

Dimensional Input Parameter Language

DIPL Specification for SciNumTools v3

Ondrej Pego Jaura

September 2026

Contents

1	DIPL - Language Specification	2
1.1	Introduction	2
1.2	Language Overview	2
1.2.1	Key Characteristics	3
1.3	Language Syntax	3
1.4	File Format	4
1.5	Execution Model	4
1.5.1	Normative Execution Model	4
1.5.2	Implementation Notes (Non-Normative)	4
1.6	Error Handling	5
1.7	Versioning	5
1.8	Reference Implementation	5
1.9	Glossary	5
2	Nodes	6
2.1	Definition	6
2.2	Modification	6
2.3	Declaration	7
2.4	Hierarchy	7
2.4.1	Value nodes	7
2.4.2	Container nodes	7
3	Data Types	9
3.1	Standard data types	10
3.2	Derived data types	10
4	Values	11
4.1	Empty values	11
4.2	Scalars	11
4.3	Arrays	11
4.4	Blocks	12
4.5	Tables	12
5	References	13
5.1	Reference Notation	13
5.1.1	Source and absolute references	13
5.1.2	Relative references	14
5.1.3	Self-reference	14
5.2	Source Declarations	15
5.3	Imports	15
5.4	Injectons	16
5.4.1	Value Injection	16
5.4.2	Array and String Slicing	16
5.4.3	Raw Source Injection	17
5.4.4	Source and Unit Declaration Injection	17

6	Expressions	17
6.1	Logical	18
6.2	Numerical	19
6.3	Templates	20
7	Units	21
7.1	Standard Units	21
7.2	Custom Units	21
8	Functions	22
8.1	Function Calls	22
8.2	Function Arguments	22
8.3	Function Results	22
8.4	Evaluation and Errors	22
9	Properties	23
9.1	Directives	23
9.1.1	Options	23
9.1.2	Condition	23
9.1.3	Format	24
9.1.4	Constants	24
9.1.5	Tags	24
9.1.6	Delimiter	24
9.2	Metadata	25
10	Conditions	25
11	Schemas	27
11.1	Declaration and Instantiation	27
11.2	Collection schemas	28
11.3	Nested Schemas	28
11.4	Rules	28

1 DIPL - Language Specification

1.1 Introduction

DIPL (Dimensional Input Parameter Language) is a domain-specific language for describing input parameters used by scientific simulations and numerical software.

Conventional configuration formats such as JSON, YAML, and INI are well suited for representing general-purpose data, but provide little support for concepts that are common in scientific computing. In particular, physical units, dimensional consistency, numerical types, parameter dependencies, and validation rules typically have to be implemented separately by the consuming application.

DIPL addresses these requirements at the language level. It provides a compact notation in which parameter values, units, structure, relationships, and validation rules can be expressed together and subsequently evaluated by a DIPL implementation.

The remainder of this specification defines the syntax and semantics of the language and describes the constructs available for building DIPL documents.

1.2 Language Overview

DIPL (Dimensional Input Parameter Language) is a declarative, strongly typed domain-specific language for defining structured scientific parameters, constraints, and relationships in a human-readable format.

A DIPL document consists of parameter declarations organized into hierarchical blocks through indentation. Each parameter combines:

- a name
- an explicit type
- an optional shape or structure
- a value expression

- an optional unit annotation
- optional properties for metadata, validation, and constraints

```
velocity float32[1,2:] = [[23.45, 23e-34, 45.1]] SI_km/s
!condition ({.} < {?cfl_limit})
```

```
burst_energy float64 = 2.34e5 US_btu
```

A DIPL document is more than a collection of static values. It represents a structured and semantically validated configuration graph in which parameters, expressions, references, units, and constraints work together.

At a conceptual level:

- **Nodes** define the values and hierarchical structure of the configuration.
- **References and expressions** establish dependencies between parameters and allow values to be derived from other values.
- **Units** provide dimensional information and ensure physical consistency during evaluation.
- **Properties** attach metadata and declarative constraints to the configuration.
- **Schemas** define expected structures and provide a basis for validating complete configurations.

This design allows DIPL to represent not only data, but also the semantic relationships and domain-specific rules governing that data. The resulting document can be interpreted both as **data** and as an **executable configuration model**: values can be evaluated, dependencies resolved, and semantic constraints validated by the DIPL interpreter.

1.2.1 Key Characteristics

Strong typing All parameters declare explicit types (e.g., `bool`, `uint32`, `float64`, `str`), enabling predictable behavior, type checking, and validation.

Units-aware computation Values may include physical units, which are automatically normalized and validated during evaluation, allowing dimensional consistency to be maintained throughout computations.

Declarative constraints and metadata Properties such as `!condition` define constraints that must be satisfied for a valid document, while metadata such as `?author` provides descriptive information about individual nodes.

Hierarchical organization An indentation-based structure allows related parameters to be organized into groups, maps, and lists without additional syntactic overhead.

Referential expressions Parameters can reference other values within the document, allowing expressions to be built from existing data and making dependencies between parameters explicit.

Derived parameters Parameters can be defined in terms of other parameters, allowing simple dependency graphs to be expressed directly within the language.

External references Values and nodes can reference data from external sources, allowing DIPL documents to incorporate information without duplicating it locally.

Composable schemas Schemas provide a mechanism for describing the expected structure and constraints of data, supporting consistent definitions and validation across documents.

1.3 Language Syntax

The following chapters describe the syntax of the Dimensional Input Parameter Language. They introduce the individual language constructs used to define, represent, reference, and process scientific input data.

The syntax is presented from the fundamental building blocks of the language towards more advanced constructs. The chapters cover nodes and data types, values and units, references and expressions, as well as functions, properties, conditions, and schemas.

The following topics are covered:

- Nodes — the fundamental building blocks of a DIPL document.
- Data Types — the types available for representing data.
- Values — how values are represented and assigned.
- References — how data can refer to other nodes and sources.
- Expressions — expressions for deriving and transforming values.
- Units — dimensional quantities and their units.

- Functions — functions available for processing and transforming data.
- Properties — metadata and properties associated with nodes.
- Conditions — conditional definitions and constraints.
- Schemas — schemas for defining and validating structured data.

1.4 File Format

The source file **MUST** have an extension of either `.dip` or `.dip1`. Nodes with the smallest indentations in the file should be considered as the root nodes with an indentation level of 0.

1.5 Execution Model

1.5.1 Normative Execution Model

Evaluation of a DIPL document **MUST** proceed in the following stages, in order:

1. Parsing

The input **MUST** be parsed into a structured representation of nodes. This includes processing indentation, node types, and structural relationships.

2. Dependency Resolution

All references **MUST** be resolved into a directed acyclic graph (DAG) of node dependencies.

- Unresolved sources or node references **MUST** result in an error.
- Cyclic dependencies **MUST** result in an error.

3. Reference Evaluation

All reference expressions (e.g. `{<source>?<path>}`, `{?<path>}`) **MUST** be resolved to their corresponding values or node sets. Remote sources **MUST** be processed independently before their results are used.

4. Expression and Value Evaluation

All node values **MUST** be evaluated, including:

- scalar values
- arrays and structured values
- expressions involving references If evaluation fails (e.g. invalid operations or unresolved values), evaluation **MUST** fail.

5. Unit Normalization

All evaluated values **MUST** be converted to the canonical unit representation defined in the Units specification. Unit incompatibility **MUST** result in an error.

