

# **Specification of MiniZinc**

The MiniZinc Team

Data61/CSIRO  
Monash University  
The University of Melbourne  
Melbourne, Australia

November 2016

(MiniZinc version 2.1.5)

# Contents

<b>1. Introduction</b>	<b>4</b>
<b>2. Notation</b>	<b>4</b>
<b>3. Overview of a Model</b>	<b>5</b>
3.1. Evaluation Phases . . . . .	5
3.2. Run-time Outcomes . . . . .	5
3.3. Output . . . . .	5
<b>4. Syntax Overview</b>	<b>6</b>
4.1. Character Set . . . . .	6
4.2. Comments . . . . .	6
4.3. Identifiers . . . . .	6
<b>5. High-level Model Structure</b>	<b>6</b>
5.1. Items . . . . .	6
5.2. Model Instance Files . . . . .	7
5.3. Namespaces . . . . .	7
5.4. Scopes . . . . .	7
<b>6. Types and Type-insts</b>	<b>8</b>
6.1. Properties of Types . . . . .	8
6.2. Instantiations . . . . .	8
6.3. Type-insts . . . . .	8
6.4. Type-inst Expressions Overview . . . . .	8
6.5. Built-in Scalar Types and Type-insts . . . . .	9
6.6. Built-in Compound Types and Type-insts . . . . .	10
6.7. Constrained Type-insts . . . . .	12
<b>7. Expressions</b>	<b>13</b>
7.1. Expressions Overview . . . . .	13
7.2. Operators . . . . .	14
7.3. Expression Atoms . . . . .	15
<b>8. Items</b>	<b>20</b>
8.1. Include Items . . . . .	21
8.2. Variable Declaration Items . . . . .	21
8.3. Enum Items . . . . .	21
8.4. Assignment Items . . . . .	22
8.5. Constraint Items . . . . .	22
8.6. Solve Items . . . . .	22
8.7. Output Items . . . . .	23
8.8. Annotation Items . . . . .	23
8.9. User-defined Operations . . . . .	23
<b>9. Annotations</b>	<b>25</b>
<b>10. Partiality</b>	<b>25</b>
10.1. Partial Assignments . . . . .	26
10.2. Partial Predicate/Function and Annotation Arguments . . . . .	26
10.3. Partial Array Accesses . . . . .	27
<b>A. Built-in Operations</b>	<b>28</b>
A.1. Comparison Operations . . . . .	28
A.2. Arithmetic Operations . . . . .	28
A.3. Logical Operations . . . . .	29
A.4. Set Operations . . . . .	29
A.5. Array Operations . . . . .	30
A.6. Coercion Operations . . . . .	30
A.7. String Operations . . . . .	31
A.8. Bound and Domain Operations . . . . .	31
A.9. Option Type Operations . . . . .	32
A.10. Other Operations . . . . .	32

<b>B. MiniZinc Grammar</b>	<b>33</b>
B.1. Items . . . . .	33
B.2. Type-Inst Expressions . . . . .	33
B.3. Expressions . . . . .	34
B.4. Miscellaneous Elements . . . . .	35
<b>C. Content-types</b>	<b>36</b>
C.1. ‘application/x-zinc-output’ . . . . .	36
<b>D. JSON support</b>	<b>37</b>

# 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. Section 2 introduces the syntax notation used throughout the specification. Section 3 provides a high-level overview of MiniZinc models. Section 4 covers syntax basics. Section 5 covers high-level structure: items, multi-file models, namespaces, and scopes. Section 6 introduces types and type-insts. Section 7 covers expressions. Section 8 describes the top-level items in detail. Section 9 describes annotations. Section 10 describes how partiality is handled in various cases. Appendix A describes the language built-ins. Appendix B gives the MiniZinc grammar. Appendix C 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.*

**Original authors.** The original version of this document was prepared by Nicholas Nethercote, Kim Marriott, Reza Rafeh, Mark Wallace and María García 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. García de la Banda, and M. Wallace. The Design of the Zinc Modelling Language. *Constraints*, 13(3):229–267, 2008.

## 2. Notation

The basics of the EBNF used in this specification are as follows.

- Non-terminals are written between angle brackets, e.g.  $\langle item \rangle$ .
- Terminals are written in fixed-width font. They are usually underlined for emphasis, e.g. constraint, although this is not always done if the meaning is clear.
- 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 )<sup>\*</sup>.
- Sequences of one or more items are written with parentheses and a plus, e.g. ( msg )<sup>+</sup>.
- Non-empty lists are written with an item, a separator/terminator terminal, and “...”. For example, this:

$\langle expr \rangle$  , ...

is short for this:

$\langle expr \rangle$  ( ,  $\langle expr \rangle$  )<sup>\*</sup> [ , ]

The final terminal is always optional in non-empty lists.

- Regular expressions, written in fixed-width font, 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 Appendix B. 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`.
- A sequence of space characters of length  $n$  is written  $n$ SP, e.g., 2SP.

### 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 an *model instance* (sometimes abbreviated to *instance*).

The model and data may be separated, or the data may be “hard-wired” into the model. Section 5.2 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. Section 8.6 specifies how the class of problem is chosen.

#### 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.

#### 3.2. Run-time Outcomes

Assuming there are no static errors, the output from the run-time phase has the following abstract form:

$$\langle output \rangle ::= \langle no-solutions \rangle [ \langle warnings \rangle ] \langle free-text \rangle \\ | ( \langle solution \rangle )^* [ \langle complete \rangle ] [ \langle warnings \rangle ] \langle free-text \rangle$$

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.

#### 3.3. Output

Implementations must be capable of producing output of content type ‘application/x-zinc-output’, which is described below and also in Appendix C. 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:

$$\langle solution \rangle ::= \langle solution-text \rangle [ \backslash n ] \text{-----} \backslash n$$

The solution text for each solution must be as described in Section 8.7. 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.

$$\langle no-solutions \rangle ::= \text{====UNSATISFIABLE=====} \backslash n$$

The completeness result is printed on a separate line. **Rationale.** *The strings are designed to clearly indicate the end of the solutions.*

$$\langle complete \rangle ::= \text{=====} \backslash n$$

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.*

```
 $\langle warnings \rangle ::= ( \langle message \rangle ) +$ 
```

```
 $\langle message \rangle ::= ( \langle line \rangle ) +$ 
```

```
 $\langle line \rangle ::= \% [^\backslash n]^* \backslash n$ 
```

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. Syntax Overview

### 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.

### 4.2. 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.

### 4.3. Identifiers

Identifiers have the following syntax:

```
 $\langle ident \rangle ::= [A-Za-z][A-Za-z0-9\_]* \quad \% \text{ excluding keywords}$ 
 $| \_ '[^'\backslash xa\backslash xd\backslash x0]^*'$ 
```

For example:

```
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`, `output`, `par`, `predicate`, `record`, `satisfy`, `set`, `solve`, `string`, `subset`, `superset`, `syndiff`, `test`, `then`, `true`, `tuple`, `type`, `union`, `var`, `where`, `xor`.

A number of identifiers are used for built-ins; see Section A for details.

## 5. High-level Model Structure

### 5.1. Items

A MiniZinc model consists of multiple *items*:

```
 $\langle model \rangle ::= [ \langle item \rangle ; \dots ]$ 
```

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>
        | <assign-item>
        | <constraint-item>
        | <solve-item>
        | <output-item>
        | <predicate-item>
        | <test-item>
        | <function-item>
        | <annotation-item>

```

Include items provide a way of combining multiple files into a single instance. This allows a model to be split into multiple files (Section 8.1).

