiable or not, we will have to find a placement for the rectangles being packed. We can thus use the coordinates of their left bottom corner as our decision variables, for which a type-inst synonym can be defined as follows.

```
type Coordinates = record(var 0..infinity: x, var 0..infinity: y);
```

To enforce that no rectangles overlap, it can be easy to start by writing a predicate that ensures that two rectangles do not overlap. This predicate will require both the information about the dimensions of the rectangle and the coordinate decision variable. As such, it can be valuable to first combine these records into a new record type. The following fragment shows the additional type-inst synonym, and a definition for the described predicate.

```
type Rectangle = Dimensions ++ Coordinates;

% No overlap predicate
predicate no_overlap(Rectangle: rectA, Rectangle: rectB) =
  rectA.x + rectA.width <= rectB.x
  \/ rectB.x + rectB.width <= rectA.x
  \/ rectA.y + rectA.height <= rectB.y
  \/ rectB.y + rectB.height <= rectA.y;
```

Using these definitions the remainder of the model is now easy to define as shown below.

列表 5.1: Partial rectangle packing model using record types (full model: `rect_packing.mzn` example data: `rect_packing.json`).

```
% Instance data
array[_] of Dimensions: rectDim;
Dimensions: area;

% Decision variables
array[index_set(rectDim)] of Coordinates: rectCoord ::no_output;

array[_] of Rectangle: rectangles ::output = [ rectDim[i] ++ rectCoord[i] | i in␣
↪index_set(rectDim)];

% Constraint: rectangles must be placed within the area
constraint forall(rect in rectangles) (
  rect.x + rect.width <= area.width
  /\ rect.y + rect.height <= area.height
);

% Constraint: no rectangles can overlap
```

```
constraint forall(i, j in index_set(rectangles) where i < j) (
  no_overlap(rectangles[i], rectangles[j])
);
```

CHAPTER 2.6

搜索

MiniZinc 默认没有我们想如何搜索解的声明。这就把搜索全部都留给下层的求解器了。但是有些时候，尤其是对组合整数问题，我们或许想规定搜索应该如何去进行。这就需要我们和求解器沟通出一个搜索策略 search 。注意，搜索策略 不真的是模型的一部分。实际上，我们不要求每个求解器把所有可能的求解策略都实现了。MiniZinc 通过使用 *annotations* 来用一个稳定的方法跟约束求解器沟通额外的信息。

2.6.1 有限域搜索

利用有限域求解器搜索涉及到检查变量剩余的可能值以及选择进一步地约束一些变量。搜索则会加一个新的约束来限制这个变量的剩余值（实际上猜测解可能存在于哪里），然后使用传播来确定其他的值是否可能存在于解中。为了确保完全性，搜索会保留另外一个选择，而它是新约束的否定。搜索会当有限域求解器发现所有的约束都被满足，此时一个解已经被找到，或者有约束不被满足时停止。当不可满足出现的时候，搜索必须换另外一个不同的选择集合继续下去。通常有限域求解器使用 深度优先搜索，它会撤销最后一个做的选择然后尝试做一个新的选择。

列表 6.1: n 皇后问题模型 (nqueens.mzn).

```
int: n;
array [1..n] of var 1..n: q; % i列的皇后在行q[i]

include "alldifferent.mzn";

constraint alldifferent(q);                        % 不同行
constraint alldifferent([ q[i] + i | i in 1..n]); % 不同对角线
constraint alldifferent([ q[i] - i | i in 1..n]); % 上+下

% 搜索
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]
```

有限域问题的一个简单例子是 n 皇后问题，它要求我们放置 n 个皇后在 $n \times n$ 棋盘上使得任何两个都不会互相攻击。变量 `q[i]` 记录了在 `i` 列的皇后放置在哪一行上。`alldifferent` 约束确保了任何两个皇后都不会在同一行或者对角线上。图 6.1 的左边给出了 `n = 9` 的典型（部分）搜索树。我们首选设置 `q[1] = 1` ，这样就可以从其他变量的定义域里面移除一些数值，例如 `q[2]` 不能取值 1 或者 2. 我们接下来设置 `q[2] = 3` ，然后进一步地从其他变量的定义域里面移除一些数值。我们设置 `q[3] = 5` （它最早的可能值）。在这三个决策后，棋盘的状态显示为 图 6.2 。其中皇后表示已经固定的皇后位置。星星表示此处放置的皇后会攻击到已经放置的皇后，所以我们不能在此处放置皇后。

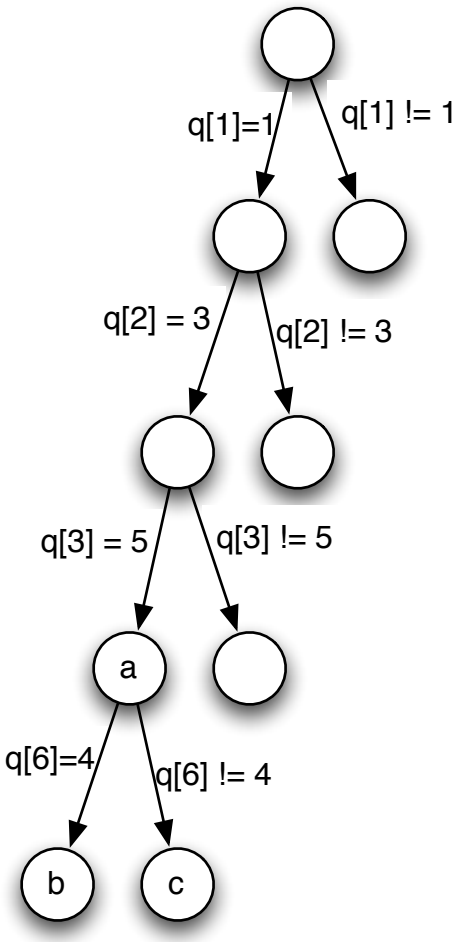


图 6.1: 9 皇后问题的部分搜索树

一个搜索策略决定要做哪一个选择。我们目前所做的决定都按照一个简单的策略：选择第一个还没有固定的变量，尝试设置它为它的最小可能值。按照这个策略，下一个决策应该是 `q[4] = 2` 。变量选择的另外一个策略是选择现在可能值集合 定义域最小的变量。按照这个所谓 最先失败 变量选择策略，下一个决策应该是 `q[6] = 4` 。如果我们做了这个决策，则初始的传播会去除掉 图 6.3 中显示的额外的值。但是它使得 `q[8]` 只剩余有一个值。所以 `q[8] = 7` 被执行。但是这又使得 `q[7]` 和 `q[9]` 也只剩余一个值 2。因此这个时候有个约束一定会被违反。我们检测到了不满足性，求解器必须回溯取消最后一个决策 `q[6] = 4` 并且加入它的否定 `q[6] != 4` （引导我们到了 图 6.1 中树的状态 (c)），即强制使 `q[6] = 9` 。这使得有些值从定义域中被去除。我们接下来继续重新启用搜索策略来决定该怎么做。

很多有限域搜索被定义为这种方式：选择一个变量来进一步约束，然后选择如何进一步地约束它。

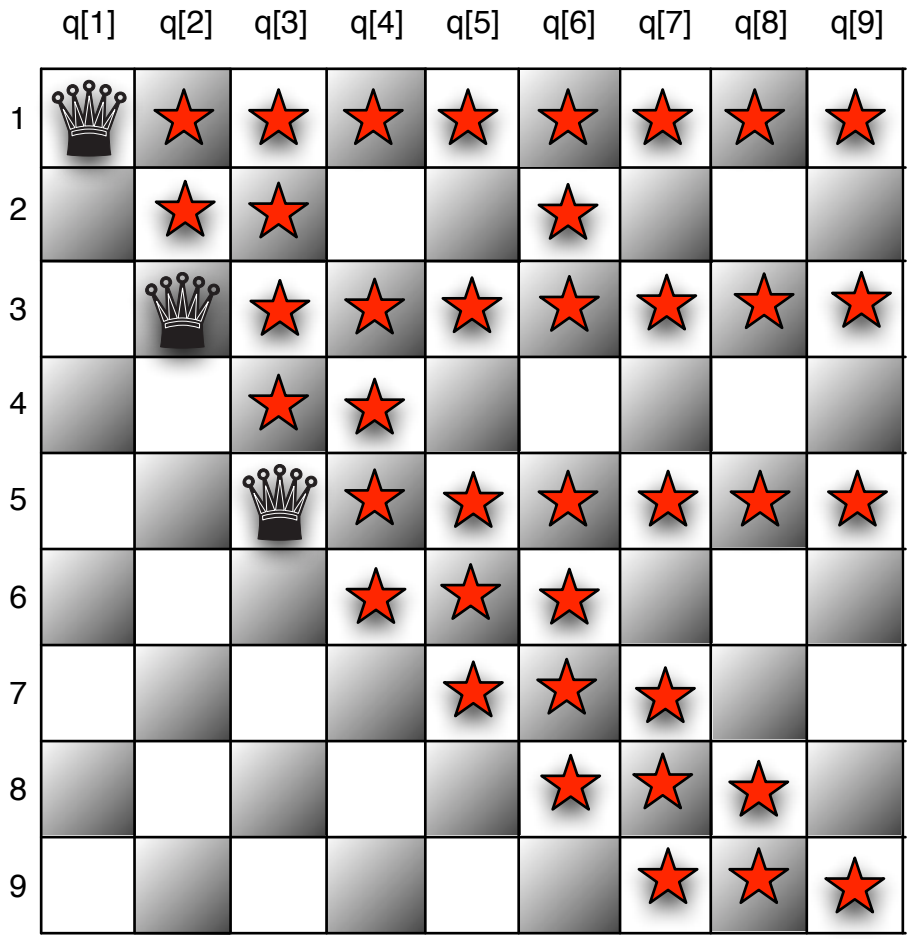


图 6.2: 在加入 `q[1] = 1`, `q[2] = 4`, `q[3] = 5` 的状态

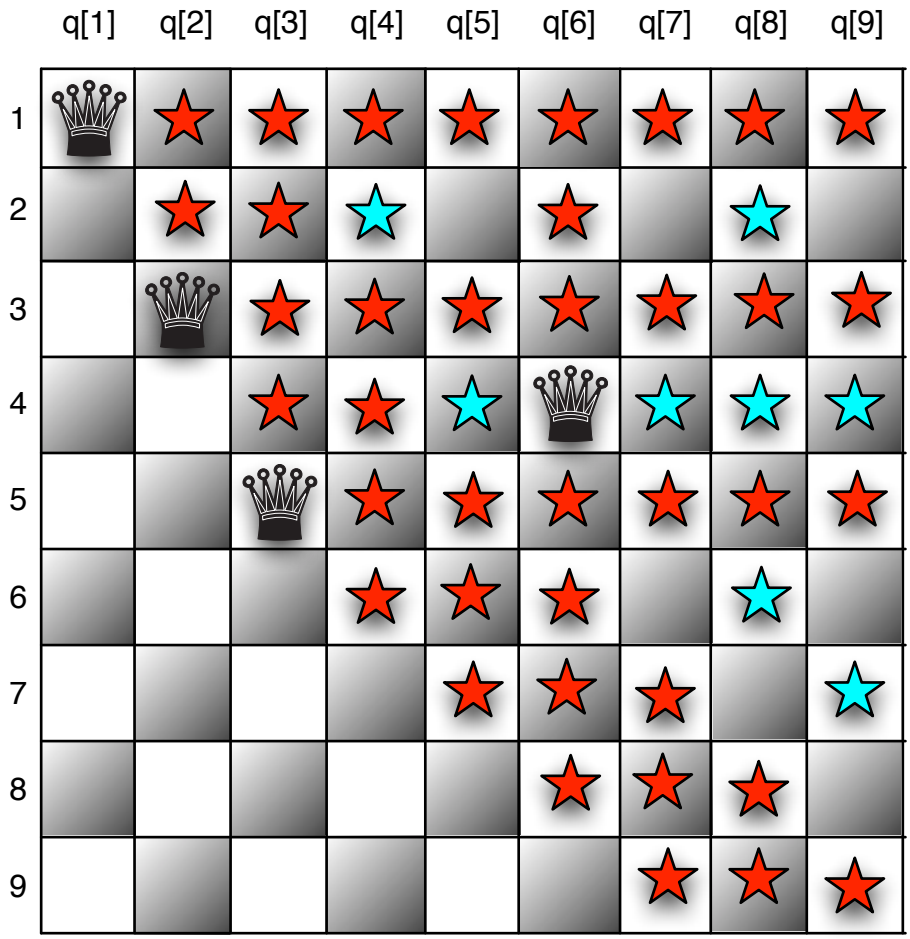


图 6.3: 在进一步加入 `q[6] = 4` 后的初始传播

2.6.2 搜索注解

MiniZinc 中的搜索注解注明了为了找到一个问题的解应如何去搜索。注解附在求解项，在关键字 solve 之后。搜索注解

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

出现在求解项中。注解使用连接符 :: 附为模型的一部分。这个搜索注解意思是我们应该按照从整型变量数组 q 中选择拥有最小现行定义域的变量（这个是 first_fail 规则），然后尝试设置其为它的最小可能值（indomain_min 值选择），纵观整个搜索树来搜索（complete 搜索）。

基本搜索注解

我们有三个基本搜索注解，相对应于不同的基本搜索类型：

- int_search(<变量>, <变量选择>, <约束选择>, <策略>) 其中 <变量> 是一个 var int 类型的一维数组，<变量选择> 是一个接下来会讨论的变量选择注解，<约束选择> 是一个接下来会讨论的如何约束一个变量的选择，<策略> 是一个搜索策略，我们暂时假设为 complete
- bool_search(<变量>, <变量选择>, <约束选择>, <策略>) 其中 <变量> 是一个 var bool 类型的一维数组，剩余的和上面一样。
- set_search(<变量>, <变量选择>, <约束选择>, <策略>) 其中 <变量> 是一个 var set of int 类型的一维数组，剩余的和上面一样。
- float_search(<变量>, <精度>, <变量选择>, <约束选择>, <策略>) 其中 <变量> 是一个一维 var float 数组，<precision> 是一个固定的用于表示 ϵ 浮点数，其中两个数之差低于这个浮点数时被认为相等。剩余的和上面一样。

变量选择注解的例子有：

- input_order: 从数组中按照顺序选择
- first_fail: 选择拥有最小定义域大小的变量，以及
- smallest: 选择拥有最小值的变量。

约束一个变量的方式有：

- indomain_min: 赋最小的定义域内的值给变量，
- indomain_median: 赋定义域内的中间值给变量，
- indomain_random: 从定义域中取一个随机的值赋给变量，以及
- indomain_split 把变量定义域一分为二然后去除掉上半部分。

对于完全搜索，<策略> 基本都是 complete 。关于一份完整的变量和约束选择注解，请参看 MiniZinc 参考文档中的 FlatZinc 说明书。

利用搜索构造注解，我们可以创建更加复杂的搜索策略。目前我们只有一个这样的注解。

```
seq_search([ <搜索注解>, ..., <搜索注解> ])
```

顺序搜索构造首先执行对列表中的第一个注解所指定的变量的搜索，当这个注解中的所有的变量都固定后，它执行第二个搜索注解，等等。直到所有的搜索注解都完成。

我们来看一下 列表 3.8 中给出的车间作业调度模型。我们可以替换求解项为

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

通过选择可以最早开始的作业并设置其为 s ，起始时间被设置为 s 。当所有的起始时间都设置完后，终止时间 end 或许还没有固定。因此我们设置其为它的最小可能取值。

2.6.3 注解

在 MiniZinc 中，注解是第一类对象。我们可以在模型中声明新的注解，以及声明和赋值给注解变量。

注解

注解有一个类型 ann 。你可以声明一个注解参数 parameter （拥有可选择的赋值）：

```
ann : <标识符>;
ann : <标识符> = <注解表达式> ;
```

对注解变量赋值和对其他参数赋值一样操作。
表达式，变量声明，和 solve 项都可以通过使用 :: 操作符来成为注解。
使用注解项 注解项，我们可以声明一个新的注解 annotation 项:

```
annotation <注解名> ( <参数定义>, ..., <参数定义> ) ;
```

列表 6.2: n 皇后问题的注解模型 (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_propagation;
constraint alldifferent([ q[i] + i | i in 1..n]) :: domain_propagation;
constraint alldifferent([ q[i] - i | i in 1..n]) :: domain_propagation;

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];
```

列表 6.2 中的程序阐述了注解声明，注解和注解变量的使用。我们声明一个新的注解 bitdomain ，意思是来告诉求解器变量定义域应该通过大小为 nwords 的比特数组来表示。注解附注在变量 q 的声明之后。每一个 alldifferent 约束都被注解为内部注解 domain ，而它指导求解器去使用 alldifferent 的定义域传播版本（如果有的话）。一个注解变量 search_ann 被声明和使用来定义搜索策略。我们可以在一个单独的数据文件中来给出搜索策略的值。

搜索注解的例子或许有以下几种（我们假设每一行都在一个单独的数据文件中）

```
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);
```

第一个只是按顺序来选择皇后然后设置其为最小值。第二个按顺序来选择皇后，但是设置中间值给它。第三个选择定义域大小最小的皇后，然后设置最小值给它。最后一个策略选择定义域大小最小的皇后，设置中间值给它。

不同的搜索策略对于能多容易找到解有显著的差异。下面的表格给出了一个简单的关于使用 4 种不同的搜索策略找到 n 皇后问题的第一个解所要做的决策个数（其中---表示超过 100,000 个决策）。很明显地看到，合适的搜索策略会产生显著的提高。

n	input-min	input-median	ff-min	ff-median
10	28	15	16	20
15	248	34	23	15
20	37330	97	114	43
25	7271	846	2637	80
30	---	385	1095	639
35	---	4831	---	240
40	---	---	---	236

CHAPTER 2.7

MiniZinc 中的有效建模实践

对同一个问题，几乎总是存在多种方式来建模。其中一些产生的模型可以很有效地求解，另外一些则不是。通常情况下，我们很难提前判断哪个模型是对解决一个特定的问题最有效的。事实上，这或许十分依赖于我们使用的底层求解器。在这一章中，我们专注于建模实践，来避免产生模型的过程和产生的模型低效。

2.7.1 变量界限

有限域传播器，是 MiniZinc 所针对的求解器中的核心类型。在当其所涉及到的变量的界限越紧凑时，此传播器越有效。它也会当问题含有会取很大整型数值的子表达式时表现得很差，因为它们可能会隐式地限制整型变量的大小。

列表 7.1: 没有无界整数的模型 (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"];
```

在 列表 7.1 中的中的杂货店问题要找寻 4 个物品使得它们的价格加起来有 7.11 元并且乘起来也有 7.11 元。变量被声明为无界限。运行

```
$ minizinc --solver g12fd grocery.mzn
```

得到

```
=====UNSATISFIABLE=====
% /tmp/mznfile85EWzj.fzn:11: warning: model inconsistency detected before search.
```

这是因为乘法中的中间表达式的类型也会是 `var int` ，也会被求解器给一个默认的界限 $-1,000,000 \ldots 1,000,000$ 。但是这个范围太小了以至于不能承载住中间表达式所可能取的值。

更改模型使得初始变量都被声明为拥有更紧致的界限

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

我们得到一个更好的模型，因为现在 MiniZinc 可以推断出中间表达式的界限，并且使用此界限而不是默认的界限。在做此更改后，求解模型我们得到

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

注意，就算是改善的模型也可能对于某些求解器来说会很难解决。运行

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

不能得到任何结果，因为求解器给中间产生的变量创建了巨大的表示。

给变量加界限

在模型中要尽量使用有界限的变量。当使用 `let` 声明来引进新的变量时，始终尽量给它们定义正确的和紧凑的界限。这会使得你的模型更有效率，避免出现意外溢出的可能性。一个例外是当你引进一个新的变量然后立刻定义它等于一个表达式，通常 MiniZinc 都可以从此表达式推断出此变量有效的界限。

2.7.2 有效的生成元

想象下我们想要计算在一个图中出现的三角形的个数（K_3 子图）。假设此图由一个邻接矩阵定义：如果点 i 和 j 邻接，则 `adj[i,j]` 为真。我们或许可以写成

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

这当然是对的，但是它检查了所有点可能组成的三元组。如果此图是稀疏的，在意识到一旦我们选择了 `i` 和 `j` ，就可以进行一些测试之后，我们可以做得更好。

```
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]]);
```

你可以使用内建 trace 函数来帮助决定在生成元内发生了什么。

追踪

函数 trace(s,e) 在对表达式 e 求值并返回它的值之前就输出字符串 s 。它可以在任何情境下使用。

例如，我们可以查看在两种计算方式下的内部循环中分别进行了多少次测试。

```
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;
```

得到输出：

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

表示内部循环进行了 64 次，而

```
  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])]);
```

得到输出

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

表示内部循环进行了 16 次。

注意你可以在 trace 中使用单独的字符串来帮助你理解模型创建过程中发生了什么。

```
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]]));
```

会输出在计算过程中找到的每个三角形。得到输出

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

我们要承认这里我们有一点点作弊: 在某些情况下, MiniZinc 编译器实际上会把 where 从句中的参数自动重新排序, 所以他们会尽快地被计算。在这种情况下, 加入 trace 函数实际上 阻止了这种优化. 一般来说, 通过分离 where 从句把它们摆到尽量接近生成元, 这其实是一个很好的主意来帮助编译器运作正常。

2.7.3 冗余约束

模型的形式会影响约束求解器求解它的效率。在很多情况下加入冗余约束，即被现有的模型逻辑上隐含的约束，可能会让求解器在更早时候产生更多可用的信息从而提高找寻解的搜索效率。

回顾下第复杂约束 (page 46) 节中的魔术串问题。

运行 n = 16 时的模型:

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

n = 16

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

统计数据显示需要有 89 个失败。

我们可以在模型中加入冗余约束。由于序列中的每个数字是用来计算某一个数字出现的次数，我们知道它们的和肯定是 n 。类似地，由于这个序列是魔术的，我们知道 s[i] * i 的和肯定也是 n 。使用如下方式把这些约束加入我们的模型 列表 7.2.

列表 7.2: 使用冗余约束求解魔术串问题模型 (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))));

constraint redundant_constraint(sum(i in 0..n-1)(s[i]) = n);
constraint redundant_constraint(sum(i in 0..n-1)(s[i] * i) = n);

solve satisfy;

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

像之前那样求解同一个问题

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

产生了同样的输出。但是统计显示只搜索了 13 个决策点。这些冗余约束使得求解器更早地去剪枝。

2.7.4 模型选择

在 MiniZinc 中有很多方式去给同一个问题建模，尽管其中有些模型或许会比另外的一些模型更自然。不同的模型或许会产生不同的求解效率。更糟糕的是，在不同的求解后端中，不同的模型或许会更好或更差。但是，我们还是可以给出一些关于普遍情况下产生更好的模型的指导:

模型之间的选择

一个好的模型倾向于有以下特征

- 更少量的变量，或者至少是更少量的没有被其他变量功能上定义的变量。
- 更小的变量定义域范围
- 模型的约束定义更简洁或者直接
- 尽可能地使用全局约束

实际情况中，所有这些都需要通过检查这个模型的搜索到底多有效率来断定模型好坏。通常除了用实验之外，我们很难判断搜索是否高效。

观察如下问题，我们要找寻 1 到 n 这 n 个数字的排列，使得相邻数字的差值也形成一个 1 到 n 的排列。列表 7.3 中给出了一个用直观的方式来建模此问题的模型。注意变量 `u` 被变量 `x` 功能性定义。所以最差情况下的搜索空间是 n^n。

列表 7.3: 对于 CSPlib `prob007` 所有间隔系列问题的模型 (`allinterval.mzn`).

```
include "alldifferent.mzn";

int: n;

array[1..n] of var 1..n: x;       % 数字的序列
array[1..n-1] of var 1..n-1: u;   % 差的序列

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 = ",show(x),"\n"];
```

在这个模型中，数组 `x` 代表 `n` 个数字的排序。约束自然地可用 `alldifferent` 来表示。

求解模型

```
$ minizinc --solver g12fd --all-solutions --statistics allinterval.mzn -D "n=10;"
```

在 84598 个决策点和 3 秒的时间内找到了所有的解。

另外一个模型是使用数组 `y` ，其中 `y[i]` 代表数字 `i` 在序列中的位置。我们同时也使用变量 `v` 来建模表示差的位置。`v[i]` 表示了绝对值差 `i` 在序列出现的位置。如果 `y[i]` 和 `y[j]` 差别为一，其中 `j > i` ，则代表了它们的位置是相邻的。所以 `v[j-i]` 被约束为两个位置中最早的那个。我们可以给这个模型加入两个冗余约束：由于我们知道差值 `n-1` 肯定会产生，我们就可以推断出 1 和 `n` 的位置必须是相邻的 `abs( y[1] - y[n] ) = 1` 。同时也告诉我们差值 `n-1` 的位置就是在 `y[1]` 和 `y[n]` 中的最早的那个位置，即 `v[n-1] = min(y[1], y[n])` 。有了这些之后，我们可以建模此问题为:numref:*ex-allint2* 。输出语句从位置数组 `y` 里重现了原本的序列 `x` 。

列表 7.4: CSPlib 中全区间序列问题 prob007 的一个逆向模型。(allinterval2.mzn).

```
include "alldifferent.mzn";

int: n;

array[1..n] of var 1..n: y;  % 每个数值的位置
array[1..n-1] of var 1..n-1: v; % 差值i的位置

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;

output [ "x = [",] ++
       [ show(i) ++ if j == n then "]\n;" else ", " endif
         | j in 1..n, i in 1..n where j == fix(y[i]) ];
```

逆向模型跟初始模型有同样的变量和定义域大小。但是相对于给变量 x 和 u 的关系建模，逆向模型使用了一个更加非直接的方式来给变量 y 和 v 的关系建模。所以我们或许期望初始模型更好些。

命令

```
$ minizinc --solver g12fd --all-solutions --statistics allinterval2.mzn -D "n=10;"
```

在 75536 个决策点和 18 秒内找到了所有的解。有趣的是，尽管这个模型不是简洁的，在变量 y 上搜索比在变量 x 上搜索更加有效率。简洁的缺乏意味着尽管搜索需要更少的决策点，但是在时间上实质上会更慢。

2.7.5 多重建模和连通

当我们对同一个问题有两个模型时，由于每个模型可以给求解器不同的信息，通过把两个模型中的变量系到一起从而同时使用两个模型或许对我们是有帮助的。

列表 7.5: CSPlib 中全区间序列问题 prob007 的一个双重模型。(allinterval3.mzn).

```
include "inverse.mzn";

int: n;

array[1..n] of var 1..n: x;  % 数值的序列
array[1..n-1] of var 1..n-1: u;  % 差值的序列

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

```
array[1..n] of var 1..n: y;  % 每个数值的位置
array[1..n-1] of var 1..n-1: v; % 差值的位置

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"];
```

列表 7.5 给出了一个结合 allinterval.mzn 和 allinterval2.mzn 特征的双重模型。模型的开始来自于 allinterval.mzn 。我们接着介绍了来自于 allinterval2.mzn 中的变量 y 和 v 。我们使用全局约束 inverse 来把变量绑到一起：inverse(x,y) 约束 y 为 x 的逆向函数（反之亦然），即，x[i] = j <-> y[j] = i 。列表 7.6 中给出了它的一个定义。这个模型没有包含把变量 y 和 v 关联起来的约束，它们是冗余的（实际上是传播冗余）。所以它们不会给基于传播的求解器多余的信息。alldifferent 也不见了。原因是它们被逆向约束变得冗余了（传播冗余）。唯一的约束是关于变量 x 和 u 和关系的约束以及 y 和 v 的冗余约束。

列表 7.6: 全局约束 inverse 的一个定义 (inverse.mzn).

```
predicate inverse(array[int] of var int: f,
                  array[int] of var int: invf) =
    forall(j in index_set(invf))(invf[j] in index_set(f)) /\
    forall(i in index_set(f))(
        f[i] in index_set(invf) /\
        forall(j in index_set(invf))(j == f[i] <-> i == invf[j])
    );
```

双重模型的一个优点是我们可以有更多的定义不同搜索策略的视角。运行双重模型，

```
$ minizinc --solver g12fd --all-solutions --statistics allinterval3.mzn -D "n=10;"
```

注意它使用逆向模型的搜索策略，标记变量 y ，在 1714 决策点和 0.5 秒内找到了所有的解。注意标记变量 x 来运行同样的模型，需要 13142 个决策点和 1.5 秒。

2.7.6 对称

对称在约束满足和优化问题中是常见的。我们可以再次通过 列表 6.1 来看一下这个问题。在 图 7.1 棋盘的左上方展示了一个 8 皇后问题的解 (标记为"original")。剩下的棋盘展示了 7 个解的对称版本: 旋转 90 度,180 度和 270 度, 还有垂直翻转。

如果我们想要穷举 8 皇后问题的 *所有解*, 很明显我们需要通过穷举 *彼此之间不对称的解*为求解器省下一些工作, 然后生成这些的对称版本。这是我们想要在约束模型中摆脱对称的一个理由。另外一个更重要的理由是, 我们的求解器也可能会 **探索非解状态的对称版本!**

举个例子, 一个典型的约束求解器可能会尝试把第 1 列皇后放在第 1 行上 (这是可以的), 然后尝试把第 2 列的皇后放到第 3 行上。这在第一眼看是没有违反任何约束的。然而, 这种设置不能被完成成为一个解 (求解器会在一些搜索之后发现这一点). 图 7.2 在左上方的棋盘展示了这种设置。现在没有任何东西会阻止求解器去尝试, 比如, 在 图 7.2 最低一行的左边第二种设置。其中

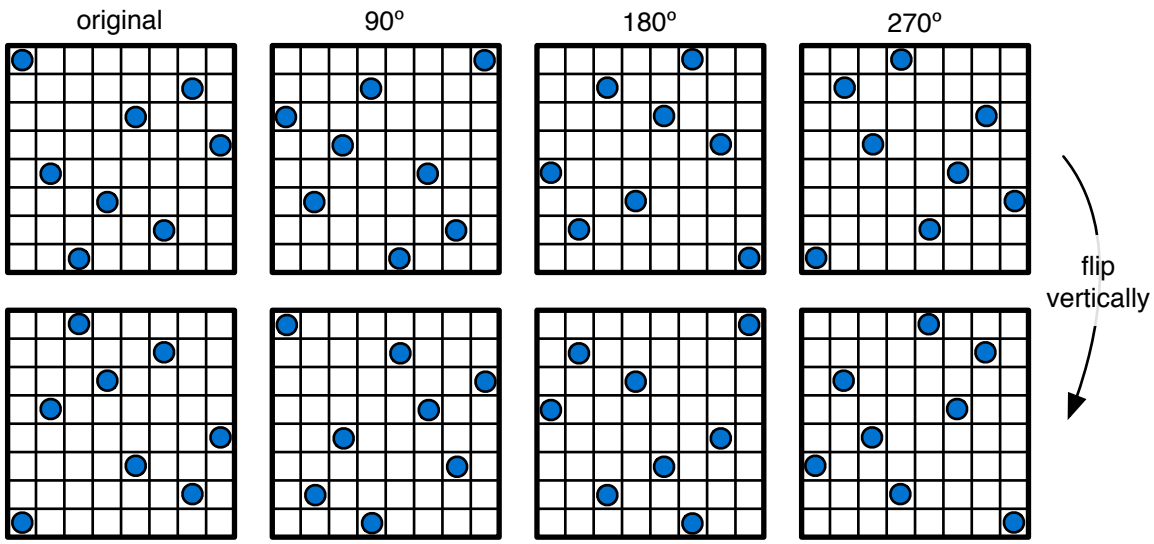


图 7.1: 8 皇后问题解的对称变化

第 1 列的皇后仍然在第 1 行, 而第 3 列的皇后放在第 2 行。于是, 即便是搜索一个解, 求解器可能需要探索很多它已经看到并证明不可满足状态的对称状态!

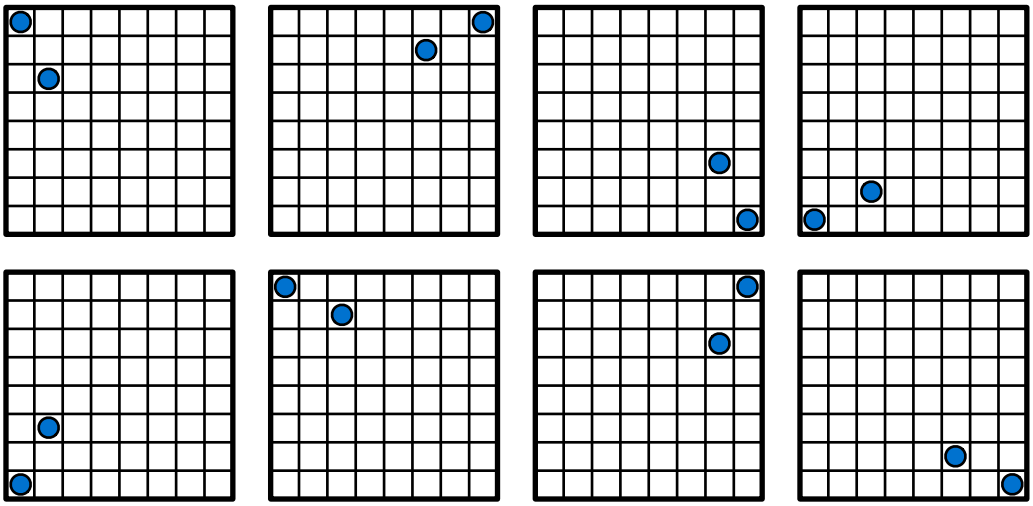


图 7.2: 8 皇后问题不可满足约束的部分解的对称版本

2.7.6.1 静态的对称性破缺

解决对称问题的建模技巧叫做 *对称性破缺*, 在它的最简单形式里, 需要在模型中加入约束使一个 (不完整的) 赋值的所有对称变换去除掉而保留一个。这些约束称作 *静态的对称性破缺约束*。

对称性破缺背后基本的想法是加入 *顺序*。举个例子, 我们可以通过简单地加入约束使第一列的皇后必须在棋盘的上半部分, 从而去除掉所有棋盘垂直翻转的。

```
constraint q[1] <= n div 2;
```

请相信以上约束会去除掉在 图 7.1 中所有对称变换的一半。为了去除 *所有对称*, 我们需要更多工作。

当我们把所有对称都表示成数组变量的排列, 一组 *字典顺序约束*可以用于破坏所有对称。举个

例子, 如果数组变量名为 `x` , 而翻转数组是这个问题的一种对称, 那么以下约束可以破坏那种对称:

```
constraint lex_lesseq(x, reverse(x));
```

那么二维数组又怎么样呢? 字典顺序同样适用, 我们只需要把数组转换成一维的. 举个例子, 下面的约束破坏了沿着其中一个对角线翻转数组的对称性 (注意到第二个生成式里对换的数组下标):

```
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 ]);
```

字典排序约束的好处在于我们可以加入多个 (同时破坏几个对称性), 而不需要它们互相干扰, 只要我们保持第一个参数中的顺序一致即可。

对于 n 皇后问题, 很不幸的是这个技巧不能马上适用, 因为又一些对称不能被描述成数组 `q` 的排列。克服这个问题的技巧是把 n 皇后问题表示成布尔变量。对于每个棋盘的每个格子, 布尔变量表示是否有一个皇后在上面。现在所有的对称性都可以表示成这个数组的排列。因为主要的 n 皇后问题的主要约束在整型数组 `q` 里面更加容易表达, 我们只需要两个模型都用起来, 然后在它们之间加入连通约束。正如多重建模和连通 (page 96) 中解释的一样。

加入布尔变量, 连通约束和对称性破缺约束的完整模型展示在 列表 7.7 里面。我们可以做一些小实验来检查它是否成功的破坏所有对称性。尝试用不断增加的 `n` 运行模型, 比如从 1 到 10, 数一下解的个数 (比如, 使用 Gecode 求解器的 `-s` 标志, 或者选择 IDE 中"Print all solutions" 和"Statistics for solving")。你应该可以获得以下数列的解: 1, 0, 0, 1, 2, 1, 6, 12, 46, 92。你可以搜索 *On-Line Encyclopedia of Integer Sequences* (http://oeis.org) 来校验这个序列。

列表 7.7: n 皇后问题对称性破缺的部分模型 (full model: `nqueens_sym.mzn`).

```
% 映射每一个位置 i,j到一个布尔变量上来表示在i,j上是否有一个皇后
array[1..n,1..n] of var bool: qb;

% 连通约束
constraint forall (i,j in 1..n) ( qb[i,j] <-> (q[i]=j) );

% 字典排序对称性破缺
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) ])
;
```

2.7.6.2 其他对称的例子

许多其他问题都有内在的对称性, 破坏这些对称性常常会令求解表现不一样。以下是一些常见的例子:

- 装箱问题: 当尝试把物品装入箱子时, 任意两个有相同容量的箱子都是对称的
- 涂色问题: 当尝试为一个图涂色使得所有相邻的节点都有不同的颜色时, 我们通常用整型变量对颜色建模。但是, 对颜色的任意排列都是一种合法的涂色方案。

- 车辆路线问题: 如果任务是给顾客分配一些车辆, 任何两辆有相同容量的车可能是对称的 (这跟装箱问题是相似的)
- 排班/时间表问题: 两个有相同能力的职员可能是可以相互交换的, 就像两个有相同容量或者设备的的房间一样

CHAPTER 2.8

在 MiniZinc 中对布尔可满足性问题建模

MiniZinc 可以被用来给布尔可满足性问题建模，这种问题的变量被限制为是布尔型 (`bool`)。MiniZinc 可以使用有效率的布尔可满足性求解器来有效地解决求得的模型。

2.8.1 整型建模

很多时候，尽管我们想要使用一个布尔可满足性求解器，我们可能也需要给问题的整数部分建模。有三种通用的方式使用布尔型变量对定义域为范围 $0 \ldots m$ 内的整型变量 I 建模，其中 $m = 2^k - 1$ 。

- 二元: I 被表示为 k 个二元变量 $i_0, \ldots, i_{k-1}$ ，其中 $I = 2^{k-1} i_{k-1} + 2^{k-2} i_{k-2} + \cdots + 2 i_1 + i_0$ 。在 MiniZinc 中，这可表示为

```
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));
```

- 一元: I 被表示为 m 个二元变量 $i_1, \ldots, i_m$ 且 $i = \sum_{j=1}^{m} \texttt{bool2int}(i_j)$ 。由于在一元表示中有大量的冗余表示，我们通常要求 $i_j \to i_{j-1}, 1 < j \leq m$ 。在 MiniZinc 中，这可表示为

```
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]));
```

- 值: 其中 I 被表示为 $m+1$ 个二元变量 $i_0, \ldots, i_m$ ，其中 $i = k \Leftrightarrow i_k$ 并且 $i_0, \ldots, i_m$ 中最多有一个为真。在 MiniZinc 中，这可表示为

```
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]);
```

每种表示都有其优点和缺点。这取决于模型中需要对整数做什么样的操作，而这些操作在哪一种表示上更为方便。

2.8.2 非等式建模

接下来，让我们考虑如何为一个拉丁方问题建模。一个拉丁方问题是在 $n \times n$ 个网格上放置 $1..n$ 之间的数值使得每个数在每行每列都仅出现一次。图 列表 8.1 中给出了拉丁方问题的的一个整数模型。

列表 8.1: 拉丁方问题的整数模型 (latin.mzn).

```
int: n; % 拉丁方的大小
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 ];
```

整型变量直接的唯一的约束实际上是非等式，而它在约束 alldifferent 中被编码。数值表示是表达非等式的最佳方式。图 列表 8.2 给出了一个关于拉丁方问题的只含有布尔型变量的模型。注意每个整型数组元素 a[i,j] 被替换为一个布尔型数组。我们使用谓词 exactlyone 来约束每个数值在每行每列都仅出现一次，也用来约束有且仅有一个布尔型变量对应于整型数组元素 a[i,j] 为真。

列表 8.2: 拉丁方问题的布尔型模型 (latinbool.mzn).

```
int: n; % 拉丁方的大小
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 ];
```

2.8.3 势约束建模

让我们来看下如何对点灯游戏建模。这个游戏由一个矩形网格组成，每个网格为空白或者被填充。每个被填充的方格可能包含 1 到 4 之间的数字，或者没有数字。我们的目标是放置灯泡在空白网格使得

- 每个空白的网格是“被照亮的”，也就是说它可以透过一行没有被打断的空白网格看到光亮。
- 任何两个灯泡都不能看到彼此。
- 一个有数值的填充的网格相邻的灯泡个数必须等于这个网格中的数值。

图 8.1 给出了点灯游戏的一个例子以及 图 8.2 给出了它的解。

这个问题很自然地可以使用布尔型变量建模。布尔型变量用来决定哪一个网格包含有一个点灯以及哪一个没有。同时我们也有一些作用于填充的网格上的整数算术运算要考虑。

列表 8.3: 点灯游戏的 SAT 模型 (lightup.mzn).

```
int: h; set of int: H = 1..h; % 板高度
int: w; set of int: W = 1..w; % 板宽度
array[H,W] of -1..5: b;       % 板
int: E = -1;                  % 空白格
set of int: N = 0..4;         % 填充并含有数字的网格
int: F =  5;                  % 填充但不含有数字的网格

% 位置 (i1,j1) 对位置 (i2,j2) 可见
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))
 \/ ((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
```

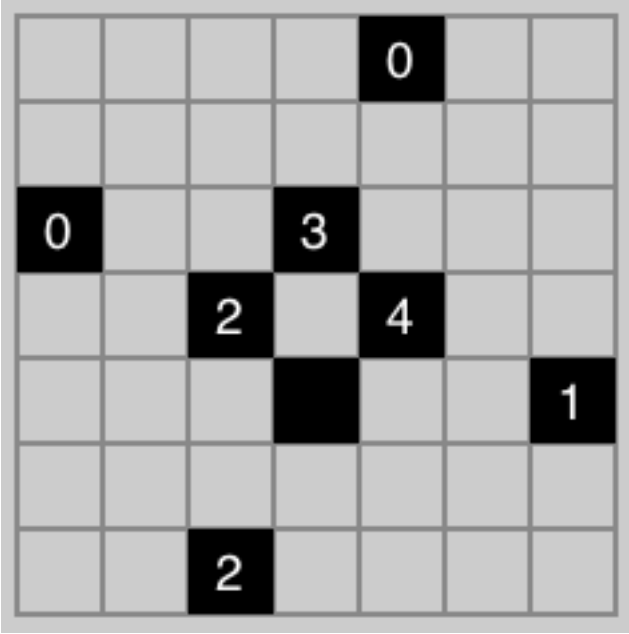


图 8.1: 点灯游戏的一个例子展示

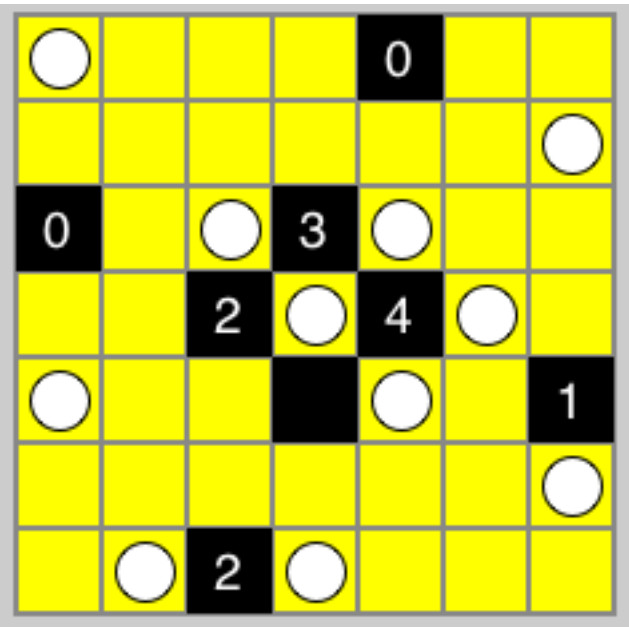


图 8.2: 点灯游戏的完整的解

```
% 填充的网格没有灯泡
constraint forall(i in H, j in W where b[i,j] != E)(l[i,j] == false);
% lights next to filled numbered square agree

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]));
% 每个空白网格是被照亮的
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])
           );
% 任何两个灯泡看不到彼此
constraint forall(i1,i2 in H, j1,j2 in W where
                  (i1 != i2 \/ j1 != j2) /\ b[i1,j1] == E
                  /\ b[i2,j2] == E /\ visible(i1,j1,i2,j2))(
             not l[i1,j1] \/ 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];
```

图 列表 8.3 中给出了这个问题的一个模型。图 图 8.1 中给出的问题的数据文件在图 列表 8.4 中给出。

列表 8.4: 点灯游戏的 图 8.1 中实例的数据文件

```
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 |];
```

模型利用了一个布尔型求和谓词

```
predicate bool_sum_eq(array[int] of var bool:x, int:s);
```

使得一个布尔型数组的和等于一个固定的整数。多种方式都能使用布尔型变量给 *cardinality* 约束建模。

- 加法器网络：我们可以使用包含加法器的一个网络给布尔型总和建立一个二元布尔型表达式
- 排序网络：我们可以通过使用一个排序网络去分类布尔型数组来创建一个布尔型总和的一元表达式
- 二元决策图：我们可以创建一个二维决策图（BDD）来编码势约束。

列表 8.5: 使用二元加法器网络表示势约束 (bboolsum.mzn).

```
% 布尔型变量x的总和 = 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";
```

列表 8.6: 创建二元求和网络的代码 (binarysum.mzn).

```
% 位 x 的的总和 = 二元表示的 s。
%            s[0], s[1], ..., s[k] 其中 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[l] 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;

% 把两个二元数值 x 和 y 加起来，位 ci 用来表示进位，来得到二元 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 == l \/ n == l+1, "length of binary_add output " ++
                        "must be equal or one more than inputs",
    let { array[0..l] 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 > l then s[n] = c[l] else c[l] == 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) \/ (ci /\ xy));
```

我们可以使用图 列表 8.5 给出的二元加法器网络代码实现 bool_sum_eq 。图 列表 8.6 中定义的谓词 binary_sum 创建了一个 x 总和的二维表示法。它把列表分为两部分，把每一部分分别加起来得到它们的一个二元表示，然后用 binary_add 把这两个二元数值加起来。如果 x 列大小是奇数，则最后一位被保存起来作为二元加法时的进位来使用。

列表 8.7: 使用二元加法器网络表示势约束 (uboolsum.mzn).

```
% 布尔型变量 x的总和 = 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";
```

列表 8.8: 奇偶归并排序网络 (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;

% 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 \/ i2)) /\ (o2 = (i1 /\ i2));
```

我们可以使用图 列表 8.7 中给出的一元排序网络代码来实现 `bool_sum_eq` 。势约束通过扩展输入 `x` 长度为 2 的次幂，然后使用奇偶归并排序网络给得到的位排序来实现。奇偶归并排序工作方式在图 列表 8.8 中给出，它递归地把输入列表拆为两部分，给每一部分排序，然后再把有序的两部分归并起来。

列表 8.9: 使用二元加法器网络表示势约束 (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;
```

我们可以使用图 列表 8.9 中给出的二元决策图代码来实现 `bool_sum_eq` 。势约束被分为两种情况：或者第一个元素 `x[1]` 为 `true` 并且剩下位的总和是 `s-1` ，或者 `x[1]` 为 `false` 并且剩下位的总和是 `s` 。它的效率的提高依赖于去除共同子表达式来避免产生太多的相同的约束。

CHAPTER 2.9

FlatZinc 和展平

约束求解器不会直接支持 MiniZinc 模型. 为了运行一个 MiniZinc 模型, 它被翻译成一个 MiniZinc 的简单子集叫 FlatZinc. FlatZinc 反映了大多数约束求解器只会求解具有 $exists c_1 \wedge \cdots \wedge c_m$ 的满足问题或者有 minimize z subject to $c_1 \wedge \cdots \wedge c_m$ 的优化问题, 其中 c_i 是基本的约束而 z 是一个具有某些限定形式的整型或者浮点表达式.

`minizinc` 工具包含了 MiniZinc 编译器, 它可以用一个 MiniZinc 模型和数据文件来创建一个展平后的 FlatZinc 模型, 它等价于给定数据的 MiniZinc 模型表达为之前提到的受限制的形式. 通常来说构建一个给求解器的 FlatZinc 模型是对用户隐藏的, 不过你也可以通过以下命令查看结合数据 `data.dzn` 来展平一个模型 `model.mzn` 的结果:

```
minizinc -c model.mzn data.dzn
```

这会创建一个 FlatZinc 模型叫 `model.fzn` .

在这一章中我们探索把 MiniZinc 翻译成 FlatZinc 的过程.

2.9.1 展平表达式

底层求解器的限制意味着复杂的 MiniZinc 表达式需要被 展平为内部不具有更复杂结构项的基本约束的合取式.

思考以下保证两个在长方形箱子的两个圆不会重叠的模型:

列表 9.1: 两个不会重叠的圆的模型 (`cnonoverlap.mzn`).

```
float: width;          % 包含圆的长方形的宽
float: height;         % 包含圆的长方形的高
float: r1;
var r1..width-r1:  x1; % (x1,y1) 是第一个圆的中心
var r1..height-r1: y1;
float: r2;
var r2..width-r2:  x2; % (x2,y2) 是第二个圆的中心
var r2..height-r2: y2;
```

```
                        % 中心之间至少有r1 + r2的距离
constraint (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) >= (r1+r2)*(r1+r2);
solve satisfy;
```

2.9.1.1 简化和求值

给定数据文件

```
width = 10.0;
height = 8.0;
r1 = 2.0;
r2 = 3.0;
```

转换到 FlatZinc 首先通过替换所有参数为它们的值来简化模型，然后求出所有取值已经固定的表达式的值. 在这步简化后参数的值将不再需要. 一个例外是大型数组的参数值. 如果它们被适用多于一次，那么为了避免重复大的表达式这个参数会被保留.

在简化后，列表 9.1 的模型的变量和参数声明部分变为

```
var 2.0 .. 8.0: x1;
var 2.0 .. 6.0: y1;
var 3.0 .. 7.0: x2;
var 3.0 .. 5.0: y2;
```

2.9.1.2 定义子表达式

现在没有约束求解器可以直接处理像在 列表 9.1 里复杂的约束表达式. 作为替代，我们可以命名表达式中的每一个子表达式，我们创建约束来构建及代表子表达式的数值. 让我们来看看约束表达式的子表达式. `(x1 - x2)` 是一个子表达式，如果我们将它命名为 `FLOAT01` 我们可以定义它为 `constraint FLOAT01 = x1 - x2;` . 注意到这个表达式只在模型中出现两次. 我们只需要构建这个值一次，然后我们可以重复使用它. 这就是 *共同子表达式消除*. 子表达式 `(x1 - x2)*(x1 - x2)` 可以命名为 `FLOAT02` ，而我们可以定义它为 `constraint FLOAT02 = FLOAT01 * FLOAT01;` . 我们可以命名 `constraint FLOAT03 = y1 - y2;` 和 `constraint FLOAT04 = FLOAT03 * FLOAT03;` 最后 `constraint FLOAT05 = FLOAT02 * FLOAT04;` . 不等约束本身则变成 `constraint FLOAT05 >= 25.0;` 因为 `(r1+r2)*(r1 + r2)` 计算出结果为 `25.0` . 于是这个约束被展平为

```
constraint FLOAT01 = x1 - x2;
constraint FLOAT02 = FLOAT01 * FLOAT01;
constraint FLOAT03 = y1 - y2;
constraint FLOAT04 = FLOAT03 * FLOAT03;
constraint FLOAT05 = FLOAT02 * FLOAT04;
constraint FLOAT05 >= 25.0
```

2.9.1.3 FlatZinc 约束形式

展平的最后步骤是把约束的形式转换成标准的 FlatZinc 形式，它总是以 $p(a_1, \ldots, a_n)$ 的形式出现. 其中的 `p` 是某个基础约束，$a_1, \ldots, a_n$ 是参数. FlatZinc 尝试使用最少的不同约束形式. 所以 `FLOAT01 = x1 - x2` 首先尝试重写为 `FLOAT01 + x2 = x1` 然后使用 `float_plus` 输出基本的约束. 得出的约束形式如下:

```
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);
```

2.9.1.4 边界分析

我们仍然缺少一项, 声明引入的变量 `FLOAT01` , ..., `FLOAT05` . 这些可以被声明为 `var float` . 不过为了令求解器的任务更简单, MiniZinc 尝试通过简单的分析确定新引入变量的上界和下界. 比如因为 `FLOAT01 = x1 - x2` 和 $2.0 \leq$ `x1` ≤ 8.0 还有 $3.0 \leq$ `x2` ≤ 7.0, 所以可以得出 $-5.0 \leq$ `FLOAT01` ≤ 5.0 然后我们可以看到 $-25.0 \leq$ `FLOAT02` ≤ 25.0 (尽管注意到如果我们发现相乘实际上是平方, 我们可以给出更精确的边界 $0.0 \leq$ `FLOAT02` ≤ 25.0).

细心的读者会发现在*定义子表达式* (page 110) 和*FlatZinc 约束形式* (page 110) 里约束的展平形式的一点不同. 在后面没有非等约束. 因为一元的不等式可以完全被一个变量的边界表示出来, 不等关系可以令 `FLOAT05` 的下界变为 `25.0` , 然后这会变得冗余. 最后 列表 9.1 的展平后形式是:

```
% 变量
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;
% 约束
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;
```

2.9.1.5 目标函数

MiniZinc 就像展平约束一样, 展平最小化和最大化目标函数. 跟其他表达式一样, 目标表达式被展平时创建一个变量. 在 FlatZinc 输出求解项永远是单一变量. 看一下例子*Let 表达式* (page 119) .

2.9.2 线性表达式

约束的一个最重要的而广泛用于建模的形式是线性约束

$$a_1x_1 + \cdots + a_nx_n \begin{array}{c} = \\ \leq \\ < \end{array} a_0$$

其中 a_i 是整数或者浮点数约束, 而 x_i 是整数或者浮点数变量. 它们非常有表达能力, 同时也是 (整数) 线性规划约束求解器唯一支持的形式. 从 MiniZinc 到 FlatZinc 的翻译器尝试创建线性约束, 而不是把线性约束变成许多子表达式.

列表 9.2: 说明线性约束展平的 MiniZinc 模型 (`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;
```

考虑在 列表 9.2 中的模型. 这里并没有为所有子表达式 $3*x$, $3*x-y$, $x*z$, $3*x-y+x*z$, $x+y+z$, $d*(x+y+z)$, $19+d*(x+y+z)$, 和 $19+d*(x+y+z)-4*d$ 创建一个变量. 这里的翻译在创建一个 FlatZinc 约束时, 会尝试创建一个尽可能记录大部分原约束内容的线性约束.

展平会创建线性表达式并将其视为一个单位, 而不视为每个字表达是构建中间变量. 这也使创建的表达式简单化. 从约束中抽取出线性表达式为

```
var 0..80: INT01;
constraint 4*x + z + INT01 <= 23;
constraint INT01 = x * z;
```

注意到 非线性表达式 $x \times z$ 是如何被抽取出来作为一个新的子表达式并赋予名字的, 与此同时剩下的项会被收集起来从而使每个变量只出现一次 (的确变量 y 的项被移除了)

最后每个约束被写到 FlatZinc 形式, 从而得到:

```
var 0..80: INT01;
constraint int_lin_le([1,4,1],[INT01,x,z],23);
constraint int_times(x,z,INT01);
```

2.9.3 展开表达式

大多数的模型需要创建一些基于输入数据的约束. MiniZinc 通过数组类型, 列表和列生成解析还有聚合函数来支持这些模型.

考虑以下从生产调度例子 列表 2.2 中出现的聚合函数表达式

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

由于这用到生成器语法, 我们可以把它重写成可以被编译器处理的相等的形式:

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

给定数据

```
nproducts = 2;
nresources = 5;
capacity = [4000, 6, 2000, 500, 500];
consumption= [| 250, 2, 75, 100, 0,
              | 200, 0, 150, 150, 75 |];
```

这首先创建 `p = 1` 的数组

```
[ capacity[r] div consumption[p,r]
                          | r in 1..5 where consumption[p,r] > 0]
```

也就是 `[16, 3, 26, 5]` 然后计算最小值为 3. 它之后为 `p = 2` 建立相同的数组 `[20, 13, 3, 6]`，并计算最小的数值为 3. 然后创建数组 `[3, 3]` 并计算最大值为 3. 在 FlatZinc 里面没有 `mproducts` 的表示，这种通过计算数值 3 的方法会被用来代替 `mproducts`.

在约束模型中最常见的聚合表达式是 `forall`. Forall 表达式会被展开成为多个约束.

考虑以下在 SEND-MORE-MONEY 例子 列表 2.4 中使用预设的分解 `alldifferent` 出现的 MiniZinc 片段.

```
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])
```

`forall` 表达式为每一对需要满足 $i < j$ 的 i, j 创建一个约束，所以创建

```
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
```

在 FlatZinc 中形成

```
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);
```

注意到临时的数组变量 `v[i]` 是如何在 FlatZinc 的输出中被原来的变量替换的.

2.9.4 数组

一维变量在 MiniZinc 中可以有任意的下标，只要它们是相邻的整数. 在 FlatZinc 所有数组都被 `1..l` 标注，其中 `l` 是数组的长度. 这意味着数组查询时需要被转换成 FlatZinc 下标的形式.

考虑以下 MiniZinc 模型来使用 `m` 个 1kg 砝码来平衡一个长为 `2 * l2` 的跷跷板, 其中上面有一个重为 `cw` kg 小孩.

```
int: cw;                                % 小孩重量
int: l2;                                % 一半跷跷板长度
int: m;                                 % 1kg砝码的数量
array[-l2..l2] of var 0..max(m,cw): w; % 在每个点的重量
var -l2..l2: p;                         % 孩子的位置
constraint sum(i in -l2..l2)(i * w[i]) = 0; % 平衡
constraint sum(i in -l2..l2)(w[i]) = m + cw; % 所有使用的砝码
constraint w[p] = cw;                   % 孩子在位置p
solve satisfy;
```

给定 `cw = 2`, `l2 = 2`, 和 `m = 3` , 展开可以产生约束

```
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;
```

不过 FlatZinc 坚持 `w` 数组从下标 1 开始. 这意味着我们需要重写所有数组获取来使用新的下标数值. 对于固定值数组查找这很简单, 对于变量值数组查找我们可能需要创建一个新的变量. 以上公式的结果为

```
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;
```

最后我们重写约束成 FlatZinc 的形式. 注意到变量数组下标查找的形式是如何映射到 `array_var_int_element` 上的.

```
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);
```

MiniZinc 支持多维数组, 但是 (目前来说)FlatZinc 只支持单维度数组. 这意味着多维度数组必须映射到单维度数组上, 而且多维度数组访问必须映射到单维度数组访问.

考虑在有限元平面模型 列表 2.1: 的 Laplace 等式约束:

```
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; % 在点 (i,j) 处的温度
var float: left;   % 左侧温度
var float: right;  % 右侧温度
var float: top;    % 顶部温度
var float: bottom; % 底部温度

% 拉普拉斯方程: 每一个内部点温度是它相邻点的平均值
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]);
```

假设 `w = 4` 和 `h = 4`，这会创建约束

```
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];
```

25 个元素的 2 维数组被转换成一维数组，而且下标也会相应改变：所以 `[i,j]` 下标会变成 `[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];
```

2.9.5 具体化

FlatZinc 模型包含了只有变量和参数声明，和一系列原始的约束. 所以当我们在 MiniZinc 用布尔连接符而不是析取式来建模时，需要进行一些处理. 处理使用连接符而不只是析取式来构建的复杂公式，其核心的方法是具体化. 具体化一个约束 c 创建新的约束等价于 $b \leftrightarrow c$，即如果约束满足则布尔变量 b 的值是 `true`，否则为 `false`.

当我们有能力 具体化约束，对待复杂公式的方式跟数学表达式并无不同. 我们为子表达式创建了一个名称和一个展平的约束来约束子表达式的数值.

考虑以下任务调度例子 列表 2.8 中出现在约束表达式：

```
constraint %% 保证任务之间没有重叠
    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]
    ) );
```

对于数据文件

```
jobs = 2;
tasks = 3;
d = [| 5, 3, 4 | 2, 6, 3 |]
```

然后展开过程生成

```
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];
```

具体化在析取式中出现的约束创建新的布尔变量来定义每个表达式的数值.

```
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;
```

每个基础的约束现在会映射到 FlatZinc 形式下. 注意到二维数组 `s` 是如何映射到一维数组里面的.

```
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);
```

`int_lin_le_reif` 是线性约束 `int_lin_le` 的具体化形式.

大多数 FlatZinc 基本约束 $p(\bar{x})$ 有一个具体化形式 $p_reif(\bar{x}, b)$, 它利用最后额外的参数 b 来定义一个约束 $b \leftrightarrow p(\bar{x})$. 定义像 `int_plus` 和 `int_plus` 的函数式关系的 FlatZinc 基本约束不需要支持具体化. 反而, 有结果的函数的等式被具体化了.

具体化的另外一个重要作用出现在当我们使用强制转换函数 `bool2int` (可能是显式地或者隐式地把布尔表达式使用成整数表达式使用). 平整过程将创建一个布尔变量来保存一个布尔表达式参数, 以及一个整型变量 (限制到 `0..1`) 来保存这个数值.

考虑 列表 2.12 中的魔术序列问题.

```
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))));
```

给定 `n = 2`，展开创造了

```
constraint s[0] = bool2int(s[0] = 0) + bool2int(s[1] = 0);
constraint s[1] = bool2int(s[0] = 1) + bool2int(s[1] = 1);
```

和展平创造了

```
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;
```

最后的 FlatZinc 形式是

```
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;
```

2.9.6 谓词

MiniZinc 支持许多不同求解器的的一个重要的因素是全局约束 (还有真正的 FlatZinc 约束) 可以根据不同的求解器专业化.

每一个求解器生命一个谓词有时会, 但有时并不会提供具体的定义. 举个例子一个求解器有一个内建的全局 `alldifferent` 谓词, 会包含定义

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

在全局约束库中, 同时一个求解器预设的分解会有定义

```
predicate alldifferent(array[int] of var int:x) =
    forall(i,j in index_set(x) where i < j)(x[i] != x[j]);
```

谓词调用 $p(\bar{t})$ 在平整时, 首先为每个参数项 t_i 创建对应变量 v_i . 如果谓词没有定义我们只需要使用创建的参数 $p(\bar{v})$ 来调用谓词. 如果一个谓词有一个定义 $p(\bar{x}) = \phi(\bar{x})$ 然后我们将用谓词的定义来替换这个谓词调用 $p(\bar{t})$ 当中形式参数被替换为对应的参数变量, 即 $\phi(\bar{v})$. 注意到如果一个谓词调用 $p(\bar{t})$ 出现在具体化位置而且它没有定义, 我们则检查我们适用这个谓词的具体化版本 $p_reif(\bar{x}, b)$.

考虑在 SEND-MORE-MONEY 例子 列表 2.4 中 `alldifferent` 约束的:

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

如果这个求解器有一个内建的 `alldifferent` 我们只需要为这个参数创建一个新的变量, 然后在调用时替换它.

```
array[1..8] of var 0..9: v = [S,E,N,D,M,O,R,Y];
constraint alldifferent(v);
```

注意到边界分析尝试在新的数组变量上找到一个紧的边界. 建造这个数组参数的理由就是, 如果我们中使用相同的数组两次,FlatZinc 求解器不会创建它两次. 在这种情况下因为它不是使用两次, 后面的转换会把 `v` 替换成它的定义.

如果求解器使用预设的定义 `alldifferent` 呢? 然后变量 `v` 会正常地定义, 谓词调用会被替换为一个当中变量被重命名的版本, 其中 `v` 替换了形式参数 `x`. 结果的程序是

```
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])
```

我们可以在展开表达式 (page 112) 中看到.

考虑到以下约束, 其中 `alldifferent` 在一个具体化位置出现.

