# Specification of FlatZinc

Version 1.0

Nicholas Nethercote

# Contents

# 1 Introduction

This document is a specification of FlatZinc. FlatZinc is a low-level solver input language. It is designed to specify problems at the level of an interface to a CP solver that supports finite domain and linear programming constraints. It is (mostly) a subset of MiniZinc and is intended to be a target language for MiniZinc.

FlatZinc includes features that are appropriate for a wide range of solvers; for this reason, it is unlikely that any single FlatZinc implementation will implement all the features. For example, an implementation for a finite domain solver will probably not be able to handle models that use floats.

For more details on the intended use, please see *MiniZinc: Towards a Standard CP Modelling Language*, by Nethercote, Stuckey, Becket, Brand, Duck and Tack.

This document has the following structure. Section 2 provides a high-level overview of FlatZinc models. Section 3 covers syntax basics. Section 4 covers high-level structure: items, multi-file models, namespaces, and scopes. Section 5 introduces types and type-insts. Section 6 covers expressions. Section 7 describes the top-level items in detail. Section 8 describes annotations. Section 9 describes output of solutions. Appendix A describes the language built-ins. Appendix B describes the standard language annotations. Appendix C gives the grammar.

# 2 Overview of a Model

## 2.1 Specifying a Problem

Conceptually, a FlatZinc 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*).

However, in FlatZinc a model and its data are always "hard-wired" together. Section 4.2 specifies how the model and data can be structured within files in a model instance.

## 2.2 Evaluation of a Model Instance

### 2.2.1 Evaluation Phases

A FlatZinc 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 instance-time phases are static, the run-time phase is dynamic.

### 2.2.2 Evaluation Outcomes

There are four possible evaluation outcomes.

1. Static error: the model instance does not compile due to a problem with the model and/or data, detected at model-time or instance-time. This could be caused by a syntax error, a type-inst error, the use of an unsupported feature or operation, etc.

2. Run-time error: the evaluation fails to complete due to a problem with the model and/or data, detected at run-time. This could be caused by an assertion failure, division by zero, an array bounds error, etc. Alternatively, it could be due to an implementation shortcoming, such as a time-out due to excessive evaluation time, failure to determine if any solutions are possible, an overflow on an integer operation, etc.

3. Failure: no solutions are returned due to unsatisfiable constraints.

4. Success: one or more solutions are returned.

Static errors must be detected prior to run-time. The remaining outcomes—which can only occur for instances without static errors—are determined at run-time. However, an implementation is free to determine them earlier if it safely can. For example, an implementation may be able to determine that unsatisfiable constraints exist prior to run-time, and the resulting messages given to the user may be more helpful than if the unsatisfiability is detected at run-time.

An implementation must produce output in all outcomes. The form of the output in the error cases is implementation-dependent. The form of the output in the failure and success cases is described in Section 9.

An implementation may produce warnings during all evaluation phases.

# 3 Syntax Overview

## 3.1 Character Set

FlatZinc currently allows only ASCII characters. In the future we hope to support Unicode.

FlatZinc is case sensitive. There are no places where upper-case or lower-case letters must be used.

FlatZinc has no layout restrictions, i.e. any single piece of whitespace (containing spaces, tabs and newlines) is equivalent to any other.

## 3.2 Comments

A `%` indicates that the rest of the line is a comment. FlatZinc has no begin/end comment symbols (such as C's `/*` and `*/` comments).

## 3.3 Syntax Notation

The basics of the EBNF used for the FlatZinc grammar are as follows.

- Non-terminals are written between angle brackets, e.g. ⟨*item*⟩.
- Terminals are written in fixed-width font and underlined, e.g. `constraint`.
- 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*⟩ )*.
- Non-empty lists are written with an item, a separator terminal, and "...". For example, this:

  ⟨*expr*⟩ `,` ...

  is short for this:

⟨*expr*⟩ ( , ⟨*expr*⟩ )*

- Regular expressions, written in fixed-width font, are used in some productions, e.g. `[-+]?[0-9]+`.

FlatZinc's grammar is presented piece-by-piece throughout this document. It is also available as a whole in Appendix C.

### 3.4 Identifiers

The syntax of identifiers is:

⟨*ident*⟩ ::= `[A-Za-z][A-Za-z0-9_]*` % excluding keywords

For example:

```
my_name_2
MyName2
```

A number of keywords are reserved and cannot be used as identifiers. The keywords are: `annotation`, `any`, `array`, `bool`, `case`, `constraint`, `diff`, `div`, `else`, `elseif`, `endif`, `enum`, `false`, `float`, `function`, `if`, `in`, `include`, `int`, `intersect`, `let`, `list`, `maximize`, `minimize`, `mod`, `not`, `of`, `satisfy`, `subset`, `superset`, `output`, `par`, `predicate`, `record`, `set`, `solve`, `string`, `symdiff`, `test`, `then`, `true`, `tuple`, `union`, `type`, `var`, `where`, `xor`. Note that some of these keywords are not used in FlatZinc. They are reserved because they are keywords in Zinc and MiniZinc.