6. Condition Evaluation

All conditions **MUST** be evaluated using the normalized value of the node. The `{.}` self-reference refers to this value.

- If a condition evaluates to false, the node **MUST** be considered invalid.
- If condition evaluation fails, evaluation **MUST** fail.

7. Validation

All validation rules **MUST** be applied, including:

- option constraints
- value format checks
- structural constraints

Any violation **MUST** result in an error.

Except for host-defined functions, conforming implementations **MUST** produce identical results for identical inputs and sources. Functions in DIPL are exception from this rule, because they are user defined and depend on the particular implementation and use case. If any stage fails, evaluation **MUST** terminate with an error.

1.5.2 Implementation Notes (Non-Normative)

A DIPL processor may implement the above stages using steps such as:

- parsing code lines into nodes
- combining multiline string blocks into single strings (e.g. text wrapped in `"""`)
- replacing special symbols with temporary markers (e.g. `\n`, `\'`, `\"`) and restoring them later
- determining node types (e.g. group, value, case)
- constructing node hierarchy and full names (e.g. `foo.bar.baz`)
- assigning property nodes to preceding value nodes
- validating indentation and structural consistency
- resolving references and expanding nodes

- evaluating expressions and parsing values (e.g. scalars like `34e+3`, arrays like `[[1,3],[4,5]]`)
- parsing and applying unit expressions (e.g. `kg/s`)
- applying modifications to existing nodes
- evaluating conditional branches and case expressions
- validating nodes against constraints (options, conditions, formats)

These steps are provided for guidance only. Implementations MAY use different internal strategies, provided the observable behavior conforms to the normative execution model defined above.

1.6 Error Handling

The following conditions MUST result in evaluation failure:

- unresolved source or node reference
- cyapic dependencies
- invalid query path
- type mismatch
- unit incompatibility
- expression evaluation failure

Errors MUST be deterministic and MUST NOT be ignored.

1.7 Versioning

Language versioning is independent of any particular implementation. Each implementation MUST explicitly declare the version of the DIPL language it supports.

Future versions of the language MAY introduce extensions or modifications. Backward compatibility SHOULD be preserved where feasible.

1.8 Reference Implementation

A reference implementation of the DIPL language is provided as part of the SciNumTools v3 project.

The reference implementation includes: - A C++ implementation - Python bindings for integration and scripting use cases

Project: SciNumTools v3

Repository: <https://github.com/vrtulka23/scinumtools3>

Documentation: <https://vrtulka23.github.io/scinumtools3/>

Issues: <https://github.com/vrtulka23/scinumtools3/issues>

Independent implementations of the DIPL language (e.g., in Rust or Julia) are encouraged and welcome, and can be added to this list on request.

1.9 Glossary

Term	Meaning
Node	Fundamental element of a DIPL document
Value node	Node containing a typed value
Container node	Node containing child nodes
Path	Fully-qualified node path in the hierarchy
Group	Named container whose children are addressed by name
Map	Container whose children are addressed by key
List	Container whose children are addressed by index
Schema	Reusable structural contract for nodes
Definition	Statement creating a node and assigning its value
Declaration	Statement creating a typed node without a value
Modification	Statement assigning a value to an existing node
Source	Named external DIPL/text domain
Reference	Expression identifying existing nodes or external content
Import	An operation that inserts referenced nodes into the current node hierarchy
Injection	An operation that injects value to a node from local, or remote sources

Term	Meaning
Unit	Named physical or dimensional quantity
Expression	Construct that evaluates to a value
Property	Information attached to a node
Condition	Boolean validation rule
Metadata	Non-semantic descriptive information attached to a node

2 Nodes

The DIPL language distinguishes three different ways to create a parameter, depending on which parts of the parameter node are specified. A parameter can be fully defined, meaning that all node components—name, type, value, and optionally shape and units—are provided. Alternatively, a parameter can be declared by specifying only its name, type, and optional shape/units. After a parameter has been either defined or declared, its value component can be modified. Additionally, nodes can be ordered into a hierarchical structure by using indents.

2.1 Definition

All members of definition are separated with at least one empty space, and their order is not interchangeable.

Indentation is at the beginning of a line and it determines node hierarchy. Every new indentation level should be separated by two empty spaces.

Node **names** are formed from letters, numbers, underscores and hyphens. Dots have a special function in names, because they separate atomic (parent and child) node names.

There are four main **data types** in DIPL that are based on data types used in Python language: boolean (`bool`), integer (`int`), float (`float`) and string (`str`).

Node **values** are separated from the left-hand members by an equal sign. Unquoted values consist only of non-whitespace characters. If a value contains spaces, it must be enclosed in double quotes. If a node value spans multiple lines, a block must be used. A node value may be continued on the following line by placing an equal sign at the beginning of a line indented exactly two spaces further than the node definition. DIPL parameter values are described in detail in a separate section.

```
street str = "Fifth Avenue"
number int = 350
sizes float
    = [3.14, 2.718, -1.25, 0.577, 42.0, -7.5, 1.618, 0.001, 12.345, -0.333]
```

The two data types that support **units** are integers and floats. Units are written directly after a node value and are separated with an empty space. More detailed description of units is given in a chapter about units. In this subsection, we only describe basic syntax with respect to the nodes.

```
weight int = 87 kg
height float = 185 cm
```

Comments are always at the end of lines and start with a hash sign. It is also possible to use comments on empty lines to describe the code.

```
resolution int = 32 # low resolution
```

2.2 Modification

The first occurrence of a node is called a definition. All subsequent occurrences of a node with the same name are called modifications. The last modification is the one used in the parameter output. The data type of node needs to be set in each definition; however, it can be omitted in subsequent modifications. If a node is a dimensional parameter, units have to be set in a definition. One can use different units of the same dimension in modifications; however, the final value of the node will always be converted into units set by the definition.

```
size float = 70 cm    # definition
size float = 80 cm    # modifying only value
size = 90 cm          # omitting datatype
size = 100            # omitting units
size = 1 pc           # using different units of length
```

```
# Final parameter value is:
# size = 3.086e18 cm
```

2.3 Declaration

Declaration in DIPL is a special case of definition where equal sign and value part is not specified. The value must be later set in subsequent DIPL code using modifications, otherwise code will not be valid. Similarly to definitions, modification units are automatically converted to their declared units.

```
weight float kg    # declaration
weight = 2.3 t     # modification

# Final parameter value is:
# weight = 2.3e3 kg
```

2.4 Hierarchy

DIPL nodes are organized in a hierarchical way using indentation, i.e. number of empty spaces before nodes. **Parent** nodes have lower indentation as their **children** nodes. **Siblings** are nodes which share a common parent and indentation level. Multiple levels of hierarchy are also allowed, and there can be empty lines between the nodes. The number of empty spaces for each indentation level is two and tabs are not permitted.

2.4.1 Value nodes

```
grandfather str = "John"    # parent of Peter and Cintia
  father str = "Peter"      # John's child, Cintia's sibling
    son str = "Benjamin"    # Ben's and Lucia's parent
    daughter str = "Lucia"  # Peter's child
  aunt str = "Cintia"       # John's child, Peter's sibling
```

Both parent and children nodes can be either definitions, modifications or declarations. Nodes in the above example are equivalent to the fully-qualified **path notation** given below.

```
grandfather str = "John"
grandfather.father str = "Peter"
grandfather.father.son str = "Benjamin"
grandfather.father.daughter str = "Lucia"
grandfather.aunt str = "Cintia"
```

2.4.2 Container nodes

Besides basic value nodes, DIPL supports three types of container nodes.