```
constraint alldifferent([A,B,C]) \/ alldifferent([B,C,D]);
```

如果求解器有 `alldifferent` 的具体化形式, 这将会被展平为

```
constraint alldifferent_reif([A,B,C],BOOL01);
constraint alldifferent_reif([B,C,D],BOOL02);
constraint array_bool_or([BOOL01,BOOL02],true);
```

适用这个预设的分解, 谓词替换会首先创建

```
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]) \/
           forall(i,j in 1..3 where i<j)(v2[i] != v2[j]);
```

它最终会展平成 FlatZinc 形式

```
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);
```

注意到共同子表达式消除是如何利用具体化不等式 `B != C` 的. (虽然有一个更好的转换把共同约束提升到最顶层的合取式中)

2.9.7 Let 表达式

Let 表达式是 MiniZinc 中可用于引入新的变量的非常强大的工具. 在展平时,let 表达式被转换成变量和约束声明. 这个 MiniZinc 的关系语义意味着这些约束必须像在第一个包含的布尔表达式中出现.

let 表达式的一个重要特征是每一次它们被使用时它们都创建新的变量.

考虑一下展平的代码

```
constraint even(u) \/ even(v);
predicate even(var int: x) =
          let { var int: y } in x = 2 * y;
```

首先谓词调用被他们的定义取代.

```
constraint (let { var int: y} in u = 2 * y) \/
           (let { var int: y} in v = 2 * y);
```

然后 let 变量会另外被重命名

```
constraint (let { var int: y1} in u = 2 * y1) \/
           (let { var int: y2} in v = 2 * y2);
```

最后变量声明会被抽取到第一层

```
var int: y1;
var int: y2;
constraint u = 2 * y1 \/ v = 2 * y2;
```

一旦 let 表达式被清除我们可以像之前那样展平.

记住 let 表达式可以定义新引入的变量 (对某些参数的确需要这样做). 这些隐式地定义了必须满足的约束.

考虑婚礼座位问题 列表 3.10 的复杂的目标函数.

```
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));
```

为了简介我们假设只使用前两个相互敌视的人, 所以

```
set of int: Hatreds = 1..2;
array[Hatreds] of Guests: h1 = [groom, carol];
array[Hatreds] of Guests: h2 = [clara, bestman];
```

展平的第一步是展开 `sum` 表达式, 给定 (为了简洁我们保留客人名字和参数 `Seats` , 在实际中他们会被他们的定义取代):

```
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));
```

然后每一个在 let 表达式的新变量会被分别命名为

```
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));
```

在 let 表达式的变量会被抽取到第一层, 并且定义约束会被抽取到正确的层 (在这里是最顶层).

```
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));
```

现在我们已经构成不需要使用 let 表达式的等价的 MiniZinc 代码, 和展平可以正常进行.

为了说明没有出现在最顶层的 let 表达式的情况, 看看以下模型

```
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;
```

我们抽取变量定义到最顶层, 约束到第一个围住的布尔语境, 这里是蕴含的右手边.

```
var 0..9: x;
var 2..9: y;
var int: z;
constraint x >= 1 -> (y = x - 1 /\ z = x * y /\ y + z * z < 14);
```

注意到如果我们知道定义一个变量的等式的真值不会为假, 我们可以抽取它到最顶层. 这通常可以是求解大幅加快.

对于上面的例子, 因为 `y` 的值域对于 `x - 1` 并不够大, 所以约束 `y = x - 1` 可能失败. 不过约束 `z = x * y` 不可以 (实际上边界分析会给予 `z` 足够大的边界来包含 `x * y` 所有的可能值). 一个更好的展平可以给出

```
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);
```

现在 MiniZinc 编译器通过总是使引入变量声明的边界足够大, 它应该可以包含所有它定义的表达式的值. 然后在正确的语境中为 let 表达式加入边界约束. 在上面的例子中这个结果是

```
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);
```

这个转换可以使求解更加高效, 因为 let 变量的所有可能的复杂计算并没有被具体化.

这种方法的另外一个原因是在引入变量出现在取反语境的时候它也可以被使用 (只要它们有一个定义). 考虑一下与之前相似的这个例子:

```
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;
```

这个 let 表达式出现在否定语境中, 不过每个引入变量都被定义了. 展平后的代码是

```
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;
```

注意到作为对比的 let 消除方法不能给出一个正确的转换:

```
var 0..9: x;
var 2..9: y;
var int: z;
constraint (y = x - 1 /\ z = x * y /\ y + z * z > 14) -> x >= 5;
```

以上转换对于所有 `x` 的可能值给出结果, 而原来的约束除掉了 `x = 4` 的可能性.

对于在 let 表达式中的处理跟对定义的变量的处理是相似的. 你可以认为一个约束等价于定义一个新的布尔变量. 新的布尔变量定义可以从最顶层中抽取出来, 而布尔保存在正确的语境下.

```
constraint z > 1 -> let { var int: y,
                          constraint (x >= 0) -> y = x,
                          constraint (x < 0)  -> y = -x
                      } in y * (y - 2) >= z;
```

可以处理成

```
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;
```

然后展平成

```
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. 节 4.3.6 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 节 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 节 4.3.6 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 节 3.1.4)
- Alternatively, you can use the MiniZinc IDE to create solver configuration files, see 节 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 节 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 Gecode (in case you have two versions installed), use `--solver Gecode@6.3.0`. 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 节 4.3.6).

`--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.

`--param-file <file>`
: Read command line options from the given JSON file. See 节 3.1.5.

`--json-stream`
: Generate output as line-break delimited JSON objects.

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).

`-i, --intermediate`
: Turns on printing of intermediate solutions for optimisation problems. Has no effect for satisfaction problems.

`-n-i, --no-intermediate`
: Turns off printing of intermediate solutions for optimisation problems. Has no effect for satisfaction problems.

`--all-satisfaction`
: Turns on printing of all solutions for satisfaction problems. Has no effect for optimisation problems.

`--disable-all-satisfaction`
: Turns off printing of all solutions for satisfaction problems. Has no effect for optimisation 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

`-m <file>, --model <file>`
: File named <file> contains the model.

`-d <file>, --data <file>`
: File named <file> contains data used by the model.

`--checker <file>, --solution-checker <file>`
: File named <file> contains the solution checker model.

`-D <data>, --cmdline-data <data>`
: Include the given data assignment in the model.

`--cmdline-json-data <data>`
: Use the given JSON data 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. Combine with `--input-is-flatzinc` when passing FlatZinc code.

`-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)

`-O<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, -O-`
: 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

`-O, --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

`--only-sections <section1[,section2,...]>`
: Include only the given sections in output

`--not-sections <section1[,section2,...]>`
: Exclude the given sections from 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.mip.cplex"],[
↪"","org.gecode.gecode"]],
  "solverDefaults": [["org.minizinc.mip.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).

3.1.5 Command-line Parameter Files

The `minizinc` tool is able to read command line options from one or more JSON files through use of the `--param-file` option. This option may be specified multiple times, and substitutes the options given in the JSON file in its place, meaning it can be combined with normal command-line options, as well as other `--param-file` configurations. MiniZinc also recognises the `.mpc` file extension for command line parameter files without requiring the `--param-file` option.

As most `minizinc` options which take a single value are given the final one they are assigned, later specifications of command-line parameter files will generally override op-

tions contained in previous ones, and later command-line options will override previous options.

The given file must be in JSON format and contain a root object whose keys and values correspond to command line options and their values respectively. For long-form options which start with two dashes, the leading dashes may be omitted from the JSON object key name. For short-form options which start with a single dash, the leading dash is required. Command line options which take multiple values through being specified multiple times can be represented using a JSON array. Options which take the JSON boolean value `true` are treated as flags without associated parameters (`false` omits the flag).

Consider the configuration file `config.mpc`:

```
{
  "solver": "gecode",
  "cmdline-data": ["x = 1", "y = 2", "z = 3"],
  "-p": 2,
  "output-paths": true
}
```

Running `minizinc --verbose config.mpc -p 4 model.mzn` (or `minizinc --verbose --param-file config.mpc -p 4 model.mzn`) is equivalent to running

```
$ minizinc --verbose \
    --solver gecode \
    --cmdline-data "x = 1" \
    --cmdline-data "y = 2" \
    --cmdline-data "z = 3" \
    -p 2 \
    --output-paths \
    -p 4 \
    model.mzn
```

Note that since `-p 4` comes after `-p 2`, its value overrides the previous one and so `-p 4` is used. However, the multiple `--cmdline-data` options do not override each other as `--cmdline-data` allows for specification of multiple strings of data.

Configuration files may also be used as a convenient way to specify instances.

For a configuration file `instance.mpc`

```
{
  "model": "model.mzn",
  "data": "data.dzn"
}
```

And another configuration file `gecode.mpc`

```
{
  "solver": "gecode",
  "intermediate-solutions": true
}
```

We can run the instance with `minizinc gecode.mpc instance.mpc`. Note that the files `model.dzn` and `data.dzn` will be resolved relative to the location of the configuration file which specified them.

In this way, multiple configuration files can be set up for different instances and solver configurations, then combined as required.

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 节 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 节 3.2.5.

节 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).

Depending on your platform, the *View* menu also lets you activate ”dark mode”, which switches the colour scheme to a dark background. On macOS 10.14 and later, the operating systems supports a global dark mode, and the MiniZinc IDE will follow that setting.

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:

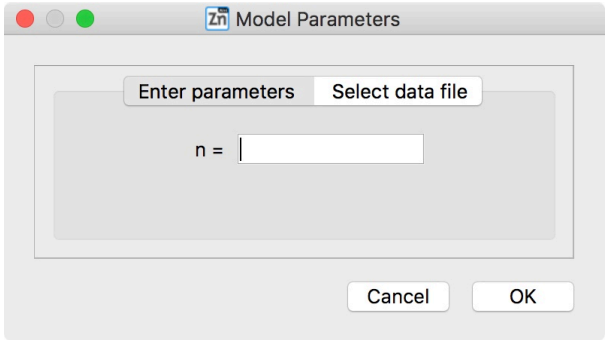
3.2.2.1 Running a model

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.

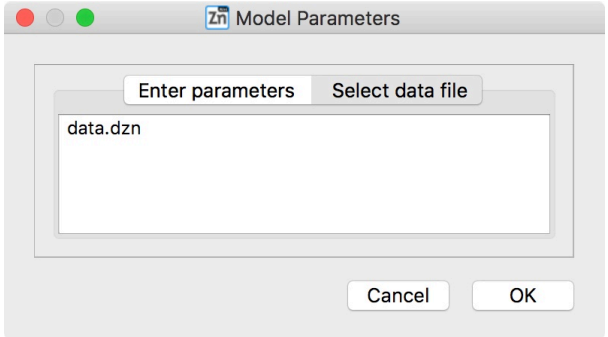
Running a model will open the *Output* window, usually located at the bottom of the IDE's main window. MiniZinc displays progress messages as well as any output received from the solver there. If compilation resulted in an error message, clicking on the error will jump to the corresponding location in 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).

图 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 Gecode 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. Solver configurations can be saved once cloned/modified as `.mpc` files to be reused later. This is done by selecting *Save solver configuration from the *File* menu. The concrete solver used is also shown in this section.

The *Options* section allows you configure common settings which can remain persistent as you switch between configurations. This behaviour can be switched off by unticking the *Maintain these options across solver configurations checkbox*. This area contains two subsections described below.

The *Solving* subsection contains a number of general options. You can set a time limit, after which the execution will be stopped. The behaviour selector controls which kinds of solutions a solver should produce. The "default behaviour" (marked with a `2`) 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 *Output* subsection (marked with a `4`) allows you to control what extra information,

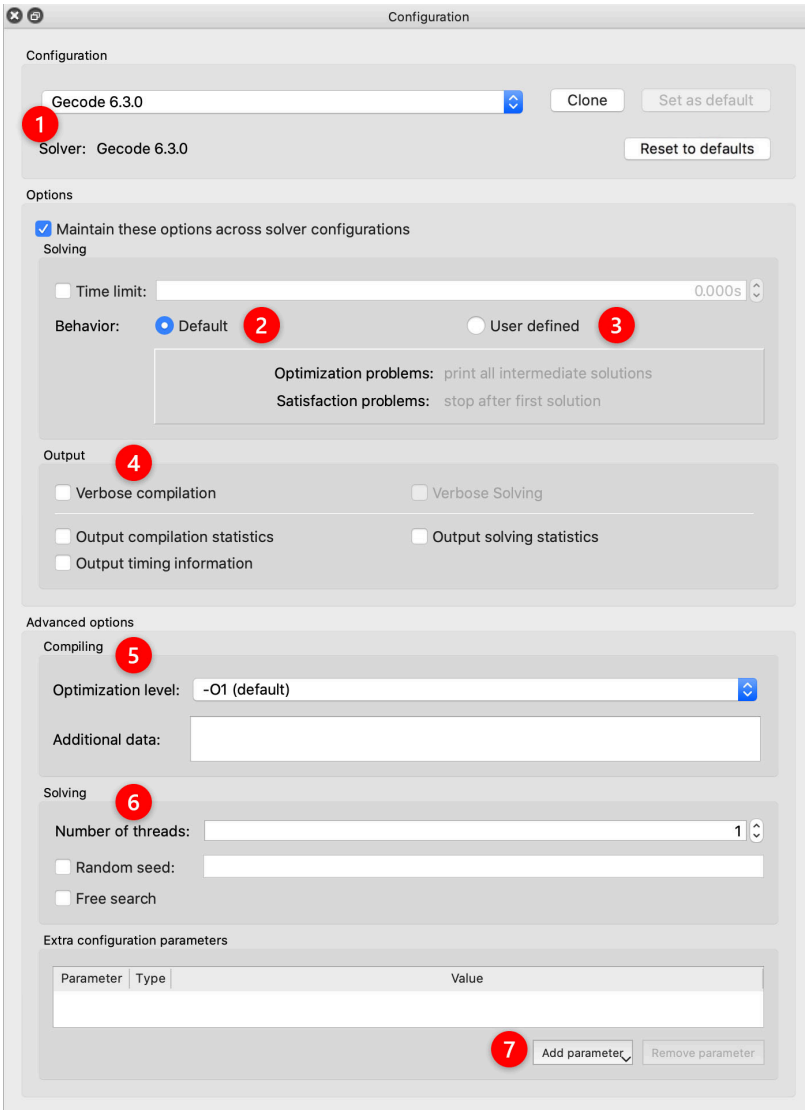


图 2.1: The solver configuration window

if any, the compiler/solver should produce. This includes verbosity options, statistics options and extra timing information where supported.

The *Advanced Options* section contains options for more fine-grained control of the compilation and solving of your instance. These options are less commonly used, and as such the settings are unique to each solver configuration. It contains three subsections described below.

The *Compiling* subsection (marked with a `5`) controls different aspects of the compilation from MiniZinc to FlatZinc for the selected solver. The drop-down 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 input field below allows you to specify additional data (passed into the compilation as if it was part of a `.dzn` file).

The *Solving* subsection (marked with a `6`) contains configuration options for the selected solver, such as the number of threads to use for solvers which support parallelism, the seed to use for random number generation, and whether or not to enable free search (i.e. permitting the solver to ignore search annotations). Only options that are supported by the solver will be available, others will be greyed out.

Finally, the *Extra configuration parameters* subsection allows you to enter any extra options supported by a solver, as well as custom ones where necessary. These options are passed to the `minizinc` command in the same manner as described in 节 3.1.5. The *Add parameter* button (marked with a `7`) opens a drop-down presenting any known options supported by a solver (with a convenience option to add all of them), as well as an option to add a custom parameter. The values associated with each added parameter can be edited by double clicking on the corresponding *Value* cell in the table. Adding a custom parameter requires you to specify the corresponding command-line flag in the first column, the data-type in the second, and the value in the third. The *Remove parameter* button can be used to remove the currently highlighted parameters.

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 output tab of the preferences window.

You can find a detailed description of MiniZinc solution checkers in *Automatic Solution Checking, Model Validation, and Benchmarking* (page 185).

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).

3.2.2.5 Profiling a model

MiniZinc can produce statistics about the translation of a model to FlatZinc, which can be helpful in order to better understand the behaviour of the model. In the IDE, this feature can be accessed from the *MiniZinc* menu: Select *Profile compilation*, which will run the compiler with the current settings (such as selected solver etc.) and output compilation statistics next to each line of code.

For the Magic Series puzzle from 列表 7.2, profiling using the Gurobi solver backend could produce the statistics shown in 图 2.2. The first column (”Cons”) represents the number of FlatZinc constraints generated from each line of code, the second column (”Vars”) the number of variables, and the third column (”Time”) the time it took to compile the line. In this example, for `n=10`, the constraint in lines 4 and 5 creates 100 variables (one 0/1 variable for each of the `bool2int` calls), and the constraints in lines 7 and 8 each result in a single FlatZinc constraint and no additional variables.

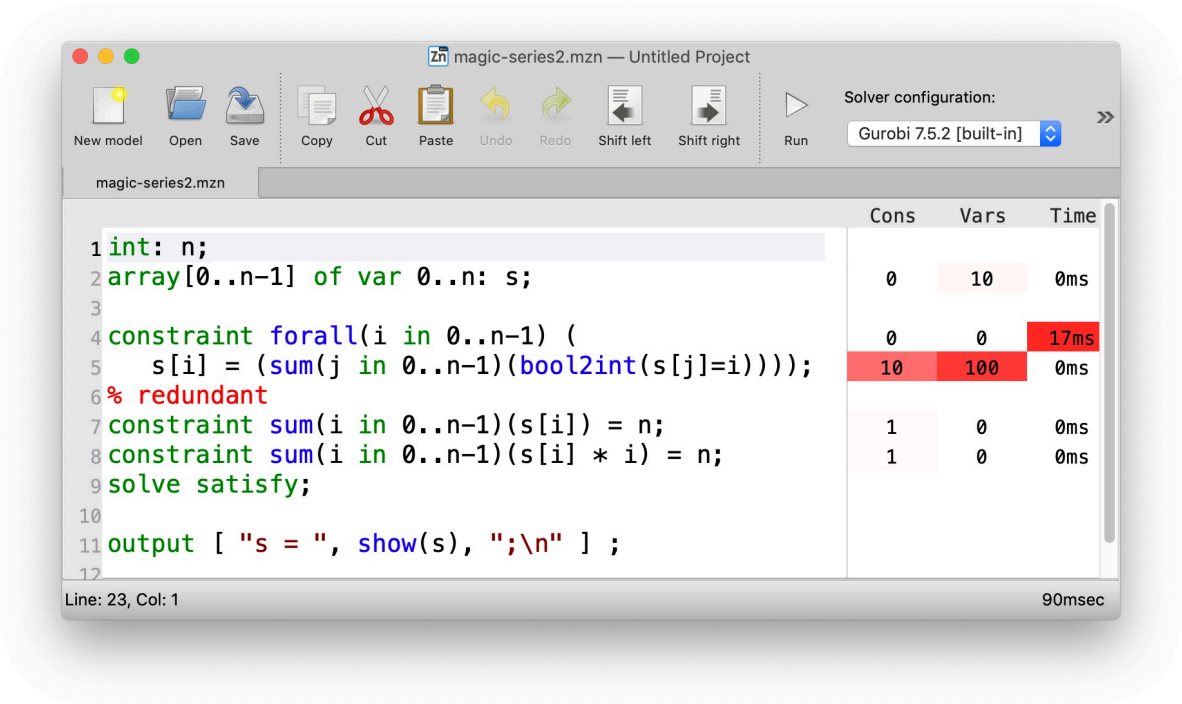


图 2.2: Profiling results for the Magic Series problem (`n=10`)

In addition to the simple statistics output, the IDE also ships an experimental integration of the CP-Profiler tool. This tool provides profiling information of the actual solving process. Since this requires integration with the backend solver, this is only available for a small number of solvers at the moment. See 节 3.4 for more details.

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).

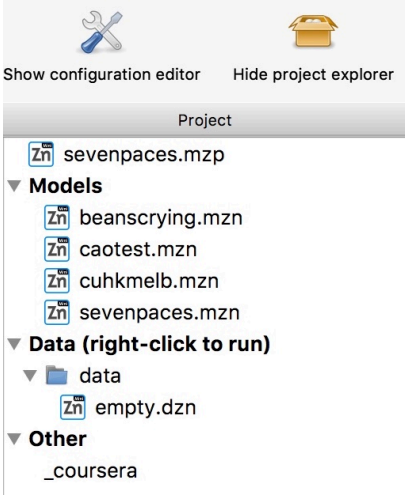


图 2.3: Project explorer

All the files that belong to the current project are shown in the *Project explorer* (see 图 2.3), 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 Optimization[6]
- Advanced Modeling for Discrete Optimization[7]

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 图 2.4). 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.

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 图 2.5.

There are four tabs which allow configuration of different aspects of the IDE:

- **MiniZinc**. Options for configuring compiler search paths. Shown in 图 2.6.
- **Solvers**. Options for built-in and user solvers. Shown in 图 2.7.
- **Editing**. Options for code editing and appearance. Shown in 图 2.9.
- **Output**. Options for solution output. Shown in 图 2.10.

3.2.5.1 Locating the MiniZinc installation

The most important configuration option is the path to the `minizinc` executable. In the example in 图 2.5, 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. 图 2.6 below shows an example where MiniZinc is located at `/home/me/minizinc-2.6.0/bin`.

You can have the MiniZinc IDE check once a day whether a new version of MiniZinc is available.

[6] https://www.coursera.org/learn/basic-modeling
[7] https://www.coursera.org/learn/advanced-modeling

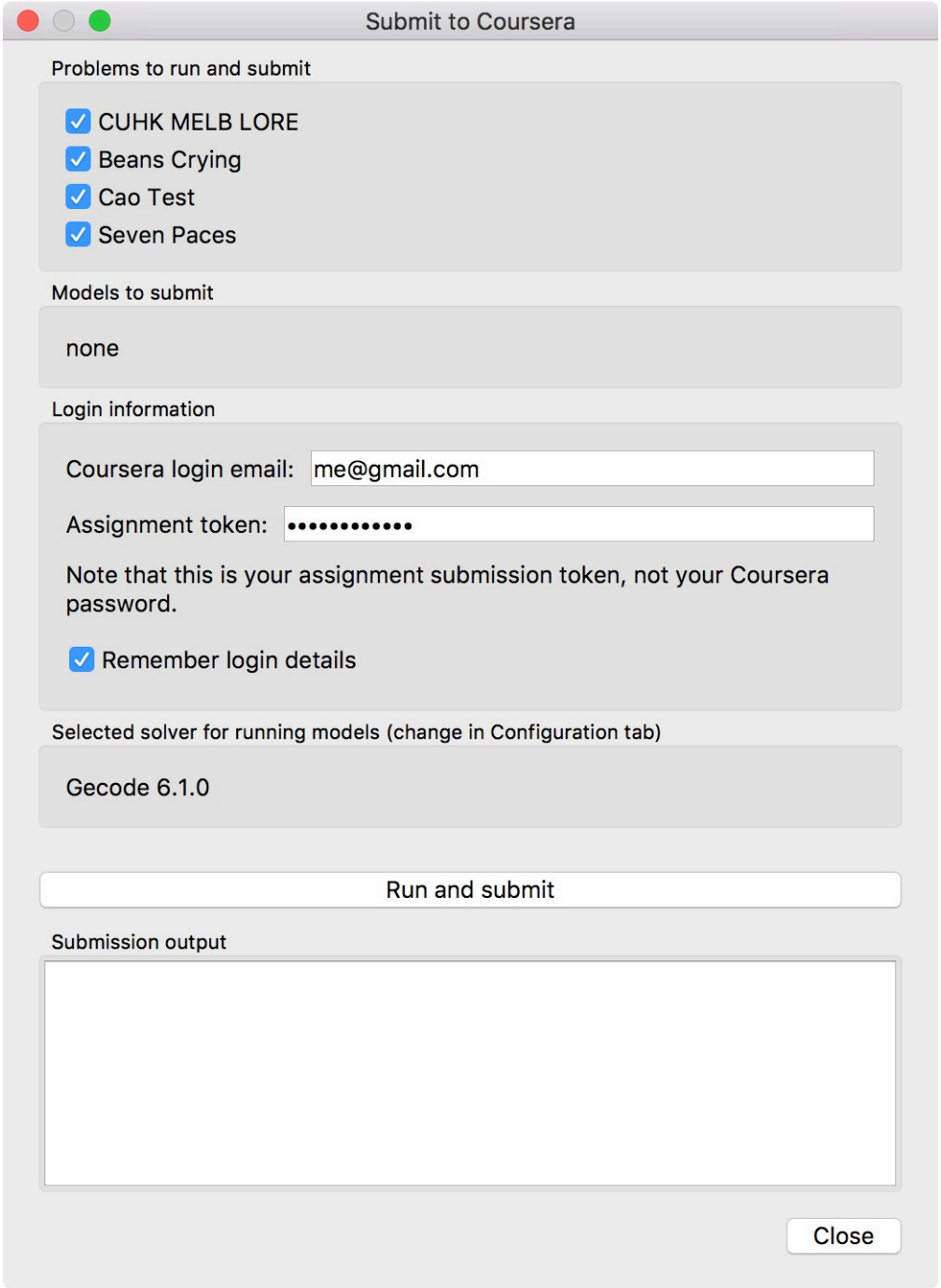


图 2.4: Submitting to Coursera

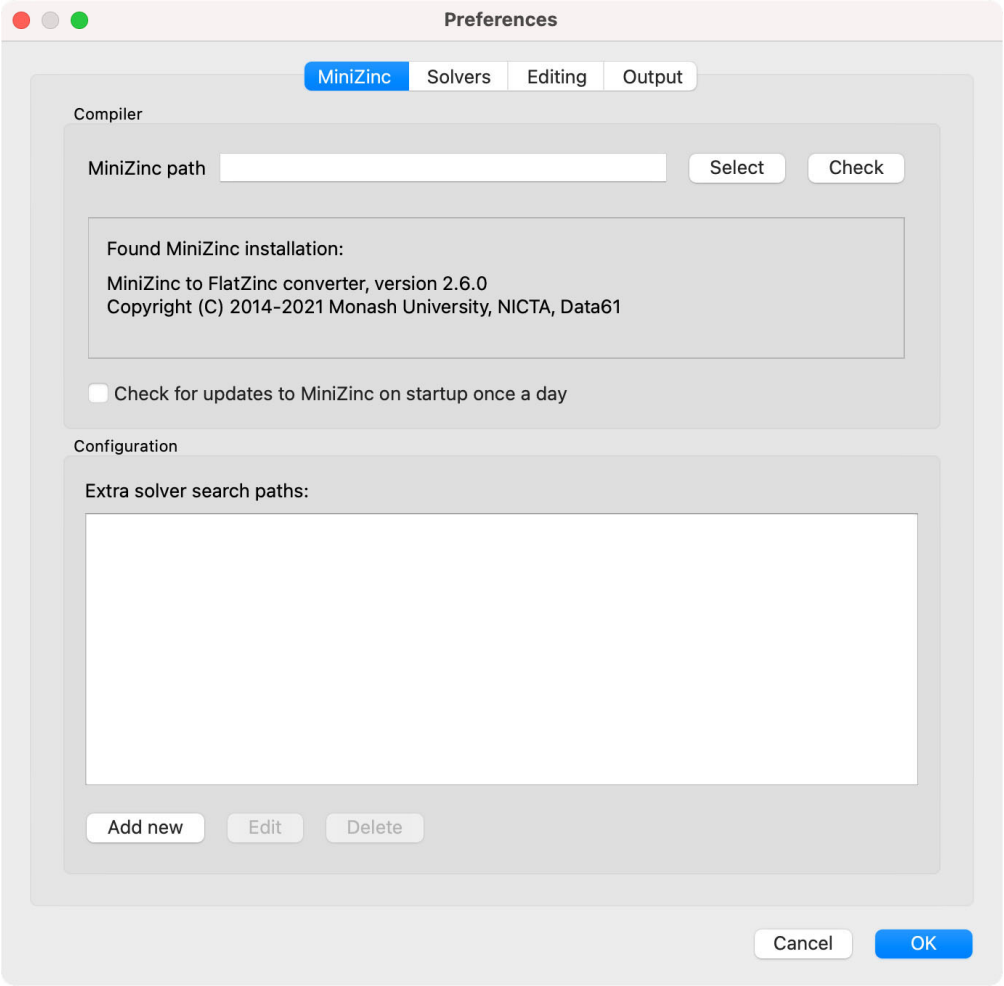


图 2.5: The MiniZincIDE preferences window

3.2.5.2 Adding Third-Party Solvers

Solvers which distribute a **.msc** solver configuration file (see 节 4.3.6) can be added to the MiniZinc IDE by adding the directory containing the solver configuration file to the **Extra solver search paths** as shown in 图 2.6.

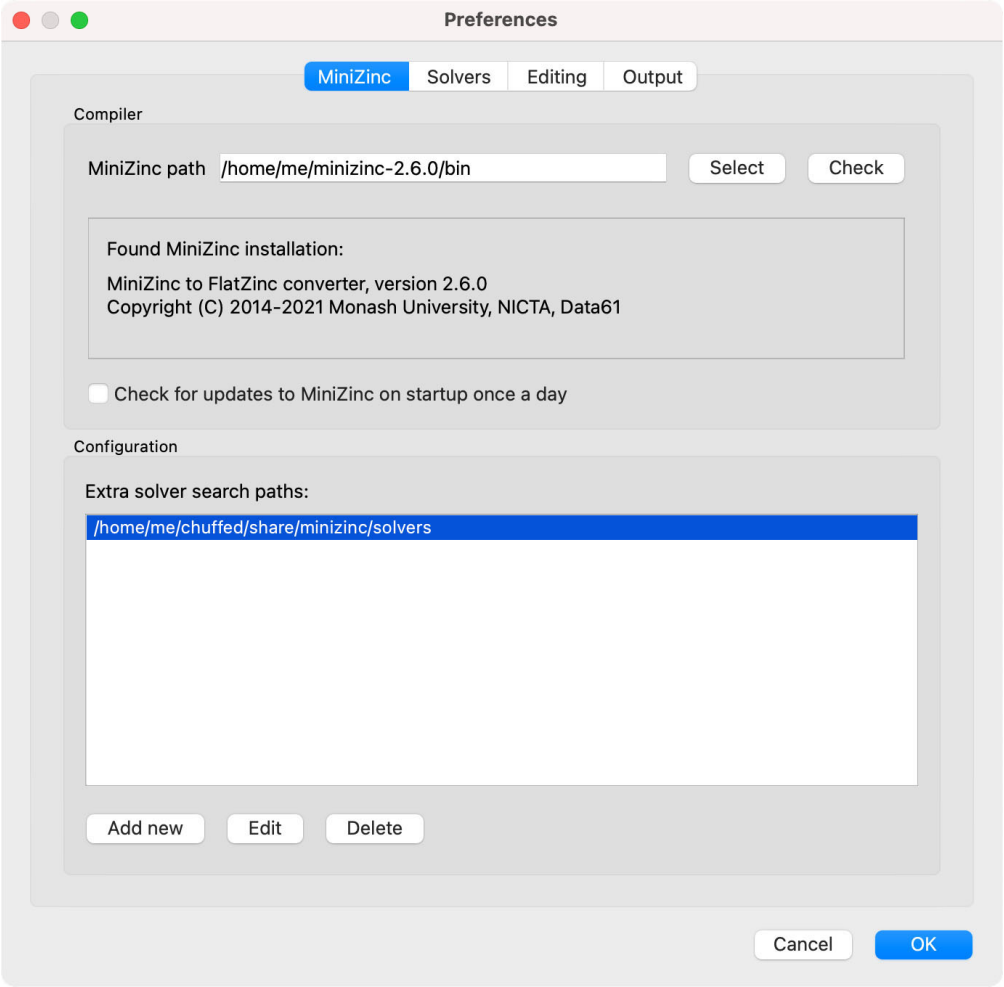


图 2.6: Configuring MiniZinc search paths

The *Solvers* tab of the configuration dialog can be used to inspect the solvers that are currently available to MiniZinc, and to add new solvers which don't have an existing solver configuration file 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, 图 2.7 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.

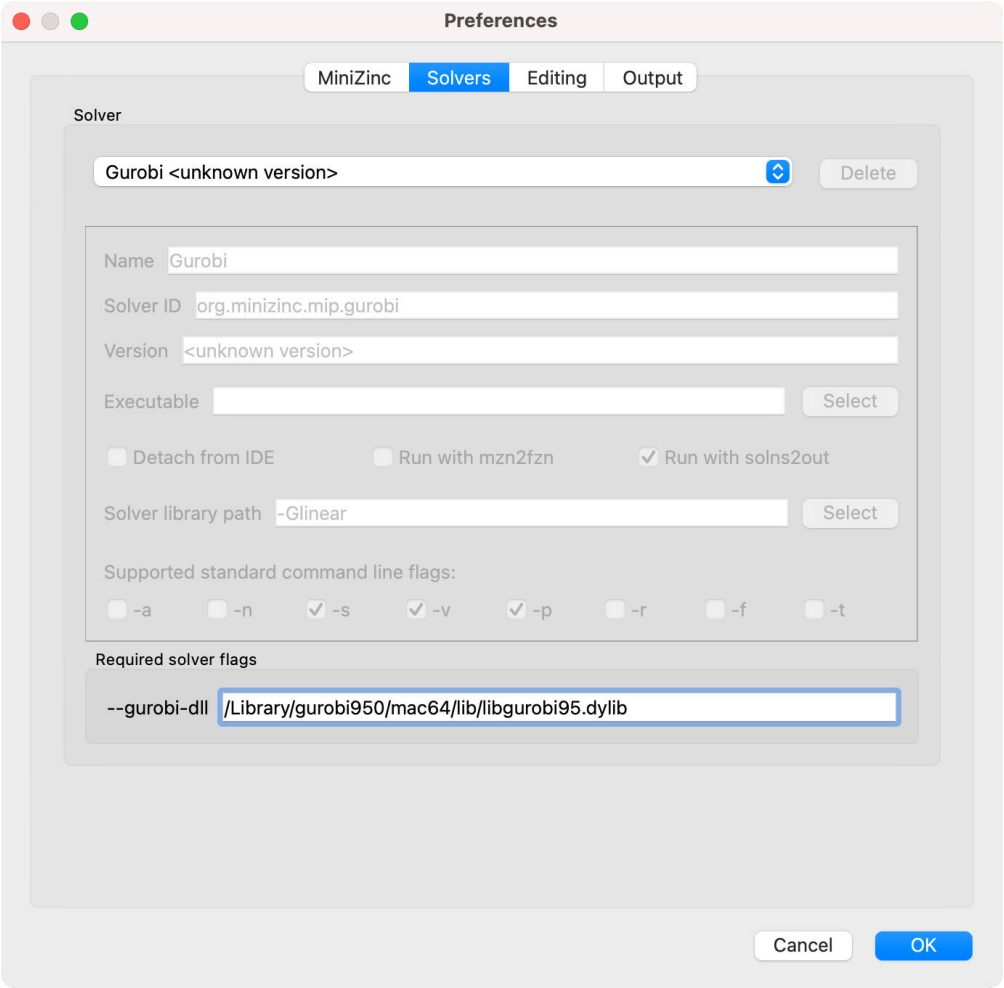


图 2.7: Configuring the Gurobi DLL path

Adding new solvers

The example in 图 2.8 shows a potential configuration for the or-tools solver, which was installed in `/Users/me/or-tools`.

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 节 4.3.4 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`).

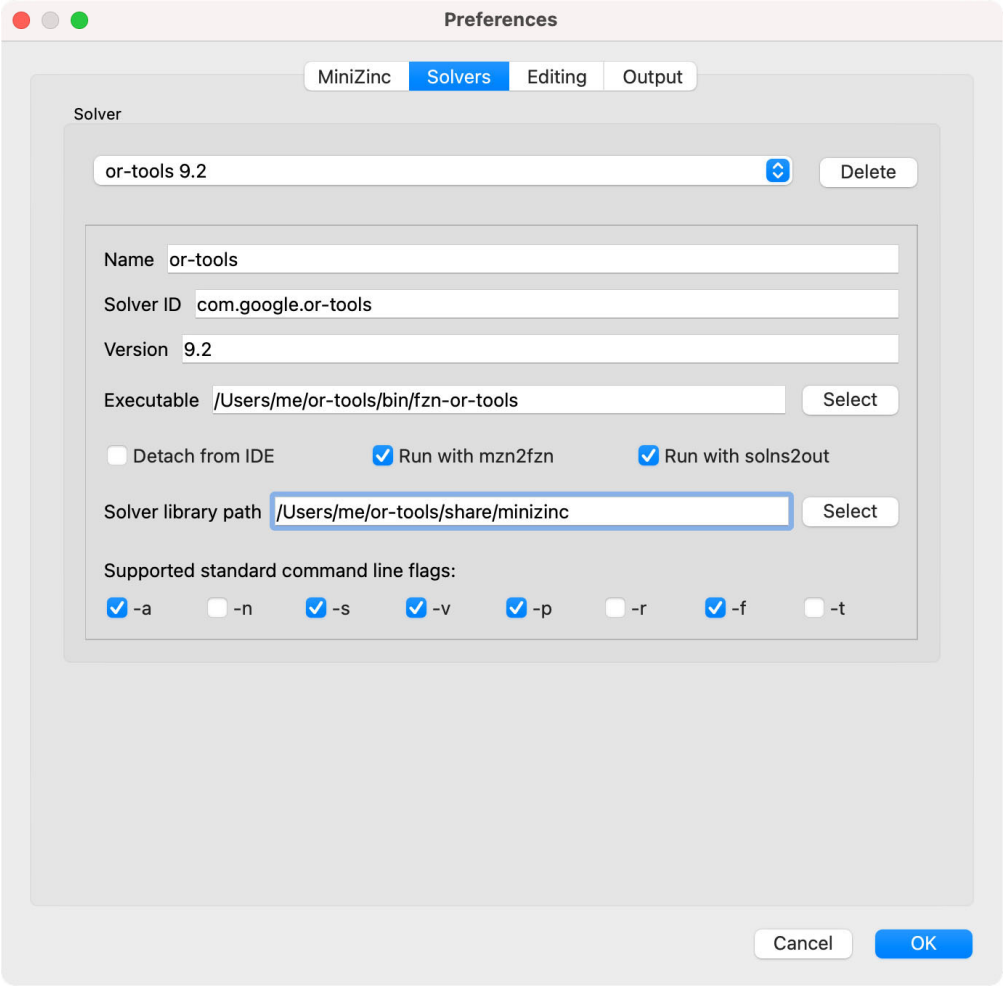


图 2.8: Adding a new solver

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.

3.2.5.3 Editing options

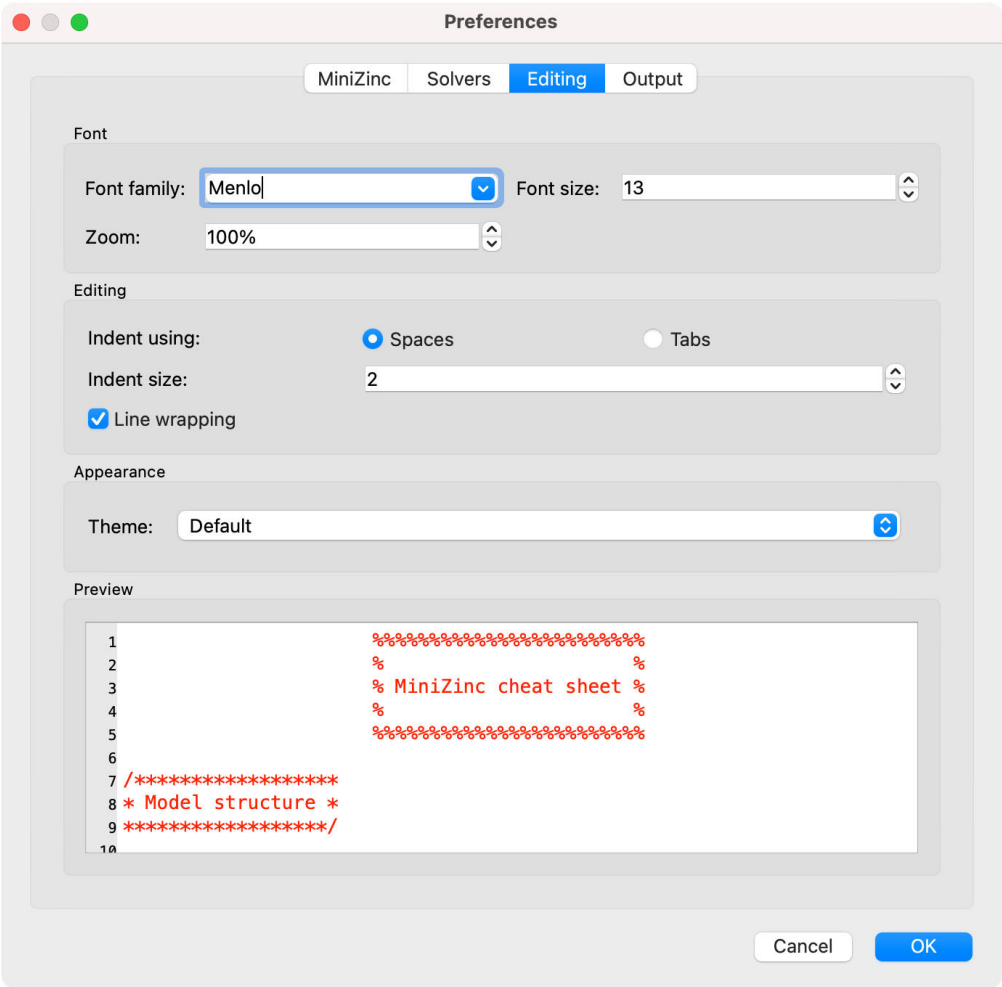


图 2.9: Configuring code editor preferences

The **Editing** tab contains options for the code editor, such as fonts, indentation, and theming.

3.2.5.4 Output options

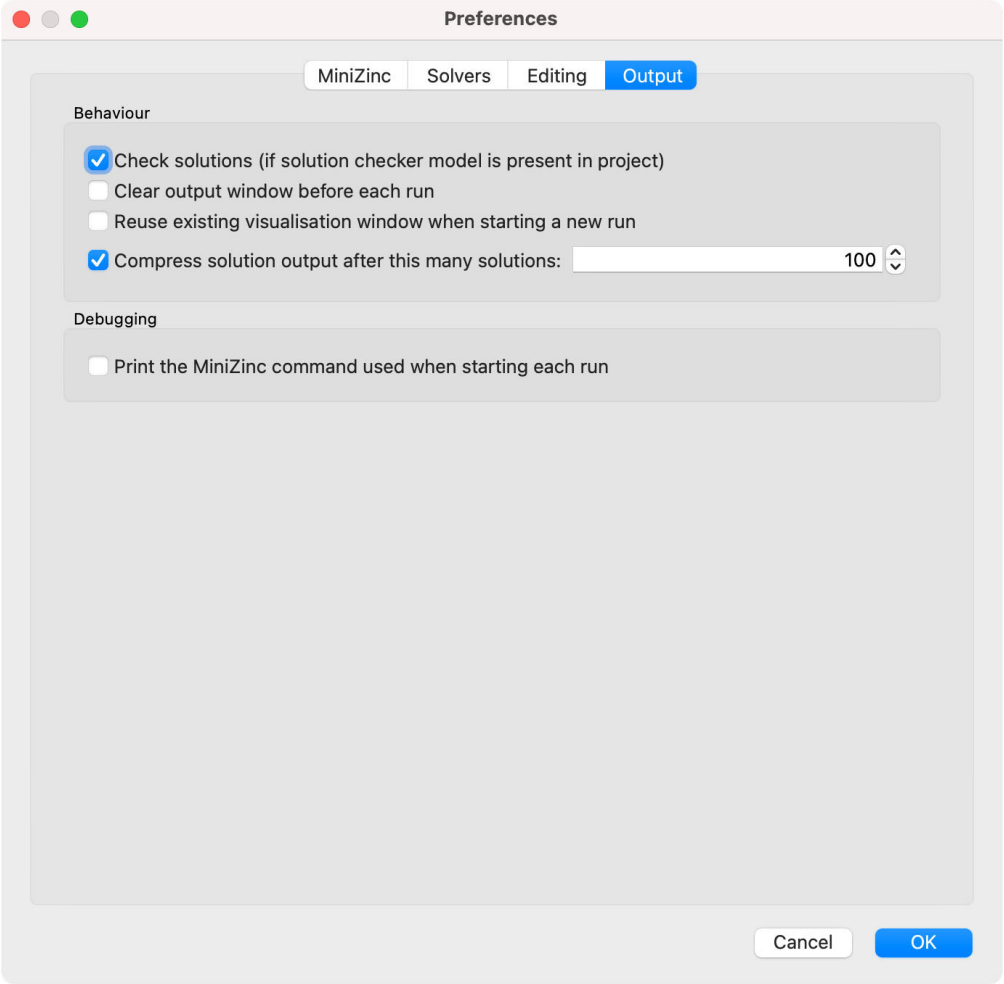


图 2.10: Configuring output settings

There are also a number of output-related options. The first tick box (*Check solutions*) enables checking as described in 节 3.2.2.3. The second tick box enables you to clear the *Output* window automatically every time you run a model. The third *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 rather than the solutions themselves. For example, the following model would produce 1000 solutions when run with *User-defined behavior* and solution limit disabled:

```
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:

```
  [ 99 more solutions ]
x = 200;
----------
  [ 199 more solutions ]
```

```
x = 400;
----------
  [ 399 more solutions ]
x = 800;
----------
  [ 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). The hidden solutions can be shown by clicking the symbol, but this may take some time where a large number of solutions have been truncated.

CHAPTER 3.3

Visualising solutions in the MiniZinc IDE

The MiniZinc IDE provides a web-browser based system for visualising solutions generated by MiniZinc.

3.3.1 Visualisation library

In order to use these features, the library file `ide/vis.mzn` must be included:

```
include "ide/vis.mzn";
```

Running a model with a visualisation will automatically launch a web browser page containing the visualisation. If this page is closed, it can be reopened by clicking the hyperlink on the right side of the run's output.

3.3.2 Pre-defined visualisations

There are a number of pre-defined visualisations to suit common use cases, such as line graphs, scatter plots, bar charts, graph visualisation, and 2D packing visualisation. Multiple visualisations can be shown concurrently.

These come in the form of functions which used as the value of an `output` item.

Documentation for these is available in *MiniZincIDE solution visualisation tools* (page 443).

An example model with a visualisation for a graph along with a time series line graph of the objective value is shown in 列表 3.1.

列表 3.1: Minimum spanning tree data visualisation (vis_mst.mzn).

```
include "globals.mzn";
include "ide/vis.mzn";

% Graph definition
enum NODE = {A, B, C, D, E, F, G, H, I, J};
array [int] of NODE: from = [A, A, A, B, B, B, B, C, C, D, D, E,  E, F, F, F, G,  G,
↪ H, H, I];
array [int] of NODE:   to = [B, C, E, C, D, E, G, D, F, F, G, G,  J, G, H, I, I,  J,
↪ I, J, J];
array [int] of int:     w = [3, 6, 9, 4, 2, 9, 9, 2, 9, 9, 8, 8, 18, 7, 4, 5, 9, 10,
↪ 1, 4, 3];

% Find the minimum spanning tree of the graph
var 0..sum(w): total_weight;
array [index_set(from)] of var bool: es;
constraint weighted_spanning_tree(
  card(NODE),   % Number of nodes
  length(from), % Number of edges
  from,         % Edge from node
  to,           % Edge to node
  w,            % Weight of edge
  es,           % Whether edge is in spanningtree
  total_weight  % Total weight of spanning tree
);
solve minimize total_weight;

% Graph visualisation
array [int] of string: edge_labels = [i: show(w[i]) | i in index_set(w)];
output vis_graph_highlight(
  from,                  % Edge from node
  to,                    % Edge to node
  edge_labels,           % Edges are labelled with their weights
  [i: true | i in NODE], % All nodes are present in a spanning tree
  es                     % Whether edge is in spanningtree
);

% Objective visualisation
output vis_line(total_weight, "Spanning tree weight");
```

3.3.3 Custom visualisations

A user-defined visualisation can be created by making use of the visualisation JavaScript API from an HTML page.

The visualisation is setup using an `output` item annotated with `vis_server`:

```
any: init_data = (
  size: 10,
  label: "My custom visualisation"
);
var 1..3: x;
var 1..3: y;
```

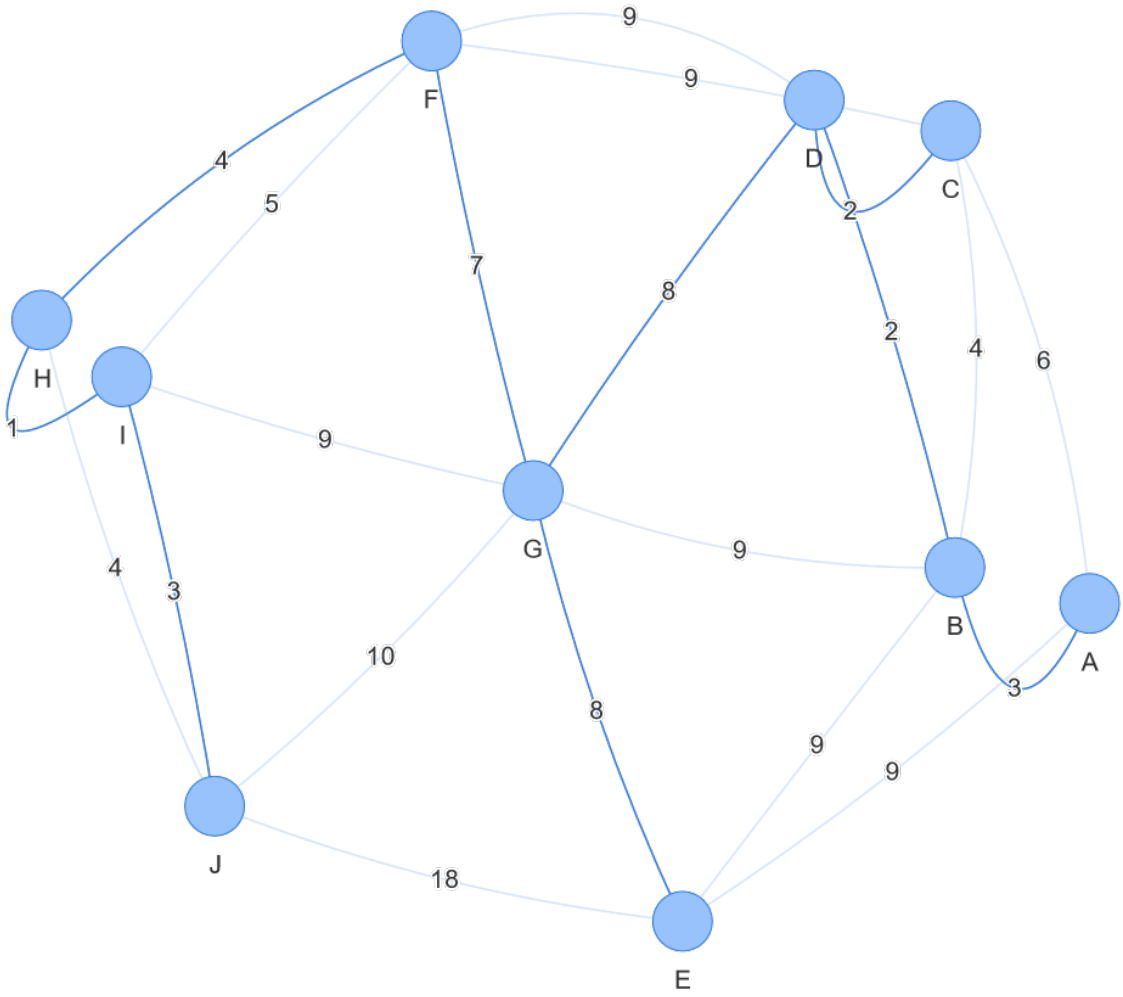


图 3.1: Minimum spanning tree visualisation result

```
any: solution_data = (
  x: x,
  y: y
);
output :: vis_server("my-vis.html", init_data) solution_data;
```

The call to `vis_server` takes a path to an HTML file to load (relative to the model), and can optionally be given some initialisation user data (`init_data`) that is sent upon loading the page. Then, on each solution, the output expression (`solution_data`) is sent to the page.

Note that the initial user data and the solution data are sent as JSON objects, so it is generally convenient to use records (see *Tuple and record types* (page 77)), and you may wish to do some processing either in

Multiple such output items can be used to start concurrent visualisations.

The HTML page is loaded in an embedded web server and has access to the MiniZincIDE JavaScript API, served at `/minizinc-ide.js`. The HTML page can use this API to subscribe to events from the IDE and update its visualisation accordingly.

3.3.3.1 Example visualisation

In this example, we will create a visualisation for a version of the colouring problem introduced in *MiniZinc 基本模型* (page 21).

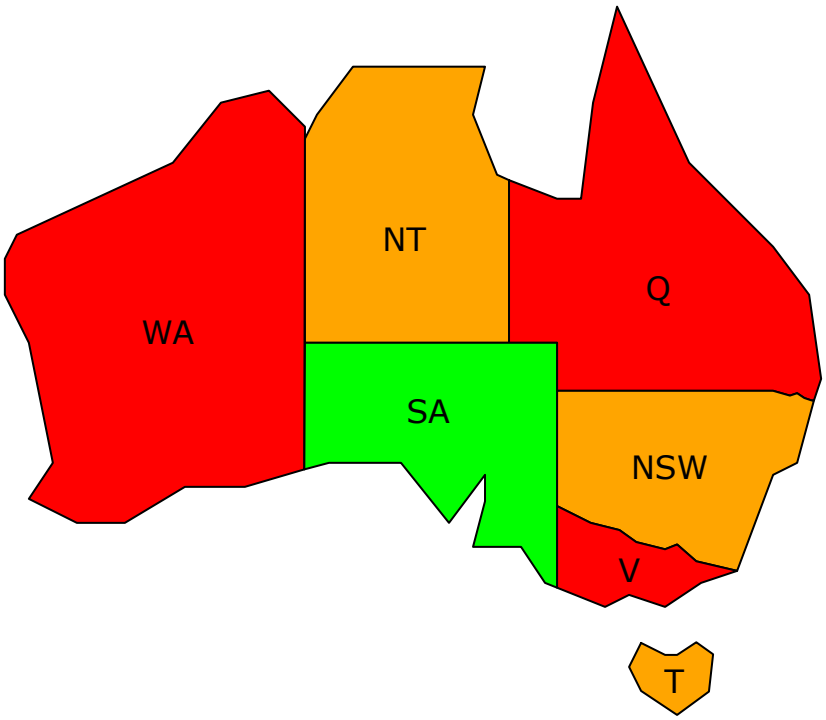


图 3.2: Australia colouring visualisation result

All files should be placed in the same directory so that they are properly served by the IDE's embedded web server.

列表 3.2: Australia colouring data visualisation model (vis_aust.mzn).

```
include "ide/vis.mzn";

int: n_colors;

enum PALETTE = { red, lime, blue, cyan, magenta, yellow, orange };
set of PALETTE: COLOR = to_enum(PALETTE, 1..n_colors);

% Color for each region
var COLOR: wa;
var COLOR: nt;
var COLOR: sa;
var COLOR: q;
var COLOR: nsw;
var COLOR: v;
var COLOR: t;

% Neighboring regions have different colours
constraint wa != nt;
constraint wa != sa;
constraint nt != sa;
constraint nt != q;
constraint sa != q;
constraint sa != nsw;
constraint sa != v;
constraint q != nsw;
constraint nsw != v;

% Data to send during initialisation of the visualisation
any: initial_data = (n: n_colors);

% Data to send on each solution
% Note the use of :: output_only
any: solution_data :: output_only = (
  wa: show(wa), % Get colors as strings
  nt: show(nt),
  sa: show(sa),
  q: show(q),
  nsw: show(nsw),
  v: show(v),
  t: show(t)
);
% Launch the visualisation
output :: vis_server("vis_aust.html", initial_data) solution_data;
```

The MiniZinc model contains an output item annotated with *vis_server* (page 453). This initialises the visualisation, telling the IDE what page to load, and the user data to pass to it. In this case, we are passing the input number of colours so it can be shown on the visualisation.

The solution data is given as a record mapping regions to their string color names. If we didn't convert these to strings, we would have to extract their names from the JSON enum output format in the visualisation script.

Passing JSON data to the visualisation

It should be noted that `showJSON()` is used on the user/solution data to convert them into JSON for the visualisation script. While this supports all MiniZinc values including sets and enums, the format used may need special handling in the visualisation script.
In many cases, it may be easier to convert sets into arrays and enums into strings to make them easier to deal with. For more information about the JSON format generated, see *JSON support* (page 287).
Another important thing to note is that when you place the solution data in its own declaration, you will likely need annotate it with `:: output_only` to ensure that does not get evaluated during compilation.

列表 3.3: Australia colouring data visualisation HTML page (`vis_aust.html`).

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Australia Visualisation</title>
    <script src="/minizinc-ide.js"></script>
    <style>
      html, body {
        margin: 0;
        padding: 0;
        overflow: hidden;
        font-family: Helvetica, sans-serif;
      }
      * {
        box-sizing: border-box;
      }
      main {
        width: 100vw;
        height: 100vh;
        padding: 50px;
      }
      svg {
        width: 100%;
        height: 100%;
      }
      .shapes {
        fill: none;
        stroke: #000;
        stroke-linejoin: round;
        stroke-linecap: round;
        stroke-width: 1;
      }
      .labels {
        font-family: sans-serif;
        font-size: 16px;
        text-anchor: middle;
        fill: #000;
      }
    </style>
  </head>
```

```
  <body>
    <main>
      <h3>Map coloring with <span id="count">n</span> colors</h3>
      <svg viewBox="0 0 412 357.33">
        <g class="shapes">
          <path id="wa" d="m152 67.333v-6l-18-18-24 6-24 30-78 36-6 12v18l12 24 12␣
↪60-12 18 24 12h24l30-18h30l29.68-8.72z" />
          <path id="nt" d="m152 67.333 6-12 18-24h66l-6 24 12 30 6 2.72v81.28h-102v-
↪102" />
          <path id="q" d="m254 88.053 24 9.28h12l6-48 12-48 36 78 42 42 18 24 6 42-
↪3.6 10.8h-1.2l-3.6-1.2-3.6-2.4-3.6 1.2-8.4-2.4h-108v-24h-24v-81.28" />
          <path id="nsw" d="m406.4 198.13-8.4 31.2-12 6-18 48-20.4-4.8-9.6-8.4-6 2.
↪4-14.4-3.6-8.4-6-14.4-3.6-16.8-8.4v-57.6h108l8.4 2.4 3.6-1.2 3.6 2.4 3.6 1.2z" />
          <path id="v" d="m368 283.33-20.4-4.8-9.6-8.4-6 2.4-14.4-3.6-8.4-6-14.4-3.
↪6-16.8-8.4v40.8l24 9.6 12-6 18 6 18-12z" />
          <path id="sa" d="m278 169.33h-126l-0.32 63.28 12.32-3.28h36l24 30 18-
↪24v13.2l-6 22.8h24l12 18 6 2.4v-122.4" />
          <path id="t" d="m320 319.33 12 6h6l9.6-6.32 8.4 6-2.08 18.64-15.92 11.68-
↪18-12-6-12 6-12" />
        </g>
        <g class="labels">
          <text x="83.533119" y="170.0192">WA</text>
          <text x="201.66409" y="123.36276">NT</text>
          <text x="328.74969" y="147.87715">Q</text>
          <text x="334.34839" y="237.23608">NSW</text>
          <text x="314.19815" y="289.03265">V</text>
          <text x="336.71103" y="344.3877">T</text>
          <text x="213.65285" y="209.55853">SA</text>
        </g>
      </svg>
    </main>
    <script src="vis_aust.js"></script>
  </body>
</html>
```

The visualisation page includes the script `/minizinc-ide.js`, which is provided by the embedded web server. This allows access to the MiniZinc IDE JavaScript API using the global `MiniZincIDE` object. The page contains an SVG element containing the basic parts of the visualisation, ready to be coloured when a solution is produced. Another script, `vis_aust.js` is loaded with the visualisation logic.

列表 3.4: Australia colouring data visualisation script (`vis_aust.js`).

```
const HTML_COLORS = ['red', 'yellow', 'blue', 'lime', 'magenta', 'cyan', 'orange'];

(async function() {
  // getUserData can be used to retrieve the JSON data passed to vis_server()
  const userData = await MiniZincIDE.getUserData();
  document.getElementById('count').textContent = userData.n;

  // Handler to set the colors for the solution
  function setSolution(data) {
    for (const r in data) {
      document.getElementById(r).setAttribute('fill', data[r]);
    }
  }
```

```
  // Visualise last solution on startup
  const numSols = await MiniZincIDE.getNumSolutions();
  if (numSols > 0) {
    const solution = await MiniZincIDE.getSolution(numSols - 1);
    setSolution(solution.data);
  }

  // Show new solutions if we're following the latest solution
  let followLatest = true;
  MiniZincIDE.on('solution', (solution) => {
    if (followLatest) {
      setSolution(solution.data);
    }
  });

  MiniZincIDE.on('goToSolution', async (index) => {
    // Requesting index -1 turns on following latest solution
    // Otherwise, we stop showing the latest solution and show the requested one
    followLatest = index === -1;
    const solution = await MiniZincIDE.getSolution(index);
    setSolution(solution.data);
  })
})();
```

The script uses an asynchronous immediately invoked function expression, allowing us to easily use the asynchronous functions in the MiniZincIDE JavaScript API. `MiniZincIDE.getUserData()` allows us to retrieve the data we passed to `vis_server`. The function `setSolution(data)` is defined to handle the solution data and update the visualisation. We load the latest solution using `MiniZincIDE.getSolution(idx)` if one exists on startup, in case the visualisation page is refreshed or open after solving has begun. The `solution` and `goToSolution` events are subscribed to, so we know when to update the visualisation.

3.3.3.2 MiniZincIDE JavaScript API

The MiniZincIDE JavaScript API provides an asynchronous interface between the IDE and visualisations.

```
MiniZincIDE.on(event, callback)
```

Subscribes to the event `event` and call `callback` whenever it occurs. The valid events are:

- `solution`: triggered when a new solution is produced. The callback function is passed an object containing the `time` of the solution in milliseconds and the `data` from the associated `vis_json` output statement.

  ```
  MiniZincIDE.on('solution', function(solution) {
    console.log(solution.time);
    console.log(solution.data);
  });
  ```

- `goToSolution`: triggered when the user has selected a particular solution to visu-

alise. The callback function is passed the solution index to display, or `-1` if the user has enabled following the latest solution. Visualisations which support showing a particular solution should subscribe to this event.

```
MiniZincIDE.on('goToSolution', async function(index) {
  const solution = await MiniZincIDE.getSolution(index);
  visualiseSolution(solution);
});
```

- `status`: triggered when a final status line is received (such as `==========`). The callback function is passed an object containing the `status` and `time` in milliseconds. The `status` value can be one of
  - `ALL_SOLUTIONS`
  - `OPTIMAL_SOLUTION`
  - `UNSATISFIABLE`
  - `UNSAT_OR_UNBOUNDED`
  - `UNBOUNDED`
  - `UNKNOWN`
  - `ERROR`
- `finish`: triggered when solving finishes (or is stopped). The callback function is passed the time when solving finished in milliseconds.

```
MiniZincIDE.off(event, callback)
```

Unsubscribes from an event given the `event` name and `callback` function.

```
MiniZincIDE.getUserData()
```

Returns a `Promise` which resolves to the user data passed to `ide_launch_server()`.

```
MiniZincIDE.goToSolution(idx)
```

Requests that the solution with the given index `idx` is shown. This causes the `goToSolution` event to be triggered for all visualisations being shown. Visualisations which allow multiple solutions to displayed simultaneously can call this function when the user interacts on a particular solution. This allows other visualisations to then also show the solution of interest.

The index `-1` is used to request that the latest solution be tracked by the visualisations.