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

## 4 High-level Model Structure

### 4.1 Items

A FlatZinc model consists of multiple *items*, of various kinds, in a particular order:

⟨*model*⟩ ::= ( ⟨*pred-decl-item*⟩ ; )* ( ⟨*var-decl-item*⟩ ; )* ( ⟨*constraint-item*⟩ ; )* ⟨*solve-item*⟩ ;
[ ⟨*output-item*⟩ ; ]

Predicate declaration items are used to declare non-standard constraints (Section 7.1).

Identifiers introduced by variable declarations must be declared before use.

Variable declaration items introduce new decision variables and possibly bind them to a value (Section 7.2).

Constraint items describe model constraints (Section 7.3).

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 7.4).

### 4.2 Model Instance Files

In FlatZinc there is no separation of model and data. Multi-file models are not allowed, and both the model and the data must be in a single file.

### 4.3 Namespaces

All names level belong to a single namespace. It includes all global variable names and all built-in operation names.

FlatZinc supports no overloading.

### 4.4 Scopes

There are no local scopes in FlatZinc.

## 5 Types and Type-insts

FlatZinc provides four scalar types: Booleans, integers, floats, and strings; and two compound types: sets and one-dimensional integer-indexed arrays.

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, and whether it can be involved in automatic coercions.

### 5.1 Properties of Types

All types are monotypes. All types are first-order.

### 5.2 Instantiations

When a FlatZinc 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).

There are also intermediate instantiations for some arrays—they can have a fixed size but may contain unfixed elements.

Decision variables can have the following types: Booleans, integers, floats, and sets; arrays can contain decision variables.

### 5.3 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. FlatZinc allow coercions only within types, i.e. only the inst can change.

## 5.4 Type-inst Expressions Overview

This section partly describes how to write type-insts in FlatZinc 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 appear in variable declaration items (Section 7.2), which have this syntax:

⟨*var-decl-item*⟩ ::= `var` ⟨*non-array-ti-expr-tail*⟩ `:` ⟨*ident-anns*⟩ [ `=` ⟨*non-array-flat-expr*⟩ ]
| ⟨*non-array-ti-expr-tail*⟩ `:` ⟨*ident-anns*⟩ `=` ⟨*non-array-flat-expr*⟩
| `array` `[` `1` `..` ⟨*int-literal*⟩ `]` `of` ⟨*array-decl-tail*⟩
⟨*array-decl-tail*⟩ ::= ⟨*non-array-ti-expr-tail*⟩ `:` ⟨*ident-anns*⟩ `=` ⟨*array-literal*⟩
| `var` ⟨*non-array-ti-expr-tail*⟩ `:` ⟨*ident-anns*⟩ [ `=` ⟨*array-literal*⟩ ]

⟨*non-array-ti-expr-tail*⟩ ::= ⟨*scalar-ti-expr-tail*⟩
| ⟨*set-ti-expr-tail*⟩

The `var` keyword (or lack thereof) determines the instantiation. A type-inst is fixed if it does not contain `var`.

One powerful feature of FlatZinc 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. 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. There are three kinds of constrained type-insts.

- Integer ranges: e.g. `1..3`.
- Float ranges: e.g. `1.0..3.0`.
- Set literals: e.g. {`1, 3, 5`}. The elements must be integer literals.

## 5.5 Built-in Scalar Types and Type-insts

### 5.5.1 Booleans

*Overview.* Booleans represent truthhood or falsity.

*Allowed Insts.* Booleans can be fixed or unfixed.

*Syntax.* Boolean type-inst expression tails have this syntax:

⟨*bool-ti-expr-tail*⟩ ::= `bool`

Booleans type-inst expressions are written `bool` or `var bool`.

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

### 5.5.2 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.* Integer type-inst expression tails have this syntax:

⟨*int-ti-expr-tail*⟩ ::= `int`
| ⟨*int-literal*⟩ `..` ⟨*int-literal*⟩
| `{` ⟨*int-literal*⟩ `,` ... `}`

Some example integer type-inst expressions:

```
int
var int
1..10
var {1,3,5}
```

*Coercions.* `int` $\xrightarrow{c}$ `var int`.

### 5.5.3 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 `NaN`s, if using IEEE754 floats).

*Allowed Insts.* Floats can be fixed or unfixed.

*Syntax.* Float type-inst expression tails have this syntax:

⟨*float-ti-expr-tail*⟩ ::= `float`
| ⟨*float-literal*⟩ `..` ⟨*float-literal*⟩

Some example float type-inst expressions:

```
float
var 1.2 .. 5.6
```

*Coercions.* `float` $\xrightarrow{c}$ `var float`.

### 5.5.4 Strings

*Overview.* Strings are primitive, i.e. they are not lists of characters. String literals are used in annotations. String variables are not allowed.

*Allowed Insts.* Strings must be fixed.

*Syntax.* Fixed strings are written `string`.

*Coercions.* None automatic.

## 5.6 Built-in Compound Types and Type-insts

### 5.6.1 Sets

*Overview.* A set is a collection with no duplicates. FlatZinc sets can only contain scalars.

*Allowed Insts.* The type-inst of a set's elements must be fixed; sets cannot contain decision variables because solvers are not powerful enough to handle them.