Structure	Example	Semantics	Similar in C++	Similar in Python
Group	foo	Named child nodes	<code>struct</code>	<code>@dataclass</code>
Map	foo[key]	Child nodes addressed by key	<code>std::map</code>	<code>dict</code>
List	foo[]	Child nodes addressed by index	<code>std::vector</code>	<code>list</code>

Unlike value nodes, container nodes do not directly hold values. Instead, they organize child nodes into hierarchical structures. Depending on the container type, child nodes are addressed by name, key, or index.

Groups

Nodes may be organized into logical hierarchies using a special container node called a **group**.

Unlike value nodes, group nodes do not directly store values. Their sole purpose is to organize child nodes into a logical structure. The group's name becomes part of the fully-qualified path of all descendant nodes.

During parsing, node names are expanded into fully-qualified paths by prepending the names of all parent groups, separated by dots (.).

Node names may themselves contain dot-separated path segments. Both hierarchical notation and explicit path notation are equivalent and may be freely combined.

```

family                                     # group node
  father str = "Peter"
    son str = "Benjamin"                 # hierarchical notation

  father.daughter str = "Lucia" # mixed notation

family.aunt.dog str = "Lassie" # explicit path notation

```

The example above produces the following fully-qualified node paths:

```

family.father = "Peter"
family.father.son = "Benjamin"
family.father.daughter = "Lucia"
family.aunt.dog = "Lassie"

```

Collections

Collections extend group nodes to represent associative and sequential data structures.

Explicit declaration of a map or list collection is optional and is used only when defining collection schemas. A collection can be explicitly declared by adding the corresponding data type keyword immediately after the group node name:

```

cars map    # declaration of a map collection
ships list  # declaration of a list collection

```

In most cases, explicit declaration is unnecessary. A collection is implicitly established by its first item. A collection item is specified by appending square brackets to the node name:

- `name[key]` denotes an item in a keyed collection (**map**).
- `name[]` denotes an item in an indexed collection (**list**). Item indices are assigned implicitly, starting at 0.

A collection path shall use a consistent collection type. Once a collection has been established as keyed or indexed, all subsequent items at the same path shall use the same collection type. Mixing keyed and indexed items within a single collection is not permitted.

Collections may be nested.

Collection selectors are interpreted according to the type of the referenced collection.

When resolving a fully-qualified path, all parent collections shall already exist. Since the container type of each parent node is therefore known, selectors of the form `[value]` can be unambiguously interpreted either as map keys or list indices.

Only the final path segment may create a new node or collection item.

```

basket                                     # group

  food map                                # declaration (optional)

  food[vegetables]                        # keyed collection
    carrots float = 0.5 kg
    potatoes float = 2 kg

  food[fruits]
    apples int = 3
    water_melons int = 1

  berries list                            # declaration (optional)

  berries[]                               # indexed collection
    name str = "strawberry"
    weight float = 300 g

  berries[]
    name str = "cherry"
    weight float = 500 g

```

Equivalent fully-qualified path notation:

```
basket.food[vegetables].carrots float = 0.5 kg
basket.food[vegetables].potatoes float = 2 kg

basket.food[fruits].apples int = 3
basket.food[fruits].water_melons int = 1

basket.food[fruits].berries[0].name str = "strawberry"
basket.food[fruits].berries[0].weight float = 300 g

basket.food[fruits].berries[1].name str = "cherry"
basket.food[fruits].berries[1].weight float = 500 g
```

Equivalent JSON representation:

```
{
  "basket": {
    "food": {
      "vegetables": {
        "carrots": 0.5,
        "potatoes": 2.0
      },
      "fruits": {
        "apples": 3,
        "water_melons": 1,
        "berries": [
          {
            "name": "strawberry",
            "weight": 300
          },
          {
            "name": "cherry",
            "weight": 500
          }
        ]
      }
    }
  }
}
```

Collection items are accessed and modified in the same manner as regular nodes. For indexed collections, the item index shall be specified explicitly.

```
basket.food[fruits].apples = 3
basket.food[fruits].berries[1].weight = 250 g
```

3 Data Types

In this section we describe standard and derived data types that can be used in DIPL code. Standard types are categorized based on their value types into **booleans**, **integer numbers**, **floating-point numbers** and **strings**.

Numerical types may have derived types that specify additional representation properties such as **width**, **precision**, or **signedness**.

For integer types, derived types include signed and unsigned variants with different widths, such as `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, and `uint64`.

For floating-point types, derived types specify the floating-point precision, such as `float16`, `float32`, and `float64`. `float16` is optional: an implementation may provide the type when its compiler and runtime support a usable 16-bit floating-point representation. The special case `float128` is a separate standard type, because its support on different platforms varies.

3.1 Standard data types

DIPL provides a minimal set of four built-in scalar data types. These types are designed to cover the most common use cases while keeping the language simple and predictable. Each type represents a distinct category of values and enforces basic constraints on how data is stored and interpreted.

Below is an overview of the supported scalar types and their typical usage.

Boolean (`bool`)

The Boolean type represents logical truth values. It is primarily used for control flow, conditional evaluation, and state representation.

```
day bool = true
night bool = false
```

Integer (`int`, `int32`, or `uint`, `uint32`)

The Integer types represent whole numbers without fractional components. The signed Integer type (`int`, or equivalently `int32`) has a signed 32-bit semantic range, allowing both positive and negative values within a fixed range. The unsigned Integer type (`uint`, or equivalently `uint32`) has an unsigned 32-bit semantic range and therefore represents only non-negative values.

In the reference implementation, Integer values are internally represented using signed or unsigned 64-bit integers (`int64_t` or `uint64_t`), respectively.

```
year int = 2023
count uint = 42
```

Float (`float`, or equivalently `float64`, internally represented as a 64-bit floating-point value)

The Float type represents real numbers with fractional components. It has a 64-bit floating-point semantic representation, providing a wide range of values, including very large and very small numbers. In the reference implementation, Float values are internally represented using `double`.

```
duration float = 10           # integer form
weight float = 23.3           # floating form
distance float = 2.3e20        # scientific form
```

String (`str`)

The String type represents sequences of characters. In DIPL, all strings must be enclosed in double quotes, or in triple double quotes when spanning multiple lines. It is used for textual data, labels, and any form of human-readable content.

```
country str = "United States" # single line text
address str = """
350 Fifth Avenue
NY 10118
"""                               # text on multiple lines
```

3.2 Derived data types

When DIPL parses parameters for programming languages such as C/C++ and Fortran, it is sometimes necessary to explicitly specify the precision and representation of integer and floating-point values. This is achieved through the use of derived data types.

Derived data types are semantically related to their corresponding standard types but may use a different internal representation. They carry additional metadata describing properties such as signedness and precision. Multiple DIPL types may share the same internal representation while retaining distinct semantics.

In C/C++, a string is typically represented as a sequence of characters. In DIPL, `char` and `str` are distinct types. The `char` type represents a single unsigned 8-bit character value, while `str` represents a sequence of characters. This distinction facilitates interoperability with C/C++ without imposing C/C++'s implementation-defined `char` signedness on DIPL.

The `byte` and `char` types are optional 8-bit types representing a single byte. Both are internally represented by `uint8_t`. The `byte` type represents an unsigned 8-bit numeric value intended for raw binary data, while `char` represents an unsigned 8-bit character value. Neither type specifies an inherent text encoding.