```
MiniZincIDE.solve(modelFile, dataFiles, options)
```

Requests that the IDE stop solving the current instance and start solving with the given `modelFile`, `dataFiles`, and parameter configuration `options` `modelFile` and `dataFiles` may be set to `null` to re-solve using the current model and data files. Otherwise, `modelFile` is the path to the `.mzn` file to solve and `dataFiles` is an array of paths to the `.dzn` or `.json`

files. These must be in the same directory or a child directory of the current model. `options` is an optional object in the parameter configuration file format (see *Command-line Parameter Files* (page 134)) to be passed to MiniZinc (in addition to the current solver configuration).

```
MiniZincIDE.getNumSolutions()
```

Returns a `Promise` which resolves to the number of solutions which have been found.

```
MiniZincIDE.getSolution(index)
```

Returns a `Promise` which resolves to the solution object (containing `time` and `data`) for the given index. Negative indexes refer to indexes from the end (e.g. `-1` refers to the latest solution).

```
MiniZincIDE.getAllSolutions()
```

Returns a `Promise` which resolves to an array of all solution objects (each containing `time` and `data`).

```
MiniZincIDE.getStatus()
```

Returns a `Promise` which resolves to an object containing the `time` and final `status`, or `null` if there is no final status.

```
MiniZincIDE.getFinishTime()
```

Returns a `Promise` which resolves to the finishing time, or `null` if it has not finished.

CHAPTER 3.4

CP-Profiler

CP-Profiler provides search tree visualisations for executions of constraint programming solvers from within the MiniZinc IDE.

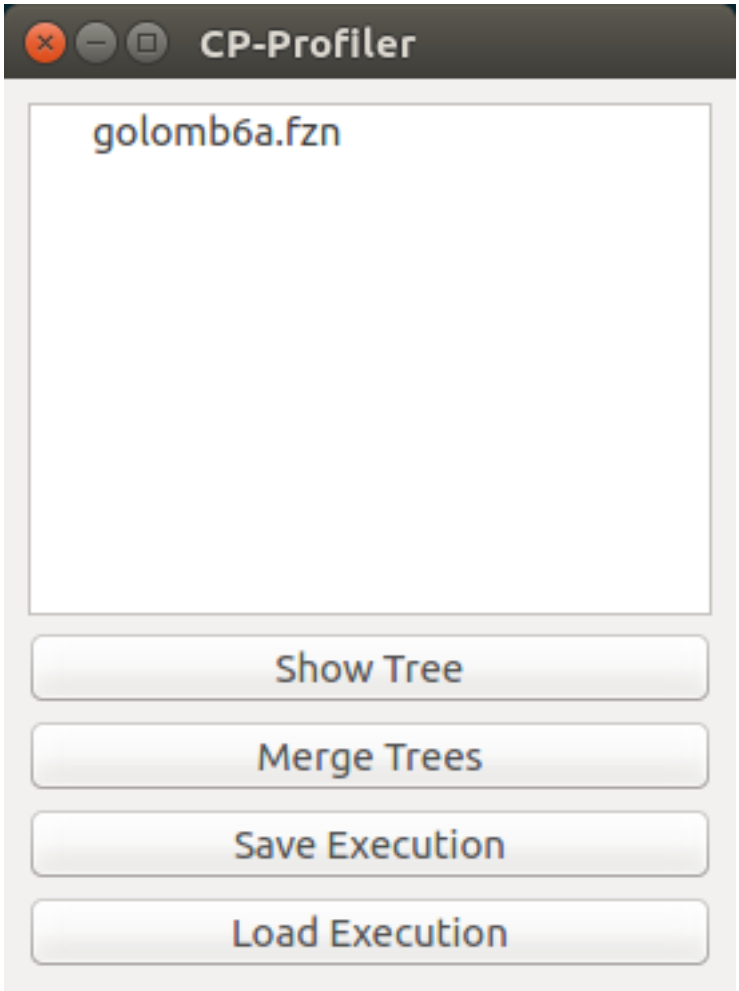
This integration is currently experimental.

3.4.1 Using the profiler

The profiler can be activated by selecting *Show search profiler* from the *View* menu, or by starting a solver with profiling enabled through the *Profile search option in the *Run* dropdown button menu. This starts a local TCP server listening on one of the available ports (6565 by default) which can be seen in at the bottom of the profiler window.

The solver must implement the profiling protocol (see below). Integration libraries are available if you wish to extend your solver to work with CP-Profiler.

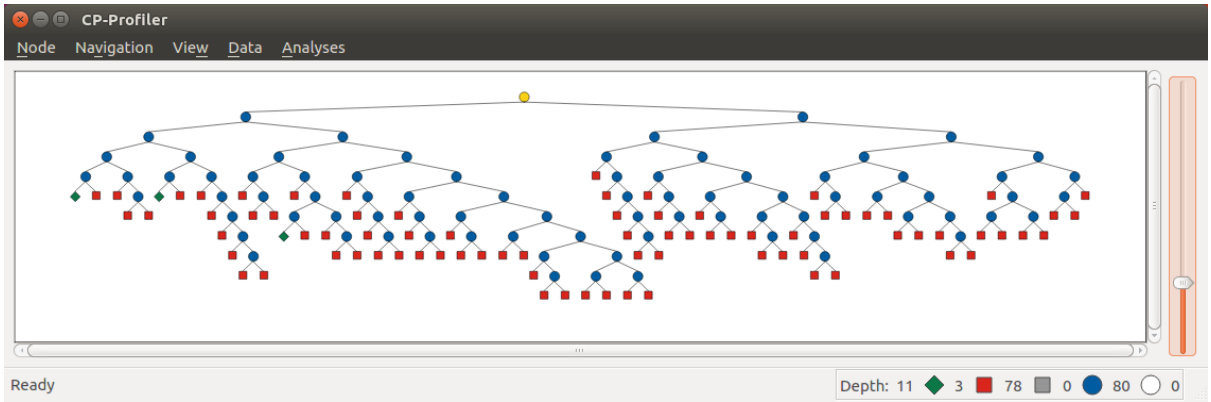
When a new execution is connected to the profiler it will be added to the list of executions displayed at the top of the profiler window. For example, in the image below execution *golomb6a.fzn* is shown to be added to the profiler. To display the execution (traditional visualisation), select its name from the list and click the *Show Tree* button. Note that the solver can still be running the execution, in which case the profiler will draw the search tree in real time.



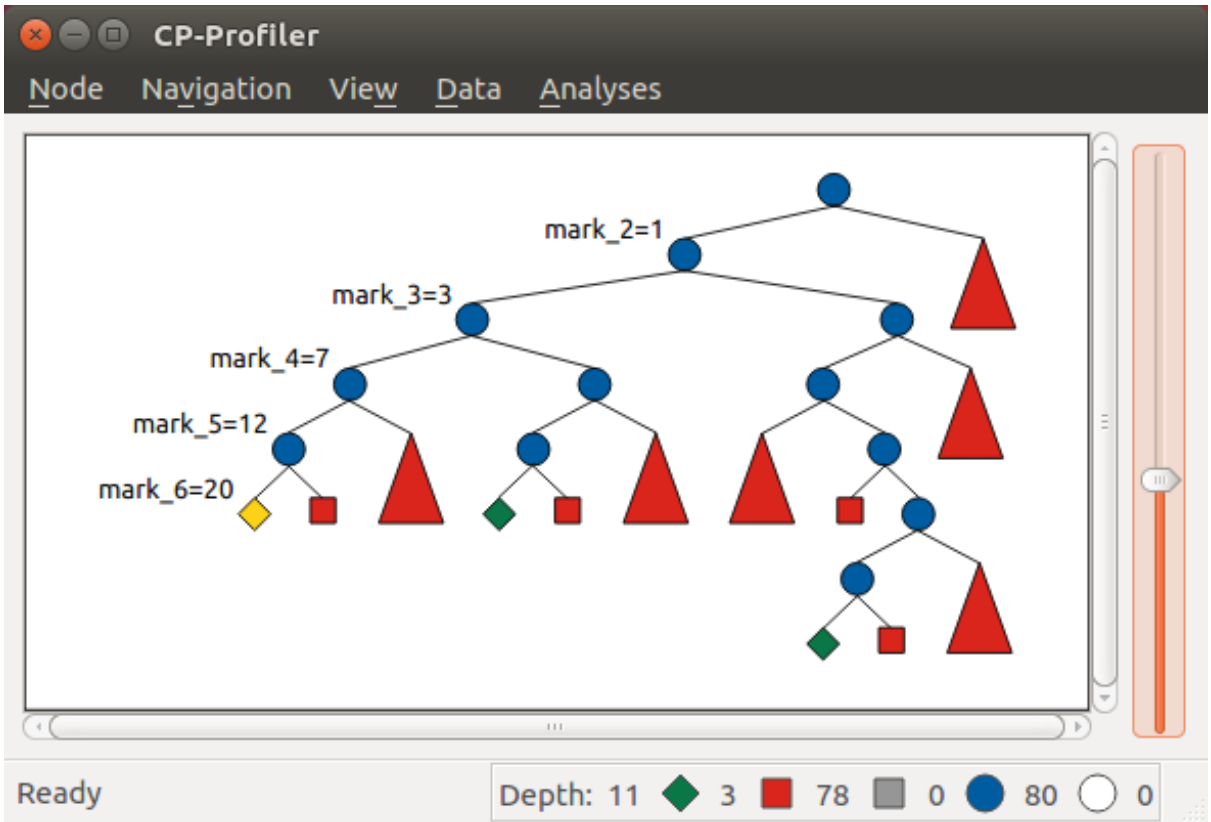
3.4.2 Traditional Tree Visualisation

The image below shows an example of a traditional (node-link) visualisation of the search tree. Different types of nodes are shown differently: branch (internal) nodes are shown as blue circles; nodes representing failures are shown as red squares; solution nodes are shown as green diamonds.

Note that the root of the tree is shown in gold colour indicating the currently selected node. Arrow keys on the keyboard allow the user to navigate the tree by changing which node is selected. `Down` navigates to the first child of the current node, `Shift+Down` -- to its last child, `Up` -- to its the parent, `Left` -- to its next sibling on the left, `Right` -- to its next sibling on the right. Additionally, pressing `R` will navigate to the root of the tree. The same actions are available under the **Navigation** menu.



If a subtree contains no solutions, it can be collapsed into a special single node displayed as a large red triangle. By default, the tree will be collapse failed subtrees automatically during its construction as a way to deal with large trees. The image below shows the same search tree as above, but with all failed subtrees collapsed.

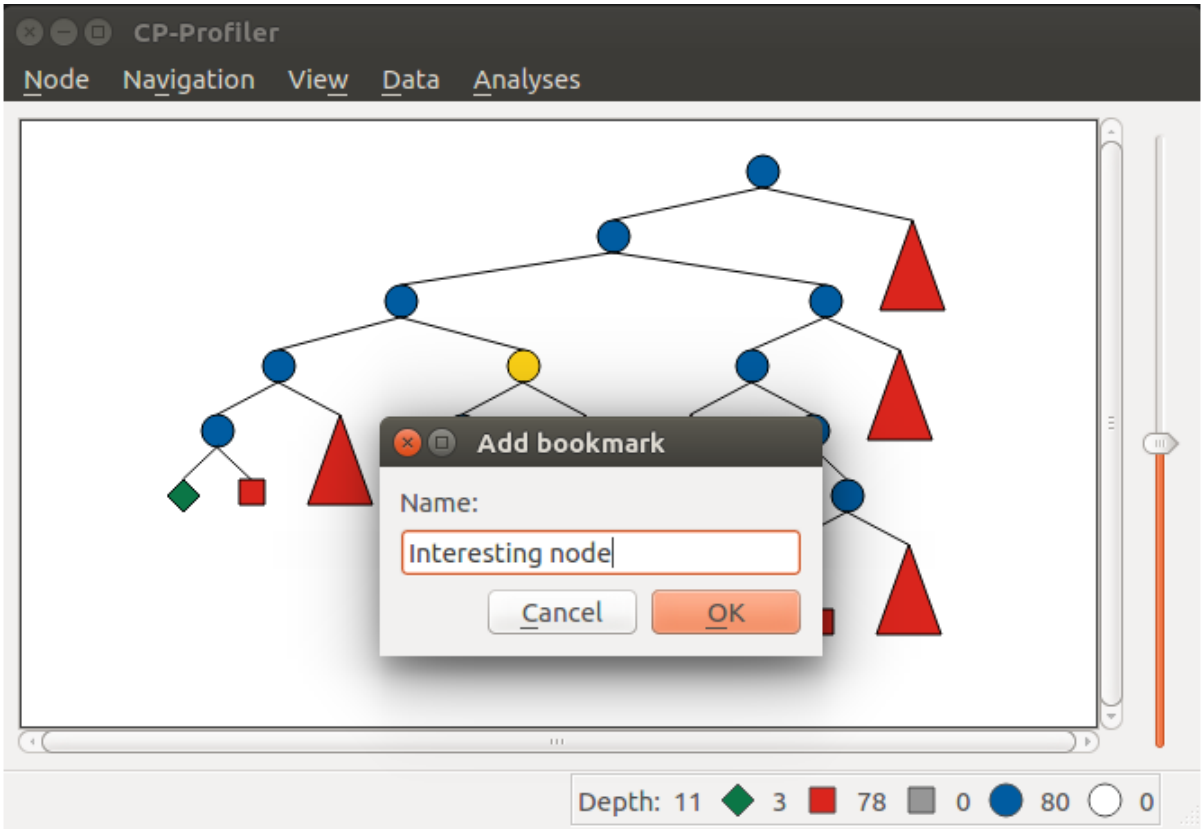


This view of the tree allows the user to show additional information for every node — its label, which usually represents the branching decision made by the solver to move from the parent node to its child. Pressing `L` on the keyboard will display labels for all descendants of the current node. `Shift+L` will display labels on the path to the current node. For example, the visualisation above shows branching decisions on the path from the first solution (shown as the new current node) to the root of the tree.

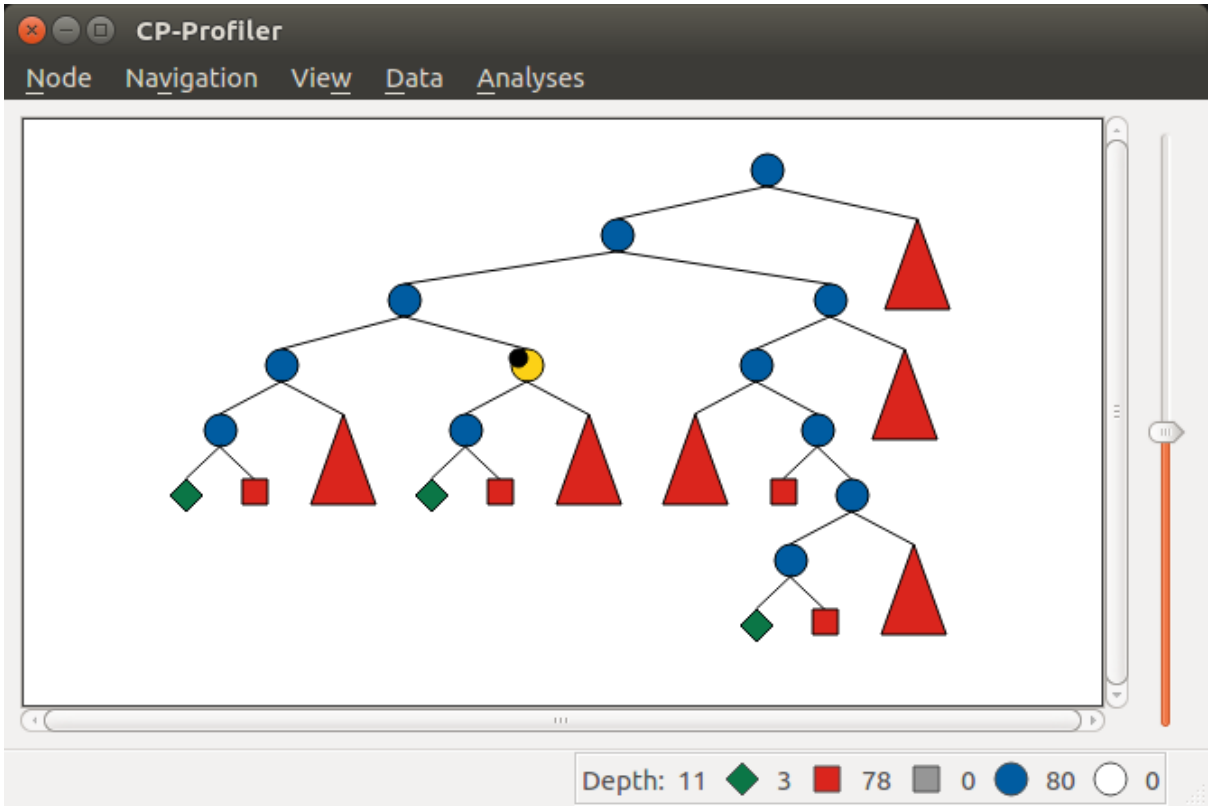
Status bar at the bottom of the window displays node statistics: the depth of the tree and the counts of different types of nodes. The scroll bar on the right lets the user to zoom in/out on the visualisation.

Adding Bookmarks

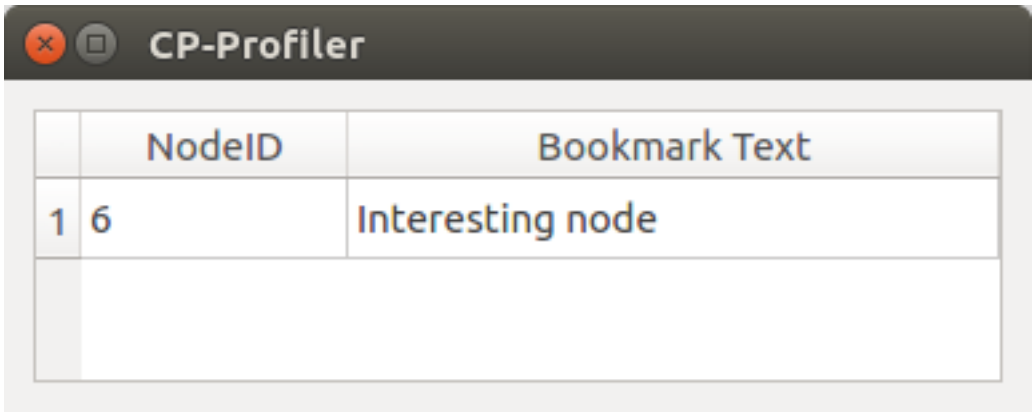
The user can *bookmark* any node on the tree associating it with an arbitrary text. This can be done by selecting the node of interest and clicking on **Add/remove bookmark** from the **Node** menu (shortcut: `Shift+B`). This will create a dialog as shown in the image below where the user can enter the bookmark text.



All bookmarked nodes are marked with a black circle on the traditional visualisation:



This message/node pair is also added to the bookmark list which can be retrieved from the **Data** menu under **Show bookmarks** action as shown below:



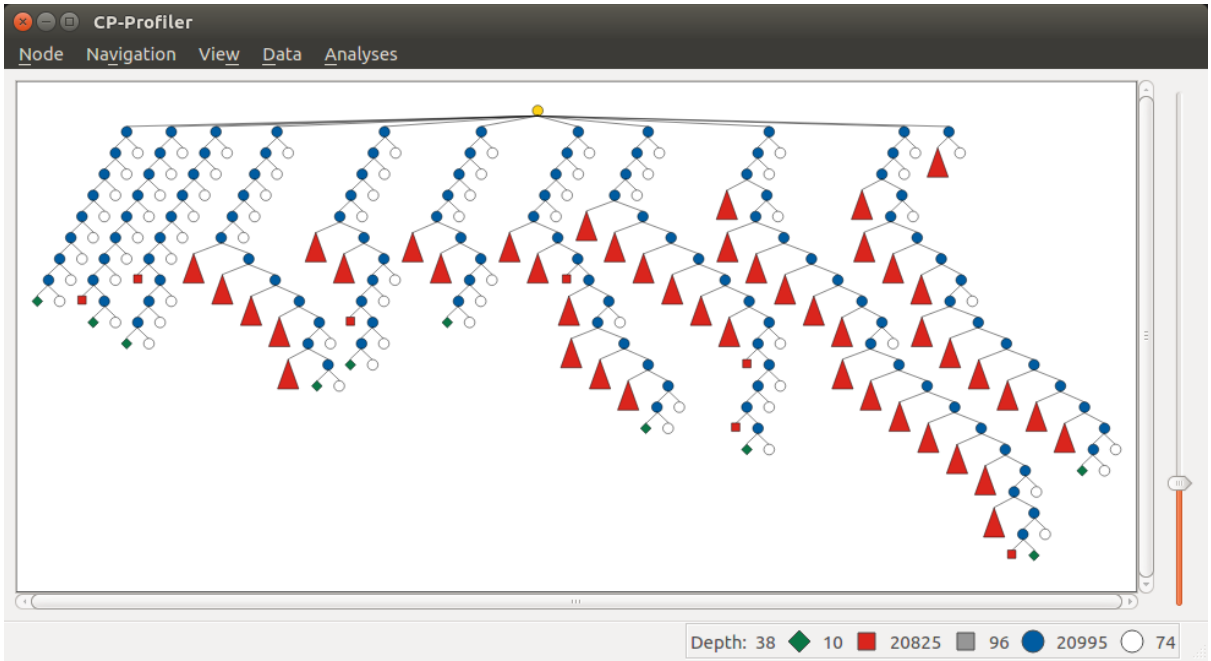
Note that each node is represented by a number -- the node's unique identifier in CP-Profiler (it is likely to be different from the identifier sent by the solver). Clicking on a row will navigate the visualisation to the corresponding bookmarked node.

3.4.3 Alternative Search Tree Visualisations

Lantern Tree

One disadvantage of the traditional visualisation described above is the fact that when the nodes are collapsed (necessary for large trees) they become indistinguishable from one another. *Lantern Tree* visualisation addresses this limitation: it draws collapsed nodes in a way that indicate the size of the underlying subtrees.

Consider, for example, the traditional visualisation of a restart-based execution below:

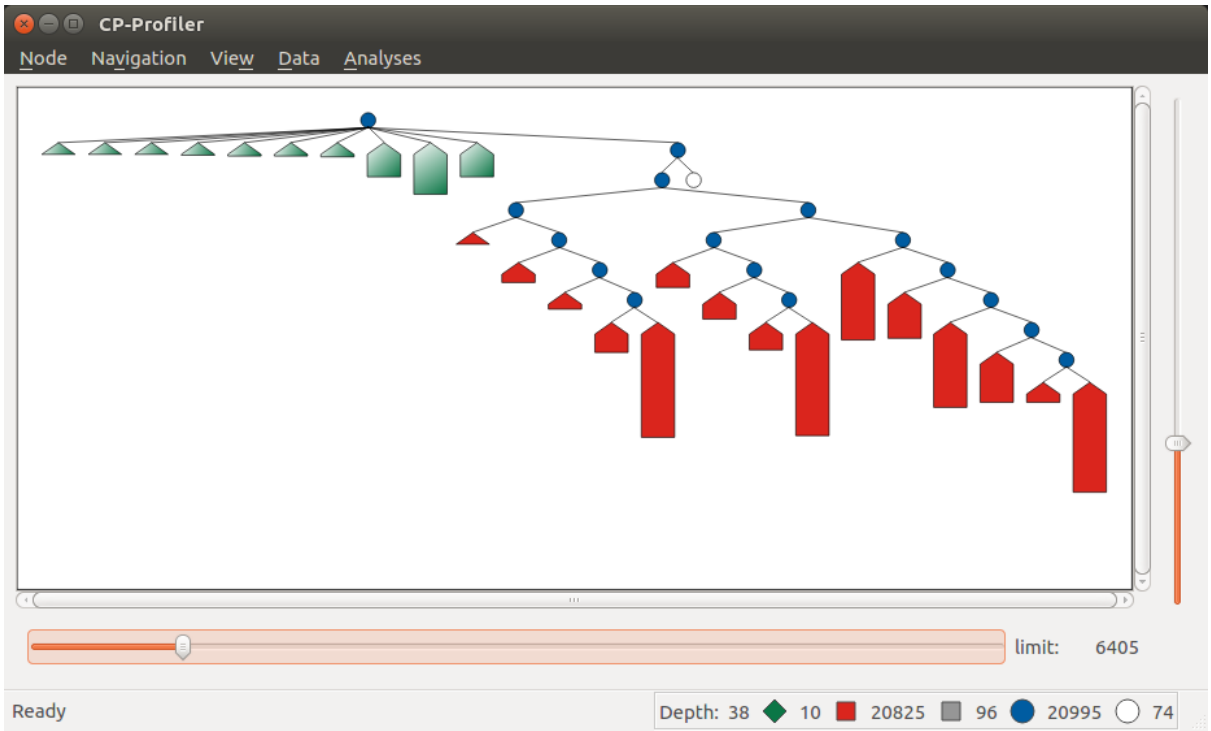


Each "restart" is represented by one of the subtrees attached to the "super" root node as a child. One can quickly determine that all but restarts terminated on finding a solution. The exception is the last restart in which no solution have been found (it corresponds to

the proof of optimality). However, not much can be concluded about the size of individual restarts, or the amount of search that they represent (at least not at a glance).

The following image shows the same restart tree using this alternative visualisation. All collapsed subtrees in this case are represented by ”lantern” nodes, which cna be seen as a triangle extended with a rectangular section at the bottom.

The lantern tree visualisation is parameterised by N, which is the maximum number of nodes a lantern node can represent before splitting into several smaller lanterns. Note that is rarely possible to collapse subtrees such that each subtree contains exactly N nodes. For this reason, lantern nodes (their rectangular sections) are dynamic in size and grow linearly with the actual number of nodes n in the subtree ($1 < n <= N$).



In this example, the use of the lantern tree reveal additional insight: all but a few last restarts required very little search, and by far the greatest amount of work was done during the ”proof of optimality” stage.

The lantern tree visualisation is available by enabling the **Lantern Tree View** option under the **View** menu (shortcut: `Ctrl+L`). Clicking on the same option again switches back to the traditional visualisation.

Pixel Tree

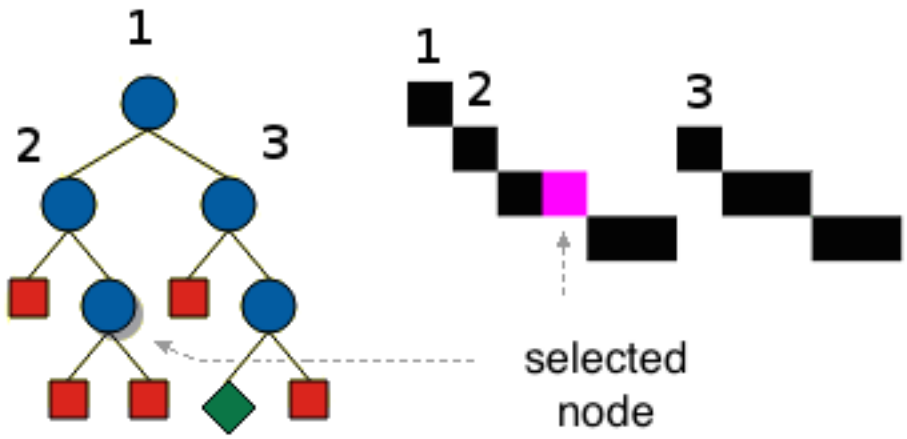
The pixel tree[1] and the icicle tree[2] visualisations are a quite different from the traditional visualisation in that they show links between nodes only implicitly, i.e. without drawing edges.

In the pixel tree view nodes are represented by squares (pixels), and edges only implicitly by the indentation between the squares. Parent nodes are placed immediately to the left of their subtree and the leaves of each subtree are grouped to the rightmost position.

The image below shows a correspondence between nodes in a traditional view and those

[1] Burch, M. et al., ”Indented Pixel Tree Plots”, 2010.
[2] Kruskal, J. B. et al., ”Icicle Plots: Better Displays for Hierarchical Clustering”, 1983.

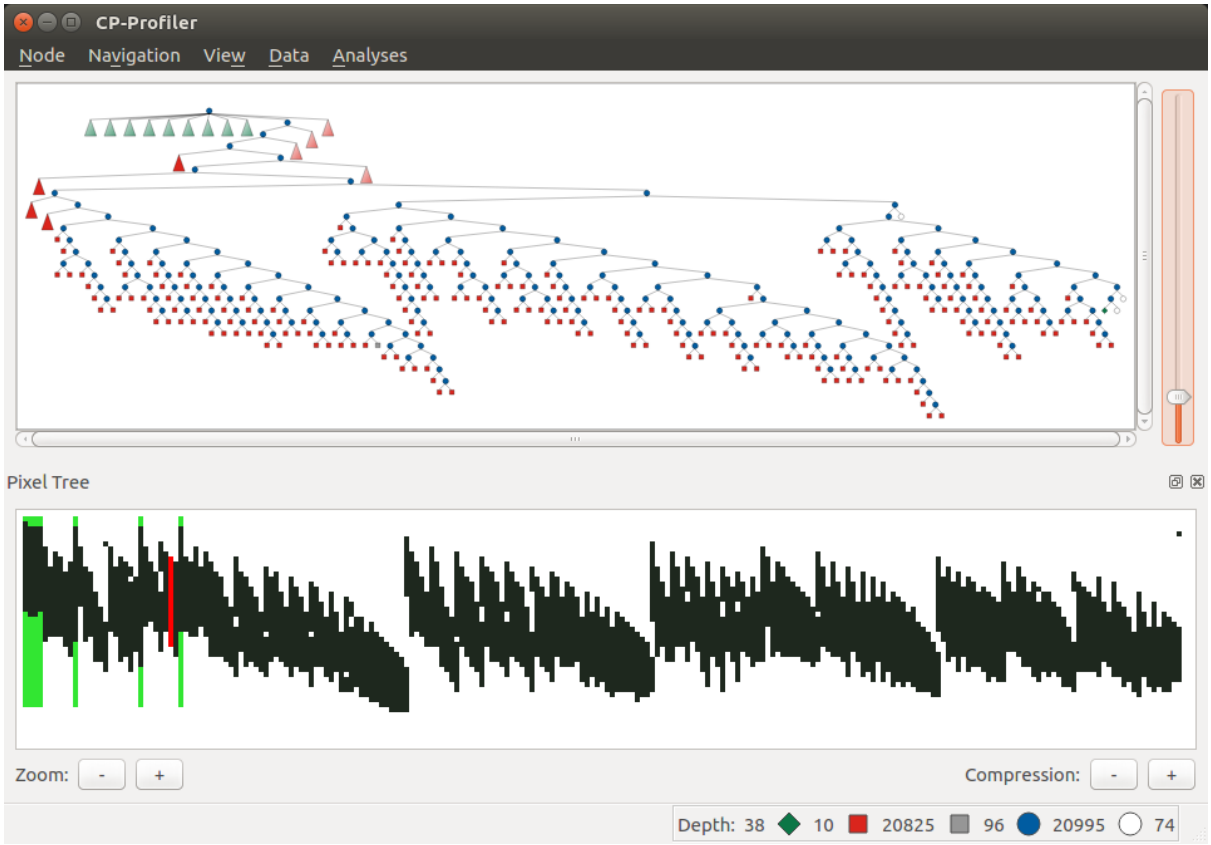
in the pixel tree view. Note that a green vertical line indicates a solution found at the corresponding node.



One of the main advantages of the pixel tree view is the ability to compress it while preserving large scale patterns and thus providing a good overview of the search. The compression is done by simply allowing multiple nodes on the same horizontal position.

In CP-Profiler the pixel tree can be shown alongside the traditional visualisation by enabling the **Pixel Tree View** option under the **View** menu (shortcut: `Shift+P`).

The following image shows the pixel tree view for the restart execution discussed earlier.



The pixel tree is compressed in this case to fit the entire tree in the window without any need for scrolling. Vertical green lines indicate that there is a solution node in the corresponding part of the tree. A vertical slices of the tree can be selected by clicking

and dragging with the left mouse key. Note that in this example a slice just before the final solution is selected (it is shown in red). Selecting a part of the tree in this way will update the traditional view to reveal the nodes that belong to that part, hiding the rest.

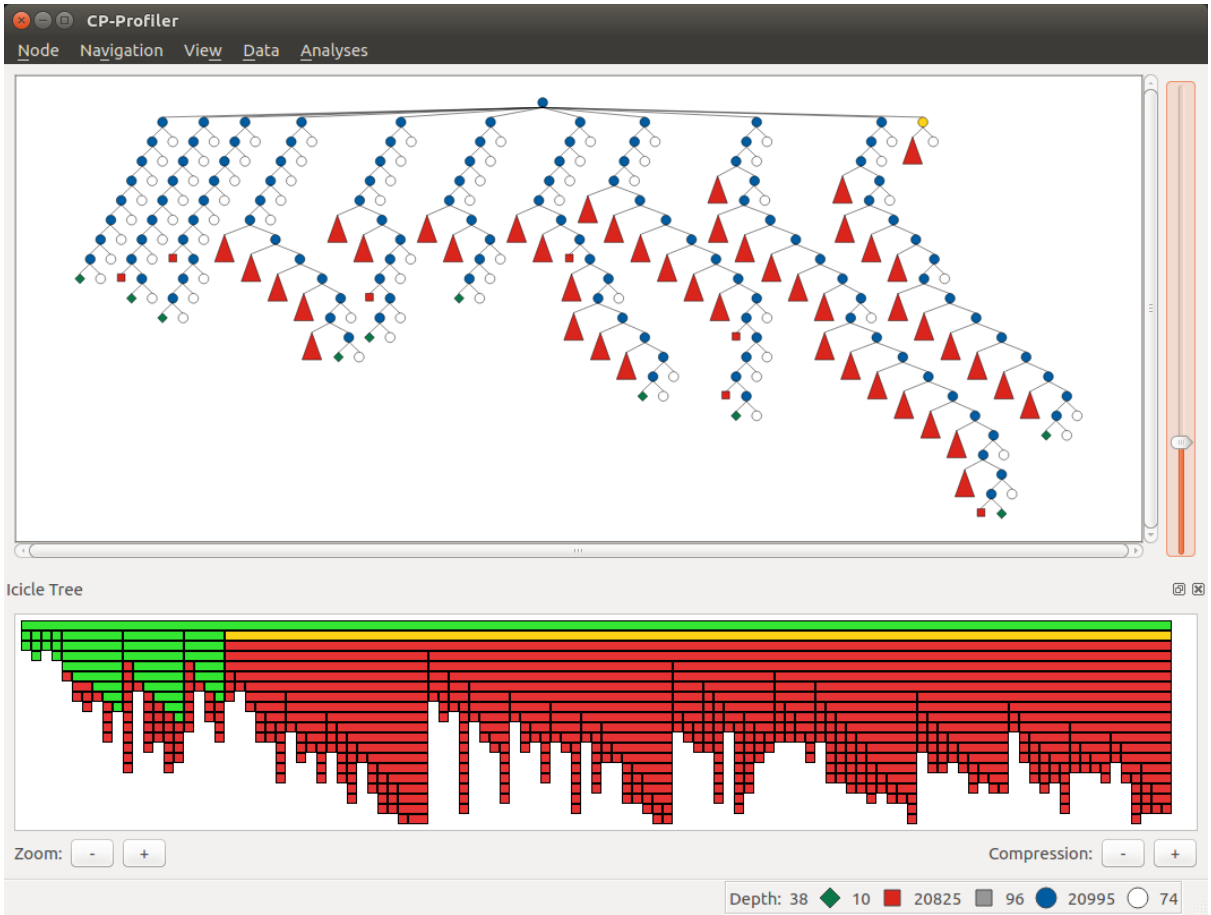
Icicle Tree

In the icicle tree parent nodes are positioned strictly above their children, i.e. they span horizontally over the same space as their children nodes. This property allows the user to quickly find a node's ancestry and determine the size of any subtree at a glance.

Icicle trees are compressed by cutting off leaf nodes, so that the corresponding parent nodes become the new leaf nodes. Doing so in a binary tree, for example, reduces the display area required for the visualisation by a factor of two. This action is repeated until the visualisation is of the desired size.

In CP-Profiler the icicle tree can be shown alongside the traditional visualisation by enabling the **Icicle Tree View** option under the **View** menu (shortcut: `Shift+I`).

The following image shows the icicle tree view for the restart execution from before.



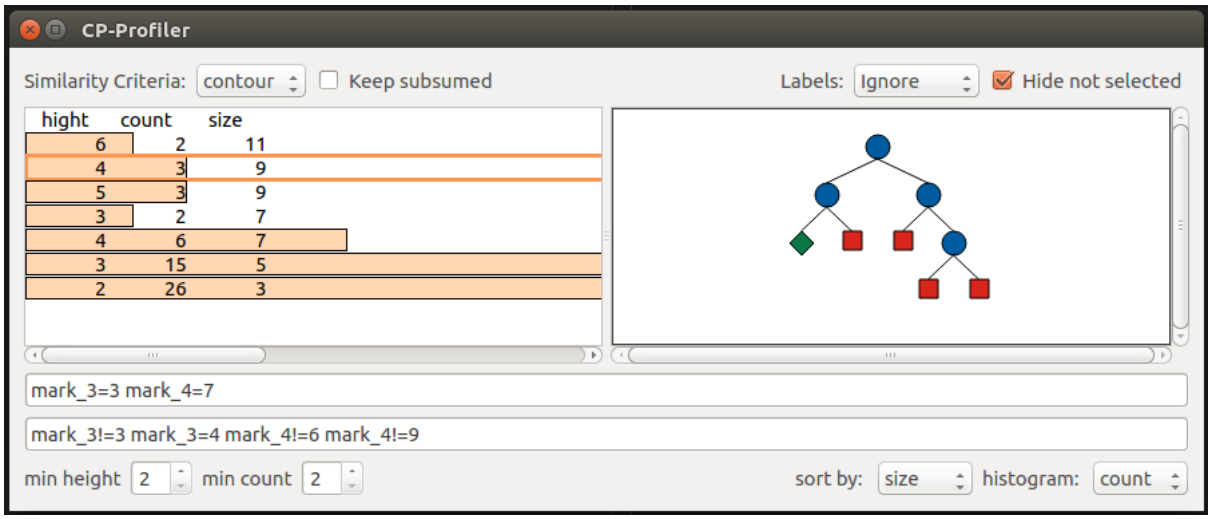
Again, the entire tree have been compressed to fit in the window. The following colour scheme is used: each icicle element that represents a subtree containing a solution is shown in green colour. All other nodes are shown in red as they represent entirely failed subtrees. Additionally elements of the icicle tree can be selected (shown in yellow colour) to show the corresponding subtree (node) in the traditional visualisation. Note that this relationship is two-way bound: selecting a different node in the traditional visualisation will update the icicle tree view correspondingly.

3.4.4 Similar Subtree Analysis

This analysis allows users to find similarities within a single search tree.

The analysis can be initiated by selecting **'Similar Subtrees'** from the menu **'Analyses'** (shortcut: `Shift+S`). The image below shows the result of running the analysis on the search tree above. Horizontal bars on the left lists all similarities (patterns) found in the tree. Here, the lengths of the bars are configured to indicate how many subtrees belong to a particular pattern (*count*). Additionally the bars are sorted so that the patterns with subtrees of larger *size* appear at the top. Another property of a pattern is its *height*, which indicates the height/depth of subtrees that the pattern represent.

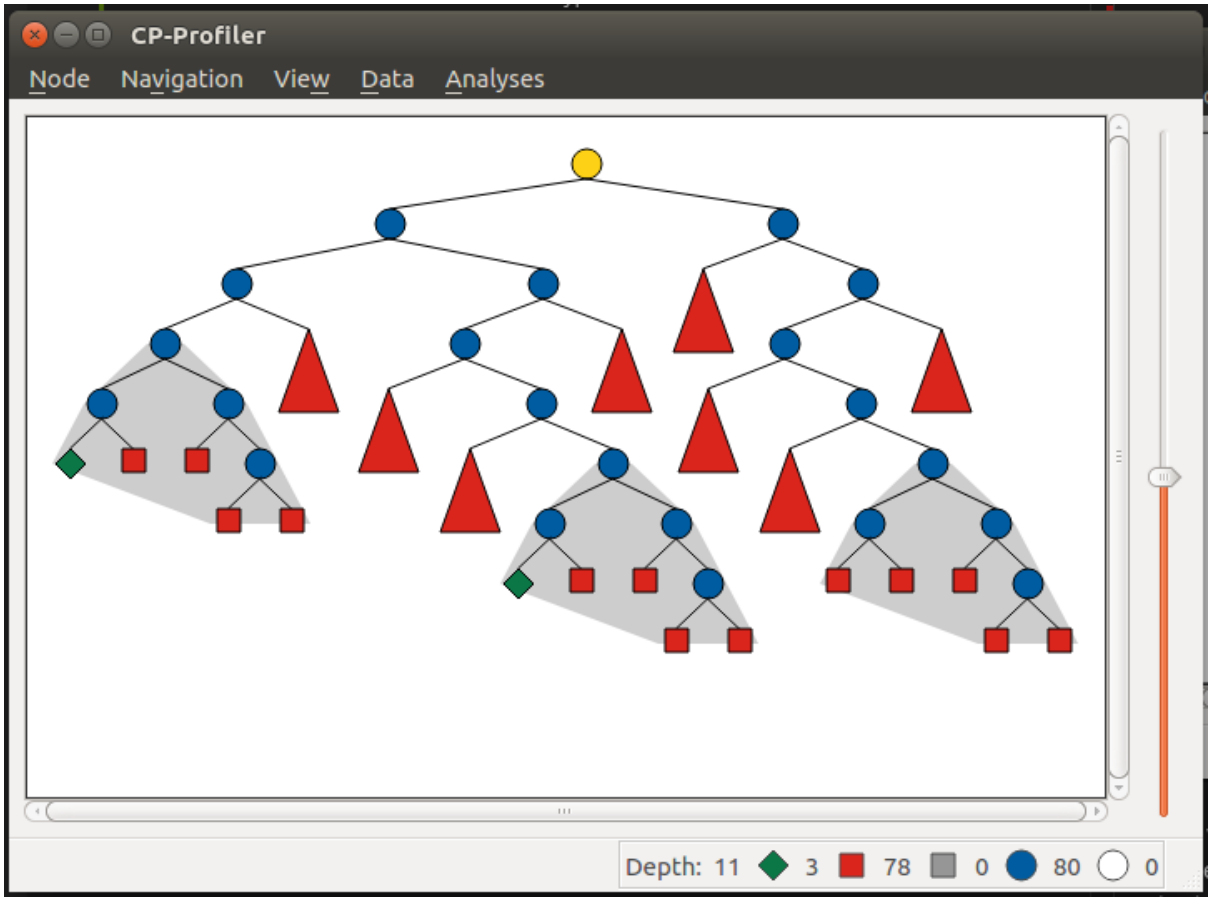
Note that in this example the second (from the top) pattern is currently selected (shown with orange outline). The view on the right shows a "preview" (traditional visualisation) of one of the subtrees representing the selected pattern. The two rows below the show the result of computing the difference in labels on the path from the root to two of the subtrees representing the pattern (in this case it is the first two subtrees encountered when the tree is traversed in the depth-first-search order).



TODO: discuss the two criteria

Changing the configuration menu at the bottom of the window, the user can filter the list of patterns based on their *count* and *height* values. They way the length of horizontal bars is determined and the sorting criteria can also be specified there.

Whenever a pattern on the left hand side is selected, the corresponding subtrees will be highlighted on the traditional visualisation by drawing their outlines in grey. Additionally, if the option *Hide not selected* is enabled (top of the window), everything but the subtrees of the current pattern will be collapsed as shown in this example.



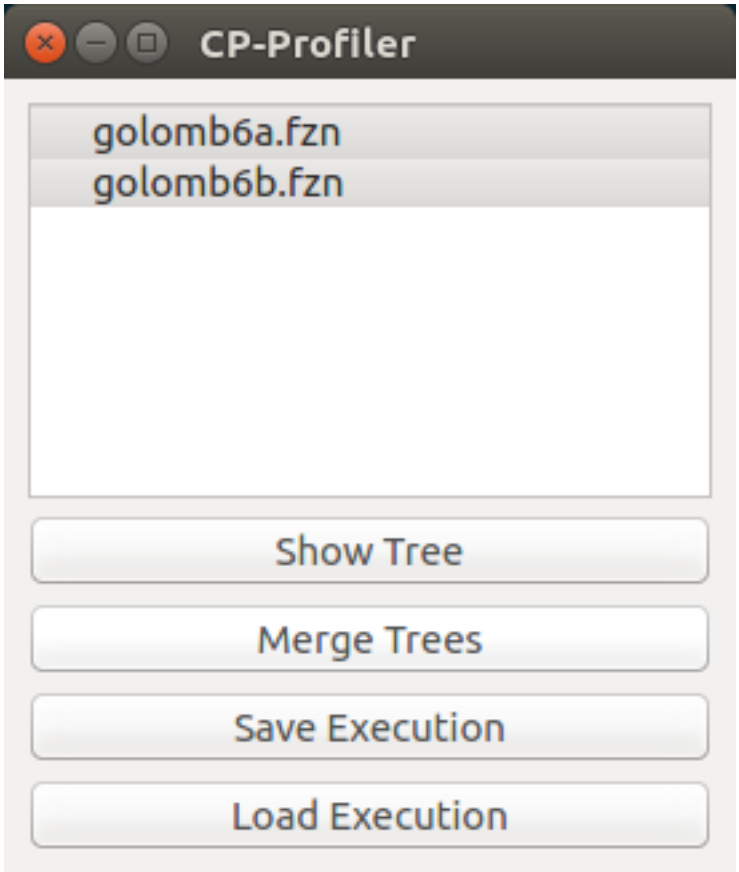
Elimination of Subsumed Patterns

A pattern P is said to be subsumed by one or more other patterns if subtrees of those patterns contain all of the subtrees of P as descendants. By default, the analysis is configured to eliminate all subsumed patterns. Enabling the *Keep subsumed* option (top of the window) will display the subsumed patterns along with non-subsumed patterns.

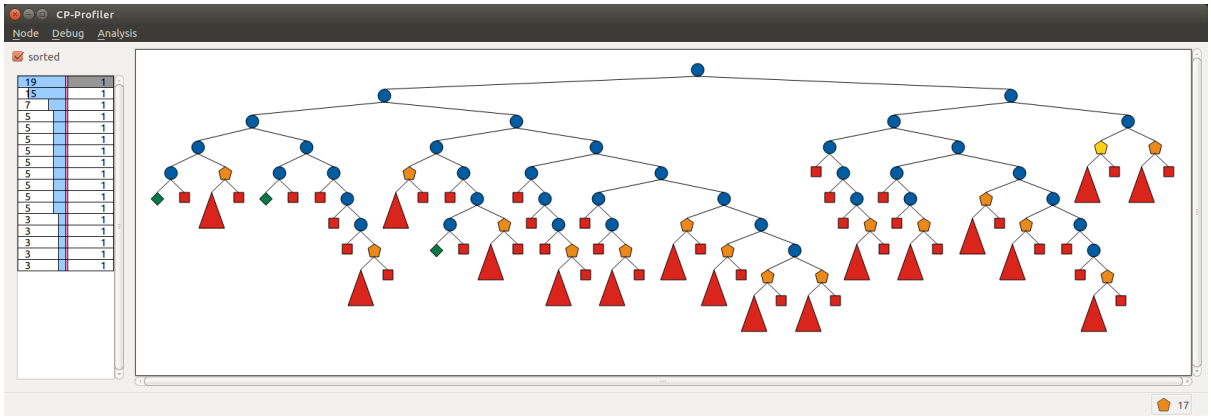
3.4.5 Execution Comparison by Merging

CP-Profiler allows users to compare two executions by merging the corresponding search trees. This is accomplished traversing both trees in lockstep in a depth-first-search manner, starting at the root nodes and comparing corresponding node at each step. A node is created on the resulting merged tree for every pair of nodes that are deemed identical. Once a difference is found, a special *pentagon* node is created, whose two children are the roots of the corresponding different subtrees.

This comparison of two execution can be done in CP-Profiler by selecting the two executions of interest from the list of executions and clicking the *Merge Trees* button.



The image below shows an example of the resulting merged tree. As indicated in the status bar, this tree contains 17 (orange) pentagons. The panel on the left lists all pentagons displaying the number of nodes on corresponding left/right subtrees. By default, the list is sorted by the absolute difference between the number of nodes. Note that in this case the top pentagon from the list (19 -- 1) is selected, which automatically selects the corresponding node in the visualisation (second last pentagon node, highlighted in yellow).



3.4.6 Replaying Search

The two main factors that affect a solver execution are propagation and search. It can be useful to consider propagation and search in isolation (e.g. to aid merging), which can be done using the technique called *search replaying*. The idea is to force the same search

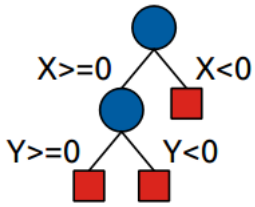
decisions by first performing one execution normally and recording the tree structure and branching decisions (labels) to form a *search log*. Then the *search log* log is used to drive the search in another execution.

The protocol for replaying

The following information (*entry*) is recorded in a *search log* per node: node's unique identifier, the number of children, and the search decisions associated with each child node. Each entry, in turn, is divided into *tokens* separated by a whitespace character. The first two tokens are the node's identifier and the number of children (0 for a leaf node). Entries of branch nodes are extended by a pair of tokens per child, containing the child's identifier and its branching decision.

Example

Consider the following basic search tree:



The corresponding *search log* is as follows:

```
0 2 1 X>=0 2 X<0
1 2 3 Y>=0 4 Y<0
3 0
4 0
2 0
```

3.4.7 Nogood Analysis

This analysis is based on the idea of comparing (*merging*) two execution of the same model (instance) using the same search (via *search replaying*), one with nogood learning and the other one -- without (resulting in *learning* and *non-learning* executions). Since the only difference between the two executions will be in the higher propagation strength in the learning execution due to no-goods, the resulting merged tree will only have pentagons of the form *1/n* (one failure node corresponding to a subtree of *n* nodes in non-learning execution). *n* in this case is called *search reduction* and it is attributed to all previously generated no-goods that contributed to this early failure. The analysis aggregates this information throughout the execution, which results in a *nogood ranking* based on how much search reduction each nogood contributed to.

The following steps can be performed to execute the nogood analysis:

1. An instance is executed with a learning solver with profiling on.

For example: `chuffed pizza.fzn --cpprofiler`.

2. The resulting search is saved to a file using CP-Profiler.
3. The same instance is executed without learning (preferably by the same solver, but a solver with similar propagation will produce similar result).

For example: `gecode --cpprofiler_replay pizza.search pizza.fzn`

4. The two resulting trees are merged using CP-Profiler.
5. **Nogood analysis** option is selected from the **Analyses** menu in CP-Profiler.

The image below shows a possible result window, in which the ranking is displayed in a tabular form with one row per nogood:

CP-Profiler

	NodeID	Total Reduction	Count	Clause
365	5687	479	16	how_7<=1 X_INTRODUCED_80_i>=1 how_1>=2 X_INTRODUCED_79_i>=1
1872	4281	476	6	how_1<=3 X_INTRODUCED_204=true how_4==2
1351	5092	451	12	how_6!=3 X_INTRODUCED_409=false X_INTRODUCED_101_i>=1 X_INTRODUCED_1...
1282	1549	440	22	X_INTRODUCED_84=false how_1!=2 how_1<=1
1990	4987	405	36	how_5<=3 X_INTRODUCED_129_i>=1 X_INTRODUCED_124_i>=1 X_INTRODUCED_1...
83	6297	397	2	X_INTRODUCED_80_i<=0 X_INTRODUCED_56_i>=1 X_INTRODUCED_124_i>=1 X_INT...
710	3085	380	45	X_INTRODUCED_58=false how_1!=1 how_1<=0
1385	4684	359	12	X_INTRODUCED_84=false how_6!=2 how_6<=1
1148	2695	355	19	X_INTRODUCED_103=false how_6!=3 how_6<=2
1208	256	351	61	X_INTRODUCED_81=false how_3!=2 how_3<=1
1183	2903	332	45	X_INTRODUCED_63=false how_1!=1 how_1<=0
347	5219	323	3	X_INTRODUCED_62=false X_INTRODUCED_101_i>=1 X_INTRODUCED_124_i>=1 ho...
1110	1192	312	65	how_10>=-3 X_INTRODUCED_225_i>=1 X_INTRODUCED_246<=0 X_INTRODUCED_2...
445	1699	311	11	how_1!=3 how_4<=-4 X_INTRODUCED_99_i>=1 X_INTRODUCED_101_i>=1
2000	1818	311	60	X_INTRODUCED_78=false how_3!=2 how_3<=1
931	4693	308	11	X_INTRODUCED_208=true X_INTRODUCED_409=false X_INTRODUCED_126_i>=1 X_...
9	6915	306	3	how_4>=-2 X_INTRODUCED_385=false X_INTRODUCED_56_i>=1 X_INTRODUCED_1...

Save Nogoods

The first column identifies the (failed) node at which the nogood has been generated. The next column shows search reduction (in the number of nodes) attributed to this nogood. This is followed by the column that shows the number of times the nogood contributed to search reduction, that is, the number of *1/n* pentagons that were partially or solely responsible for the early failure. Finally, the last column shows the nogood itself in a form of a clause (disjunction of literals) where its literals are separated by a whitespace character.

Note that dy default the entries are sorted based on search reduction. The sorting criteria can be changed by clicking on corresponding column headers.

This result can be saved to a file (by clicking the **Save Nogoods** button) for further examination. Each row in the resulting file will correspond to one nogood, and the tab character (`\t`) will separate different columns.

3.4.8 The protocol (high level)

The following describes the protocol that a solver must implement to communicate with the profiler.

The protocol distinguishes between the following types of messages: **Start**, **Restart**, **Done**, and **Node**.

The **Start** message is sent at the beginning of the execution. The message has two optional parameters:

- `Name`: execution's descriptor to be presented to the user (e.g. the model's name)
- `Execution ID`: globally unique identifier used to distinguish between different executions.

The **Restart** message is sent whenever a solver performs a restart in case of a restart-based search.

The **Done** message is sent to the profiler at the end of the execution to indicate that no further nodes should be expected.

The **Node** message is sent whenever a new node is explored by the solver and contains information necessary for reconstructing the search tree. The required parameters are:

- `Node ID`: unique node identifier in the execution.
- `Parent ID`: the identifier (`Node ID`) of the parent node. A root node can have an identifier of *-1*.
- `Alternative`: the node's position relative to its siblings; for the left-most child it is `0`, for the second left-most it is `1` etc.
- `Number of Children`: the number of children nodes. If not known, can be set to `0` and the node will be extended with extra children later on if necessary.It is, however, advisable to specify the number of children the profiler should expect (for example, the yet to arrive nodes can be visualised to give a better idea about the search).
- `Status`**: allows to distinguish between different types of nodes. Supported values are:**
  - *BRANCH*: internal node in the tree;
  - *SOLUTION*: leaf node representing a solution;
  - *FAILURE*: leaf node representing a failure;
  - *SKIPPED*: leaf node representing unexplored search space due to back-jumping.

Example. The following sequence of nodes (excluding the *Start* and *Done* messages) produces the simple tree with three nodes:

Label	Node ID	Parent ID	Alternative	Number of Children	Status
Root	0	-1	-1	2	BRANCH
Failure	1	0	0	0	FAILED
Solution	2	0	1	0	SOLVED

3.4.9 The protocol (low level)

Each message starts with a four-byte integer encoding the size of the remainder of the message in bytes. This is followed by a single byte encoding the type of the message. The corresponding values are: `Node`: `0`, `Done`: `1`, `Start`: `2`, `Restart`: `3`.

Node message

In case the message is of the type `Node`, the following fields are added in order: `id`, `pid`, `alt`, `children` and `status`.

Node identifiers `id` and `pid` are represented using three four-byte integers: first identifies the identifier of the node within a thread, the second —the identifier of the restart (in a restart-based search), and the third —the identifier of the thread. The `alt` and `children` fields are represented by a single four byte integer each. The `status` field is represented by a single byte; its possible values are: *SOLVED*: 0, *FAILED*: 1, *BRANCH*: 2, *SKIPPED*: 3. All multi-byte integer values are encoded using the *two's compliment* notation in the *big-endian order*.

Additionally, each node message can contain the following optional fields:

- `label`: branching decision (or any arbitrary string to be drawn together with the node);
- `nogood`: string representation of a newly generated nogood in a learning solver;
- `info`: arbitrary information about the node.

Field identifiers and their sizes in bytes:

field name	field id	size (bytes)
`id`	n/a	12
`pid`	n/a	12
`alt`	n/a	4
`children`	n/a	4
`status`	n/a	1
`label`	0	any
`nogood`	1	any
`info`	2	any

Example. The following is a possible correspondence between a solver and the profiler that generates a trivial tree. The order in which different fields arrive is shown from top to bottom (rows are numbered for convenience).

Message 1:

Row	Bytes	Interpretation
1	`00 00 00 21`	message size (33)
2	`02`	message type (*START*)
3	`02`	field (*info*)
4	`00 00 00 1B`	string size (27)
5	`7b 22 6e 61 6d 65 22 3a 20 22 6d 69 6e 69 6d 61 6c 20 65 78 61 6d 70 6c 65 22 7d`	'{"name": "minimal example"}'

Message 2:

Row	Bytes	Interpretation
6	`00 00 00 2B`	message size (43)
7	`00`	message type (**NODE**)
8	`00 00 00 00`	node id (0)
9	`FF FF FF FF`	node restart id (-1)
10	`FF FF FF FF`	node thread id (-1)
11	`FF FF FF FF`	parent id (-1)
12	`FF FF FF FF`	parent restart id (-1)
13	`FF FF FF FF`	parent thread id (-1)
14	`FF FF FF FF`	alternative (-1)
15	`00 00 00 02`	children (2)
16	`02`	status (*BRANCH*)
17	`00`	field (label)
18	`00 00 00 04`	string size (4)
19	`52 6f 6f 74`	'Root'

Message 3:

Row	Bytes	Interpretation
20	`00 00 00 01`	message size (1)
21	`01`	message type (**DONE**)

CHAPTER 3.5

Solving Technologies and Solver Backends

The `minizinc` tool can use various solver backends for a given model. Some solvers are separate executables that are called by `minizinc`; other solvers are part of the `minizinc` binary (either hard-coded or loaded as a dynamic library or "plugin"). In the former case, a temporary FlatZinc file is created and passed to the solver binary; in the latter one, the flattened model is passed to the backend directly in memory. Some solvers are part of the binary MiniZinc distribution, others have to be installed separately. You can find instructions for installing these solvers from source code and integrating into the `minizinc` tool in *Installation from Source Code* (page 227). This chapter summarises usage options available for various target solvers.

The help text of `minizinc` shows a list of configured solver backends and their tags. You can see solver-specific command-line options by running

```
$ minizinc --help <solver-id or tag>
```

3.5.1 Constraint Programming Solvers

Constraint Programming is the 'native' paradigm of MiniZinc. Below we discuss most common CP solvers. For their performance, consult MiniZinc Challenges (https://www.minizinc.org/challenge/).

3.5.1.1 Gecode

Gecode is an open-source constraint programming system (see https://www.gecode.org). It supports many of MiniZinc's global constraints natively, and has support for integer, set and float variables.

Gecode supports a number of constraints and search annotations that are not part of the MiniZinc standard library. You can get access to these by adding `include "gecode.mzn";` to your model. The additional declarations are documented in 节 4.2.4.

3.5.1.2 Chuffed

Chuffed is a constraint solver based on *lazy clause generation* (see https://github.com/chuffed/chuffed). This type of solver adapts techniques from SAT solving, such as conflict clause learning, watched literal propagation and activity-based search heuristics, and can often be much faster than traditional CP solvers.

In order to take full advantage of Chuffed's performance, it is often useful to add a search annotation to the model (see 搜索 (page 83)), but allow Chuffed to switch between this defined search and its activity-based search. In order to enable this behaviour, use the `-f` (free search) command line option or select *Free search* in the solver configuration pane of the MiniZinc IDE.

Chuffed supports a number of additional search annotations that are not part of the MiniZinc standard library. The additional declarations are documented in 节 4.2.5.

3.5.1.3 OR-Tools

OR-Tools is a powerful open-source CP/SAT/LP solver (see https://developers.google.com/optimization/). It supports many of MiniZinc's global constraints natively. It often performs better multi-threaded (option `-p`) so it can employ various solving technologies. A search annotation (see 搜索 (page 83)) can be useful, however allowing OR-Tools to mix the prescribed strategy with its own (option `-f`) usually is best, analogously to Chuffed.

3.5.2 Mixed-Integer Programming Solvers

MiniZinc has built-in support for Mixed Integer Programming solvers. If you have any MIP solver installed (and MiniZinc was compiled with its support), you can run a model using MIP like this on the command line:

```
minizinc --solver mip -v -s -a model.mzn data.dzn
```

Of course, you can also select a particular solver, e.g. Gurobi (in case it is available):

```
minizinc --solver gurobi -v -s -a model.mzn data.dzn
```

3.5.2.1 MIP-Aware Modeling (But Mostly Useful for All Backends)

Avoid mixing positive and negative coefficients in the objective. Use 'complementing' variables to revert sense.

Avoid nested expressions which are hard to linearize (decompose for MIP). For example, instead of

```
constraint forall(s in TASKS)(exists([whentask[s]=0] ++
  [whentask[s]>= start[s]+(t*numslots) /\ whentask[s]<=stop[s]+(t*numslots) | t in␣
↪0..nummachines-1]));
```

prefer the tight domain constraint

```
constraint forall(s in TASKS)(whentask[s] in
  {0} union array_union([ start[s]+(t*numslots) .. stop[s]+(t*numslots) | t in 0..
↪nummachines-1]));
```

To avoid **numerical issues**, make variable domains as tight as possible (compiler can deduce bounds in certain cases but explicit bounding can be stronger). Try to keep magnitude difference in each constraint below 1e4. Especially for variables involved in logical constraints, if you cannot reduce the domains to be in +/-1e4, consider indicator constraints (available for some solvers, see below), or use the following trick: instead of saying `b=1 -> x<=0` where x can become very big, use e.g. `b=1 -> 0.001*x<=0.0`. Especially for integer variables, the domain size of 1e4 should be an upper bound if possible -- what is the value of integrality otherwise? Avoid large coefficients too, as well as large values in the objective function. See more on tolerances in the Solver Options section.

Example 1: *basic big-M constraint vs implication*. Instead of `<expr> <= 1000000*y` given `var 0..1: y` and where you use the 'big-M' value of 1000000 because you don't know a good upper bound on `<expr>`, prefer `y=0 -> <expr> <= 0` so that MiniZinc computes a possibly tighter bound, and consider the above trick: `y=0 -> 0.0001*<expr> <= 0.0` to reduce magnitudes.

Example 2: *cost-based choice*. Assume you want the model to make a certain decision, e.g., constructing a road, but then its cost should be minimal among some others, otherwise not considered. This can be modeled as follows:

```
var 0..1: c;                                          %% Whether we construct␣
↪the road
var int: cost_road = 286*c + 1000000*(1-c);
var int: cost_final = min( [ cost_road, cost1, cost2 ] );
```

Note the big coefficient in the definition of `cost_road`. It can lead to numerical issues and a wrong answer: when the solver's integrality tolerance is 1e-6, it can assume `c=0.999999` as equivalent to `c=1` leading to `cost_road=287` after rounding.

A better solution, given reasonable bounds on `cost1` and `cost2`, is to replace the definition as follows:

```
int: cost_others_ub = 1+2*ub_array( [cost1, cost2] );    %% Multiply by 2 for a␣
↪stronger LP relaxation
var int: cost_road = 286*c + cost_others_ub*(1-c);
```

3.5.2.2 Useful Flattening Parameters

The following parameters can be given on the command line or modified in `share/minizinc/linear/options.mzn`:

```
-D nSECcuts=0/1/2                          %% Subtour Elimination Constraints,␣
↪see below
-D fMIPdomains=true/false                  %% The unified domains feature, see␣
↪below
-D float_EPS=1e-6                          %% Epsilon for floats' strict␣
↪comparison,
                                           %% used e.g. for the following cases:
```

```
                                                %% x!=y, x<y, b -> x<y, b <-> x<=y
-DfIndConstr=true -DfMIPdomains=false           %% Use solver's indicator constraints,␣
↪see below
-DMinMaxGeneral=true                            %% Send min/max constraints to the␣
↪solver (Gurobi only)
-DQuadrFloat=false -DQuadrInt=false             %% Not forward float/integer␣
↪multiplications for MIQCP backends, see below
-DUseCumulative=false                           %% Not forward cumulative with fixed␣
↪durations/resources (SCIP only)
-DUseOrbisack=false                             %% Not forward lex_lesseq for binary/
↪bool vectors (SCIP only)
-DOrbisackAlwaysModelConstraint=true            %% lex_lesseq ignores being in␣
↪symmetry_breaking_constraint() (SCIP only)
                                                %% Required for SCIP 7.0.2, or use␣
↪patch: http://listserv.zib.de/pipermail/scip/2021-February/004213.html
-DUseOrbitope=false                             %% Not forward lex_chain_lesseq for␣
↪binary/bool matrices (SCIP only)
--no-half-reifications                          %% Turn off half-reification (full␣
↪reification was until v2.2.3)
```

3.5.2.3 Some Solver Options and Changed Default Values

The following command-line options affect the backend or invoke extra functionality. Note that some of them have default values which may be different from the backend's ones. For example, tolerances have been tightened to enable more precise solving with integer variables and objective. This slightly deteriorates performance on average, so when your model has moderate constant and bound magnitudes, you may want to pass negative values to use solver's defaults.