Sets of integers may be fixed or unfixed; in unfixed sets the element type must be a *finite* integer type, i.e. an integer type defined via a range expression or a set expression.

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

⟨*set-ti-expr-tail*⟩ ::= `set of` ⟨*scalar-ti-expr-tail*⟩

Some example set type-inst expressions:

```
set of bool
var set of 1..3
```

Note that `var set of int` is not allowed as `int` is not a finite type.

*Coercions.* `set of int` $\xrightarrow{c}$ `var set of int`.

### 5.6.2 Arrays

*Overview.* FlatZinc arrays are maps from integers to values. They can only be one-dimensional and are one-indexed. The index set in a array variable declaration must be a contiguous integer range (e.g. `1..3`). The elements can only be Booleans, integers, floats or sets of integers (all fixed or unfixed).

The initialisation of an array must be part of the variable declaration.

Arrays can be accessed. See Section 6 for details.

*Allowed Insts.* An array's size must be fixed at instance-time. Its indices must also have fixed type-insts. Its elements may be fixed or unfixed.

*Syntax.* An array is written `array[1..<n>] of <TI>`, where `<n>` is an integer literal giving the length of the array, and `<TI>` is the type-inst of the array elements.

*Coercions.* `array[1..<n>] of TI` $\xrightarrow{c}$ `array[1..<n>] of UI`, if `TI` $\xrightarrow{c}$ `UI`.

## 5.7 Other Types and Type-insts

### 5.7.1 Higher-order Types

*Overview.* Operations (e.g. predicates) have higher-order types.

Higher-order types can never be used as values.

*Allowed Insts.* The type-inst of a higher-order type is determined by the type-insts it can take as its arguments.

*Syntax.* The term `(TI1,...,TIn) -> TI` represents an operation that takes arguments with type-insts `TI,...,TIn` and returns a value with type-inst `TI`. This notation is useful for expressing the type-inst signatures of operations (e.g. see Section A).

*Coercions.* None.

# 6 Expressions

## 6.1 Expressions Overview

Expressions represent values. They occur in various kinds of items. They have the following syntax:

⟨*ann-expr*⟩ ::= ⟨*ident*⟩ `(` ⟨*ann-expr*⟩ `,` ... `)`
| ⟨*flat-expr*⟩

⟨*flat-expr*⟩ ::= ⟨*non-array-flat-expr*⟩
| ⟨*array-literal*⟩

⟨*non-array-flat-expr*⟩ ::= ⟨*scalar-flat-expr*⟩
| ⟨*set-literal*⟩

⟨*scalar-flat-expr*⟩ ::= ⟨*ident*⟩
| ⟨*array-access-expr*⟩
| ⟨*bool-literal*⟩
| ⟨*int-literal*⟩
| ⟨*float-literal*⟩
| ⟨*string-literal*⟩

⟨*int-flat-expr*⟩ ::= ⟨*ident*⟩

| ⟨*array-access-expr*⟩
| ⟨*int-literal*⟩

⟨*variable-expr*⟩ ::= ⟨*ident*⟩
| ⟨*array-access-expr*⟩

⟨*solve-expr*⟩ ::= ⟨*ident*⟩
| ⟨*array-access-expr*⟩
| ⟨*ident*⟩ `(` ⟨*flat-expr*⟩ `,` ... `)`

FlatZinc has no operators.

The remaining kinds of expressions (from ⟨*ident*⟩ to ⟨*array-literal*⟩) are described in Section 6.2.

## 6.2 Flat Expressions

### 6.2.1 Identifier Expressions

Syntactically, any normal identifier can serve as an expression. However, in a valid model any identifier serving as an expression must be the name of a variable.

### 6.2.2 Array Access Expressions

Array elements are accessed using square brackets after an identifier:

⟨*array-access-expr*⟩ ::= ⟨*ident*⟩ `[` ⟨*int-index-expr*⟩ `]`
⟨*int-index-expr*⟩ ::= ⟨*ident*⟩
| ⟨*int-literal*⟩

For example:

```
a1[1]
```

### 6.2.3 Boolean Literals

Boolean literals have this syntax:

⟨*bool-literal*⟩ ::= `false` | `true`

### 6.2.4 Integer and Float Literals

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

⟨*int-literal*⟩ ::= `-?[0-9]+`
| `-?0x[0-9A-Fa-f]+`
| `-?0o[0-7]+`

For example: `0`, `005`, `123`, `0x1b7`, `0o777`, `-1`.

Float literals have the following form:

⟨*float-literal*⟩ ::= `-?[0-9]+.[0-9]+`
| `-?[0-9]+.[0-9]+[Ee][-+]?[0-9]+`
| `-?[0-9]+[Ee][-+]?[0-9]+`

For example: `1.05`, `-1E05`, `1.3e-5`, `-1.3+e5`; but not `1.`, `.5`, `1.e5`, `.1e5`.

A '`-`' symbol preceding an integer or float literal (without any intervening whitespace) signifies a negative integer or float literal. This allows negative numbers to be expressed in FlatZinc, which lacks the unary minus operator.

### 6.2.5 String Literals

String literals are written as in C:

⟨*string-literal*⟩ ::= `"[^"\n]*"`

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.