Variable declaration items introduce new global variables and possibly bind them to a value (Section 8.2).

Assignment items bind values to global variables (Section 8.4).

Constraint items describe model constraints (Section 8.5).

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 (Section 8.6).

Output items are used for nicely presenting the result of a model execution (Section 8.7).

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 (Section 8.9). 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.

## 5.2. Model Instance Files

MiniZinc models can be constructed from multiple files using include items (see Section 8.1). 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 Section 8.4).

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.

## 5.3. 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 (Section 5.2), all files share this top-level namespace. Therefore a variable `v` 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.

## 5.4. Scopes

Within the top-level namespace, there are several kinds of local scope that introduce local names:

- Comprehension expressions (Section 7.3.7).
- Let expressions (Section 7.3.14).
- Function and predicate argument lists and bodies (Section 8.9).

The listed sections specify these scopes in more detail. In each case, any names declared in the local scope overshadow identical global names.

## 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.

### 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, types defined via set expression type-insts such as range types (see Section 6.7.1), 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 (Section 6.6.1) and array indices (Section 6.6.2).

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.

### 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. Arrays and `ann` can contain decision variables.

### 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 array accesses.

### 6.4. Type-inst Expressions 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 (Section 8.2), user-defined operation items (Section 8.9).

Type-inst expressions have this syntax:



$$\begin{aligned}
\langle ti\text{-}expr \rangle &::= \langle base\text{-}ti\text{-}expr \rangle \\
\langle base\text{-}ti\text{-}expr \rangle &::= \langle var\text{-}par \rangle \langle base\text{-}ti\text{-}expr\text{-}tail \rangle \\
\langle var\text{-}par \rangle &::= \underline{\text{var}} \mid \underline{\text{par}} \mid \epsilon \\
\langle base\text{-}ti\text{-}expr\text{-}tail \rangle &::= \langle ident \rangle \\
&\mid \underline{\text{bool}} \\
&\mid \underline{\text{int}} \\
&\mid \underline{\text{float}} \\
&\mid \underline{\text{string}} \\
&\mid \langle set\text{-}ti\text{-}expr\text{-}tail \rangle \\
&\mid \langle array\text{-}ti\text{-}expr\text{-}tail \rangle \\
&\mid \underline{\text{ann}} \\
&\mid \underline{\text{opt}} \langle base\text{-}ti\text{-}expr\text{-}tail \rangle \\
&\mid \{ \langle expr \rangle , \dots \} \\
&\mid \langle num\text{-}expr \rangle \underline{\dots} \langle num\text{-}expr \rangle
\end{aligned}$$

(The final alternative, for range types, uses the numeric-specific  $\langle num\text{-}expr \rangle$  non-terminal, defined in Section 7.1, rather than the  $\langle expr \rangle$  non-terminal. If this were not the case, the rule would never match because the ‘ $\dots$ ’ operator would always be matched by the first  $\langle expr \rangle$ .)

This fully covers the type-inst expressions for scalar types. The compound type-inst expression syntax is covered in more detail in Section 6.6.

The **par** and **var** keywords (or lack of them) determine the instantiation. The **par** annotation can be omitted; the following two type-inst expressions are equivalent:

```

par int
int

```

**Rationale.** The use of the explicit **var** keyword allows an implementation to check that all parameters are initialised in the model or the instance. It also clearly documents which variables are parameters, and allows more precise type-inst checking.

A type-inst is fixed if it does not contain **var** or **any**, with the exception of **ann**.

Note that several type-inst expressions that are syntactically expressible represent illegal type-insts. For example, although the grammar allows **var** in front of all these base type-inst expression tails, it is a type-inst error to have **var** in the front of a string or array expression.

## 6.5. Built-in Scalar Types and Type-insts

### 6.5.1. 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  $\{\text{false}, \text{true}\}$ .

*Variable?* `par bool`  $\xrightarrow{v}$  `var bool`, `var bool`  $\xrightarrow{v}$  `var bool`.

*Ordering.* The value `false` is considered smaller than `true`.

*Initialisation.* A fixed Boolean variable must be initialised at instance-time; an unfixed Boolean variable need not be.

*Coercions.* `par bool`  $\xrightarrow{c}$  `var bool`.

Also Booleans can be automatically coerced to integers; see Section 6.5.2.

### 6.5.2. 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 Section 6.7.1).

*Variable?* `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 Section 6.5.3.

### 6.5.3. 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 NaNs, 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 Section 6.7.1).

*Variable?* `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`.

### 6.5.4. 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.

*Variable?* `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`.

### 6.5.5. 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 Section 6.7.1).

*Variable?* 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 Section 7.3.17).

## 6.6. Built-in Compound Types and Type-insts

### 6.6.1. Sets

*Overview.* A set is a collection with no duplicates.

*Allowed Insts.* The type-inst of a set’s elements must be fixed. ***Rationale.*** *This is because current solvers are not powerful enough to handle sets containing decision variables.* Sets may contain any type, and may be fixed or unfixed. If a set is unfixed, its elements must be finite, unless it occurs in one of the following contexts:

- the argument of a predicate, function or annotation.
- the declaration of a variable or let local variable with an assigned value.

*Syntax.* A set base type-inst expression tail has this syntax:

$\langle \text{set-ti-expr-tail} \rangle ::= \text{set of } \langle \text{base-type} \rangle$

Some example set type-inst expressions:

```

set of int
var set of bool

```

*Finite?* Yes, if the set elements are finite types. Otherwise, no.

The domain of a set type that is a finite type is the powerset of the domain of its element type. For example, the domain of `set of 1..2` is `powerset(1..2)`, which is  $\{\{\}, \{1\}, \{1,2\}, \{2\}\}$ .

*Variable?* `par set of TI`  $\xrightarrow{v}$  `var set of TI`, `var set of TI`  $\xrightarrow{v}$  `var set of TI`,

*Ordering.* The pre-defined ordering on sets is a lexicographic ordering of the *sorted set form*, where  $\{1,2\}$  is in sorted set form, for example, but  $\{2,1\}$  is not. This means, for instance,  $\{\} < \{1,3\} < \{2\}$ .

*Initialisation.* A fixed set variable must be initialised at instance-time; an unfixed set variable need not be.

*Coercions.* `par set of TI`  $\xrightarrow{c}$  `par set of UI` and `par set of TI`  $\xrightarrow{c}$  `var set of UI` and `var set of TI`  $\xrightarrow{c}$  `var set of UI`, if `TI`  $\xrightarrow{c}$  `UI`.

### 6.6.2. 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.

1. *Explicitly-indexed* arrays have index types in the declaration that are finite types. For example:

```

array[0..3] of int: a1;
array[1..5, 1..10] of var float: a5;

```

For such arrays, the index type specifies exactly the indices that will be in the array—the array’s index set is the *domain* of the index type—and if the indices of the value assigned do not match then it is a run-time error.

For example, the following assignments cause run-time errors:

```

a1 = [4,6,4,3,2];    % too many elements
a5 = [];              % too few elements

```

2. *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 Section 7.3.11 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:

$$\langle \text{array-ti-expr-tail} \rangle ::= \underline{\text{array}} \ [ \ \langle \text{ti-expr} \rangle \ , \ \dots \ ] \ \underline{\text{of}} \ \langle \text{ti-expr} \rangle$$

$$| \ \underline{\text{list of}} \ \langle \text{ti-expr} \rangle$$

Some example array type-inst expressions:

```

array[1..10] of int
list of var int

```

Note that `list of <T>` is just syntactic sugar for `array[int] of <T>`. **Rationale.** Integer-indexed arrays of this form are very common, and so worthy of special support to make things easier for modellers. Implementing it using syntactic sugar avoids adding an extra type to the language, which keeps things simple for implementers.

