

# **Tridash 0.10 Reference Manual**

# Contents

<b>1</b>	<b>Syntax</b>	<b>1</b>
1.1	Atom Nodes . . . . .	1
1.2	Functors . . . . .	2
1.3	Node Lists . . . . .	3
1.4	Literals . . . . .	3
1.4.1	Numbers . . . . .	3
1.4.2	Strings . . . . .	4
<b>2</b>	<b>Nodes</b>	<b>5</b>
2.1	Glossary . . . . .	5
2.2	Declaring Nodes . . . . .	5
2.3	Declaring Bindings . . . . .	6
2.4	Propagation of Changes . . . . .	7
2.5	Evaluation Strategy . . . . .	8
2.6	Contexts . . . . .	8
2.6.1	Two-Way Bindings . . . . .	9
2.6.2	Cyclic Bindings . . . . .	10
2.6.3	Literal Bindings . . . . .	10
2.6.4	Explicit Contexts . . . . .	11
2.7	Failures . . . . .	11
2.7.1	Conditional Bindings . . . . .	12
2.7.2	Explicit Failures and Failure Types . . . . .	13
2.7.3	Conditionally Active Bindings based on Failure Type . . . . .	13
2.8	Input Nodes . . . . .	13
2.9	Attributes . . . . .	14
2.10	Subnodes . . . . .	15
2.11	Node States . . . . .	15
<b>3</b>	<b>Meta-Nodes</b>	<b>17</b>
3.1	Defining Meta-Nodes . . . . .	18
3.1.1	Optional Arguments . . . . .	19
3.1.2	Rest Arguments . . . . .	19
3.1.3	Local Nodes . . . . .	20
3.1.4	Self Node . . . . .	21
3.1.5	Nested Meta-Nodes . . . . .	22
3.2	Recursive Meta-Nodes . . . . .	22
3.3	Outer Node References . . . . .	22

---

3.4	External Meta-Nodes	23
3.5	Higher-Order Meta-Nodes	24
3.6	Macro Nodes	24
3.6.1	Literal Symbols	25
3.6.2	Node References	25
3.7	Instances as Targets	26
3.8	Target Node Transforms	26
<b>4</b>	<b>Modules</b>	<b>27</b>
4.1	Creating Modules	27
4.2	Referencing Nodes in Different Modules	27
4.2.1	Module Pseudo-Nodes	27
4.2.2	Importing Nodes	29
4.2.3	Direct References	30
4.3	Operator Table	31
4.3.1	Registering Infix Operators	32
<b>5</b>	<b>Core Module</b>	<b>32</b>
5.1	Literals	32
5.1.1	Macro-Node: ' (x)	32
5.1.2	Macro-Node: c (x)	33
5.1.3	Macro-Node: & (node)	33
5.2	Bindings	33
5.2.1	Macro-Node: -> (source, target)	33
5.2.2	Macro-Node: <- (target, source)	33
5.2.3	Macro-Node: @ (node, context : ' (default))	34
5.2.4	Macro-Node: :: (node, state)	34
5.3	Meta-Node Definitions	34
5.3.1	Macro-Node: : (id (args...), body)	34
5.3.2	Macro-Node: .. (node)	35
5.4	Failures	35
5.4.1	Meta-Node: fail (: (type))	35
5.4.2	Meta-Node: fail-type (x)	35
5.4.3	Meta-Node: fails? (x)	35
5.4.4	Meta-Node: ? (x)	35
5.4.5	Meta-Node: fail-type? (x, type)	36
5.4.6	Meta-Node: !! (x)	36
5.4.7	Meta-Node: !- (test, value)	36
5.4.8	Macro-Node: ! (functor)	36

---

5.4.9	Meta-Node: <code>catch(try, catch, :(test))</code> . . . . .	36
5.5	Builtin Failure Types . . . . .	37
5.5.1	Failure Type Node: <code>No-Value</code> . . . . .	37
5.5.2	Failure Type Node: <code>Type-Error</code> . . . . .	37
5.5.3	Failure Type Node: <code>Index-Out-Bounds</code> . . . . .	37
5.5.4	Failure Type Node: <code>Invalid-Integer</code> . . . . .	37
5.5.5	Failure Type Node: <code>Invalid-Real</code> . . . . .	37
5.5.6	Failure Type Node: <code>Arity-Error</code> . . . . .	37
5.6	Arithmetic . . . . .	37
5.6.1	Meta-Node: <code>+(x, y)</code> . . . . .	37
5.6.2	Meta-Node: <code>-(x, :(y))</code> . . . . .	38
5.6.3	Meta-Node: <code>*(x, y)</code> . . . . .	38
5.6.4	Meta-Node: <code>/(x, y)</code> . . . . .	38
5.6.5	Meta-Node: <code>%(x, y)</code> . . . . .	38
5.7	Comparison . . . . .	39
5.7.1	Meta-Node: <code>&lt;(x, y)</code> . . . . .	39
5.7.2	Meta-Node: <code>&lt;=(x, y)</code> . . . . .	39
5.7.3	Meta-Node: <code>&gt;(x, y)</code> . . . . .	39
5.7.4	Meta-Node: <code>&gt;=(x, y)</code> . . . . .	39
5.7.5	Meta-Node: <code>=(a, b)</code> . . . . .	40
5.7.6	Meta-Node: <code>!=(a, b)</code> . . . . .	40
5.8	Logical Operators . . . . .	40
5.8.1	Meta-Node: <code>and(x, y)</code> . . . . .	40
5.8.2	Meta-Node: <code>or(x, y)</code> . . . . .	41
5.8.3	Meta-Node: <code>not(x)</code> . . . . .	41
5.9	Selection Operators . . . . .	42
5.9.1	Meta-Node: <code>if(condition, true-value, :(false-value))</code> . . . . .	42
5.9.2	Macro-Node: <code>case(..(clauses))</code> . . . . .	42
5.9.3	Node <code>True</code> . . . . .	42
5.9.4	Node <code>False</code> . . . . .	43
5.10	Types . . . . .	43
5.10.1	Meta-Node: <code>int(x)</code> . . . . .	43
5.10.2	Meta-Node: <code>real(x)</code> . . . . .	43
5.10.3	Meta-Node: <code>string(x)</code> . . . . .	43
5.10.4	Meta-Node: <code>to-int(x)</code> . . . . .	44
5.10.5	Meta-Node: <code>to-real(x)</code> . . . . .	44
5.10.6	Meta-Node: <code>to-string(x)</code> . . . . .	44
5.10.7	Meta-Node: <code>int?(x)</code> . . . . .	44
5.10.8	Meta-Node: <code>real?(x)</code> . . . . .	44

---

5.10.9	Meta-Node: <code>string?(x)</code>	45
5.10.10	Meta-Node: <code>symbol?</code>	45
5.10.11	Meta-Node: <code>char?</code>	45
5.10.12	Meta-Node: <code>inf?(x)</code>	45
5.10.13	Meta-Node: <code>NaN?(x)</code>	45
5.11	Lists	45
5.11.1	Meta-Node: <code>cons(head, tail)</code>	45
5.11.2	Meta-Node: <code>head(list)</code>	46
5.11.3	Meta-Node: <code>tail(list)</code>	46
5.11.4	Meta-Node: <code>cons?(thing)</code>	46
5.11.5	Node: <code>Empty</code>	46
5.11.6	Node: <code>Empty!</code>	46
5.11.7	Meta-Node: <code>list(..(xs))</code>	46
5.11.8	Meta-Node: <code>list*(..(xs))</code>	47
5.11.9	Meta-Node: <code>list!(..(xs))</code>	47
5.11.10	Meta-Node: <code>nth(list, n)</code>	47
5.11.11	Meta-Node: <code>append(list1, list2)</code>	47
5.11.12	Meta-Node: <code>foldl'(x, f, list)</code>	47
5.11.13	Meta-Node: <code>foldl(f, list)</code>	48
5.11.14	Meta-Node: <code>foldr(f, list, :(x))</code>	48
5.11.15	Meta-Node: <code>map(f, list)</code>	48
5.11.16	Meta-Node: <code>filter(f, list)</code>	48
5.11.17	Meta-Node: <code>every?(f, list)</code>	49
5.11.18	Meta-Node: <code>some?(f, list)</code>	49
5.11.19	Meta-Node: <code>not-any?(f, list)</code>	49
5.11.20	Meta-Node: <code>not-every?(f, list)</code>	49
5.12	Strings	49
5.12.1	Meta-Node: <code>string-at(string, index)</code>	49
5.12.2	Meta-Node: <code>string-concat(string, str1, str2)</code>	50
5.12.3	Meta-Node: <code>string-&gt;list(string)</code>	50
5.12.4	Meta-Node: <code>list-&gt;string(list)</code>	50
5.12.5	Meta-Node: <code>format(string, ..(args))</code>	50
5.13	Dictionaries	50
5.13.1	Meta-Node: <code>member(dict, key)</code>	50
5.14	Functions	51
5.14.1	Meta-Node: <code>apply(f, ..(xs))</code>	51
5.15	Introspection	51
5.15.1	Meta-Node: <code>node?(thing)</code>	51
5.15.2	Meta-Node: <code>find-node(node, :(module))</code>	51

---

5.15.3	Meta-Node: <code>get-attribute(node, attribute)</code> . . . . .	51
5.16	Pattern Matching . . . . .	52
5.16.1	Nested Patterns . . . . .	52
5.16.2	Constant Patterns . . . . .	53
5.16.3	Matchers . . . . .	53
5.17	Module: <code>core/patterns</code> . . . . .	54
5.17.1	Meta-Node: <code>Pattern(condition, :(binding))</code> . . . . .	54
5.17.2	Meta-Node: <code>get-matcher(node)</code> . . . . .	55
5.17.3	Meta-Node: <code>make-pattern(place, pattern)</code> . . . . .	55
5.17.4	Meta-Node: <code>combine-conditions(c1, c2)</code> . . . . .	55
5.17.5	Meta-Node: <code>conditionalize-bindings(condition, bindings)</code> . . . . .	55
5.17.6	Failure Type Node: <code>Match-Fail</code> . . . . .	56
5.17.7	Meta-Node: <code>fail-match(condition)</code> . . . . .	56
5.17.8	Meta-Node: <code>make-match-bind(src, target)</code> . . . . .	56
5.17.9	Meta-Node: <code>make-pattern-declarations(pattern)</code> . . . . .	56
5.18	Operator Table . . . . .	56
<b>6</b>	<b>Optimizations</b> . . . . .	<b>57</b>
6.1	Coalescing . . . . .	57
<b>7</b>	<b>Interfacing with Other Languages</b> . . . . .	<b>58</b>
7.1	Calling External Functions . . . . .	59
7.2	Accessing and Setting Node Values . . . . .	59
7.2.1	Runtime Module Objects . . . . .	59
7.2.2	Runtime Node Objects . . . . .	59
7.3	JavaScript Backend . . . . .	60
7.3.1	Value Types . . . . .	60
7.3.2	Failure Values . . . . .	60
7.3.3	Builtin Failure Types . . . . .	61
7.3.4	Lazy Evaluation . . . . .	61
7.3.5	Linking Meta-Nodes to External Functions . . . . .	62
7.3.6	Runtime Module Object . . . . .	63
7.4	WebAssembly Backend . . . . .	64
7.4.1	Memory . . . . .	64
7.4.2	Value Types . . . . .	64
7.4.3	Builtin Failure Types . . . . .	65
7.4.4	Marshaller . . . . .	65
7.4.5	Lazy Evaluation and Garbage Collection . . . . .	66
7.4.6	Linking Meta-Nodes to External Functions . . . . .	67
7.4.7	Runtime Module Object . . . . .	68

---

---

<b>8</b>	<b>Compiler Options</b>	<b>69</b>
8.1	Compilation Targets . . . . .	69
8.2	Output Options . . . . .	69
8.2.1	JavaScript Backend . . . . .	69
8.2.2	WebAssembly Backend . . . . .	70
<b>9</b>	<b>Index</b>	<b>72</b>

---

Complete reference manual for version 0.10 of the Tridash programming language.