```
-h <solver-tag>    full description of the backend options
--relGap <n>       relative gap |primal-dual|/<solver-dep> to stop. Default 1e-8,␣
↪set <0 to use backend's default
--feasTol <n>      primal feasibility tolerance (Gurobi). Default 1e-8
--intTol <n>       integrality tolerance for a variable. Default 1e-8
--solver-time-limit-feas <n>, --solver-tlf <n>
                   stop after <n> milliseconds after the first feasible solution␣
↪(some backends)
--writeModel <file>
                   write model to <file> (.lp, .mps, .sav, ...). All solvers␣
↪support the MPS format
                   which is industry standard. Most support the LP format. Some␣
↪solvers have own formats,
                   for example, the CIP format of SCIP ("constraint integer␣
↪programming").
--readParam <file>
                   read backend-specific parameters from file (some backends)
--writeParam <file>
                   write backend-specific parameters to file (some backends)
--readConcurrentParam <file>
                   each of these commands specifies a parameter file of one␣
↪concurrent solve (Gurobi only)
--keep-paths       this standard flattening option annotates every item in FlatZinc␣
↪by its "flattening history".
                   For MIP solvers, it additionally assigns each constraint's name␣
```

```
↪as the first 255 symbols of that.
--cbcArgs '-guess -cuts off -preprocess off -passc 1'
                    parameters for the COIN-OR CBC backend
```

All MIP solvers directly support multi-threading (option `-p`). For COIN-BC to use it, it needs to be configured with `--enable-cbc-parallel`.

3.5.2.4 Subtour Elimination Constraints

Optionally use the SEC cuts for the circuit global constraint. Currently only Gurobi, IBM ILOG CPLEX, and COIN-OR CBC (trunk as of Nov 2019). If compiling from source, this needs boost and cmake flag `-DCOMPILE_BOOST_MINCUT=ON` (or `#define` it in `lib/algorithms/min_cut.cpp`). Compile your model with the flag `-DnSECcuts=<n>` with the following possible `<n>`: 0,1: use MTZ formulation; 1,2: pass on circuit constraints to the SEC cut generator, so 1 would use both.

3.5.2.5 Unified Domains (MIPdomains)

The 'MIPdomains' feature of the Flattener aims at reducing the number of binary flags encoding linearized domain constraints, see the paper *Belov, Stuckey, Tack, Wallace. Improved Linearization of Constraint Programming Models. CP 2016.*

By default it is on. To turn it off which might be good for some models, add option `-D fMIPdomains=false` during flattening. Some parameters of the unification are available, run with `--help`.

3.5.2.6 Indicator Constraints

Some solvers (IBM ILOG CPLEX, Gurobi, SCIP) have indicator constrains with greater numerical stability than big-M decomposition. Moreover, they can be applied to decompose logical constraints on *unbounded variables*. However, for reified comparisons with reasonable big-M bounds they perform worse because solvers don't include them in the LP relaxation. Add command-line parameters `-D fIndConstr=true -D fMIPdomains=false` when flattening to use them.

3.5.2.7 Quadratic Constraints and Objectives (MIQCP)

Gurobi 9.0 and SCIP support MIQCP (invoking non-convex global optimizer because MiniZinc translates multiplication to equality with an intermediate variable: whenever the model uses an expression x*y it is converted to z with z==x*y which is non-convex). While this is mostly advantageous for integer multiplication (which is linearly decomposed for other solvers), for float variables this is the only way to go. To switch off forwarding float/integer multiplications to the backend, run compiler with either or both of `-DQuadrFloat=false -DQuadrInt=false`.

3.5.2.8 Pools of User Cuts and Lazy Constraints

Some constraints in the model can be declared as user and/or lazy cuts and they will be added to the corresponding pools for the solvers supporting them. For that, apply annotations `::MIP_cut` and/or `::MIP_lazy` after a constraint. For Gurobi and IBM ILOG CPLEX, see `share/minizinc/linear/options.mzn` for their exact meaning.

3.5.2.9 Warm Starts

For general information of warm start annotations, see sec_warm_starts. Warm starts are currently implemented for Gurobi, IBM ILOG CPLEX, Xpress, and COIN-OR CBC.

3.5.3 Non-Linear Solvers via NL File Format

MiniZinc has experimental support for non-linear solvers that conform to the AMPL NL standard. There are a number of open-source solvers, such as Ipopt, Bonmin and Couenne, that can be interfaced to MiniZinc in this way.

You can download binaries of these solvers from AMPL (https://ampl.com/products/solvers/open-source/). In order to use them with MiniZinc, you need to create a solver configuration file. Future version of MiniZinc will make this easier, but for now you can follow these steps:

1. Download the solver binary. For this example, we assume you chose the Couenne solver, which supports non-linear, non-convex, mixed discrete and continuous problems.
2. Create a solver configuration file called `couenne.msc` in the `share/minizinc/solvers` directory of your MiniZinc installation, with the following contents:

   ```
   {
     "id" : "org.coin-or.couenne",
     "name" : "Couenne",
     "executable" : "/Users/tack/Downloads/couenne-osx/couenne",
     "version": "0.5.6",
     "supportsFzn":false,
     "supportsNL":true
   }
   ```

   You can adapt the `version` field if you downloaded a different version (it's only used for displaying).
3. Run `minizinc --solvers`. The Couenne solver should appear in the list of solvers now.
4. Run `minizinc --solver couenne model.mzn` on some MiniZinc model, or use Couenne from the MiniZinc IDE.

The AMPL NL support is currently experimental, and your MiniZinc model is translated to NL without regard for the capabilities of the target solver. For example, Ipopt only supports continuous variables, so translating a model with integer variables will result in a solver-level error message. There is currently no support for translating Boolean variables and constraints into 0/1 integer variables (as required by e.g. Couenne). You can experiment with the standard linearisation library, using the `-Glinear [-DQuadrFloat=true -DQuadrInt=true]` flag. However, this will either linearise all integer constraints, even the ones that solvers like Couenne may support natively, or use non-convex representation. We will ship dedicated solver libraries for some NL solvers with future versions of MiniZinc.

CHAPTER 3.6

Automatic Solution Checking, Model Validation, and Benchmarking

MiniZinc supports checking the correctness of solutions of a model in two ways. First, it is the *autonomous checking* where compiler substitutes solution values into the model and tries to find inconsistencies, possibly invoking a solver to a much smaller instance. Second, it is by using a *checker model*. This latter method can also be seen as model validation.

3.6.1 Autonomous Automatic Solution Checking and Benchmarking

For this method, compiler needs the values of as many variables as possible, ideally all variables without the right-hand side. For all top-level variables (not those inside let constructs) the values can be output using `--output-mode dzn` compiler option. For optimization problems, when no explicit objective variable exists, a variable named `_objective` is added by `--output-objective`. Then, each produced solution can be copied into a `.dzn` and compiled together with the original instance, adding `--allow-multiple-assignments` option if `--output-objective` was used. In the case that some variables inside let constructs need to be assigned, a solver has to be invoked for the smaller instance remaining after fixing the supplied values. For example, for the following model:

```
var int: a;
var float: b = a;
solve
  maximize a-b;
output
  [ "a+b is \(a+b)\n" ];
```

running it with `minizinc --solver gecode test_output_mode.mzn --output-mode dzn --output-objective` ignores the provided output annotation and prints

```
a = -2147483646;
_objective = 0.0;
```

```
----------
==========
```

which can be added as an extra `.dzn` file. The process of compilation for autonomous checking and re-solving with the output variables fixed, is automated by a Python script `tests/benchmarking/mzn-test.py`. To solve an instance with autonomous checking by variable value substitution, run, e.g.,

```
mzn-test.py --solver gecode model.mzn data.dzn
```

Moreover, `mzn-test.py` provides facility to run a list of instances and compare results from various test runs and different solvers.

3.6.2 Model Validation: Automatic Solution Checking with a Checker Model

This approach has two main applications:

- Instructors can provide checker models for student assignments. This provides students with immediate, detailed feedback on their modelling attempts.
- A simplified checker model can be used to verify a complex model used for solving. This can be useful when experimenting with new decompositions of constraints, or for post-processing solutions if some constraints cannot be added to the original model (e.g. in case the solver does not support certain constructs).

Running MiniZinc with solution checking is easy. On the command line, simply pass the name of the checker model in addition to the problem model:

```
minizinc model.mzn model.mzc.mzn data.dzn
```

Checker models can be pre-compiled in order to obfuscate their contents (e.g. if they contain clues to students how to model the problem):

```
minizinc --compile-solution-checker model.mzc.mzn
```

This will create the compiled checker model `model.mzc`, which can be used instead of the clear text version for checking:

```
minizinc model.mzn model.mzc data.dzn
```

The MiniZinc IDE has built-in support for solution checkers. Whenever the current project contains a file with the same file name as the current model but file extension `.mzc` or `.mzc.mzn`, the "Run" button turns into a "Run+Check" button.

The rest of this section describes how to implement checker models.

3.6.2.1 Basic checker models

At its core, a checker model takes each solution that a solver produces as input, and outputs whether the solution is correct or not. Let's use the simple map colouring model from the tutorial as an example. Here is the model again:

列表 6.1: 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"];
```

A checker model for this model requires the values of the variables `wa`, `nt`, and so on, for each solution, and then has to test whether all constraints hold. The output of the checker model should contain a line starting with `CORRECT` if the solution passes the test, or `INCORRECT` if it doesn't.

Since these values will be fixed in any solution, checker models simply declare parameters with the same name as the model variables:

列表 6.2: A MiniZinc checker model `aust.mzc.mzn` for the map colouring problem

```
int: wa;
int: nt;
int: sa;
int: q;
int: nsw;
int: v;
int: t;

output [
  if wa!=nt /\ wa!=sa /\ nt!=sa /\ nt!=q /\ sa!=q /\
     sa!=nsw /\ sa!=v /\ q!=nsw /\ nsw!=v
  then "CORRECT\n"
  else "INCORRECT\n"
  endif
];
```

Running the model and the checker will produce output like this:

```
wa=3   nt=2    sa=1
q=3    nsw=2   v=3
t=3
% Solution checker report:
% CORRECT
----------
```

The solution checker report is embedded as comments in the original output.

3.6.2.2 Detailed feedback

The basic checker model above only reports whether the solutions satisfy the constraints, but it doesn't provide any insights into the nature of the error if a constraint is violated.

We can use standard MiniZinc functionality to provide much more detailed feedback. The following checker model introduces a helper function `check` which outputs a detailed error message if a constraint doesn't hold. The results of all the checks are combined, and if any of the constraints was violated, the output is `INCORRECT`, otherwise it is `CORRECT`.

列表 6.3: A checker model `aust-2.mzc.mzn` for the map colouring problem with more detailed error messages

```
int: wa;
int: nt;
int: sa;
int: q;
int: nsw;
int: v;
int: t;

test check(bool: b,string: s) =
  if b then true else trace_stdout("ERROR: "++s++"\n",false) endif;

output [
  if check(wa!=nt, "wa and nt have the same colour")
  /\ check(wa!=sa, "wa and sa have the same colour")
  /\ check(nt!=sa, "nt and sa have the same colour")
  /\ check(nt!=q, "nt and q have the same colour")
  /\ check(sa!=q, "sa and q have the same colour")
  /\ check(sa!=nsw, "sa and nsw have the same colour")
  /\ check(sa!=v, "sa and v have the same colour")
  /\ check(q!=nsw, "q and nsw have the same colour")
  /\ check(nsw!=v, "nsw and v have the same colour")
  then "CORRECT: all constraints hold.\n"
  else "INCORRECT\n"
  endif
];
```

However, the checker model will only report the first violated constraint, since the conjunction operator short-circuits the evaluation when one of its arguments is false. For example, if we remove all constraints from the original model, the output would be:

```
wa=3   nt=3    sa=3
q=3    nsw=3   v=3
t=3
% Solution checker report:
% ERROR: wa and nt have the same colour
% INCORRECT
----------
```

In order to get all error messages, we can force the evaluation of all checkers by creating an auxiliary array of check results:

列表 6.4: A checker model `aust-3.mzc.mzn` for the map colouring problem without short-circuit evaluation

```
int: wa;
int: nt;
int: sa;
int: q;
int: nsw;
int: v;
int: t;

test check(bool: b,string: s) =
  if b then true else trace_stdout("ERROR: "++s++"\n",false) endif;

array[int] of bool: checks = [
  check(wa!=nt, "wa and nt have the same colour"),
  check(wa!=sa, "wa and sa have the same colour"),
  check(nt!=sa, "nt and sa have the same colour"),
  check(nt!=q, "nt and q have the same colour"),
  check(sa!=q, "sa and q have the same colour"),
  check(sa!=nsw, "sa and nsw have the same colour"),
  check(sa!=v, "sa and v have the same colour"),
  check(q!=nsw, "q and nsw have the same colour"),
  check(nsw!=v, "nsw and v have the same colour")
];

output [
  if forall(checks)
  then "CORRECT: all constraints hold.\n"
  else "INCORRECT\n"
  endif
];
```

Now the output contains all error messages (for the case where the model has no constraints):

```
wa=3   nt=3    sa=3
q=3    nsw=3   v=3
t=3
% Solution checker report:
% ERROR: nsw and v have the same colour
% ERROR: q and nsw have the same colour
% ERROR: sa and v have the same colour
% ERROR: sa and nsw have the same colour
```

```
% ERROR: sa and q have the same colour
% ERROR: nt and q have the same colour
% ERROR: nt and sa have the same colour
% ERROR: wa and sa have the same colour
% ERROR: wa and nt have the same colour
% INCORRECT
----------
```

3.6.2.3 Instance data in checker models

The map colouring example was quite simple because the model did not contain any parameter declarations. For a model that is parameterised, the checker model simply contains the same parameter declarations. MiniZinc will then pass the actual parameters of the instance being solved to the checker model.

For example, the following checker model could be used for the n-Queens problem.

列表 6.5: A checker model `nqueens.mzc.mzn` for the n-Queens problem

```
int: n; % number of queens
set of int: ROW = 1..n;
set of int: COL = 1..n;
array[int] of int: q; % col of queen in each row

test check(bool: b,string: s) =
  if b then true else trace_stdout("ERROR: "++s++"\n",false) endif;

output [
  if check(index_set(q)=1..n, "ERROR: array q should have index set 1..\(n)")
  /\ forall(i in 1..n)(check(q[i] in 1..n, "ERROR: q[\(i)] should have a value in 1.
↪.\(n)"))
  /\ forall(r1, r2 in 1..n where r1 < r2)
           (check(q[r1] != q[r2],
            "queens in rows \(r1) and \(r2) are on same column\n")
            /\
            check(q[r1]+r1 != q[r2]+r2,
            "queens in rows \(r1) and \(r2) are on same up diagonal\n")
            /\
            check(q[r1]-r1 != q[r2]-r2,
            "queens in rows \(r1) and \(r2) are on same down diagonal\n")
           )
  then "CORRECT: All constraints hold"
  else "INCORRECT" endif ];
```

The checker model first makes sure that the solution has the right dimensions (correct array index set, and each variable is assigned a value in the correct domain), and then uses standard MiniZinc constructs to check each constraint.

3.6.2.4 Checking optimisation problems

Optimisation problems pose a difficulty for automatic checking. When a solver claims to prove optimality, we cannot easily verify this claim without solving the problem again (using a different model that is known to be correct). At the moment, solution checking

for optimisation problems is restricted to checking that the objective has been computed correctly.

To check that the objective value is calculated correctly, define a parameter called `_objective` of the appropriate type (`int` or `float`) in the checker. The example in the next section illustrates this.

3.6.2.5 Hidden variables

For many models, the decision variables which describe the solution may not be the variables that are natural to describe the constraints. There may be other internal variables which are functionally defined by the decision variables, and which are likely to be used in the model for building constraints and/or are much more natural for describing the correctness of a candidate solution.

Consider the problem of lining up n people numbered from 1 to n for a photo in a single line. We want to make sure that there are no more than two people of the same gender (male, female or other) in sequence in the line. We want to minimize the total distance between the pairs of people who are numbered consecutively. The decisions are an array `pos` which for each person gives their position in the line. A correct model for this is given in 列表 6.6.

列表 6.6: A model `photo.mzn` for the photo lineup problem

```
int: n;                          % number of people
set of int: PERSON = 1..n;       % set of people
enum GENDER = { M, F, O };       % set of genders
array[PERSON] of GENDER: g;      % the gender of each person
set of int: POSN = 1..n;         % set of positions

array[PERSON] of var POSN: pos;  % decisions: a position for each person

array[POSN] of var PERSON: who;  % view: a person for each position
include "inverse.mzn";
constraint inverse(pos,who);     % channel from decisions to view
constraint forall(i in 1..n-2)
                 (g[who[i]] != g[who[i+1]] \/ g[who[i+1]] != g[who[i+2]]);

solve minimize sum(i in 1..n-1)(abs(pos[i] - pos[i+1]));
```

A critical part of learning how to model this problem is to realise that it is worth introducing the inverse of the `pos` variables, called `who` in this model. The `who` array makes it much easier to specify that no more than two people of the same gender can be adjacent. The checker model **should not** include the `who` variables, because that would effectively give away the key to the solution.

The use of hidden variables (such as `who`) makes checking harder in two ways. First, we may need to define these in the checker model as decision variables (rather than parameters), and MiniZinc will then need to invoke a solver to generate the values of these variables in order to check a given solution. Second, given an incorrect candidate solution, there may be no possible value for these hidden variables, hence we have to guard the checking to ensure we do not assume that there is some value for the hidden variables.

Consider this data file for the photo problem:

列表 6.7: Example data for the photo.mzn photo lineup problem (photo.dzn)

```
n = 9;
g = [M,M,M,M,F,F,F,M,M];
```

```
n = 9;
g = [M,M,M,M,F,F,F,M,M];
```

and assume that an incorrect model returns the erroneous candidate solution `pos = [1,2,3,4,5,6,7,2,9]; _objective=25;`. If we simply assert `inverse(pos,who)` in the checker model, then this constraint will fail since there is no inverse of this position array, because persons 2 and 8 are both at position 2. Hence the checker must guard the inverse constraint, and only when that succeeds use the computed values of the `who` variables for checking the solution. A checking model for the photo problem might look like this:

列表 6.8: A checker model photo.mzc.mzn for the photo lineup problem

```
int: n;
set of int: PERSON = 1..n;
enum GENDER = { M, F, O };
array[PERSON] of GENDER: g;
set of int: POSN = 1..n;
array[int] of int: pos;

array[POSN] of var PERSON: who;
int: _objective;

test alldifferent(array[int] of int: x) =
                 forall(i,j in index_set(x) where i < j)(x[i] != x[j]);

include "inverse.mzn";
constraint if alldifferent(pos) then inverse(pos,who)
           else forall(i in 1..n)(who[i] = 1) endif;

output [if
  check_array(pos, n, POSN, "pos") /\
  check_alldifferent(pos,"pos") /\
  check_array(fix(pos), n, PERSON, "who") /\
  forall(i in 1..n-2)
        (check(g[fix(who[i])] != g[fix(who[i+1])] \/
               g[fix(who[i+1])] != g[fix(who[i+2])],
               "three people of the same gender \(g[fix(who[i])]) in positions \(i).
↪.\(i+2)\n")) /\
  let { int: obj = sum(i in 1..n-1)(abs(pos[i] - pos[i+1])); } in
    check(obj = _objective, "calculated objective \(obj) does not agree with␣
↪objective from the model (\(_objective))\n")
  then
    "CORRECT: All constraints hold"
  else
    "INCORRECT"
  endif];
```

```
test check(bool: b,string: s) =
  if b then true else trace_stdout("ERROR: "++s++"\n",false) endif;

test check_alldifferent(array[int] of int: x, string: name) =
  forall(i, j in index_set(x) where i < j)
        (check(x[i] != x[j], name ++ "[\(i)] = \(x[i]) = " ++
         name ++ "[\(j)] " ++
         "when they should be different\n"));

test check_int(int: x, set of int: vals, string: name) =
  check(x in vals, "integer \(name) is not in values \(vals)\n");

function bool: check_array(array[int] of int: x, int: length, set of int: vals,␣
↪string: name) =
  check(length(x) = length, "array \(name) is not of length \(length)\n") /\
  forall(i in index_set(x))
        (check(x[i] in vals, "element \(i) of array \(name), \(x[i]) is not in␣
↪values \(vals)\n"));
```

The checker model first tests whether the given `pos` array satisfies the `alldifferent` property (using a custom `test` for `alldifferent` on a par array). If it passes the test, the `inverse` constraint is applied. Otherwise, the `who` array is simply fixed to a list of ones.

The check for the `alldifferent` constraint is a good example for a checker that tries to give a detailed description of the error.

CHAPTER 3.7

Globalizer

Globalizer[1] analyses a model and set of data to provide suggestions of global constraints that can replace or enhance a user's model.

3.7.1 Basic Usage

To use Globalizer simply execute it on a model and set of data files:

```
minizinc --solver 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 problem[2], `cars.mzn`.

列表 7.1: 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;

% The class of car being started at each step.
array [steps] of var classes: class;
```

[1] Leo, K. et al., "Globalizing Constraint Models", 2013.
[2] http://www.csplib.org/Problems/prob001/

```
% 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`.

列表 7.2: 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]));
```

3.7.2 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));
```

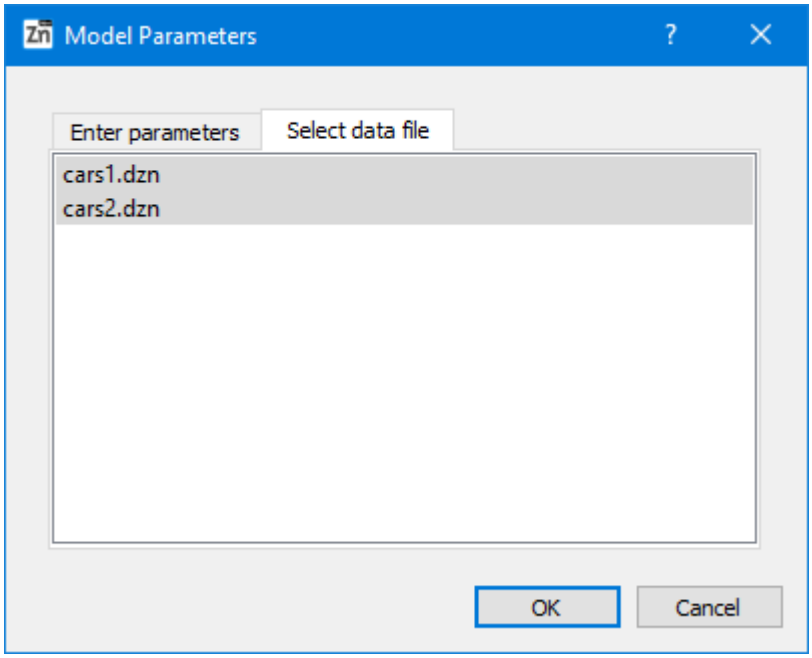
3.7.3 Supported Constraints

Globalizer currently supports detection of the following constraints:

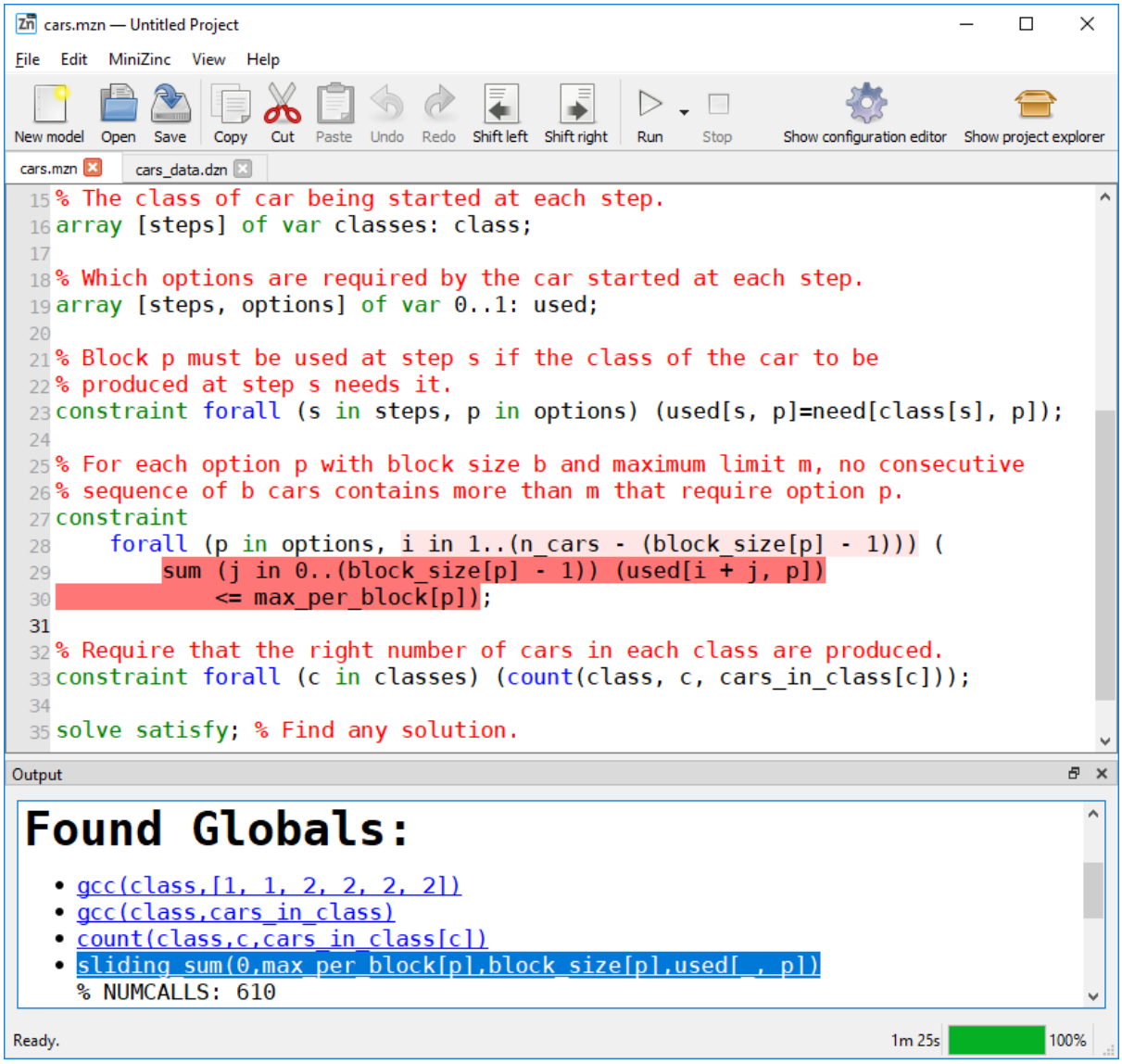
```
alldifferent                alldifferent_except_0       all_equal_int
bin_packing                 bin_packing_capa            bin_packing_load
binaries_represent_int      binaries_represent_int_3A   binaries_represent_int_3B
binaries_represent_int_3C   channel                     channelACB
count_geq                   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
```

3.7.4 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.



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.

3.7.5 How it works

A summary of the algorithm is presented here. For a more detailed exploration of the approach see the Globalizing Constraint Models paper[Page 195, 1].

- **Normalize.** In this step the model is transformed to make analysis easier.
  - Conjunctions of constraints are broken into individual constraints. For example: `C1 /\ C2;` becomes `C1; C2;`
  - Conjunctions under `forall` constraints are broken into individual `forall` constraints. For example: `forall(...) (C1 /\ 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.

3.7.6 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 sub-

problems 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 time-limit. 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.

3.7.7 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 3.8

FindMUS

Version: 0.6.0

FindMUS[1] 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.

3.8.1 Basic Usage

To use FindMUS on the command line simply execute it on a model and set of data files by typing:

```
findMUS model.mzn data-1.dzn
```

This will perform a search for a single instance level MUS. To acquire a low level MUS the `--paramset fzn` argument can be used. To search for more than one MUS use the `-a` or `-n <count>` arguments. By default FindMUS will use `gecode_presolver` for checking the satisfiability of a subset of constraints. This can be changed using the `--subsolver` argument. For example, given a model that is better suited to a Mixed Integer Programming solver, the argument `--solver cbc` could be used to instruct FindMUS to flatten the model with the linear library and use the CBC solver for satisfiability checking.

Note: FindMUS requires a fully instantiated constraint model.

3.8.1.1 Commandline arguments

The FindMUS tool supports the following arguments:

Driver options

The driver creates the map solver and sub-solver and requests MUSes from FindMUS's enumeration algorithm HierMUS.

[1] Leo, K. et al., "Debugging Unsatisfiable Constraint Models", 2017.

`-n <count>` Stop after the *n* th MUS is found (Default: 1)

`-a` Find all MUSes (equivalent to `-n 0`)

`-t` or `--timeout <s>` Stop search after *s* seconds (Default: 1800)

`--paramset hint,mzn,fzn`
: Use a preset collection of parameters. (Default: mzn) The `mzn` and `fzn` options can acquire MUSes in more detail.

  `hint` --structure gen --depth mzn --shrink-alg map_lin

  `mzn` --structure normal --depth mzn --shrink-alg map_qx

  `fzn` --structure normal --depth fzn --shrink-alg map_qx

`--no-leftover`
: Do not print leftover candidate on timeout If FindMUS is exploring an unsatisfiable subset when it runs out of time, it behaves as if the set could not be reduced to a smaller set and prints it as an MUS. The `--no-leftover` setting disables this behaviour as these sets can sometimes be quite large.

`--frequent-stats` Print statistics after every MUS discovered.

`--no-progress` Disable `%%%mzn-progress` output

`--no-stats` Disable output of initial parsing stats and summary stats

`--output-{html, json, brief, human}` Changes output mode

  `html` HTML output mode for use with the MiniZinc IDE

  `json` Simple json output (each MUS should be parsed separately)

  `brief` Same as the default output mode but without outputting the traces (paths)

  `human` Tries to format the MUS in a more readable form grouping constraints by their constraint names

Compiler Options

`-g` or `--domains` Record domain changes during compilation (same as `minizinc -c -g`)

`--verbose-compile` Pass `--verbose` to MiniZinc compiler

Enumeration options

The enumeration algorithm (HierMUS) explores the constraint hierarchy provided in the user's model and proposes potential MUSes to the sub-solver. A `shrink` algorithm is used to reduce found UNSAT subsets to MUSes. There are currently four available shrink algorithms available with FindMUS.

`--shrink-alg lin,map_lin,qx,map_qx,qx2`
: Select shrink algorithm (Default: map_lin)

  `lin` simple linear shrink algorithm.

  `map_lin` alternative implementation of the linear algorithm that updates a global map of explored subsets while shrinking the subset.

`qx` uses the `QuickXplain`[4] algorithm to shrink the subset to a MUS.

`map_qx` is an alternative implementation of `QuickXplain` that uses the global map to split subsets, and to avoid repeating any SAT checks while shrinking

`qx2` a lightweight alternative to `map_qx` that only uses the map to avoid repeating SAT checks.

`--depth mzn,fzn,<n>`
How deep in the tree should search explore. (Default: mzn)

`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.

`--restarts` Enable restart to flat mode (plain Marco) if large run of SAT sets encountered

`--seed <n>` Set the random seed used for splitting with `map_qx`

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.

`--subsolver <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)

`--adapt-timelimit`
Automatically tighten solver time-limit based on performance during initial sanity check (prove background is SAT, background + entire foreground is UNSAT)

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.

`--soft-defines`
Consider functional constraints as part of MUSes

`--hard-domains`
Consider domain constraints (added by `-g` option) as background

`--named-only`
Only consider constraints annotated with string annotations

`--filter-mode fg,ex` Filtering mode

[4] Junker, U. et al., "QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems", 2004.

`fg` Foreground: Excluded items go to the background

`ex` Exclusive: Excluded items are omitted entirely (Useful if they are part of MUSes you are not interested in)

`--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. The hierarchy is constructed in two steps. The first takes the existing model structure and can modify it in several ways. The second step adds extra binary structure.

`--structure flat,gen,normal,mix`

Alters initial structure: (Default: `gen`)

`flat` Remove all structure

`gen` Remove instance specific structure (merges constraints)

`normal` No change

`mix` Apply `gen` before `normal`

`idx` Remove all location information

`idxmix` Apply `idx` before `normal`

`--no-binarize`

By default FindMUS will introduce extra binary structure to the hierarchy. This option disables this behaviour (This is likely to be the default in future since QX performs a type of binarization).

Verbosity options

`--verbose-{map,enum,subsolve} <n>` Set verbosity level for different components. Currently `n` can only take values 0, 1, and 2;

`--verbose` Set verbosity level of all components to 1

Misc options

`--dump-dot <dot>` Write tree in GraphViz format to file <dot>

3.8.1.2 Example

The following demonstrates the basic usage of FindMUS on a simple example. Below, we see a model for the latin squares puzzle[2] with some incompatible symmetry breaking constraints added.

[2] https://en.wikipedia.org/wiki/Latin_square

列表 8.1: 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(X[i,..])           :: "AD(row \(i))"
        );
constraint :: "ADCols"
    forall (j in N)
        (alldifferent(X[..,j])           :: "AD(col \(j))"
        );

constraint :: "LLRows"
    forall (i in 1..n-1)
        (lex_less(X[i,..], X[i+1,..])    :: "LL(rows \(i) \(i+1))"
        );
constraint :: "LGCols"
    forall (j in 1..n-1)
        (lex_greater(X[..,j], 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 `findMUS -n 2 latin_squares.mzn` returns the following output. Note that the `-n 2` argument requests that just the first two MUSes are found.

```
FznSubProblem:  hard cons: 0    soft cons: 10   leaves: 10      branches: 11    ␣
↪Built tree in 0.02854 seconds.
MUS: 7 3 6 4 8
Brief: gecode_all_different_int;@{ADCols@AD(col 1)}:(j=1)
gecode_all_different_int;@{ADCols@AD(col 2)}:(j=2)
gecode_array_int_lt;@{LGCols@LG(cols 1 2)}:(j=1)
gecode_array_int_lt;@{LLRows@LL(rows 1 2)}:(i=1)
gecode_array_int_lt;@{LLRows@LL(rows 2 3)}:(i=2)

Traces:
latin_squares.mzn|18|5|20|9|ca|forall;latin_squares.mzn|18|5|20|9|ac;latin_squares.
↪mzn|18|13|18|13|i=2;latin_squares.mzn|19|10|19|37|ca|lex_less
```

```
latin_squares.mzn|13|5|15|9|ca|forall;latin_squares.mzn|13|5|15|9|ac;latin_squares.
↪mzn|13|13|13|13|j=1;latin_squares.mzn|14|10|14|30|ca|alldifferent
latin_squares.mzn|18|5|20|9|ca|forall;latin_squares.mzn|18|5|20|9|ac;latin_squares.
↪mzn|18|13|18|13|i=1;latin_squares.mzn|19|10|19|37|ca|lex_less
latin_squares.mzn|13|5|15|9|ca|forall;latin_squares.mzn|13|5|15|9|ac;latin_squares.
↪mzn|13|13|13|13|j=2;latin_squares.mzn|14|10|14|30|ca|alldifferent
latin_squares.mzn|22|5|24|9|ca|forall;latin_squares.mzn|22|5|24|9|ac;latin_squares.
↪mzn|22|13|22|13|j=1;latin_squares.mzn|23|10|23|40|ca|lex_greater

MUS: 7 3 5 6 8 9
Brief: gecode_all_different_int;@{ADCols@AD(col 1)}:(j=1)
gecode_all_different_int;@{ADCols@AD(col 3)}:(j=3)
gecode_array_int_lt;@{LGCols@LG(cols 1 2)}:(j=1)
gecode_array_int_lt;@{LGCols@LG(cols 2 3)}:(j=2)
gecode_array_int_lt;@{LLRows@LL(rows 1 2)}:(i=1)
gecode_array_int_lt;@{LLRows@LL(rows 2 3)}:(i=2)

Traces:
latin_squares.mzn|18|5|20|9|ca|forall;latin_squares.mzn|18|5|20|9|ac;latin_squares.
↪mzn|18|13|18|13|i=2;latin_squares.mzn|19|10|19|37|ca|lex_less
latin_squares.mzn|13|5|15|9|ca|forall;latin_squares.mzn|13|5|15|9|ac;latin_squares.
↪mzn|13|13|13|13|j=1;latin_squares.mzn|14|10|14|30|ca|alldifferent
latin_squares.mzn|13|5|15|9|ca|forall;latin_squares.mzn|13|5|15|9|ac;latin_squares.
↪mzn|13|13|13|13|j=3;latin_squares.mzn|14|10|14|30|ca|alldifferent
latin_squares.mzn|18|5|20|9|ca|forall;latin_squares.mzn|18|5|20|9|ac;latin_squares.
↪mzn|18|13|18|13|i=1;latin_squares.mzn|19|10|19|37|ca|lex_less
latin_squares.mzn|22|5|24|9|ca|forall;latin_squares.mzn|22|5|24|9|ac;latin_squares.
↪mzn|22|13|22|13|j=1;latin_squares.mzn|23|10|23|40|ca|lex_greater
latin_squares.mzn|22|5|24|9|ca|forall;latin_squares.mzn|22|5|24|9|ac;latin_squares.
↪mzn|22|13|22|13|j=2;latin_squares.mzn|23|10|23|40|ca|lex_greater
Total Time: 0.39174     nmuses: 2       map:     40     sat:     21     total:    ␣
↪61
```

The first line, starting with `FznSubProblem:` provides some useful information for debugging the `findMUS` tool. Next we have the list of MUSes. Each MUS is described with three sections:

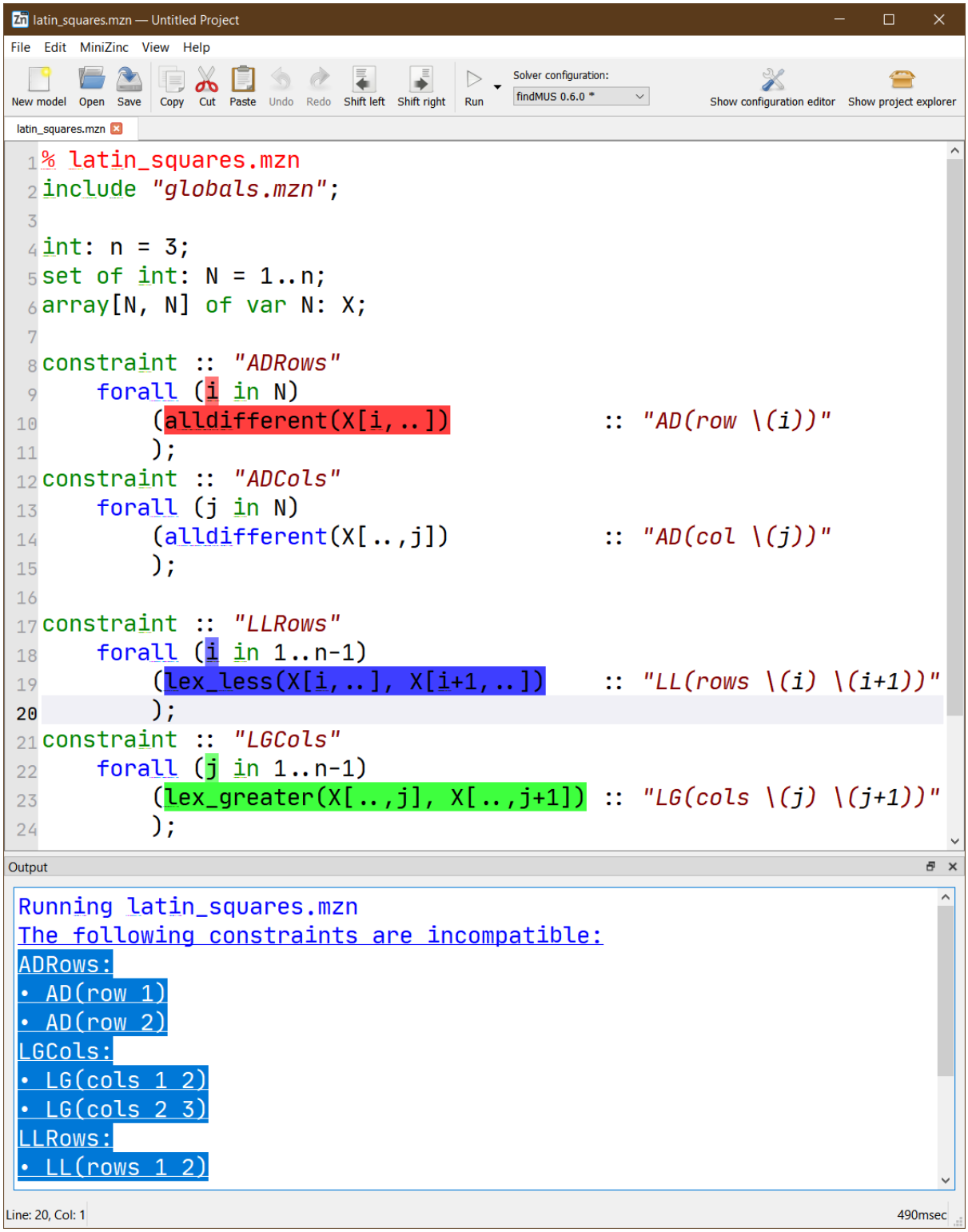
1. `MUS:` lists the indices of the FlatZinc constraints involved in this MUS.
2. `Brief:` lists the FlatZinc constraint name, the expression name, the constraint name, and the loop index assignments for each involved FlatZinc constraint.
3. `Traces:` lists the MiniZinc paths corresponding to the constraints 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 these two MUSes we see that the `lex` constraints are incompatible with the `alldifferent` constraints.

3.8.2 Using FindMUS in the MiniZinc IDE

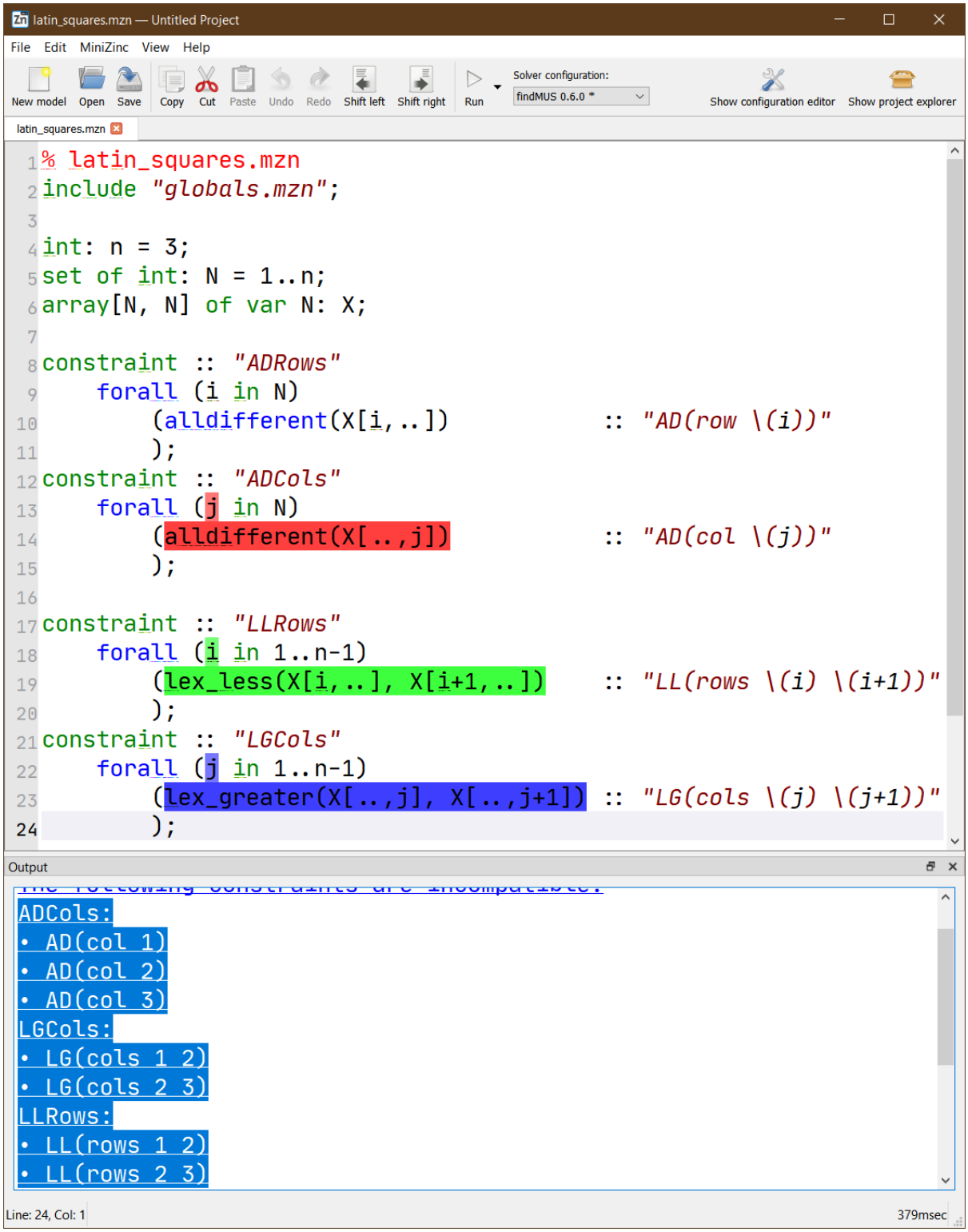
To use FindMUS in the MiniZinc IDE, upon discovering that a model is unsatisfiable. Select `FindMUS` from the solver configuration dropdown menu and click the solve button (play symbol). This will look for MUSes using the default `--paramset mzn` argument.



In this case we can see that the output pane shows the constraints involved in the MUS in terms of the expression annotations, grouped by the constraint annotations. Clicking

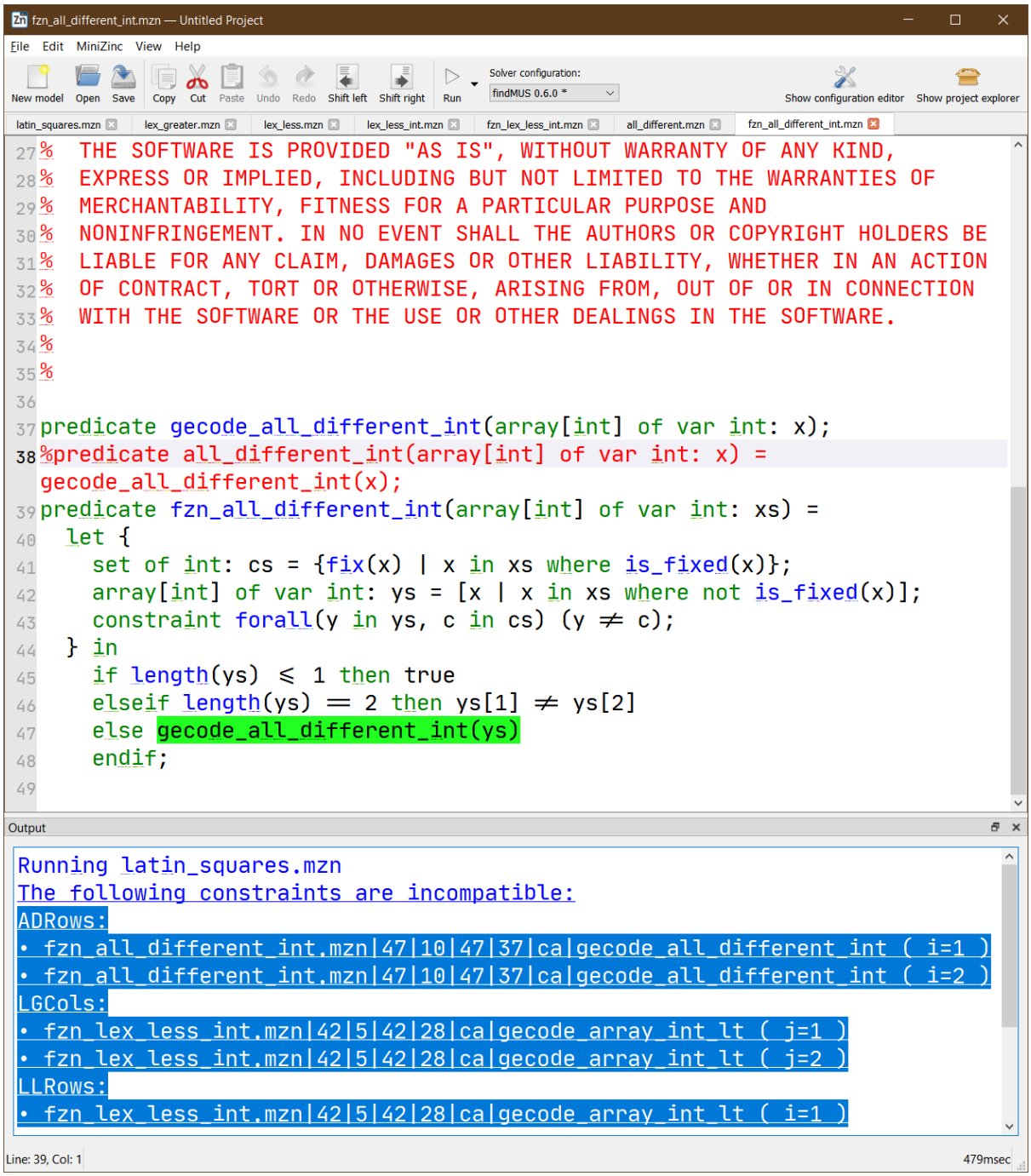
on the MUS in the output pane will highlight the constraints in your model.

If finding MUSes with `--paramset mzn` is prohibitively slow, you can try running instead with `--paramset hint`. This merges many of the low level constraints into larger units and should be relatively fast and help locate the relevant constraint items. The following shows the result:



Selecting the returned MUS highlights three top level constraints as shown: `ADRows`, `LLRows` and `LGCols`.

Alternatively, in some cases you may wish to look at the MUSes of your model in terms of the flattened FlatZinc program. To display these MUSes run FindMUS with the `--paramset fzn` argument.



In this figure you can see that the MUSes are expressed in terms of FlatZinc constraints along with any assignments to relevant loop variables. Clicking on this MUS also opens and highlights the source of the constraints in the MiniZinc library (in this case in Gecode's custom redefinition of `fzn_all_different_int`.

3.8.3 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 paperPage 203, 1.

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 using the MARCO[3] MUS enumeration algorithm. 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 unsatisfiable subsets 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.

3.8.4 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.

When parameter tuning for FindMUS the oracle checker mode can speed up testing of different sets of parameters.

1. Run `findMUS -a --paramset fzn --output-brief ... > muses.log` to save the output log to a file.
2. Run subsequent sets of parameters with `findMUS -a --paramset fzn --output-brief --oracle-only muses.log ...`. The oracle mode will perform enumeration without making calls to the sub-solver, instead deciding SAT/UNSAT based on the MUS log (omit `--oracle-only` to call the sub-solver when the status of a subset is unknown)

3.8.5 Limitations / Future work

There are several features that we aim to include quite soon:

Solver core/MUS/IIS integration
: Set shrinking can be accelerated by using problem specific MUS extraction methods that some solvers provide.

Context-find
: Once a MUS has been found flip the foreground + background and narrow down the parts of the background that complete the MUS

[3] Liffiton, M. H. et al., "Fast, Flexible MUS Enumeration", 2016.

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.

CHAPTER 3.9

Mzn-Analyse

`mzn-analyse` is an informal collection of experimental MiniZinc compiler "passes" that analyse and manipulate MiniZinc models. The passes pipeline structure allows relatively complex tasks to be performed on MiniZinc models by combining multiple passes together.

The tool provides the `get-diversity-anns` pass used by the `diverse_solutions()` solution diversity features of the MiniZinc python interface.

3.9.1 Basic Usage

```
mzn-analyse input.mzn [passes]
```

where `[passes]` is the list of passes and arguments for those passes (`cmd:arg1,arg2,...` with no spaces unless quoted)

An implicit `out:-` command (or `out_fzn:-` if reading FlatZinc) is appended at the end of the pipeline if no `out` commands occur in the sequence. An explicit final `out` must be added to the end if you wish to output models throughout the pipeline.

Typically, each pass takes a MiniZinc model, and outputs a MiniZinc model to the next pass. Optionally, a pass may store some information in a global json store which will be dumped to stdout by default. The contents of the json output can be inspected at any point by using the `json_out` pass. The json store can be cleared by using the `json_clear` pass.

3.9.1.1 Passes: read/write models & json

These passes control the reading and writing of MiniZinc models and the json store. In most cases, if no argument is passed to one of these passes, they will read/write to stdout.

`in:in.mzn` Read input file (no support for stdout)

`out:out.mzn` Write model to out.mzn (- for stdout)

`out_fzn:out.fzn` Write model to out.fzn (- for stdout)

`no_out` Disable automatic output insertion

`json_out:out.json` Write collected json output to out.json (- for stdout)

`json_clear` Clear collected json output

`no_json` Disable automatic json output

3.9.1.2 Passes: Manipulate Models

These passes are for making modifications to models or isolating/removing parts of models.

`inline-includes` Inline non-library includes (this allows a model to be relocated without needing to move models it includes)

`inline-all-includes` Inline all includes

`remove-anns:name1,[name2,...]` Remove Id and Call annotations matching names

`remove-includes:name1,[name2,...]` Remove includes matching names

`output-all` Add 'add_to_output' annotation to all VarDecls

`remove-stdlib` Remove stdlib includes

`get-items:idx1,[idx2,...]` Narrow to items indexed by idx1,...

`filter-items:iid1,[iid2,...]` Only keep items matching iids

`remove-items:iid1,[iid2,...]` Remove items matching iids

`filter-typeinst:{var|par}` Just show var/par parts of model

`replace-with-newvar:location1,location2` Replace expressions with 'let' expressions

`repeat-model:k` Produce `k` copies of the model (with unique variable names)

`get-solve-anns` Get information about the solving annotations used (useful if rewriting the original objective function)

3.9.1.3 Passes: Special Passes

These passes are needed to support other tools that are developed for MiniZinc. They are more specific to the projects where they are used.

`get-diversity-anns` Extract diversity information from the model

`annotate-data-deps` Annotate expressions with their data dependencies

`get-term-types:out.terms` Write .terms file with types of objective terms

`get-data-deps:out.cons` (FlatZinc only) Write .cons file with data dependencies of FlatZinc constraints

`get-exprs:location1,location2` Extract list of expressions occurring inside location location = path.mzn|sl|sc|el|ec

`get-ast:location1,location2` Build JSON representation of AST for whole model or just for expression matching location1 or location2, place in json_-store

3.9.2 Examples

The following are a few examples of the usage of the `mzn-analyse` tool.

1. Remove the solve and output items from the model and write the model to solveless.mzn. It then inlines the local includes and outputs to stdout as "FlatZinc" (no line breaks while printing an item).

```
mzn-analyse in.mzn remove-items:solve,output out:solveless.mzn
inline-includes out_fzn
```

2. Remove all items except constraint items, picks out the 50th constraint, remove any annotations, then output to stdout. The implicit output will default to *out_fzn* since the input was fzn.

```
mzn-analyse in.fzn filter-items:constraint get-items:50 remove-anns
```

3. The following requests the data-deps information for constraints 60, 61, and 62 from a FlatZinc file. The *no_out* command disables the automatic insertion of *out_fzn*.

```
mzn-analyse rcpsp-wet-r0.annotated.fzn filter-items:constraint
get-items:60,61,62 get-data-deps no_out
```

Gives the following output:


```
{"constraint_info": [
  [
    ["in", "i", "Tasks"],
    ["in", "j", "suc[i]"],
    ["assign", "j", "24"],
    ["assign", "i", "2"],
    ["eq", "suc[i]", "23..24"],
    ["eq", "Tasks", "1..32"]],
  [
    ["in", "i", "Tasks"],
    ["in", "j", "suc[i]"],
    ["assign", "j", "5"],
    ["assign", "i", "3"],
    ["eq", "suc[i]", "{5,6,17}"],
    ["eq", "Tasks", "1..32"]],
  [
    ["in", "i", "Tasks"],
    ["in", "j", "suc[i]"],
    ["assign", "j", "6"],
    ["assign", "i", "3"],
    ["eq", "suc[i]", "{5,6,17}"],
    ["eq", "Tasks", "1..32"]]]}
```

3.9.3 Limitations / Future work

`mzn-analyse` is an experimental tool. The passes were implemented for very specific use cases and may not behave as expected in more general cases. The code in the repo is likely to make strong assumptions about the underlying models that will not always hold. The passes should be thought of more as proof-of-concept/AST manipulation sample code to be used as a starting point for implementing transformations.

CHAPTER 3.10

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.

3.10.1 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.

3.10.2 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?
```

CHAPTER 3.11

Python Interface

MiniZinc can be directly used from Python using the MiniZinc Python package. This package provides an native python interface that interacts directly with the MiniZinc bundle installed on the machine. For installation instructions and more information about the package that provides the MiniZinc Python interface please visit https://python.minizinc.dev/.

警告:

The development of MiniZinc Python is still in the very early stages. We appreciate any feedback for the package, but we would recommend refraining from using the package in production software. The package can drastically change and backwards compatible changes are not guaranteed.

CHAPTER 3.12

JavaScript Interface

MiniZinc can be used from JavaScript either in the browser using the WebAssembly build of MiniZinc, or in NodeJS using the MiniZinc bundle installed on the machine. For installation instructions and more information about the package providing the JavaScript interface, please visit https://js.minizinc.dev/.

CHAPTER 3.13

Installation from Source Code

The easiest way to install a complete version of MiniZinc is to use the binary packages we provide for several platforms, as explained in *Installation* (page 7). In the following, we describe how to install MiniZinc 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. The `master` branch of each repository points to the latest release version, while the `develop` branch points to the current state of development (and may be buggy or not even compile sometimes). Installation details can be found in each project's `README.md` files.

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.

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 232) introduces the syntax notation used throughout the specification. *Overview of a Model* (page 233) provides a high-level overview of MiniZinc models. *Syntax Overview* (page 235) covers syntax basics. *High-level Model Structure* (page 236) covers high-level structure: items, multi-file models, namespaces, and scopes. *Types and Type-insts* (page 238) introduces types and type-insts. *Expressions* (page 248) covers expressions. *Items* (page 265) describes the top-level items in detail. *Annotations* (page 275) describes annotations. *Partiality* (page 276)

describes how partiality is handled in various cases. *Built-in Operations* (page 278) describes the language built-ins. *Full grammar* (page 288) gives the MiniZinc grammar. *Content-types* (page 287) 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 288). 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 237) 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 270) 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 287). 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 270). 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 278) 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>
          | <type-inst-syn-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 265)).

Variable declaration items introduce new global variables and possibly bind them to a value (*Variable Declaration Items* (page 266)).

Assignment items bind values to global variables (*Assignment Items* (page 269)).

Constraint items describe model constraints (*Constraint Items* (page 270)).

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 270)).

Output items are used for nicely presenting the result of a model execution (*Output Items* (page 270)).

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 271)). 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 265)). 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 269)).

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 237)), 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 255)).
- Let expressions (*Let Expressions* (page 263)).

- Function and predicate argument lists and bodies (*User-defined Operations* (page 271)).

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 248)), 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; >, <, ==/=, !=, <= and >= 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 `par int` $\xrightarrow{c}$ `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 `par int` $\xrightarrow{v}$ `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 266)) and user-defined operation items (*User-defined Operations* (page 271)).

Type-inst expressions have this syntax:

```
<ti-expr> ::= <base-ti-expr>
            | <array-ti-expr>

<base-ti-expr> ::= <var-par> <opt-ti> <set-ti> <base-ti-expr-tail> ["++" <base-ti-
↪expr>]
                 | "any" <ti-variable-expr-tail>

<var-par> ::= "var" | "par" |
```

```
<opt-ti> ::= "opt" |

<set-ti> ::= "set" "of" |

<base-type> ::= "bool"
              | "int"
              | "float"
              | "string"

<base-ti-expr-tail> ::= <ident>
                      | <base-type>
                      | <ti-variable-expr-tail>
                      | <tuple-ti-expr-tail>
                      | <record-ti-expr-tail>
                      | "ann"
                      | <expr>
```

(The final alternative, for range types, uses the numeric-specific `<num-expr>` non-terminal, defined in *Expressions Overview* (page 248), 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 242).

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`, 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 explicit 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`.

Varifiable? `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 241).

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 248)).

Varifiable? `par int` $\xrightarrow{v}$ `var int`, `var int` $\xrightarrow{v}$ `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. `par int` $\xrightarrow{c}$ `var int`, `par bool` $\xrightarrow{c}$ `par int`, `par bool` $\xrightarrow{c}$ `var int`, `var bool` $\xrightarrow{c}$ `var int`.

Also, integers can be automatically coerced to floats; see *Floats* (page 241).

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 248)).

Varifiable? `par float` $\xrightarrow{v}$ `var float`, `var float` $\xrightarrow{v}$ `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. `par int` $\xrightarrow{c}$ `par float`, `par int` $\xrightarrow{c}$ `var float`, `var int` $\xrightarrow{c}$ `var float`, `par float` $\xrightarrow{c}$ `var float`.

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 248)).

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 254)).

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 is a special case of the base type-inst rule:

```
<base-ti-expr> ::= <var-par> <opt-ti> "set" "of" <base-ti-expr-tail>
```

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 power set 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}`.