### 6.2.6 Set Literals

Set literals have this syntax:

⟨*set-literal*⟩ ::= { [ ⟨*scalar-flat-expr*⟩ , ... ] }
| ⟨*int-flat-expr*⟩ .. ⟨*int-flat-expr*⟩

For example:

```
{ 1.0, 3.5, 5.7 }
{ }
1 .. 10
```

The type-insts of all elements in a literal set must be the same, or coercible to the same type-inst.

### 6.2.7 Array Literals

Array literals have this syntax:

⟨*array-literal*⟩ ::= [ [ ⟨*non-array-flat-expr*⟩ , ... ] ]

For example:

```
[1, 2, 3, 4]
[]
```

In a simple array literal all elements must have the same type-inst, or be coercible to the same type-inst.

The indices of an array literal are implicitly 1..$n$, where $n$ is the length of the literal.

# 7 Items

This section describes the top-level program items.

## 7.1 Predicate Declaration Items

Predicate declarations have this syntax (repeated from Section 5.4):

⟨*pred-decl-item*⟩ ::= `predicate` ⟨*ident*⟩ ( ⟨*pred-arg*⟩ , ... )

These declare constraints supported by the FlatZinc backend that are not part of standard FlatZinc. For example:

```
predicate all_different_int(array [int] of var int: xs);
```

## 7.2 Variable Declaration Items

Variable declarations have this syntax (repeated from Section 5.4):

⟨*var-decl-item*⟩ ::= `var` ⟨*non-array-ti-expr-tail*⟩ `:` ⟨*ident-anns*⟩ [ `=` ⟨*non-array-flat-expr*⟩ ]
| ⟨*non-array-ti-expr-tail*⟩ `:` ⟨*ident-anns*⟩ `=` ⟨*non-array-flat-expr*⟩
| `array` `[` `1` `..` ⟨*int-literal*⟩ `]` `of` ⟨*array-decl-tail*⟩
⟨*array-decl-tail*⟩ ::= ⟨*non-array-ti-expr-tail*⟩ `:` ⟨*ident-anns*⟩ `=` ⟨*array-literal*⟩
| `var` ⟨*non-array-ti-expr-tail*⟩ `:` ⟨*ident-anns*⟩ [ `=` ⟨*array-literal*⟩ ]

For example:

```
var int: i;
set of int: s = {1,2,3};
array[1..3] of float: af = [1.0, 2.0, 3.0];
array[1..5] of var bool: ab;
```

A variable must be assigned in its declaration, if it is assigned at all.
Variables can have one or more annotations. Section 8 has more on annotations.

## 7.3 Constraint Items

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

Constraint items have this syntax:

⟨*constraint-item*⟩ ::= `constraint` ⟨*constraint-elem*⟩ ⟨*annotations*⟩
⟨*constraint-elem*⟩ ::= ⟨*ident*⟩ `(` ⟨*flat-expr*⟩ `,` ... `)`
| ⟨*variable-expr*⟩

For example:

```
constraint int_lt(x, 3);
constraint b;
```

The expression in a constraint item must have type-inst `bool` or `var bool`; note however that constraints with fixed expressions are not very useful.

Constraints can have one or more annotations. Section 8 has more on annotations.

## 7.4 Solve Items

Every model must have exactly one solve item, of the following form:

⟨*solve-item*⟩ ::= `solve` ⟨*annotations*⟩ ⟨*solve-kind*⟩
⟨*solve-kind*⟩ ::= `satisfy`
| `minimize` ⟨*solve-expr*⟩
| `maximize` ⟨*solve-expr*⟩

Example solve items:

```
solve satisfy;
solve minimize x;
solve maximize a[1];
solve minimize int_float_lin([3.0, 4.0], [5.0], [xi, yi], [zf]);
```

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 must have type-inst `int`, `float`, `var int` or `var float`. The first two of these are not very useful as they mean that the model requires no constraint solving.

Solve items can be annotated. Section 8 has more details on annotations.

# 8 Annotations

Annotations are term structures that 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), constraint expressions, and solve items, as Sections 7.2, 7.3 and 7.4 showed. They have the following syntax:

⟨*annotations*⟩ ::= ( `::` ⟨*annotation*⟩ )*
⟨*annotation*⟩ ::= ⟨*ident*⟩ [ `(` ⟨*ann-expr*⟩ `,` ... `)` ]

For example:

```
var int: x::foo;
constraint p(x)::bar("abc");
solve :: blah(4, blah(5, nil))
    minimize x;
```

An implementation may not recognise all annotations in a valid FlatZinc model instance. An implementation must type-inst check the arguments for those annotations it recognises, and it must issue a warning for those annotations it does not recognise.

FlatZinc's standard annotations are listed in Appendix B.

# 9 Output

Error messages and warnings must be written to the standard error stream (`stderr`). All other output must be written to the standard output stream (`stdout`).

If a solution is found, only variables with output annotations should be printed. An attempt to print a variable whose value has not been fixed by the solver should cause the solver to report an error and abort. Variables should be printed in alphabetical order (or, to be more precise, in ascending lexicographic ASCII order). The output for a solution should be suitable for input to a MiniZinc model as a data file.