Because arrays must be fixed-size it is a type-inst error to precede an array type-inst expression with `var`.

*Finite?* Yes, if the index types and element type are all finite types. Otherwise, no.

The domain of an array type that is a finite array is the set of all distinct arrays whose index set equals the domain of the index type and whose elements are of the array element type.

*Variable?* No.

*Ordering.* Arrays are ordered lexicographically, taking absence of a value for a given key to be before any value for that key. For example, `[1, 1]` is less than `[1, 2]`, which is less than `[1, 2, 3]` and `array1d(2..4, [0, 0, 0])` is less than `[1, 2, 3]`.

*Initialisation.* An explicitly-indexed array variable must be initialised at instance-time only if its elements must be initialised at instance time. An implicitly-indexed array variable must be initialised at instance-time so that its length and index set is known.

*Coercions.* `array[TIO] of TI`  $\xrightarrow{c}$  `array[UIO] of UI` if `TI`  $\xrightarrow{c}$  `UI` and `TIO`  $\xrightarrow{c}$  `UIO`.

### 6.6.3. 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.

*Variable?* 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`.

### 6.6.4. The Annotation Type `ann`

*Overview.* The annotation type, `ann`, can be used to represent arbitrary term structures. It is augmented by annotation items (8.8).

*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.

*Variable?* 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.

## 6.7. Constrained Type-insts

One powerful feature of MiniZinc is *constrained type-insts*. A constrained type-inst is a restricted version of a *base* type-inst, i.e. a type-inst with fewer values in its domain.

### 6.7.1. 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.

## 6.7.2. 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.

# 7. Expressions

## 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⟩
                    | ⟨simple-array-literal⟩
                    | ⟨simple-array-literal-2d⟩
                    | ⟨indexed-array-literal⟩
                    | ⟨simple-array-comp⟩
                    | ⟨indexed-array-comp⟩
                    | ⟨ann-literal⟩
                    | ⟨if-then-else-expr⟩
                    | ⟨let-expr⟩
                    | ⟨call-expr⟩
                    | ⟨gen-call-expr⟩
⟨expr-atom-tail⟩ ::= ε
                    | ⟨array-access-tail⟩ ⟨expr-atom-tail⟩
```

Expressions can be composed from sub-expressions combined with operators. All operators (binary and unary) are described in Section 7.2, 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;
```

Section 9 has more on annotations.

Expressions can be contained within parentheses.

The array access operations all bind more tightly than unary and binary operators and annotations. They are described in more detail in Sections 7.3.11.

The remaining kinds of expression atoms (from `⟨ident⟩` to `⟨gen-call-expr⟩`) are described in Sections 7.3.1–7.3.16.

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⟩
```

Operator	Unicode symbol	UTF-8 code
<code>&lt;-&gt;</code>	↔	E2 86 94
<code>-&gt;</code>	→	E2 86 92
<code>&lt;-</code>	←	E2 86 90
<code>not</code>	¬	C2 AC
<code>\ </code>	√	E2 88 A8
<code>/\</code>	∧	E2 88 A7
<code>!=</code>	≠	E2 89 A0
<code>&lt;=</code>	≤	E2 89 A4
<code>&gt;=</code>	≥	E2 89 A5
<code>in</code>	∈	E2 88 88
<code>subset</code>	⊆	E2 8A 86
<code>superset</code>	⊇	E2 8A 87
<code>union</code>	∪	E2 88 AA
<code>intersect</code>	∩	E2 88 A9

Table 1: Unicode equivalents of binary operators.

| *<if-then-else-expr>*  
| *<case-expr>*  
| *<let-expr>*  
| *<call-expr>*  
| *<gen-call-expr>*

## 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` | `syndiff`  
| `..` | `intersect` | `++` | *<builtin-num-bin-op>*  
*<builtin-un-op>* ::= `not` | *<builtin-num-un-op>*

Again, we syntactically distinguish numeric operators.

*<num-bin-op>* ::= *<builtin-num-bin-op>*  
| *‘<ident>’*  
*<builtin-num-bin-op>* ::= `+` | `-` | `*` | `/` | `div` | `mod`  
*<builtin-num-un-op>* ::= `+` | `-`

Some operators can be written using their unicode symbols, which are listed in Table 1 (recall that MiniZinc input is UTF-8).

The binary operators are listed in Table 2.

A user-defined binary operator is created by backquoting a normal identifier, for example:

```
A ‘min2’ B
```

This is a static error if the identifier is not the name of a binary function or predicate.

The unary operators are: `+`, `-` and `not`. User-defined unary operators are not possible.

As Section 4.3 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 Section A.

Symbol(s)	Assoc.	Prec.
<->	left	1200
->	left	1100
<-	left	1100
\	left	1000
xor	left	1000
/\	left	900
<	none	800
>	none	800
<=	none	800
>=	none	800
==, =	none	800
!=	none	800
in	none	700
subset	none	700
superset	none	700
union	left	600
diff	left	600
syndiff	left	600
..	none	500
+	left	400
-	left	400
*	left	300
div	left	300
mod	left	300
/	left	300
intersect	left	300
++	right	200
'<ident>'	left	100

Table 2: Binary infix operators. 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.

## 7.3. Expression Atoms

### 7.3.1. Identifier Expressions and Quoted Operator Expressions

Identifier expressions and quoted operator expressions have the following syntax:

$$\langle \textit{ident-or-quoted-op} \rangle ::= \langle \textit{ident} \rangle \mid \_ \langle \textit{builtin-op} \rangle \_$$

Examples of identifiers were given in Section 4.3. The following are examples of quoted operators:

```
'+'
'union'
```

In quoted operators, whitespace is not permitted between either quote and the operator. Section 7.2 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.

### 7.3.2. 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.

### 7.3.3. Boolean Literals

Boolean literals have this syntax:

$$\langle \textit{bool-literal} \rangle ::= \text{false} \mid \text{true}$$

### 7.3.4. Integer and Float Literals

There are three forms of integer literals—decimal, hexadecimal, and octal—with these respective forms:

$$\langle \text{int-literal} \rangle ::= [0-9]^+ \\ \quad \quad \quad | \text{0x}[0-9\text{A-Fa-f}]^+ \\ \quad \quad \quad | \text{0o}[0-7]^+$$

For example: 0, 005, 123, 0x1b7, 0o777; but not -1.

Float literals have the following form:

$$\langle \text{float-literal} \rangle ::= [0-9]^+ \cdot [0-9]^+ \\ \quad \quad \quad | [0-9]^+ \cdot [0-9]^+ [\text{Ee}] [-+]? [0-9]^+ \\ \quad \quad \quad | [0-9]^+ [\text{Ee}] [-+]? [0-9]^+$$

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.

### 7.3.5. String Literals

String literals are written as in C:

$$\langle \text{string-contents} \rangle ::= ([^"\backslash n] \mid \backslash [^"n])^* \\ \langle \text{string-literal} \rangle ::= "\langle \text{string-contents} \rangle" \\ \quad \quad \quad | "\langle \text{string-contents} \rangle \backslash (\langle \text{string-interpolate-tail} \rangle \\ \langle \text{string-interpolate-tail} \rangle ::= \langle \text{expr} \rangle \backslash \langle \text{string-contents} \rangle" \\ \quad \quad \quad | \langle \text{expr} \rangle \backslash \langle \text{string-contents} \rangle \backslash (\langle \text{string-interpolate-tail} \rangle)$$

This includes C-style escape sequences, such as ‘\’ for double quotes, ‘\’ for backslash, and ‘\n’ for newline.

For example: "Hello, world!\n".

String literals must fit on a single line. Long string literals can be split across multiple lines using string concatenation. For example:

```
string: s = "This is a string literal "  
        ++ "split across two lines.";
```

### 7.3.6. Set Literals

Set literals have this syntax:

$$\langle \text{set-literal} \rangle ::= \{ \mid \langle \text{expr} \rangle \mid \dots \}$$

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`).

### 7.3.7. Set Comprehensions

Set comprehensions have this syntax:

$$\langle \text{set-comp} \rangle ::= \{ \langle \text{expr} \rangle \mid \langle \text{comp-tail} \rangle \} \\ \langle \text{comp-tail} \rangle ::= \langle \text{generator} \rangle \mid \dots \mid \text{where } \langle \text{expr} \rangle \\ \langle \text{generator} \rangle ::= \langle \text{ident} \rangle \mid \dots \mid \text{in } \langle \text{expr} \rangle$$

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.
- They are visible within generator 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 }
```

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. Only one where-expression per comprehension is allowed.