# Syntax

Syntactically a Tridash program consists of a sequence of **node** declarations, each consisting of a single node expression. Node expressions are divided into three types: **atom** nodes, **functor** nodes and **literal** nodes. Declarations are separated by a semicolon ; character or a line break.

Anything occurring between a # character and the end of the line is treated as a comment and is discarded.

## Basic Grammar

```
<declaration> = <node>('; ' | <line break> | <end of file>)

<node> = <functor> | <identifier> | <literal>
<node> = '(' <node> ')'

<functor> = <prefix-functor> | <infix-functor>
<prefix-functor> = <node> '(' [<node> {',' <node>}*] ')'
<infix-functor> = <node> ' ' <identifier> ' ' <node>

<literal> = <number> | <string>
<number> = <integer> | <real>
```

## Grammar Notation

The grammar notation used in this manual adheres to the following conventions:

- Angle brackets < . . . > indicate named syntactical entities.
- = designates that the syntactical entity on the left is defined by the rules on the right. Multiple definitions of the same syntactical entity indicate disjunction of the rules in each definition.
- | indicates disjunction in a single definition.
- Literal tokens are enclosed in single quotes ' . . . '.
- . . indicates ranges, e.g. ' 0 ' . . ' 9 ' indicates the digit characters in the range 0 to 9: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Rules enclosed in square brackets [ . . . ] are optional.
- Multiple rules enclosed in braces { . . . } are grouped into a single rule.
- \* indicates zero or more repetitions of the preceding rule.
- + indicates one or more repetitions of the preceding rule.

### 1.1 Atom Nodes

An **atom** node consists of a single node identifier, referred to as a symbol. Node identifiers may consist of any sequence of characters excluding whitespace, line breaks, and the following special characters: (, ), {, }, ", , , . , ; , #. Node identifiers must consist of at least one non-digit character, otherwise they are interpreted as numbers.

The following are all examples of valid node identifiers:

- name
- full-name



- `node1`
- `1node`

The following are not valid node identifiers:

- `123`
- `a.b`
- `j#2` — Only the `j` is part of an identifier, the `#2` is a comment
- `1e7` — As the `e` indicates a real-number in scientific notation. See [Numbers](#).

## 1.2 Functors

A functor node is an expression consisting of an operator node applied to zero or more argument nodes. Syntactically the operator node is written first followed by the comma-separated list of argument nodes in parenthesis `( . . . )`.

### Examples

```
op(arg)
func(arg1, arg2)
m.fn()
```



#### Important

A line break occurring before the closing parenthesis `)` is not treated as a declaration separator. Instead the following line is treated as a continuation of the current line.

In the special case of an operator applied to two arguments, the operator may be placed in infix position, that is between the two argument nodes. The operator may only be an **atom** node and must be registered as an infix operator.



#### Important

Spaces between the infix operator and its operands are required, in order to distinguish the operator from the operands. This is due to there being few restrictions on the characters allowed in node identifiers.



#### Important

A line break occurring between the infix operator and its right argument is not treated as a declaration separator. However a line break between the left argument and infix operator is treated as terminating the declaration consisting of the left argument. The following line is then treated as a new separate declaration.

Functor expressions written in infix and prefix form are equivalent, thus the following infix functor expression:

```
a + b
```

is equivalent to the following prefix functor expression (*either expression may be written in source code*):

```
+(a, b)
```

Each node registered as an infix operator has, associated with it, a precedence and associativity. The precedence is a number that controls the priority with which the operator consumes its arguments. Operators with a higher precedence consume arguments before operators with a lower precedence. The associativity controls whether the operands are grouped starting from the left or right, in an expression containing multiple instances of the same infix operator.

```
a + b * c
```

The `*` operator has a greater precedence than the `+` operator, thus it consumes its arguments first, consuming the `b` and `c` arguments.

The `+` operator has a lower precedence thus it consumes its arguments after `*`. The arguments available to it are `a` and `*(b, c)`.

As a result the infix functor expression is parsed to the following:

```
+(a, *(b, c))
```

Use parenthesis to control which arguments are grouped with which operators. Thus for an infix expression to be parsed to the following, assuming the `*` operator has a greater precedence than `+`:

```
*(+(a, b), c)
```

`a + b` must be surrounded in parenthesis:

```
(a + b) * c
```

## 1.3 Node Lists

Multiple declarations can be syntactically grouped into a single node expression using the `{` and `}` delimiters. All declarations between the delimiters are processed and accumulated into a special *node list* syntactic entity. In most contexts the node list is treated as being equivalent to the last node expression before the closing brace `}`, however special operators and macros may process node lists in a different manner.

### Node Lists

```
<node> = <node list>
<node list> = '{' <declaration>* '}'
```



#### Warning

Each node list must be terminated by a closing brace `}` further on in the file otherwise a parse error is triggered.

---

## 1.4 Literals

Literal nodes include numbers and strings.

### 1.4.1 Numbers

There are two types of numbers: **integers** and real-valued numbers, referred to as **reals**, which are represented as floating-point numbers.

Integers consist of a sequence of digits in the range 0--9, optionally preceded by the integer's sign. A preceding `-` indicates a negative number. A preceding `+` indicates a positive integer, which is the default if the sign is omitted.

#### Integer Syntax

```
<integer> = ['+'|'-'] ('0'..'9')+
```

---

There are numerous syntaxes for real-valued numbers. The most basic is the decimal syntax which comprises an **integer** followed by the decimal dot `.` character and a sequence of digits in the range 0–9.

### Decimal Real Syntax

```
<real> = <decimal> | <magnitude-exponent>
```

```
<decimal> = <integer> '.' ('0'..'9')+
```

#### Note

The decimal `.` must be preceded and followed by at-least one digit character. Thus `.5` and `1.` are not valid **real** literals, `0.5` and `1.0` have to be written instead.

The exponent syntax allows a real-number to be specified in scientific notation as a magnitude  $m$  and exponent  $n$  pair  $m \times 10^n$ . The exponent syntax comprises a real in decimal syntax or an integer, followed by the character `e`, `f`, `d`, or `l` which indicates the precision of the real-number, followed by the exponent as an integer.

### Exponent Syntax

```
<magnitude-exponent> = (<decimal>|<integer>)[ 'e'|'f'|'d'|'l']<integer>
```

`e` and `f` indicate a single precision floating point number, `d` indicates double precision and `l` indicates long precision.

## 1.4.2 Strings

Literal strings consist of a sequence of characters enclosed in double quotes `" . . . "`.

### String Syntax

```
<string> = '"'<unicode char>*'"
```

where `<unicode char>` can be any Unicode character.

A literal `"` character can appear inside a string if it is preceded by the backslash escape character `\`.

### Example

```
"John said \"Hello\""
```

Certain escape sequence, consisting of a `\` followed by a character, are shorthands for special characters, allowing the character to appear in the parsed string without having to write the actual character in the string literal.

Table 1: Escape Sequences

Sequence	Character	ASCII Character Code (Hex)
<code>\n</code>	Line Feed (LF) / New Line	0A
<code>\r</code>	Carriage Return (CR)	0D
<code>\t</code>	Tab	09
<code>\u{&lt;code&gt;}</code>	Unicode Character	<code>

The `\u{<code>}` escape sequence is replaced with the Unicode character with code (in hexadecimal) `<code>`. There must be an opening brace `{` following `\u` otherwise the escape sequence is treated as an ordinary literal character escape, in which `\u` is replaced with `u`. Currently the closing brace `}` is optional, as only the characters up to the first character that is not a hexadecimal digit are considered part of the character code. However, it is good practice to insert the closing brace as it clearly delimits which characters are to be interpreted as the character code and which characters are literal characters.

---

**Tip**

The `\n`, `\r` and `\t` escape sequences can alternatively be written as `\u{A}`, `\u{D}` and `\u{9}` respectively.

---

**Caution**

In a future release, omitting either the opening or closing brace, in a Unicode escape sequence, may result in a parse error.

---

## 2 Nodes

Semantically a Tridash program is composed of a set of stateful components called nodes, each holding a particular value at a given moment in time.

Each node has a set of *dependency* nodes. A change in the value of at least one of the dependency nodes causes the node to recompute its own value. The node is said to be an *observer* of its dependency nodes, as it actively observes and responds to changes in their values. Similarly, each node has a set of *observer* nodes which it notifies whenever its own value changes.

This *dependency* — *observer* relation is referred to as a binding.

### 2.1 Glossary

**dependency**

A node *a* is said to be a *dependency* of node *b* if a change in the value of *a* triggers a change in the value of *b*.

**observer**

A node *a* is said to be an *observer* of node *b* if a change in the value of *b* triggers a change in the value of *a*.

**binding**

A *binding* is a relation between two nodes *a* and *b*, in which one node *a* is a *dependency* of the other *b*, and likewise the other node *b* is its *observer*.

**ancestor**

A node *a* is said to be an *ancestor* of a node *b* if *a* is a *dependency* of *b* or it is an *ancestor* of a *dependency* of *b*.

**successor**

A node *a* is said to be a *successor* of a node *b* if *a* is an *observer* of *b* or it is a successor of an *observer* of *b*.