List of standard and derived data types

Standard	Derived	Internal representation
<code>bool</code>		<code>uint8_t</code>
<code>int</code>	<code>int8</code> , <code>int16</code> , <code>int32</code> , <code>int64</code>	<code>int64_t</code>
<code>uint</code>	<code>uint8</code> , <code>uint16</code> , <code>uint32</code> , <code>uint64</code>	<code>uint64_t</code>
<code>float</code>	<code>float16</code> (optional), <code>float32</code> , <code>float64</code>	<code>double</code>
<code>float128</code>		implementation-defined
<code>char</code>		<code>uint8_t</code>
<code>byte</code>		<code>uint8_t</code>
<code>str</code>		<code>string</code>

[!NOTE] The `float16` and `float128` types are optional. Many platforms and compilers do not provide native support for 16-bit and 128-bit floating-point arithmetic. An implementation that does not support `float16` MUST reject it as an unsupported datatype; an implementation that supports it MUST document its representation, conversion, and arithmetic behavior.

The `byte` and `char` types are optional. Both types are unsigned 8-bit types and are internally represented by `uint8_t`. `byte` values may be assigned from integer literals, while `char` values may be assigned from character or string literals. Neither type specifies a text encoding.

4 Values

4.1 Empty values

Similarly, as in Python, DIPL parameters can also be set empty using a special keyword `none`. Operations with empty parameters conform to basic Python rules and may lead to corresponding warnings, or errors. DIPL parameters with `none` values are considered to be fully defined, rather than only declared.

```
john
  car str = none
```

4.2 Scalars

Nodes with a single boolean, number or string value are called scalar nodes. If a node is scalar at its definition, it can be modified only with one value, or as empty. Example of scalar node values were already shown in section about data types.

```
hair bool = false
name str = "John"
age int = 73
height float = 173 cm
```

4.3 Arrays

Nodes can also store multiple values in arrays. Dimensionality of such arrays have to be specified next to the type using bracket notation.

Dimensionality of arrays

Example,	Description
<code>[:]</code>	Array can have arbitrary number of values
<code>[3:]</code>	Array must have minimum of 3 values
<code>[:5]</code>	Array must have maximum of 5 values
<code>[1:4]</code>	Array must have between 1 and 4 values
<code>[6:, :8, 2:7]</code>	Settings of individual dimensions are separated by comma

Parsing of array values is handled by a custom parser that extracts individual values and array shape. Final values of nodes are automatically validated according to defined conditions.

```

data1 bool[4]   = [true,false,false,true] # exactly four values
data2 int[:]    = [0,1,2,3,4,5,6]         # any number of values
data3 float[3:] = [0,1.34,1.34e4]         # at least 3 values
data4 float[:4] = [0,1.34,1.34e4]         # maximum of 4 values
data4 str[3:4]  = ["John","Peter","Simon"] # between 3 and 4 values

```

Multiple values above are written in a tight notation without empty spaces in between the values. Loose notation, with empty spaces, is also permitted. Additionally, arrays can be parsed from string values, wrapped into double quotes.

```

data1 bool[4]   = [true, false, false, true]
data2 int[4]    = "[1, 2, 3, 4]"
data3 str[3:4]  = "[\"John\", \"Peter\", \"Simon\"]"

```

Multidimensional arrays are defined similarly using nested bracket notation. Irregular array shapes, in which items along the same dimension contain different numbers of elements, are not allowed. Array values may span multiple lines.

```

matrix int[2,3] = [
    [0,1,2],
    [3,4,5]
]

```

Node units apply to all values in an array.

```

mass float[2:,:2] = [[25,50],[34.2,95.1],[1e3,1e4]] kg

```

4.4 Blocks

If node values are large or span multiple lines, block notation can be used. Block notation encloses values in triple quotation marks, similarly to Python. For numerical data types, units can be specified after the closing quotation marks.

An important advantage of parsing arrays from strings is that large datasets can be imported from external files using raw imports. See the chapter on references for more information.

```

# velocity field
velocity int[4,4] = """
[[ 0, 1, 2, 4],
 [ 5, 6, 7, 8],
 [ 9,10,11,12],
 [13,14,15,16]]
""" km/s # units are optional for numerical data types

```

```

# large text
text str = """
Lorem ipsum dolor sit amet, consectetur adipiscing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat
nulla pariatur. Excepteur sint occaecat cupidatat non proident,
sunt in culpa qui officia deserunt mollit anim id est laborum.
"""

```

4.5 Tables

When working with large amounts of data, presenting the information in a table can be more concise and easier to read. For this purpose, a dedicated node type called `table` is provided. This data type parses DIPL nodes from tabulated datasets that are supplied as a block value. As a convention, imported tabulated data files should have an extension `.dipt`.

The table structure closely resembles the standard CSV format but uses a specialized header syntax. The table header contains node declarations corresponding to the columns of the table. Each declaration must begin on a new line and must not be indented.

The table data section is separated from the header by a line containing three minus signs (---). Within the table body, individual values are separated by a single space. Alternative delimiters can be specified using the `!delimiter` node option, as described in the chapter on node properties.

```
output table = """
snapshot int
time float s
intensity float W/m2
---
0 0.234 2.34
1 1.355 9.4
2 2.535 3.4
3 3.255 2.3
4 4.455 23.4
"""
```

Table notation above is equivalent to:

```
output
  snapshot int[5] = [0,1,2,3,4]
  time float[5] = [0.234,1.355,2.535,3.255,4.455]
  intensity float[5] = [2.34,9.4,3.4,2.3,23.4] W/m2
```

[!NOTE] Table nodes can be declared similarly to other value nodes and defined later. This is especially useful when using `schemas`.

5 References

References have two main applications. One can either import some already parsed DIPL nodes into a new location, or inject other node values or contents of text files into a new node. Besides the two cases, references are also used in conditions and condition properties that are explained in a separate chapter.

5.1 Reference Notation

A DIPL reference identifies either content from a source or a node within a domain. A reference consists of an optional source identifier and an optional path, separated by `?`. The `.` symbol has different meanings depending on its position within a reference: within a path, it separates hierarchical node names; at the beginning of a relative reference, it determines the reference level; at the end of a path, it selects the complete subtree rooted at the target node rather than the node itself; and alone, `{.}`, it denotes a self-reference to the current node.

5.1.1 Source and absolute references

The following forms are absolute references:

Reference	Meaning
<code>{source}</code>	Raw content of source
<code>{?path}</code>	Value of the node path in the local domain
<code>{?path.}</code>	All descendants of path in the local domain
<code>{?} / {?.}</code>	Complete source node tree in the local domain
<code>{source?path}</code>	Value of the node path in source
<code>{source?path.}</code>	All descendants of path in source
<code>{source?} / {source?.}</code>	Complete source node tree in source

The `?` separates the source identifier from the node path. If the source component is omitted, the reference targets the **local domain**, consisting of the nodes parsed from the current DIPL file.

A path without a trailing `.` MUST resolve to a single node. A path ending in `.` selects the complete subtree rooted at the target node, including all descendant nodes recursively, and therefore produces a node set. A path MUST resolve to an existing node; otherwise evaluation MUST fail. A node set used where a single value is required MUST also cause evaluation to fail. The trailing `.` MAY be omitted when the entire source node tree is imported (`{?}`, `{source?}`). In these cases, the omitted trailing `.` is semantically equivalent to the `?` symbol.

A reference of the form `{source}` has no path component and returns the raw content of the referenced source as a string. Node-based access is provided only when a path component is specified. A referenced DIPL source is processed as a separate remote domain.