Non-array output variables (annotated with `output_var`) should be printed on a single line as ⟨*name*⟩ `=` ⟨*value*⟩`;`. Array output variables (annotated with `output_array(` ⟨*index-range*⟩$_1$`,...,`⟨*index-range*⟩$_N$`)` should be printed on a single line as ⟨*name*⟩ `= array`$N$`d(` ⟨*index-range*⟩$_1$`,...,`⟨*index-range*⟩$_N$`, [` ⟨*literal*⟩`,...]);`, where ⟨*literal*⟩ is one of ⟨*bool-literal*⟩, ⟨*int-literal*⟩, ⟨*float-literal*⟩, or ⟨*set-literal*⟩.

A solution is terminated by ten consecutive dashes "`----------`" on a line by themselves.

For example: given the following model

```
var 1..10: x :: output_var;
array [1..4] of var 1..10: ys :: output_array([0..1, 0..1]);
```

```
var 1..10: z;
solve satisfy;
```

a solver may produce the following output:

```
x = 1;
ys = array2d(0..1, 0..1, [2, 7, 1, 8]);
----------
```

A solver may produce more than one solution for a `solve satisfy` goal. For a `solve maximize` or `solve minimize` goal it is permissible to print increasing approximations to an optimal result, one after another (this makes is possible to obtain a sub-optimal solution from the output before the solver has time to find the optimal result).

If a solver finishes searching its entire search tree it should print ten consecutive equals signs "`==========`" on a line by themselves. For optimisation goals, this indicates that the last solution printed is the best the solver was able to find. If the output consists solely of this line of equals signs then the solver was unable to find any solution.

For example: given the following model

```
var 1..3: x :: output_var;
solve minimize x;
```

a solver may produce the following output:

```
x = 3;
----------
x = 2;
----------
x = 1;
----------
==========
```

Output may include FlatZinc comments provided they appear on a line by themselves and start with `%` followed by a space. This is useful, for instance, for reporting statistics gathered by the solver.

## 9.1 Output of values

Values should be printed according to the following specification.

**bool** : as per

⟨*bool-literal*⟩ ::= `false` | `true`

**float** : as per

⟨*float-literal*⟩ ::= `-?[0-9]+.[0-9]+`
| `-?[0-9]+.[0-9]+[Ee][-+]?[0-9]+`
| `-?[0-9]+[Ee][-+]?[0-9]+`

**int** : as per

⟨*int-literal*⟩ ::= `-?[0-9]+`
| `-?0x[0-9A-Fa-f]+`
| `-?0o[0-7]+`

**set of int** : as either { ⟨*int-literal*⟩ , ... } or ⟨*int-literal*⟩ .. ⟨*int-literal*⟩

**array of int** : as
`array`*N*`d(` ⟨*index-range*⟩$_1$ , ... ,⟨*index-range*⟩$_N$ `,[` ⟨*int-literal*⟩ , ... `]` `)`
where

⟨*index-range*⟩ ::=⟨*int-literal*⟩ .. ⟨*int-literal*⟩

Other array types are handled similarly.

# A Built-in Operations

This appendix lists built-in predicates. They may be implemented as true built-ins, or in libraries that are automatically imported for all models. None of them are overloaded.

In all constraints, the "result" is the final argument. For example, `int_plus(x, y, z)` represents `x + y = z`.

Many built-ins have a reified equivalent with an extra final `var bool` argument; these are not listed explicitly but mentioned at the bottom of the relevant parts.

It is unlikely that a single FlatZinc implementation will implement all of these operations. For example, an implementation for a finite domain solver will probably not implement the built-ins involving floats. An implementation should abort with a clear message at compile-time if any unsupported constraints are used.

## A.1 Comparison Operations

Equality (`*_eq`). Other comparisons are similar: disequality (`*_ne`), less than (`*_lt`), greater than (`*_gt`), less than or equal (`*_le`), greater than or equal (`*_ge`). The ordering on integers and floats is the usual one. For Booleans, `false` is considered smaller than `true`. For sets, ordering is lexicographic.

```
  int_eq(var int,        var int)
float_eq(var float,      var float)
 bool_eq(var bool,       var bool)
  set_eq(var set of int, var set of int)
% also: int_eq_reif, float_eq_reif, bool_eq_reif, set_eq_reif
```

Linear equality ($a \cdot x = c$). Other linear comparisons are similar: disequality (`*_ne`), less than (`*_lt`), greater than (`*_gt`), less than or equal (`*_le`), greater than or equal (`*_ge`).

```
  int_lin_eq(array[int] of int,   array[int] of var int,   int)
float_lin_eq(array[int] of float, array[int] of var float, float)
% also: int_lin_eq_reif, float_lin_eq_reif
```

There is also a function representing a linear expression. It can be used only in a solve item.

```
int_float_lin(array[int] of float, array[int] of float,
              array[int] of var int, array[int] of var float)
```

For example, the first line of the following two lines is an example use of `int_float_lin`; the second line shows what it means (using MiniZinc syntax and operations).

```
solve minimize int_float_lin([3.0, 4.0], [5.0], [xi, yi], [zf]);
solve minimize 3.0*xi + 4.0*yi + 5.0*zf;
```

The lengths of the 1st and 3rd arguments must match, as must the lengths of the 2nd and 4th arguments; if not, it is a run-time error.