### 2.2 Declaring Nodes

In the global scope, nodes are created on the first reference, that is when their identifier first appears in source code. This can either be a declaration consisting of the identifier itself or a functor node declaration of which the node is an argument.

**Caution**

The `self` identifier is reserved as an alias for the current *meta-node*, see [Section 3](#).

---

### Examples

---

```
# Results in the creation of node `name`
name

# Results in the creation of nodes `a` and `b`
a -> b

# Results in the creation of nodes `x`, `y` and `f(x, y)`
f(x, y)
```

This allows for a relaxed ordering of declarations. A node's definition need not be complete in order for it to be referenced as an argument.

Notice that in the last example, above, a node  $f(x, y)$  was created. This node corresponds to the functor node expression, thus functors, with the exception of a few special declarations, are nodes themselves. From now on a functor node expression refers only to functors in which the operator refers to a function. Functor expressions in which the operator does not refer to a function are referred to as special declarations.

---

#### Note

Operators which correspond to functions, such as  $f$  in the example above are referred to as meta-nodes and the functor expression as an instance of the meta-node. See [Section 3](#).

---



---

#### Note

Nodes are also created for functors written in infix form, e.g. for the functor  $a + b$ , the node  $+(a, b)$  is created.

---

A node corresponding to  $a \rightarrow b$  was not created. This is a bind declaration which is treated rather specially. The  $\rightarrow$  is not a meta-node, that is it does not compute a value, but is a special operator.

## 2.3 Declaring Bindings

A binding between two nodes is declared with the special bind operator  $\rightarrow$ .

```
a -> b
```

The above declares  $b$  an observer of  $a$  and likewise  $a$  a dependency of  $b$ . The result is that a change in the value of  $a$  will trigger a change in the value of  $b$ . This is an example of a simple binding, since the value of  $b$  is simply set to the value of  $a$ .

---

#### Note

In an explicit binding declaration the dependency, i.e. the left-hand side of the  $\rightarrow$  operator, is referred to as the *source* node and the observer, i.e. the right-hand side, is referred to as the *target* node.

---



---

#### Note

The bind operator is registered as an infix operator with precedence **10** and **right** associativity.

---



---

#### Tip

The  $\rightarrow$  operator is in the form of an arrow which indicates the direction of data-flow, from the node on the left to the node on the right.

---

Functional bindings involve a function of one or more argument nodes. Functional bindings are created implicitly in functor node expressions, with each argument node added as a dependency of the functor node. A change in the value of at least one argument node results in the value of the functor node being updated to the result of reevaluating the expression with the new values of the argument nodes.

#### Example

---

```
a + b
```

In the example, above, a functor node  $+(a, b)$  is created with the arguments  $a$  and  $b$  implicitly added as dependencies of  $+(a, b)$ . A change in either  $a$  or  $b$  will result in the value of  $+(a, b)$  being recomputed.

## 2.4 Propagation of Changes

As emphasized in the previous sections, changes in the value of a node are propagated to its observer nodes. The new value is propagated to each of the observers simultaneously. Each observer then proceeds to recompute its own value in parallel with the other observers.

### Example

```
a -> b
a + n -> c
a + 1 -> d
```

Node  $a$  has three observers:  $b$ ,  $+(a, n)$ ,  $+(a, 1)$ . Each of  $b$ ,  $+(a, n)$  and  $+(a, 1)$  receives the new value of  $a$  and immediately begins computing its new value. There is no strict sequential ordering of the updating of the values of the observer nodes. The following orderings are all possible:

- $b, +(a, n), +(a, 1)$
- $+(a, n), b, +(a, 1)$
- $+(a, 1), +(a, n), b$

Other orderings, including interleaved orderings, are also possible or it may be that the values of all the observers are updated in parallel.

It is important to note the semantics when nodes share a common observer and the change in value of each node is triggered by a common ancestor node. A node is said to be *dirtied* if either its value has changed, or at least one of its dependency nodes has been *dirtied*. If a node is *dirtied*, all its observers are *dirtied*, and likewise their observers are *dirtied* and so on. A node with multiple dependencies will only recompute its value when it receives a value change notification from each of its *dirtied* dependency nodes. Thus there is no intermediate value where the node's value is recomputed before all the dependency nodes have recomputed their values.



### Caution

This is only the case when the changes in each of the dependency nodes are triggered by a change in a common ancestor node. These semantics do not apply when the changes in the dependency nodes are not triggered by a change in a common ancestor node but by multiple simultaneous changes in an ancestor of each dependency, unless the changes in each ancestor are the setting of the initial values, in which case it is treated as though they have been triggered by a single common ancestor. See [Literal Bindings](#).

### Example

```
a -> b
a + 1 -> c

b + c -> out
```

In the example, above,  $a$  is a common ancestor of dependency nodes  $b$  and  $c$  of node  $+(b, c)$ . A change in  $a$  will *dirty* the following nodes:

- $a$

- `b`
- `+(a, 1)`
- `c`
- `+(b, c)`
- `out.`

The value of `+(b, c)` will only be recomputed when the values of both `b` and `c` have been recomputed.

If `b` and `c` did not have the common ancestor `a`, the value of `+(b, c)` would be computed on each change in the value of either `b` or `c`, regardless of whether the changes in values of `b` and `c` are triggered simultaneously or not.

## 2.5 Evaluation Strategy

The value of a node is not strictly evaluated. This means that a node's value is only evaluated if it is actually used. In most cases the result of this is that nodes are evaluated lazily, that is they are evaluated on their first use. However if it can be statically determined that a node's value will always be used it may be evaluated before its first use.

### Example: Lazy Evaluation in If Conditions

```
a - b -> d1
b - a -> d2

if(a > b, d1, d2)
```

In the example, above, `d1` is only evaluated if `a > b` evaluates to true. Likewise, `d2` is only evaluated if `a > b` evaluates to false. `a > b` is always evaluated as its value is always used. In this example, this only results in a performance optimization since the values of node's which are not used are not needlessly computed. However, if `d1` or `d2` were bound to a recursive meta-node call, *see Section 3*, an infinite loop of recursive calls would result had `d1` and `d2` not been evaluated lazily.

A node's value is evaluated at most once. Referencing the node's value in more than one location will not cause it to be evaluated more than once. This applies to functor nodes as well as atom nodes.

### Example: Multiple Usage of Nodes

```
# Node `f(x, y)` is used in 2 places however it will only be evaluated
# once.

f(x, y) + a -> node1
f(x, y) + b -> node2
```

## 2.6 Contexts

The function which computes a node's value is controlled by the node's context at that moment in time. The node context stores information about the function and which of the dependency nodes are operands to the function. Contexts are created whenever a binding between two nodes is established.

The most simple context function is the passthrough, created when a simple binding between two nodes is established. With this function, the node's value is simply set to the value of its dependency node.

### Passthrough Example

```
# `b` is set to the value of `a` whenever it changes

a -> b.
```

Contexts with more complex functions, of more than one operand, are created for each functor node expression. The created context has the operator as the context function and the arguments as the context operands.

### Functor Node Example

```
# A functor node `+(a, b)` is created with a `+` context.
# `a` and `b` are added to the operands of the `+` context.

a + b
```

A node can have more than one context. A context is *activated*, meaning its function is evaluated to compute the node's value, whenever the value of one of its operand nodes changes.

### Multiple Context Example

```
a -> x
b -> x
c -> x
```

When the value of `a` changes, the `a` context of `x` is activated and the value of `x` is set to the value of `a`. Similarly when `b` or `c`'s value changes, the `b` or `c` context is activated, respectively, and `x`'s value is set to the value of `b` or `c`, respectively.

#### Warning

It is an error for two or more contexts of a single node to be activated at the same time. This occurs when either both contexts have a common operand or an operand from one context has a common ancestor with an operand from the other context.

#### Example 1



```
# Node `a` is a dependency of `b`
# Node `a` is a dependency of `+(a, c)`
# Both `b` and `+(a, c)` are dependencies of `x`

a -> b
b -> x

a + c -> x
```

In the example, above, node `a` is a dependency node of `b` which is an operand of the `b` context of `x`. However, node `a` is also a dependency of node `+(a, c)` (`a + c`), which is an operand of the `+(a, c)` context of `x`. A change in the value of `a` would trigger a change in the value of both `b` and `+(a, c)` thus the value to which `b` should be set is ambiguous.

Structure checking is performed at compile-time, thus the above example, and all such scenarios, will result in a compilation error along the lines: `Semantic Error: Node x has multiple contexts activated by a single common ancestor.`

### 2.6.1 Two-Way Bindings

A dependency of a node may also be an observer of the same node. This allows for a two-way binding in which data may flow from either direction. In this case only the observer nodes which are not also operands of the node's current context are notified of a change in the node's value.

#### Example

```
# A two-way binding is established between `a` and `b`
a -> b
b -> a

a -> c
d -> a
```

In the above example, both `b` and `c`, which are observers of `a`, will be notified of a change in the value of `a` triggered by a change in the value of `d`. This will trigger a change in the value of `b` however `a` will not be notified of this change as the change was triggered by `a`, itself.



In the case of a change in the value of `a` triggered by a change in the value of `b`, only the observer `c` of `a` will be notified of the change.

### 2.6.2 Cyclic Bindings

Cyclic bindings are bindings between a set of nodes, such that there is a path, via bindings, from a node to itself, consisting of at least three nodes. The resulting value of the node contains a *cyclic reference* to itself.



#### Important

A cycle comprising just a pair of nodes is interpreted as a two-way binding rather than a cyclic binding.

---

#### Example

```
cons(a, y) -> x
cons(b, x) -> y
```

The cycle in this example involves the nodes:

- `x`
- `cons(b, x)`
- `y`
- `cons(a, y)`

In this example the value function of `x` is effectively:

```
cons(a, cons(b, x))
```

where `x`, is substituted with a reference to the result of the evaluation of the expression, itself. This is well-formed since the arguments to the `cons` meta-node are evaluated lazily.



#### Caution

In order for a cyclic binding to have a meaningful result, the cyclic reference must be evaluated lazily. The following will result in an infinite loop, as the cyclic reference to `i`, in `i + 1`, is strictly evaluated. Currently there is no compiler warning or error.

```
i + 1 -> i
```

---

### 2.6.3 Literal Bindings

A binding in which the dependency is a literal value, is interpreted as setting the initial value of a node. A special `init` context is created, which has no operands and has the literal value as its function.

Initial values are set on the launch of the application, and are treated as an ordinary value change to the initial value. The initial active context of the node is the `init` context. If a node is not given an initial value, its initial value is a failure value of type [No-Value](#), see [Section 2.7](#).