Absolute references can therefore be used both to **import nodes** and to **inject values**:

```
driver
  {?child}           # single local node
  {garage?worker}    # single remote node

passengers
  {?family.}         # local subtree
  {tourists?family.} # remote subtree

sites
  {?.}               # all local nodes
  {london?.}         # all remote nodes
  # can be used also without trailing dot: {?} and {london?}

corpus str = {book}      # raw source content
chapter str = {?introduction} # local node value
citation str = {law?paragraph} # remote node value
```

The expected result type **MUST** be compatible with the context in which the reference is used.

5.1.2 Relative references

A relative reference resolves its path against a node in the current node hierarchy:

```
{.<path>}
{..<path>}
{...<path>}
...
{.<path>..}
{..<path>..}
{...<path>..}
...
```

The number of leading dots determines the reference level:

- `{.<path>}` resolves `path` relative to the current node;
- `{..<path>}` resolves `path` relative to the parent;
- `{...<path>}` resolves `path` relative to the grandparent;
- in general, `N` leading dots resolve the path relative to the node `N − 1` levels above the current node.

As with absolute references, a trailing `.` selects the complete descendant subtree rather than a single node.

A relative reference **MUST NOT** traverse beyond the root node. Such a reference is invalid and **MUST** result in an error.

```
uncle str = "John"
aunt str = "Jessica"
father str = "William"
  daughter str = {..aunt}
  son str = "Elijah"
    grandson str = {...uncle}
brother-in-law str = {..father}
```

Here `{..aunt}` resolves `aunt` from the parent context of `daughter`, while `{...uncle}` resolves `uncle` from the corresponding ancestor context of `grandson`. `{..father}` resolves `father` relative to the hierarchy containing `brother-in-law`.

5.1.3 Self-reference

`{.}` is a special self-reference and is distinct from `{.<path>}`. It refers to the **fully evaluated value of the current node** and is valid only within condition properties.

Before `{.}` is evaluated, the current node **MUST** have undergone:

1. resolution of all references;
2. evaluation of all expressions; and
3. normalization to the canonical unit representation defined by PUEL.

The resulting value retains its complete type information, including dimensionality and unit.

If evaluation of the current node fails, including because of an unresolved reference, type error, or unit incompatibility, `{.}` is undefined and evaluation of the containing condition **MUST** result in an error.

```
speed int = 78 kph
!condition ({.} < 80 kph)
```

`{.}` **MUST NOT** be used outside condition properties.

5.2 Source Declarations

Sources are named using the `$source` declarator:

```
$source name = "path/to/file.dip"
```

A source declaration associates a source identifier with a file path. The source may subsequently be referenced any number of times using the absolute reference forms described above.

Sources declared inside a DIPL file **MUST** resolve their paths relative to the location of that DIPL file.

Implementations **SHOULD** additionally support defining sources through the host environment (code interface). Source paths supplied through the code interface **MUST** be interpreted relative to the calling context.

For example:

```
$source garage = "vehicles/garage.dip"
$source tourists = "people/tourists.dip"
$source book = "texts/book.txt"
```

```
driver
  {garage?worker}

passengers
  {tourists?family.}

corpus str = {book}
```

Thus, `$source` declarations establish the mapping between source identifiers and external files, while reference notation determines whether the reference accesses raw source content, a node value, a node subtree, or the evaluated value of a node in the current hierarchy.

5.3 Imports

Imports can be used to insert referenced nodes directly into the current DIPL hierarchy. Name paths of imported nodes are embedded into the current node hierarchy as shown in the following examples.

```
icecream
  waffle str = "standard"
  scoops
    strawberry int = 1
    chocolate int = 2

bowl
  {?icecream.scoops.}      # select all children nodes
plate {?icecream.waffle}   # select a specific node
```

The node hierarchy above is equivalent to the one given below.

```
icecream.waffle = "standard"
icecream.scoops.strawberry = 1
icecream.scoops.chocolate = 2
```

```

bowl.strawberry = 1
bowl.chocolate = 2
plate.waffle = "standard"

```

The example above demonstrates importing local nodes; the same mechanism applies to external DIPL files. In this case, a source name must be specified before the question mark.

```
$source pantry = "pantry.dip"
```

```

bag {pantry?.}           # import all nodes
bowl
  {pantry?fruits}        # selecting a specific node
  {pantry?veggies.potato} # selecting a specific subnode
plate {pantry?veggies.}  # selecting all subnodes

```

So far, we have shown how to import regular nodes from a local or remote source. It is, however, also possible to import sources and custom units in the similar way. The request can select either one {<source>?<path>} or all {<source>?.} sources/units.

[!NOTE] Request path is in this case not a node path but name of a source/unit.

Importing sources/units enables users to dynamically modify numerical code units and setting scripts via their DIPL.

```

$source init = "initial/settings.dip"
$source {init?.}           # all sources of 'init' are imported
$unit {units?.}           # all units are imported from an imported source 'units'
weight float = 23 [mass]  # using imported unit

```

5.4 Injections

Injections provide values to node definitions and modifications. They do not insert complete nodes or node sets.

5.4.1 Value Injection

A valid value injection MUST resolve to either:

- a single node, referenced without a trailing .;
- raw source content, referenced without a path.

When a node is referenced, its value is injected into the receiving expression and is subject to the type, dimensionality, and unit rules of that expression.

```

pop float = 5 km
foo float = 34 cm
bar float = {?foo} m      # resulting value is 0.34 m
baz float = {?foo}        # dimension mismatch
foo = {?pop}              # modification results in 500000 cm

```

Values can also be injected from remote DIPL sources:

```

$source pressure = "pressures.dip"
pressure float = {pressure?magnetic}

```

For injections, a path always identifies the node whose **value** is to be injected. Unlike node imports, injection paths MUST NOT select a node set.

5.4.2 Array and String Slicing

Array values can be injected directly or sliced to match the required dimensions of the receiving node. Strings and arrays use the same slicing notation as Python.

A single array element can be selected:

```

sizes float[3] = [34,23.34,1e34] cm
mysize float = {?sizes}[1]

```

A range of elements can be selected:

```
masses float[2,2] = [[34,23.34],[1,1e34]] cm
mymass float[2] = {?masses}[:,1]
```

Strings can be sliced in the same way:

```
person str = "Will Smith"
surname str = {?person}[5:]
```

The resulting values are:

```
sizes = [3.400e+01, 2.334e+01, 1.000e+34]
mysize = 2.334e+01
masses = [[34,23.34],[1,1e34]]
mymass = [23.34,1e34]
person = "Will Smith"
surname = "Smith"
```

5.4.3 Raw Source Injection

Raw source injection provides a convenient way to keep large text or data blocks in external files. This separates data from the DIPL structure and makes both the source and the data easier to maintain.

When a reference contains no `?` path component, the referenced source is injected as text rather than as a node list.

```
$source velocity = "velocity.txt"
$source outputs = "outputs.txt"
$source message = "message.txt"

velocity int[3,4] = {velocity} km/s # import an array
outputs table = {outputs} # import a table
message str = {message} # import text
```

The receiving node is responsible for interpreting the injected text according to its declared type.

5.4.4 Source and Unit Declaration Injection

Values used by `$source` and `$unit` declarations can themselves be injected from nodes. In these declarations, the injected value supplies the declaration argument rather than the value of an ordinary node.