**Rationale.** Allowing one where-expression per generator is another possibility, and one that could seemingly result in more efficient evaluation in some cases. For example, consider the following comprehension:

```
[f(i, j) | i in A1, j in A2 where p(i) /\ q(i,j)]
```

If multiple where-expressions were allowed, this could be expressed more efficiently in the following manner, which avoids the fruitless “inner loop iterations” for each “outer loop iteration” that does not satisfy  $p(i)$ :

```
[f(i, j) | i in A1 where p(i), j in A2 where q(i,j)]
```

However, this efficiency can also be achieved with nested comprehensions:

```
[f(i, j) | i in [k | k in A1 where p(k)], j in A2 where q(i,j)]
```

Therefore, a single where-expression is all that is supported.

### 7.3.8. Array Literals

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.

### 7.3.9. 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.

### 7.3.10. Array Comprehensions

Array comprehensions have this syntax:

$$\langle \text{array-comp} \rangle ::= \underline{\underline{[ \langle \text{expr} \rangle \mid \langle \text{comp-tail} \rangle ]}}$$

For example (with the literal equivalents on the right):

```
[2*i | i in 1..5]          % [2, 4, 6, 8, 10]
```

Array comprehensions have more flexible type and inst requirements than set comprehensions (see Section 7.3.7).

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.

### 7.3.11. Array Access Expressions

Array elements are accessed using square brackets after an expression:

$$\langle \text{array-access-tail} \rangle ::= \underline{\underline{[ \langle \text{expr} \rangle , \dots ]}}$$

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
```

### 7.3.12. Annotation Literals

Literals of the `ann` type have this syntax:

$$\langle \text{ann-literal} \rangle ::= \langle \text{ident} \rangle [ \underline{\underline{[ \langle \text{expr} \rangle , \dots ]}}$$

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.

### 7.3.13. If-then-else Expressions

MiniZinc provides if-then-else expressions, which provide selection from two alternatives based on a condition. They have this syntax:

$$\langle \text{if-then-else-expr} \rangle ::= \underline{\text{if}} \langle \text{expr} \rangle \underline{\text{then}} \langle \text{expr} \rangle \\ (\underline{\text{elseif}} \langle \text{expr} \rangle \underline{\text{then}} \langle \text{expr} \rangle)^* \\ \underline{\text{else}} \langle \text{expr} \rangle \underline{\text{endif}}$$

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`.

### 7.3.14. 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:

$$\langle \text{let-expr} \rangle ::= \underline{\text{let}} \{ \langle \text{let-item} \rangle ; \dots \} \underline{\text{in}} \langle \text{expr} \rangle \\ \langle \text{let-item} \rangle ::= \langle \text{var-decl-item} \rangle \\ | \langle \text{constraint-item} \rangle$$

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 type-inst expressions can include type-inst variables if the let is within a function or predicate body in which the same type-inst variables were present in the function or predicate signature.

The initialiser for a let local variable can be omitted only if the variable is a decision variable. For example:

```
let { var int: x; } in ...; % ok
let { int: x; } in ...; % illegal
```

The type-inst of the entire let expression is the type-inst of the expression after the `in` keyword.

There is a complication involving let expressions in negative contexts. A let expression occurs in a negative context if it occurs in an expression of the form `not X`, `X <-> Y`, or in the sub-expression `X in X -> Y` or `Y <- X`, or in a subexpression `bool2int(X)`.

If a let expression is used in a negative context, then any let-local decision variables must be defined only in terms of non-local variables and parameters. This is because local variables are implicitly existentially quantified, and if the let expression occurred in a negative context then the local variables would be effectively universally quantified which is not supported by MiniZinc.

Constraints in let expressions float to the nearest enclosing Boolean context. For example

```
constraint b -> x + let { var 0..2: y; constraint y != -1;} in y >= 4;
```

is equivalent to

```
var 0..2: y;
constraint b -> (x + y >= 4 /\ y != 1);
```

### 7.3.15. Call Expressions

Call expressions are used to call predicates and functions.

Call expressions have this syntax:

$$\langle \text{call-expr} \rangle ::= \langle \text{ident-or-quoted-op} \rangle [ \_ \langle \text{expr} \rangle \_ \dots \_ ]$$

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  $\wedge$ ,  $\vee$ ,  $\rightarrow$  and  $\leftarrow$  which could be lazy in one argument. The one exception is **assert**, which is lazy in its third argument (Section A.10).

**Rationale.** Boolean operations are strict because: (a) this minimises exceptional cases; (b) in an expression like  $A \rightarrow B$ , 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.

### 7.3.16. 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:

$$\langle \text{gen-call-expr} \rangle ::= \langle \text{ident-or-quoted-op} \rangle [ \_ \langle \text{comp-tail} \rangle \_ \_ \langle \text{expr} \rangle \_ ]$$

A generator call expression  $P(\text{Gs})(E)$  is equivalent to the call expression  $P([E \mid \text{Gs}])$ . 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 (Section 7.3.10).

### 7.3.17. String Interpolation Expressions

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.

String interpolation has the following syntax:

$$\langle \text{string-contents} \rangle ::= ( [ \text{^} \text{"} \text{\_n} \_ ] \mid \_ [ \text{^} \text{"} \text{\_n} \_ ] ) *$$
$$\langle \text{string-literal} \rangle ::= " \langle \text{string-contents} \rangle "$$
$$\mid " \langle \text{string-contents} \rangle \backslash \langle \text{string-interpolate-tail} \rangle$$
$$\langle \text{string-interpolate-tail} \rangle ::= \langle \text{expr} \rangle \_ \langle \text{string-contents} \rangle "$$
$$\mid \langle \text{expr} \rangle \_ \langle \text{string-contents} \rangle \backslash \langle \text{string-interpolate-tail} \rangle$$

For example:

```
var set of 1..10: q;
solve satisfy;
output [show("The value of q is \q), and it has \card(q) elements.")];
```

## 8. Items

This section describes the top-level program items.

## 8.1. Include Items

Include items allow a model to be split across multiple files. They have this syntax:

$\langle include-item \rangle ::= \text{include } \langle string-literal \rangle$

For example:

```
include "foo.zinc";
```

includes the file `foo.zinc`.

Include items are particularly useful for accessing libraries or breaking up large models into small pieces. They are not, as Section 5.2 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.

## 8.2. Variable Declaration Items

Variable declarations have this syntax:

$\langle var-decl-item \rangle ::= \langle ti-expr-and-id \rangle \langle annotations \rangle [ \equiv \langle expr \rangle ]$

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 (Section 8.4). For example, the above item can be separated into the following two items:

```
int: A;  
...  
A = 10;
```

All variables that contain a parameter component must be defined at instance-time.

Variables can have one or more annotations. Section 9 has more on annotations.

## 8.3. Enum Items

Enumerated type items have this syntax:

$\langle enum-item \rangle ::= \text{enum } \langle ident \rangle \langle annotations \rangle [ \equiv \langle enum-cases \rangle ]$   
 $\langle enum-cases \rangle ::= \{ \langle ident \rangle , \dots \}$

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 (Section 5.3), case names in different enums must be distinct.

An enum can be declared but not defined, in which case it must be defined elsewhere within the model, or in a data file. For example, a model file could contain this:

```
enum Workers;  
enum Shifts;
```

and the data file could contain this:

```
Workers = { welder, driller, stamper };  
Shifts = { idle, day, night };
```

Sometimes it is useful to be able to refer to one of the enum case names within the model. This can be achieved by using a variable. The model would read:

```
enum Shifts;  
Shifts: idle;           % Variable representing the idle constant.
```

and the data file:

```
enum Shifts = { idle_const, day, night };  
idle = idle_const;      % Assignment to the variable.
```

Although the constant `idle_const` cannot be mentioned in the model, the variable `idle` can be.

All enums must be defined at instance-time.

Enum items can be annotated. Section 9 has more details on annotations.

Each case name can be coerced automatically to the integer corresponding to its index in the type.

```
int: oz = Australia; % oz = 1
```

For each enumerated type  $T$ , the following functions exist:

```
% Return next greater enum value of x in enum type X
function T: enum_next(set of T: X, T: x);
function var T: enum_next(set of T: X, var T: x);

% Return next smaller enum value of x in enum type X
function T: enum_prev(set of T: X, T: x);
function var T: enum_prev(set of T: X, var T: x);

% Convert x to enum type X
function T: to_enum(set of T: X, int: x);
function var T: to_enum(set of T: X, var int: x);
```

## 8.4. Assignment Items

Assignments have this syntax:

$$\langle \text{assign-item} \rangle ::= \langle \text{ident} \rangle \triangleq \langle \text{expr} \rangle$$

For example:

```
A = 10;
```

## 8.5. Constraint Items

Constraint items form the heart of a model. Any solutions found for a model will satisfy all of its constraints.

Constraint items have this syntax:

$$\langle \text{constraint-item} \rangle ::= \underline{\text{constraint}} \langle \text{expr} \rangle$$

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.

## 8.6. Solve Items

Every model must have exactly one solve item. Solve items have the following syntax:

$$\begin{aligned} \langle \text{solve-item} \rangle ::= & \underline{\text{solve}} \langle \text{annotations} \rangle \underline{\text{satisfy}} \\ & | \underline{\text{solve}} \langle \text{annotations} \rangle \underline{\text{minimize}} \langle \text{expr} \rangle \\ & | \underline{\text{solve}} \langle \text{annotations} \rangle \underline{\text{maximize}} \langle \text{expr} \rangle \end{aligned}$$

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. In the latter case 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. Section 9 has more details on annotations.

## 8.7. Output Items

Output items are used to present the solutions of a model instance. They have the following syntax:

$\langle \text{output-item} \rangle ::= \underline{\text{output}} \langle \text{expr} \rangle$

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` (Appendix A). 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.

## 8.8. Annotation Items

Annotation items are used to augment the `ann` type. They have the following syntax:

$\langle \text{annotation-item} \rangle ::= \underline{\text{annotation}} \langle \text{ident} \rangle \langle \text{params} \rangle$

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 Section 9.

## 8.9. User-defined Operations

MiniZinc models can contain user-defined operations. They have this syntax:

$\langle \text{predicate-item} \rangle ::= \underline{\text{predicate}} \langle \text{operation-item-tail} \rangle$   
 $\langle \text{test-item} \rangle ::= \underline{\text{test}} \langle \text{operation-item-tail} \rangle$   
 $\langle \text{function-item} \rangle ::= \underline{\text{function}} \langle \text{ti-expr} \rangle \dot{=} \langle \text{operation-item-tail} \rangle$   
 $\langle \text{operation-item-tail} \rangle ::= \langle \text{ident} \rangle \langle \text{params} \rangle \langle \text{annotations} \rangle [ \dot{=} \langle \text{expr} \rangle ]$   
 $\langle \text{params} \rangle ::= [ \langle \text{ti-expr-and-id} \rangle \dot{,} \dots \dot{,} ]$

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. Section 9 has more details on annotations.

### 8.9.1. 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.

### 8.9.2. 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 declared and/or defined more than once in a model. For example:

```
predicate p(1..5: x);
predicate p(1..5: x) = true;           % error: repeated declaration
```

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  $(S_1, \dots, S_n)$  and  $(T_1, \dots, T_n)$  then there must be another overloaded version with input type-inst  $(R_1, \dots, R_n)$  where each  $R_i$  is the greatest lower bound (*glb*) of  $S_i$  and  $T_i$ .

Also, all overlapping overloadings of an operation must be monotonic. That is, if there are overloaded versions with input type-insts  $(S_1, \dots, S_n)$  and  $(T_1, \dots, T_n)$  and output type-inst  $S$  and  $T$ , respectively, then  $S_i \preceq T_i$  for all  $i$ , implies  $S \preceq T$ . At call sites, the matching overloading that is lowest on the type-inst lattice is always used.

For `g` above, the type-inst intersection (or *glb*) of  $(\text{float}, \text{int})$  and  $(\text{float}, \text{int})$  is  $(\text{int}, \text{int})$ . Thus, the overloaded versions are not closed under intersection and the user needs to provide another overloading for `g` with input type-inst  $(\text{int}, \text{int})$ . 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.



### 8.9.3. 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 Section 7.3.14 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);
```

## 9. Annotations

Annotations—values of the `ann` type—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), expressions, type-inst synonyms, enum items, solve items and on user defined operations. They have the following syntax:

```
 $\langle annotations \rangle ::= ( \text{::} \langle annotation \rangle )^*$   
 $\langle annotation \rangle ::= \langle expr\text{-}atom\text{-}head \rangle \langle expr\text{-}atom\text{-}tail \rangle$ 
```

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. Unlike user-defined predicates and functions, annotations cannot be overloaded. ***Rationale.*** *There is no particular strong reason for this, it just seemed to make things simpler.*

Annotation signatures can contain type-inst variables.

The order and nesting of annotations do not matter. For the expression case it can be helpful to view the annotation connector `'::'` as an overloaded operator:

```
ann: '::'(any $T: e, ann: a);           % associative
ann: '::'(ann: a, ann: b);             % associative + commutative
```

Both operators are associative, the second is commutative. This means that the following expressions are all equivalent:

```
e :: a :: b
e :: b :: a
(e :: a) :: b
(e :: b) :: a
e :: (a :: b)
e :: (b :: a)
```

This property also applies to annotations on solve items and variable declaration items. ***Rationale.*** *This property make things simple, as it allows all nested combinations of annotations to be treated as if they are flat, thus avoiding the need to determine what is the meaning of an annotated annotation. It also makes the MiniZinc abstract syntax tree simpler by avoiding the need to represent nesting.*

## 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.*

## 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 assigned value does not statically match the variable's constraint at run-time it fails, and 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 tuples, 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.*

## 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.

### 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.

## A. 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.

### A.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)
```

### A.2. Arithmetic Operations

Addition. Other numeric operations are similar: subtraction (`-`), and multiplication (`*`).