#### Examples

```
0 -> counter
"hello" -> message
10.5 -> threshold
```

---

**Important**

The setting of the initial values of each node, is treated as having been triggered by a single common ancestor node. See [Section 2.4](#) for the implications of this.

### 2.6.4 Explicit Contexts

The context to which a binding is established can be set explicitly with the special `/context` operator.

**Syntax**

```
/context (node, context-id)
```

The effect of this expression, when it appears as the target of a binding, is that the binding to `node` will be established in the context with identifier `context-id`. The identifier can be a symbol or a functor.

**Example**

```
# Context `my-context` of b has a passthrough value function to the
# value of the dependency `a`.

a -> /context (b, my-context)
```

When a `/context` declaration appears in source position it is equivalent to an ordinary reference to the `node`.

Multiple bindings to the same explicit context can be established. The function of the context then selects the value of the first dependency, ordered by the declaration order in the source file, which does not *fail* to evaluate to a value, see [Section 2.7](#).

**Example**

```
a -> /context (node, ctx)
b -> /context (node, ctx)
c -> /context (node, ctx)
```

`node` evaluates to:

- The value of `a` if `a` evaluates to a value.
- The value of `b` if `a` fails to evaluate to a value.
- The value of `c` if both `a` and `b` fail to evaluate to a value.

If `a`, `b` and `c` all fail to evaluate to a value, `node` evaluates to the failure value of `c`.

**Tip**

The `@` macro from the `core` module, which is a shorthand for the `/context` operator, is the preferred way of establishing bindings to explicit contexts in source code.

## 2.7 Failures

Failures are a special type of value which represents the absence of a value or the failure to compute a value. Failures can either be created by *conditional bindings*, in which the condition node evaluates to *false*, or by the `fail` meta-node, from the `builtin` module.

Functions which expect an argument `node` to evaluate to a value will fail if at least one argument fails. In formal terms, if the result of a function requires that the value of an argument, which fails to evaluate to a value, be evaluated, the entire function fails to evaluate to a value. The following are examples of functions which fail if at least one of argument fails: `+`, `-`, `*`, `/`.

If the result of a function is a dictionary, and a dictionary entry fails to evaluate to a value, it is only that dictionary entry that fails, the function still returns a dictionary.

### 2.7.1 Conditional Bindings

A binding declaration `a -> b` can, itself, be treated as a node, to which an explicit binding can be established with the binding node as the target.

```
c -> (a -> b)
```

The result of this declaration is that the binding `a -> b` is only active if the condition node `c` evaluates to *true*, the value of the builtin [True](#) node,. If `c` evaluates to *false*, the value of the builtin [False](#) node, `b` is not set to the value of `a` but is set to a failure value of type [No-Value](#).

A binding declaration, with a binding node as the target, changes the function of the context of the binding to return a failure value if the value of the condition node is *false*. The binding node `a -> b` (`->(a, b)` in prefix notation), is added as a dependency of `b` and as an operand of the context corresponding to the binding `a -> b`. The binding node is itself an observer of `c` with a simple passthrough function. This allows you to reference the *status* of the binding by referencing the binding node, `a -> b`.

#### Example: Simple Validation

```
# Validate that `i` has a value > 0
# Propagate value of `i` to `j`

i > 0 -> (i -> j)

# Perform some computation with `j` which is guaranteed to either be a
# numeric value greater than zero or a failure.
...
```

---

#### Tip

The bind `->` operator has *right* associativity, thus the parenthesis in `c -> (a -> b)` can be omitted: `c -> a -> b`.

---

Conditional bindings to an explicit context can also be established, *see* [Explicit Contexts](#). If a condition node evaluates to *false*, it is treated as though the corresponding dependency node has failed to evaluate to a value. The context's function then evaluates to the next dependency which does not fail to evaluate to a value. If all condition nodes evaluate to *false*, the node fails to evaluate to a value.

#### Example: Conditional Bindings and Explicit Contexts

```
cond1 -> (a -> /context(node, ctx))
cond2 -> (b -> /context(node, ctx))
c -> /context(node, ctx)
```

- If `cond1` evaluates to false, it is treated as though `a` has failed to evaluate to a value.
- If `cond2` evaluates to false, it is treated as though `b` has failed to evaluate to a value.

The net result is that `node` evaluates to:

- `a` if `cond1` evaluates to true.
  - `b` if `cond2` evaluates to true.
  - `c` if neither `cond1` not `cond2` evaluate to true, or both `a` and `b` fail to evaluate to a value.
-

### 2.7.2 Explicit Failures and Failure Types

Failure values can also be created explicitly with the `fail` meta-node, from the `core` module. This meta-node takes one optional argument: a value indicating the failure type. If the failure type is not provided, the failure returned does not have a type.

#### Example: Explicit Failure with Type

```
# Bind `b` to `a` if `c` is true
c -> (a -> /context(b, ctx))

# If `c` is false set `b` to an explicit failure
fail("my-type") -> /context(b, ctx)
```

The failure type of a *failure value* can be retrieved with the `fail-type` meta-node. This meta-node takes a single argument, which if it evaluates to a failure, returns the failure type associated with the failure. If the argument does not fail to evaluate to a value, or the failure has no type associated with it, `fail-type` returns a failure.

#### Example: Querying Failure Type

```
# Compare failure type of `b`, to "my-type" from example above

fail-type(c) = "my-type" -> c-fails?
```

The failure type is useful to identify the cause of a failure, since failures are used to represent many classes of errors, such as type errors, out of range errors, no value errors, as well as representing special classes of values.

### 2.7.3 Conditionally Active Bindings based on Failure Type

The special `/context` operator takes an optional third argument which is a test function that is evaluated prior to activating the binding after the previous binding fails. The test function is applied on a single argument, the failure type of the previous binding. If the function returns *true* the binding is activated otherwise this binding fails with the same failure type as the preceding binding.

#### Tip

The `@` macro, from the `core` module, contains a shorthand syntax for establishing a binding to an explicit context with a test function that compares the failure type to a given value.

## 2.8 Input Nodes

Input nodes are the nodes which receive the application input, which could be the value entered in a text field of the user interface (UI), data received from the network, etc. Input nodes do not have any dependencies and have a special `input` context, which does not have a value computation function. Instead the value of the node is meant to be set explicitly through some external event.

Input nodes have to be explicitly designated as such by setting the `input` attribute to *true*. *See Section 2.9 for more information about node attributes.*

#### Example: Setting Input Attribute

```
a -> b

# Designate `a` as an input node
/attribute(a, input, True)
```



#### Caution

A compilation error is triggered if a node has a dependency that is not reachable from any input node, however has at least one dependency that is reachable from an input node. The error is not signalled if all of the node's dependencies are unreachable from all the input nodes.

## 2.9 Attributes

Attributes are arbitrary key value pairs associated with a node, which control various compilation options. These are set using the special `/attribute` operator.

The first argument is the node of which to set the attribute, the second argument is the attribute key (not interpreted as a node) and the last argument is the value, which is interpreted as a literal value, not a node reference.

`/attribute` declarations may only appear at top-level and may not appear in binding declarations or as arguments in functor nodes.

### Attribute Declaration Syntax

```
/attribute(node, attribute, value)
```

---

#### Note

The `attribute` key need not be a string, it may simply be an identifier as it is not interpreted as a node.

---

---

#### Note

Attribute keys are case insensitive. Additionally a string attribute key and an equivalent identifier key both refer to the same attribute. Thus the following keys all refer to the same attribute: `key`, `Key`, `"key"`, `"KEY"`.

---



#### Important

The `value` is treated as a literal value, not a reference to the value of a node, since attributes do not form part of the runtime node's state.

---

The `input` attribute has already been introduced. The following is a list of some attributes and a summary of their effect:

#### **input**

When set to true, designates a node as an input node. *See Section 2.8.*

#### **coalescable**

When set to false, prevents the node from being coalesced into other nodes. *See Section 6.1.*

#### **removable**

When set to false, prevents the node from being removed.

#### **public-name**

The name with which the runtime node can be referenced from non-Tridash code.

#### **macro**

When set to true, indicates that a meta-node is a macro and should be invoked at compile-time. *See Section 3.6.*

#### **target-node**

Sets the name of a meta-node to use as the value function, in the contexts of the bindings of the meta-node instance (as the source node) to its arguments (as the target node). *See Section 3.7.*

#### **target-transform**

The name of a meta-node to invoke if the meta-node, of which the attribute is set, appears as the target of a binding. *See Section 3.7.*

### Examples

```
/attribute(a, input, True)
/attribute(a, public-name, "app-input")
```

---

## 2.10 Subnodes

Subnodes are nodes which reference a value, with a particular key, out of a dictionary of values stored in another node, referred to as the `parent` node.

Subnodes are referenced using the special `.` operator, which is also an infix operator. The `parent` node appears on the left-hand side and the key on the right-hand side. The key is treated as a literal identifier.

### Syntax

```
<parent node>.<key identifier>
```

---

#### Note

The `.` operator is lexically special in that spaces are not required to separate it from its operand.

---



---

#### Note

The `.` infix operator has precedence **1000** and **left** associativity.

---

### Example

```
string-concat (
  person.first-name, ❶
  person.last-name   ❷
) -> full-name
```

- ❶ References the `first-name` subnode of node `person`.
- ❷ References the `last-name` subnode of node `person`.

An implicit two-way binding is established between the subnode and parent node. The binding in the direction `parent -> subnode` has a value function which extracts the subnode key from the dictionary stored in `parent`. The binding in the reverse direction, `subnode -> parent`, has a function which creates a dictionary with an entry which has the subnode key as the key and the value of subnode as the value. This allows a dictionary to be created in the `parent` node by establishing an explicit binding with subnode as the target. Multiple such bindings, with different subnodes of `parent`, will result in a dictionary being created with an entry for each subnode.

### Example: Creating Dictionaries

```
"John" -> person.first-name
"Smith" -> person.last-name
```

The value of a subnode is only evaluated when the value of its dictionary entry is referenced. A subnode is not evaluated when only the value of its parent node, which evaluates to the dictionary, is referenced. *See Section 2.5*. If a subnode fails to evaluate to a value, it does not cause the parent node to fail to evaluate to value. The parent node evaluates to a dictionary however the dictionary entry, corresponding to the subnode, evaluates to a failure. *See Section 2.7*.

Accessing a non-existent entry, or accessing a subnode of a parent node which does not evaluate to a dictionary will result in a failure.

## 2.11 Node States

A binding can be established between the same node, as both the source and target of the binding, however in different *states*. The binding thus acts as a state transition function which computes the value of the node in its current *state* given its value in the previous *state*.