For `$source`, the referenced node **MUST** contain a string. For `$unit`, the referenced node **MUST** contain a floating-point or integer value.

```
refs str = "path/to/sources.dip"
$source refs = {?refs}

mass float = 1 kg
$unit mass = {?mass}
```

In both cases, the path in the injection refers to the **value of a node**. Injection paths therefore differ from node imports: injections always request a node value and cannot request a node set.

6 Expressions

Node values can also be defined indirectly using expressions. Expression values consist of multiple values combined with mathematical expressions wrapped in additional parentheses, while they can span on multiple lines. The code will be automatically wrapped into a single line based on the balance of the parentheses

```
result float = (
  12 + 3.43 - ( 6 * 7.89 ) / 1.2e3
)

# result = 15.39055
```

There are 4 different types of expressions in DIPL. Three of them (logical, numerical and textual, i.e. templates) are described below. Dimensional expressions, i.e. units, are described in a separate chapter.

[!NOTE] All but unary operators used in expressions must be separated from values with at least one empty space.

Possible input values used by expressions are summarized in the following table:

Input values

Schema	Name	Example
<bool>	boolean	true/false
<num>	numerical	24 cm
<ref>	reference	{?energy}
<expr>	expression	12 cm == {?width} && true

Operators summarized below are evaluated according to their priority from highest 1 to lowest 6, and can be nested accordingly.

6.1 Logical

Logical expressions are used to generate values for boolean nodes. The expressions can be composed of multiple nested logical operators that always return a true or false value.

An example of a logical expression is given below.

```
a bool = true
b float = 23.43 cm
c bool = (false || ( {?b} == 23.43 cm || ~{?a} )
          && {?a} || ndef( {?c} ))
# Node 'c' will be evaluated as 'true'
```

Expression operators evaluate only boolean values, either given directly or as a result of a nested expression.

Parentheses

Syntax	Priority	Description
(<expr>)	2	Parentheses operator evaluates expression A in a separate thread and returns its value
def(<ref>)	1	Definition operator returns true if node exists
ndef(<ref>)	1	Non-definition operator returns true if node does not exist

Logical operators

Syntax	Priority	Description
A \ \ B \ \ C	6	Logical OR operator returns true if at least one expression (A, B, C, ...) is true
...		
A && B && C ...	5	Logical AND operator returns true if all expressions (A, B, C, ...) are true

Numerical values with dimensions compared using comparison operations are automatically converted into same units. The result of such comparison is always a boolean value.

[!NOTE] At the moment, it is possible to compare only scalar numerical values. Support for array comparison is planned to be provided in next versions of DIPL.

Comparison operators

Syntax	Priority	Description
A == B	3	Equality operator returns true if A and B have same dimensions and numerical value up to EQUAL_PRECISION
A != B	3	Inequality operator returns true if A and B do not have same dimension or numerical value up to EQUAL_PRECISION

Syntax	Priority	Description
A >= B	3	Greater or equal operator returns true if A is greater or equal (up to EQUAL_PRECISION) than B
A <= B	3	Smaller or equal operator returns true if A is smaller or equal (up to EQUAL_PRECISION) than B
A > B	3	Greater than operator returns true if A is greater than B
A < B	3	Smaller than operator returns true if A is smaller than B

Single value operators

Syntax	Priority	Description
~<bool>	4	Negation operator returns true if value A is false

6.2 Numerical

Numerical expressions are used to generate values for numerical node types. If given, the expression result is automatically converted into node units. The target units should be placed after the expression parentheses end.

```
const
  c float = 299792458 m/s
energy float = (2 kg * pow( {?const.c}, 2 )) eV
```

Node 'energy' will be parsed as 1.79751 eV

[!NOTE] DIPL does not aim to substitute advanced numerical programming languages. Numerical expressions in DIPL are supposed to give a quick tool for generation of input values that can be easily derived from other parameters. Therefore, it implements only the most basic mathematical operations on scalar values. More advanced operations can be added in the future versions of DIPL.

Operators used in numerical expressions are summarized below:

Arithmetics

Syntax	Priority	Description
A + B	6	Addition of two values of a same dimension
A - B	6	Subtraction of two values of a same dimension
A * B	5	Multiplication of two values
A / B	5	Division of two values
A ** B	4	First value on the power of the second value
+B	3	Unary plus
-B	3	Unary minus

Parentheses operators evaluate expressions in a separate thread and return its final value. Most of the following operators require, that the final value has no dimensions.

Parentheses operators

Syntax	Priority	Description
(<expr>)	2	Standard parenthesis operator
exp(<expr>)	1	Returns exponential value of a dimensionless expression.
pow(<expr>, <expr>)	1	Returns first expression risen on a power of second dimensionless expression.
ln(<expr>)	1	Returns natural logarithmic value of a dimensionless expression.
log10(<expr>)	1	Returns common logarithmic value of a dimensionless expression.
sin(<expr>)	1	Returns sine value of a dimensionless expression.
cos(<expr>)	1	Returns cosine value of a dimensionless expression.
tan(<expr>)	1	Returns tangent value of a dimensionless expression.

Syntax	Priority	Description
<code>min(<expr>, <expr>)</code>	1	Returns minimum of the two expressions.
<code>max(<expr>, <expr>)</code>	1	Returns maximum of the two expressions.
<code>floor(<expr>)</code>	1	Returns the greatest integer less than or equal to the expression
<code>ceil(<expr>)</code>	1	Returns the smallest integer greater than or equal to the expression
<code>round(<expr>)</code>	1	Returns the closest integer or equal to the expression

[!NOTE] Unlike the outer parentheses that define an expression, the inner parentheses must contain at least one whitespace separating the enclosed content. For example, `exp(2 cm)` is invalid, whereas `exp( 2 cm )` is valid.

6.3 Templates

Templates, or formatted text values, are used to convert node values into a textual representation. They follow the same syntax as standard string values, but are prefixed with an `f` before the opening quotation mark. All value types can be embedded into the text using standard Python-style formatting notation.

```
id int = 345
name str = "Tina"
body
  weight float = 62.3 kg
  height float = 177 cm
married bool = true

person str = f"""
ID:      {{?id}:05d}
Name:    {{?name}}
Weight:  {{?body.weight}:.3e}
Height:  {{?body.height}:.2f}
Married: {{?married}}
\{not a reference}
"""

# Node 'person' will be parsed as:
#
# ID:      00345
# Name:    Tina
# Weight:  6.230e+01
# Height:  177.00
# Married: True
```

String expressions interpret starting double curly brackets always as a reference. Single curly brackets are interpreted as a plain test.

Basic syntax of parsing operators is described below:

Syntax	Description
<code>{{<ref>}}</code>	Default reference of a node value.
<code>{{<ref>}<slice>}}</code>	Default reference of a node value slice.
<code>{{<ref>}:<format>}}</code>	Formatted reference of a node value.
<code>{{<ref>}<slice>}:<format>}}</code>	Formatted reference of a node value slice.

Arrays can also be parsed using templates, however, without specifying a format. The format depends on the default Python string casting functions:

```
name str = "Will Smith"
widths float[2,3] = [[23.4,235.4,34],[1e10,2e23,5e20]]
```

```

preview str = f"""
Surname:  {{?name}[5:]}
Scalar:   {{?widths}[1,1]:.2e}
Array:
{{?widths}[:,1:]}
"""

# Node 'preview' will be parsed as:
#
# Surname:  Smith
# Scalar:   2.00e+23
# Array:
# [[2.354e+02 3.400e+01]
#  [2.000e+23 5.000e+20]]

```

7 Units

Units in DIPL follow a dedicated domain specific language PUEL designed for this purpose and defined in a separate specification. Consequently, every DIPL implementation **MUST** either provide a fully compliant units parser or utilize the reference units parser available in the SciNumTools v3 repository.