Varifiable? `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[1..3] of int: a1;
  array[0..3] of int: a2;
  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
  a2 = [3,2,6,5];     % index set mismatch
  a5 = [||];          % too few elements
  ```

  For `a2` above, the index set of the array literal `[3,2,6,5]` is (implicitly) `1..4`, so it cannot be assigned to an array declared with index set `0..3`, even though the length matches. A correct assignment would use the `array1d` function (see *Array Operations* (page 282)):

  ```
  a2 = array1d(0..3,[3,2,6,5]);     % correct
  ```

- *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 *Indexed Array Comprehensions* (page 259) 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> ::= "array" [ <ti-expr> "," ... ] "of" <base-ti-expr>
                  | "list" "of" <base-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.

Varifiable? 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. `array[TI0] of TI` $\xrightarrow{c}$ `array[UI0] of UI` if `TI0` $\xrightarrow{c}$ `UI0` and `TI` $\xrightarrow{c}$ `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>` if 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. `TI` $\xrightarrow{c}$ `opt UI` if `TI` $\xrightarrow{c}$ `UI`..

Tuple Types

Overview. Tuples are fixed-size, heterogeneous collections. They must contain at least two elements; unary tuples are not allowed.

Allowed Insts. Tuples may contain unfixed elements.

Syntax. A tuple base type-inst expression tail has this syntax:

```
<tuple-ti-expr-tail> ::= "tuple" ( <ti-expr> "," ... )
```

An example tuple type-inst expression:

```
tuple(int, var float)
```

Finite? Yes, if all its constituent elements are finite types. Otherwise, no. The domain of a tuple type that is a finite type is the Cartesian product of the domains of the element types. For example, the domain of `tuple(1..2, {3,5})` is `{(1,3), (1,5), (2,3), (2,5)}`.

Varifiable? Yes, if all its constituent elements are varifiable.

Ordering. Tuples are ordered lexicographically.

Initialisation. A tuple variable must be initialised at instance-time if any of its constituent elements must be initialised at instance-time.

Coercions. `tuple(TI1, ..., TIn)` $\xrightarrow{c}$ `tuple(UI1, ..., UIn)` if `TI1` $\xrightarrow{c}$ `UI1`, ..., `TIn` $\xrightarrow{c}$ `UIn`.

Record Types

Overview. Records are fixed-size, heterogeneous collections. They are similar to tuples, but have named fields. Field names in different records can be identical, because each record's field names belong to a different namespace.

The order in which a record's field are specified is irrelevant; the following two record type-insts are equivalent:

```
record(var int: x, var int: y)
record(var int: y, var int: x)
```

Allowed Insts. Records may contain unfixed elements.

Syntax. A record base type-inst expression tail has this syntax:

```
<record-ti-expr-tail> ::= "record" ( <ti-expr-and-id> "," ... )
```

An example record type-inst expression:

```
record(int: x, int: y)
```

Finite? Yes, if all its constituent elements are finite types. Otherwise, no. The domain of a record type that is a finite type is the same as that of a tuple type, but with the fields included. For example, the domain of `record(1..2: x, {3,5}: y)` is `{(x: 1,y: 3), (x: 1,y: 5), (x: 2,y: 3), (x: 2,y: 5)}`.

Varifiable? Yes, if all its constituent elements are varifiable.

Ordering. Records are ordered lexicographically according to the values of the fields, where the fields are sorted in alphabetical order.

Initialisation. A record variable must be initialised at instance-time if any of its constituent elements must be initialised at instance-time.

Coercions. `record(TI1: x1, ..., TIn: xn)` $\xrightarrow{c}$ `record(UI1: x1, ..., UIn: xn)` if `TI1` $\xrightarrow{c}$ `UI1`, ..., `TIn` $\xrightarrow{c}$ `UIn`.

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 271)).

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 User-defined Types and Type-insts

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`.

Type-inst Synonyms

Overview. A type-inst synonym is an alternative name for a pre-existing type-inst which can be used interchangeably with the pre-existing type-inst. For example, if `MyFixedInt` is a synonym for `par int` then `MyFixedInt` can be used anywhere `par int` can, and vice versa.

Allowed Insts. Preceding a type-inst synonym with `var` varifies it, unless the type-inst is not varifiable, in which case it is a type-inst error. Preceding a type-inst synonym with `par` has the reverse effect, and the type is explicitly made `par`.

Syntax. A type-inst synonym named "X" is represented by the term X.

Finite? As for the pre-existing type-inst.

Varifiable? Yes, if the pre-existing type-inst is varifiable.

Ordering. As for the pre-existing type-inst.

Initialisation. As for the pre-existing type-inst.

Coercions. As for the pre-existing type-inst.

4.1.6.8 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>
                   | <indexed-array-literal>
                   | <indexed-array-literal-2d>
                   | <tuple-literal>
                   | <record-literal>
                   | <array-comp>
                   | <indexed-array-comp>
                   | <ann-literal>
```

```
                    | <if-then-else-expr>
                    | <let-expr>
                    | <call-expr>
                    | <gen-call-expr>

<expr-atom-tail> ::=
                    | <array-access-tail> <expr-atom-tail>
                    | <field-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 250), 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 275) has more on annotations.

Expressions can be contained within parentheses.

The array and field access operations all bind more tightly than unary and binary operators and annotations. They are described in more detail in *Indexed Array Comprehensions* (page 259).

The remaining kinds of expression atoms (from `<ident>` to `<gen-call-expr>`) are described in *Identifier Expressions and Quoted Operator Expressions* (page 252) to *Generator Call Expressions* (page 265).

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" | "++" | "default" | <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" | "^"
                       | "~+" | "~-" | "~*" | "~/" | "~div"

<builtin-num-un-op> ::= "+" | "-"
```

Some operators can be written using their unicode symbols, which are listed in 表 1.1 (recall that MiniZinc input is UTF-8).

表 1.1: Unicode equivalents of binary operators

Operator	Unicode symbol	UTF-8 code
`<->`	↔	E2 86 94
`->`	→	E2 86 92
`<-`	←	E2 86 90
`not`	¬	C2 AC
`\/`	∨	E2 88 A8
`/\`	∧	E2 88 A7
`!=`	≠	E2 89 A0
`<=`	≤	E2 89 A4
`>=`	≥	E2 89 A5
`in`	∈	E2 88 88
`subset`	⊆	E2 8A 86
`superset`	⊇	E2 8A 87
`union`	∪	E2 88 AA
`intersect`	∩	E2 88 A9
`^-1`	$^{-1}$	E2 81 BB C2 B9

The binary operators are listed in 表 1.2. 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.

Note that the last entry in the table, `^-1`, is a combination of the binary power operator and the constant -1, which effectively turns it into a unary inverse operator. Other special cases (such as power of 2) are currently not supported yet.

表 1.2: Binary infix operators

Symbol(s)	Assoc.	Prec.
`<->`	left	1600
`->`	left	1500
`<-`	left	1500
`\/`	left	1400
`xor`	left	1400
`/\`	left	1300
`<`	none	1200
`>`	none	1200
`<=`	none	1200
`>=`	none	1200
`==`	none	1100
`=`	none	1100
`!=`	none	1100
`in`	none	1000
`subset`	none	1000
`superset`	none	1000
`union`	left	900
`diff`	left	900
`symdiff`	left	900

续下页

表 1.2 – 接上页

Symbol(s)	Assoc.	Prec.
intersect	left	800
..	none	700
<..	none	700
..<	none	700
<..<	none	700
+	left	600
-	left	600
*	left	500
div	left	500
mod	left	500
/	left	500
^	left	400
++	right	300
default	left	200
` <ident> `	left	100
::	left	0

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`. Unary `+` and `-` have the same precedence as the binary multiplication operator `*`. The logic `not` operator has precedence 350, i.e., it binds more tightly than all other Boolean and numeric operators.

User-defined unary operators are not possible.

As *Identifiers* (page 235) 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 278).

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 235). The following are examples of quoted operators:

```
'+'
'union'
```

In quoted operators, whitespace is not permitted between either quote and the operator. *Operators* (page 250) 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]+
                  | 0[xX]([0-9a-fA-F]*"."[0-9a-fA-F]+|[0-9a-fA-F]+".")([pP][+-]?[0-
↪9]+)
                  | (0[xX][0-9a-fA-F]+[pP][+-]?[0-9]+)
```

For example: `1.05`, `1.3e-5`, `1.3+e5`; 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\] |  \[0-7][0-7?][0-7]? | \x[0-9a-fA-F][0-9a-fA-F]? | \
↪n | \t | \" | \\)*

<string-literal> ::= """ <string-contents> """
                   | """ <string-contents> "\(" <string-interpolate-tail>

<string-interpolate-tail> ::= <expr> ")"<string-contents>"""
                            | <expr> ")"<string-contents>"\(" <string-interpolate-
↪tail>
```

This includes the following C-style escape sequences:

- `\"` for double quotes
- `\\` for backslash
- `\n` for newline
- `\t` for tab
- `\x[0-9a-fA-F][0-9a-fA-F]?` for an arbitrary 8-bit character specified by the given hexadecimal number
- `\x[0-7][0-7]?[0-7]?` for an arbitrary 8-bit character specified by the given octal number

For example: `"Hello, world!\n"`.

String literals must fit on a single line.

All string literals are represented and interpreted as UTF-8 strings. This means that you can either include UTF-8 characters directly in the strings, or you can use the hexadecimal or octal notation to insert the UTF-8 byte sequences, for example: `"\xe4\xbd\xa0\xe5\xa5\xbd means hello"` is a valid string.

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. This is called *string interpolation*. You need to use the `\(e)` syntax to interpolate an expression *e* into a 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.

Simple Array Literals

Simple array literals have this syntax:

```
<array-literal> ::= "[" [ <expr> "," ... ] "]"
```

For example:

```
[1, 2, 3, 4]
[]
[1, _]
```

In an 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 an array literal are implicitly `1..n`, where `n` is the length of the literal.

Indexed Array Literals

Indexed array literals have this syntax:

```
<indexed-array-literal> ::= "[" [ ( <index-tuple> ":" <expr> ) "," ... ] "]"
                          | "[" [ <index-tuple> ":" <expr> "," <expr> "," ... ] "]"

<index-tuple> ::= <expr>
                | "(" [ <expr> "," ... ] ")"
```

For example:

```
[ 1: 1, 2: 2, 3: 3, 4: 4, 5: 5]
[ A: 0, B: 3, C: 5]
[ (1,2): 1, (1,3): 2, (2,2): 3, (2,3): 4]
[ 1: 1, 4: 2, 5: 3, 3: 4, 2: 5]
```

```
[ 0: A, B, C ]
```

The expressions before the colons are the indexes (or keys), those after are the values.

The indexes must be of integer or enumerated type. The index set of the resulting array is, for each dimension, the union of the indexes of that dimension. For example, the index set of the first and fourth arrays above is `1..5`, the index set of the second array is `{A,B,C}`, and the index sets of the third array are `1..2` and `2..3`.

For one-dimensional arrays, it is possible to give only the starting index of the first element, as in the last example above. All following elements are indexed consecutively.

The index sets of indexed array literals must be contiguous and, for multi-dimensional arrays, "rectangular". For example, the following literals are not allowed:

```
% not contiguous, index 2 missing:
[ 1: 1, 3:2, 4:, 3]

% not contiguous, index 2 missing in dimension 2:
[ (1,1): 0, (1,3): 1, (2,1): 0, (2,3): 2]

% not rectangular, second row has fewer entries than first row:
[ (1,1): 0, (1,2): 0, (1,3): 0, (2,1): 1, (2,2): 2]
```

The indexes need not be specified in order (e.g. the last example above denotes the same array as the first one).

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)..(m,n)`, where `m` and `n` are determined by the shape of the literal.

Indexed 2d Array Literals

Indexed 2d array literals have this syntax:

```
<indexed-array-literal-2d> ::= "[|" [ <expr> ":" ... ] [ [ <expr> ":" ] <expr> "," .
↪.. ] "|]"
```

For example:

```
% only column index:
[| A: B: C:
 | 0, 0, 0
 | 1, 1, 1
 | 2, 2, 2 |];

% only row index:
[| A: 0, 0, 0
 | B: 1, 1, 1
 | C: 2, 2, 2 |];

% row and column index:
[|    A: B: C:
 | A: 0, 0, 0
 | B: 1, 1, 1
 | C: 2, 2, 2 |];
```

The index sets are either `1..n` for `n` rows/columns if no index is given, or the union of the given indexes. As for general indexed array literals, the union of the given indexes must be a contiguous set.

Simple Array Comprehensions

Simple array comprehensions have this syntax:

```
<array-comp> ::= "[" <expr> "|" <comp-tail> "]"
```

A generator expression (the part following the `in`) can be either a set or an array (multi-dimensional arrays are iterated over in row-major order).

For example (with the literal equivalents on the right):

```
[2*i | i in 1..5]         % [2, 4, 6, 8, 10]
[2*i | i in [3,5,7] ]     % [6, 10, 14]
[i   | i in [|1,2|3,4|] ] % [1, 2, 3, 4]
```

A generator variable in an array comprehension can also be the anonymous variable `_`. For instance, the following generates an array of `n*n` ones:

```
[ 1 | _, _ in 1..n ]
```

Array comprehensions have more flexible type and inst requirements than set comprehensions (see *Set Comprehensions* (page 255)).

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..n`, where `n` is the length of the evaluated comprehension.

Indexed Array Comprehensions

Indexed array comprehensions have this syntax:

```
<indexed-array-comp> ::= "[" <index-tuple> ":" <expr> "|" <comp-tail> "]"
```

For example:

```
% This equals [ 3:9, 4:12, 5:15 ]:
[ i: 3*i | i in 3..5 ]

% This generates a 2d array
% [|    1:  2:  3:
%  | 2:  7,  8,  9
%  | 3: 10, 11, 12
%  | 4: 13, 14, 15
%  |]:
[ (i,j): i*3+j | i in 2..4, j in 1..3]
```

The keys for indexed array comprehensions have the same requirements as for indexed array literals (see *Indexed Array Literals* (page 256)).

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`.

In addition to the `..` operator, you can also use the `<..`, `..<` and `<..<` operators (节 4.1.11.4) to exclude the first, last or both elements from the slice. This also works if you leave one or both of the bounds:

```
enum X = {A, B, C, D};
array[X, 1..3] of var int: x;
% x1 = x[A..C, 2..3]
array[_, _] of var int: x1 = x[A..<, <..];
% x2 = x[B..C, 3..3]
array[_, _] of var int: x2 = x[<..<, 2<..];
```

Record and Tuple Field Access Expressions

Fields of records and tuples are accessed using a dot after an expression:

```
<field-access-tail> ::= "."<ident> | "."<int-literal>
```

For example:

```
record(int:a, string: n): r = (a: 3, n: "foo");
int: x = r.a;
string: y = r.n;

tuple(float, bool): t = (42.0, true);
float: f = t.1;
bool: b = t.2;
```

The type-inst of the result is the type-inst of the field.

Field access can be chained, for example:

```
record(record(int:a, string:n): r1, bool: b): r2 =
    (r1: (a: 3, n: "foo"), b: true);
int: x = r2.r1.a;
```

Field access syntax can be used to project fields from arrays of records or tuples. For example:

```
array[1..3] of record(int:a, string:n): r_array =
    [ (a: 1, n: "one"), (a: 2, n: "two"), (a: 3, n: "three") ];
array[1..3] of int: a_array = r_array.a;
array[1..3] of string: n_array = r_array.n;
```

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`.

The `else` branch is optional if the type of the `then` branches is `bool`, `string`, `ann` or an `array` type. In that case, the `else` branch is taken to be the default value of the type:

- `false` for `bool`
- `""` for `string`
- `empty_annotation` for `ann`
- `[]` for `array`

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
```

For let local variables with initialisers, the type-inst can be omitted by using the `any` keyword, in which case the type-inst is inferred from the initialiser:

```
let { any: x = [5,6,7]; } in ...;    % x has type array[1..3] of int
```

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 286)).

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 | 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 (*Simple Array Comprehensions* (page 258)).

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 237) 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> ]
                  | "any" ":" <ident> <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 269)). 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.

Variable declarations with a definition can omit the type-inst by using the `any` keyword:

```
any: x = [1, 2, 3];
any: y = x[2];
```

The type-inst of these variables is inferred by MiniZinc to be that of the definition. In the example above, `x` would have type *array[1..3] of int*, and `y` would have type *int*. Note that variables with type-inst `any` can only be defined in-place, not using separate assignment items.

Variables can have one or more annotations. *Annotations* (page 275) has more on annotations.

4.1.8.3 Enum Items

Enumerated type items have this syntax:

```
<enum-item>  ::= "enum" <ident> <annotations> [ "=" <enum-cases-list> ]

<enum-cases-list> ::= <enum-cases>
                    | <enum-cases-list> "++" <enum-cases>

<enum-cases> ::= "{" <ident> "," ... "}"
               | "_" "(" <expr> ")"
```

```
                | <ident> "(" <ident> ")"
                | "anon_enum" "(" <expr> ")"
```

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 237)), case names in different enums must be distinct.

Instead of defining explicit, named enum cases, it is also possible to define an *anonymous enum*. The special enum constructor `_` takes an integer set argument. For example,

```
enum Slot = _(1..n);
```

defines an enumerated type `Slot` with `n` cases. An alternative syntax for anonymous enums is `enum Slot = anon_enum(n);`, which defines the same enum as the code above.

Enumerated type items can also be defined in terms of other sets and enumerated types by using an *enumerated type constructor* and the `++` operator, as in the following example:

```
enum country_or_none = C(country) ++ {None};
```

The `country_or_none` type now contains all of the cases of the `country` type, plus the new case `None`. The two enumerated types are however completely separate: `country` is *not* a subtype of `country_or_none`. In order to coerce a value of type `country` to type `country_or_none`, use the constructor. E.g., `C(Canada)` is of type `country_or_none`. Similarly, you can use the *inverse constructor* to coerce a value back:

```
country_or_none: c_o_n = C(England);
country: c = C ¹(c_o_n);
```

The inverse operator can be written using unicode characters (as in the example) or using ASCII syntax, e.g. `C^-1(c_o_n)`.

The same enum constructor syntax also works with integer sets, for example

```
enum Node = Left(1..n) ++ Right(1..n);
```

declares an enum for nodes in a bipartite graph with `n` left nodes and `n` right nodes.

The enum constructor can also be used without arguments to return the set of values of the particular constructor. For example, given a `Node: n`, we can use the syntax *n in Left()* to test if `n` is in the `Left` set.

Enum constructors can be used to map non-contiguous sets to (contiguous) enumerated types. Consider the following example:

```
enum Person = { Amy, Bert, Celeste, Doug, Emely };
enum Staff = S({Amy, Doug});
enum Customers = C(Person diff S ¹(Staff));
```

An enum constructor and its inverse can be applied to sets and arrays of enum values as well:

```
array[int] of Customers: custs = S([Bert])
array[int] of Person: custs_as_people = C ¹(custs);
```

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 275) 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(T: x);
function var T: enum_next(var T: x);
```

```
% Return next smaller enum value of x in enum type X
function T: enum_prev(T: x);
function var T: enum_prev(var T: x);

% Return enum base set for value x in enum type x
function set of T: enum_of(T: x);
function set of T: enum_of(var T: x);
function set of T: enum_of(set of T: x);
function set of T: enum_of(var set of T: x);

% Convert enum type x to an integer
function int: enum2int(T: x);
function var int: enum2int(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 Type-inst Synonym Items

Type-inst synonym items have this syntax:

```
<type-inst-syn-item> ::= "type" <ident> <annotations> "=" <ti-expr>
```

For example:

```
type MyInt     = int;
type Person    = record(string: name, int: height);
type Domain    = 1..n;
```

It is a type-inst error if a type-inst synonym is declared and/or defined more than once in a model. Type-inst synonym items can be annotated. Section (*Annotation Items* (page 271)) has more details on annotations. All type-inst synonyms must be defined at model-time.

4.1.8.5 Assignment Items

Assignments have this syntax:

```
<assign-item> ::= <ident> "=" <expr>
```

For example:

```
A = 10;
```

4.1.8.6 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.

4.1.8.7 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 275) has more details on annotations.

4.1.8.8 Output Items

Output items are used to present the solutions of a model instance. They have the following syntax:

```
<output-item> ::= "output" [ <string-annotation> ] <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 278)), as well as string interpolations (*String Literals and String Interpolation* (page 254)). 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.

Output items can optionally given a string annotation denoting a section name which the output is associated with. Sections can be turned on or off using the `--only-sections` and `--not-sections` command line arguments (*The MiniZinc Command Line Tool* (page 125)).

JSON Output Items

Output items can also be used to format an expression as JSON (see *JSON support* (page 287)) by annotating the output item with `json_section("<section name>")`:

```
output :: json_section("my_section") (a: 1, b: [2, 3]);
```

The output expression can have any type and is encoded using `showJSON` function. If there are multiple output items with the same JSON section name, only the first one is used.

4.1.8.9 Annotation Items

Annotation items are used to augment the `ann` type. They have the following syntax:

```
<annotation-item> ::= "annotation" <ident> <params> [ "=" <expr> ]
```

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 275).

4.1.8.10 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).

Declarations for user-defined operations can be annotated. *Annotations* (page 275) 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.

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 overloadings 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 overloadings 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 overloadings of an operation must be semantically equivalent with respect to coercion. For example, the two overloadings 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 overloadings 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 overloadings of an operation must be closed under intersection of their input type-insts. That is, if overloaded versions have input type-inst $(S_1, ..., S_n)$ and $(T_1, ..., T_n)$ then there must be another overloaded version with input type-inst $(R_1, ..., R_n)$ where each R_i is the greatest lower bound (*glb*) of S_i and T_i.

Also, all overlapping overloadings of an operation must be monotonic. That is, if there are overloaded versions with input type-insts $(S_1, ..., S_n)$ and $(T_1, ..., T_n)$ 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 overloadings.

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 263) 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: '::'(var $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 \/ 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 div y) * y + (x 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.

```
$T:     min(    $T,     $T)
var $T: min(var $T, var $T)
```

Minimum of an array of values; maximum (`max`) is similar. Aborts if the array is empty.

```
$U:     min(array[$T]  of     $U)
var $U: min(array[$T]  of var $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 (`\/`) 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 $$E,   var set of $$E)
```

Non-strict subset. Non-strict superset (`superset`) is similar.

```
    bool: 'subset'(    set of $T ,     set of $T )
var bool: 'subset'(var set of $$E, var set of $$E)
```

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 $$E: 'union'(var set of $$E, var set of $$E )
```

Set ranges. Constructs a set given a lower and an upper bound. Whether the operator constructs a set that includes or excludes the bounds depends on the placement of the < symbol in the operator. If the first argument is larger than the second (e.g. `1..0`), it returns the empty set. The arguments to the operators can be either integers or values of enumerated types (`l+1` and `u-1` in the comments mean `enum_next(l)` and `enum_prev(l)` in that case).

```
% Return set {l, ..., u}
set of $$E: '..'($$E: l, $$E: u)
% Return set {l+1, ..., u}
set of $$E: '<..'($$E: l, $$E: u)
% Return set {l, ..., u-1}
set of $$E: '..<'($$E: l, $$E: u)
% Return set {l+1, ..., u-1}
set of $$E: '<..<'($$E: l, $$E: u)
```

Note that set range operators can be used with only one of the bounds given. The minimum or maximum of the type of the bound is then used for the other bound:

```
% x = {l, ..., max(enum_of(l))}
set of X: x = l..;
% x = {min(enum_of(u)), ..., u}
set of X: x = ..u;
```

Cardinality of a set.

```
    int: card(    set of  $T)
var int: card(var set of $$E)
```

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 $$E:    array_union(array[$T]  of var set of $$E)
```

4.1.11.5 Array Operations

Length of an array.

```
int: length(array[$T] of $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 $T: '++'(array[int] of $T, array[int] of $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 $V)
set of $T:  index_set_1of2(array[$T, $U]  of $V)
set of $U:  index_set_2of2(array[$T, $U]  of $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 $V: array1d(set of $T1, array[$U] of $V)
array[$T1,$T2] of $V:
    array2d(set of $T1, set of $T2, array[$U] of $V)
array[$T1,$T2,$T3] of $V:
```

```
    array3d(set of $T1, set of $T2, set of $T3, array[$U] of $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)
```

Capturing of undefinedness and optionality. These binary infix operators return `x` if it is both defined and not absent, and `y` otherwise.

```
                      $T: 'default'(opt $T:x, $T: y);
                  opt $T: 'default'(opt $T:x, opt $T: y);
                  var $T: 'default'(var opt $T:x, var $T: y);
              var opt $T: 'default'(var opt $T:x, var opt $T: y);
array[$U] of         $T: 'default'(array[$U] of $T:x, array[$U] of $T: y);
array[$U] of opt     $T: 'default'(array[$U] of opt $T:x, array[$U] of opt $T: y);
array[$U] of var     $T: 'default'(array[$U] of var $T:x, array[$U] of var $T: y);
array[$U] of var opt $T: 'default'(array[$U] of var opt $T:x, array[$U] of var opt
↪$T: y);
```

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($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 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.

```
$T:   assert(bool, string, s$T)
par bool: assert(bool, string)
```

Abort evaluation, printing the given string.

```
$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.

```
$T: trace(string, $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(var $T)
```

As above, but return `false` if the argument's value is not fixed.

```
par bool: is_fixed(var $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 234), and is discussed in detail in *Output* (page 234).

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 270). 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). The list may additionally contain an el-

ement which is a sublist comprising a pair of values representing a minimum and maximum of a range in the set.

- A value of an enumerated type encoded as an object with a single member with key "e" and a string value (the identifier of the enumerated value).
- The value of an enumerated type constructor call encoded as an object with a member with key "c" and a string value (the identifier of the constructor call), and a member with key "e" and a enumerated type (or constructor) object value (the argument to the call).
- A list of enum constructors used to define the members of an enumerated type. Each constructor is either a string value for an atomic enum value, or an object with members "c" and "e" as above for a constructor call.

MiniZinc supports coercion of some JSON input types for more convenient input:

- A list of values can be coerced to a set
- A string value can be coerced to an enumerated type value (but not a constructor call)

Assume a MiniZinc model declaring the following parameters:

```
int: n;
enum Customers;
array[1..n,Customers] of int: distances;
array[1..n] of set of int: patterns;
```

Here is an example of a JSON parameter file using all of the above features:

```
{
  "n" : 2,
  "Customers" : [ {"e" : "Customer A"}, {"e" : "Customer B"}, {"e" : "Customer C"}␣
↪],
  "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 `Customers` parameter defines the members of an enumerated type. 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 `patterns` parameter is an array of sets of integers.

4.1.14 Full grammar

4.1.14.1 Items

```
% A MiniZinc model
<model> ::= [ <item> ";" ... ]

% Items
<item>  ::= <include-item>
```

```
            | <var-decl-item>
            | <enum-item>
            | <type-inst-syn-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> ]
                  | "any" ":" <ident> <annotations> [ "=" <expr> ]

% Enum items
<enum-item>  ::= "enum" <ident> <annotations> [ "=" <enum-cases-list> ]

<enum-cases-list> ::= <enum-cases>
                    | <enum-cases-list> "++" <enum-cases>

<enum-cases> ::= "{" <ident> "," ... "}"
               | "_" "(" <expr> ")"
               | <ident> "(" <ident> ")"
               | "anon_enum" "(" <expr> ")"

% Type-inst synonym items
<type-inst-syn-item> ::= "type" <ident> <annotations> "=" <ti-expr>

% 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" [ <string-annotation> ] <expr>

% Annotation items
<annotation-item> ::= "annotation" <ident> <params> [ "=" <expr> ]

% 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>
            | <array-ti-expr>

<base-ti-expr> ::= <var-par> <opt-ti> <set-ti> <base-ti-expr-tail> ["++" <base-ti-
↪expr>]
                 | "any" <ti-variable-expr-tail>

<var-par> ::= "var" | "par" |

<opt-ti> ::= "opt" |

<set-ti> ::= "set" "of" |

<base-type> ::= "bool"
              | "int"
              | "float"
              | "string"

<base-ti-expr-tail> ::= <ident>
                      | <base-type>
                      | <ti-variable-expr-tail>
                      | <tuple-ti-expr-tail>
                      | <record-ti-expr-tail>
                      | "ann"
                      | <expr>

% Type-inst variables
<ti-variable-expr-tail> ::= $[A-Za-z$][A-Za-z0-9_]*

% Array type-inst expressions
<array-ti-expr> ::= "array" [ <ti-expr> "," ... ] "of" <base-ti-expr>
                  | "list" "of" <base-ti-expr>

% Tuple type-inst expressions
<tuple-ti-expr-tail> ::= "tuple" ( <ti-expr> "," ... )

% Record type-inst expressions
<record-ti-expr-tail> ::= "record" ( <ti-expr-and-id> "," ... )
```

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>
                   | <indexed-array-literal>
                   | <indexed-array-literal-2d>
                   | <tuple-literal>
                   | <record-literal>
                   | <array-comp>
                   | <indexed-array-comp>
                   | <ann-literal>
                   | <if-then-else-expr>
                   | <let-expr>
                   | <call-expr>
                   | <gen-call-expr>

<expr-atom-tail> ::=
                   | <array-access-tail> <expr-atom-tail>
                   | <field-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" | "++" | "default" | <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" | "^"
                       | "~+" | "~-" | "~*" | "~/" | "~div"

<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]+
                  | 0[xX]([0-9a-fA-F]*"."[0-9a-fA-F]+|[0-9a-fA-F]+".")([pP][+-]?[0-
↪9]+)
                  | (0[xX][0-9a-fA-F]+[pP][+-]?[0-9]+)

% String literals
<string-contents> ::= ([^"\n\] |  \[0-7][0-7?][0-7]? | \x[0-9a-fA-F][0-9a-fA-F]? | \
↪n | \t | \" | \\)*

<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> ] "," ...

<generator> ::= ( <ident> | "_" ) "," ... "in" <expr>

% Array literals
<array-literal> ::= "[" [ <expr> "," ... ] "]"
```

```
% 2D Array literals
<array-literal-2d> ::= "[|" [ [ <expr> "," ... ] "|" ... ] "|]"

% Indexed 2D Array literals
<indexed-array-literal-2d> ::= "[|" [ <expr> ":" ... ] [ [ <expr> ":" ] <expr> "," .
↪.. ] "|]"

% Indexed array literals
<indexed-array-literal> ::= "[" [ ( <index-tuple> ":" <expr> ) "," ... ] "]"
                          | "[" [ <index-tuple> ":" <expr> "," <expr> "," ... ] "]"

<index-tuple> ::= <expr>
                | "(" [ <expr> "," ... ] ")"

% Tuple literals
<tuple-literal> ::= "(" <expr> "," [ <expr> "," ... ] ")"

% Record literals
<record-literal> ::= "(" <ident> ":" <expr> "," [ <ident> ":" <expr> "," ... ] ")"

% Array comprehensions
<array-comp> ::= "[" <expr> "|" <comp-tail> "]"

% Indexed array comprehensions
<indexed-array-comp> ::= "[" <index-tuple> ":" <expr> "|" <comp-tail> "]"

% Array access
<array-access-tail> ::= "[" <expr> "," ... "]"

% Field access
<field-access-tail> ::= "."<ident> | "."<int-literal>

% 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 Standard Library

These functions and predicates define built-in operations of the MiniZinc language.

4.2.1.1 Built-in functions and operators

These functions and operators provide the core of the MiniZinc expression language.

Comparison Builtins

These builtins implement comparison operations.

```
1.  test '!='($T: x, $T: y)

2.  test '!='(opt $T: x, opt $T: y)

3.  predicate '!='(any $T: x, any $T: y)

4.  test '!='(array [$U] of $T: x, array [$U] of $T: y)

5.  test '!='(array [$U] of opt $T: x, array [$U] of opt $T: y)

6.  predicate '!='(array [$U] of var $T: x, array [$U] of var $T: y)

7.  predicate '!='(array [$U] of var opt $T: x,
                   array [$U] of var opt $T: y)
```

1-3.

Return if `x` is not equal to `y`

Usage: `x != y`

4-7.

Return if array x is not equal to array y

Usage: `x != y`

```
1.  test '<'($T: x, $T: y)
2.  predicate '<'(var $T: x, var $T: y)
3.  predicate '<'($T: x, var $T: y)
4.  predicate '<'(var $T: x, $T: y)
5.  predicate '<'(var opt $$E: x, var opt $$E: y)
6.  test '<'(opt $$E: x, opt $$E: y)
7.  predicate '<'(var opt float: x, var opt float: y)
8.  test '<'(opt float: x, opt float: y)
9.  test '<'(array [$U] of $T: x, array [$U] of $T: y)
10. predicate '<'(array [$U] of var int: x, array [$U] of var int: y)
11. predicate '<'(array [$U] of var bool: x, array [$U] of var bool: y)
12. predicate '<'(array [$U] of var float: x, array [$U] of var float: y)
13. predicate '<'(array [$U] of var set of $$E: x,
                  array [$U] of var set of $$E: y)
```

1-4.

Return if x is less than y

Usage: `x < y`

5-8.

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.

Usage: `x < y`

9-13.

Return if array x is lexicographically smaller than array y

Usage: `x < y`

```
1.  test '<='($T: x, $T: y)
2.  predicate '<='(var $T: x, var $T: y)
3.  predicate '<='($T: x, var $T: y)
```

```
4.  predicate '<='(var $T: x, $T: y)

5.  predicate '<='(var opt $$E: x, var opt $$E: y)

6.  test '<='(opt $$E: x, opt $$E: y)

7.  predicate '<='(var opt float: x, var opt float: y)

8.  test '<='(opt float: x, opt float: y)

9.  test '<='(array [$U] of $T: x, array [$U] of $T: y)

10. predicate '<='(array [$U] of var int: x, array [$U] of var int: y)

11. predicate '<='(array [$U] of var bool: x, array [$U] of var bool: y)

12. predicate '<='(array [$U] of var float: x, array [$U] of var float: y)

13. predicate '<='(array [$U] of var set of $$E: x,
                   array [$U] of var set of $$E: y)
```

1-4.

Return if `x` is less than or equal to `y`

Usage: `x <= y`

5-8.

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`.

Usage: `x <= y`

9-12.

Return if array `x` is lexicographically smaller than or equal to array `y`

Usage: `x <= y`

13. Return if array `x` is lexicographically $$E than or equal to array `y`

Usage: `x <= y`

```
1.  test '='($T: x, $T: y)

2.  test '='(opt $T: x, opt $T: y)

3.  predicate '='(any $T: x, any $T: y)

4.  test '='(array [$U] of $T: x, array [$U] of $T: y)

5.  test '='(array [$U] of opt $T: x, array [$U] of opt $T: y)

6.  predicate '='(array [$U] of var $T: x, array [$U] of var $T: y)

7.  predicate '='(array [$U] of var opt $T: x, array [$U] of var opt $T: y)
```

1-3.
Return if x is equal to y

Usage: x = y

4-7.
Return if array x is equal to array y

Usage: x = y

```
1.  test '>'($T: x, $T: y)

2.  predicate '>'(var $T: x, var $T: y)

3.  predicate '>'($T: x, var $T: y)

4.  predicate '>'(var $T: x, $T: y)

5.  predicate '>'(var opt $$E: x, var opt $$E: y)

6.  test '>'(opt $$E: x, opt $$E: y)

7.  predicate '>'(var opt float: x, var opt float: y)

8.  test '>'(opt float: x, opt float: y)

9.  test '>'(array [$U] of $T: x, array [$U] of $T: y)

10. predicate '>'(array [$U] of var int: x, array [$U] of var int: y)

11. predicate '>'(array [$U] of var bool: x, array [$U] of var bool: y)

12. predicate '>'(array [$U] of var float: x, array [$U] of var float: y)

13. predicate '>'(array [$U] of var set of $$E: x,
                  array [$U] of var set of $$E: y)
```

1-4.
Return if x is greater than y

Usage: x > y

5-8.
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.

Usage: x > y

9-13.
Return if array x is lexicographically greater than array y

Usage: x > y

```
1.  test '>='($T: x, $T: y)

2.  predicate '>='(var $T: x, var $T: y)

3.  predicate '>='($T: x, var $T: y)

4.  predicate '>='(var $T: x, $T: y)

5.  predicate '>='(var opt $$E: x, var opt $$E: y)

6.  test '>='(opt $$E: x, opt $$E: y)

7.  predicate '>='(var opt float: x, var opt float: y)

8.  test '>='(opt float: x, opt float: y)

9.  test '>='(array [$U] of $T: x, array [$U] of $T: y)

10. predicate '>='(array [$U] of var int: x, array [$U] of var int: y)

11. predicate '>='(array [$U] of var bool: x, array [$U] of var bool: y)

12. predicate '>='(array [$U] of var float: x, array [$U] of var float: y)

13. predicate '>='(array [$U] of var set of $$E: x,
                   array [$U] of var set of $$E: y)
```

1-4.

Return if `x` is greater than or equal to `y`

Usage: `x >= y`

5-8.

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`.

Usage: `x >= y`

9-13.

Return if array `x` is lexicographically greater than or equal to array `y`

Usage: `x >= y`

```
predicate '~!='(var opt bool: x, var opt bool: y)
predicate '~!='(var opt $$E: x, var opt $$E: y)
predicate '~!='(var opt float: x, var opt float: y)
```

Weak disequality. True if either `x` or `y` are absent, or present and not equal.

```
predicate '~='(var opt bool: x, var opt bool: y)
predicate '~='(var opt $$E: x, var opt $$E: y)
predicate '~='(var opt float: x, var opt float: y)
```

Weak equality. True if either x or y are absent, or present and equal.

Arithmetic Builtins

These builtins implement arithmetic operations.

```
1. function int: '*'(int: x, int: y)
2. function var int: '*'(var int: x, var int: y)
3. function var int: '*'(var opt int: x, var opt int: y)
4. function int: '*'(opt int: x, opt int: y)
5. function var float: '*'(var opt float: x, var opt float: y)
6. function float: '*'(opt float: x, opt float: y)
7. function float: '*'(float: x, float: y)
8. function var float: '*'(var float: x, var float: y)
```

1, 2.
Return x * y

Usage: `x * y`

3-6.
Optional multiplication. Return product of x and y, with absent replaced by 1.

Usage: `x * y`

7, 8.
Return x * y

Usage: `x * y`

```
1. function int: '+'(int: x, int: y)
2. function var int: '+'(var int: x, var int: y)
3. function float: '+'(float: x, float: y)
4. function var float: '+'(var float: x, var float: y)
5. function var int: '+'(var opt int: x, var opt int: y)
6. function int: '+'(opt int: x, opt int: y)
7. function var float: '+'(var opt float: x, var opt float: y)
8. function float: '+'(opt float: x, opt float: y)
```

1-4.
Return x + y

Usage: x + y

5-8.
Optional addition. Return sum of x and y, with absent replaced by 0.

Usage: x + y

```
1.  function int: '-'(int: x, int: y)
2.  function var int: '-'(var int: x, var int: y)
3.  function float: '-'(float: x, float: y)
4.  function var float: '-'(var float: x, var float: y)
5.  function var opt int: '-'(var opt int: x, var opt int: y)
6.  function var opt float: '-'(var opt float: x, var opt float: y)
7.  function opt int: '-'(opt int: x, opt int: y)
8.  function opt float: '-'(opt float: x, opt float: y)
9.  function int: '-'(int: x)
10. function var int: '-'(var int: x)
11. function float: '-'(float: x)
12. function var float: '-'(var float: x)
13. function opt int: '-'(opt int: x)
14. function var opt int: '-'(var opt int: x)
15. function opt float: '-'(opt float: x)
16. function var opt float: '-'(var opt float: x)
```

1-4.
Return x - y

Usage: x - y

5-8.
Optional subtraction. Return absent if x is absent, x if y is absent, difference of x and y if both are present.

Usage: x - y

9-12.
Return negative x

Usage: - x

13-16.
Return negative x if it occurs, otherwise return absent

Usage: `- x`

```
1. function float: '/'(float: x, float: y)

2. function var float: '/'(var float: x, var float: y)

3. function var opt float: '/'(var opt float: x, var opt float: y)

4. function opt float: '/'(opt float: x, opt float: y)
```

1, 2.
Return result of floating point division x / y

Usage: `x / y`

3, 4.
Optional division. Return absent if x is absent, x if y is absent, x divided by y if both are present.

Usage: `x / y`

```
function int: '^'(int: x, int: y)
function var int: '^'(var int: x, var int: y)
function float: '^'(float: x, float: y)
function var float: '^'(var float: x, var float: y)
```

Return $\mathbf{x}^{\mathbf{y}}$

Usage: `x ^ y`

```
1. function int: 'div'(int: x, int: y)

2. function var int: 'div'(var int: x, var int: y)

3. function var opt int: 'div'(var opt int: x, var opt int: y)

4. function opt int: 'div'(opt int: x, opt int: y)
```

1, 2.
Return result of integer division x / y

Usage: `x div y`

3, 4.
Optional division. Return absent if x is absent, x if y is absent, x divided by y if both are present.

Usage: `x div y`

```
1. function int: 'mod'(int: x, int: y)
2. function var int: 'mod'(var int: x, var int: y)
3. function var opt int: 'mod'(var opt int: x, var opt int: y)
4. function opt int: 'mod'(opt int: x, opt int: y)
```

1, 2.
Return remainder of integer division `x % y`

Usage: `x mod y`

3, 4.
Optional modulo. Return absent if `x` or `y` is absent, `x` modulo `y` if both are present.

Usage: `x mod y`

```
1. function int: abs(int: x)
2. function var int: abs(var int: x)
3. function opt int: abs(opt int: x)
4. function var opt int: abs(var opt int: x)
5. function float: abs(float: x)
6. function var float: abs(var float: x)
7. function opt float: abs(opt float: x)
8. function var opt float: abs(var opt float: x)
```

1, 2.
Return absolute value of `x`

3, 4.
Return absolute value of `x` if it occurs, otherwise return absent

5, 6.
Return absolute value of `x`

7, 8.
Return absolute value of `x` if it occurs, otherwise return absent

```
function $$E: arg_max(array [$$E] of bool: x)
function $$E: arg_max(array [$$E] of int: x)
function $$E: arg_max(array [$$E] of float: x)
```

Returns the index of the maximum value in the array x. When breaking ties the least index is returned.

```
function $$E: arg_min(array [$$E] of bool: x)
function $$E: arg_min(array [$$E] of int: x)
function $$E: arg_min(array [$$E] of float: x)
```

Returns the index of the minimum value in the array x. When breaking ties the least index is returned.

```
function int: count(array [$T] of bool: x)
function var int: count(array [$T] of var bool: x)
function var int: count(array [$T] of var opt bool: x)
```

Return number of true elements in array x

```
1.  function $T: max($T: x, $T: y)

2.  function var float: max(var float: x, var float: y)

3.  function $T: max(array [$U] of $T: x)

4.  function var $$T: max(var $$T: x, var $$T: y)

5.  function $$T: max(array [$X] of opt $$T: x)

6.  function float: max(array [$X] of opt float: x)

7.  function var $$T: max(array [$U] of var $$T: x)

8.  function var float: max(array [$U] of var float: x)

9.  function var $$T: max(array [$X] of var opt $$T: x)

10. function var float: max(array [$X] of var opt float: x)

11. function $$E: max(set of $$E: x)
```

1, 2.
Return maximum of x and y

3. Return maximum of elements in array x

4. Return maximum of x and y

5, 6.
Return maximum of elements in x that are not absent. Result is undefined when all values are absent.

7, 8.
Return maximum of elements in array x

9, 10.
Return maximum of elements in x that are not absent. Result is undefined when all values are absent.

11. Return maximum of elements in set x

```
function opt $$T: max_weak(array [$X] of opt $$T: x)
function opt float: max_weak(array [$X] of opt float: x)
function var opt $$T: max_weak(array [$X] of var opt $$T: x)
function var opt float: max_weak(array [$X] of var opt float: x)
```

Return maximum of elements in x that are not absent. Result is absent when all values are absent.

```
1.  function $T: min($T: x, $T: y)

2.  function var $$T: min(var $$T: x, var $$T: y)

3.  function var float: min(var float: x, var float: y)

4.  function $T: min(array [$U] of $T: x)

5.  function $$T: min(array [$X] of opt $$T: x)

6.  function float: min(array [$X] of opt float: x)

7.  function var $$T: min(array [$U] of var $$T: x)

8.  function var float: min(array [$U] of var float: x)

9.  function var $$T: min(array [$X] of var opt $$T: x)

10. function var float: min(array [$X] of var opt float: x)

11. function $$E: min(set of $$E: x)
```

1-3.
Return minimum of x and y

4. Return minimum of elements in array x

5, 6.
Return minimum of elements in x that are not absent. Result is undefined when all values are absent.

7, 8.
Return minimum of elements in array x

9, 10.
Return minimum of elements in x that are not absent. Result is undefined when all values are absent.

11. Return minimum of elements in set x

```
function opt $$T: min_weak(array [$X] of opt $$T: x)
function opt float: min_weak(array [$X] of opt float: x)
function var opt $$T: min_weak(array [$X] of var opt $$T: x)
function var opt float: min_weak(array [$X] of var opt float: x)
```

1, 2.
Return maximum of elements in `x` that are not absent. Result is absent when all values are absent.

3, 4.
Return minimum of elements in `x` that are not absent. Result is absent when all values are absent.

```
function int: pow(int: x, int: y)
function var int: pow(var int: x, int: y)
function var int: pow(var int: x, var int: y)
function float: pow(float: x, float: y)
function var float: pow(var float: x, var float: y)
```

1-3.
Return $\mathbf{x}^{\mathbf{y}}$, when $\mathbf{y} < 0$ it returns `1 div pow(x, abs(y))`.

4, 5.
Return $\mathbf{x}^{\mathbf{y}}$

```
function int: product(array [$T] of int: x)
function var int: product(array [$T] of var int: x)
function float: product(array [$T] of float: x)
function var float: product(array [$T] of var float: x)
function int: product(array [$T] of opt int: x)
function var int: product(array [$T] of var opt int: x)
function float: product(array [$T] of opt float: x)
function var float: product(array [$T] of var opt float: x)
```

1-4.
Return product of elements in array `x`

5-8.
Return product of non-absent elements of x.

```
1. function float: sqrt(float: x)
2. function var float: sqrt(var float: x)
3. function opt float: sqrt(opt float: x)
4. function var opt float: sqrt(var opt float: x)
```

1, 2.
Return $\sqrt{\mathbf{x}}$

3, 4.
Return $\sqrt{\mathbf{x}}$ if x occurs, otherwise return absent

---

```
1. function int: sum(array [$T] of int: x)
2. function var int: sum(array [$T] of var int: x)
3. function float: sum(array [$T] of float: x)
4. function var float: sum(array [$T] of var float: x)
5. function var int: sum(array [int] of var opt int: x)
6. function int: sum(array [int] of opt int: x)
7. function var float: sum(array [int] of var opt float: x)
8. function float: sum(array [int] of opt float: x)
```

1-4.
Return sum of elements in array x

5-8.
Return sum of non-absent elements of x.

```
function var opt int: '~*'(var opt int: x, var opt int: y)
function opt int: '~*'(opt int: x, opt int: y)
function var opt float: '~*'(var opt float: x, var opt float: 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)
function opt int: '~+'(opt int: x, opt int: y)
function var opt float: '~+'(var opt float: x, var opt float: 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)
function opt int: '~-'(opt int: x, opt int: y)
function var opt float: '~-'(var opt float: x, var opt float: y)
```

Weak subtraction. Return difference of x and y if both are present, otherwise return absent.

```
function var opt float: '~/'(var opt float: x, var opt float: y)
function opt float: '~/'(opt float: x, opt float: y)
```

Weal optional division. Return absent if x or y is absent, x divided by y if both are present.

```
function var opt int: '~div'(var opt int: x, var opt int: y)
function opt int: '~div'(opt int: x, opt int: y)
```

Weak optional division. Return absent if x or y is absent, x divided by y if both are present.

Exponential and logarithmic builtins

These builtins implement exponential and logarithmic functions.

```
1. function float: exp(float: x)

2. function var float: exp(var float: x)

3. function opt float: exp(opt float: x)

4. function var opt float: exp(var opt float: x)
```

1, 2.
Return $e^{\mathbf{x}}$

3, 4.
Return $e^{\mathbf{x}}$ if x occurs, otherwise return absent

```
1. function float: ln(float: x)
2. function var float: ln(var float: x)
3. function opt float: ln(opt float: x)
4. function var opt float: ln(var opt float: x)
```

1, 2.
Return $\ln \mathbf{x}$

3, 4.
Return $\ln \mathbf{x}$ if x occurs, otherwise return absent

```
function float: log(float: x, float: y)
```

Return $\log_{\mathbf{x}} \mathbf{y}$

```
1. function float: log10(float: x)
2. function var float: log10(var float: x)
3. function opt float: log10(opt float: x)
4. function var opt float: log10(var opt float: x)
```

1, 2.
Return $\log_{10} \mathbf{x}$

3, 4.
Return $\log_{10} \mathbf{x}$ if x occurs, otherwise return absent

---

```
1. function float: log2(float: x)
2. function var float: log2(var float: x)
3. function opt float: log2(opt float: x)
4. function var opt float: log2(var opt float: x)
```

1, 2.
Return $\log_2 \mathbf{x}$

3, 4.
Return $\log_2 \mathbf{x}$ if x occurs, otherwise return absent

---

Trigonometric functions

These builtins implement the standard trigonometric functions.

```
1. function float: acos(float: x)
2. function var float: acos(var float: x)
3. function opt float: acos(opt float: x)
4. function var opt float: acos(var opt float: x)
```

1, 2.
Return acos **x**

3, 4.
Return acos **x** if x occurs, otherwise return absent

```
1. function float: acosh(float: x)
2. function var float: acosh(var float: x)
3. function opt float: acosh(opt float: x)
4. function var opt float: acosh(var opt float: x)
```

1, 2.
Return acosh **x**

3, 4.
Return acosh **x** if x occurs, otherwise return absent

```
1. function float: asin(float: x)
2. function var float: asin(var float: x)
3. function opt float: asin(opt float: x)
4. function var opt float: asin(var opt float: x)
```

1, 2.
Return asin **x**

3, 4.
Return asin **x** if x occurs, otherwise return absent

```
1. function float: asinh(float: x)
2. function var float: asinh(var float: x)
```

```
3. function opt float: asinh(opt float: x)
4. function var opt float: asinh(var opt float: x)
```

1, 2.
Return asinh **x**

3, 4.
Return asinh **x** if x occurs, otherwise return absent

```
1. function float: atan(float: x)
2. function var float: atan(var float: x)
3. function opt float: atan(opt float: x)
4. function var opt float: atan(var opt float: x)
```

1, 2.
Return atan **x**

3, 4.
Return atan **x** if x occurs, otherwise return absent

```
1. function float: atanh(float: x)
2. function var float: atanh(var float: x)
3. function opt float: atanh(opt float: x)
4. function var opt float: atanh(var opt float: x)
```

1, 2.
Return atanh **x**

3, 4.
Return atanh **x** if x occurs, otherwise return absent

```
1. function float: cos(float: x)
2. function var float: cos(var float: x)
3. function opt float: cos(opt float: x)
4. function var opt float: cos(var opt float: x)
```

1, 2.
Return cos **x**

3, 4.
Return cos **x** if x occurs, otherwise return absent

```
1. function float: cosh(float: x)
2. function var float: cosh(var float: x)
3. function opt float: cosh(opt float: x)
4. function var opt float: cosh(var opt float: x)
```

1, 2.
Return cosh **x**

3, 4.
Return cosh **x** if x occurs, otherwise return absent

```
1. function float: sin(float: x)
2. function var float: sin(var float: x)
3. function opt float: sin(opt float: x)
4. function var opt float: sin(var opt float: x)
```

1, 2.
Return sin **x**

3, 4.
Return sin **x** if x occurs, otherwise return absent

```
1. function float: sinh(float: x)
2. function var float: sinh(var float: x)
3. function opt float: sinh(opt float: x)
4. function var opt float: sinh(var opt float: x)
```

1, 2.
Return sinh **x**

3, 4.
Return sinh **x** if x occurs, otherwise return absent

```
1. function float: tan(float: x)
2. function var float: tan(var float: x)
```

```
3. function opt float: tan(opt float: x)
4. function var opt float: tan(var opt float: x)
```

1, 2.
Return tan **x**

3, 4.
Return tan **x** if x occurs, otherwise return absent

```
1. function float: tanh(float: x)
2. function var float: tanh(var float: x)
3. function opt float: tanh(opt float: x)
4. function var opt float: tanh(var opt float: x)
```

1, 2.
Return tanh **x**

3, 4.
Return tanh **x** if x occurs, otherwise return absent

Coercions

These functions implement coercions, or channeling, between different types.

```
1. function float: bool2float(bool: b)
2. function opt float: bool2float(opt bool: b)
3. function array [$T] of float: bool2float(array [$T] of bool: x)
4. function array [$T] of var float: bool2float(array [$T] of var bool: x)
5. function var float: bool2float(var bool: b)
6. function var opt float: bool2float(var opt bool: x)
7. function array [$T] of opt float: bool2float(array [$T] of opt bool: x)
8. function array [$T] of var opt float: bool2float(array [$T] of var opt bool: x)
```

1, 2.
Return Boolean b coerced to a float

3, 4.
Return array of Booleans x coerced to an array of floats

5. Return Boolean b coerced to a float

6. Return optional 0/1 float that is absent iff x is absent, and 1 iff x occurs and is true.

7, 8.
Return array of Booleans x coerced to an array of floats

```
1.  function int: bool2int(bool: b)
2.  function opt int: bool2int(opt bool: b)
3.  function var opt int: bool2int(var opt bool: x)
4.  function var int: bool2int(var bool: b)
5.  function array [$T] of var int: bool2int(array [$T] of var bool: b)
6.  function array [$T] of int: bool2int(array [$T] of bool: x)
7.  function array [$T] of set of int: bool2int(array [$T] of set of bool: x)
8.  function array [$T] of var int: bool2int(array [$T] of var bool: x)
9.  function array [$T] of opt int: bool2int(array [$T] of opt bool: x)
10. function array [$T] of var opt int: bool2int(array [$T] of var opt bool: x)
```

1, 2.
Return Boolean b coerced to an integer

3. Return optional 0/1 integer that is absent iff x is absent, and 1 iff x occurs and is true.
4. Return Boolean b coerced to an integer
5. Return array of Booleans b coerced to an array of integers
6. Return array of Booleans x coerced to an array of integers
7. Return array of sets of Booleans x coerced to an array of sets of integers

8-10.
Return array of Booleans x coerced to an array of integers

```
function int: ceil(float: x)
function var int: ceil(var float: x)
```

Return $\lceil \mathbf{x} \rceil$

```
function int: enum2int($$E: x)
function var int: enum2int(var $$E: x)
function opt int: enum2int(opt $$E: x)
function var opt int: enum2int(var opt $$E: x)
function opt set of int: enum2int(opt set of $$E: x)
```

```
function set of int: enum2int(set of $$E: x)
function var set of int: enum2int(var set of $$E: x)
function array [$X] of int: enum2int(array [$X] of $$E: x)
function array [$X] of var int: enum2int(array [$X] of var $$E: x)
function array [$X] of opt int: enum2int(array [$X] of opt $$E: x)
function array [$X] of var opt int: enum2int(array [$X] of var opt $$E: x)
function array [$X] of opt set of int: enum2int(array [$X] of opt set of $$E: x)
function array [$X] of set of int: enum2int(array [$X] of set of $$E: x)
function array [$X] of var set of int: enum2int(array [$X] of var set of $$E: x)
```

Return the underlying integer representation of x

```
function int: floor(float: x)
function var int: floor(var float: x)
```

Return $\lfloor \mathbf{x} \rfloor$

```
1.  function array [int] of any $T: index2int(array [$$E] of any $T: x)

2.  function array [int,int] of any $T: index2int(array [$$E,$$F] of any $T: x)

3.  function array [int,int,int] of any $T: index2int(array [$$E,$$F,$$G] of any
↪$T: x)

4.  function array [int,int,int,int] of any $T: index2int(array [$$E,$$F,$$G,$$H]␣
↪of any $T: x)

5.  function array [int,int,int,int,int] of any $T: index2int(array [$$E,$$F,$$G,$
↪$H,$$I] of any $T: x)

6.  function array [int,int,int,int,int,int] of any $T: index2int(array [$$E,$$F,$
↪$G,$$H,$$I,$$J] of any $T: x)
```

1. Return x with its index set coerced to integers

2-6.
Return x with its index sets coerced to integers

```
1.  function float: int2float(int: x)

2.  function opt float: int2float(opt int: x)

3.  function var float: int2float(var int: x)

4.  function var opt float: int2float(var opt int: x)

5.  function array [$T] of float: int2float(array [$T] of int: x)

6.  function array [$T] of opt float: int2float(array [$T] of opt int: x)
```

```
7.  function array [$T] of var float: int2float(array [$T] of var int: x)

8.  function array [$T] of var opt float: int2float(array [$T] of var opt int: x)
```

1-3.
Return integer x coerced to a float

4. Return optional 0/1 integer that is absent iff x is absent, and 1 iff x occurs and is true.

5-7.
Return array of integers x coerced to an array of floats

8. Return array of optional integers x coerced to an array of optional floats

---

```
function int: round(float: x)
function var int: round(var 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

Array operations

These functions implement the basic operations on arrays.

Functions and Predicates

```
1.  function array [int] of any $T: '++'(array [$$X] of any $T: x,
                                         array [$$Y] of any $T: y)

2.  function array [int,$$E] of any $T: '++'(array [$$X,$$E] of any $T: x,
                                             array [$$Y,$$E] of any $T: y)

3.  function array [int,$$E,$$F] of any $T: '++'(array [$$X,$$E,$$F] of any $T: x,
                                                 array [$$Y,$$E,$$F] of any $T: y)

4.  function array [int,$$E,$$F,$$G] of any $T: '++'(array [$$X,$$E,$$F,$$G] of any
↪$T: x,
                                                     array [$$Y,$$E,$$F,$$G] of any
↪$T: y)

5.  function array [int,$$E,$$F,$$G,$$H] of any $T: '++'(array [$$X,$$E,$$F,$$G,$
↪$H] of any $T: x,
                                                         array [$$Y,$$E,$$F,$$G,$
```

```
↪$H] of any $T: y)

6.  function array [int,$$E,$$F,$$G,$$H,$$I] of any $T: '++'(array [$$X,$$E,$$F,$$G,
↪$$H,$$I] of any $T: x,
                                                                array [$$Y,$$E,$$F,$$G,
↪$$H,$$I] of any $T: y)
```

1. Return the concatenation of arrays `x` and `y`

   Usage: `x ++ y`

2-6.

Return the concatenation of the multidimensional arrays `x` and `y`

The index sets of the arrays must match except for the first index set.