The special `/state` operator allows a binding to a particular state of a node to be established.

### Syntax

---

```
/state(node, state-identifier)
```

where `node` is the node and `state-identifier` is a symbol identifying the state.

When a `/state` expression appears as the target of a binding, the binding will only take effect when `node` switches to the state with identifier `state-identifier`. This allows `node` to appear in the source of the binding either directly or as part of a node expression, in which case the current value of `node` is referenced to compute its value in the new state.



#### Important

When a `/state` expression appears as the source of a binding, it reduces to a reference to the value of `node`. It is not a reference to the value of `node` in a particular state.

The `/state` operator may also take an additional third argument, in which case the arguments are interpreted as follows:

#### Syntax: With Explicit From and To States

```
/state(node, from-state, to-state)
```

This specifies that the binding, of which the `/state` expression is a target, is only active when the state of `node` switches from the state with identifier `from-state` to the state with identifier `to-state`.

The state of a node `n` is determined by the value of the special node `/state(n)`, to which bindings can be established. The value of `/state(n)` must be a symbol which is interpreted as a state identifier. The symbol may name a node state to which no bindings are established.



#### Important

Any value change of the node `/state(n)` is considered a change in the state of the node, even if the new state is identical to the previous state.

Multiple bindings, to the same node state, can be declared however bindings declared later in the source file take priority over bindings declared earlier in the file.

#### Example 1: Simple Counter

```
counter + 1 -> /state(counter, increment)

if (increment, '(increment), '(default)) -> /state(counter)
```

#### Tip

The `'` operator is a macro which returns its argument as a literal symbol, see [Literal Symbols](#).

The first declaration establishes the binding `counter + 1 -> counter`, which is active only when `counter` switches to the `increment` state. When `counter` switches to the `increment` state, its value is updated to its current value incremented by one.

The second declaration establishes a binding to the node `/state(counter)`, the value of which determines the state of `counter`. When the node `increment` changes to true, the value of `/state(counter)`, and thus the state of `counter`, is `increment` resulting in the value of `counter` being incremented by one. When `increment` is false, the state of `counter` is `default` and thus the binding established by the first declaration has no effect.

**Important**

A change in the value of `increment` from `true` to `true` will result in a change in the state of `counter` from `increment` to `increment`. Even though the new state is identical to the previous state, it is still considered a change to the `increment` state and thus the value of `counter` is incremented. To avoid this use the three argument form of the `/state` operator which specifies both a *from* and *to* state.

**Example 2: Simple Counter with Explicit From and To States**

```
counter + 1 -> /state(counter, default, increment)

if (increment, '(increment)', '(default)) -> /state(counter)
```

Using the three argument form of the `/state` operator, the first declaration establishes a binding, between `counter + 1` and `counter`, which is only active when the state of `counter` changes from `default` to `increment`. This differs from the previous example, in which the binding is active when the state of `counter` changes from any state to `increment`.

In this example, the value of `counter` is not incremented if the state of `counter` changes from `increment` to `increment`, as the binding `counter + 1 -> counter` is only active when the state changes from `default` to `increment`.

**Tip**

The `::` macro from the `core` module, which is a shorthand for the `/state` operator, is the preferred way of establishing bindings to explicit node states. The shorthand for the two argument form is `node :: state` and the shorthand for the three argument form is `node :: from => to`.

The first declarations of the previous examples can thus be rewritten as follows:

```
# First Declaration of Example 1
counter + 1 -> counter :: increment

# First Declaration of Example 2
counter + 1 -> counter :: default => increment
```

### 3 Meta-Nodes

A meta-node is a function, of one or more arguments, which returns a value. Meta-nodes are nodes, themselves, however without a runtime node object. For the most part you can treat meta-nodes as ordinary nodes, e.g. you can set meta-node attributes using the same `/attribute` declaration. Referencing the value of a meta-node references the meta-node function.

Meta-node identifiers reside in the same namespace as that of ordinary nodes, that is you cannot have both an ordinary node and meta-node with identifier `f`. If there is a meta-node `f`, the node expression `f` references the meta-node function.

**Note**

Functor nodes with the meta-node as the operator are referred to as instances of the meta-node.

**Tip**

Meta-nodes are referred to as meta-nodes, since they are nodes which describe how to compute the value of their instance nodes. Meta-nodes may also be macro-nodes which are evaluated at compile-time, with the result being interpreted as Tridash code.



### 3.1 Defining Meta-Nodes

Meta-nodes are defined using the special `:` definition operator which has the following syntax:

#### Definition Operator Syntax

```
name(arg1, arg2, ...) : {
    declarations*
}
```

The meta-node identifier, `name`, appears on the left-hand side of the `:` operator followed by the comma-separated list of arguments in parenthesis. Each item, at position  $n$ , of the argument list is the identifier of the local node to which the  $n^{\text{th}}$  argument is bound.



#### Caution

Identifiers beginning with `/`, followed by an alphanumeric character, are reserved for special operators. A meta-node cannot have the same identifier as a special operator. Currently no warning or compilation error is triggered, if the identifier is a reserved identifier but does not name an existing special operator, however that may change in a future release.

The body consists of a sequence of ordinary node declarations enclosed in braces `{ ... }`. The braces are simply a way of grouping multiple declarations into a single expression, *See Section 1.3*. If the body of the meta-node contains just a single expression, the braces may be omitted.

The meta-node function returns the value of the last node in the body.

#### Example

```
# Returns 1 + `n`

1+(n) : n + 1
```

#### Factorial Example

```
# Computes the factorial of `n`

factorial(n) : {
    case (
        n > 1 : n * factorial(n - 1),
        1
    )
}
```

The following example demonstrates that the body can contain any valid node declaration:

#### Fibonacci Example

```
fib(n) : {
    fib(n - 1) -> fib1
    fib(n - 2) -> fib2

    case (
        n <= 1 : 1,
        fib1 + fib2
    )
}
```



#### Important

Meta-nodes must be defined before they can occur as operators in functors.

**Important**

Meta-node bodies are only processed after all global (or the scope in which the meta-node declaration occurs) declarations in the same file have been processed. This allows a meta-node `g` to be used within the body of another meta-node `f` even if the definition of `g` appears after the definition of `f`. Effectively this allows for mutual recursion.

**3.1.1 Optional Arguments**

A node in the argument list, of a meta-node definition, may also be of the form `name : value`. This designates that the argument, which is bound to local node `name`, is optional. If it is omitted, in an instance of the meta-node, the local argument node is set to `value` instead.

**Note**

`value` is processed in the global scope as the meta-node definition is processed. Thus `value` cannot (as of yet), refer to the preceding argument nodes of the meta-node.

The `value` may be omitted, written in prefix form `:(name)`, in which case if the argument is omitted, the local argument node is set to a failure, *see Section 2.7*, of type [No-Value](#).

**Example**

```
# Increment `x` by 1 or given delta
inc(x, d : 1) : x + d

# Increment `a` by default delta 1
inc(a)

# Increment `b` by explicit delta 2
inc(b, 2)
```

**Important**

Optional arguments may only be followed by optional arguments or a rest argument. An optional argument may not be followed by a required argument.

**3.1.2 Rest Arguments**

The last node in the argument list, of a meta-node definition, may also be of the form `..(name)`. This designates that the local node `name` is bound to the list containing the remaining arguments, on which the meta-node is applied, after the last optional or required argument. This allows for a variable number of arguments.

**Example**

```
# Add `n` to each remaining argument
add-n(n, ..(xs)) : {
  inc(x) : x + n
  map(inc, xs)
}
```

See [Section 5.11](#) for the documentation of lists and the list processing functions.

### 3.1.3 Local Nodes

Nodes local to the meta-node's definition may only be referenced from within the definition itself even if they have the same identifiers as global nodes. Local nodes are created for each of the argument nodes.

A node reference, within the definition of a meta-node, primarily refers to the local node. If there is no local node with that identifier, it refers to the node in the enclosing scope. If the enclosing scope does not contain a node with that identifier, the scope's enclosing scope is searched until the global scope is reached. If the node is not found in any enclosing scope a compilation error is triggered.

Local nodes are created if they appear as the target of a binding, whether implicit or explicit. This is the means by which local nodes, storing intermediate results are created. Local nodes are also created for each top-level atom node declaration, *See Section 1.1*.

---

#### Note

The node creation rules inside meta-node definitions differ from the node creation rules at the global scope.

---



---

#### Tip

A global node, with the same identifier as a local node, can be referenced using the outer `..` operator.

---

#### Example: Local Nodes

```
a + b -> x
x + y -> n

addx(n) : {
  # `n` refers to the local argument node `n`, not the global `n`
  # `x` refers to the global node `x`
  n + x
}
```

#### Example: Meta-Nodes

```
1-(n) : n - 1

factorial(n) :
  case (
    # The `1-` refers to the global `1-` meta-node
    n > 1 : n * factorial(1-(n)),
    1
  )
```

#### Example: Local nodes storing intermediate results

```
x + 1 -> next

factorial(n) :

  # A local node `next` is created since it appears as the target of
  # a binding. `next` does not refer to the global node of the same
  # name.

  n - 1 -> next

  case (
    n > 1 : n * factorial(next),
    1
  )
```

---

**Example: Local Node Declarations**

```
cycle(a, b) : {
  x; y; # Declare local nodes `x` and `y` ❶

  cons(a, y) -> x
  cons(b, x) -> y

  x
}
```

- ❶ Top-level atom node declarations, resulting in the creation of local nodes `x` and `y`.

**3.1.4 Self Node**

The special `self` node is a local node which represents the meta-node's value. This node can be used to set the value, returned by the meta-node, using explicit bindings.

When an explicit binding to `self` is established, the meta-node no longer returns the value of the last node in its definition.

**Caution**

A meta-node may not have more than a single context, *see Section 2.6*, as it is ambiguous which context's value function to use as the meta-node function.

**Note**

In the absence of an explicit binding to `self`, the last node in the meta-node's definition is implicitly bound to `self`.

**Example**

```
factorial(n) : {
  n * factorial(n - 1) -> next
  case (n > 1 : next, 1) -> self ❶
}
```

- ❶ Explicit binding to `self`.

In the example, above, the value returned by the `factorial` meta-node is set by an explicit binding to the `self` node. The meta-node no longer evaluates to the value of the last node in the declaration list.