## A.2 Arithmetic Operations

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

```
  int_plus(var int,   var int,   var int)
float_plus(var float, var float, var float)
```

Unary minus. There is no unary plus operation.

```
  int_negate(var int,   var   int)
float_negate(var float, var float)
```

Integer and floating-point division and modulo. The integer operations all round towards $-\infty$.

```
  int_div(var int,   var int,   var int)
  int_mod(var int,   var int,   var int)
float_div(var float, var float, var float)
```

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

```
x == (x div y) * y + (x mod y)
```

Some illustrative examples:

```
int_div( 7,  4,  1)      int_mod( 7,  4,  3)
int_div(-7,  4, -2)      int_mod(-7,  4,  1)
int_div( 7, -4, -2)      int_mod( 7, -4, -1)
int_div(-7, -4,  1)      int_mod(-7, -4, -3)
```

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

```
  int_min(var int,   var int,   var int)
float_min(var float, var float, var float)
```

Absolute value of a number.

```
  int_abs(var int,   var int)
float_abs(var float, var float)
```

## A.3 Logical Operations

Conjunction. Other logical operations are similar: disjunction (`bool_or`) reverse implication (`bool_left_imp`), forward implication (`bool_right_imp`), bi-implication (`bool_eq`), exclusive disjunction (`bool_xor`), logical negation (`bool_not`).

```
bool_and(var bool, var bool, var bool)
```

N-ary conjunction. N-ary disjunction (`array_bool_or`) is similar.

```
array_bool_and(array[int] of var bool, var bool)
```

Clauses. `bool_clause([x1, ..., xm], [y1, ..., yn])` has the same meaning as `array_bool_or([x1, ..., xm, not y1, ..., not yn], true)`.

```
bool_clause(array[int] of var bool, array[int] of var bool)
bool_clause_reif(array[int] of var bool, array[int] of var bool,
                 var bool)
```

## A.4 Set Operations

Set membership.

```
set_in(var int,  var set of int)
% also: set_in_reif
```

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

```
set_subset(var set of int, var set of int)
% also: set_subset_reif
```

Set union. Other set operations are similar: intersection (`set_intersect`), difference (`set_diff`), symmetric difference (`set_symdiff`).

```
set_union(var set of int, var set of int, var set of int)
```

Cardinality of a set.

```
set_card(var set of int, var int)
```

## A.5 Array Operations

Element constraints for fixed and variable array accesses ($x[i] = y$). The first argument is the index.

```
array_bool_element     (var int, array[int] of     bool,  var bool)
array_var_bool_element (var int, array[int] of var bool,  var bool)
array_int_element      (var int, array[int] of     int,   var int)
array_var_int_element  (var int, array[int] of var int,   var int)
array_float_element    (var int, array[int] of     float, var float)
array_var_float_element(var int, array[int] of var float, var float)
array_set_element      (var int, array[int] of     set of int, var set of int)
array_var_set_element  (var int, array[int] of var set of int, var set of int)
```

## A.6 Coercion Operations

Explicit casts from one type-inst to another.

```
int2float(var int, var float)
bool2int (var bool, var int)
```

# B Standard Annotations

This appendix lists standard FlatZinc annotations.

## B.1 General Annotations

The following annotations can be used anywhere that annotations can be used.

```
null
```

The null annotation has no meaning. It can be useful as a place-holder.

## B.2 Solve Annotations

We base our solve annotations around the `search` predicate of ECLiPSe. Individual solvers can support extensions of the various parameters described here, and need not support all the parameter choices given; an implementation should warn about any unsupported parameters that are used in a FlatZinc model. We provide five search predicates, one for each of the decision variable types (integers, floats, Booleans and sets of integers), and a default labelling strategy.

**Integer search.**

```
int_search(array[int] of var int:vs, ann:var_select,
           ann:choice, ann:explore)
```

This specifies a search strategy to fix all the variables in the specified array of integer variables (`vs`) using the specified variable selection strategy (`var_select`), value choice method (`choice`), and exploration strategy (`explore`).

Variable selection strategies can be:

`input_order` The first entry in the list.

`first_fail` The entry with the smallest domain size.

`anti_first_fail` The entry with the largest domain size.

`smallest` The entry with the smallest value in the domain.

`largest` The entry with the largest value in the domain.

`occurrence` The entry with largest number of attached constraints.

`most_constrained` The entry with the smallest domain size, breaking ties using the number of constraints.

`max_regret` The entry with the largest difference between smallest and second smallest domain value.

Choice methods can be:

`indomain` Values are tried in increasing order, on failure the previously tried value is not removed.

`indomain_min` Values are tried in increasing order, on failure the previously tried value is removed.

`indomain_max` Values are tried in decreasing order, on failure the previously tried value is removed.

`outdomain_min` On the first branch, the smallest value is removed from the domain. On the other branch, the variable is assigned the smallest value in the domain.

`outdomain_max` On the first branch, the largest value is removed from the domain. On the other branch, the variable is assigned the largest value in the domain.