Usage: `x ++ y`

```
test 'in'($X: x, array [$T] of $X: y)
test 'in'(opt $X: x, array [$T] of opt $X: y)
predicate 'in'(var $X: x, array [$T] of var $X: y)
predicate 'in'(var opt $X: x, array [$T] of var opt $X: y)
test 'in'(set of $X: x, array [$T] of set of $X: y)
predicate 'in'(var set of $$X: x, array [$T] of var set of $$X: y)
```

Return if `y` contains `x`

Usage: `x in y`

```
1.  function array [int] of any $V: array1d(array [$U] of any $V: x)

2.  function array [$$E] of any $V: array1d(set of $$E: S,
                                            array [$U] of any $V: x)
```

1. Return array `x` coerced to index set 1..length(`x`). Coercions are performed by considering the array `x` in row-major order.
2. 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 any $V: array2d(set of $$E: S1,
                                            set of $$F: S2,
                                            array [$U] of any $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.

```
1.  function var set of $$T: array2set(array [$X] of var $$T: x)

2.  function var set of $$T: array2set(array [$X] of var opt $$T: x)

3.  function set of $$T: array2set(array [int] of $$T: x)

4.  function set of $$T: array2set(array [int] of opt $$T: x)

5.  function set of bool: array2set(array [int] of bool: x)

6.  function set of float: array2set(array [int] of float: x)
```

1. Return the set containing the elements of x
2. Return the set containing the non-absent elements of x
3. Return the set containing the elements of x
4. Return the set containing the non-absent elements of x

5, 6.
Return the set containing the elements of x

```
function array [$$E,$$F,$$G] of any $V: array3d(set of $$E: S1,
                                               set of $$F: S2,
                                               set of $$G: S3,
                                               array [$U] of any $V: x)
```

Return array x coerced to three-dimensional array with index sets S1, S2 and S3. Coercions are performed by considering the array x in row-major order.

```
function array [$$E,$$F,$$G,$$H] of any $V: array4d(set of $$E: S1,
                                                    set of $$F: S2,
                                                    set of $$G: S3,
                                                    set of $$H: S4,
                                                    array [$U] of any $V: x)
```

Return array x coerced to 4-dimensional array with index sets S1, S2, S3 and S4. Coercions are performed by considering the array x in row-major order.

```
function array [$$E,$$F,$$G,$$H,$$I] of any $V: array5d(set of $$E: S1,
                                                        set of $$F: S2,
                                                        set of $$G: S3,
                                                        set of $$H: S4,
                                                        set of $$I: S5,
                                                        array [$U] of any $V: x)
```

Return array x coerced to 5-dimensional array with index sets S1, S2, S3, S4 and S5. Coercions are performed by considering the array x in row-major order.

```
function array [$$E,$$F,$$G,$$H,$$I,$$J] of any $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 any $V: x)
```

Return array x coerced to 6-dimensional array with index sets S1, S2, S3, S4, S5 and S6. Coercions are performed by considering the array x in row-major order.

```
function array [$T] of any $V: arrayXd(array [$T] of any $X: x,
                                       array [$U] of any $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 [$$E] of any $T: col(array [$$E,$$F] of any $T: x,
                                    $$F: c)
```

Return column c of array x

```
function array [int] of $T: concat_index_1(array [$$X11] of any $T: x,
                                           array [$$X12] of any $T: y)
function array [int,$$X2] of $T: concat_index_1(array [$$X11,$$X2] of any $T: x,
                                                array [$$X12,$$X2] of any $T: y)
function array [int,$$X2,$$X3] of $T: concat_index_1(array [$$X11,$$X2,$$X3] of any
↪$T: x,
                                                     array [$$X12,$$X2,$$X3] of any
↪$T: y)
function array [int,$$X2,$$X3,$$X4] of $T: concat_index_1(array [$$X11,$$X2,$$X3,$
↪$X4] of any $T: x,
                                                          array [$$X12,$$X2,$$X3,$
↪$X4] of any $T: y)
function array [int,$$X2,$$X3,$$X4,$$X5] of $T: concat_index_1(array [$$X11,$$X2,$
↪$X3,$$X4,$$X5] of any $T: x,
                                                               array [$$X12,$$X2,$
↪$X3,$$X4,$$X5] of any $T: y)
function array [int,$$X2,$$X3,$$X4,$$X5,$$X6] of $T: concat_index_1(array [$$X11,$
↪$X2,$$X3,$$X4,$$X5,$$X6] of any $T: x,
                                                                    array [$$X12,$
↪$X2,$$X3,$$X4,$$X5,$$X6] of any $T: y)
```

Return the concatenation of the multidimensional arrays x and y in the first index.

```
function array [$$X1,int] of $T: concat_index_2(array [$$X1,$$X21] of any $T: x,
                                                array [$$X1,$$X22] of any $T: y)
function array [$$X1,int,$$X3] of $T: concat_index_2(array [$$X1,$$X21,$$X3] of any
↪$T: x,
                                                     array [$$X1,$$X22,$$X3] of any
↪$T: y)
function array [$$X1,int,$$X3,$$X4] of $T: concat_index_2(array [$$X1,$$X21,$$X3,$
↪$X4] of any $T: x,
                                                          array [$$X1,$$X22,$$X3,$
↪$X4] of any $T: y)
function array [$$X1,int,$$X3,$$X4,$$X5] of $T: concat_index_2(array [$$X1,$$X21,$
↪$X3,$$X4,$$X5] of any $T: x,
                                                               array [$$X1,$$X22,$
↪$X3,$$X4,$$X5] of any $T: y)
function array [$$X1,int,$$X3,$$X4,$$X5,$$X6] of $T: concat_index_2(array [$$X1,$
↪$X21,$$X3,$$X4,$$X5,$$X6] of any $T: x,
                                                                    array [$$X1,$
↪$X22,$$X3,$$X4,$$X5,$$X6] of any $T: y)
```

Return the concatenation of the multidimensional arrays `x` and `y` in the second index.

```
function array [$$X1,$$X2,int] of $T: concat_index_3(array [$$X1,$$X2,$$X31] of any
↪$T: x,
                                                     array [$$X1,$$X2,$$X32] of any
↪$T: y)
function array [$$X1,$$X2,int,$$X4] of $T: concat_index_3(array [$$X1,$$X2,$$X31,$
↪$X4] of any $T: x,
                                                          array [$$X1,$$X2,$$X32,$
↪$X4] of any $T: y)
function array [$$X1,$$X2,int,$$X4,$$X5] of $T: concat_index_3(array [$$X1,$$X2,$
↪$X31,$$X4,$$X5] of any $T: x,
                                                               array [$$X1,$$X2,$
↪$X32,$$X4,$$X5] of any $T: y)
function array [$$X1,$$X2,int,$$X4,$$X5,$$X6] of $T: concat_index_3(array [$$X1,$
↪$X2,$$X31,$$X4,$$X5,$$X6] of any $T: x,
                                                                    array [$$X1,$
↪$X2,$$X32,$$X4,$$X5,$$X6] of any $T: y)
```

Return the concatenation of the multidimensional arrays `x` and `y` in the third index.

```
function array [$$X1,$$X2,$$X3,int] of $T: concat_index_4(array [$$X1,$$X2,$$X3,$
↪$X41] of any $T: x,
                                                          array [$$X1,$$X2,$$X3,$
↪$X42] of any $T: y)
function array [$$X1,$$X2,$$X3,int,$$X5] of $T: concat_index_4(array [$$X1,$$X2,$
↪$X3,$$X41,$$X5] of any $T: x,
                                                               array [$$X1,$$X2,$
↪$X3,$$X42,$$X5] of any $T: y)
function array [$$X1,$$X2,$$X3,int,$$X5,$$X6] of $T: concat_index_4(array [$$X1,$
↪$X2,$$X3,$$X41,$$X5,$$X6] of any $T: x,
                                                                    array [$$X1,$
↪$X2,$$X3,$$X42,$$X5,$$X6] of any $T: y)
```

Return the concatenation of the multidimensional arrays x and y in the fourth index.

```
function array [$$X1,$$X2,$$X3,$$X4,int] of $T: concat_index_5(array [$$X1,$$X2,$
↪$X3,$$X4,$$X51] of any $T: x,
                                                                array [$$X1,$$X2,$
↪$X3,$$X4,$$X52] of any $T: y)
function array [$$X1,$$X2,$$X3,$$X4,int,$$X6] of $T: concat_index_5(array [$$X1,$
↪$X2,$$X3,$$X4,$$X51,$$X6] of any $T: x,
                                                                    array [$$X1,$
↪$X2,$$X3,$$X4,$$X52,$$X6] of any $T: y)
```

Return the concatenation of the multidimensional arrays x and y in the fifth index.

```
function array [$$X1,$$X2,$$X3,$$X4,$$X5,int] of $T: concat_index_6(array [$$X1,$
↪$X2,$$X3,$$X4,$$X5,$$X61] of any $T: x,
                                                                    array [$$X1,$
↪$X2,$$X3,$$X4,$$X5,$$X61] of any $T: y)
```

Return the concatenation of the multidimensional arrays x and y in the sixth index.

```
test has_element($T: e, array [int] of $T: x)
test has_element($T: e, array [int] of opt $T: x)
predicate has_element($T: e, array [$$E] of any $T: x)
```

Test if e is an element of array x

```
test has_index($$E: i, array [$$E] of any $T: x)
```

Test if i is in the index set of x

```
function set of $$E: index_set(array [$$E] of any $U: x)
```

Return index set of one-dimensional array x

```
function set of $$E: index_set_1of2(array [$$E,$$F] of any $U: x)
```

Return index set of first dimension of two-dimensional array x

```
function set of $$E: index_set_1of3(array [$$E,$$F,$$G] of any $U: x)
```

Return index set of first dimension of 3-dimensional array x

```
function set of $$E: index_set_1of4(array [$$E,$$F,$$G,$$H] of any $U: x)
```

Return index set of first dimension of 4-dimensional array x

```
function set of $$E: index_set_1of5(array [$$E,$$F,$$G,$$H,$$I] of any $U: x)
```

Return index set of first dimension of 5-dimensional array x

```
function set of $$E: index_set_1of6(array [$$E,$$F,$$G,$$H,$$I,$$J] of any $U: x)
```

Return index set of first dimension of 6-dimensional array x

```
function set of $$F: index_set_2of2(array [$$E,$$F] of any $U: x)
```

Return index set of second dimension of two-dimensional array x

```
function set of $$F: index_set_2of3(array [$$E,$$F,$$G] of any $U: x)
```

Return index set of second dimension of 3-dimensional array x

```
function set of $$F: index_set_2of4(array [$$E,$$F,$$G,$$H] of any $U: x)
```

Return index set of second dimension of 4-dimensional array x

```
function set of $$F: index_set_2of5(array [$$E,$$F,$$G,$$H,$$I] of any $U: x)
```

Return index set of second dimension of 5-dimensional array x

```
function set of $$F: index_set_2of6(array [$$E,$$F,$$G,$$H,$$I,$$J] of any $U: x)
```

Return index set of second dimension of 6-dimensional array x

```
function set of $$G: index_set_3of3(array [$$E,$$F,$$G] of any $U: x)
```

Return index set of third dimension of 3-dimensional array x

```
function set of $$G: index_set_3of4(array [$$E,$$F,$$G,$$H] of any $U: x)
```

Return index set of third dimension of 4-dimensional array x

```
function set of $$G: index_set_3of5(array [$$E,$$F,$$G,$$H,$$I] of any $U: x)
```

Return index set of third dimension of 5-dimensional array x

```
function set of $$G: index_set_3of6(array [$$E,$$F,$$G,$$H,$$I,$$J] of any $U: x)
```

Return index set of third dimension of 6-dimensional array x

```
function set of $$H: index_set_4of4(array [$$E,$$F,$$G,$$H] of any $U: x)
```

Return index set of fourth dimension of 4-dimensional array x

```
function set of $$H: index_set_4of5(array [$$E,$$F,$$G,$$H,$$I] of any $U: x)
```

Return index set of fourth dimension of 5-dimensional array x

```
function set of $$H: index_set_4of6(array [$$E,$$F,$$G,$$H,$$I,$$J] of any $U: x)
```

Return index set of fourth dimension of 6-dimensional array x

```
function set of $$I: index_set_5of5(array [$$E,$$F,$$G,$$H,$$I] of any $U: x)
```

Return index set of fifth dimension of 5-dimensional array x

```
function set of $$I: index_set_5of6(array [$$E,$$F,$$G,$$H,$$I,$$J] of any $U: x)
```

Return index set of fifth dimension of 6-dimensional array x

```
function set of $$J: index_set_6of6(array [$$E,$$F,$$G,$$H,$$I,$$J] of any $U: x)
```

Return index set of sixth dimension of 6-dimensional array x

```
test index_sets_agree(array [$T] of any $U: x, array [$T] of any $W: y)
```

Test if x and y have the same index sets

```
function int: length(array [$T] of any $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 [$$E] of any $T: reverse(array [$$E] of any $T: x)
```

Return the array x in reverse order

The resulting array has the same index set as x.

```
function array [$$E] of any $T: row(array [$$F,$$E] of any $T: x,
                                    $$F: r)
```

Return row r of array x

```
function array [int] of $T: select_from_index_1(array [$$X1] of any $T: x,
                                                set of $$X1: S)
function array [int,$$X2] of $T: select_from_index_1(array [$$X1,$$X2] of any $T: x,
                                                     set of $$X1: S)
function array [int,$$X2,$$X3] of $T: select_from_index_1(array [$$X1,$$X2,$$X3] of␣
↪any $T: x,
                                                          set of $$X1: S)
function array [int,$$X2,$$X3,$$X4] of $T: select_from_index_1(array [$$X1,$$X2,$
↪$X3,$$X4] of any $T: x,
                                                               set of $$X1: S)
function array [int,$$X2,$$X3,$$X4,$$X5] of $T: select_from_index_1(array [$$X1,$
↪$X2,$$X3,$$X4,$$X5] of any $T: x,
                                                                    set of $$X1: S)
function array [int,$$X2,$$X3,$$X4,$$X5,$$X6] of $T: select_from_index_1(array [$
↪$X1,$$X2,$$X3,$$X4,$$X5,$$X6] of any $T: x,
                                                                         set of $
↪$X1: S)
```

Return a (potentially non-contiguous) slice of array x, in the first index, specified by the set S. The result is undefined if S is not a subset of the first index set of x.

```
function array [$$X1,int] of $T: select_from_index_2(array [$$X1,$$X2] of any $T: x,
                                                    set of $$X2: S)
function array [$$X1,int,$$X3] of $T: select_from_index_2(array [$$X1,$$X2,$$X3] of␣
↪any $T: x,
                                                          set of $$X2: S)
function array [$$X1,int,$$X3,$$X4] of $T: select_from_index_2(array [$$X1,$$X2,$
↪$X3,$$X4] of any $T: x,
                                                                set of $$X2: S)
function array [$$X1,int,$$X3,$$X4,$$X5] of $T: select_from_index_2(array [$$X1,$
↪$X2,$$X3,$$X4,$$X5] of any $T: x,
                                                                      set of $$X2: S)
function array [$$X1,int,$$X3,$$X4,$$X5,$$X6] of $T: select_from_index_2(array [$
↪$X1,$$X2,$$X3,$$X4,$$X5,$$X6] of any $T: x,
                                                                            set of $
↪$X2: S)
```

Return a (potentially non-contiguous) slice of array `x`, in the second index, specified by the set `S`. The result is undefined if `S` is not a subset of the second index set of `x`.

```
function array [$$X1,$$X2,int] of $T: select_from_index_3(array [$$X1,$$X2,$$X3] of␣
↪any $T: x,
                                                          set of $$X3: S)
function array [$$X1,$$X2,int,$$X4] of $T: select_from_index_3(array [$$X1,$$X2,$
↪$X3,$$X4] of any $T: x,
                                                                set of $$X3: S)
function array [$$X1,$$X2,int,$$X4,$$X5] of $T: select_from_index_3(array [$$X1,$
↪$X2,$$X3,$$X4,$$X5] of any $T: x,
                                                                      set of $$X3: S)
function array [$$X1,$$X2,int,$$X4,$$X5,$$X6] of $T: select_from_index_3(array [$
↪$X1,$$X2,$$X3,$$X4,$$X5,$$X6] of any $T: x,
                                                                            set of $
↪$X3: S)
```

Return a (potentially non-contiguous) slice of array `x`, in the third index, specified by the set `S`. The result is undefined if `S` is not a subset of the third index set of `x`.

```
function array [$$X1,$$X2,$$X3,int] of $T: select_from_index_4(array [$$X1,$$X2,$
↪$X3,$$X4] of any $T: x,
                                                                set of $$X4: S)
function array [$$X1,$$X2,$$X3,int,$$X5] of $T: select_from_index_4(array [$$X1,$
↪$X2,$$X3,$$X4,$$X5] of any $T: x,
                                                                      set of $$X4: S)
function array [$$X1,$$X2,$$X3,int,$$X5,$$X6] of $T: select_from_index_4(array [$
↪$X1,$$X2,$$X3,$$X4,$$X5,$$X6] of any $T: x,
                                                                            set of $
↪$X4: S)
```

Return a (potentially non-contiguous) slice of array `x`, in the fourth index, specified by the set `S`. The result is undefined if `S` is not a subset of the fourth index set of `x`.

```
function array [$$X1,$$X2,$$X3,$$X4,int] of $T: select_from_index_5(array [$$X1,$
↪$X2,$$X3,$$X4,$$X5] of any $T: x,
                                                                    set of $$X5: S)
function array [$$X1,$$X2,$$X3,$$X4,int,$$X6] of $T: select_from_index_5(array [$
↪$X1,$$X2,$$X3,$$X4,$$X5,$$X6] of any $T: x,
                                                                        set of $
↪$X5: S)
```

Return a (potentially non-contiguous) slice of array `x`, in the fifth index, specified by the set `S`. The result is undefined if `S` is not a subset of the fifth index set of `x`.

```
function array [$$X1,$$X2,$$X3,$$X4,$$X5,int] of $T: select_from_index_6(array [$
↪$X1,$$X2,$$X3,$$X4,$$X5,$$X6] of any $T: x,
                                                                        set of $
↪$X6: S)
```

Return a (potentially non-contiguous) slice of array `x`, in the sixth index, specified by the set `S`. The result is undefined if `S` is not a subset of the sixth index set of `x`.

```
function array [$$F] of any $T: slice_1d(array [$E] of any $T: x,
                                         array [int] of set of int: s,
                                         set of $$F: dims1)
```

Return slice of array `x` specified by sets `s`, coerced to new 1d array with index set `dims1`

```
function array [$$F,$$G] of any $T: slice_2d(array [$E] of any $T: x,
                                             array [int] of set of int: s,
                                             set of $$F: dims1,
                                             set of $$G: dims2)
```

Return slice of array `x` specified by sets `s`, coerced to new 2d array with index sets `dims1` and `dims2`

```
function array [$$F,$$G,$$H] of any $T: slice_3d(array [$E] of any $T: x,
                                                 array [int] of set of int: s,
                                                 set of $$F: dims1,
                                                 set of $$G: dims2,
                                                 set of $$H: dims3)
```

Return slice of array `x` specified by sets `s`, coerced to new 3d array with index sets `dims1`, `dims2` and `dims3`

```
function array [$$F,$$G,$$H,$$I] of any $T: slice_4d(array [$E] of any $T: x,
                                                     array [int] of set of int: s,
                                                     set of $$F: dims1,
                                                     set of $$G: dims2,
                                                     set of $$H: dims3,
                                                     set of $$I: dims4)
```

Return slice of array `x` specified by sets `s`, coerced to new 4d array with index sets `dims1`, `dims2`, `dims3`, `dims4`

```
function array [$$F,$$G,$$H,$$I,$$J] of any $T: slice_5d(array [$E] of any $T: x,
                                                         array [int] of set of int:␣
↪s,
                                                         set of $$F: dims1,
                                                         set of $$G: dims2,
                                                         set of $$H: dims3,
                                                         set of $$I: dims4,
                                                         set of $$J: dims5)
```

Return slice of array `x` specified by sets `s`, coerced to new 5d array with index sets `dims1`, `dims2`, `dims3`, `dims4`, `dims5`

```
function array [$$F,$$G,$$H,$$I,$$J,$$K] of any $T: slice_6d(array [$E] of any $T:␣
↪x,
                                                             array [int] of set of␣
↪int: s,
                                                             set of $$F: dims1,
                                                             set of $$G: dims2,
                                                             set of $$H: dims3,
                                                             set of $$I: dims4,
                                                             set of $$J: dims5,
                                                             set of $$K: dims6)
```

Return slice of array `x` specified by sets `s`, coerced to new 6d array with index sets `dims1`, `dims2`, `dims3`, `dims4`, `dims5`, `dims6`

```
function tuple(array [$$E] of any $T, array [$$E] of any $U): unzip(array [$$E] of␣
↪tuple($T, $U): x)
```

Split array `x` of pairs into a pair of arrays.

The resulting arrays keep the index set of `x`.

```
function array [$$E] of tuple(any $T, any $U): zip(array [$$E] of any $T: x,
                                                   array [$$E] of any $U: y)
```

Pair elements of arrays `x` and `y` into tuples.

The arrays must have the same index set; the result has the same index set.

Annotations

```
annotation array_check_form
```

Forces a arrayNd call to throw an error if the new index sets are offsets of the current index sets.

Logical operations

Logical operations are the standard operators of Boolean logic.

```
test '->'(bool: x, bool: y)
test '->'(opt bool: x, opt bool: y)
predicate '->'(var bool: x, var bool: y)
predicate '->'(var opt bool: x, var opt bool: y)
```

Return truth value of `x` implies `y`

Usage: `x -> y`

```
1.  test '/\'(bool: x, bool: y)

2.  test '/\'(opt bool: x, opt bool: y)

3.  predicate '/\'(var bool: x, var bool: y)

4.  predicate '/\'(var opt bool: x, var opt bool: y)
```

1. Return truth value of `x` ∧ `y`

   Usage: `x /\ y`
2. Return truth value of `x` ∧ `y` with identity <> = True

   Usage: `x /\ y`
3. Return truth value of `x` ∧ `y`

   Usage: `x /\ y`
4. Return truth value of `x` ∧ `y` with identity <> = True

   Usage: `x /\ y`

```
test '<-'(bool: x, bool: y)
test '<-'(opt bool: x, opt bool: y)
predicate '<-'(var bool: x, var bool: y)
predicate '<-'(var opt bool: x, var opt bool: y)
```

Return truth value of `y` implies `x`

Usage: `x <- y`

```
test '<->'(bool: x, bool: y)
test '<->'(opt bool: x, opt bool: y)
predicate '<->'(var bool: x, var bool: y)
predicate '<->'(var opt bool: x, var opt bool: y)
```

Return truth value of `x` if-and-only-if `y`

Usage: `x <-> y`

```
1.  test '\/'(bool: x, bool: y)

2.  test '\/'(opt bool: x, opt bool: y)

3.  predicate '\/'(var bool: x, var bool: y)

4.  predicate '\/'(var opt bool: x, var opt bool: y)
```

1. Return truth value of `x` ∨ `y`

   Usage: `x \/ y`
2. Return truth value of `x` ∨ `y` with identity <> = False

   Usage: `x \/ y`
3. Return truth value of `x` ∨ `y`

   Usage: `x \/ y`
4. Return truth value of `x` ∨ `y` with identity <> = False

   Usage: `x \/ y`

```
1.  test 'not'(bool: x)

2.  function opt bool: 'not'(opt bool: x)

3.  predicate 'not'(var bool: x)

4.  function var opt bool: 'not'(var opt bool: x)
```

1. Return truth value of the negation of `x`

Usage: `not x`

2. Negation of `x` if it occurs, otherwise absent

   Usage: `not x`

3. Return truth value of the negation of `x`

   Usage: `not x`

4. Negation of `x` if it occurs, otherwise absent

   Usage: `not x`

```
test 'xor'(bool: x, bool: y)
test 'xor'(opt bool: x, opt bool: y)
predicate 'xor'(var bool: x, var bool: y)
predicate 'xor'(var opt bool: x, var opt bool: y)
```

Return truth value of `x` xor `y`

Usage: `x xor y`

```
predicate bool_not(var bool: b)
```

Return negation of `b`

```
test clause(array [$T] of bool: x, array [$T] of bool: y)
test clause(array [$T] of opt bool: x, array [$T] of opt bool: y)
predicate clause(array [$T] of var bool: x, array [$T] of var bool: y)
predicate clause(array [$T] of var opt bool: x,
                 array [$T] of var opt bool: y)
```

Return truth value of $(\bigvee_i \mathbf{x}[i]) \vee (\bigvee_j \neg \mathbf{y}[j])$

```
1.  test exists(array [$T] of bool: x)

2.  predicate exists(array [$T] of var bool: x)

3.  predicate exists(array [int] of var opt bool: x)
```

1, 2.

Return truth value of $\bigvee_i \mathbf{x}[i]$

3. True iff for at least one `i`, `x`[i] occurs and is true

```
1.  test forall(array [$T] of bool: x)

2.  predicate forall(array [$T] of var bool: x)

3.  predicate forall(array [int] of var opt bool: x)
```

1, 2.
Return truth value of $\bigwedge_i \mathbf{x}[i]$

3. True iff for any `i`, `x`[i] is absent or true

```
test iffall(array [$T] of bool: x)
test iffall(array [$T] of opt bool: x)
predicate iffall(array [$T] of var bool: x)
predicate iffall(array [$T] of var opt bool: x)
```

Return truth value of true $\oplus$ $(\oplus_i \mathbf{x}[i])$

```
test xorall(array [$T] of bool: x)
test xorall(array [$T] of opt bool: x)
predicate xorall(array [$T] of var bool: x)
predicate xorall(array [$T] of var opt bool: x)
```

Return truth value of $\oplus_i \mathbf{x}[i]$

Set operations

These functions implement the basic operations on sets.

```
function set of $$E: '..'($$E: a, $$E: b)
function set of float: '..'(float: a, float: b)
function set of bool: '..'(bool: a, bool: b)
```

Return the set $\{\mathbf{a}, \ldots, \mathbf{b}\}$

Usage: `a .. b`

```
function set of $$E: '..<'($$E: a, $$E: b)
```

Return the set $\{\mathbf{a}, \ldots, \mathbf{b} - 1\}$

Usage: `a ..< b`

```
function set of $$E: '<..'($$E: a, $$E: b)
```

Return the set $\{\mathbf{a}+1,\ldots,\mathbf{b}\}$

Usage: `a <.. b`

```
function set of $$E: '<..<'($$E: a, $$E: b)
```

Return the set $\{\mathbf{a}+1,\ldots,\mathbf{b}-1\}$

Usage: `a <..< b`

```
function set of $T: 'diff'(set of $T: x, set of $T: y)
function var set of $$T: 'diff'(var set of $$T: x, var set of $$T: y)
```

Return the set difference of sets $\mathbf{x} \setminus \mathbf{y}$

Usage: `x diff y`

```
1.  test 'in'($$E: x, set of $$E: y)
2.  predicate 'in'(var $$E: x, set of $$E: y)
3.  predicate 'in'(var $$E: x, var set of $$E: y)
4.  predicate 'in'(var opt $$E: x, set of $$E: y)
5.  test 'in'(opt $$E: x, set of $$E: y)
6.  predicate 'in'(var opt $$E: x, var set of $$E: y)
7.  test 'in'(float: x, set of float: y)
8.  predicate 'in'(var float: x, set of float: y)
9.  test 'in'(opt float: x, set of float: y)
10. predicate 'in'(var opt float: x, set of float: y)
```

1. Test if `x` is an element of the set `y`

   Usage: `x in y`

2, 3.

   `x` is an element of the set `y`

   Usage: `x in y`

4. `x` is an element of the set `y`. False if `x` is absent.

   Usage: `x in y`

5. Test if `x` is an element of the set `y`. False if `x` is absent.

   Usage: `x in y`

6. x is an element of the set y. False if x is absent.

   Usage: `x in y`

7, 8.

   Test if x is an element of the set y

   Usage: `x in y`

9. Test if x is an element of the set y. False if x is absent.

   Usage: `x in y`

10. x is an element of the set y. False if x is absent.

    Usage: `x in y`

```
function set of $T: 'intersect'(set of $T: x, set of $T: y)
function var set of $$T: 'intersect'(var set of $$T: x,
                                     var set of $$T: y)
```

Return the intersection of sets x and y

Usage: `x intersect y`

```
1.  test 'subset'(set of $T: x, set of $T: y)

2.  predicate 'subset'(var set of $$E: x, var set of $$E: y)
```

1. Test if x is a subset of y

   Usage: `x subset y`

2. x is a subset of y

   Usage: `x subset y`

```
1.  test 'superset'(set of $T: x, set of $T: y)

2.  predicate 'superset'(var set of $$E: x, var set of $$E: y)
```

1. Test if x is a superset of y

   Usage: `x superset y`

2. x is a superset of y

   Usage: `x superset y`

```
function set of $T: 'symdiff'(set of $T: x, set of $T: y)
function var set of $$T: 'symdiff'(var set of $$T: x,
                                   var set of $$T: y)
```

Return the symmetric set difference of sets `x` and `y`

Usage: `x symdiff y`

```
function set of $T: 'union'(set of $T: x, set of $T: y)
function var set of $$T: 'union'(var set of $$T: x, var set of $$T: y)
```

Return the union of sets `x` and `y`

Usage: `x union y`

```
function set of $$E: '..<o'($$E: a)
```

Return the set $\{\mathbf{a}, \ldots, \mathbf{b} - 1\}$

```
function set of $$E: '..o'($$E: a)
```

Return the set $\{\mathbf{a}, \ldots, \mathbf{b} - 1\}$

```
function set of $$E: '<..<o'($$E: a)
```

Return the set $\{\mathbf{a} + 1, \ldots, \mathbf{b} - 1\}$

```
function set of $$E: '<..o'($$E: a)
```

Return the set $\{\mathbf{a} + 1, \ldots, \mathbf{b}\}$

```
1.  function set of $U: array_intersect(array [$T] of set of $U: x)

2.  function var set of $$E: array_intersect(array [$T] of var set of $$E: x)
```

1. Return the intersection of the sets in array `x`
2. Return the intersection of the sets in array `x`. Undefined for empty `x`.

```
function set of $U: array_union(array [$T] of set of $U: x)
function var set of $$E: array_union(array [$T] of var set of $$E: x)
```

Return the union of the sets in array `x`

```
function int: card(set of $T: x)
function var int: card(var set of $$E: 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`

```
function set of $$E: 'o..'($$E: a)
```

Return the set $\{\mathbf{a}, \ldots, \mathbf{b} - 1\}$

```
function set of $$E: 'o..<'($$E: a)
```

Return the set $\{\mathbf{a}, \ldots, \mathbf{b} - 1\}$

```
function set of $$E: 'o<..'($$E: a)
```

Return the set $\{\mathbf{a} + 1, \ldots, \mathbf{b}\}$

```
function set of $$E: 'o<..<'($$E: a)
```

Return the set $\{\mathbf{a} + 1, \ldots, \mathbf{b} - 1\}$

```
function array [int] of $$E: set_to_ranges(set of $$E: S)
function array [int] of float: set_to_ranges(set of float: S)
function array [int] of bool: set_to_ranges(set of bool: S)
```

Return a sorted list of the non-overlapping ranges in `S`

String operations

These functions implement operations on strings.

```
function string: '++'(string: s1, string: s2)
```

Return concatenation of `s1` and `s2`

Usage: `s1 ++ 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

```
1.  function string: format(any $T: x)
2.  function string: format(array [$U] of any $T: x)
3.  function string: format(int: w, int: p, any $T: x)
4.  function string: format(int: w, int: p, array [$U] of any $T: x)
5.  function string: format(int: w, any $T: x)
6.  function string: format(int: w, array [$U] of any $T: x)
```

1, 2.
Convert `x` into a string

3. 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.

4. 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.

5, 6.

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.

```
1.  function string: format_justify_string(int: w, string: x)

2.  function string: format_justify_string(int: w, int: p, string: x)
```

1. String justification

   Returns the string `x` right justified by the number of characters given by `w`, or left justified if `w` is negative.

2. String justification

   Returns the string `x` right justified by the number of characters given by `w`, or left justified if `w` is negative.

   The maximum length of of `x` to be formatted is given by `p`. It is a run-time error for `p` to be negative.

```
function string: join(string: d, array [$T] of string: s)
```

Join string in array `s` using delimiter `d`

```
function string: json_array(array [int] of string: arr)
```

Create a JSON array from an array of JSON strings `arr`.

The elements are assumed to be strings already in JSON format (usually generated using showJSON)

```
function string: json_object(array [int,1..2] of string: obj)
```

Create a JSON object from an array of key-value pairs `obj`.

`obj[i, 1]` is the key name for the `i`-th entry `obj[i, 2]` is the JSON value for the o i-th entry (usually generated using showJSON)

```
1.  function array [int] of string: outputJSON()

2.  function array [int] of string: outputJSON(bool: b)
```

1. Return array for output of all variables in JSON format

2. Return array for output of all variables in JSON format, including objective if `b` is true

---

```
function array [int] of string: outputJSONParameters()
```

Return array for output of all parameters in JSON format

```
test output_to_json_section(string: s, any $T: o)
test output_to_json_section(string: s, array [$U] of any $T: o)
```

Output `o` to the JSON section `s`

```
test output_to_section(string: s, string: o)
```

Add `o` to the output section `s`

```
function string: show(any $T: x)
function string: show(array [$U] of any $T: x)
```

Convert `x` into a string

```
function string: show2d(array [$$E,$$F] of any $T: x)
```

Convert two-dimensional array `x` into a string

```
function string: show2d_indexed(array [int] of string: row_hdr,
                                array [int] of string: col_hdr,
                                array [int,int] of string: vs)
```

Convert two-dimensional array `vs` into a string with row and column headers `row_hdr` and `col_hdr`

```
function string: show3d(array [$$E,$$F,$$G] of any $T: x)
```

Convert three-dimensional array `x` into a string

```
function string: showJSON(any $T: x)
function string: showJSON(array [$U] of any $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

This function interprets the string as UTF-8 encoded. The length is the number of Unicode code points in the string.

```
function array [int] of string: string_split(string: s, string: sep)
```

Split s by separator sep

Returns an array of strings containing the substrings of s that are separated by sep. If sep is empty, the result is an array of characters of s.

This function interprets the string as UTF-8 encoded, so when splitting with an empty separator, the result is an array of Unicode code points.

Functions for enums

```
1. function $$E: enum_next($$E: x)

2. function opt $$E: enum_next(opt $$E: x)

3. function var $$E: enum_next(var $$E: x)
```

```
4.  function var opt $$E: enum_next(var opt $$E: x)

5.  function $$E: enum_next(set of $$E: e, $$E: x)

6.  function opt $$E: enum_next(set of $$E: e, opt $$E: x)

7.  function var $$E: enum_next(set of $$E: e, var $$E: x)

8.  function var opt $$E: enum_next(set of $$E: e, var opt $$E: x)
```

1-4.
Return next greater enum value of x

5-8.
Return next greater enum value of x in enum e

```
function set of $$E: enum_of(var opt $$E: x)
function set of $$E: enum_of(var set of $$E: x)
function set of $$E: enum_of(array [$T] of var opt $$E: x)
function set of $$E: enum_of(array [$T] of var set of $$E: x)
```

Return enum set of x

```
1.  function $$E: enum_prev($$E: x)

2.  function opt $$E: enum_prev(opt $$E: x)

3.  function var $$E: enum_prev(var $$E: x)

4.  function var opt $$E: enum_prev(var opt $$E: x)

5.  function $$E: enum_prev(set of $$E: e, $$E: x)

6.  function opt $$E: enum_prev(set of $$E: e, opt $$E: x)

7.  function var $$E: enum_prev(set of $$E: e, var $$E: x)

8.  function var opt $$E: enum_prev(set of $$E: e, var opt $$E: x)
```

1-4.
Return next smaller enum value of x

5-8.
Return next smaller enum value of x in enum e

```
function $$E: to_enum(set of $$E: X, int: x)
function opt $$E: to_enum(set of $$E: X, opt int: x)
function var $$E: to_enum(set of $$E: X, var int: x)
```

```
function var opt $$E: to_enum(set of $$E: X, var opt int: x)
function array [$U] of $$E: to_enum(set of $$E: X,
                                    array [$U] of int: x)
function array [$U] of opt $$E: to_enum(set of $$E: X,
                                        array [$U] of opt int: x)
function array [$U] of var $$E: to_enum(set of $$E: X,
                                        array [$U] of var int: x)
function array [$U] of var opt $$E: to_enum(set of $$E: X,
                                            array [$U] of var opt int: x)
function set of $$E: to_enum(set of $$E: X, set of int: x)
function array [$U] of set of $$E: to_enum(set of $$E: X,
                                           array [$U] of set of int: x)
function array [$U] of var set of $$E: to_enum(set of $$E: X,
                                               array [$U] of var set of int: x)
```

Convert `x` to enum type `X`

4.2.1.2 Annotations

These annotations control evaluation and solving behaviour.

General annotations

Functions and Predicates

```
annotation json_section(string: s)
```

Declare that the annotated output item should be output to the section `s` and formatted as JSON.

```
annotation mzn_output_section(string: s)
```

Used internally by the compiler to set the output section `s` for an output item. An output item can be annotated directly with a string literal to set its output section as in `output :: "my_section" ["hello, world\n"]` or using the syntax `output ["hello, world\n"] to "my_section`.

Annotations

```
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 annotated_expression
```

Mark annotation argument as annotated expression

```
annotation cache_result
```

Enables caching of the result of the annotated function declaration such that further calls with the same arguments yield the cached initial result.

Currently only supported for par functions with at least one parameter.

```
annotation computed_domain
```

Declare the annotated variable as having a domain computed for the convenience of the solver. The domain itself does not impose a constraint in the overall problem and can be ignored.

```
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

---

```
1.  annotation defines_var(var opt $T: c)

2.  annotation defines_var(array [$U] of var opt $T: arr)
```

1. Declare variable: `c` as being functionally defined by the annotated constraint. This annotation is introduced into FlatZinc code by the compiler.
2. Declare an array of variables `arr` 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 domain_change_constraint
```

Marks a constraint as a recorded domain changing constraint (when mzn2fzn called with -g flag

```
annotation empty_annotation
```

Empty annotation, will be removed during compilation

```
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 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_add_annotated_expression(int: idx)
```

Used internally by the compiler

```
annotation mzn_check_enum_var(array [int] of set of int: x)
```

Declare that the annotated variable is required for checking solutions and has an enum type `x`.

```
annotation mzn_check_var
```

Declare that the annotated variable is required for checking solutions.

```
annotation mzn_constraint_name(string: n)
```

Declare a name `n` for the annotated constraint.

```
annotation mzn_deprecated(string: version, string: explanation)
```

State that a function is deprecated since MiniZinc version `version` with humand readable `explanation`.

```
annotation mzn_expression_name(string: n)
```

Declare a name `n` 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 mzn_rhs_from_assignment
```

Used internally by the compiler

```
annotation no_cse
```

This annotation will prevent calls to the function annotated to be added to the Common Subexpression Elimination map.

WARNING: using this annotation might result in duplicated constraints when used incorrectly.

```
annotation no_output
```

Declare that the annotated variable should not be added to the output of the model. This annotation only has an effect when the model does not have an output item.

```
annotation 'output'(any $T: x)
annotation 'output'(array [$U] of any $T: x)
```

Declare that the annotated variable `x` 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 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.

```
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_commutative
```

Declare the function to be commutative, i.e., the function has the same result regardless of the order in which the function parameters are provided.

WARNING: a commutative function must have only arguments of a single type, or a single array as an argument.

```
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.

Propagation strength annotations

```
annotation bounds
```

Annotate a constraint to use bounds propagation

Note: This annotation will be deprecated. Use bounds_propagation instead.

```
annotation bounds_propagation
```

Annotate a constraint to use bounds propagation

```
annotation domain
```

Annotate a constraint to use domain propagation

Note: This annotation will be deprecated. Use domain_propagation instead.

```
annotation domain_propagation
```

Annotate a constraint to use domain propagation

```
annotation value_propagation
```

Annotate a constraint to use value propagation

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, or the smaller of the two middle values in case of an even number of elements 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

```
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`

```
annotation restart_none
```

Do not restart

Other declarations

Functions and Predicates

```
1.  annotation bool_search(array [$X] of var bool: x,
                           ann: select,
                           ann: choice,
                           ann: explore)

2.  annotation bool_search(array [$X] of var bool: x,
                           ann: select,
                           ann: choice)

3.  annotation bool_search(array [$X] of var opt bool: x,
                           ann: select,
                           ann: choice,
                           ann: explore)

4.  annotation bool_search(array [$X] of var opt bool: x,
                           ann: select,
                           ann: choice)
```

1. 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).

2. 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).
3. Search annotation for optional Boolean variables. 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).
4. Search annotation for optional Boolean variables. 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).

```
1.  annotation float_search(array [$X] of var float: x,
                            float: prec,
                            ann: select,
                            ann: choice,
                            ann: explore)

2.  annotation float_search(array [$X] of var float: x,
                            float: prec,
                            ann: select,
                            ann: choice)

3.  annotation float_search(array [$X] of var opt float: x,
                            float: prec,
                            ann: select,
                            ann: choice,
                            ann: explore)

4.  annotation float_search(array [$X] of var opt float: x,
                            float: prec,
                            ann: select,
                            ann: choice)
```

1. 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).
2. 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).
3. Search annotation for optional float variables. 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).
4. Search annotation for optional float variables. 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).

```
1.  annotation int_search(array [$X] of var $$E: x,
                          ann: select,
                          ann: choice,
                          ann: explore)

2.  annotation int_search(array [$X] of var $$E: x,
                          ann: select,
                          ann: choice)

3.  annotation int_search(array [$X] of var opt $$E: x,
                          ann: select,
                          ann: choice,
                          ann: explore)

4.  annotation int_search(array [$X] of var opt $$E: x,
                          ann: select,
                          ann: choice)
```

1. 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).
2. 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).
3. Search annotation for optional integer variables. 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).
4. Search annotation for optional integer variables. 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).

```
1.  annotation set_search(array [$X] of var set of int: x,
                          ann: select,
                          ann: choice,
                          ann: explore)

2.  annotation set_search(array [$X] of var set of int: x,
                          ann: select,
                          ann: choice)
```

1. 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).
2. 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).

Annotations

```
annotation seq_search(array [int] of ann: s)
```

Sequentially perform the searches specified in array `s`

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.

```
1.  annotation warm_start(array [int] of var bool: x,
                          array [int] of opt bool: v)

2.  annotation warm_start(array [int] of var int: x,
                          array [int] of opt int: v)

3.  annotation warm_start(array [int] of var float: x,
                          array [int] of opt float: v)

4.  annotation warm_start(array [int] of var set of int: x,
                          array [int] of opt set of int: v)
```

1. Specify warm start values `v` for an array of booleans `x`
2. Specify warm start values `v` for an array of integers `x`
3. Specify warm start values `v` for an array of floats `x`
4. Specify warm start values `v` for an array of sets `x`

Other declarations

```
1.  annotation warm_start(array [$X] of var bool: x, array [$X] of bool: v)

2.  annotation warm_start(array [int] of var bool: x,
                          array [int] of bool: v)

3.  annotation warm_start(array [$X] of var $$E: x, array [$X] of $$E: v)

4.  annotation warm_start(array [int] of var $$E: x, array [int] of $$E: v)
```

```
5.  annotation warm_start(array [$X] of var float: x,
                          array [$X] of float: v)

6.  annotation warm_start(array [int] of var float: x,
                          array [int] of float: v)

7.  annotation warm_start(array [$X] of var set of $$E: x,
                          array [$X] of set of $$E: v)

8.  annotation warm_start(array [int] of var set of int: x,
                          array [int] of set of int: v)
```

1, 2.
Specify warm start values `v` for an array of booleans `x`

3, 4.
Specify warm start values `v` for an array of integers `x`

5, 6.
Specify warm start values `v` for an array of floats `x`

7, 8.
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.

Large Neighbourhood Search annotations

These annotations can be used on the solve item (similar to search annotations) to specify Large Neighbourhood Search meta-heuristics.

These annotations specify how the search is restricted to a neighbourhood when a solver restarts. The annotations define which variables will be fixed to their values from a previous solution in that case. All other variables will be reset to their initial domains by the restart.

The annotations can be used together with search annotations and restart annotations.

```
annotation relax_and_reconstruct(array [int] of var int: x, int: p)
```

Random neighbourhood over variables `x` with probability `p` percent

Upon restart, each variable in `x` is fixed to the value of the incumbent solution with a probability of `p` percent.

Context annotations

These are used internally by the compiler, and should not be used in models.

```
annotation ctx_mix
```

Mixed context

```
annotation ctx_neg
```

Negative context

```
annotation ctx_pos
```

Positive context

```
annotation ctx_root
```

Root context

Redundant and symmetry breaking 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

4.2.1.3 Option type support

These functions and predicates implement the standard library for working with option types. Note that option type support is still incomplete.

Option type support for Booleans

```
predicate absent(var opt bool: x)
```

True iff x is absent

```
predicate deopt(var opt bool: x)
```

Return value of x (assumes that x is not absent)

```
predicate occurs(var opt bool: x)
```

True iff x is not absent

Option type support for integers

```
predicate absent(var opt $$E: x)
```

True iff x is absent

```
function var $$E: deopt(var opt $$E: x)
```

Return value of x (assumes that x is not absent)

```
test had_zero(var opt $$E: x)
test had_zero(opt $$E: x)
test had_zero(array [int] of var opt $$E: x)
```

True if x had zero in its original domain.

Returns true if absent zero representation is disabled or if it is possible that $\text{occurs}(x) \wedge \text{deopt}(x) = 0$

```
predicate occurs(var opt $$E: x)
```

True iff x is not absent

Option type support for floats

```
predicate absent(var opt float: x)
```

True iff x is absent

```
function var float: deopt(var opt float: x)
```

Return value of x (assumes that x is not absent)

```
predicate occurs(var opt float: x)
```

True iff x is not absent

Other declarations

```
1. test absent($T: x)
2. test absent(var $T: x)
3. test absent(set of $$T: x)
4. test absent(var set of $$T: x)
5. test absent(opt $T: x)
```

1-4.
Test if x is absent (always returns false)

5. Test if x is absent

```
1. function $$T: deopt(opt $$T: x)
2. function $T: deopt(opt $T: x)
3. function array [$$U] of $$T: deopt(array [$$U] of opt $$T: x)
```

```
4.  function array [$$U] of $T: deopt(array [$$U] of opt $T: x)

5.  function var $T: deopt(var $T: x)

6.  function $T: deopt($T: x)

7.  function array [$U] of var $T: deopt(array [$U] of var $T: x)

8.  function array [$$U] of var bool: deopt(array [$$U] of var opt bool: x)

9.  function array [$$U] of var $$E: deopt(array [$$U] of var opt $$E: x)

10. function array [$$U] of var float: deopt(array [$$U] of var opt float: x)
```

1. Return value of x if x is not absent. Returns undefined when evaluated on absent value.
2. Return value of x if x is not absent. Returns undefined when evaluated on absent value.

3, 4.
Return array of the value of x[i] where x[i] is not absent. Returns undefined when evaluated with an absent element.

5-7.
Return value x unchanged (since x is guaranteed to be non-optional).

8-10.
Return array of the value of x[i] (assumes that x[i] is not absent).

```
1.  test occurs($T: x)

2.  test occurs(var $T: x)

3.  test occurs(set of $$T: x)

4.  test occurs(var set of $$T: x)

5.  test occurs(opt $T: x)
```

1-4.
Test if x is not absent (always returns true)

5. Test if x is not absent

4.2.1.4 Array sorting operations

```
function array [int] of $$E: arg_sort(array [$$E] of $$T: x)
function array [int] of $$E: arg_sort(array [$$E] of float: x)
```

Returns the permutation p which causes x to be in sorted order hence x[p[i]] <= x[p[i+1]].

The permutation is the stable sort hence x[p[i]] = x[p[i+1]] → p[i] < p[i+1].

```
function array [$$E] of $$F: sort(array [$$E] of $$F: x)
function array [$$E] of float: sort(array [$$E] of float: x)
function array [$$E] of bool: sort(array [$$E] of bool: x)
```

Return values from array x sorted in non-decreasing order

```
function array [$$E] of any $T: sort_by(array [$$E] of any $T: x,
                                        array [$$E] of int: y)
function array [$$E] of any $T: sort_by(array [$$E] of any $T: x,
                                        array [$$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].

4.2.1.5 Language information

These functions return information about the MiniZinc system.

Constants

```
opt int: mzn_max_version_required
```

If defined, this can be used to check that the MiniZinc compiler is not newer than the given version.

```
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.

Functions and Predicates

```
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

---

4.2.1.6 Compiler options

Constants

```
opt bool: mzn_absent_zero
```

Whether to use zero as the underlying representation for absent optional variables.

If not specified or set to true, zero is used as the underlying representation for absent optional variables, breaking symmetry. If set to false, the underlying representation of absent optional variables is left unconstrained.

```
opt bool: mzn_half_reify_clause
```

Whether to half reify bool_clause constraints

```
opt bool: mzn_ignore_redundant_constraints
```

Whether to ignore redundant constraints

If not specified or set to false, it depends on the solver library whether constraints that are wrapped in a redundant_constraint call are in fact compiled. If set to true, they are not compiled, independent of the solver library.

```
opt bool: mzn_ignore_symmetry_breaking_constraints
```

Whether to ignore symmetry breaking constraints

If not specified or set to false, it depends on the solver library whether constraints that are wrapped in a symmetry_breaking_constraint call are in fact compiled. If set to true, they are not compiled, independent of the solver library.

```
opt bool: mzn_opt_annotate_computed_domains
```

Whether to generate computed_domain annotations

```
opt bool: mzn_opt_annotate_defines_var
```

Whether to generate defines_var annotations

```
opt bool: mzn_opt_only_range_domains
```

Whether to only generate domains that are contiguous ranges

Functions and Predicates

```
test mzn_check_absent_zero()
```

Check whether to use zero as the underlying representation of absent optional variables

---

```
test mzn_check_annotate_computed_domains()
```

Check whether to generate computed_domain annotations

```
test mzn_check_annotate_defines_var()
```

Check whether to generate defines_var annotations

```
test mzn_check_half_reify_clause()
```

Check to half reify bool_clause constraints

```
test mzn_check_ignore_redundant_constraints()
```

Check whether to ignore redundant constraints

```
test mzn_check_ignore_symmetry_breaking_constraints()
```

Check whether to ignore symmetry breaking constraints

```
test mzn_check_only_range_domains()
```

Check whether to only generate domains that are contiguous ranges

---

4.2.1.7 Conditionals

These functions implement conditional (if-then-else-endif) constraints.

```
predicate if_then_else(array [int] of var bool: c,
                       array [int] of int: x,
                       var int: y)
predicate if_then_else(array [int] of var bool: c,
                       array [int] of var int: x,
                       var int: y)
predicate if_then_else(array [int] of var bool: c,
                       array [int] of opt int: x,
                       var opt int: y)
predicate if_then_else(array [int] of var bool: c,
                       array [int] of var opt int: x,
                       var opt int: y)
predicate if_then_else(array [int] of var bool: c,
                       array [int] of bool: x,
                       var bool: y)
predicate if_then_else(array [int] of var bool: c,
                       array [int] of var bool: x,
                       var bool: y)
predicate if_then_else(array [int] of var bool: c,
                       array [int] of opt bool: x,
                       var opt bool: y)
predicate if_then_else(array [int] of var bool: c,
                       array [int] of var opt bool: x,
                       var opt bool: y)
predicate if_then_else(array [int] of var bool: c,
                       array [int] of float: x,
                       var float: y)
predicate if_then_else(array [int] of var bool: c,
                       array [int] of var float: x,
                       var float: y)
predicate if_then_else(array [int] of var bool: c,
                       array [int] of opt float: x,
                       var opt float: y)
predicate if_then_else(array [int] of var bool: c,
                       array [int] of var opt float: x,
                       var opt float: y)
predicate if_then_else(array [int] of var bool: c,
                       array [int] of set of int: x,
                       var set of int: y)
predicate if_then_else(array [int] of var bool: c,
                       array [int] of var set of int: x,
                       var set of int: y)
```

Conditional constraint $\{\mathbf{c}[i] \wedge \nexists \mathbf{c}[1..i-1] \rightarrow \mathbf{y} = \mathbf{x}[i]\}$

This constraint is generated by the compiler for if-then-else expressions. The last entry in the `c` array is always the constant true, corresponding to the else case.

```
predicate if_then_else_partiality(array [int] of var bool: c,
                                  array [int] of var bool: d,
                                  var bool: b)
```

Conditional partiality constraint

This constraint is generated by the compiler for if-then-else expressions with potentially undefined cases. The last entry in the `c` array is always the constant true, corresponding to the else case. The `d`[i] variable represents whether case `i` is defined. Constrains that if `b` is defined, then the selected case must be defined, and if the selected case is undefined, then `b` must be undefined.

4.2.1.8 Reflection operations

These functions return information about declared or inferred variable bounds and domains.

```
test annotate(any $T: x, ann: a)
```

Annotate declaration of `x` with annotation `a`

```
function set of $$E: dom(var $$E: x)
function set of $$E: dom(var opt $$E: x)
```

Return domain of `x`

```
function set of $$E: dom_array(array [$T] of var $$E: x)
function set of $$E: dom_array(array [$T] of var opt $$E: x)
```

Return union of all domains of the elements in array `x`

```
function set of $$E: dom_array_occurring(array [$T] of var opt $$E: x)
```

Return union of all domains of the elements in array `x`, ignoring absent elements

---

```
function set of $$E: dom_bounds_array(array [$T] of var $$E: x)
function set of $$E: dom_bounds_array(array [$T] of var opt $$E: 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

```
1.  function $T: fix(var $T: x)

2.  function opt $T: fix(var opt $T: x)

3.  function array [$U] of $T: fix(array [$U] of var $T: x)

4.  function array [$U] of opt $T: fix(array [$U] of var opt $T: x)
```

1, 2.
Check if the value of x is fixed at this point in evaluation. If it is fixed, return its value, otherwise abort.

3, 4.
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_ann(any $T: x, ann: a)
```

Test if x is annotated a

```
test has_bounds(var $$E: x)
test has_bounds(var float: x)
```

Test if variable x has declared, finite bounds

```
test has_ub_set(var set of $$E: x)
```

Test if variable x has a declared, finite upper bound

```
1.  test is_fixed(any $T: x)

2.  test is_fixed(array [$U] of any $T: x)
```

1. Test if x is fixed
2. Test if every element of array x is fixed

```
test is_same(any $T: x, any $U: y)
```

Test if x and y are the same variable

```
1. function $$E: lb(var $$E: x)
2. function $$E: lb(var opt $$E: x)
3. function float: lb(var float: x)
4. function float: lb(var opt float: x)
5. function set of $$E: lb(var set of $$E: x)
6. function array [$U] of $$E: lb(array [$U] of var $$E: x)
7. function array [$U] of float: lb(array [$U] of var float: x)
8. function array [$U] of set of $$E: lb(array [$U] of var set of $$E: x)
```

1-5.
Return lower bound of x

6-8.
Return array of lower bounds of the elements in array x

```
1. function $$E: lb_array(array [$U] of var opt $$E: x)
2. function float: lb_array(array [$U] of var opt float: x)
3. function set of $$E: lb_array(array [$U] of var set of $$E: x)
```

1, 2.
Return minimum of all lower bounds of the elements in array x

3. Return intersection of all lower bounds of the elements in array x

```
1. function $$E: ub(var $$E: x)
2. function $$E: ub(var opt $$E: x)
3. function float: ub(var float: x)
4. function float: ub(var opt float: x)
5. function set of $$E: ub(var set of $$E: x)
6. function array [$U] of $$E: ub(array [$U] of var $$E: x)
7. function array [$U] of float: ub(array [$U] of var float: x)
8. function array [$U] of set of $$E: ub(array [$U] of var set of $$E: x)
```

1-5.
Return upper bound of `x`

6-8.
Return array of upper bounds of the elements in array `x`

```
1.  function $$E: ub_array(array [$U] of var opt $$E: x)

2.  function float: ub_array(array [$U] of var opt float: x)

3.  function set of $$E: ub_array(array [$U] of var set of $$E: x)
```

1, 2.
Return maximum of all upper bounds of the elements in array `x`

3. Return union of all upper bounds of the elements in array `x`

4.2.1.9 Assertions and debugging functions

These functions help debug models and check that input data conforms to the expectations.

Constants

```
bool: debug_mode
```

Set to true iff `--debug` flag is set.

Functions and Predicates

```
test abort(string: msg)
```

Abort evaluation and print message `msg`.

```
1.  function any $T: assert(bool: b, string: msg, any $T: x)

2.  function array [$U] of any $T: assert(bool: b,
                                          string: msg,
                                          array [$U] of any $T: x)

3.  test assert(bool: b, string: msg)
```

1, 2.
If `b` is true, return `x`, otherwise abort with message `msg`.

3. If `b` is true, return true, otherwise abort with message `msg`.

```
test assert_dbg(bool: b, string: msg)
```

Return true, when the `--debug` flag is set, check if `b` is true, otherwise abort with message `msg`.

```
1.  function $T: 'default'(opt $T: x, $T: y)

2.  function opt $T: 'default'(opt $T: x, opt $T: y)

3.  function var $T: 'default'(var opt $T: x, var $T: y)

4.  function var opt $T: 'default'(var opt $T: x, var opt $T: y)

5.  function array [$U] of $T: 'default'(array [$U] of $T: x,
                                         array [$U] of $T: y)

6.  function array [$U] of opt $T: 'default'(array [$U] of opt $T: x,
                                             array [$U] of opt $T: y)

7.  function array [$U] of var $T: 'default'(array [$U] of var $T: x,
                                             array [$U] of var $T: y)

8.  function array [$U] of var opt $T: 'default'(array [$U] of var opt $T: x,
                                                 array [$U] of var opt $T: y)

9.  function set of $$E: 'default'(set of $$E: x, set of $$E: y)

10. function var set of $$E: 'default'(var set of $$E: x,
                                       var set of $$E: y)

11. function set of $T: 'default'(opt set of $T: x, set of $T: y)

12. function opt set of $T: 'default'(opt set of $T: x, opt set of $T: y)
```

1-4.
If `x` is defined and not absent, return `x`, otherwise return `y`.

5-10.
If `x` is defined, return `x`, otherwise return `y`.

11, 12.
If `x` is define amd mpt absent, return `x`, otherwise return `y`.

```
annotation expression_name_dbg(string: s)
```

Used to attach a `expression_name` annotation with name `s` to an expression when --debug is enabled.

```
function string: logstream_to_string()
```

Return logging stream as string

```
test mzn_internal_check_debug_mode()
```

Returns true iff `--debug` flag is set.

```
1.  function any $T: trace(string: msg, any $T: x)

2.  function array [$U] of any $T: trace(string: msg,
                                         array [$U] of any $T: x)

3.  test trace(string: msg)
```

1, 2.
Return `x`, and print message `msg`.

3. Return true, and print message `msg`.

```
test trace_dbg(string: msg)
```

Return true, and when `--debug` flag is set print message `msg`.

```
function any $T: trace_exp(any $T: x)
function array [$U] of any $T: trace_exp(array [$U] of any $T: x)
```

Return `x`, and print a string representation of it.

```
1.  function any $T: trace_logstream(string: msg, any $T: x)

2.  function array [$U] of any $T: trace_logstream(string: msg,
                                                   array [$U] of any $T: x)

3.  test trace_logstream(string: msg)
```

1, 2.
Return `x`, and print message `msg` to logging stream.

3. Return true, and print message `msg` to logging stream.

```
1.  function any $T: trace_stdout(string: msg, any $T: x)

2.  function array [$U] of any $T: trace_stdout(string: msg,
                                                array [$U] of any $T: x)

3.  test trace_stdout(string: msg)
```

1, 2.
Return `x`, and print message `msg`.

3. Return true, and print message `msg`.

```
test trace_to_json_section(string: section, opt $T: x)
test trace_to_json_section(string: section, array [$U] of opt $T: x)
```

Return true, and print `x` to the JSON section `section`.

```
1.  function any $T: trace_to_section(string: section,
                                      string: msg,
                                      any $T: x)

2.  function array [$U] of any $T: trace_to_section(string: section,
                                                    string: msg,
                                                    array [$U] of any $T: x)

3.  test trace_to_section(string: section, string: msg)
```

1, 2.
Return `x`, and print message `msg` to section `section`.

3. Return true, and print message `msg` to section `section`.

Annotations

```
annotation mzn_break_here
```

With debug build of the MiniZinc compiler, call MiniZinc::mzn_break_here when flattening this expression to make debugging easier. This annotation is ignored by the release build.

4.2.1.10 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)
function float: cauchy(int: mean, float: scale)
```

Return a sample from the cauchy distribution defined by **mean**, **scale**

```
function float: chisquared(int: 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)
function float: exponential(float: lambda)
```

Return a sample from the exponential distribution defined by **lambda**

```
function float: fdistribution(float: d1, float: 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)
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)
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)
function float: normal(int: mean, float: std)
```

Return a sample from the normal distribution defined by **mean**, **std**

```
1.  function int: poisson(float: mean)

2.  function int: poisson(int: mean)
```

1. Return a sample from the poisson distribution defined by `mean`
2. Return a sample from the poisson distribution defined by an integer `mean`

```
function float: tdistribution(float: n)
function float: tdistribution(int: n)
```

Return a sample from the student's t-distribution defined by the sample size **n**

```
1.  function float: uniform(float: lowerbound, float: upperbound)

2.  function int: uniform(int: lowerbound, int: upperbound)

3.  function int: uniform(set of int: S)
```

1, 2.
Return a sample from the uniform distribution defined by **lowerbound**, **upperbound**

3. Return a sample from the uniform distribution defined by `S`

```
function float: weibull(float: shape, float: scale)
function float: weibull(int: shape, float: scale)
```

Return a sample from the Weibull distribution defined by **shape**, **scale**

4.2.1.11 Structured Output

These functions provide structured output for common structures modelled in the MiniZinc language.

```
function string: show_array2d_bool(array [int,int] of var bool: x)
```

Create string to visualise a 2D boolean array `x` as a checkered board.

```
1.  function string: show_gantt(array [$$T] of var int: start,
                                array [$$T] of var int: dur,
                                array [$$T] of string: name)

2.  function string: show_gantt(array [$$T] of var int: start,
                                array [$$T] of var int: dur)
```

1. Create string Gantt chart from task `start` times, duration `dur`, and their names `name`.
2. Create string Gantt chart from task `start` times and duration `dur`, using the index set as name for the tasks.

4.2.2 Global constraints

These constraints represent high-level modelling abstractions. Many solvers implement dedicated, efficient inference algorithms for these constraints, or provide a MiniZinc library decomposition that is better suited to the particular solving technology than the standard library decomposition.

4.2.2.1 All-Different and related constraints

```
1.  predicate all_different(array [$X] of var $$E: x)

2.  predicate all_different(array [$X] of var set of $$E: x)

3.  predicate all_different(array [$X] of var opt $$E: x)
```

1, 2.
Constrain the elements in the array `x` to be pairwise different.

3. Constrain the non-absent elements in the array `x` to be pairwise different. The absent value <> is allowed to occur multiple times.

```
predicate all_different_except(array [$X] of var $$E: vs,
                               set of $$E: S)
```

Constrain the elements of the array of integers `vs` to be pairwise different except for those values that appear in the set `S`.

```
predicate all_different_except_0(array [$X] of var int: vs)
```

Constrain the elements of the array of integers `vs` to be pairwise different except for those elements that are assigned the value 0.

```
predicate all_disjoint(array [$X] of var set of $$E: S)
```

Constrain the array of sets of integers `S` to be pairwise disjoint.

```
predicate all_equal(array [$X] of var $$E: x)
predicate all_equal(array [$X] of var set of $$E: x)
```

Constrain the elements of the array `x` to be all equal.

```
1.  predicate nvalue(var int: n, array [$X] of var $$E: x)

2.  predicate nvalue(var int: n, array [$X] of var opt $$E: x)

3.  function var int: nvalue(array [$X] of var $$E: x)

4.  function var int: nvalue(array [$X] of var opt $$E: x)
```

1, 2.
Requires that the number of distinct values in `x` is `n`.

3, 4.
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` → `x[j]=i`.

4.2.2.2 Lexicographic constraints

```
predicate lex2(array [$$E,$$F] of var $$T: x)
```

Require adjacent rows and adjacent columns in the array `x` to be lexicographically ordered. Adjacent rows and adjacent columns may be equal.

```
predicate lex2_strict(array [$$E,$$F] of var $$T: x)
```

Require adjacent rows and adjacent columns in the array `x` to be lexicographically ordered. Adjacent rows and adjacent columns cannot be equal.

```
predicate lex_chain(array [$$E,$$F] of var bool: a)
predicate lex_chain(array [$$E,$$F] of var $$T: a)
```

Requires that the columns of matrix `a` are lexicographically sorted, non-decreasing.

```
predicate lex_chain_greater(array [$$E,$$F] of var bool: a)
predicate lex_chain_greater(array [$$E,$$F] of var $$T: a)
```

Requires that the columns of matrix `a` are lexicographically sorted, strictly decreasing.

```
predicate lex_chain_greatereq(array [$$E,$$F] of var bool: a)
predicate lex_chain_greatereq(array [$$E,$$F] of var $$T: a)
```

Requires that the columns of matrix `a` are lexicographically sorted, non-increasing.

```
predicate lex_chain_greatereq_orbitope(array [$$E,$$F] of var int: a,
                                       int: kind)
```

Requires that the columns of binary matrix `a` are lexicographically sorted, non-increasing. Moreover, the second parameter `kind` has the following meaning: 0: no further constraints, 1: set-partitioning orbitope, 2: set-packing orbitope

---

```
predicate lex_chain_less(array [$$E,$$F] of var bool: a)
predicate lex_chain_less(array [$$E,$$F] of var $$T: a)
```

Requires that the columns of matrix `a` are lexicographically sorted, strictly increasing.

```
predicate lex_chain_lesseq(array [$$E,$$F] of var bool: a)
predicate lex_chain_lesseq(array [$$E,$$F] of var $$T: a)
```

Requires that the columns of matrix `a` are lexicographically sorted, non-decreasing.

```
predicate lex_chain_lesseq_orbitope(array [$$E,$$F] of var int: a,
                                    int: kind)
```

Requires that the columns of binary matrix `a` are lexicographically sorted, non-decreasing. Moreover, the second parameter `kind` has the following meaning: 0: no further constraints, 1: set-partitioning orbitope, 2: set-packing orbitope

---

```
predicate lex_greater(array [$$E] of var bool: x,
                      array [$$F] of var bool: y)
predicate lex_greater(array [$$E] of var $$T: x,
                      array [$$F] of var $$T: y)
predicate lex_greater(array [$$E] of var float: x,
                      array [$$F] of var float: y)
predicate lex_greater(array [$$E] of var set of $$T: x,
                      array [$$F] of var set of $$T: 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 [$$E] of var bool: x,
                        array [$$F] of var bool: y)
predicate lex_greatereq(array [$$E] of var $$T: x,
                        array [$$F] of var $$T: y)
predicate lex_greatereq(array [$$E] of var float: x,
                        array [$$F] of var float: y)
predicate lex_greatereq(array [$$E] of var set of $$T: x,
                        array [$$F] of var set of $$T: 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 [$$E] of var bool: x,
                   array [$$F] of var bool: y)
predicate lex_less(array [$$E] of var $$T: x,
                   array [$$F] of var $$T: y)
predicate lex_less(array [$$E] of var float: x,
                   array [$$F] of var float: y)
predicate lex_less(array [$$E] of var set of $$T: x,
                   array [$$F] of var set of $$T: 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 [$$E] of var bool: x,
                     array [$$F] of var bool: y)
predicate lex_lesseq(array [$$E] of var float: x,
                     array [$$F] of var float: y)
predicate lex_lesseq(array [$$E] of var $$T: x,
                     array [$$F] of var $$T: y)
predicate lex_lesseq(array [$$E] of var set of $$T: x,
                     array [$$F] of var set of $$T: 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.

```
1.  predicate seq_precede_chain(array [int] of var int: x)

2.  predicate seq_precede_chain(array [int] of var set of int: x)
```

1. Requires that i precedes i+1 in the array x for all positive i.
2. Requires that i appears in a set in array x before i+1 for all positive i

---

```
predicate strict_lex2(array [$$E,$$F] of var $$T: x)
```

Requires adjacent rows and adjacent columns in the array x to be lexicographically ordered. Adjacent rows and adjacent columns cannot be equal.

```
1.  predicate value_precede($$E: s, $$E: t, array [$$X] of var $$E: x)

2.  predicate value_precede($$E: s, $$E: t, array [$$X] of var opt $$E: x)

3.  predicate value_precede($$E: s,
                            $$E: t,
                            array [$$X] of var set of $$E: x)
```

1, 2.

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.

3. 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.

```
1.  predicate value_precede_chain(array [$$X] of $$E: c,
                                  array [$$Y] of var $$E: x)

2.  predicate value_precede_chain(array [$$X] of $$E: c,
                                  array [$$Y] of var opt $$E: x)

3.  predicate value_precede_chain(array [$$X] of $$E: c,
                                  array [$$Y] of var set of $$E: x)
```

1, 2.

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].

3. 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].

```
predicate var_perm_sym(array [$$X] of var $$Y: x,
                       array [$$Z,$$X] of $$X: p)
```

Requires that the array x is lex least under the given list of permutations p

---

```
predicate var_sqr_sym(array [$$X,$$X] of var $$Y: x)
```

Requires that the square array x is lex least under the 7 symmetries: 3 rotations and 4 reflections

---

4.2.2.3 Sorting constraints

```
1.  predicate arg_sort(array [$$E] of var $$T: x,
                       array [int] of var $$E: p)

2.  predicate arg_sort(array [$$E] of var float: x,
                       array [int] of var $$E: p)

3.  function array [int] of var $$E: arg_sort(array [$$E] of var $$T: x)

4.  function array [int] of var $$E: arg_sort(array [$$E] of var float: x)
```

1, 2.

Constrains p to be the permutation which causes x to be in sorted order hence x[p[i]] <= x[p[i+1]].

The permutation is the stable sort hence x[p[i]] = x[p[i+1]] → p[i] < p[i+1].

3, 4.

Returns the permutation p which causes x to be in sorted order hence x[p[i]] <= x[p[i+1]].