```
int:  '+'(    int,    int)
var int: '+'(var int, var int)
float: '+'(    float,  float)
var float: '+'(var float, var float)
```

Unary minus. Unary plus (`+`) is similar.

```
int:  '-'(    int)
var int: '-'(var int)
float: '-'(    float)
var float: '-'(var float)
```

Integer and floating-point division and modulo.

```
int:  'div'(    int,    int)
var int: 'div'(var int, var int)
int:  'mod'(    int,    int)
var int: 'mod'(var int, var int)
float: '/'(    float,  float)
var float: '/'(var float, var float)
```

The result of the modulo operation, if non-zero, always has the same sign as its first operand. The integer division and modulo operations are connected by the following identity:

$$x = (x \text{ div } y) * y + (x \text{ mod } y)$$

Some illustrative examples:

```
7 div 4 = 1      7 mod 4 = 3
-7 div 4 = -1    -7 mod 4 = -3
7 div -4 = -1    7 mod -4 = 3
-7 div -4 = 1    -7 mod -4 = -3
```

Sum multiple numbers. Product (`product`) is similar. Note that the sum of an empty array is 0, and the product of an empty array is 1.

```
int:  sum(array[$T] of    int )
var int: sum(array[$T] of var int )
float: sum(array[$T] of    float)
var float: sum(array[$T] of var float)
```

Minimum of two values; maximum (`max`) is similar.

```
any $T:  min(any $T,    any $T )
```

Minimum of an array of values; maximum (`max`) is similar. Aborts if the array is empty.

```
any $U:  min(array[$T] of any $U)
```

Minimum of a fixed set; maximum (`max`) is similar. Aborts if the set is empty.

```
$T:    min(set of $T)
```

Absolute value of a number.

```
int:    abs(    int)
var int: abs(var int)
float:  abs(    float)
var float: abs(var float)
```

Square root of a float. Aborts if argument is negative.

```
float: sqrt(    float)
var float: sqrt(var float)
```

Power operator. E.g. `pow(2, 5)` gives 32.

```
int: pow(int,    int)
float: pow(float, float)
```

Natural exponent.

```
float: exp(float)
var float: exp(var float)
```

Natural logarithm. Logarithm to base 10 (`log10`) and logarithm to base 2 (`log2`) are similar.

```
float: ln(float)
var float: ln(var float)
```

General logarithm; the first argument is the base.

```
float: log(float, float)
```

Sine. Cosine (`cos`), tangent (`tan`), inverse sine (`asin`), inverse cosine (`acos`), inverse tangent (`atan`), hyperbolic sine (`sinh`), hyperbolic cosine (`cosh`), hyperbolic tangent (`tanh`), inverse hyperbolic sine (`asinh`), inverse hyperbolic cosine (`acosh`) and inverse hyperbolic tangent (`atanh`) are similar.

```
float: sin(float)
var float: sin(var float)
```

### A.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)
```

### A.4. Set Operations

Set membership.

```
bool: 'in'(    $T,    set of $T )
var bool: 'in'(var int, var set of int)
```

Non-strict subset. Non-strict superset (`superset`) is similar.

```
bool: 'subset'(    set of $T ,    set of $T )
var bool: 'subset'(var set of int, var set of int)
```

Set union. Other set operations are similar: intersection (`intersect`), difference (`diff`), symmetric difference (`symdiff`).

```

    set of $T: 'union'(    set of $T,    set of $T )
var set of int: 'union'(var set of int, var set of int )

```

Set range. If the first argument is larger than the second (e.g. 1..0), it returns the empty set.

```

set of int: '..'(int, int)

```

Cardinality of a set.

```

    int: card(    set of $T)
var int: card(var set of int)

```

Union of an array of sets. Intersection of multiple sets (`array_intersect`) is similar.

```

    set of $U:    array_union(array[$T] of    set of $U)
var set of int:  array_union(array[$T] of var set of int)

```

## A.5. Array Operations

Length of an array.

```

int: length(array[$T] of any $U)

```

List concatenation. Returns the list (integer-indexed array) containing all elements of the first argument followed by all elements of the second argument, with elements occurring in the same order as in the arguments. The resulting indices are in the range 1..*n*, where *n* is the sum of the lengths of the arguments. ***Rationale.*** *This allows list-like arrays to be concatenated naturally and avoids problems with overlapping indices. The resulting indices are consistent with those of implicitly indexed array literals.* Note that '++' also performs string concatenation.

```

array[int] of any $T: '++'(array[int] of any $T, array[int] of any $T)

```

Index sets of arrays. If the argument is a literal, returns 1..*n* where *n* is the (sub-)array length. Otherwise, returns the declared or inferred index set. This list is only partial, it extends in the obvious way, for arrays of higher dimensions.

```

set of $T:  index_set    (array[$T]    of any $V)
set of $T:  index_set_1of2(array[$T, $U] of any $V)
set of $U:  index_set_2of2(array[$T, $U] of any $V)
...

```

Replace the indices of the array given by the last argument with the Cartesian product of the sets given by the previous arguments. Similar versions exist for arrays up to 6 dimensions.

```

array[$T1] of any $V: array1d(set of $T1, array[$U] of any $V)
array[$T1,$T2] of any $V:
    array2d(set of $T1, set of $T2, array[$U] of any $V)
array[$T1,$T2,$T3] of any $V:
    array3d(set of $T1, set of $T2, set of $T3, array[$U] of any $V)

```

## A.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)

```

## A.7. String Operations

To-string conversion. Converts any value to a string for output purposes. The exact form of the resulting string is implementation-dependent.

```
string: show(any $T)
```

Formatted to-string conversion for integers. Converts the integer given by the second argument into a string right justified by the number of characters given by the first argument, or left justified if that argument is negative. If the second argument is not fixed, the form of the string is implementation-dependent.

```
string: show_int(int, var int);
```

Formatted to-string conversion for floats. Converts the float given by the third argument into a string right justified by the number of characters given by the first argument, or left justified if that argument is negative. The number of digits to appear after the decimal point is given by the second argument. It is a run-time error for the second argument to be negative. If the third argument is not fixed, the form of the string is implementation-dependent.

```
string: show_float(int, int, var float)
```

String concatenation. Note that '++' also performs array concatenation.

```
string: '++'(string, string)
```

Concatenate an array of strings. Equivalent to folding '++' over the array, but may be implemented more efficiently.

```
string: concat(array[$T] of string)
```

Concatenate an array of strings, putting a separator between adjacent strings. Returns the empty string if the array is empty.

```
string: join(string, array[$T] of string)
```

## A.8. Bound and Domain Operations

The bound operations lb/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.

## A.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);
```

## A.10. Other Operations

Check a Boolean expression is true, and abort if not, printing the second argument as the error message. The first one returns the third argument, and is particularly useful for sanity-checking arguments to predicates and functions; importantly, its third argument is lazy, i.e. it is only evaluated if the condition succeeds. The second one returns `true` and is useful for global sanity-checks (e.g. of instance data) in constraint items.

```
any $T: assert(bool, string, any $T)  
par bool: assert(bool, string)
```

Abort evaluation, printing the given string.

```
any $T: abort(string)
```

Return true. As a side-effect, an implementation may print the first argument.

```
bool: trace(string)
```

Return the second argument. As a side-effect, an implementation may print the first argument.

```
any $T: trace(string, any $T)
```

Check if the argument's value is fixed at this point in evaluation. If not, abort; if so, return its value. This is most useful in output items when decision variables should be fixed—it allows them to be used in places where a fixed value is needed, such as if-then-else conditions.

```
$T: fix(any $T)
```

As above, but return `false` if the argument's value is not fixed.

```
par bool: is_fixed(any $T)
```



## B. MiniZinc Grammar

Section 2 describes the notation used in the following Zinc grammar.