`indomain_middle` Values are tried from the middle (average of the bounds) of the domain, on failure the previously tried value is removed.

`indomain_median` Values are tried from the median of the domain, on failure the previously tried value is removed.

`indomain_random` Values are tried in random order, on failure the previously tried value is removed.

`indomain_split` Values are tried by successive domain splitting trying the lower half first.

`indomain_reverse_split` Values are tried by successive domain splitting trying the upper half first.

`indomain_interval` If the domain consists of several intervals we first branch on interval and otherwise we split.

Exploration methods can be:

`complete` Search explores all alternative choices.

`bbs(`$s$`)` Bounded backtracking search allows $s$ backtracks.

`lds(`$d$`)` Limited discrepancy search allowing at most $d$ discrepancies.

`fail` Fails immediately.

`restart(`$s$`,`$k$`,`*explore*`)` Restart search exploring using strategy *explore* allowing $s$ backtracks before restarting, and allowing at most $k$ restarts.

`geom_restart(`$s$`,`$m$`,`$k$`,`*explore*`)` Restart search allowing $s$ backtracks for the first search, then $ms$, then $m^2s$, until $m^{k-1}s$.

`credit(`$c$`,`*explore*`)` Credit based search for the top of the tree, when the credit runs out the search switches to exploration strategy *explore*.

`dbs(`$l$`,`*explore*`)` Depth-bounded search searches the top $l$ levels and then switches to exploration strategy *explore*.

**Boolean search.**

```
bool_search(array[int] of var bool: vs, string: var_select,
            string: choice, string: explore)
```

As above but for Booleans. For the choice method, only `indomain_min`, `indomain_max`, and `indomain_random` can be used.

**Float search.**

```
float_search(array[int] of var float: vs, float: prec,
             string: var_select, string: choice, string: explore)
```

As above but for floats until the precision is within `prec`. For the choice method, only `indomain_split`, `indomain_reverse_split`, and `indomain_interval` can be used.

**Set search.**

```
set_search(array[int] of var set of int: vs, string: var_select,
           string: choice, string: explore)
```

As `int_search` above but for sets of integers. We interpret the choice methods as selecting a value to try to include in the set; on failure it is excluded from the set. Only `indomain_min`, `indomain_max`, `indomain_median`, and `indomain_random` can be used. Reversed versions which try to exclude the value from the set can also be used: `outdomain_min`, `outdomain_max`, `outdomain_median`, and `outdomain_random`.

**Labelling.**

```
labelling_ff
```

Use first-fail labelling (minimal values tried first) on all problem variables.

**Multiple search annotations.** Annotations are unordered, although one can force a sequential order on multiple search annotations using `seq_search`. For example, the following forces the `int_search` to occur before the `set_search`:

```
seq_search([
  int_search(xs, first_fail, indomain_min, complete),
  set_search(ys, input_order, indomain_max, complete)
])
```

Nested `seq_search` annotations are allowed.

## B.3 Constraint Annotations

These are example annotations that can be placed on constraints. The annotations about the basic type of propagator (bounds, domain) should be standard, while priorities and more advanced approaches are unlikely to be supported by many solvers.

`bounds` Use bounds($\mathcal{Z}$) propagation for the constraint.

`boundsZ` Use bounds($\mathcal{Z}$) propagation for the constraint.

`boundsR` Use bounds($\mathcal{R}$) propagation for the constraint.

`boundsD` Use bounds($\mathcal{D}$) propagation for the constraint.

`domain` Use domain (GAC) propagation for the constraint.

`priority(int:k)` Give the propagator priority `k`.

`multiple(array[int] of string:a)` Use propagators for each of the consistencies shown in the array `a`. E.g. `multiple(["bounds","domain"])`.

`staged(array[int] of string:a)` Use the staged version of the propagator using stage array `a`.

## B.4 Variable Annotations

These are example annotations that might be placed on variable declarations. Most solvers do not support this kind of information presently.

`bounds` Only maintain bounds information for the variable.

`bitmap(int:l,int:u)` Use a bitmap from least value `l` to greatest value `u` to store the domain of the variable.

`cardinality` Maintain cardinality information for the variable.

`var_is_introduced` identifies variables that have been introduced by a source-to-source transformation (e.g., when converting MiniZinc to FlatZinc).

`is_defined_var` identifies a variable that is defined by some constraint as a function of other variables. The corresponding constraint should be annotated with `defines_var(x)`, where `x` is the variable being functionally defined by the constraint.

`output_var` identifies a non-array variable to be output.

`output_array([IndexSet1, IndexSet2, ...])` identifies an array variable to be output.

# C FlatZinc Grammar

Section 3.3 describes the notation used in the following grammar.