The `self` node is particularly useful for creating a dictionary of values to which the meta-node evaluates to, *see Section 2.10*:

**Example: Creating Dictionaries**

```
Person(first, last): {
  first -> self.first-name
  last -> self.last-name
}
```

### 3.1.5 Nested Meta-Nodes

The body of a meta-node can contain other meta-node definitions nested inside it. These meta-nodes are local to the body, and can only be used inside it, even if the same meta-node identifier appears in an expression outside the body. If a meta-node with the same identifier is already defined at global scope, the nested meta-node shadows it in the scope of the body. This means that references to the meta-node within the body refer to the nested meta-node and not the global node.

#### Example: Factorial with Nested Tail-Recursive Helper Meta-Node

```
factorial(n) : {
  # `iter` is local to `factorial`
  iter(n, acc) : {
    case (
      n > 1 : iter(n - 1, n * acc),
      acc
    )
  }

  iter(n, 1)
}
```

## 3.2 Recursive Meta-Nodes

Meta-nodes may be recursive and mutually recursive, i.e. when a meta-node *f* contains an instance of another meta-node *g* in its definition, and *g* contains an instance of *f* in its definition.

Each call to a meta-node consumes an amount of stack space. Further calls, within the meta-node, increase the amount of stack space if they are strictly evaluated. However, if a call to a meta-node is conditionally evaluated, i.e. lazily, it does not increase the amount of stack space used, since a *thunk* is returned, rather than the final result, thus freeing the amount of stack space used by the current call. See [Section 2.5](#).

The following are examples of meta-nodes in which one or more of the arguments are evaluated lazily:

- In the `if` meta-node, from the [core](#) module, the `if-true` and `if-false` arguments are evaluated lazily since only one of the arguments is actually evaluated, depending on the value of the first `test` argument. The `test` argument is evaluated strictly as its value is always required in order to compute the return value of the meta-node.

```
if(test, if-true, if-false)
```

- In the `and` and `or` meta-nodes, from the [core](#) module, the first argument is strictly evaluated however the second is lazily evaluated, as whether it is actually evaluated depends on the value of the first argument.

```
and(a, b)
or(a, b)
```

## 3.3 Outer Node References

The value of a node, declared in the global scope, can be referenced from within a meta-node, either directly by its identifier, as described in [Local Nodes](#), or with the outer node reference operator (`. .`). This is a special operator which takes a node identifier as an argument and searches for a node with that identifier, in each enclosing scope, starting from the scope in which the meta-node is defined. The first node found is referenced.

#### Note

It is not necessary for the node to have been declared prior to the meta-node definition, as meta-node definitions are only processed after all declarations in the source file have been processed. However, in general the node should be declared in the same source file.

### Example

```
n

# ..(n) references the global node `n`
addn(n) : n + ..(n)
```

Referenced outer nodes, whether implicitly or by the `..` operator, are treated as additional hidden arguments, that are added to the argument list of each instance of the meta-node. The result is that any change in the values of the referenced nodes, will trigger a value update in each instance of the meta-node.

The previous example can be thought of as:

```
# Not valid syntax.

# Illustrates that outer node references are equivalent to additional
# arguments.

addn(n, ..(n)) : n + ..(n)
```

Thus the value of `n` is appended to the argument list of all instances of `addn`, e.g. `addn(node)` becomes `addn(node, n)`.

Meta-nodes reference all outer nodes referenced by the meta-nodes which are used in their body. In the previous example, if a meta-node makes use of `addn`, it will also reference the node `n` declared in the global scope.



#### Important

Whilst the value of an outer-node can be referenced from within the body of a meta-node, bindings with the node as the target cannot be established, from within the body of the meta-node.

## 3.4 External Meta-Nodes

External meta-nodes are meta-nodes without a definition, which are used to invoke functions defined outside of Tridash code. The special `/external` declaration creates a meta-node without a definition.

### Syntax

```
/external(id, args...)
```

<code>id</code>	The meta-node identifier
<code>args</code>	The argument list

The argument list has to be provided in order for the arity of the meta-node to be known. The same rules apply for external meta-node argument lists as for ordinary meta-node argument lists. Symbols designate required arguments, arguments of the form `:(arg, value)` designate optional arguments and `..(rest)` designates a rest argument. The argument identifiers, however, do not name local nodes.

An external definition for the meta-node has to be provided, and linked with the generated code. In the JavaScript backend, instances of the meta-node are compiled to a call to the JavaScript function with the name given by the value of the `js-name` attribute. If the `js-name` attribute is not set, the result is a compilation error indicating that the JavaScript backend does not support the external meta-node.

### 3.5 Higher-Order Meta-Nodes

An atom node expression consisting of the meta-node itself references the meta-node's function as a value. This function can be passed to other meta-nodes as an argument, or bound to another node.

In a functor expression, in which the operator is not a meta-node but is an ordinary node, the function stored in the node's value is called. If the operator node does not evaluate to a function, the entire functor node evaluates to a failure of type [Type-Error](#), see [Section 2.7](#). If the function is invoked with more, or less, arguments than it expects, the functor node evaluates to a failure of type [Arity-Error](#).

#### Example: Binding Meta-Node to other Nodes

```
inc(x) : x + 1
```

```
inc -> f ❶  
f(a) -> x ❷
```

- ❶ Value function of `inc` meta-node bound to `f` node.
- ❷ Function stored in `f` meta-node applied on argument `a`.

See [Section 3.3](#) for an example in which a meta-node is passed as an argument to another meta-node.

The function of a meta-node which does not have optional arguments or outer nodes is effectively a constant, as is the case with the `inc` meta-node in the example above. If, however, the meta-node references outer nodes, a reference to the meta-node's function also references the values of the outer nodes. As such, if a node is bound to the meta-node's function, a binding between the outer nodes and the node is also established.

#### Example: Reference Meta-Node Function with Outer Nodes

```
# Increments `x` by the global `delta`  
inc(x) : x + delta  
  
inc -> f  
f(a) -> x
```

In the example, above, node `f` is bound to the value function of `inc`. However, since `inc` references the global `delta` node, a binding between `f` and `delta` is also established. The value function of `f` creates a function which invokes the `inc` with the value of `delta`. As a result, when the value of `delta` changes, the value of `f` is recomputed, and likewise the value of `f(a)` is recomputed.

The same semantics apply for optional arguments with default values which are not constant literals.

### 3.6 Macro Nodes

A macro-node is a meta-node which is evaluated at compile-time with the result interpreted as a Tridash node declaration.

A meta-node is marked as a macro-node by setting the `macro` attribute to `true`. Once set, the meta-node's function will be evaluated when each instance of the meta-node is processed. The arguments passed to the function are the raw argument node expressions of the functor node expression.

#### Tip

Attributes are set on meta-nodes in the same way as they are set for ordinary nodes. The `macro` attribute of a meta-node `f` is set to `true`, with the following declaration:

```
/attribute(f, macro, True)
```

Atom node expressions are represented by a special symbol type and functor node expressions are represented as a list with the operator in the first element of the list.

The return value of the meta-node function is processed as though it is a parsed node declaration appearing in source code.

### 3.6.1 Literal Symbols

The special `/quote` operator returns its argument, treated as a literal symbol rather than a node expression.

---

**Tip**

The `'` macro from the core module is the preferred shorthand for the `/quote` operator.

---

**Example**

```
# This is interpreted as the literal symbol `x` rather than the node
# with identifier `x`.

/quote(x)

# The following is a shorthand for the above
'(x)
```

These can be used inside macro nodes to insert literal node or operator names.

**Example: Definition of `'` macro**

```
'(thing) :
  list(/quote(/quote), thing)

/attribute(', macro, True)
```

### 3.6.2 Node References

Generally a macro-node expands to a declaration involving some other meta-node. The meta-node might not be located in the same module, *see Section 4*, as the module in which the macro-node instance occurs. Using the quote operator to generate a declaration involving the meta-node may result in a compilation error, if the meta-node is not present in the module in which the macro-node instance occurs, or may result in a node declaration involving an entirely different meta-node, if the module contains a node with the same identifier.

Node objects can be referenced directly with the node reference operator, `&`. When the declaration returned by a macro-node contains a raw node object, no node lookup is done and the raw node object is used as though it has been returned by node lookup. This is useful in macros as the node is looked up once in the module containing the macro-node's definition.

**Example: Definition of `<-` Macro**

```
<-(target, src) :
  list(&(->), src, target)

/attribute(<-, macro, True)
```

The `<-` macro function, in the example above, returns a functor expression where the operator is the node object `->`. When the functor expression is processed, the operator is taken to be the `->` node, rather than the node with identifier `->` in the module in which the instance is processed.

Any node can be referenced including ordinary nodes and macro-nodes. Special operators, however, cannot be referenced and have to be returned as quoted symbols instead. There is no issue with directly quoting the special operator's identifier, in expressions returned by macros, as there is for meta-nodes since the meaning of a special operator cannot be overridden and does not change with the module. Most of the *special operators* mentioned till this point, which are not an identifier prefixed with `/`, such as `->`, `:`, `&`, `.`, `..` are actually builtin macro nodes which expand to an internal special declaration, thus can be referenced with the `&` operator. Special operators beginning with `/`, such as `/attribute`, `/operator`, `/module` are actual special operators and cannot be referenced with `&`.

When a raw node referenced occurs in code which is intended to be evaluated at runtime, rather than during macro expansion, the runtime node object, of the node, is referenced. The nature of this object is dependent on the backend.

---



### 3.7 Instances as Targets

By default, a meta-node instance appearing as the target of a binding, that is on the right hand side of the `->` operator, will result in a compilation error. You may have noticed, however, that some meta-nodes in the `core` module, can also appear as targets of a binding, particularly `to-int`, `to-real` and `to-string`. This is achieved by setting the `target-node` attribute.

The `target-node` attribute stores the meta-node, which is applied on the value of the meta-node instance, in order to compute the value of the arguments. When the `target-node` attribute is set, a binding is established between the meta-node instance, as the dependency, and each argument node, as the observer. The function of the binding context is set to the meta-node stored in the `target-node` attribute.

---

#### Note

The `target-node` meta-node is looked up immediately when the attribute is set, and in the same module in which the `/attribute` declaration is processed.

---

As an example consider a meta-node `f` with the `target-node` attribute set to `g`. A declaration of the form:

```
x -> f(arg)
```

results in the following binding also being established, alongside the main binding of `arg -> f(arg)`:

```
g(f(arg)) -> arg
```

---

#### Note

The functor node `g(f(arg))` is not created, rather `f(arg)` is bound to `arg` directly and `g` is set as the value function.

---

This is useful for creating *invertable* meta-nodes where instead of computing a result given the values of the argument nodes, the values of the argument nodes can be computed given the result. This is achieved by binding to the meta-node instance, with the `target-node` attribute set to the *inverse* function.