The permutation is the stable sort hence x[p[i]] = x[p[i+1]] → p[i] < p[i+1].

```
predicate decreasing(array [$X] of var bool: x)
predicate decreasing(array [$X] of var opt float: x)
predicate decreasing(array [$X] of var opt $$E: x)
predicate decreasing(array [$X] of var set of $$E: x)
```

Requires that the array x is in decreasing order (duplicates are allowed).

```
predicate increasing(array [$X] of var bool: x)
predicate increasing(array [$X] of var float: x)
predicate increasing(array [$X] of var opt float: x)
predicate increasing(array [$X] of var $$E: x)
predicate increasing(array [$X] of var opt $$E: x)
predicate increasing(array [$X] of var set of $$E: x)
```

Requires that the array x is in increasing order (duplicates are allowed).

```
1.  predicate sort(array [$$E] of var $$T: x, array [$$F] of var $$T: y)

2.  function array [$$E] of var $$T: sort(array [$$E] of var $$T: x)
```

1. Requires that the multiset of values in x are the same as the multiset of values in y but y is in sorted order.
2. Return a multiset of values that is the same as the multiset of values in x but in sorted order.

```
predicate strictly_decreasing(array [$X] of var opt $$E: x)
predicate strictly_decreasing(array [$X] of var opt float: x)
predicate strictly_decreasing(array [$X] of var set of $$E: x)
```

Requires that the array x is in a strictly decreasing order (duplicates are *not* allowed).

```
1.  predicate strictly_increasing(array [$X] of var $$E: x)

2.  predicate strictly_increasing(array [$X] of var opt $$E: x)

3.  predicate strictly_increasing(array [$X] of var float: x)

4.  predicate strictly_increasing(array [$X] of var opt float: x)
```

```
5.  predicate strictly_increasing(array [$X] of var set of $$E: x)
```

1, 2.
Requires that the array `x` is in a strictly increasing order (duplicates are *not* allowed).

3-5.
Requires that the array `x` is in increasing order (duplicates are allowed).

4.2.2.4 Channeling constraints

```
predicate int_set_channel(array [$$I] of var $$J: x,
                          array [$$J] of var set of $$I: y)
```

Requires that array of int variables `x` and array of set variables `y` are related such that (`x[i] = j`) ↔ (`i in y[j]`).

```
1.  predicate inverse(array [$$X] of var $$Y: f,
                      array [$$Y] of var $$X: invf)

2.  function array [$$E] of var $$F: inverse(array [$$F] of var $$E: f)

3.  function array [$$E] of var opt $$F: inverse(array [$$F] of var opt $$E: f)

4.  function array [$$E] of $$F: inverse(array [$$F] of $$E: f)
```

1. 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.

2-4.
Given a function `f` represented as an array, return the inverse function.

```
predicate inverse_in_range(array [$$A] of var $$B: X,
                           array [$$B] of var $$A: Y)
```

If the `i` th variable of the collection `X` is assigned to `j` and if `j` is in the index set of `Y` then the `j` th variable of the collection `Y` is assigned to `i`.

Conversely, if the `j` th variable of the collection `Y` is assigned to `i` and if `i` is in the index set of `X` then the `i` th variable of the collection `X` is assigned to `j`.

```
predicate inverse_set(array [$$X] of var set of $$Y: f,
                      array [$$Y] of var set of $$X: invf)
```

Constrains two arrays of set of int variables, `f` and `invf`, so that a `j` in f[`i`] iff `i` in invf[`j`]. 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 $$E: s,
                               array [$$E] of var bool: b)
```

Constrains the array of Booleans `b` to be a representation of the set `s`: `i` in `s` ↔ `b`[`i`].

The index set of `b` must be a superset of the possible values of `s`.

4.2.2.5 Counting constraints

These constraints count and restrict how many times certain values occur in an array of variables. MiniZinc will automatically generate the basic counting constraints below from expressions such as `count(i in x)(i=c) <= d`, so you can write models in this much more readable style instead of using these predicates. However, if your model contains multiple counting constraints over the same array, constraints like `distribute` or `global_cardinality` below may be useful.

```
1.  predicate among(var int: n, array [$X] of var $$E: x, set of $$E: v)

2.  function var int: among(array [$X] of var $$E: x, set of $$E: v)

3.  function int: among(array [$X] of $$E: xi, set of $$E: v)
```

1. Requires exactly `n` variables in `x` to take one of the values in `v`.
2. Returns the number of variables in `x` that take one of the values in `v`.
3. Returns the number of variables in `xi` that take one of the values in `v`.

```
predicate at_least(int: n,
                   array [$X] of var set of $$E: x,
                   set of $$E: v)
```

Requires at least `n` variables in `x` to take the value `v`.

```
predicate at_most(int: n,
                  array [$X] of var set of $$E: x,
                  set of $$E: v)
```

Requires at most `n` variables in `x` to take the value `v`.

```
predicate at_most1(array [$X] of var set of $$E: s)
```

Requires that each pair of sets in s overlap in at most one element.

```
1.  function var int: count(array [$X] of var opt $$E: x, var $$E: y)

2.  predicate count(array [$X] of var opt $$E: x, var $$E: y, var int: c)
```

1. Returns the number of occurrences of y in x.
2. Constrains c to be the number of occurrences of y in x.

```
1.  predicate count_eq(array [$X] of var $$E: x, var $$E: y, var int: c)

2.  predicate count_eq(array [$X] of var bool: x, var bool: y, var int: c)

3.  predicate count_eq(array [$X] of var float: x,
                       var float: y,
                       var int: c)

4.  predicate count_eq(array [$X] of var set of $$E: x,
                       var set of $$E: y,
                       var int: c)

5.  predicate count_eq(array [$X] of var opt $$E: x,
                       var $$E: y,
                       var int: c)

6.  predicate count_eq(array [$X] of var $$E: x, $$E: y, int: c)

7.  predicate count_eq(array [$X] of var bool: x, bool: y, int: c)

8.  predicate count_eq(array [$X] of var float: x, float: y, int: c)

9.  predicate count_eq(array [$X] of var set of $$E: x,
                       set of $$E: y,
                       int: c)

10. function var int: count_eq(array [$X] of var opt $$E: x, var $$E: y)
```

1-9.
Constrains c to be the number of occurrences of y in x.

10. Returns the number of occurrences of y in x.

```
1.  predicate count_geq(array [$X] of var $$E: x, var $$E: y, var int: c)

2.  predicate count_geq(array [$X] of var bool: x, var bool: y, var int: c)

3.  predicate count_geq(array [$X] of var float: x,
                       var float: y,
                       var int: c)
```

```
4.  predicate count_geq(array [$X] of var set of $$E: x,
                        var set of $$E: y,
                        var int: c)

5.  predicate count_geq(array [$X] of var opt $$E: x,
                        var $$E: y,
                        var int: c)

6.  predicate count_geq(array [$X] of var $$E: x, $$E: y, int: c)

7.  predicate count_geq(array [$X] of var bool: x, bool: y, int: c)

8.  predicate count_geq(array [$X] of var float: x, float: y, int: c)

9.  predicate count_geq(array [$X] of var set of $$E: x,
                        set of $$E: y,
                        int: c)
```

1-4.
Constrains c to be greater than or equal to the number of occurrences of y in x.

5. Constrains c to be strictly greater than the number of occurrences of y in x.

6-9.
Constrains c to be greater than or equal to the number of occurrences of y in x.

```
predicate count_gt(array [$X] of var $$E: x, var $$E: y, var int: c)
predicate count_gt(array [$X] of var bool: x, var bool: y, var int: c)
predicate count_gt(array [$X] of var float: x,
                   var float: y,
                   var int: c)
predicate count_gt(array [$X] of var set of $$E: x,
                   var set of $$E: y,
                   var int: c)
predicate count_gt(array [$X] of var opt $$E: x,
                   var $$E: y,
                   var int: c)
predicate count_gt(array [$X] of var $$E: x, $$E: y, int: c)
predicate count_gt(array [$X] of var bool: x, bool: y, int: c)
predicate count_gt(array [$X] of var float: x, float: y, int: c)
predicate count_gt(array [$X] of var set of $$E: x,
                   set of $$E: y,
                   int: c)
```

Constrains c to be strictly greater than the number of occurrences of y in x.

```
predicate count_leq(array [$X] of var $$E: x, var $$E: y, var int: c)
predicate count_leq(array [$X] of var bool: x, var bool: y, var int: c)
predicate count_leq(array [$X] of var float: x,
                    var float: y,
                    var int: c)
predicate count_leq(array [$X] of var set of $$E: x,
```

```
                    var set of $$E: y,
                    var int: c)
predicate count_leq(array [$X] of var opt $$E: x,
                    var $$E: y,
                    var int: c)
predicate count_leq(array [$X] of var $$E: x, $$E: y, int: c)
predicate count_leq(array [$X] of var bool: x, bool: y, int: c)
predicate count_leq(array [$X] of var float: x, float: y, int: c)
predicate count_leq(array [$X] of var set of $$E: x,
                    set of $$E: y,
                    int: c)
```

Constrains `c` to be less than or equal to the number of occurrences of `y` in `x`.

```
predicate count_lt(array [$X] of var $$E: x, var $$E: y, var int: c)
predicate count_lt(array [$X] of var bool: x, var bool: y, var int: c)
predicate count_lt(array [$X] of var float: x,
                   var float: y,
                   var int: c)
predicate count_lt(array [$X] of var set of $$E: x,
                   var set of $$E: y,
                   var int: c)
predicate count_lt(array [$X] of var opt $$E: x,
                   var $$E: y,
                   var int: c)
predicate count_lt(array [$X] of var $$E: x, $$E: y, int: c)
predicate count_lt(array [$X] of var bool: x, bool: y, int: c)
predicate count_lt(array [$X] of var float: x, float: y, int: c)
predicate count_lt(array [$X] of var set of $$E: x,
                   set of $$E: y,
                   int: c)
```

Constrains `c` to be strictly less than the number of occurrences of `y` in `x`.

```
predicate count_neq(array [$X] of var $$E: x, var $$E: y, var int: c)
predicate count_neq(array [$X] of var bool: x, var bool: y, var int: c)
predicate count_neq(array [$X] of var float: x,
                    var float: y,
                    var int: c)
predicate count_neq(array [$X] of var set of $$E: x,
                    var set of $$E: y,
                    var int: c)
predicate count_neq(array [$X] of var opt $$E: x,
                    var $$E: y,
                    var int: c)
predicate count_neq(array [$X] of var $$E: x, $$E: y, int: c)
predicate count_neq(array [$X] of var bool: x, bool: y, int: c)
predicate count_neq(array [$X] of var float: x, float: y, int: c)
predicate count_neq(array [$X] of var set of $$E: x,
                    set of $$E: y,
                    int: c)
```

Constrains `c` to be not equal to the number of occurrences of `y` in `x`.

```
1.  predicate distribute(array [$X] of var int: card,
                         array [$X] of var $$E: value,
                         array [$Y] of var $$E: base)

2.  function array [$X] of var int: distribute(array [$X] of var $$E: value,
                                               array [$Y] of var $$E: base)
```

1. Requires that `card[i]` is the number of occurrences of `value[i]` in `base`. The values in `value` need not be distinct.
2. 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 [$X] of var set of $$E: x,
                  set of $$E: v)
```

Requires exactly `n` variables in `x` to take the value `v`.

```
1.  predicate global_cardinality(array [$X] of var $$E: x,
                                 array [$Y] of $$E: cover,
                                 array [$Y] of var int: counts)

2.  predicate global_cardinality(array [$X] of var opt $$E: x,
                                 array [$Y] of $$E: cover,
                                 array [$Y] of var int: counts)

3.  predicate global_cardinality(array [$X] of var $$E: x,
                                 array [$Y] of $$E: cover,
                                 array [$Y] of int: lbound,
                                 array [$Y] of int: ubound)

4.  predicate global_cardinality(array [$X] of var opt $$E: x,
                                 array [$Y] of $$E: cover,
                                 array [$Y] of int: lbound,
                                 array [$Y] of int: ubound)

5.  predicate global_cardinality(array [$X] of var set of $$E: x,
                                 array [$Y] of $$E: cover,
                                 array [$Y] of var int: counts)

6.  predicate global_cardinality(array [$X] of var set of $$E: x,
                                 array [$Y] of $$E: cover,
                                 array [$Y] of int: lbound,
                                 array [$Y] of int: ubound)

7.  function array [$Y] of var int: global_cardinality(array [$X] of var $$E: x,
                                                       array [$Y] of $$E: cover)
```

```
8.  function array [$Y] of var int: global_cardinality(array [$X] of var opt $$E: x,
                                                       array [$Y] of $$E: cover)
```

1, 2.
Requires that the number of occurrences of `cover[i]` in `x` is `counts[i]`.

3, 4.
Requires that for all `i`, the value `cover[i]` appears at least `lbound[i]` and at most `ubound[i]` times in the array `x`.

5. Requires that the number of occurrences of `cover[i]` in `x` is `counts[i]`.

6. Requires that for all `i`, the value `cover[i]` appears at least `lbound[i]` and at most `ubound[i]` times in the array `x`.

7, 8.
Returns the number of occurrences of `cover[i]` in `x`.

```
1.  predicate global_cardinality_closed(array [$X] of var $$E: x,
                                        array [$Y] of $$E: cover,
                                        array [$Y] of var int: counts)

2.  predicate global_cardinality_closed(array [$X] of var opt $$E: x,
                                        array [$Y] of $$E: cover,
                                        array [$Y] of var int: counts)

3.  predicate global_cardinality_closed(array [$X] of var $$E: x,
                                        array [$Y] of $$E: cover,
                                        array [$Y] of int: lbound,
                                        array [$Y] of int: ubound)

4.  predicate global_cardinality_closed(array [$X] of var opt $$E: x,
                                        array [$Y] of $$E: cover,
                                        array [$Y] of int: lbound,
                                        array [$Y] of int: ubound)

5.  predicate global_cardinality_closed(array [$X] of var set of $$E: x,
                                        array [$Y] of $$E: cover,
                                        array [$Y] of var int: counts)

6.  predicate global_cardinality_closed(array [$X] of var set of $$E: x,
                                        array [$Y] of $$E: cover,
                                        array [$Y] of int: lbound,
                                        array [$Y] of int: ubound)

7.  function array [$Y] of var int: global_cardinality_closed(array [$X] of var $
↪$E: x,
                                                              array [$Y] of $$E:␣
↪cover)

8.  function array [$Y] of var int: global_cardinality_closed(array [$X] of var opt
↪$$E: x,
                                                              array [$Y] of $$E:␣
↪cover)
```

1. Requires that the number of occurrences of `cover[i]` in `x` is `counts[i]`.

   The elements of `x` must take their values from `cover`.

2. Requires that the number of occurrences of `cover[i]` in `x` is `counts[i]`.

   The elements of `x` must take their values from `cover` or be absent.

3. 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. 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` or be absent.

5. Requires that the number of occurrences of `cover[i]` in `x` is `counts[i]`.

   The elements of `x` must take their values from `cover` or be absent.

6. 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` or be absent.

7. Returns an array with number of occurrences of `cover[i]` in `x`.

   The elements of `x` must take their values from `cover`.

8. Returns an array with number of occurrences of `cover[i]` in `x`.

   The elements of `x` must take their values from `cover` or be absent.

```
1.  predicate sliding_among(int: low,
                            int: up,
                            int: seq,
                            array [$X] of var $$E: x,
                            set of $$E: v)

2.  predicate sliding_among(int: low,
                            int: up,
                            int: seq,
                            array [$X] of var bool: x,
                            bool: b)
```

1. Every subsequence of `x` of length `seq` has at least `low` and at most `up` occurrences of the values in `v`
2. Every subsequence of `x` of length `seq` has at least `low` and at most `up` occurrences of the value `b`

4.2.2.6 Array-related constraints

```
predicate element(var $$E: i, array [$$E] of var bool: x, var bool: y)
predicate element(var $$E: i,
                  array [$$E] of var float: x,
                  var float: y)
predicate element(var $$E: i, array [$$E] of var $$T: x, var $$T: y)
predicate element(var $$E: i,
                  array [$$E] of var set of $$T: x,
                  var set of $$T: y)
```

Constrains `i` to be the index of the element `y` in the array `x`.

```
1.  predicate member(array [$X] of var bool: x, var bool: y)

2.  predicate member(array [$X] of var float: x, var float: y)

3.  predicate member(array [$X] of var $$E: x, var $$E: y)

4.  predicate member(array [$X] of var set of $$E: x, var set of $$E: y)

5.  predicate member(var set of $$E: x, var $$E: y)
```

1-4.
Requires that `y` occurs in the array `x`.

5. Requires that `y` occurs in the set `x`.

```
1.  predicate write(array [$$E] of var $$T: I,
                    var $$E: i,
                    var $$T: v,
                    array [$$E] of var $$T: O)

2.  function array [$$E] of var $$T: write(array [$$E] of var $$T: I,
                                           var $$E: i,
                                           var $$T: v)
```

1. Creates a new array `O` from an input array `I` with a change at position `i` to take value `v`

   `I` is an array of values `O` is an array of values with same index set as `I` `i` is an index for `I` `v` is the value to write

2. Return the input array `I` with a change at position `i` to take value `v`

   `I` is an array of values `i` is an index for `I` `v` is the value to write

---

```
1.  predicate writes(array [$$X] of var $$T: I,
                     array [$$Y] of var $$X: P,
                     array [$$Y] of var $$T: V,
```

```
                    array [$$X] of var $$T: O)

2.  function array [$$X] of var $$T: writes(array [$$X] of var $$T: I,
                                            array [$$Y] of var $$X: P,
                                            array [$$Y] of var $$T: V)
```

1. Creates a new array `O` from an input array `I` with a simultaneous change at positions `P` to values `V`

   `I` is an array of input values `O` is an array of integers with same index set as `I` `P` is an array of index values in `I` `V` is an array of values to write

2. Returns a new array from an input array `I` with a simultaneous change at positions `P` to values `V`

   `I` is an array of input values `P` is an array of index values in `I` `V` is an array of values to write

---

```
1.  predicate writes_seq(array [$$X] of var $$T: I,
                         array [$$Y] of var $$X: P,
                         array [$$Y] of var $$T: V,
                         array [$$X] of var $$T: O)

2.  function array [$$X] of var $$T: writes_seq(array [$$X] of var $$T: I,
                                                array [$$Y] of var $$X: P,
                                                array [$$Y] of var $$T: V)
```

1. Creates a new array `O` from an input array `I` with a sequence of changes at positions `P` to take values `V`

   `I` is an array of input values `O` is an array of output values with same index set as `I` `P` is an array of indices in `I` `V` is an array of values to write

2. Return the input array `I` with a sequence of changes at positions `P` to take values `V`

   `I` is an array of input values `P` is an array of index values in `I` `V` is an array of values to write

---

4.2.2.7 Set-related constraints

```
predicate disjoint(var set of $$E: s1, var set of $$E: s2)
```

Requires that sets `s1` and `s2` do not intersect.

```
predicate partition_set(array [$X] of var set of $$E: S,
                        set of $$E: universe)
```

Constrains the sets in array `S` to partition the `universe`.

```
1.  predicate roots(array [$$X] of var $$Y: x,
                    var set of $$X: s,
                    var set of $$Y: t)

2.  function var set of $$X: roots(array [$$X] of var $$Y: x,
                                   var set of $$Y: t)
```

1. Requires that `x[i]` in `t` for all `i` in `s`
2. Returns `s` such that `x[i]` in `t` for all `i` in `s`

4.2.2.8 Mathematical constraints

```
1.  function var $$E: arg_max(array [$$E] of var $$T: x)

2.  function var $$E: arg_max(array [$$E] of var bool: x)

3.  function var $$E: arg_max(array [$$E] of var float: x)

4.  function var $$E: arg_max(array [$$E] of var opt $$T: x)

5.  function var $$E: arg_max(array [$$E] of var opt bool: x)

6.  function var $$E: arg_max(array [$$E] of var opt float: x)
```

1-3.
Returns the index of the maximum value in the array `x`. When breaking ties the least index is returned.

4-6.
Returns the index of the maximum non-absent value in the array `x`. When breaking ties the least index is returned.

```
function var opt $$E: arg_max_weak(array [$$E] of var opt $$T: x)
function var opt $$E: arg_max_weak(array [$$E] of var opt bool: x)
function var opt $$E: arg_max_weak(array [$$E] of var opt float: x)
```

Returns the index of the maximum non-absent value in the array `x`. When breaking ties the least index is returned. Returns absent when all elements are absent.

```
1.  function var $$E: arg_min(array [$$E] of var $$T: x)

2.  function var $$E: arg_min(array [$$E] of var bool: x)

3.  function var $$E: arg_min(array [$$E] of var float: x)
```

```
4.  function var $$E: arg_min(array [$$E] of var opt $$T: x)

5.  function var $$E: arg_min(array [$$E] of var opt bool: x)

6.  function var $$E: arg_min(array [$$E] of var opt float: x)
```

1-3.
Returns the index of the minimum value in the array x. When breaking ties the least index is returned.

4-6.
Returns the index of the minimum non-absent value in the array x. When breaking ties the least index is returned.

```
function var opt $$E: arg_min_weak(array [$$E] of var opt $$T: x)
function var opt $$E: arg_min_weak(array [$$E] of var opt bool: x)
function var opt $$E: arg_min_weak(array [$$E] of var opt float: x)
```

Returns the index of the minimum non-absent value in the array x. When breaking ties the least index is returned. Returns absent when all elements are absent.

```
function $$E: arg_val(array [$$E] of $T: x, $T: v)
function var $$E: arg_val(array [$$E] of var bool: x, var bool: v)
function var $$E: arg_val(array [$$E] of var opt bool: x,
                          var opt bool: v)
function var $$E: arg_val(array [$$E] of var float: x, var float: v)
function var $$E: arg_val(array [$$E] of var opt float: x,
                          var float: v)
function var $$E: arg_val(array [$$E] of var $$V: x, var $$V: v)
function var $$E: arg_val(array [$$E] of var opt $$V: x,
                          var opt $$V: v)
```

Returns the index of the value v in the array x. When breaking ties the least index is returned. Note that this function forces the value v to occur in x.

```
function opt $$E: arg_val_weak(array [$$E] of $T: x, $T: v)
function var opt $$E: arg_val_weak(array [$$E] of var bool: x,
                                   var bool: v)
function var opt $$E: arg_val_weak(array [$$E] of var opt bool: x,
                                   var opt bool: v)
function var opt $$E: arg_val_weak(array [$$E] of var float: x,
                                   var float: v)
function var opt $$E: arg_val_weak(array [$$E] of var opt float: x,
                                   var opt float: v)
function var opt $$E: arg_val_weak(array [$$E] of var $$V: x,
                                   var $$V: v)
function var opt $$E: arg_val_weak(array [$$E] of var opt $$V: x,
                                   var opt $$V: v)
```

Returns the index of the value `v` in the array `x` when `v`. When breaking ties the least index is returned. Returns absent when `v` does not occur in `x`.

```
predicate maximum(var $$E: m, array [$X] of var $$E: x)
predicate maximum(var float: m, array [$X] of var float: x)
```

Constrains `m` to be the maximum of the values in `x`.

Assumptions: |`x`| > 0.

```
1.  predicate maximum_arg(array [$$E] of var $$T: x, var $$E: i)

2.  predicate maximum_arg(array [$$E] of var bool: x, var $$E: i)

3.  predicate maximum_arg(array [$$E] of var float: x, var $$E: i)
```

1, 2.

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

3. 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 minimum(var float: m, array [$X] of var float: x)
predicate minimum(var $$E: m, array [$X] of var $$E: x)
```

Constrains `m` to be the minimum of the values in `x`.

Assumptions: |`x`| > 0.

```
predicate minimum_arg(array [$$E] of var $$T: x, var $$E: i)
predicate minimum_arg(array [$$E] of var bool: x, var $$E: i)
predicate minimum_arg(array [$$E] of var float: x, var $$E: 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

```
1.  function var float: piecewise_linear(var float: x,
                                         array [int] of float: xi,
                                         array [int] of float: vi)
```

```
2.  predicate piecewise_linear(var float: x,
                               var float: y,
                               array [int] of float: xi,
                               array [int] of float: vi)

3.  function var float: piecewise_linear(var float: x,
                                         array [int] of float: x_start,
                                         array [int] of float: x_end,
                                         array [int] of float: v_start,
                                         array [int] of float: v_end)

4.  predicate piecewise_linear(var float: x,
                               var float: y,
                               array [int] of float: x_start,
                               array [int] of float: x_end,
                               array [int] of float: v_start,
                               array [int] of float: v_end)
```

1. Return the piecewise-linear function of `x` on the given point-value sequence. The array `xi` defines the points, and `vi` the corresponding values.
2. Constrains `y`(`x`) to be the piecewise-linear function on the provided point-value sequence. The array `xi` defines the points, and `vi` the corresponding values.
3. Return the piecewise-linear function of `x` on the given (possibly disconnected) intervals. Each interval `i` connects (`x_start`[`i`], `v_start`[`i`]) to (`x_end`[`i`], `v_end`[`i`]).
4. Constrains `y`(`x`) to be the piecewise-linear function on the given (possibly disconnected) intervals. Each interval `i` connects (`x_start`[`i`], `v_start`[`i`]) to (`x_end`[`i`], `v_end`[`i`]).

```
1.  predicate range(array [$$X] of var $$Y: x,
                    var set of $$X: s,
                    var set of $$Y: t)

2.  function var set of int: range(array [int] of var int: x,
                                   var set of int: s)
```

1. 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.
2. 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 sliding_sum(int: low,
                      int: up,
                      int: seq,
                      array [$$E] of var int: vs)
```

Requires that in each subsequence `vs`[`i`], ..., `vs`[`i` + `seq` - 1] the sum of the values belongs to the interval [`low`, `up`].

```
predicate sum_pred(var $$X: i,
                   array [$$X] of set of $$Y: sets,
                   array [$$Y] of int: cs,
                   var int: s)
```

Requires that the sum of `cs[i1]..cs[iN]` equals `s`, where `i1..iN` 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.

```
predicate sum_set(array [$$X] of $$Y: vs,
                  array [$$X] of int: ws,
                  var set of $$Y: x,
                  var int: s)
```

Requires that the sum of the weights `ws[i1]..ws[iN]` equals `s`, where `vs[i1]..vs[iN]` are the elements appearing in set `x`

4.2.2.9 Packing constraints

```
1.  predicate bin_packing(int: c,
                          array [$$I] of var $$B: bin,
                          array [$$I] of int: w)

2.  predicate bin_packing(int: c,
                          array [$$I] of var opt $$B: bin,
                          array [$$I] of int: w)
```

1. 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]` >=0
   - `c` >=0

2. 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`. An item may not be put into any bin, in which case `bin[i]` is absent.

   Assumptions:

   - forall `i`, `w[i]` >=0
   - `c` >=0

```
1.  predicate bin_packing_capa(array [$$B] of int: c,
                               array [$$I] of var $$B: bin,
                               array [$$I] of int: w)

2.  predicate bin_packing_capa(array [$$B] of int: c,
                               array [$$I] of var opt $$B: bin,
                               array [$$I] of int: w)
```

1. 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]` >=0
   - forall `b`, `c[b]` >=0

2. 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]`. An item may not be put into any bin, in which case `bin[i]` is absent.

   Assumptions:

   - forall `i`, `w[i]` >=0
   - forall `b`, `c[b]` >=0

```
1.  predicate bin_packing_load(array [$$B] of var int: load,
                               array [$$I] of var $$B: bin,
                               array [$$I] of int: w)

2.  predicate bin_packing_load(array [$$B] of var int: load,
                               array [$$I] of var opt $$B: bin,
                               array [$$I] of int: w)

3.  function array [$$B] of var int: bin_packing_load(array [$$I] of var $$B: bin,
                                                      array [$$I] of int: w)

4.  function array [$$B] of var int: bin_packing_load(array [$$I] of var opt $$B:␣
↪bin,
                                                      array [$$I] of int: w)
```

1. 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]` >=0

2. 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]`. An item may not be put into any bin, in which case `bin[i]` is absent.

   Assumptions:

   - forall `i`, `w[i]` >=0

3. 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]` >=0

4. 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. An item may not be put into any bin, in which case `bin[i]` is absent.

   Assumptions:

   - forall `i`, `w[i]` >=0

```
predicate diffn(array [$$E] of var int: x,
                array [$$E] of var int: y,
                array [$$E] of var int: dx,
                array [$$E] 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 [$$E,$$F] of var int: box_posn,
                  array [$$E,$$F] of var int: box_size)
```

Constrains `k`-dimensional boxes to be non-overlapping. For each box `i` and dimension `j`, `box_posn[i, j]` is the base position of the box in dimension `j`, and `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 [$$E] of var int: x,
                          array [$$E] of var int: y,
                          array [$$E] of var int: dx,
                          array [$$E] 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 [$$E,$$F] of var int: box_posn,
                            array [$$E,$$F] of var int: box_size)
```

Constrains `k`-dimensional boxes to be non-overlapping. For each box `i` and dimension `j`, `box_posn[i, j]` is the base position of the box in dimension `j`, and `box_size[i, j]` is the size in that dimension. Boxes whose size is 0 in at least one dimension can be packed anywhere.

```
1.  predicate geost(array [$$R,$$K] of int: rect_size,
                    array [$$R,$$K] of int: rect_offset,
                    array [$$S] of set of $$R: shape,
                    array [$$O,$$K] of var int: x,
                    array [$$O] of var $$S: kind)

2.  predicate geost(int: k,
                    array [$$R,$$K] of int: rect_size,
                    array [$$R,$$K] of int: rect_offset,
                    array [$$S] of set of $$R: shape,
                    array [$$O,$$K] of var int: x,
                    array [$$O] of var $$S: kind)
```

1. A global non-overlap constraint for `k` dimensional objects. It enforces that no two objects overlap.

   Parameters:

   - `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.
   - `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.

2. 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.
   - `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(array [$$R,$$K] of int: rect_size,
                   array [$$R,$$K] of int: rect_offset,
                   array [$$S] of set of $$R: shape,
                   array [$$O,$$K] of var int: x,
                   array [$$O] of var $$S: kind,
                   array [$$K] of var int: l,
                   array [$$K] of var int: u)
predicate geost_bb(int: k,
                   array [$$R,$$K] of int: rect_size,
```

```
                    array [$$R,$$K] of int: rect_offset,
                    array [$$S] of set of $$R: shape,
                    array [$$O,$$K] of var int: x,
                    array [$$O] of var $$S: kind,
                    array [$$K] of var int: l,
                    array [$$K] 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.
- `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_nonoverlap_k(array [$$K] of var int: x1,
                             array [$$K] of int: w1,
                             array [$$K] of var int: x2,
                             array [$$K] of int: w2)
```

A non-overlap constraint for two `k` dimensional objects. It enforces that there is at least one dimension where the objects occupy a different space.

Parameters:

- `x1`: coordinate of each dimension for the first object
- `w1`: width of each dimension for the first object
- `x2`: coordinate of each dimension for the second object
- `w2`: width of each dimension for the second object

```
1.  predicate geost_smallest_bb(array [$$R,$$K] of int: rect_size,
                                array [$$R,$$K] of int: rect_offset,
                                array [$$S] of set of $$R: shape,
                                array [$$O,$$K] of var int: x,
                                array [$$O] of var $$S: kind,
                                array [$$K] of var int: l,
                                array [$$K] of var int: u)
```

```
2.  predicate geost_smallest_bb(int: k,
                                array [$$R,$$K] of int: rect_size,
                                array [$$R,$$K] of int: rect_offset,
                                array [$$S] of set of $$R: shape,
                                array [$$O,$$K] of var int: x,
                                array [$$O] of var $$S: kind,
                                array [$$K] of var int: l,
                                array [$$K] of var int: u)
```

1. 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:

   - `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.
   - `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`.

2. 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.
   - `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 [$$I] of int: w,
                   array [$$I] of int: p,
                   array [$$I] 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

4.2.2.10 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. The task with start time `s0` and duration `d0` spans the optional tasks with start times `s[i]` and durations `d[i]`), and at most one of those tasks can occur.

```
1.  predicate cumulative(array [$$E] of var int: s,
                         array [$$E] of var int: d,
                         array [$$E] of var int: r,
                         var int: b)

2.  predicate cumulative(array [$$E] of var opt int: s,
                         array [$$E] of var int: d,
                         array [$$E] of var int: r,
                         var int: b)
```

1. 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

2. 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]` >= 0 and `r[i]` >= 0

```
1.  predicate cumulatives(array [$$E] of var int: s,
                          array [$$E] of var int: d,
                          array [$$E] of var int: r,
                          array [$$E] of var $$M: m,
                          array [$$M] of var int: b,
                          bool: upper)

2.  predicate cumulatives(array [$$E] of var opt int: s,
                          array [$$E] of var int: d,
                          array [$$E] of var int: r,
                          array [$$E] of var $$M: m,
                          array [$$M] of var int: b,
                          bool: upper)
```

1. We have a set of tasks given by start times `s`, durations `d`, resource requirements `r`, and machines `m`. We also have a set of machines given by resource bounds `b`. Finally, we have a Boolean flag `upper`. The constraint requires that forall `i`, `m[i]` is in the index set of `b` and that for each machine `j` and time instant `t`, either no task assigned to `j` executes at `t`, or the total resource requirement of such tasks is not greater than, if `upper` is true, or not less than, if `upper` is false, the given bound `b[j]`. Resource requirements can be positive (for consumption) or negative (for production).

   Assumptions:

   - forall `i`, `d[i]` >= 0

2. We have a set of optional tasks given by start times `s`, durations `d`, resource requirements `r`, and machines `m`. We also have a set of machines given by resource bounds `b`. Finally, we have a Boolean flag `upper`. The constraint requires that forall `i`, `m[i]` is in the index set of `b` and that for each machine `j` and time instant `t`, either no task assigned to `j` executes at `\t`, or the total resource requirement of such tasks is not greater than, if `upper` is true, or not less than, if `upper` is false, the given bound `b[j]`. Resource requirements can be positive (for consumption) or negative (for production).

   Assumptions:

   - forall `i`, `d[i]` >= 0

```
1.  predicate disjunctive(array [$$T] of var int: s,
                          array [$$T] of var int: d)

2.  predicate disjunctive(array [$$T] of var opt int: s,
                          array [$$T] of var int: d)
```

1. 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]` >= 0

2. 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]` >= 0

```
1.  predicate disjunctive_strict(array [$$T] of var int: s,
                                 array [$$T] of var int: d)

2.  predicate disjunctive_strict(array [$$T] of var opt int: s,
                                 array [$$T] of var int: d)
```

1. 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]` >= 0

2. 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]` >= 0

```
predicate span(var opt int: s0,
               var int: d0,
               array [$$E] of var opt int: s,
               array [$$E] of var int: d)
```

Span constraint for optional tasks. Task (`s0`,`d0`) spans the optional tasks (`s[i]`,`d[i]`) in the array arguments.

4.2.2.11 Graph constraints

Many of these constraints operate on graphs that are represented as lists of edges. The constraints take a given (fixed) graph as arguments, which is represented using two arrays, `from` and `to`, representing the graph with edges `(from[i],to[i])`, where `from[i]` and `to[i]` are nodes of the graph.

```
1.  predicate bounded_dpath(int: N,
                            int: E,
                            array [int] of int: from,
                            array [int] of int: to,
                            array [int] of int: w,
                            var int: s,
                            var int: t,
                            array [int] of var bool: ns,
                            array [int] of var bool: es,
                            var int: K)

2.  predicate bounded_dpath(array [int] of $$N: from,
                            array [int] of $$N: to,
                            array [int] of int: w,
                            var $$N: s,
                            var $$N: t,
                            array [$$N] of var bool: ns,
                            array [int] of var bool: es,
                            var int: K)
```

1. Constrains the subgraph `ns` and `es` of a given directed graph to be a path from `s` to `t` of weight `K`.

   Parameters:

   - `N`: the number of nodes in the given graph
   - `E`: the number of edges in the given graph
   - `from`: the leaving node 1..`N` for each edge
   - `to`: the entering node 1..`N` for each edge
   - `w`: the weight of each edge
   - `s`: the source node (which may be variable)
   - `t`: the dest node (which may be variable)
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph
   - `K`: the cost of the path

2. Constrains the subgraph `ns` and `es` of a given directed graph to be a path from `s` to `t` of weight `K`.

   Parameters:

   - `from`: the leaving node for each edge
   - `to`: the entering node for each edge

- w: the weight of each edge
- s: the source node (which may be variable)
- t: the dest node (which may be variable)
- ns: a Boolean for each node whether it is in the subgraph
- es: a Boolean for each edge whether it is in the subgraph
- K: the cost of the path

```
1.  predicate bounded_path(int: N,
                           int: E,
                           array [int] of int: from,
                           array [int] of int: to,
                           array [int] of int: w,
                           var int: s,
                           var int: t,
                           array [int] of var bool: ns,
                           array [int] of var bool: es,
                           var int: K)

2.  predicate bounded_path(array [int] of $$N: from,
                           array [int] of $$N: to,
                           array [int] of int: w,
                           var $$N: s,
                           var $$N: t,
                           array [$$N] of var bool: ns,
                           array [int] of var bool: es,
                           var int: K)
```

1. Constrains the subgraph ns and es of a given undirected graph to be a path from s to t of weight K.

   Parameters:

   - N: the number of nodes in the given graph
   - E: the number of edges in the given graph
   - from: the leaving node 1..N for each edge
   - to: the entering node 1..N for each edge
   - w: the weight of each edge
   - s: the source node (which may be variable)
   - t: the dest node (which may be variable)
   - ns: a Boolean for each node whether it is in the subgraph
   - es: a Boolean for each edge whether it is in the subgraph
   - K: the cost of the path

2. Constrains the subgraph ns and es of a given undirected graph to be a path from s to t of weight K.

   Parameters:

- `from`: the leaving node for each edge
- `to`: the entering node for each edge
- `w`: the weight of each edge
- `s`: the source node (which may be variable)
- `t`: the dest node (which may be variable)
- `ns`: a Boolean for each node whether it is in the subgraph
- `es`: a Boolean for each edge whether it is in the subgraph
- `K`: the cost of the path

```
1. predicate circuit(array [$$E] of var $$E: x)

2. predicate circuit(array [$$E] of var opt $$E: x)
```

1. Constrains the elements of `x` to define a circuit where `x[i] = j` means that `j` is the successor of `i`.
2. Constrains the elements of `x` to define a circuit where `x[i] = j` means that `j` is the successor of `i`. Absent elements do not take part in the circuit.

```
predicate connected(array [$$E] of $$N: from,
                    array [$$E] of $$N: to,
                    array [$$N] of var bool: ns,
                    array [$$E] of var bool: es)
```

Constrains the subgraph `ns` and `es` of a given undirected graph to be connected.

Parameters:

- `from`: is the leaving node for each edge
- `to`: is the entering node for each edge
- `ns`: is a Boolean for each node whether it is in the subgraph
- `es`: is a Boolean for each edge whether it is in the subgraph

---

```
1. predicate d_weighted_spanning_tree(array [$$E] of $$N: from,
                                      array [$$E] of $$N: to,
                                      array [$$E] of int: w,
                                      var $$N: r,
                                      array [$$E] of var bool: es,
                                      var int: K)

2. predicate d_weighted_spanning_tree(int: N,
                                      int: E,
                                      array [int] of int: from,
                                      array [int] of int: to,
```

```
                                        array [int] of int: w,
                                        var int: r,
                                        array [int] of var bool: es,
                                        var int: K)
```

1. Constrains the set of edges `es` of a given directed graph to be a weighted spanning tree rooted at `r` of weight `W`.

   Parameters:

   - `from`: the leaving node for each edge
   - `to`: the entering node each edge
   - `w`: the weight of each edge
   - `r`: the root node (which may be variable)
   - `es`: a Boolean for each edge whether it is in the subgraph
   - `K`: the weight of the tree

2. Constrains the set of edges `es` of a given directed graph to be a weighted spanning tree rooted at `r` of weight `W`.

   Parameters:

   - `N`: the number of nodes in the given graph
   - `E`: the number of edges in the given graph
   - `from`: the leaving node 1..`N` for each edge
   - `to`: the entering node 1..`N` for each edge
   - `w`: the weight of each edge
   - `r`: the root node (which may be variable)
   - `es`: a Boolean for each edge whether it is in the subgraph
   - `K`: the weight of the tree

```
predicate dag(array [$$E] of $$N: from,
              array [$$E] of $$N: to,
              array [$$N] of var bool: ns,
              array [$$E] of var bool: es)
```

Constrains the subgraph `ns` and `es` of a given directed graph to be a DAG.

Parameters:

- `from`: the leaving node for each edge
- `to`: the entering node for each edge
- `ns`: a Boolean for each node whether it is in the subgraph
- `es`: a Boolean for each edge whether it is in the subgraph

```
predicate dconnected(array [$$E] of $$N: from,
                     array [$$E] of $$N: to,
                     array [$$N] of var bool: ns,
                     array [$$E] of var bool: es)
```

Constrains the subgraph `ns` and `es` of a given directed graph to be connected.

Parameters:

- `from`: the leaving node for each edge
- `to`: the entering node for each edge
- `ns`: a Boolean for each node whether it is in the subgraph
- `es`: a Boolean for each edge whether it is in the subgraph

```
1.  predicate dpath(int: N,
                    int: E,
                    array [int] of int: from,
                    array [int] of int: to,
                    var int: s,
                    var int: t,
                    array [int] of var bool: ns,
                    array [int] of var bool: es)

2.  predicate dpath(array [int] of $$N: from,
                    array [int] of $$N: to,
                    var $$N: s,
                    var $$N: t,
                    array [$$N] of var bool: ns,
                    array [int] of var bool: es)
```

1. Constrains the subgraph `ns` and `es` of a given directed graph to be a path from `s` to `t`.

   Parameters:

   - `N`: the number of nodes in the given graph
   - `E`: the number of edges in the given graph
   - `from`: the leaving node 1..`N` for each edge
   - `to`: the entering node 1..`N` for each edge
   - `s`: the source node (which may be variable)
   - `t`: the dest node (which may be variable)
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph
2. Constrains the subgraph `ns` and `es` of a given directed graph to be a path from `s` to `t`.

   Parameters:

- from: the leaving node for each edge
- to: the entering node for each edge
- s: the source node (which may be variable)
- t: the dest node (which may be variable)
- ns: a Boolean for each node whether it is in the subgraph
- es: a Boolean for each edge whether it is in the subgraph

```
1.  predicate dreachable(int: N,
                         int: E,
                         array [int] of int: from,
                         array [int] of int: to,
                         var int: r,
                         array [int] of var bool: ns,
                         array [int] of var bool: es)

2.  predicate dreachable(array [$$E] of $$N: from,
                         array [$$E] of $$N: to,
                         var $$N: r,
                         array [$$N] of var bool: ns,
                         array [$$E] of var bool: es)
```

1. Constrains the subgraph ns and es of a given directed graph to be reachable from r.

   Parameters:

   - N: the number of nodes in the given graph
   - E: the number of edges in the given graph
   - from: the leaving node 1..N for each edge
   - to: the entering node 1..N for each edge
   - r: the root node (which may be variable)
   - ns: a Boolean for each node whether it is in the subgraph
   - es: a Boolean for each edge whether it is in the subgraph
2. Constrains the subgraph ns and es of a given directed graph to be reachable from r.

   Parameters:

   - from: the leaving node for each edge
   - to: the entering node for each edge
   - r: the root node (which may be variable)
   - ns: a Boolean for each node whether it is in the subgraph
   - es: a Boolean for each edge whether it is in the subgraph

---

```
predicate dsteiner(int: N,
                   int: E,
                   array [int] of int: from,
                   array [int] of int: to,
                   array [int] of int: w,
                   var int: r,
                   array [int] of var bool: ns,
                   array [int] of var bool: es,
                   var int: K)
```

Constrains the subgraph `ns` and `es` of a given directed graph to be a weighted spanning tree rooted at `r` of weight `W`.

Parameters:

- `N`: the number of nodes in the given graph
- `E`: the number of edges in the given graph
- `from`: the leaving node 1..`N` for each edge
- `to`: the entering node 1..`N` for each edge
- `w`: the weight of each edge
- `r`: the root node (which may be variable)
- `ns`: a Boolean for each node whether it is in the subgraph
- `es`: a Boolean for each edge whether it is in the subgraph
- `K`: the weight of the tree

```
1.  predicate dtree(int: N,
                    int: E,
                    array [int] of int: from,
                    array [int] of int: to,
                    var int: r,
                    array [int] of var bool: ns,
                    array [int] of var bool: es)

2.  predicate dtree(array [$$E] of $$N: from,
                    array [$$E] of $$N: to,
                    var $$N: r,
                    array [$$N] of var bool: ns,
                    array [$$E] of var bool: es)
```

1. Constrains the subgraph `ns` and `es` of a given directed graph to be a tree rooted at `r`.

   Parameters:

   - `N`: the number of nodes in the given graph
   - `E`: the number of edges in the given graph
   - `from`: the leaving node 1..`N` for each edge

- `to`: the entering node 1..`N` for each edge
- `r`: the root node (which may be variable)
- `ns`: a Boolean for each node whether it is in the subgraph
- `es`: a Boolean for each edge whether it is in the subgraph

2. Constrains the subgraph `ns` and `es` of a given directed graph to be at tree rooted at `r`.

   Parameters:

   - `from`: the leaving node for each edge
   - `to`: the entering node for each edge
   - `r`: the root node (which may be variable)
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph

---

```
predicate network_flow(array [$$E,1..2] of $$N: arc,
                       array [$$N] of int: balance,
                       array [$$E] 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 amount of flow added or removed by each node. If the balance is positive then flow is added by the node, and it is removed when it is negative, i.e., `balance = output - input`.
- `flow`: the flow going through each arc.

```
predicate network_flow_cost(array [$$E,1..2] of $$N: arc,
                            array [$$N] of int: balance,
                            array [$$E] of int: weight,
                            array [$$E] 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 amount of flow added or removed by each node. If the balance is positive then flow is added by the node, and it is removed when it is negative, i.e., `balance = output - input`.
- `weight`: the unit cost of the flow through the arc.
- `flow`: the flow going through each arc.

- cost: the overall cost of the flow.

```
1.  predicate path(int: N,
                   int: E,
                   array [int] of int: from,
                   array [int] of int: to,
                   var int: s,
                   var int: t,
                   array [int] of var bool: ns,
                   array [int] of var bool: es)

2.  predicate path(array [int] of $$N: from,
                   array [int] of $$N: to,
                   var $$N: s,
                   var $$N: t,
                   array [$$N] of var bool: ns,
                   array [int] of var bool: es)
```

1. Constrains the subgraph ns and es of a given undirected graph to be a path from s to t.

   Parameters:

   - N: the number of nodes in the given graph
   - E: the number of edges in the given graph
   - from: the leaving node 1..N for each edge
   - to: the entering node 1..N for each edge
   - s: the source node (which may be variable)
   - t: the dest node (which may be variable)
   - ns: a Boolean for each node whether it is in the subgraph
   - es: a Boolean for each edge whether it is in the subgraph

2. Constrains the subgraph ns and es of a given undirected graph to be a path from s to t.

   Parameters:

   - from: the leaving node for each edge
   - to: the entering node for each edge
   - s: the source node (which may be variable)
   - t: the dest node (which may be variable)
   - ns: a Boolean for each node whether it is in the subgraph
   - es: a Boolean for each edge whether it is in the subgraph

---

```
1.  predicate reachable(int: N,
                        int: E,
                        array [int] of int: from,
                        array [int] of int: to,
                        var int: r,
                        array [int] of var bool: ns,
                        array [int] of var bool: es)

2.  predicate reachable(array [$$E] of $$N: from,
                        array [$$E] of $$N: to,
                        var $$N: r,
                        array [$$N] of var bool: ns,
                        array [$$E] of var bool: es)
```

1. Constrains the subgraph `ns` and `es` of a given undirected graph to be reachable from `r`.

   Parameters:

   - `N`: the number of nodes in the given graph
   - `E`: the number of edges in the given graph
   - `from`: the leaving node 1..`N` for each edge
   - `to`: the entering node 1..`N` for each edge
   - `r`: the root node (which may be variable)
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph
2. Constrains the subgraph `ns` and `es` of a given undirected graph to be reachable from `r`.

   Parameters:

   - `from`: the leaving node for each edge
   - `to`: the entering node for each edge
   - `r`: the root node (which may be variable)
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph

---

```
predicate steiner(int: N,
                  int: E,
                  array [int] of int: from,
                  array [int] of int: to,
                  array [int] of int: w,
                  array [int] of var bool: ns,
                  array [int] of var bool: es,
                  var int: K)
```

Constrains the set of edges `es` of a given undirected graph to be a weighted spanning tree of weight `W`.

-

Parameters:

- `N`: the number of nodes in the given graph
- `E`: the number of edges in the given graph
- `from`: the leaving node 1..`N` for each edge
- `to`: the entering node 1..`N` for each edge
- `w`: the weight of each edge
- `ns`: a Boolean for each node whether it is in the subgraph
- `es`: a Boolean for each edge whether it is in the subgraph
- `K`: the weight of the tree

```
predicate subcircuit(array [$$E] of var $$E: 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.

```
1.  predicate subgraph(int: N,
                       int: E,
                       array [int] of int: from,
                       array [int] of int: to,
                       array [int] of var bool: ns,
                       array [int] of var bool: es)

2.  predicate subgraph(array [$$E] of $$N: from,
                       array [$$E] of $$N: to,
                       array [$$N] of var bool: ns,
                       array [$$E] of var bool: es)
```

1. Constrains that `ns` and `es` is a subgraph of a given directed graph.

   Parameters:

   - `N`: the number of nodes in the given graph
   - `E`: the number of edges in the given graph
   - `from`: the leaving node 1..`N` for each edge
   - `to`: the entering node 1..`N` for each edge
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph

2. Constrains that `ns` and `es` is a subgraph of a given directed graph.

   Parameters:

   - `from`: the leaving node for each edge
   - `to`: the entering node for each edge
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph

---

```
1.  predicate tree(int: N,
                   int: E,
                   array [int] of int: from,
                   array [int] of int: to,
                   var int: r,
                   array [int] of var bool: ns,
                   array [int] of var bool: es)

2.  predicate tree(array [$$E] of $$N: from,
                   array [$$E] of $$N: to,
                   var $$N: r,
                   array [$$N] of var bool: ns,
                   array [$$E] of var bool: es)
```

1. Constrains the subgraph `ns` and `es` of a given undirected graph to be a tree rooted at `r`.

   Parameters:

   - `N`: the number of nodes in the given graph
   - `E`: the number of edges in the given graph
   - `from`: the leaving node 1..`N` for each edge
   - `to`: the entering node 1..`N` for each edge
   - `r`: the root node (which may be variable)
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph
2. Constrains the subgraph `ns` and `es` of a given undirected graph to be at tree rooted at `r`.

   Parameters:

   - `from`: the leaving node for each edge
   - `to`: the entering node for each edge
   - `r`: the root node (which may be variable)
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph

---

```
predicate weighted_spanning_tree(array [$$E] of int: from,
                                 array [$$E] of int: to,
                                 array [$$E] of int: w,
                                 array [$$E] of var bool: es,
                                 var int: K)
predicate weighted_spanning_tree(int: N,
                                 int: E,
                                 array [int] of int: from,
                                 array [int] of int: to,
                                 array [int] of int: w,
                                 array [int] of var bool: es,
                                 var int: K)
```

Constrains the set of edges `es` of a given undirected graph to be a weighted spanning tree of weight `W`.

Parameters:

- `N`: the number of nodes in the given graph
- `E`: the number of edges in the given graph
- `from`: the leaving node 1..`N` for each edge
- `to`: the entering node 1..`N` for each edge
- `w`: the weight of each edge
- `es`: a Boolean for each edge whether it is in the subgraph
- `K`: the weight of the tree

4.2.2.12 Extensional constraints (table, regular etc.)

```
predicate cost_mdd(array [int] of var int: x,
                   int: N,
                   array [int] of int: level,
                   int: E,
                   array [int] of int: from,
                   array [int] of set of int: label,
                   array [int] of int: cost,
                   array [int] of int: to,
                   var int: totalcost)
```

Requires that `x` defines a path in the cost MDD with total edge weight `totalcost`.

Parameters:

- `N`: the number of nodes, the root node is node 1
- `level`: the level of each node, the root is level 1, T is level `length`(x)+1
- `E`: the number of edges
- `from`: the leaving node (1..`N`)for each edge
- `label`: the set of value of the x variable for each edge

- `cost`: the cost for each edge
- `to`: the entering node for each edge, where 0 = T node
- `totalcost`: the total cost of the path defined by `x`

```
predicate cost_regular(array [int] of var int: x,
                       int: Q,
                       int: S,
                       array [int,int] of int: d,
                       int: q0,
                       set of int: F,
                       array [int,int] of int: c,
                       var int: C)
```

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`) -> 0..`Q`)) and initial state `q0` (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. Each edge has an associated cost `c`, and `C` is the sum of costs taken on the accepting path for `x`.

```
predicate mdd(array [int] of var int: x,
              int: N,
              array [int] of int: level,
              int: E,
              array [int] of int: from,
              array [int] of set of int: label,
              array [int] of int: to)
```

Requires that `x` defines a path from root to true node T through the (deterministic) Multivalued Decision Diagram (MDD) defined by the parameters.

The MDD must be deterministic, i.e., there cannot be two edges with the same label leaving the same node.

Parameters:

- `N`: the number of nodes, the root node is node 1
- `level`: the level of each node, the root is level 1, T is level `length`(x)+1
- `E`: the number of edges
- `from`: the leaving node (1..`N`)for each edge
- `label`: the set of values of the `x` variable for each edge
- `to`: the entering node for each edge, where 0 = T node

```
predicate mdd_nondet(array [int] of var int: x,
                     int: N,
                     array [int] of int: level,
                     int: E,
```

```
                    array [int] of int: from,
                    array [int] of set of int: label,
                    array [int] of int: to)
```

Requires that `x` defines a path from root to true node T through the (nondeterministic) Multivalued Decision Diagram (MDD) defined by the parameters.

The MDD can be nondeterministic, i.e., there can be two edges with the same label leaving the same node.

Parameters:

- `N`: the number of nodes, the root node is node 1
- `level`: the level of each node, the root is level 1, T is level `length`(x)+1
- `E`: the number of edges
- `from`: the leaving node (1..`N`)for each edge
- `label`: the set of values of the `x` variable for each edge
- `to`: the entering node for each edge, where 0 = T node

```
1.  predicate regular(array [$$X] of var $$Val: x,
                      array [$$State,$$Val] of opt $$State: d,
                      $$State: q0,
                      set of $$State: F)

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

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

4.  predicate regular(array [int] of var $$E: x, string: r)
```

1. The sequence of values in array `x` is accepted by the DFA with transition function `d`, initial state `q0` and accepting states `F`. The transition function maps states (from enumerated type `State`) and values (from type `Val`) to states, or the absent value if the transition is not possible.
2. 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`) -> 0..`Q`)) and initial state `q0` (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.

3. The sequence of values in array `x` (which must all be in the set `S`) is accepted by the DFA of `Q` states with input `S` and transition function `d` (which maps (1..`Q`, `S`) -> 0..`Q`)) and initial state `q0` (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.
4. 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.

```
1.  predicate regular_nfa(array [$$X] of var $$Val: x,
                          array [$$State,$$Val] of set of $$State: d,
                          $$State: q0,
                          set of $$State: F)

2.  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)

3.  predicate regular_nfa(array [int] of var int: x,
                          int: Q,
                          set of int: S,
                          array [int,int] of set of int: d,
                          int: q0,
```

```
                                set of int: F)
```

1. The sequence of values in array `x` is accepted by the NFA with transition function `d`, initial state `q0` and accepting states `F`. The transition function maps states (from enumerated type `State`) and values (from type `Val`) to sets of states.
2. 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`).
3. The sequence of values in array `x` (which must all be in the range `S`) is accepted by the NFA of `Q` states with input `S` and transition function `d` (which maps (1..`Q`, `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`).

```
1.  predicate table(array [$$E] of var bool: x, array [$$X,$$E] of bool: t)

2.  predicate table(array [$$E] of var int: x, array [$$X,$$E] of int: t)

3.  predicate table(array [$$E] of var opt int: x,
                    array [$$X,$$E] of opt int: t)
```

1, 2.
Represents the constraint `x` in `t` where we consider each row in `t` to be a tuple and `t` as a set of tuples.

3. Represents the constraint `x` in `t` if the variable `x_i` and the value `t_i` occur where we consider each row in `t` to be a tuple and `t` as a set of tuples.

4.2.2.13 Machine learning constraints

```
predicate neural_net(array [int] of var float: inputs,
                     array [int] of int: input_ids,
                     array [int] of var float: outputs,
                     array [int] of int: output_ids,
                     array [int] of float: bias,
                     array [int] of float: edge_weight,
                     array [int] of int: edge_parent,
                     array [int] of int: first_edge,
                     NEURON_TYPE: neuron_type)
```

Constrain the output layer of a neural net to take the value defined by the input layer.

Parameters:

- `inputs`: an array of float variables
- `input_ids`: array[int] of node
- `outputs`: an array of float variables

- `output_ids`: array[int] of node
- `bias`: array[node] of float
- `edge_weight`: array[edge] of float
- `edge_parent`: array[edge] of neuron (start neuron for edge)
- `first_edge`: array[node] of 1..m+1
- `neuron_type`: { NT_RELU, NT_STEP, NT_LINEAR, NT_SOFTPLUS }

4.2.2.14 Deprecated constraints

```
predicate at_least(int: n, array [$X] of var $$E: x, $$E: v)
```

Requires at least `n` variables in `x` to take the value `v`.

This constraint is deprecated. Use count(i in x)(i=v) >= n instead.

```
predicate at_most(int: n, array [$X] of var $$E: x, $$E: v)
```

Requires at most `n` variables in `x` to take the value `v`.

This constraint is deprecated. Use count(i in x)(i=v) <= n instead.

```
predicate exactly(int: n, array [$X] of var $$E: x, $$E: v)
```

Requires exactly `n` variables in `x` to take the value `v`.

This constraint is deprecated. Use count(i in x)(i=v) = n instead.

```
predicate global_cardinality_low_up(array [$X] of var int: x,
                                    array [$Y] of int: cover,
                                    array [$Y] of int: lbound,
                                    array [$Y] 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`.

This constraint is deprecated. Use global_cardinality(x, cover, lbound, ubound) instead.

```
predicate global_cardinality_low_up_closed(array [$X] of var int: x,
                                           array [$Y] of int: cover,
                                           array [$Y] of int: lbound,
                                           array [$Y] 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`.

This constraint is deprecated. Use global_cardinality_closed(x, cover, lbound, ubound) instead.

4.2.3 FlatZinc builtins

These are the standard constraints that need to be supported by FlatZinc solvers (or redefined in the redefinitions.mzn file).

4.2.3.1 Integer FlatZinc builtins

```
predicate array_int_element(var int: b,
                            array [int] of int: as,
                            var int: c)
```

Constrains `as[b] = c`

```
predicate array_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`, where `c` is rounded to the value closest to zero (i.e., truncated).

```
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 □ b) $\leftrightarrow$ r

```
predicate int_lin_eq(array [int] of int: as,
                     array [int] of var int: bs,
                     int: c)
```

Constrains $\mathbf{c} = \sum_i \mathbf{as}[i] * \mathbf{bs}[i]$

```
predicate int_lin_eq_reif(array [int] of int: as,
                          array [int] of var int: bs,
                          int: c,
                          var bool: r)
```

Constrains $\mathbf{r} \leftrightarrow (\mathbf{c} = \sum_i \mathbf{as}[i] * \mathbf{bs}[i])$

```
predicate int_lin_le(array [int] of int: as,
                     array [int] of var int: bs,
                     int: c)
```

Constrains $\sum$ as[i]*bs[i] □ c

```
predicate int_lin_le_reif(array [int] of int: as,
                          array [int] of var int: bs,
                          int: c,
                          var bool: r)
```

Constrains r $\leftrightarrow$ ($\sum$ as[i]*bs[i] □ c)

```
predicate int_lin_ne(array [int] of int: as,
                     array [int] of var int: bs,
                     int: c)
```

Constrains $\mathbf{c} \neq \sum_i \mathbf{as}[i] * \mathbf{bs}[i]$

```
predicate int_lin_ne_reif(array [int] of int: as,
                          array [int] of var int: bs,
                          int: c,
                          var bool: r)
```

Constrains $\mathbf{r} \leftrightarrow (\mathbf{c} \neq \sum_i \mathbf{as}[i] * \mathbf{bs}[i])$

```
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 ↔ (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 ≠ b

```
predicate int_ne_reif(var int: a, var int: b, var bool: r)
```

r ↔ (a ≠ 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 = $\mathbf{x}^{\mathbf{y}}$, {bf z} is constrained to `1 div pow(x, abs(y))` when $\mathbf{y} < 0$

```
predicate int_times(var int: a, var int: b, var int: c)
```

Constrains a * b = c

```
predicate set_in(var int: x, set of int: S)
```

Constrains x ∈ S

4.2.3.2 Bool FlatZinc builtins

```
predicate array_bool_and(array [int] of var bool: as, var bool: r)
```

Constrains $\mathbf{r} \leftrightarrow \bigwedge_i \mathbf{as}[i]$

```
predicate array_bool_element(var int: b,
                             array [int] of bool: as,
                             var bool: c)
```

Constrains as[b] = c

```
predicate array_bool_xor(array [int] of var bool: as)
```

Constrains $\oplus_i \mathbf{as}[i]$

```
predicate array_var_bool_element(var int: b,
                                 array [int] of var bool: as,
                                 var bool: c)
```

Constrains as[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 $\mathbf{r} \leftrightarrow \mathbf{a} \wedge \mathbf{b}$

```
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 ↔ (a = b)

```
predicate bool_le(var bool: a, var bool: b)
```

Constrains a □ b

```
predicate bool_le_reif(var bool: a, var bool: b, var bool: r)
```

Constrains r ↔ (a □ b)

```
predicate bool_lin_eq(array [int] of int: as,
                      array [int] of var bool: bs,
                      var int: c)
```

Constrains $\mathbf{c} = \sum_i \mathbf{as}[i] * \mathbf{bs}[i]$

```
predicate bool_lin_le(array [int] of int: as,
                      array [int] of var bool: bs,
                      int: c)
```

Constrains $\sum_i \mathbf{as}[i] * \mathbf{bs}[i] \leq \mathbf{c}$

```
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 $\mathbf{r} \leftrightarrow \mathbf{a} \vee \mathbf{b}$

```
predicate bool_xor(var bool: a, var bool: b, var bool: r)
```

Constrains $\mathbf{r} \leftrightarrow \mathbf{a} \oplus \mathbf{b}$

4.2.3.3 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 \ 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 ↔ (x = y)

```
predicate set_in(var int: x, var set of int: S)
```

Constrains x ∈ S

```
predicate set_in_reif(var int: x, set of int: S, var bool: r)
predicate set_in_reif(var int: x, var set of int: S, var bool: r)
```

Constrains $\mathbf{r} \leftrightarrow (\mathbf{x} \in \mathbf{S})$

```
predicate set_intersect(var set of int: x,
                        var set of int: y,
                        var set of int: r)
```

Constrains r = x ∩ y

```
predicate set_le(var set of int: x, var set of int: y)
```

Constrains x □ y (lexicographic order of the sorted lists of elements)

```
predicate set_le_reif(var set of int: x,
                      var set of int: y,
                      var bool: r)
```

Constrains $\mathbf{r} \leftrightarrow (\mathbf{x} \leq \mathbf{y})$ (lexicographic order of the sorted lists of elements)

```
predicate set_lt(var set of int: x, var set of int: y)
```

Constrains x < y (lexicographic order of the sorted lists of elements)

```
predicate set_lt_reif(var set of int: x,
                      var set of int: y,
                      var bool: r)
```

Constrains $\mathbf{r} \leftrightarrow (\mathbf{x} < \mathbf{y})$ (lexicographic order of the sorted lists of elements)

```
predicate set_ne(var set of int: x, var set of int: y)
```

Constrains x ≠ y

```
predicate set_ne_reif(var set of int: x,
                      var set of int: y,
                      var bool: r)
```

Constrains r ↔ (x ≠ y)

```
predicate set_subset(var set of int: x, var set of int: y)
```

Constrains x ⊆ y

```
predicate set_subset_reif(var set of int: x,
                          var set of int: y,
                          var bool: r)
```

Constrains $\mathbf{r} \leftrightarrow (\mathbf{x} \subseteq \mathbf{y})$

```
predicate set_superset(var set of int: x, var set of int: y)
```

Constrains x ⊇ y

```
predicate set_superset_reif(var set of int: x,
                            var set of int: y,
                            var bool: r)
```

Constrains $\mathbf{r} \leftrightarrow (\mathbf{x} \subseteq \mathbf{y})$

```
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 ∪ y

4.2.3.4 Float FlatZinc builtins

```
predicate array_float_element(var int: b,
                              array [int] of float: as,
                              var float: c)
```

Constrains as[b] = c

```
predicate array_var_float_element(var int: b,
                                  array [int] of var float: as,
                                  var float: c)
```

Constrains as[b] = c

```
predicate float_abs(var float: a, var float: b)
```

Constrains b to be the absolute value of a

```
predicate float_acos(var float: a, var float: b)
```

Constrains b = acos(a)

```
predicate float_acosh(var float: a, var float: b)
```

Constrains b = acosh(a)

```
predicate float_asin(var float: a, var float: b)
```

Constrains b = asin(a)

```
predicate float_asinh(var float: a, var float: b)
```

Constrains b = asinh(a)

```
predicate float_atan(var float: a, var float: b)
```

Constrains b = atan(a)

```
predicate float_atanh(var float: a, var float: b)
```

Constrains b = atanh(a)

```
predicate float_cos(var float: a, var float: b)
```

Constrains b = cos(a)

```
predicate float_cosh(var float: a, var float: b)
```

Constrains b = cosh(a)

```
predicate float_div(var float: a, var float: b, var float: c)
```

Constrains a / b = c

```
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 ↔ (a = b)

```
predicate float_exp(var float: a, var float: b)
```

Constrains b = exp(a)

```
predicate float_le(var float: a, var float: b)
```

Constrains a □ b

```
predicate float_le_reif(var float: a, var float: b, var bool: r)
```

Constrains r ↔ (a □ b)

```
predicate float_lin_eq(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_eq_reif(array [int] of float: as,
                            array [int] of var float: bs,
                            float: c,
                            var bool: r)
```

Constrains $\mathbf{r} \leftrightarrow (\mathbf{c} = \sum_i \mathbf{as}[i] * \mathbf{bs}[i])$

```
predicate float_lin_le(array [int] of float: as,
                       array [int] of var float: bs,
                       float: c)
```

Constrains $\sum_i \mathbf{as}[i] * \mathbf{bs}[i] \leq \mathbf{c}$

```
predicate float_lin_le_reif(array [int] of float: as,
                            array [int] of var float: bs,
                            float: c,
                            var bool: r)
```

Constrains $\mathbf{r} \leftrightarrow (\sum_i \mathbf{as}[i] * \mathbf{bs}[i] \leq \mathbf{c})$

```
predicate float_lin_lt(array [int] of float: as,
                       array [int] of var float: bs,
                       float: c)
```

Constrains $\sum_i \mathbf{as}[i] * \mathbf{bs}[i] < \mathbf{c}$

```
predicate float_lin_lt_reif(array [int] of float: as,
                            array [int] of var float: bs,
                            float: c,
                            var bool: r)
```

Constrains $\mathbf{r} \leftrightarrow (\sum_i \mathbf{as}[i] * \mathbf{bs}[i] < \mathbf{c})$

```
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 $\mathbf{r} \leftrightarrow (\mathbf{c} \neq \sum_i \mathbf{as}[i] * \mathbf{bs}[i])$

```
predicate float_ln(var float: a, var float: b)
```

Constrains b = ln(a)

```
predicate float_log10(var float: a, var float: b)
```

Constrains b = log₁₀(a)

```
predicate float_log2(var float: a, var float: b)
```

Constrains b = log₂(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 ↔ (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 ≠ b

```
predicate float_ne_reif(var float: a, var float: b, var bool: r)
```

Constrains r ↔ (a ≠ 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 = $\mathbf{x}^{\mathbf{y}}$

```
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 $\mathbf{b} = \sqrt{\mathbf{a}}$

```
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

4.2.3.5 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 $\mathbf{b} \leftrightarrow \bigvee_i \mathbf{as}[i] \vee \bigvee_j \neg \mathbf{bs}[j]$

4.2.3.6 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[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[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[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[idx] = c` This can be overridden in a solver that can perform the index calculation more efficiently than using a MiniZinc decomposition.