### B.1. Items

$\langle model \rangle ::= [ \langle item \rangle \text{ ; } \dots ]$

$\langle item \rangle ::= \langle include-item \rangle$   
           $| \langle var-decl-item \rangle$   
           $| \langle assign-item \rangle$   
           $| \langle constraint-item \rangle$   
           $| \langle solve-item \rangle$   
           $| \langle output-item \rangle$   
           $| \langle predicate-item \rangle$   
           $| \langle test-item \rangle$   
           $| \langle function-item \rangle$   
           $| \langle annotation-item \rangle$

$\langle type-inst-syn-item \rangle ::= \text{type} \langle ident \rangle \langle annotations \rangle \equiv \langle ti-expr \rangle$

$\langle ti-expr-and-id \rangle ::= \langle ti-expr \rangle \text{ ; } \langle ident \rangle$

$\langle include-item \rangle ::= \text{include} \langle string-literal \rangle$

$\langle var-decl-item \rangle ::= \langle ti-expr-and-id \rangle \langle annotations \rangle [ \equiv \langle expr \rangle ]$

$\langle assign-item \rangle ::= \langle ident \rangle \equiv \langle expr \rangle$

$\langle constraint-item \rangle ::= \text{constraint} \langle expr \rangle$

$\langle solve-item \rangle ::= \text{solve} \langle annotations \rangle \text{ satisfy}$   
                   $| \text{solve} \langle annotations \rangle \text{ minimize } \langle expr \rangle$   
                   $| \text{solve} \langle annotations \rangle \text{ maximize } \langle expr \rangle$

$\langle output-item \rangle ::= \text{output} \langle expr \rangle$

$\langle annotation-item \rangle ::= \text{annotation} \langle ident \rangle \langle params \rangle$

$\langle predicate-item \rangle ::= \text{predicate} \langle operation-item-tail \rangle$

$\langle test-item \rangle ::= \text{test} \langle operation-item-tail \rangle$

$\langle function-item \rangle ::= \text{function} \langle ti-expr \rangle \text{ ; } \langle operation-item-tail \rangle$

$\langle operation-item-tail \rangle ::= \langle ident \rangle \langle params \rangle \langle annotations \rangle [ \equiv \langle expr \rangle ]$

$\langle params \rangle ::= [ \text{ ( } \langle ti-expr-and-id \rangle \text{ , } \dots \text{ ) } ]$

### B.2. Type-Inst Expressions

$\langle ti-expr \rangle ::= \langle base-ti-expr \rangle$

$\langle base-ti-expr \rangle ::= \langle var-par \rangle \langle base-ti-expr-tail \rangle$

$\langle var-par \rangle ::= \text{var} \text{ | } \text{par} \text{ | } \epsilon$

$\langle base-ti-expr-tail \rangle ::= \langle ident \rangle$

$| \text{bool}$   
           $| \text{int}$   
           $| \text{float}$   
           $| \text{string}$   
           $| \langle set-ti-expr-tail \rangle$   
           $| \langle array-ti-expr-tail \rangle$   
           $| \text{ann}$   
           $| \text{opt} \langle base-ti-expr-tail \rangle$   
           $| \{ \langle expr \rangle \text{ , } \dots \}$   
           $| \langle num-expr \rangle \text{ .. } \langle num-expr \rangle$

$\langle set-ti-expr-tail \rangle ::= \text{set of} \langle base-type \rangle$

$\langle \text{array-ti-expr-tail} \rangle ::= \text{array } [ \langle \text{ti-expr} \rangle , \dots ] \text{ of } \langle \text{ti-expr} \rangle$   
 $| \text{list of } \langle \text{ti-expr} \rangle$

$\langle \text{ti-variable-expr-tail} \rangle ::= \$[A-Za-z][A-Za-z0-9_]*$

$\langle \text{op-ti-expr-tail} \rangle ::= \text{op } ( \langle \text{ti-expr} \rangle : ( \langle \text{ti-expr} \rangle , \dots ) )$

### B.3. Expressions

$\langle \text{expr} \rangle ::= \langle \text{expr-atom} \rangle \langle \text{expr-binop-tail} \rangle$

$\langle \text{expr-atom} \rangle ::= \langle \text{expr-atom-head} \rangle \langle \text{expr-atom-tail} \rangle \langle \text{annotations} \rangle$

$\langle \text{expr-binop-tail} \rangle ::= [ \langle \text{bin-op} \rangle \langle \text{expr} \rangle ]$

$\langle \text{expr-atom-head} \rangle ::= \langle \text{builtin-un-op} \rangle \langle \text{expr-atom} \rangle$

$| ( \langle \text{expr} \rangle )$

$| \langle \text{ident-or-quoted-op} \rangle$

$| =$

$| \langle \text{bool-literal} \rangle$

$| \langle \text{int-literal} \rangle$

$| \langle \text{float-literal} \rangle$

$| \langle \text{string-literal} \rangle$

$| \langle \text{set-literal} \rangle$

$| \langle \text{set-comp} \rangle$

$| \langle \text{simple-array-literal} \rangle$

$| \langle \text{simple-array-literal-2d} \rangle$

$| \langle \text{indexed-array-literal} \rangle$

$| \langle \text{simple-array-comp} \rangle$

$| \langle \text{indexed-array-comp} \rangle$

$| \langle \text{ann-literal} \rangle$

$| \langle \text{if-then-else-expr} \rangle$

$| \langle \text{let-expr} \rangle$

$| \langle \text{call-expr} \rangle$

$| \langle \text{gen-call-expr} \rangle$

$\langle \text{expr-atom-tail} \rangle ::= \epsilon$

$| \langle \text{array-access-tail} \rangle \langle \text{expr-atom-tail} \rangle$

$\langle \text{num-expr} \rangle ::= \langle \text{num-expr-atom} \rangle \langle \text{num-expr-binop-tail} \rangle$

$\langle \text{num-expr-atom} \rangle ::= \langle \text{num-expr-atom-head} \rangle \langle \text{expr-atom-tail} \rangle \langle \text{annotations} \rangle$

$\langle \text{num-expr-binop-tail} \rangle ::= [ \langle \text{num-bin-op} \rangle \langle \text{num-expr} \rangle ]$

$\langle \text{num-expr-atom-head} \rangle ::= \langle \text{builtin-num-un-op} \rangle \langle \text{num-expr-atom} \rangle$

$| ( \langle \text{num-expr} \rangle )$

$| \langle \text{ident-or-quoted-op} \rangle$

$| \langle \text{int-literal} \rangle$

$| \langle \text{float-literal} \rangle$

$| \langle \text{if-then-else-expr} \rangle$

$| \langle \text{case-expr} \rangle$

$| \langle \text{let-expr} \rangle$

$| \langle \text{call-expr} \rangle$

$| \langle \text{gen-call-expr} \rangle$

$\langle \text{builtin-op} \rangle ::= \langle \text{builtin-bin-op} \rangle$

$| \langle \text{builtin-un-op} \rangle$

$\langle \text{bin-op} \rangle ::= \langle \text{builtin-bin-op} \rangle$

$| \langle \text{ident} \rangle \langle \text{ident} \rangle$

$\langle \text{builtin-bin-op} \rangle ::= \leq \mid \geq \mid \leq \mid \geq \mid \text{xor} \mid \wedge$