The `to-int` meta-node from the `core` module has its `target-node` attribute set to `int`. Thus the binding `x -> to-int(y)`, will result in the value of `y` being set to the value `int(x)`, on changes in the value of `x`.



#### Caution

In order for the bindings to the argument nodes, to be established, the `/attribute` declaration, which sets the `target-node` attribute, must occur before between the definition of the meta-node and its first instance.

---

### 3.8 Target Node Transforms

The `target-node` attribute allows for a binding of a simple function to be established in the reverse direction, from the meta-node instance to its arguments. However, it lacks the functionality for setting a different function for each argument or generating more complex binding declarations.

The `target-transform` attribute allows a meta-node to be set as the function which is called whenever an instance of the meta-node appears as the target of a binding. The function is called with two arguments: the *source* node of the binding and the functor expression, which appears as the *target* of the binding. The function should return a declaration which is processed instead of the binding declaration. The result is processed as though it appears at top-level and unlike with a macro-node, the result is not substituted directly in the place of the meta-node instance.

---

#### Note

The *source* argument is not necessarily the actual source node declaration but is generally an atom node, with a randomly generated identifier, which should serve as the source node for the binding declarations generated by the `target-transform` node.

---

## 4 Modules

Modules provide a means of avoiding name collisions between nodes. A module is a namespace which contains all global nodes, including meta-nodes, created in it. A node with identifier  $x$  in a module  $m1$  is distinct from a node with the same identifier  $x$  in another module  $m2$ .

### 4.1 Creating Modules

Each new node, that is created as a result of processing a declaration in the source file, is added to the current module. Initially the current module is a nameless `init` module until it is changed explicitly.

The current module can be changed using the special `/module` operator, which takes the identifier of the module as its only argument. If there is no such module a new module is created.

#### Example

```
# Change to module with identifier `mod1`
/module(mod1)

# Nodes `a` and `b` added to `mod1`
a -> b

# Change to module with identifier `mod2`
/module(mod2)

# Nodes `a` and `b` added to `mod2`
# Distinct nodes from nodes `a` and `b` in `mod1`
a -> b
```

---

#### Note

Module identifiers reside in a different namespace from node identifiers, thus there is no risk of collision between a node and module with the same identifier, unless a pseudo-node for the module is added to the module containing the node.

---

---

#### Note

Modules reside in a single global, flat namespace. Hierarchical relations between modules have to be *faked* with a separator such as `/`, e.g. `module/submodule`.

---

### 4.2 Referencing Nodes in Different Modules

There are two ways to reference a node in another module, different from the current module. One way is to create a *pseudo-node* for the module in the current module. Nodes in the module can then be referenced as subnodes of the module's *pseudo-node*.

#### 4.2.1 Module Pseudo-Nodes

The special `/use` operator creates pseudo-nodes for the modules passed as arguments. The pseudo-nodes are created with the same identifiers as the modules.

---

#### Note

Module pseudo-nodes are referred to as such, since syntactically they are the same as any other node, however the value of a module pseudo-node cannot be referenced nor can bindings involving it be established.

---

#### Syntax

---

```
/use(mod1, mod2, ...)
```

**Caution**

Creating a pseudo node in a module, which contains a node with same identifier as the pseudo-node, results in a compilation error.

Nodes from the *used* modules can then be referenced as subnodes of the module pseudo nodes.

**Example**

```
/module(mod1)

a -> b

/module(mod2)
/use(mod1)

# Reference node `b` from module `mod1`
mod1.b -> b
x -> mod1.b
```

Meta-nodes from a different module can be also referenced as subnodes of the module pseudo-node.

**Example**

```
/module(mod1)

add(x, y) : x + y

/module(mod2)
/use(mod1)

# Use the `add` meta-node from module `mod1`
mod1.add(a, b) -> c
```

**Tip**

Nodes referenced from other modules, can appear both as dependencies or observers of bindings.

**Important**

Referencing a subnode of a module pseudo-node does not result in the automatic creation of a node in that module. Referencing a node that does not exist in the module results in a compilation error.

A pseudo-node with a different identifier, from the identifier of the module, can be created using the special `/use-as` operator. This is useful for when the module identifier is too long to type out repeatedly, or there is already a node, in the current module, with the same identifier.

the `/use-as` operator takes two arguments, the identifier of the module and the name of the pseudo-node to create in the current module:

```
/use-as(module-name, pseudo-node-name)
```

The above examples can be rewritten using `/use-as` declarations:

**Example**

```
/module(mod1)

a -> b

/module(mod2)
/use-as(mod1, m1)

# Reference node `b` from module `mod1`
m1.b -> b
x -> m1.b
```

### Example

```
/module(mod1)

add(x, y) : x + y

/module(mod2)
/use-as(mod1, m1)

# Use the `add` meta-node from module `mod1`
m1.add(a, b) -> c
```

## 4.2.2 Importing Nodes

The second approach to referencing a node, residing in another module, is to add it directly to the current module. With this approach there is no need to reference the node as a subnode of a module pseudo-node. This is referred to as *importing* the node and is achieved using the `/import` operator.

The `/import` operator adds the identifiers of nodes, residing in another module, to the current module. The result is that the node can be referenced directly by its identifier, as though it were declared in the current module.

The `/import` operator has two forms:

- A short form that *imports* all the nodes *exported* from another module. Takes the module identifier as its only argument.
- A long form that can be used to *import* specific nodes. The first argument is the module identifier, the following arguments are the identifiers of the nodes to *import*

### Syntax

```
# Short form: Import all nodes exported from `module`
/import(module)

# Long form: Import only the nodes listed in the arguments after the
# module identifier.
/import(module, node1, node2, ...)
```

### Example: Long form

```
/module(mod1)

a -> b

/module(mod2)

# Import node `b` from `mod1`
/import(mod1, b)

# Node `b` is the same `b` as in `mod1`
b -> a
x -> b
```

The short form only imports those nodes which are explicitly *exported* from the module. Nodes are explicitly exported from the current module with the `/export` operator which simply takes the identifiers of the nodes to *export* as arguments.

### Syntax

```
/export(node1, node2, ...)
```

Each `/export` declaration adds (does not replace) nodes, that are in the arguments, to the exported nodes of the current module.

### Example: `/export` and short form `/import`

```
/module(mod1)

a -> b

# Export node `b`
/export(b)

/module(mod2)
# Import all nodes exported from `mod1`
/import(mod1)

# Node `b` is the same `b` as in `mod1`
b -> a
x -> b
```

### Meta-node Example: `/export` and short form `/import`

```
/module(mod1)

add(x, y) : x + y

/export(add)

/module(mod2)
/import(mod1)

# Use the `add` meta-node from module `mod1`
add(a, b) -> c
```

A side effect of `/import` is that if the identifier of an imported node, whether imported by the long or short form of `/import`, is registered as an infix operator in the module, from which it is being imported, its entry in the module's operator table is copied over to the current module. This allows the operator to be placed in infix position, in the current module.

---

#### Note

All nodes in the `builtin` module, which contains the special operators mentioned so far (`->`, `:`, `...`, `&`, `.`), are automatically imported into the `init` module.

---

## 4.2.3 Direct References

It may be necessary to reference a node in another module without creating a pseudo-node, for its module, and without importing it in the current module. The special `/in` operator directly references a node in another module by the module and node identifiers.

### Syntax

```
/in(module-id, node-id)
```

The first argument is the identifier of the module containing the node, and the second argument is the identifier of the node.

---

**Important**

`module-id` is the identifier of the module itself, and not the identifier of a module pseudo-node.

**Example: /in Operator**

```
/module(mod1)

a -> b

/module(mod2)

# Reference node `b` in `mod1` directly
/in(mod1, b) -> a
x -> /in(mod1, b)
```

**Example: /in Operator**

```
/module(mod1)

add(x, y) : x + y

/module(mod2)

# Use the `add` meta-node from module `mod1`
(/in(mod1, add))(a, b) -> c
```

**4.3 Operator Table**

Each module may register a number of node identifiers as infix operators. This means that those identifiers may appear in infix position in declarations parsed while the module is the current module. The module's *operator table* stores the identifier, precedence and associativity of each infix operator. *See Section 1.2 for more information about infix operators, operator precedence and associativity.*

Initially the operator table of each module contains a single entry which is the special entry for function application. The precedence of function application controls whether an expression is treated as the operator of a functor or the operand of an infix expression.

**Note**

The precedence of function application is set to 900. This value cannot be changed.

**Example: Precedence of Function Application**

```
# If `.` has a higher precedence than function application, the
# following is parsed to: ((. m1 add) a b)

# If `.` has a lower precedence than function application, the
# following is parsed to: (. m1 (add a b))

m1.add(a, b)
```

### 4.3.1 Registering Infix Operators

New infix operators can be registered using the special `/operator` declaration. This declaration modifies the operator table of the current module.

#### Syntax

```
/operator(identifier, precedence [, (left | right)])
```

The first argument is the node identifier to register as an infix operator. The second argument is the precedence as an integer value. The third argument specifies the associativity. This is either the identifier `left` or `right` for *left* or *right* associativity. If the third argument is omitted, *left* associativity is assumed.

---

#### Note

The identifier does not have to name a node or meta-node that exists at the time the `/operator` declaration is processed. The table only stores the identifier for syntactic transformations, no information about the actual node is stored.

---

An `/operator` declaration adds an entry to the current module's operator table if it does not already contain an entry for the identifier. If the table already contains an entry for the identifier, the *precedence* and *associativity* values of the existing entry are replaced with those given in the arguments to the `/operator` declaration.

---

#### Note

The *precedence* and *associativity* of all operators can be changed, with the exception of *function application*.

---

When a node is imported into the current module and there is an entry, for the node's identifier, in the operator table of the module, from which the node is being imported, the entry is copied over into the current module's operator table, replacing any existing entries for the identifier.

#### Example: + infix operator from core module

```
# Register '+' as infix operator
/operator(+, 100, left)

# Now '+' can appear in infix position
a + b

# It can also still appear in prefix position
+(a, b)
```

## 5 Core Module

The `core` module provides the language primitives and the standard library.

### 5.1 Literals

#### 5.1.1 Macro-Node: ' (x)

Interprets `x` as a literal symbol rather than a node declaration. See [Literal Symbols](#).

The same symbol object is always returned for a given symbol name.

<code>x</code>	An atom node expression.
----------------	--------------------------

## Pattern Matching

Matches the literal symbol `x`. See [Pattern Matching](#).

### 5.1.2 Macro-Node: `c (x)`

Returns `x` interpreted as a literal character.