4.2.3.7 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 the domain of `x` using the values in `as`, using each pair of values `as[2*i-1]..as[2*i]` for `i` in `1..n/2` as a possible range

---

```
predicate float_in(var float: a, float: b, float: c)
```

Constrains $\mathbf{a} \in [\mathbf{b}, \mathbf{c}]$

4.2.3.8 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.

4.2.3.9 FlatZinc builtins added in MiniZinc 2.2.1.

These functions and predicates define built-in operations of the MiniZinc language that have been added in MiniZinc 2.2.1. Solvers that support these natively need to include a file called redefinitions-2.2.1.mzn in their solver library that redefines these predicates as builtins.

```
predicate int_pow_fixed(var int: x, int: y, var int: z)
```

Constrains `z` = $\mathbf{x}^{\mathbf{y}}$, {bf z} is constrained to `1 div pow(x, abs(y))` when $\mathbf{y} < 0$ *

4.2.3.10 FlatZinc builtins added in MiniZinc 2.3.3.

These functions and predicates define built-in operations of the MiniZinc language that have been added in MiniZinc 2.3.3. Solvers that support these natively need to include a file called redefinitions-2.3.3.mzn in their solver library that redefines these predicates as builtins.

```
predicate float_set_in(var float: x, set of float: S)
```

Constrains `x` $\in$ `S`

4.2.3.11 FlatZinc builtins added in MiniZinc 2.5.2.

These functions and predicates define built-in operations of the MiniZinc language that have been added in MiniZinc 2.5.2. Solvers that support these natively need to include a

file called redefinitions-2.5.2.mzn in their solver library that redefines these predicates as builtins.

```
predicate array_var_bool_element2d_nonshifted(var int: idx1,
                                              var int: idx2,
                                              array [int,int] of var bool: x,
                                              var bool: c)
```

Element constraint on 2d array with MiniZinc index set, constrains `x[idx1,idx2] = 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_element2d_nonshifted(var int: idx1,
                                               var int: idx2,
                                               array [int,int] of var float: x,
                                               var float: c)
```

Element constraint on 2d array with MiniZinc index set, constrains `x[idx1,idx2] = 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_element2d_nonshifted(var int: idx1,
                                             var int: idx2,
                                             array [int,int] of var int: x,
                                             var int: c)
```

Element constraint on 2d array with MiniZinc index set, constrains `x[idx1,idx2] = 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_element2d_nonshifted(var int: idx1,
                                             var int: idx2,
                                             array [int,int] of var set of int: x,
                                             var set of int: c)
```

Element constraint on 2d array with MiniZinc index set, constrains `x[idx1,idx2] = c` This can be overridden in a solver that can perform the index calculation more efficiently than using a MiniZinc decomposition.

4.2.3.12 FlatZinc builtins added in MiniZinc 2.7.1.

These functions and predicates define built-in operations of the MiniZinc language that have been added in MiniZinc 2.7.1. Solvers that support these natively need to include a file called redefinitions-2.7.1.mzn in their solver library that redefines these predicates as builtins.

```
predicate float_ceil(var float: x, var int: y)
```

Constrains $\mathbf{y} = \lceil \mathbf{x} \rceil$

```
predicate float_floor(var float: x, var int: y)
```

Constrains $\mathbf{y} = \lfloor \mathbf{x} \rfloor$

```
predicate float_round(var float: x, var int: y)
```

Constrains that `y` is the nearest integer to `x`

4.2.3.13 Deprecated FlatZinc builtins

These builtin constraints are no longer produced by the MiniZinc compiler, and they will be removed in future versions of MiniZinc.

```
predicate array_bool_or(array [int] of var bool: as, var bool: r)
```

Constrains $\mathbf{r} \leftrightarrow \bigvee_i \mathbf{as}[i]$

4.2.4 Additional declarations for Gecode

These annotations and predicates are available for the Gecode solver. In order to use them in a model, include the file "gecode.mzn".

4.2.4.1 Additional Gecode search annotations

```
annotation action_max
```

Select variable with largest action count

```
annotation action_min
```

Select variable with smallest action count

```
annotation action_size_max
```

Select variable with largest action count divided by domain size

```
annotation action_size_min
```

Select variable with smallest action count divided by domain size

```
annotation afc_max
```

Select variable with largest accumulated failure count

```
annotation afc_min
```

Select variable with smallest accumulated failure count

```
annotation afc_size_max
```

Select variable with largest accumulated failure count divided by domain size

```
annotation afc_size_min
```

Select variable with smallest accumulated failure count divided by domain size

```
annotation bool_default_search(ann: varsel, ann: valsel)
```

Specify default search strategy for Boolean variables to use variable selection strategy `varsel`, and value choice strategy `valsel`.

```
annotation float_default_search(ann: varsel, ann: valsel)
```

Specify default search strategy for float variables to use variable selection strategy `varsel`, and value choice strategy `valsel`.

```
annotation int_default_search(ann: varsel, ann: valsel)
```

Specify default search strategy for integer variables to use variable selection strategy `varsel`, and value choice strategy `valsel`.

```
annotation random
```

Select random variable

```
1.  annotation relax_and_reconstruct(array [int] of var int: x,
                                     int: percentage)

2.  annotation relax_and_reconstruct(array [int] of var int: x,
                                     int: percentage,
                                     array [int] of int: y)
```

1. Simple large neighbourhood search strategy: upon restart, for each variable in `x`, the probability of it being fixed to the previous solution is `percentage` (out of 100).
2. Simple large neighbourhood search strategy: upon restart, for each variable in `x`, the probability of it being fixed to the previous solution is `percentage` (out of 100). Start from an initial solution `y`.

```
annotation set_default_search(ann: varsel, ann: valsel)
```

Specify default search strategy for set variables to use variable selection strategy `varsel`, and value choice strategy `valsel`.

4.2.4.2 Additional Gecode constraints

```
1.  predicate among_seq(array [int] of var int: x,
                        set of int: S,
                        int: l,
                        int: m,
                        int: n)

2.  predicate among_seq(array [int] of var bool: x,
                        bool: b,
                        int: l,
                        int: m,
                        int: n)
```

1. Every subsequence of `x` of length `l` has at least `m` and at most `n` occurrences of the values in `S`
2. Every subsequence of `x` of length `l` has at least `m` and at most `n` occurrences of the value `b`

```
predicate circuit_cost(array [int] of int: c,
                       array [int] of var int: x,
```

```
                        var int: z)
```

Constrains the elements of x to define a circuit where x[i] = j means that j is the successor of i. Additionally, constrain z to be the cost of the circuit. Each edge cost is defined by array c.

```
predicate circuit_cost_array(array [int] of int: c,
                             array [int] of var int: x,
                             array [int] of var int: y,
                             var int: z)
```

Constrains the elements of x to define a circuit where x[i] = j means that j is the successor of i. Additionally, constrain z to be the cost of the circuit. Each edge cost is defined by array c. The variables y[i] are constrained to be the edge cost of the node x[i].

```
predicate gecode_array_set_element_intersect(var set of int: x,
                                             array [int] of var set of int: y,
                                             var set of int: z)
```

Constrain z to be the intersection of all sets in y that are selected by x: $i \in \mathbf{z} \leftrightarrow \forall j \in \mathbf{x} : (i \in \mathbf{y}[j])$

```
predicate gecode_array_set_element_intersect_in(var set of int: x,
                                                array [int] of var set of int: y,
                                                var set of int: z,
                                                set of int: u)
```

Constrain z to be a subset of u, and z to be the intersection of all sets in y that are selected by x: $i \in \mathbf{z} \leftrightarrow \forall j \in \mathbf{x} : (i \in \mathbf{y}[j])$

```
predicate gecode_array_set_element_partition(var set of int: x,
                                             array [int] of var set of int: y,
                                             var set of int: z)
```

Constrain z to be the disjoint union of all sets in y that are selected by x: $i \in \mathbf{z} \leftrightarrow \exists j \in \mathbf{x} : (i \in \mathbf{y}[j])$ and $i \in \mathbf{x} \wedge j \in \mathbf{x} \wedge i \neq j \rightarrow \mathbf{y}[i] \cap \mathbf{y}[j] = \emptyset$

4.2.5 Additional declarations for Chuffed

These annotations and predicates are available for the Chuffed solver. In order to use them in a model, include the file "chuffed.mzn".

4.2.5.1 Additional Chuffed search annotations

```
annotation assume(array [int] of var bool)
```

annotation for assumption handling.

```
annotation largest_smallest
```

Choose the variable with the largest smallest value in its domain

---

```
annotation priority_search(array [int] of var int: x,
                           array [int] of ann: search,
                           ann: select,
                           ann: explore)
```

Specify search on a group of searches `search`, for which variables `x` act as delegates for the search selection with variable selection strategy `select`, and exploration strategy `explore`.

```
annotation random_order
```

Choose a random variable

```
annotation smallest_largest
```

Choose the variable with the smallest largest value in its domain

---

4.2.5.2 Other declarations

```
predicate chuffed_minimal_spanning_tree(int: N,
                                        int: E,
                                        array [int] of int: from,
                                        array [int] of int: to,
                                        array [int] of int: w,
                                        array [int] of var bool: vs,
                                        array [int] of var bool: es,
                                        var int: K)
```

Constrains the subgraph `vs` and `es` of a given directed graph to be a minimal spanning tree with weight `K`.

Parameters:

- `N`: the number of nodes in the given graph
- `E`: the number of edges in the given graph
- `from`: the leaving node 1..`N` for each edge
- `to`: the entering node 1..`N` for each edge
- `w`: the weight of each edge
- `vs`: a Boolean for each node whether it is in the subgraph
- `es`: a Boolean for each edge whether it is in the subgraph
- `K`: the cost of the edges in the minimal spanning tree

4.2.6 MiniZincIDE tools

This library contains features designed for use from within the MiniZinc IDE. These are not guaranteed to work when run outside the IDE.

4.2.6.1 MiniZincIDE solution visualisation tools

Provides functionality for web-based solution visualisation.

All predefined visualisations are designed to used from `output` items:

```
var 1..10: x;
output vis_line(x, "x-value");
```

Multiple such output items can be used in a single model to open multiple visualisations concurrently.

Custom visualisations can be created using the `vis_server` function:

```
int: n;
var 1..10: x;
output :: vis_server("my-vis.html", (n: n)) (x: x);
```

The file `my-vis.html` will be opened and the MiniZinc IDE JavaScript API can be used to listen for events and retrieve solutions. For more information, see *Visualising solutions in the MiniZinc IDE* (page 153).

```
1.  annotation vis_bar(array [$$D] of var int: x)

2.  annotation vis_bar(array [$$D] of var float: x)

3.  annotation vis_bar(array [$$D] of var int: x,
                       array [$$D] of string: data_labels)

4.  annotation vis_bar(array [$$D] of var float: x,
                       array [$$D] of string: data_labels)
```

```
5.  annotation vis_bar(array [$$S,$$D] of var int: x)

6.  annotation vis_bar(array [$$S,$$D] of var float: x)

7.  annotation vis_bar(array [$$S,$$D] of var int: x,
                       array [$$S] of string: series_labels,
                       array [$$D] of string: data_labels)

8.  annotation vis_bar(array [$$S,$$D] of var float: x,
                       array [$$S] of string: series_labels,
                       array [$$D] of string: data_labels)
```

1, 2.
Visualise a bar chart of the values in `x` (labelled by index set)

3, 4.
Visualise a bar chart of the values in `x`. Chart values are labelled with `data_labels`.

5, 6.
Visualise a bar chart with grouped data (labelled by index set). `x[i, j]` is the value for `j` in data series `i`.

7, 8.
Visualise a bar chart with grouped data. `x[i, j]` is the value for `j` in data series `i`. Series are labelled with `series_labels`. Chart values are labelled with `data_labels`.

```
1.  annotation vis_column(array [$$D] of var int: x)

2.  annotation vis_column(array [$$D] of var float: x)

3.  annotation vis_column(array [$$D] of var int: x,
                          array [$$D] of string: data_labels)

4.  annotation vis_column(array [$$D] of var float: x,
                          array [$$D] of string: data_labels)

5.  annotation vis_column(array [$$S,$$D] of var int: x)

6.  annotation vis_column(array [$$S,$$D] of var float: x)

7.  annotation vis_column(array [$$S,$$D] of var int: x,
                          array [$$S] of string: series_labels,
                          array [$$D] of string: data_labels)

8.  annotation vis_column(array [$$S,$$D] of var float: x,
                          array [$$S] of string: series_labels,
                          array [$$D] of string: data_labels)
```

1, 2.
Visualise a column chart of the values in `x` (labelled by index set)

3, 4.
Visualise a column chart of the values in `x`. Chart values are labelled with `data_labels`.

5, 6.
Visualise a column chart with grouped data (labelled by index set). `x[i, j]` is the value for `j` in data series `i`.

7, 8.
Visualise a column chart with grouped data. `x[i, j]` is the value for `j` in data series `i`. Series are labelled with `series_labels`. Chart values are labelled with `data_labels`.

```
1.  annotation vis_digraph(array [$$E] of $$N: from,
                           array [$$E] of $$N: to,
                           array [$$E] of string: edge_labels,
                           array [$$N] of var bool: ns,
                           array [$$E] of var bool: es)

2.  annotation vis_digraph(array [$$E] of $$N: from,
                           array [$$E] of $$N: to,
                           array [$$N] of var bool: ns,
                           array [$$E] of var bool: es)

3.  annotation vis_digraph(array [$$N] of string: node_labels,
                           array [$$E] of $$N: from,
                           array [$$E] of $$N: to,
                           array [$$E] of string: edge_labels,
                           array [$$N] of var bool: ns,
                           array [$$E] of var bool: es)
```

1. Create a visualisation of the subgraph `ns` and `es` of a given directed graph. The nodes are labelled using the index set of `ns`.

   Parameters:

   - `from`: the leaving node for each edge
   - `to`: the entering node for each edge
   - `edge_labels`: labels to use for each edge
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph

2. Create a visualisation of the subgraph `ns` and `es` of a given directed graph. The nodes are labelled using the index set of `ns` and the edges are labelled using the index set of `es`.

   Parameters:

   - `from`: the leaving node for each edge
   - `to`: the entering node for each edge
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph

3. Create a visualisation of the subgraph `ns` and `es` of a given directed graph.

   Parameters:

- node_labels: the label for each node
- from: the leaving node for each edge
- to: the entering node for each edge
- edge_labels: labels to use for each edge
- ns: a Boolean for each node whether it is in the subgraph
- es: a Boolean for each edge whether it is in the subgraph

```
1.  annotation vis_digraph_highlight(array [$$E] of $$N: from,
                                     array [$$E] of $$N: to,
                                     array [$$E] of string: edge_labels,
                                     array [$$N] of var bool: ns,
                                     array [$$E] of var bool: es)

2.  annotation vis_digraph_highlight(array [$$E] of $$N: from,
                                     array [$$E] of $$N: to,
                                     array [$$N] of var bool: ns,
                                     array [$$E] of var bool: es)

3.  annotation vis_digraph_highlight(array [$$N] of string: node_labels,
                                     array [$$E] of $$N: from,
                                     array [$$E] of $$N: to,
                                     array [$$E] of string: edge_labels,
                                     array [$$N] of var bool: ns,
                                     array [$$E] of var bool: es)
```

1. Create a visualisation of the subgraph ns and es of a given directed graph. The nodes are labelled using the index set of ns. The entire graph is drawn, with the given subgraph highlighted.

   Parameters:

   - from: the leaving node for each edge
   - to: the entering node for each edge
   - edge_labels: labels to use for each edge
   - ns: a Boolean for each node whether it is in the subgraph
   - es: a Boolean for each edge whether it is in the subgraph

2. Create a visualisation of the subgraph ns and es of a given directed graph. The nodes are labelled using the index set of ns and the edges are labelled using the index set of es. The entire graph is drawn, with the given subgraph highlighted.

   Parameters:

   - from: the leaving node for each edge
   - to: the entering node for each edge
   - ns: a Boolean for each node whether it is in the subgraph
   - es: a Boolean for each edge whether it is in the subgraph

3. Create a visualisation of the subgraph `ns` and `es` of a given directed graph. The entire graph is drawn, with the given subgraph highlighted.

   Parameters:

   - `node_labels`: the label for each node
   - `from`: the leaving node for each edge
   - `to`: the entering node for each edge
   - `edge_labels`: labels to use for each edge
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph

---

```
1.  annotation vis_gantt(array [$$E] of var int: start,
                         array [$$E] of var int: dur)

2.  annotation vis_gantt(array [$$E] of var int: start,
                         array [$$E] of var int: dur,
                         array [$$E] of string: labels)

3.  annotation vis_gantt(array [$$E] of var int: start,
                         array [$$E] of var int: dur,
                         array [$$E] of string: labels,
                         array [$$E] of string: colors)
```

1. Visualise a Gantt chart of tasks with start times `start` and durations `dur`. The tasks are labelled using the index set of `start`.
2. Visualise a Gantt chart of tasks with start times `start` and durations `dur`. The tasks are labelled using `labels`.
3. Visualise a Gantt chart of tasks with start times `start` and durations `dur`. The tasks are labelled using `labels` and given the colours `colors`.

```
1.  annotation vis_geost_2d(array [$$E] of int: rect_x,
                            array [$$E] of int: rect_y,
                            array [$$E] of int: rect_dx,
                            array [$$E] of int: rect_dy,
                            array [$$K] of set of $$E: shape,
                            array [$$T] of var int: x,
                            array [$$T] of var int: y,
                            array [$$T] of var $$K: kind)

2.  annotation vis_geost_2d(array [$$E,1..2] of int: rect_size,
                            array [$$E,1..2] of int: rect_offset,
                            array [$$K] of set of $$E: shape,
                            array [$$T,1..2] of var int: x,
                            array [$$T] of var $$K: kind)

3.  annotation vis_geost_2d(array [$$E] of tuple(int, int): rect_size,
                            array [$$E] of tuple(int, int): rect_offset,
```

```
                                array [$$K] of set of $$E: shape,
                                array [$$T] of tuple(var int, var int): x,
                                array [$$T] of var $$K: kind)
```

1. Visualise the positions of 2-dimensional objects which take a particular shape composed of rectangles.

   Parameters:

   - `rect_x`: x-position of each rectangle
   - `rect_y`: y-position of each rectangle
   - `rect_dx`: width of each rectangle
   - `rect_dy`: height of each rectangle
   - `shape`: the rectangles which comprise each shape
   - `x`: the x-position of each object
   - `y`: the y-position of each object
   - `kind`: which shape each object is

2, 3.

Visualise the positions of 2-dimensional objects which take a particular shape composed of rectangles.

Parameters:

- `rect_size`: the size of each rectangle
- `rect_offset`: the position of each rectangle
- `shape`: the rectangles which comprise each shape
- `x`: the position of each object
- `kind`: which shape each object is

```
1.  annotation vis_graph(array [$$E] of $$N: from,
                         array [$$E] of $$N: to,
                         array [$$E] of string: edge_labels,
                         array [$$N] of var bool: ns,
                         array [$$E] of var bool: es)

2.  annotation vis_graph(array [$$E] of $$N: from,
                         array [$$E] of $$N: to,
                         array [$$N] of var bool: ns,
                         array [$$E] of var bool: es)

3.  annotation vis_graph(array [$$N] of string: node_labels,
                         array [$$E] of $$N: from,
                         array [$$E] of $$N: to,
                         array [$$E] of string: edge_labels,
                         array [$$N] of var bool: ns,
                         array [$$E] of var bool: es)
```

1. Create a visualisation of the subgraph `ns` and `es` of a given undirected graph. The nodes are labelled using the index set of `ns`.

   Parameters:

   - `from`: the leaving node for each edge
   - `to`: the entering node for each edge
   - `edge_labels`: labels to use for each edge
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph

2. Create a visualisation of the subgraph `ns` and `es` of a given undirected graph. The nodes are labelled using the index set of `ns` and the edges are labelled using the index set of `es`.

   Parameters:

   - `from`: the leaving node for each edge
   - `to`: the entering node for each edge
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph

3. Create a visualisation of the subgraph `ns` and `es` of a given undirected graph.

   Parameters:

   - `node_labels`: the label for each node
   - `from`: the leaving node for each edge
   - `to`: the entering node for each edge
   - `edge_labels`: labels to use for each edge
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph

---

```
1.  annotation vis_graph_highlight(array [$$E] of $$N: from,
                                   array [$$E] of $$N: to,
                                   array [$$E] of string: edge_labels,
                                   array [$$N] of var bool: ns,
                                   array [$$E] of var bool: es)

2.  annotation vis_graph_highlight(array [$$E] of $$N: from,
                                   array [$$E] of $$N: to,
                                   array [$$N] of var bool: ns,
                                   array [$$E] of var bool: es)

3.  annotation vis_graph_highlight(array [$$N] of string: node_labels,
                                   array [$$E] of $$N: from,
                                   array [$$E] of $$N: to,
                                   array [$$E] of string: edge_labels,
                                   array [$$N] of var bool: ns,
```

```
                                    array [$$E] of var bool: es)
```

1. Create a visualisation of the subgraph `ns` and `es` of a given undirected graph. The nodes are labelled using the index set of `ns`. The entire graph is drawn, with the given subgraph highlighted.

   Parameters:

   - `from`: the leaving node for each edge
   - `to`: the entering node for each edge
   - `edge_labels`: labels to use for each edge
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph

2. Create a visualisation of the subgraph `ns` and `es` of a given undirected graph. The nodes are labelled using the index set of `ns` and the edges are labelled using the index set of `es`. The entire graph is drawn, with the given subgraph highlighted.

   Parameters:

   - `from`: the leaving node for each edge
   - `to`: the entering node for each edge
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph

3. Create a visualisation of the subgraph `ns` and `es` of a given undirected graph. The entire graph is drawn, with the given subgraph highlighted.

   Parameters:

   - `node_labels`: the label for each node
   - `from`: the leaving node for each edge
   - `to`: the entering node for each edge
   - `edge_labels`: labels to use for each edge
   - `ns`: a Boolean for each node whether it is in the subgraph
   - `es`: a Boolean for each edge whether it is in the subgraph

---

```
1.  annotation vis_line(var int: x, string: label)

2.  annotation vis_line(var float: x, string: label)

3.  annotation vis_line(array [$$E] of var int: x)

4.  annotation vis_line(array [$$E] of var float: x)

5.  annotation vis_line(array [$$E] of var int: x,
                        array [$$E] of string: series_labels)
```

```
6.  annotation vis_line(array [$$E] of var float: x,
                        array [$$E] of string: series_labels)

7.  annotation vis_line(array [$$E] of var int: x,
                        array [$$E] of var int: y,
                        string: x_label,
                        string: y_label)

8.  annotation vis_line(array [$$E] of var float: x,
                        array [$$E] of var float: y,
                        string: x_label,
                        string: y_label)
```

1, 2.
Add a line graph of `x` with respect to solution time, labelling the trace `label`.

3, 4.
Add line graphs of the values in `x` with respect to solution time on the same axes, labelling the series using their index.

5, 6.
Add line graphs of the values in `x` with respect to solution time on the same axes, labelling the series with `series_labels`.

7, 8.
Add a line graph of `x` against `y`, labelling the axes `x_label` and `y_label` respectively.

```
annotation vis_scatter(array [$$E] of var int: x,
                       array [$$E] of var int: y,
                       string: x_label,
                       string: y_label)
annotation vis_scatter(array [$$E] of var float: x,
                       array [$$E] of var float: y,
                       string: x_label,
                       string: y_label)
```

Add a scatter plot of `x` against `y`, labelling the axes `x_label` and `y_label` respectively.

```
1.  annotation vis_scatter_cumulative(var int: x,
                                      var int: y,
                                      string: x_label,
                                      string: y_label,
                                      string: series_label)

2.  annotation vis_scatter_cumulative(var float: x,
                                      var float: y,
                                      string: x_label,
                                      string: y_label,
                                      string: series_label)

3.  annotation vis_scatter_cumulative(array [$$E] of var int: x,
                                      array [$$E] of var int: y,
```

```
                                        string: x_label,
                                        string: y_label)

4.  annotation vis_scatter_cumulative(array [$$E] of var float: x,
                                        array [$$E] of var float: y,
                                        string: x_label,
                                        string: y_label)

5.  annotation vis_scatter_cumulative(array [$$E] of var int: x,
                                        array [$$E] of var int: y,
                                        string: x_label,
                                        string: y_label,
                                        array [$$E] of string: series_labels)

6.  annotation vis_scatter_cumulative(array [$$E] of var float: x,
                                        array [$$E] of var float: y,
                                        string: x_label,
                                        string: y_label,
                                        array [$$E] of string: series_labels)
```

1, 2.

Create a scatter-plot visualisation. The point (`x`, `y`) is added to the plot for each solution.

Parameters:

- `x`: the x coordinate
- `y`: the y coordinate
- `x_label`: the x-axis label
- `y_label`: the y-axis label
- `series_label`: the series label

3, 4.

Create a scatter-plot visualisation with multiple series. The point (`x`[i], `y`[i]) for each series `i` is added to the plot for each solution. Series are labelled using the index set of `x` and `y`.

Parameters:

- `x`: the x coordinate for each series
- `y`: the y coordinate for each series
- `x_label`: the x-axis label
- `y_label`: the y-axis label

5, 6.

Create a scatter-plot visualisation with multiple series. The point (`x`[i], `y`[i]) for each series `i` is added to the plot for each solution.

Parameters:

- `x`: the x coordinate for each series
- `y`: the y coordinate for each series

- `x_label`: the x-axis label
- `y_label`: the y-axis label
- `series_labels`: the label for each series

```
1.  annotation vis_server(string: file)

2.  annotation vis_server(string: file, opt $T: user_data)

3.  annotation vis_server(string: file, array [$X] of opt $T: user_data)
```

1. Launch a visualisation server for a custom visualisation using the given `file`.

2, 3.
Launch a visualisation server for a custom visualisation using the given `file`. The `user_data` value can be retrieved in the page using `MiniZincIDE.getUserData()`.

4.2.7 MiniZinc definitions for the MiniZinc diversity extension.

These annotations and predicates can be used to produce diverse set of solutions.In order to use them in a model, include the file ”diversity.mzn”.

4.2.7.1 Annotations provided to guide the MiniZinc diversity extension.

Functions and Predicates

```
1.  annotation diverse_pairwise(array [int] of var int: x,
                                string: distance_metric)

2.  annotation diverse_pairwise(array [$T] of var int: x,
                                string: distance_metric)

3.  annotation diverse_pairwise(array [int] of var int: x,
                                string: distance_metric,
                                int: min_dist,
                                int: max_dist,
                                int: weight)

4.  annotation diverse_pairwise(array [$T] of var int: x,
                                string: distance_metric,
                                int: min_dist,
                                int: max_dist,
                                int: weight)

5.  annotation diverse_pairwise(array [int] of var float: x,
                                string: distance_metric)

6.  annotation diverse_pairwise(array [$T] of var float: x,
                                string: distance_metric)
```

```
7.  annotation diverse_pairwise(array [int] of var float: x,
                                string: distance_metric,
                                float: min_dist,
                                float: max_dist,
                                float: weight)

8.  annotation diverse_pairwise(array [$T] of var float: x,
                                string: distance_metric,
                                float: min_dist,
                                float: max_dist,
                                float: weight)
```

1, 2.
Use `distance_metric` on `x` when computing distance to solutions. Here same index on different solutions are compared. For example, when obtaining the distance to k-th diverse solution using the incremental approach (diversity_incremental) each index of `x` in k-th solution is compared to the corresponding index of all previously obtained diverse solutions.

3, 4.
Use `distance_metric` on `x` to compute the distance to solutions, and apply constraints using `min_dist` to enforce a minimum distance and `max_dist` to enforce a maximum distance. The `weight` parameter can be used to set relative priorities between multiple distance measures. Distance measures with higher weights are given greater priority. For example, when obtaining the distance to k-th diverse solution using the incremental approach (diversity_incremental) each index of `x` in k-th solution is compared to the corresponding index of all previously obtained diverse solutions.

5, 6.
Use `distance_metric` on `x` when computing distance to solutions. Here same index on different solutions are compared. For example, when obtaining the distance to k-th diverse solution using the incremental approach (diversity_incremental) each index of `x` in k-th solution is compared to the corresponding index of all previously obtained diverse solutions.

7, 8.
Use `distance_metric` on `x` to compute the distance to solutions, and apply constraints using `min_dist` to enforce a minimum distance and `max_dist` to enforce a maximum distance. The `weight` parameter can be used to set relative priorities between multiple distance measures. Distance measures with higher weights are given greater priority. For example, when obtaining the distance to k-th diverse solution using the incremental approach (diversity_incremental) each index of `x` in k-th solution is compared to the corresponding index of all previously obtained diverse solutions.

Annotations

```
annotation diversity_aggregator(string)
```

An aggregator combines pairwise diversity measures from each pair of solutions into an overall diversity score to identify the k most diverse solutions, using an incremental approach (i.e., diversity_incremental). Currently, it supports three methods: sum, which sums up the pairwise diversity measures; min, which selects the minimum; and max, which selects the maximum.

```
annotation diversity_combinator(string)
```

Use to combine distance functions to diversity of a diverse solution. Currently, it supports three methods: sum, which sums up the distance measures; min, which selects the minimum; and max, which selects the maximum.

```
annotation diversity_incremental(int: k, float: gap)
```

Produce at most `k` number of diverse solutions with thier objective values bounded between optimal value and `gap` % from the optimal. Diverse solutions are obtained one after the other and distance to k-th diverse solution is sum of pairwise distance to all the previous diverse solutions

4.2.7.2 Distance functions provided in the MiniZinc diversity extension.

```
function int: hamming_distance(array [$A] of opt $T: x,
                               array [$A] of opt $T: y)
function var int: hamming_distance(array [$A] of var opt $T: x,
                                   array [$A] of var opt $T: y)
```

Returns the Hamming distance between `x` and `y`.

```
function int: manhattan_distance(array [$A] of $$T: x,
                                 array [$A] of $$T: y)
function var int: manhattan_distance(array [$A] of var $$T: x,
                                     array [$A] of var $$T: y)
function float: manhattan_distance(array [$A] of float: x,
                                   array [$A] of float: y)
function var float: manhattan_distance(array [$A] of var float: x,
                                       array [$A] of var float: y)
```

Returns the Manhattan distance between `x` and `y`.

4.2.8 Experimental Features

These features are experimental and might significantly change or be removed in future versions of MiniZinc.

4.2.8.1 On Restart

These annotations and functions provide the basic building blocks to implement meta-heuristics triggered on restart and governed search with the use of solver internal values and functionality.

Constants

```
set of int: STATUS
```

Reports the status of the solver (before restarting).

Functions and Predicates

```
predicate basic_lns(var bool: nbh)
```

'basic_lns' provides LNS in its simplest form where one nbh is applied on every restart.

```
test complete()
```

When complete is set to 'true', then it marks the search as complete.

```
predicate last_val(var bool: x)
function var float: last_val(var float: x)
function var $$E: last_val(var $$E: x)
function var opt $$E: last_val(var opt $$E: x)
function var set of $$E: last_val(var set of $$E: x)
```

Provides access to last values taken by a model variables. The last_val functions are only safe to use when the STATUS has been SAT or OPT.

```
predicate round_robin(array [int] of var bool: nbhs)
```

round_robin provides a metaheuristic for LNS where each neighbourhood is chosen sequentially.

```
predicate sol(var bool: x)
function var float: sol(var float: x)
function var $$E: sol(var $$E: x)
function var opt $$E: sol(var opt $$E: x)
function var set of $$E: sol(var set of $$E: x)
function array [$$E] of var bool: sol(array [$$E] of var bool: x)
function array [$$E] of var float: sol(array [$$E] of var float: x)
function array [$$E] of var $$F: sol(array [$$E] of var $$F: x)
```

Provides access to solution values of model variables. The sol function are only safe to use when the STATUS has been SAT or OPT.

```
1.  function var $$E: uniform_on_restart($$E: low, $$E: high)

2.  function var $$E: uniform_on_restart(set of $$E: S)

3.  function var float: uniform_on_restart(float: low, float: high)
```

1. Returns a new random value between `low` and `high` chosen from a uniform distribution whenever the solver restarts.
2. Returns a new random value from the set `S` chosen from a uniform distribution whenever the solver restarts.
3. Returns a new random value between `low` and `high` chosen from a uniform distribution whenever the solver restarts.

Annotations

```
annotation on_restart(string: pred)
```

A search annotation that triggers a function to be executed on restart

---

CHAPTER 4.3

Interfacing Solvers to Flatzinc

This document describes the interface between the MiniZinc system and FlatZinc solvers. Interfacing a solver with MiniZinc usually requires three components:

1. The solver binary (or a script that runs the solver). It should support the command line options specified in 节 4.3.5, understand either the FlatZinc input language as defined in 节 4.3.1, or the FlatZinc-JSON input language as defined in 节 4.3.2. Solvers must produce output as defined in 节 4.3.3.
2. The solver's MiniZinc library. This library defines the global constraints that the solver supports, and redefines FlatZinc built-in constraints that the solver does not support (see 节 4.3.4).
3. The solver's configuration file. This file makes the solver known to the `minizinc` driver program and the MiniZinc IDE. Configuration files are discussed in 节 4.3.6.

We recommend to install solvers following something similar to the Linux Filesystem Hierarchy Standard: When installing a solver called `$SOLVERNAME` in a directory `$PREFIX`, its binary should be put in `$PREFIX/bin`, its MiniZinc library files should be in `$PREFIX/share/minizinc/$SOLVERNAME/`, and its solver configuration file should be `$PREFIX/share/minizinc/solvers/$SOLVERNAME.mzc`. The prefix can either be a global installation path (such as `/usr/local`) or any other path.

4.3.1 Specification of FlatZinc

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; $x_1, x_2, \ldots, x_k, i, j, k$ denote int literals; $y_1, y_2, \ldots, y_k, y_i$ 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 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 节 4.3.7).

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 `..` r_b	bounded float
x_a `..` x_b	int in range
`{` $x_a, x_b, \ldots, x_k$ `}`	int in set
`set of` x_a `..` x_b	subset of int range
`set of {` $x_a, x_b, \ldots, x_k$ `}`	subset of int set
`array [1..` n `] of` r_a `..` r_b	array of floats in range
`array [1..` n `] of` x_a `..` x_b	array of ints in range
`array [1..` n `] of set of` x_a `..` x_b	array of sets of ints in range
`array [1..` n `] of set of {` $x_a, x_b, \ldots, x_k$ `}`	array of subsets of set

A range x_a `..` 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 节 4.3.7).

Variable type
`var bool`
`var float`
`var` r_a `..` r_b
`var int`
`var` x_a `..` x_b
`var {` $x_a, x_b, \ldots, x_k$ `}`
`var set of` x_a `..` x_b
`var set of {` $x_a, x_b, \ldots, x_k$ `}`
`array [1..` n `] of var bool`
`array [1..` n `] of var float`
`array [1..` n `] of var` r_a `..` r_b
`array [1..` n `] of var int`
`array [1..` n `] of var` x_a `..` x_b
`array [1..` n `] of var set of` x_a `..` x_b
`array [1..` n `] of var set of {` $x_a, x_b, \ldots, x_k$ `}`

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 节 4.3.7)

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` `[]`, `[` $y_a, \ldots, y_k$ `]`

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
```

节 4.3.7 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 节 4.3.7)

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 节 4.3.7)

Parameters have fixed values and must be assigned values:

```
<par-decl-item> ::= "array" "[" <index-set> "]" "of" <basic-par-type> ":" <var-par-
↪identifier> "=" <par-array-literal> ";"
```

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 节 4.3.7)

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 节 4.3.7)

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_propagation': use domain consistency for this constraint:
    % 2x + 3y = 10
constraint int_lin_eq([2, 3], [x, y], 10) :: domain_propagation;
```

4.3.1.9 Solve item

(See rule <solve-item> in 节 4.3.7)

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>   := <basic-ann-expr>
              | "[" [ <basic-ann-expr> "," ... ] "]"
```

```
<basic-ann-expr>   := <basic-literal-expr>
                    | <var-par-identifier>
                    | <string-literal>
                    | <annotation>

<string-contents> ::= ([^"\n\] | \[^\n(])*

<string-literal> ::= """ <string-contents> """
```

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):

input_-order	⋆	Choose variables in the order they appear in vars.
first_-fail	⋆	Choose the variable with the smallest domain.
anti_-first_-fail		Choose the variable with the largest domain.
smallest		Choose the variable with the smallest value in its domain.
largest		Choose the variable with the largest value in its domain.
occurrence		Choose the variable with the largest number of attached constraints.
most_-constrained		Choose the variable with the smallest domain, breaking ties using the number of constraints.
max_-regret		Choose the variable with the largest difference between the two smallest values in its domain.
dom_-w_deg		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_-split		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([` 1 `..` x "])" where 1 `..` x is the index set range of the array. See 节 4.3.3 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:

bounc	**Use bounds propagation. The concrete propagation strength is not further specified.**
bounc	Use integer bounds propagation.
bounc	Use real bounds propagation.
bounc	A tighter version of `boundsZ` where support for the bounds must exist.
doma:	Use domain propagation.
value	Use value propagation.
propa	
prior	where `k` is an integer constant indicating propagator priority.

4.3.2 Specification of FlatZinc-JSON

FlatZinc-JSON is an alternative syntax for FlatZinc that is compatible with the JSON data format. Solvers can indicate that they require FlatZinc output in their solver configuration files (see 节 4.3.6) using the *inputType* property.

The JSON representation of FlatZinc contains the same elements as a FlatZinc file: declarations of variables, arrays, constraints and a solve item. A full specification is given as a JSON schema at the end of this section. Here is an example of the cake optimisation model from 列表 1.2, compiled into FlatZinc-JSON:

```
{
  "variables": {
    "b" : { "type" : "int", "domain" : [[0, 3]] },
    "c" : { "type" : "int", "domain" : [[0, 6]] },
    "X_INTRODUCED_0_" : { "type" : "int", "domain" : [[0, 85000]], "defined" : true␣
↪}
  },
  "arrays": {
    "X_INTRODUCED_2_" : { "a": [250, 200] },
    "X_INTRODUCED_6_" : { "a": [75, 150] },
    "X_INTRODUCED_8_" : { "a": [100, 150] }
  },
  "constraints": [
    { "id" : "int_lin_le", "args" : ["X_INTRODUCED_2_", ["b", "c"], 4000]},
    { "id" : "int_lin_le", "args" : ["X_INTRODUCED_6_", ["b", "c"], 2000]},
    { "id" : "int_lin_le", "args" : ["X_INTRODUCED_8_", ["b", "c"], 500]},
    { "id" : "int_lin_eq", "args" : [[400, 450, -1], ["b", "c", "X_INTRODUCED_0_"],␣
↪0],
      "ann" : ["ctx_pos"], "defines" : "X_INTRODUCED_0_"}
  ],
  "output": ["b", "c"],
  "solve": { "method" : "maximize", "objective" : "X_INTRODUCED_0_" },
  "version": "1.0"
}
```

There are a number of small differences compared to FlatZinc:

- Variables and arrays are declared separately. Each variable must have an identifier and a type, and can optionally have a domain, a flag indicating that the variable was introduced by compilation, a flag that the variable is functionally defined by a constraint, and a property `"rhs"` to define the "right hand side" of the variable, i.e., its value or alias to another variable.
- Domains are written as lists of contiguous, ordered, non-overlapping ranges. E.g., the set `{1,2,4,6,7}` would be written as `[[1,2],[4,4],[6,7]]`.
- Arrays are declared with an identifier and an object that has the property `"a"`, which maps to the actual array. Arrays can contain numeric literals or identifiers (written as strings).
- Constraints are declared as objects with a property `"id"` indicating the name of the constraint, a property `"args"` for the array of arguments to the constraint, and an optional property `"defines"` that indicates the variable identifier that this constraint functionally defines.

- The "output" property is a list of all identifiers for which the solver must produce output for each solution. The output must follow the normal FlatZinc output rules as defined in 节 4.3.3.
- The "solve" property maps to an object with a property "method" that indicates whether the problem is a satisfaction problem ("satisfy") or an optimisation problem ("minimize" or "maximize"), in which case the "objective" property must be present, which maps to an identifier.
- Variables, arrays, constraints and the solve item can have an optional list of annotations, indicated by the "ann" property. Atomic annotations (such as indomain_min) are represented simply as strings. Non-atomic annotations (such as int_search) follow the same syntax as constraints, i.e., an object with "id" and "args" properties.
- String literals can occur in FlatZinc as arguments to annotations or special solver-defined predicates. In FlatZinc-JSON, these string literals are represented as objects { "string" : "<string contents>" }.

The full schema is listed below, and is available from https://www.minizinc.org/schemas/fznjson.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "$id": "https://www.minizinc.org/schemas/fznjson",
  "title" : "FlatZincJSON",
  "description" : "A JSON representation of a FlatZinc model",

  "$defs" : {
    "identifier" : { "type" : "string", "pattern" : "[A-Za-z][A-Za-z0-9_]*" },
    "literal" : { "oneOf" : [
      { "type" : "number" },
      { "$ref" : "#/$defs/identifier" },
      { "type" : "boolean" },
      { "type" : "object",
        "properties" : {
          "set" : {
            "type" : "array",
            "items" : {
                        "type" : "array",
                        "items" : [
                          { "type" : "number" },
                          { "type" : "number" }
                        ]
                      }
          }
        },
        "required" : [ "set" ]
      },
      { "type" : "object",
        "properties" : {
          "string" : { "type" : "string" }
        },
        "required" : [ "string" ]
      }
    ]},
    "literals" : { "type" : "array", "items" : { "$ref" : "#/$defs/literal" } },
    "argument" : { "oneOf" : [
```

```
      { "$ref" : "#/$defs/literals"},
      { "$ref" : "#/$defs/literal" }
    ]},
    "annotation" : { "oneOf" : [
      { "$ref" : "#/$defs/annotationCall" },
      { "type" : "string" }
    ]},
    "annotations" : {
      "type" : "array",
      "items" : { "$ref" : "#/$defs/annotation" }
    },
    "annotationArgument" : { "oneOf" : [
      { "$ref" : "#/$defs/annotationLiterals"},
      { "$ref" : "#/$defs/annotationLiteral" }
    ]},
    "annotationCall" : {
      "type" : "object",
      "properties" : {
        "id": { "$ref": "#/$defs/identifier" },
        "args": {
          "type": "array",
          "items": { "$ref" : "#/$defs/annotationArgument" }
        }
      },
      "required": [
        "id",
        "args"
      ]
    },
    "annotationLiterals" : { "type" : "array", "items" : { "$ref" : "#/$defs/
↪annotationLiteral" } },
    "annotationLiteral" : { "oneOf" : [
      { "$ref" : "#/$defs/literal"},
      { "$ref" : "#/$defs/annotationCall" }
    ]},
    "domain" : {
      "type" : "array",
      "items" : {
        "type" : "array",
        "items" : [
          { "type" : "number" },
          { "type" : "number" }
        ]
      }
    }
  },

  "type": "object",
  "properties": {
    "version" : { "type" : "string" },
    "variables": {
      "type": "object",
      "patternProperties": {
        "[A-Za-z][A-Za-z0-9_]*": {
          "type": "object",
          "properties": {
            "type": {
```

```
              "enum": [
                "bool",
                "float",
                "int",
                "set of int"
               ]
            },
            "domain": { "$ref" : "#/$defs/domain" },
            "introduced": {
              "type": "boolean"
            },
            "defined": {
              "type": "boolean"
            },
            "ann" : { "$ref" : "#/$defs/annotations"}
          },
          "required": [
            "type"
          ]
        }
      }
    },
    "arrays": {
      "type": "object",
      "patternProperties": {
        "[A-Za-z][A-Za-z0-9_]*": {
          "type": "object",
          "properties": {
            "a": {
              "type": "array",
              "items": { "$ref" : "#/$defs/literal" }
            },
            "ann" : { "$ref" : "#/$defs/annotations"},
            "introduced": {
              "type": "boolean"
            },
            "defined": {
              "type": "boolean"
            }
          },
          "required": [
            "a"
          ]
        }
      }
    },
    "constraints": {
      "type": "array",
      "items": {
        "type" : "object",
        "properties" : {
          "id": { "$ref": "#/$defs/identifier" },
          "args": {
            "type": "array",
            "items": { "$ref" : "#/$defs/argument" }
          },
          "ann" : { "$ref" : "#/$defs/annotations"},
```

```
            "defines": { "$ref": "#/$defs/identifier" }
          },
          "required": [
            "id",
            "args"
          ]
        }
      },
      "output": {
        "type": "array",
        "items": { "$ref" : "#/$defs/identifier" }
      },
      "solve": {
        "type": "object",
        "properties": {
          "method": {
            "enum": [ "satisfy", "minimize", "maximize" ]
          },
          "objective": { "$ref" : "#/$defs/literal" },
          "ann" : { "$ref" : "#/$defs/annotations"}
      },
        "required": [
          "method"
        ]
      }
    },
    "required": [
      "version",
      "variables",
      "arrays",
      "output",
      "constraints",
      "solve"
    ]
  }
```

4.3.3 Output

An implementation can produce three types of output: solutions, statistics, and errors.

4.3.3.1 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) in a FlatZinc model, or those variables and arrays whose identifiers are listed in the `"output"` property of a FlatZinc-JSON model. 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.
array [1..4] of var int: zs :: output_array([1..4]);
```

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> = [<y1>, <y2>, ... <yk>];
```

where `<y1>`, `<y2>`, `...` `<yk>` are literals of the element type.

Note: The argument to the `output_array` annotation is kept for compatibility with older versions of FlatZinc. Previous versions of the MiniZinc compiler generated multi-dimensional `output_array` annotations, and required the solver to output corresponding `arrayNd` calls (with `N` being the number of dimensions). Starting from MiniZinc version 2.7.8, all `output_array` annotations are guaranteed to be one-dimensional and 1-based, so that no `arrayNd` calls are required.

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 节 4.3.5).

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).

4.3.3.2 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. Statistics output corresponding to a solution should be the last one before its '----------' separator.

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
```

Example

```
%%%mzn-stat: objective=1e+308
%%%mzn-stat: objectiveBound=0
%%%mzn-stat: nodes=0
%%%mzn-stat: solveTime=2.3567
%%%mzn-stat-end

(no feasible solution found yet but something can be printed...)

%%%mzn-stat: objective=12345
%%%mzn-stat: objectiveBound=122
%%%mzn-stat: nodes=35
%%%mzn-stat: solveTime=78.5799
%%%mzn-stat-end

(the corresponding feasible solution with value 12345 goes here
   or before its statistics but above the separator)
----------                (<- the solution separator)

%%%mzn-stat: objective=379
```

```
%%%mzn-stat: objectiveBound=379
%%%mzn-stat: nodes=4725
%%%mzn-stat: solveTime=178.5799
%%%mzn-stat-end

(the corr. optimal solution with value 379 goes here)
----------
==========                (<- the 'search complete' marker)

%%%mzn-stat: objective=379      (<- this is the concluding output)
%%%mzn-stat: objectiveBound=379
%%%mzn-stat: nodes=13456
%%%mzn-stat: solveTime=2378.5799
%%%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
`openNodes`	int	Number of open search nodes
`objective`	float	Current objective value
`objectiveBound`	float	Dual bound on the objective value
`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)

4.3.3.3 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.

4.3.4 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 节 4.3.6.

4.3.4.1 Standard predicates

FlatZinc solvers need to support the predicates listed as `FlatZinc builtins` in the library reference documentation, see 节 4.2.3.

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 (节 4.2.3) 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);
```

4.3.4.2 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 `fzn_all_different_int.mzn`, with the following implementation:

```
predicate fzn_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, `fzn_all_different_int.mzn`, in its library, and redefine it as follows:

```
predicate optisolve_alldifferent(array[int] of var int: x);

predicate fzn_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, the generated FlatZinc will contain calls to the newly defined predicate `optisolve_alldifferent`.

Note: The solver-specific library has been reorganised for MiniZinc version 2.3.0. Previously, a solver library would contain e.g. the file `bin_packing.mzn` in order to override the `bin_packing` constraint. With version 2.3.0, this is still possible (in order to maintain backwards compatibility). However, the predicate `bin_packing` from file `bin_packing.mzn` now delegates to the predicate `fzn_bin_packing` in `fzn_bin_packing.mzn`. This enables the `bin_packing` predicate to check that the arguments are correct using assertions, before delegating to the solver-specific predicate. If your solver still uses the old library layout (i.e., overriding `bin_packing.mzn` instead of `fzn_bin_packing.mzn`), you should consider updating it to the new standard.

4.3.4.3 Reified and half-reified predicates

A reified constraint is a constraint that is not simply enforced, but whose truth value is bound to a Boolean variable. For example, a MiniZinc expression `var bool: b = all_different(x);` would constrain `b` to be true if and only if the variables `x` take pairwise different values.

If a predicate is called in such a reified context, the MiniZinc compiler will try to find a version of the predicate with `_reif` added to its identifier and an additional `var bool` argument. For the above example, the compiler will try to generate the following FlatZinc code:

```
var bool: b;
constraint all_different_reif(x, b);
```

If the `_reif` predicate does not exist, the compiler will try to use the definition of the original predicate. However, this may not be ideal: the original definition may make use of free variables in a `let` expression (which is not allowed in reified contexts), or it may lead to inefficient solving.

Solver libraries should therefore provide reified versions of constraints whenever possible. The library contains files `fzn_<constraintname>_reif.mzn` for this purpose.

When a reified constraint is used in a *positive context* (see `pred-context`), the MiniZinc compiler can use a special version, called a half-reified predicate and identified by an `_imp` suffix, instead of the `_reif` predicate. Half-reified predicates essentially represent constraints that are *implied* by a Boolean variable rather than being equivalent to one. This typically leads to simpler translations or more efficient propagation (e.g., a half-reified `all_different` only needs to *check* whether it is false, but it never has to implement the negation of the actual constraint).

For example, `constraint y=0 \/ all_different(x)` might be translated as follows:

```
var bool: X_INTRODUCED_1;
var bool: X_INTRODUCED_2;
constraint int_eq_imp(y,0,X_INTRODUCED_1);
constraint all_different_imp(x, X_INTRODUCED_2);
constraint bool_clause([X_INTRODUCED_1,X_INTRODUCED_2]);
```

MiniZinc will decide whether to use half-reification case by case based on the availability of the `_imp` predicate. As for reified constraints, it may be benefitial to provide specialised half-reified versions if the solver supports them.

4.3.5 Command-Line Interface and Standard Options

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).

`-i`

Instructs the solver to print *intermediate* solutions of increasing quality (only used with optimisation problems). This option should be supported rather than `-a` for solvers which only support printing of intermediate solutions for optimisation problems but no reporting of all solutions for 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 节 4.3.3.2 for the 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).

`-t <ms>`

Wall time limit `ms` milliseconds.

4.3.6 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. Examples are given in 节 3.5.

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 节 3.1.4)
- Alternatively, you can use the MiniZinc IDE to create solver configuration files, see 节 3.2.5.2 for details.

You can also query the `minizinc` driver about these directories using the `--config-dirs` command line option.

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 or list of strings, 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), an absolute path to the executable, or a relative path (which is interpreted relative to the location of the configuration file). When a list of strings is provided, the first string is treated as the executable and consecutive strings are treated as arguments or flags that are always passed to the executable (i.e., arguments and flags that are not configurable by the solver user).
- `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`, `-i`, `-s`, `-v`, `-p`, `-r`, `-f`, `-t`.
- `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"`, `"string"` or `"opt"`). The fourth string is the default value. The following types have an additional extended syntax:
  - `"int:n:m"` where `n` and `m` are integers, gives lower and upper bounds for the supported values
  - `"float:n:m"` where `n` and `m` are floating point numbers, gives lower and upper bounds for the supported values
  - `"bool:onstring:offstring"` specifies strings to add to the command line flag to turn it on (`onstring`) and off (`offstring`). E.g., `["-interrupt","whether to catch Ctrl-C","bool:yes:no","yes"]` specifies a command line option that can be called as `-interrupt yes` or `-interrupt no`. The standard behaviour (just `"bool"`) means that the option is either added to the command line or not.
  - `"opt:first option:second option:...:last option"` specifies a list of possible values for the option
- `inputType` (string, default `FZN`, options `MZN`, `FZN`, `NL`, `JSON`): The type of input expected by the solver.
- `supportsMzn` (bool, default `false`): Whether the solver can run MiniZinc directly (i.e., it implements its own compilation or interpretation of the model). [Deprecated, use `inputType`]
- `supportsFzn` (bool, default `true`): Whether the solver can run FlatZinc. This should be the case for most solvers. [Deprecated, use `inputType`]
- `supportsNL` (bool, default `false`): Whether the solver supports NL input. Used for non-linear solvers. [Deprecated, use `inputType`]
- `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.

4.3.7 Grammar

This is the full grammar for FlatZinc. It is a proper subset of the MiniZinc grammar (see 节 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 节 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"

<basic-var-type> ::= "var" <basic-par-type>
                   | "var" <int-literal> ".." <int-literal>
                   | "var" "{" <int-literal> "," ... "}"
                   | "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> "," ... ] "}"

<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>

<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> ::= "array" "[" <index-set> "]" "of" <basic-par-type> ":" <var-par-
↪identifier> "=" <par-array-literal> ";"

% 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>   := <basic-ann-expr>
              | "[" [ <basic-ann-expr> "," ... ] "]"

<basic-ann-expr>   := <basic-literal-expr>
                    | <var-par-identifier>
                    | <string-literal>
                    | <annotation>

<string-contents> ::= ([^"\n\] | \[^\n(])*

<string-literal> ::= """ <string-contents> """

% End of FlatZinc grammar
```

CHAPTER 4.4

Machine-readable JSON output format

MiniZinc supports a machine-readable output format activated with the `--json-stream` (see *Options* (page 127)) option.

In this mode, MiniZinc outputs a line-break delimited stream of JSON objects (called messages). These are output to the standard output stream. The standard error stream is not affected by this mode. Each message contains the key `type` with a string value denoting the kind of message.

A note on the message format

In the following sections, messages are formatted on multiple lines for readability, however actual messages are always a single line.

4.4.1 Solution messages

```
{
  "type": "solution",
  "time": 1000,
  "output": {
    "foo": "foo output section",
    "bar": "bar output section"
  },
  "sections": ["foo", "bar"]
}
```

This message is produced whenever a solution is output (at the point where a `----------` would typically be produced).

The `time` field is optional and is only present if run with `--output-time`. It gives the elapsed time in milliseconds.

The `output` field contains an object mapping output section names to their contents. Output section contents can be either:

- A string containing the output value
- A JSON value if the section is the `json` section (generated when using `--output-mode json`), or if the section was specified using the `json_section` annotation.

The `sections` field contain a list of output section names (strings) to show the order that they were produced.

Note that all sections are included in the `output` key object. The resulting output for enabled sections is available in the `raw` section.

4.4.2 Checker messages

```
{
  "type": "checker",
  "messages": [
    {
      "type": "solution",
      "output": {
        "default": "foo output section"
      },
      "sections": ["default"]
    }
  ]
}
```

This message gives solution checker results and is produced directly preceding the solution it is associated with. The `messages` field gives a list of JSON stream messages produced by the solution checker (usually a `solution` message).

4.4.3 Status messages

```
{
  "type": "status",
  "status": "ALL_SOLUTIONS",
  "time": 1000
}
```

This message is produced when a final status message is received from a solver (at the point where a status line such as `==========` would be produced).

The `status` field contains the name of the solver status and can take one of the follow values:

- `"ALL_SOLUTIONS"`
- `"OPTIMAL_SOLUTION"`
- `"UNSATISFIABLE"`

- "UNBOUNDED"
- "UNSAT_OR_UNBOUNDED"
- "UNKNOWN"
- "ERROR"

The `time` field is optional and is only present if run with `--output-time`. It gives the elapsed time in milliseconds.

4.4.4 Statistics messages

```
{
  "type": "statistics",
  "statistics": {
    "method": "satisfy",
    "flatTime": 1000
  }
}
```

This message is produced whenever a set of statistics is output (at the point where `%%%mzn-stat-end` would typically be produced).

The `statistics` field contains an object mapping statistics names to their values.

4.4.5 Timestamp messages

```
{
  "type": "time",
  "time": 1000
}
```

This message is produced to indicate the current solve time in a standalone way. For example, when running with `--canonicalize` and `--output-time`, solution messages are printed at the end, however as solutions are produced, `time` messages are output.

The `time` field gives the elapsed time in milliseconds.

4.4.6 Comment messages

```
{
  "type": "comment",
  "comment": "% comment produced by solver\n"
}
```

This message is produced when a solver outputs a comment.

The `comment` field contains the comment as a string (including the leading `%` and trailing newline).

4.4.7 Trace messages

```
{
  "type": "trace",
  "section": "default",
  "message": "traced message\n"
}
```

This message is produced when a `trace_stdout`, `trace_to_section()` or similar call is evaluated.

The `section` field gives the section name as a string (`default` when using `trace_stdout`).

The `message` field gives the message contents as a string, or as a JSON value if appropriate (such as when using `trace_exp`).

Note that `trace()` calls output to standard error as normal.

4.4.8 Profiling messages

```
{
  "type": "profiling",
  "entries": [...]
}
```

This message is produced when `--output-detailed-timing`

The `entries` field contains a list of timing information entry objects.

Entries have the following format:

```
{
  "filename": "model.mzn",
  "line": 1,
  "time": 100
}
```

4.4.9 Paths messages

```
{
  "type": "paths",
  "paths": [...]
}
```

This message is produced when `--output-paths-to-stdout` is used.

The `paths` field contains a list of objects representing either a variable path or a constraint path.

Variable paths have the following format:

```
{
  "flatZincName": "X_INTRODUCED_0_",
  "niceName": "x[1]",
  "path": "model.mzn|1|27|1|27|id|x;|0|0|0|0|il|0;"
}
```

Constraint paths have the following format:

```
{
  "constraintIndex": 9,
  "path": "model.mzn|3|12|3|59|ca|forall;model.mzn|3|12|3|59|ac;model.
↪mzn|3|20|3|20|i=4;model.mzn|3|23|3|23|j=5;model.mzn|3|47|3|58|bin|'!=';model.
↪mzn|3|47|3|58|ca|int_lin_ne;"
}
```

4.4.10 Error messages

```
{
  "type": "error",
  "what": "type error",
  "location": {...},
  "message": "cannot determine coercion from type float to type var int"
}
```

This message is produced when an error occurs.

The `what` field contains the kind of error as a string.

The `message` field contains the error message as a string.

Error messages can also optionally contain one or both of:

- A `location` field containing a location object as described in *Locations* (page 491).
- A `stack` field containing a list of stack trace objects as described in *Stack traces* (page 492).

Some kinds of error messages have additional properties (depending on the `what` of the error message):

- The `syntax error` message can optionally contain the field `includedFrom` giving a list of file names.
- The `cyclic include error` message contains the field `cycle` giving a list of file names.

4.4.10.1 Locations

Locations in a file are represented as follows:

```
{
  "filename": "model.mzn",
  "firstLine": 1,
  "firstColumn": 1,
```

```
  "lastLine": 3,
  "lastColumn": 10
}
```

4.4.10.2 Stack traces

Stack traces are represented as a list of objects with the following format:

```
{
  "location": {...}
  "isCompIter": false,
  "description": "variable declaration"
}
```

4.4.11 Warning messages

```
{
  "type": "warning",
  "location": {...},
  "stack": [...],
  "message": "Warning message"
}
```

This message is produced when a warning is emitted.

The optional `location` field contains the relevant location (see *Locations* (page 491)) if present.

The optional `stack` field contains a stack trace (see *Stack traces* (page 492)) if present.

The `message` field contains the warning message as a string.

Note that if `-Werror` is used then `error` messages are produced instead.