$\mid \leq \mid \geq \mid \leq \mid \geq \mid == \mid \neq \mid !=$

$\mid \text{in} \mid \text{subset} \mid \text{superset} \mid \text{union} \mid \text{diff} \mid \text{syndiff}$

$\mid \text{..} \mid \text{intersect} \mid ++ \mid \langle \text{builtin-num-bin-op} \rangle$

$\langle \text{builtin-un-op} \rangle ::= \text{not} \mid \langle \text{builtin-num-un-op} \rangle$

$\langle \text{num-bin-op} \rangle ::= \langle \text{builtin-num-bin-op} \rangle$

$| \langle \text{ident} \rangle \langle \text{ident} \rangle$

$\langle \text{builtin-num-bin-op} \rangle ::= + \mid - \mid * \mid / \mid \text{div} \mid \text{mod}$

$\langle \text{builtin-num-un-op} \rangle ::= \pm \mid -$

$\langle \text{bool-literal} \rangle ::= \underline{\text{false}} \mid \underline{\text{true}}$

$\langle \text{int-literal} \rangle ::= [0-9]^+ \mid \text{0x}[0-9\text{A-Fa-f}]^+ \mid \text{0o}[0-7]^+$

$\langle \text{float-literal} \rangle ::= [0-9]^+.[0-9]^+ \mid [0-9]^+.[0-9]^+[\text{Ee}][+]?[0-9]^+ \mid [0-9]^+[\text{Ee}][+]?[0-9]^+$

$\langle \text{string-contents} \rangle ::= ([^"\backslash n\_]\mid \_\backslash[^"\backslash n\_])^*$

$\langle \text{string-literal} \rangle ::= "\langle \text{string-contents} \rangle" \mid "\langle \text{string-contents} \rangle\_ \langle \text{string-interpolate-tail} \rangle$

$\langle \text{string-interpolate-tail} \rangle ::= \langle \text{expr} \rangle \_ \langle \text{string-contents} \rangle" \mid \langle \text{expr} \rangle \_ \langle \text{string-contents} \rangle\_ \langle \text{string-interpolate-tail} \rangle$

$\langle \text{set-literal} \rangle ::= \{ \_ [ \langle \text{expr} \rangle \_ \dots ] \}$

$\langle \text{set-comp} \rangle ::= \{ \_ \langle \text{expr} \rangle \_ \langle \text{comp-tail} \rangle \}$   
 $\langle \text{comp-tail} \rangle ::= \langle \text{generator} \rangle \_ \dots [ \underline{\text{where}} \langle \text{expr} \rangle ]$   
 $\langle \text{generator} \rangle ::= \langle \text{ident} \rangle \_ \dots \underline{\text{in}} \langle \text{expr} \rangle$

$\langle \text{array-literal} \rangle ::= [ [ \langle \text{expr} \rangle \_ \dots ] ]$

$\langle \text{array-literal-2d} \rangle ::= [ [ [ ( \langle \text{expr} \rangle \_ \dots ) \_ \dots ] ] ]$

$\langle \text{array-comp} \rangle ::= [ \_ \langle \text{expr} \rangle \_ \langle \text{comp-tail} \rangle ]$

$\langle \text{array-access-tail} \rangle ::= [ \_ \langle \text{expr} \rangle \_ \dots ]$

$\langle \text{ann-literal} \rangle ::= \langle \text{ident} \rangle [ ( \_ \langle \text{expr} \rangle \_ \dots ) ]$

$\langle \text{if-then-else-expr} \rangle ::= \underline{\text{if}} \langle \text{expr} \rangle \underline{\text{then}} \langle \text{expr} \rangle ( \underline{\text{elseif}} \langle \text{expr} \rangle \underline{\text{then}} \langle \text{expr} \rangle )^* \underline{\text{else}} \langle \text{expr} \rangle \underline{\text{endif}}$

$\langle \text{call-expr} \rangle ::= \langle \text{ident-or-quoted-op} \rangle [ ( \_ \langle \text{expr} \rangle \_ \dots ) ]$

$\langle \text{let-expr} \rangle ::= \underline{\text{let}} \{ \_ \langle \text{let-item} \rangle \_ \dots \} \underline{\text{in}} \langle \text{expr} \rangle$   
 $\langle \text{let-item} \rangle ::= \langle \text{var-decl-item} \rangle \mid \langle \text{constraint-item} \rangle$

$\langle \text{gen-call-expr} \rangle ::= \langle \text{ident-or-quoted-op} \rangle ( \_ \langle \text{comp-tail} \rangle \_ ( \_ \langle \text{expr} \rangle \_ )$

## B.4. Miscellaneous Elements

$\langle \text{ident} \rangle ::= [\text{A-Za-z}][\text{A-Za-z0-9\_}]^* \quad \% \text{ excluding keywords}$   
 $\mid '\_\backslash[^'\backslash xa\backslash xd\backslash x0]^*'$

$\langle \text{ident-or-quoted-op} \rangle ::= \langle \text{ident} \rangle \mid '\_ \langle \text{builtin-op} \rangle \_'$

$\langle \text{annotations} \rangle ::= ( \_ :: \langle \text{annotation} \rangle )^*$   
 $\langle \text{annotation} \rangle ::= \langle \text{expr-atom-head} \rangle \langle \text{expr-atom-tail} \rangle$

## C. Content-types

### C.1. ‘application/x-zinc-output’

The content-type ‘application/x-zinc-output’ defines a text output format for Zinc. The format extends the abstract syntax and semantics given in Section 3.2, and is discussed in detail in Section 3.3.

The full syntax is as follows:

$$\langle output \rangle ::= \langle no-solutions \rangle [ \langle warnings \rangle ] \langle free-text \rangle \\ | ( \langle solution \rangle )^* [ \langle complete \rangle ] [ \langle warnings \rangle ] \langle free-text \rangle$$
$$\langle solution \rangle ::= \langle solution-text \rangle [ \backslash n ] \text{-----} \backslash n$$
$$\langle no-solutions \rangle ::= \text{====UNSATISFIABLE====} \backslash n$$
$$\langle complete \rangle ::= \text{=====} \backslash n$$
$$\langle warnings \rangle ::= ( \langle message \rangle )^+$$
$$\langle message \rangle ::= ( \langle line \rangle )^+$$
$$\langle line \rangle ::= \% [^\backslash n]^* \backslash n$$

The solution text for each solution must be as described in Section 8.7. A newline must be appended if the solution text does not end with a newline.

## D. JSON support

MiniZinc can support reading input parameters and providing output formatted as JSON objects. A JSON input file needs to have the following structure:

- Consist of a single top-level object
- The members of the object (the key-value pairs) represent model parameters
- Each member key must be a valid MiniZinc identifier (and it supplies the value for the corresponding parameter of the model)
- Each member value can be one of the following:
  - A string (assigned to a MiniZinc string parameter)
  - A number (assigned to a MiniZinc int or float parameter)
  - The values `true` or `false` (assigned to a MiniZinc bool parameter)
  - An array of values. Arrays of arrays are supported only if all inner arrays are of the same length, so that they can be mapped to multi-dimensional MiniZinc arrays.
  - A set of values encoded as an object with a single member with key `"set"` and a list of values (the elements of the set).

This is an example of a JSON parameter file using all of the above features:

```
{
  "n" : 3,
  "distances" : [ [1,2,3],
                   [4,5,6]],
  "patterns" : [ {"set" : [1,3,5]}, {"set" : [2,4,6]} ]
}
```

The first parameter declares a simple integer `n`. The `distances` parameter is a two-dimensional array; note that all inner arrays must be of the same size in order to map to a (rectangular) MiniZinc two-dimensional array. The third parameter is an array of sets of integers.