All values **MUST** be converted to a canonical unit representation before comparison or condition evaluation.

7.1 Standard Units

Each numerical value node has default units assigned at definition, or declaration. Subsequent modification without given units assume to be in default units. Values of modifications with different units (but same dimension) are converted to default units.

```

# definitions
age int = 30 a
height float = 185 cm
weight float = 80 kg

# modifications
age = 35
height = 190 cm
weight = 90000 g

# Modified values:
#
# age = 35 a
# height = 190 cm
# weight = 90 kg

```

7.2 Custom Units

Similarly as in case of references, it is also possible to define new units directly in the DIPL code. This can be achieved by a special node directive **\$unit**. Names of the custom units are automatically wrapped into square brackets. If the name of a custom unit is already used, the code will raise an error.

```

$unit mass = 30 AU
$unit length = 10 pc
$unit time = 1 Gy

velocity float = 2 [length]/[time]
density float = 34 [mass]/[length]3

```

Each DIPL implementation **MUST** support the definition and integration of custom units directly through the code interface.

8 Functions

DIPL allows functions provided by the host implementation to be invoked from within a DIPL document. Functions extend the built-in expression syntax with custom logic and allow parameter values or nodes to be generated dynamically.

Note: The input and output data of a user-defined function may differ depending on the host implementation and the particular use case. However, the function interface defined by DIPL remains consistent: the environment list is passed to the function in the defined form, and the function returns a value/node list in the defined form.

8.1 Function Calls

A function call consists of a function name followed by parentheses:

```
# standard node definition
side float = 5 cm

# value injection from functions
volume float = fn_volume() cm3
surface int = fn_surface() mm2
prime bool = is_prime()
value str = print_value()

# node import from functions
parameters fetch_params()
shape
    shape_settings()
```

The function name identifies a function provided by the interpreter implementation. Implementations may provide functions written in the host language, such as C++, Python, or another supported language.

8.2 Function Arguments

Every function invocation receives an implicit **data** argument containing a copy of the value nodes that have already been parsed at the point of invocation.

In the reference implementation, this argument is represented by a `dip::Environment` object.

Explicit arguments may additionally be supplied in the function call. These arguments are evaluated before the function is invoked and are passed to the function together with the implicit **data** argument.

```
scale float = 10 cm
result float = calculate({?scale}, 2)
```

The exact representation and calling convention of the implicit **data** argument are implementation-defined.

8.3 Function Results

A function can be used either as a **value injection** or as a **node import**.

When used as a value injection, the function **MUST** return a value compatible with the declared DIPL node type. In the reference implementation, this is represented by `dip::ValueNodeData`.

When used as a node import, the function **MUST** return a list of nodes. In the reference implementation, this is represented by `dip::ValueNode::ListType`.

The function result **MUST** therefore match the syntactic context in which the function is invoked. An incompatible result **MUST** cause evaluation to fail.

8.4 Evaluation and Errors

Function calls are evaluated during parsing, as specified by the implementation.

Evaluation **MUST** fail if:

- the function cannot be resolved;

- the function execution results in an error; or
- the returned value is incompatible with the expected DIPL type or result context.

Implementations SHOULD provide diagnostic information identifying the function and the cause of the failure.

Unless explicitly specified otherwise by the function or implementation, function evaluation SHOULD be deterministic.

9 Properties

Numerical codes usually require initial parameter values with a specific format. In this section, we summarize properties that can be used in DIPL to restrict node values and describe their format. Each node property directive is given on a new line immediately after node definitions, declarations or modification. All properties must have consistent indent, two whitespaces higher than their parent node.

9.1 Directives

9.1.1 Options

Used by: `int`, `float`, `str`

Initial code parameters often accept only a few discrete input values, also called options. These can be explicitly described during node definition or declaration.

The expected value of this clause is a list of values. It can be given explicitly or as a reference.

Node options can be specified for all data types except boolean. In case of boolean, the two options (`true` or `false`) are implicitly set. If the node has an other value than one of the options, DIPL will throw an error message.

Node definition and individual options can have different units, but they must have the same dimensionality. The final value of such modified node will be, however, converted into units specified in the definition.

```
animal str = "dog"
!options ["cat", "dog", "horse"]

energy float = 23 erg    # definition
!options [43, 89]        # option in same units
!options [23, 34] erg    # option in same units
!options [3e-7, 5e+3] J  # option in different unit of energy

energy = 3e-7 J          # modification
# energy = 3 erg         # final value
```

Multiple `!options` clauses are allowed for the same node (e.g. each with different units). They combine into a single array of options against which the node value is evaluated.

```
energy float = 23 J
!options [23, 45, 10, 234, 490, 1939] J
!options [34, 234] erg
# all options: [23, 45, 10, 234, 490, 1939, 3.4e-6, 2.34e-7] J
```

If individual options need some additional description, this can be included in a comment.

```
resolution int = 16
!options [
    16, # low
    32, # medium
    64  # high
]
```

9.1.2 Condition

Used by: `int`, `float`, `str`, `bool`

Numerical values can usually have values ranging in some intervals. To restrict node values to some particular interval, it is possible to set a logical condition using `!condition` directive and a logical expression. A given expression has to be evaluated as `true` after each definition or modification of a node.

In the example below, node `energy` can have values in a range of 23 and 26 erg. The actual value of node `energy` is matched using a special self-reference sign `{.}`.

```
energy float = 25 erg
!condition (23 < {.} && {.} < 26)
```

9.1.3 Format

Used by: `str`

In general, string values wrapped into quote marks can contain all characters and can be arbitrary long. This can be restricted by defining their `!format` using standard regular expressions.

In the following example, node 'name' can contain only small and capital letters:

```
name str = "Ferdinant"
!format "[a-zA-Z]+"
```

9.1.4 Constants

Used by: `int`, `float`, `str`, `bool`

Sometimes nodes have to stay constant and exclude all possible modifications. This can be achieved by a directive `!constant`.

Node `name` in the following example cannot be further modified.

```
name str = "John"
!constant
name = "Mary" # this modification will raise an error exception
```

9.1.5 Tags

Used by: `int`, `float`, `str`, `bool`

Tags provide a simple and effective mechanism for organizing and categorizing large sets of parameters. Any data type that supports tagging may define the dedicated `!tags` property, which accepts a list of tag values. In practice, it is recommended to use string values for tags to ensure consistency and interoperability.

Once assigned, tags enable nodes to be queried and filtered after parsing through the use of tag selectors. Every DIPL implementation SHOULD provide support for selecting parsed nodes based on their associated tags.

```
resolution int = 32
!tags ["box", "grid"]
```

9.1.6 Delimiter

Used by: `table`

When entering values using a `table` node, columns are separated by whitespace by default.

For some input formats, such as CSV or TSV, a different column delimiter is required. DIPL therefore allows additional options to be specified for `table` nodes to define the delimiter used when parsing the table.

This makes it possible, for example, to prepend a DIPL header to a standard CSV file and use it directly as a DIPL table.

```
output table = ""
snapshot int
time float s
intensity float W/m2
---
0, 0.234, 2.34
1, 1.355, 9.4
2, 2.535, 3.4
```

```
"""
    !delimiter ",,"
```

9.2 Metadata

Used by: `int`, `float`, `str`, `bool`

Metadata provide additional information about a parameter without affecting its value or interpretation. They are intended for documentation, provenance, licensing, and versioning purposes, and may be used by documentation generators, scientific workflows, and other tooling.