## C.1 Items

⟨*model*⟩ ::= ( ⟨*pred-decl-item*⟩ ; )* ( ⟨*var-decl-item*⟩ ; )* ( ⟨*constraint-item*⟩ ; )* ⟨*solve-item*⟩ ;
[ ⟨*output-item*⟩ ; ]

⟨*pred-decl-item*⟩ ::= predicate ⟨*ident*⟩ ( ⟨*pred-arg*⟩ , ... )
⟨*var-decl-item*⟩ ::= var ⟨*non-array-ti-expr-tail*⟩ : ⟨*ident-anns*⟩ [ = ⟨*non-array-flat-expr*⟩ ]
| ⟨*non-array-ti-expr-tail*⟩ : ⟨*ident-anns*⟩ = ⟨*non-array-flat-expr*⟩
| array [ 1 .. ⟨*int-literal*⟩ ] of ⟨*array-decl-tail*⟩
⟨*array-decl-tail*⟩ ::= ⟨*non-array-ti-expr-tail*⟩ : ⟨*ident-anns*⟩ = ⟨*array-literal*⟩
| var ⟨*non-array-ti-expr-tail*⟩ : ⟨*ident-anns*⟩ [ = ⟨*array-literal*⟩ ]
⟨*ident-anns*⟩ ::= ⟨*ident*⟩ ⟨*annotations*⟩

⟨*constraint-item*⟩ ::= constraint ⟨*constraint-elem*⟩ ⟨*annotations*⟩
⟨*constraint-elem*⟩ ::= ⟨*ident*⟩ ( ⟨*flat-expr*⟩ , ... )
| ⟨*variable-expr*⟩

⟨*solve-item*⟩ ::= solve ⟨*annotations*⟩ ⟨*solve-kind*⟩
⟨*solve-kind*⟩ ::= satisfy
| minimize ⟨*solve-expr*⟩
| maximize ⟨*solve-expr*⟩

## C.2 Type-Inst Expression Tails

⟨*non-array-ti-expr-tail*⟩ ::= ⟨*scalar-ti-expr-tail*⟩
| ⟨*set-ti-expr-tail*⟩

⟨*scalar-ti-expr-tail*⟩ ::= ⟨*bool-ti-expr-tail*⟩
| ⟨*int-ti-expr-tail*⟩
| ⟨*float-ti-expr-tail*⟩

⟨*bool-ti-expr-tail*⟩ ::= bool

⟨*int-ti-expr-tail*⟩ ::= int
| ⟨*int-literal*⟩ .. ⟨*int-literal*⟩
| { ⟨*int-literal*⟩ , ... }

⟨*float-ti-expr-tail*⟩ ::= float
| ⟨*float-literal*⟩ .. ⟨*float-literal*⟩

⟨*set-ti-expr-tail*⟩ ::= set of ⟨*scalar-ti-expr-tail*⟩

## C.3 Expressions

⟨*ann-expr*⟩ ::= ⟨*ident*⟩ ( ⟨*ann-expr*⟩ , ... )
| ⟨*flat-expr*⟩
⟨*flat-expr*⟩ ::= ⟨*non-array-flat-expr*⟩
| ⟨*array-literal*⟩

⟨*non-array-flat-expr*⟩ ::= ⟨*scalar-flat-expr*⟩
| ⟨*set-literal*⟩

⟨*scalar-flat-expr*⟩ ::= ⟨*ident*⟩
| ⟨*array-access-expr*⟩
| ⟨*bool-literal*⟩
| ⟨*int-literal*⟩
| ⟨*float-literal*⟩
| ⟨*string-literal*⟩

⟨*int-flat-expr*⟩ ::= ⟨*ident*⟩
| ⟨*array-access-expr*⟩
| ⟨*int-literal*⟩

⟨*variable-expr*⟩ ::= ⟨*ident*⟩
| ⟨*array-access-expr*⟩

⟨*solve-expr*⟩ ::= ⟨*ident*⟩
| ⟨*array-access-expr*⟩
| ⟨*ident*⟩ `(` ⟨*flat-expr*⟩ `,` ... `)`

⟨*array-access-expr*⟩ ::= ⟨*ident*⟩ `[` ⟨*int-index-expr*⟩ `]`
⟨*int-index-expr*⟩ ::= ⟨*ident*⟩
| ⟨*int-literal*⟩

⟨*bool-literal*⟩ ::= `false` | `true`

⟨*int-literal*⟩ ::= `-?[0-9]+`
| `-?0x[0-9A-Fa-f]+`
| `-?0o[0-7]+`

⟨*float-literal*⟩ ::= `-?[0-9]+\.[0-9]+`
| `-?[0-9]+\.[0-9]+[Ee][-+]?[0-9]+`
| `-?[0-9]+[Ee][-+]?[0-9]+`

⟨*set-literal*⟩ ::= `{` [ ⟨*scalar-flat-expr*⟩ `,` ... ] `}`
| ⟨*int-flat-expr*⟩ `..` ⟨*int-flat-expr*⟩

⟨*array-literal*⟩ ::= `[` [ ⟨*non-array-flat-expr*⟩ `,` ... ] `]`

⟨*string-literal*⟩ ::= `"[^"\n]*"`

## C.4 Miscellaneous Elements

⟨*ident*⟩ ::= `[A-Za-z][A-Za-z0-9_]*` % excluding keywords

⟨*annotations*⟩ ::= ( `::` ⟨*annotation*⟩ )*
⟨*annotation*⟩ ::= ⟨*ident*⟩ [ `(` ⟨*ann-expr*⟩ `,` ... `)` ]