In particular, provenance metadata describe the origin of a parameter's value, such as the publication, dataset, or other source from which it was obtained. This allows scientific and engineering data to remain traceable and properly attributed.

Unlike comments, metadata are part of the parsed document and are preserved during processing.

The following metadata properties are currently defined:

Property	Description
?descr	Human-readable description of the parameter.
?authors	Author(s) of the referenced publication or data source.
?title	Title of the referenced publication, report, or dataset.
?journal	Journal, book, conference proceedings, or other publication.
?year	Year of publication or release.
?volume	Publication volume.
?issue	Publication issue.
?pages	Page range within the publication.
?doi	Digital Object Identifier (DOI).
?url	URL of the referenced resource.
?version	Version of the referenced document or dataset.
?created	Creation date of the referenced resource or metadata.
?modified	Last modification date of the referenced resource or metadata.
?license	License governing the referenced resource or dataset.

Example

```
thermal_conductivity float = 401 W/(m*K)
    ?descr "Thermal conductivity of high-purity copper measured at 293.15 K"
    ?authors "John A. Smith, Emily R. Johnson, Michael T. Brown"
    ?title "Thermal Conductivity of High-Purity Copper from 100 K to 500 K"
    ?journal "Journal of Materials Science"
    ?year 2024
    ?volume 59
    ?issue 8
    ?pages "4123-4137"
    ?doi "10.1007/s10853-024-12345-6"
    ?url "https://doi.org/10.1007/s10853-024-12345-6"
    ?version "Published Version"
    ?created "2024-03-18"
    ?modified "2024-04-02"
    ?license "CC BY 4.0"
```

10 Conditions

Conditions in DIPL provide a mechanism for controlling node parsing flow based on the evaluation of logical expressions. They allow values and parameters to be assigned dynamically depending on runtime state, enabling branching behavior within scripts and configurations. Each condition is evaluated in sequence, and only the first matching branch is executed.

DIPL supports the following set of condition clauses that define this branching structure. These clauses can be combined to express simple decisions as well as more complex, multi-branch logic. The structure is indentation-

sensitive, meaning that all statements belonging to a clause must be nested under it with a higher indentation level.

- `@if (<expression>)` is a starting clause of each condition and requires as an argument result of a logical expression. If the expression is evaluated as true, its child nodes will be taken for further parsing and the condition is terminated.
- `@elif (<expression>)` is a subsequent condition clause and requires as an argument result of a logical expression. If the expression is evaluated as true, its child nodes will be taken for further parsing and the condition is terminated. If the expression is false, its child nodes are skipped and subsequent clauses are evaluated. For testing purposes, one can also use direct `true` or `false` keywords instead of expressions.
- `@else` is a special, optional, clause that is at the very end of conditions and does not require any expression. Each condition can have only one `@else` clause. If all previous `@if`-s and `@elif`-s are evaluated as false, child nodes of `@else` are taken.
- `@end` is a terminating clause whose main purpose is to separate two consecutive conditions. If omitted, a single condition terminates after last `@if`, `@elif` or `@else`.

```
winner int = 1

@if ({?winner} == 1)           # first case
    name str = "John"
@elif ({?winner} == 2)        # second case
    name str = "Jenny"
@else                         # default case
    name str = "Jonas"
@end                           # end of first condition

@if ({?name} == "Jenny")
    toy str = "doll"
@else
    toy str = "robot"
```

Nested conditions are also possible, provided that each child condition has indent higher than the parent condition.

```
@if false
    flower str = "rose"
@else
    flower str = "dandelion"
    @if false
        color str = "red"
    @elif false
        color str = "blue"
    @else
        @if true
            leaves int = 234
        color str = "yellow"
    tree str = "maple"

# The nodes above will be parsed as:
#
# flower = "dandelion"
# leaves = 234
# color = "yellow"
# tree = "maple"
```

[!NOTE] Although nodes within a clause are indented more deeply than their surrounding siblings, this indentation does not affect the naming hierarchy. Internally, nodes inside a branching clause are assigned a temporary “clause parent” (e.g. `foo.@if1.bar`). Once the condition is resolved and the branch is applied, this intermediate scope is removed, and the node assumes its final name (e.g. `foo.bar`).

11 Schemas

Schemas provide a mechanism for defining reusable node structures.

A schema describes a set of nodes that may later be instantiated within groups, maps, or lists. By allowing common structures to be declared once and reused multiple times, schemas reduce duplication and promote consistency across related data.

Conceptually, a schema is similar to a type definition in a programming language. It defines the structure and properties that its instances shall contain, but does not itself create any nodes within the document hierarchy.

11.1 Declaration and Instantiation

Schemas are declared using the `$schema` directive. The schema name defines a reusable type. Nodes declared within the schema become members of that type and are inherited by all schema instances.

For example, the following schemas define common structures that can be combined to describe a car:

```
$schema car
  manufacturer str;
  weight float kg;
  max_speed float km/h;
```

```
$schema vehicle
  registration str;
  year int;
```

A schema is instantiated by appending one or more schema names after a colon (`:`) to a group, map item, or list item declaration. Multiple schema names are separated by commas. For example, the following creates two car instances, each of which also conforms to the `vehicle` schema:

```
car[civic] : car, vehicle
  manufacturer = "Honda"
  weight = 1350 kg
  max_speed = 200 km/h
  registration = "M-AB 1234"
  year = 2020
```

```
car[320i] : car, vehicle
  manufacturer = "BMW"
  weight = 1570 kg
  max_speed = 235 km/h
  registration = "M-CD 5678"
  year = 2022
```

Schemas are independent of container type and may therefore be instantiated within groups, maps, or lists.

For example:

```
$schema person
  name str;
  age int years;
```

As a group:

```
father : person
  name = "Peter"
  age = 45 years
```

As a map item:

```
people[john] : person
  name = "John"
  age = 32 years
```

As a list item:

```
people[] : person
  name = "Anna"
  age = 28 years
```

11.2 Collection schemas

Schemas can also be applied to entire `map` and `list` collections. In this case, the schema is specified once in the collection declaration and is automatically applied to all subsequent items.

```
$schema vehicle
  speed float = 212 kph
  weight float = 1320 kg

$schema road
  tires str = "Continental"

vehicles map : vehicle      # schema applies to all items
vehicles[car] : road        # schema applies to this specific item
vehicles[ship]
```

In this example, both `car` and `ship` inherit the nodes defined by the `vehicle` schema, because it is applied to the entire `vehicles` collection. The `car` item additionally inherits the nodes defined by the `road` schema, while `ship` does not.

11.3 Nested Schemas

Schemas may contain both value nodes and container nodes, including nested groups, maps, lists, and other schema instances.

```
$schema address
  street str;
  city str;
  country str;

$schema person
  name str;
  age int years;

  residence : address
```

Because schemas describe structure independently of container type, they provide a uniform mechanism for defining reusable data models throughout a DIPL document.

11.4 Rules

The following rules shall apply:

- Schema names shall be unique within a document.
- A schema shall be declared before it is instantiated.
- A schema declaration does not create any nodes in the document hierarchy.
- A schema may contain value nodes and container nodes.
- A schema may instantiate other schemas.
- Schemas may be instantiated within groups, map items, and list items.
- The structure defined by a schema is inherited by all of its instances.