- If `x` is a symbol of one character, the character is returned.
- If `x` is a string, the first character in the string is returned.
- If `x` is an integer in the range 0 — 9, the character corresponding to the digit is returned.

`x`                      The value to convert to a character.

## Pattern Matching

Matches the character literal produced by the given argument. See [Pattern Matching](#).

### 5.1.3 Macro-Node: `& (node)`

Returns the raw node object corresponding to the node with identifier `node`. See [Node References](#).

`node`                      The node identifier, which can be any valid node expression. The expression is processed in the module in which the macro node instance occurs.

## 5.2 Bindings

### 5.2.1 Macro-Node: `-> (source, target)`

Establishes a binding between node `source` and node `target`.

`source`                      The source node.

`target`                      The target node.

---

#### Note

Registered as an infix operator with precedence **10** and **right** associativity.

---

### 5.2.2 Macro-Node: `<- (target, source)`

Establishes a binding between node `source` and node `target`.

Same as `->` however the argument order is reversed with the first argument being the target node and the second argument being the source node.

`target`                      The target node.

---



---

<code>source</code>	The source node.
---------------------	------------------

---

**Note**

Registered as an infix operator with precedence **10** and **left** associativity.

---

**5.2.3 Macro-Node: @ (node, context : ' (default))**

Indicates an explicit context to which bindings, involving `node` as the *target*, should be established.

When the `@` expression appears as the target of a binding, the binding to `node` is established in the context with identifier `context`, *see Section 2.6*. The context identifier may be omitted in which case the identifier `default` is assumed.

`context` may also be a functor of the form `when (context-id, type)`, where `context-id` is the context identifier and `type` is a node, of which the value is interpreted as a failure type. In this case the binding will only be activated if the failure type, of the previous binding in the context, is equal to `type` by `=`. *See [Conditionally Active Bindings based on Failure Type](#)*. If `context-id` is omitted, that is `context` is of the form `when (type)`, the identifier `default` is assumed.

<code>node</code>	The node.
<code>context</code>	The context identifier, or <code>when</code> expression, <i>see above</i> .

---

**Note**

Registered as an infix operator with precedence **800** and **left** associativity. The `when` symbol is also registered as an infix operator with precedence **850** and **left** associativity.

---

**5.2.4 Macro-Node: :: (node, state)**

When appearing as the target of a binding, a binding is established to `node` which is only active when `node` is in the state with identifier `state`.

`state` may also be a functor of the form `from => state`, in which case the binding is only active when the state of `node` changes from the state with identifier `from` to the state with identifier `to`.

*See Section 2.11 for more information.*

<code>node</code>	The node.
<code>state</code>	The state identifier or <code>=&gt;</code> expression, <i>see above</i> .

---

**Note**

Registered as an infix operator with precedence **700** and **left** associativity. The `=>` symbol is also registered as an infix operator with precedence **750** and **left** associativity.

---

**5.3 Meta-Node Definitions****5.3.1 Macro-Node: : (id (args...), body)**

Defines a meta-node with identifier `id`, argument list `args` and definition `body`. *See Section 3*.

---

---

<code>id</code>	Meta-node identifier.
<code>args</code>	Comma-separated list of node identifiers, to which the meta-node arguments are bound.
<code>body</code>	Node expression making up the definition of the meta-node. If the definition consists of more than a single expression, it should be enclosed in braces, <i>see</i> <a href="#">Node Lists</a> .

---

**Note**

Registered as an infix operator with precedence **5** and **right** associativity.

---

**5.3.2 Macro-Node: `.. (node)`**

Explicitly references a node defined in the enclosing scope of the meta-node. *See* [Section 3.3](#).

<code>node</code>	The node identifier, which can be any valid node expression.
-------------------	--

**5.4 Failures****5.4.1 Meta-Node: `fail (: (type))`**

Returns a failure with a given failure type.

**`type` (*Optional*)**

The failure type. If not provided the failure returned has no type.

**5.4.2 Meta-Node: `fail-type (x)`**

Returns the failure type of `x`.

Returns a failure if `x` does not evaluate to a failure or evaluates to a failure with no type.

<code>x</code>	The value of which to return the failure type.
----------------	--

**5.4.3 Meta-Node: `fails? (x)`**

Returns true if `x` fails to evaluate to a value.

<code>x</code>	The value to test for failure.
----------------	--------------------------------

**Pattern Matching**

Matches if the source node evaluates to a failure. If the argument `x` is provided matches only failures of type `x` otherwise matches any failure. *See* [Pattern Matching](#).

**5.4.4 Meta-Node: `? (x)`**

Returns true if `x` evaluates to a value, false if `x` fails to evaluate to a value.

---

`x`                      The value to test for failure.

#### 5.4.5 Meta-Node: `fail-type?(x, type)`

Tests for failure with a given type.

Returns true if `x` fails with failure type equal to `type`, by `=`. Returns false if the failure type of `x` is not equal to `type` or `x` does not fail.

`x`                      The value to check.

`type`                  The failure type.

#### 5.4.6 Meta-Node: `!!(x)`

Evaluates to true if `x` evaluates to a value. If `x` evaluates to a failure, evaluates to the failure value.

`x`                      The value to test for failure.

#### 5.4.7 Meta-Node: `!-(test, value)`

Returns `value` if `test` does not fail. If `test` fails, the failure is returned.

`test`                  The value which is checked for failure.

`value`                The value which should be returned if `test` does not fail.

#### 5.4.8 Macro-Node: `!(functor)`

Tests that each argument of a functor expression does not fail, before evaluating the expression.

If at least one argument fails, then the entire functor node fails.

`functor`              The functor expression.

#### 5.4.9 Meta-Node: `catch(try, catch, :(test))`

Returns the value of `try` if it does not evaluate to a failure. If `try` evaluates to a failure returns the value of `catch`.

**`try`**

The value, which is returned if it does not evaluate to a failure.

**`catch`**

The value, which is returned when `try` evaluates to a failure.

**`test`**

An optional function, which is applied on the failure type of `try`. If the function returns *true*, the value of `catch` is returned otherwise the value of `try` is returned.

## 5.5 Builtin Failure Types

### 5.5.1 Failure Type Node: No-Value

Failure type representing the absense of a value.

Optional meta-nodes arguments, for which no value is provided, are bound to a failure of this type.

Node `No-Value!` is bound to a failure of this type.

### 5.5.2 Failure Type Node: Type-Error

A failure of this type is returned when an argument to a meta-node is not of the expected type.

Node `Type-Error!` is bound to a failure of this type.

### 5.5.3 Failure Type Node: Index-Out-Bounds

A failure of this type is returned when attempting to access an element at an index that is outside the bounds of the list or string.

Node `Index-Out-Bounds!` is bound to a failure of this type.

### 5.5.4 Failure Type Node: Invalid-Integer

A failure of this type is returned by `int` when a string, from which an integer cannot be parsed, is provided as an argument.

Node `Invalid-Integer!` is bound to a failure of this type.

### 5.5.5 Failure Type Node: Invalid-Real

A failure of this type is returned by `real` when a string, from which a real number cannot be parsed, is provided as an argument.

Node `Invalid-Real!` is bound to a failure of this type.

### 5.5.6 Failure Type Node: Arity-Error

A failure of this type is returned when a meta-node is invoked indirectly, by a meta-node reference *see Section 3.5*, with an incorrect number of arguments.

Node `Arity-Error!` is bound to a failure of this type.

## 5.6 Arithmetic

### 5.6.1 Meta-Node: + (x, y)

Computes the sum of `x` and `y`.

`x`                      A number.

`y`                      A number.

---

#### Note

Registered as an infix operator with precedence **100** and **left** associativity.

---

**5.6.2 Meta-Node:  $- (x, : (y))$** 

Computes the difference of  $x$  and  $y$ .

If  $y$  is not provided, returns the negation of  $x$ , i.e.  $x$  multiplied by  $-1$ .

**$x$**

A number.

**$y$  (Optional)**

A number.

**Note**

Registered as an infix operator with precedence **100** and **left** associativity.

**5.6.3 Meta-Node:  $* (x, y)$** 

Computes the product of  $x$  and  $y$ .

$x$

A number.

$y$

A number.

**Note**

Registered as an infix operator with precedence **200** and **left** associativity.

**5.6.4 Meta-Node:  $/ (x, y)$** 

Computes the quotient of  $x$  and  $y$ .

$x$

A number.

$y$

A number.

**Note**

Registered as an infix operator with precedence **200** and **left** associativity.

**5.6.5 Meta-Node:  $\% (x, y)$** 

Computes the remainder of the division of  $x$  by  $y$ .

$x$

A number.

$y$

A number.

**Note**

Registered as an infix operator with precedence **200** and **left** associativity.

## 5.7 Comparison

### 5.7.1 Meta-Node: < (**x**, **y**)

Returns true if **x** is less than **y**.

**x**                      A number.

**y**                      A number.

---

**Note**

Registered as an infix operator with precedence **50** and **left** associativity.

---

### 5.7.2 Meta-Node: <= (**x**, **y**)

Returns true if **x** is less than or equal to **y**.

**x**                      A number.

**y**                      A number.

---

**Note**

Registered as an infix operator with precedence **50** and **left** associativity.

---

### 5.7.3 Meta-Node: > (**x**, **y**)

Returns true if **x** is greater than **y**.

**x**                      A number.

**y**                      A number.

---

**Note**

Registered as an infix operator with precedence **50** and **left** associativity.

---

### 5.7.4 Meta-Node: >= (**x**, **y**)

Returns true if **x** is greater than or equal to **y**.

**x**                      A number.

**y**                      A number.

---

---

**Note**

Registered as an infix operator with precedence **50** and **left** associativity.

---

**5.7.5 Meta-Node: = (a, b)**

Returns true if a is equal to b.

- *Numbers* are equal if they represent the same numeric value.
- *Characters* are equal if they represent the same character.
- *Strings* are equal if they have the same contents.
- Otherwise a and b are equal if they evaluate to the same object.

x                      A value.

y                      A value.

---

**Note**

Registered as an infix operator with precedence **50** and **left** associativity.

---

**5.7.6 Meta-Node: != (a, b)**

Returns true if a is not equal to b.

See [=](#) for the rules of equality.

x                      A value.

y                      A value.

---

**Note**

Registered as an infix operator with precedence **50** and **left** associativity.

---

**5.8 Logical Operators****5.8.1 Meta-Node: and (x, y)**

Logical AND.

Returns [True](#) if x and y evaluate to `True`. Returns [False](#) otherwise.

x                      A Boolean value

y                      A Boolean value

---

**Note**

Registered as an infix operator with precedence **25** and **left** associativity.

**Pattern Matching**

Matches if both the nested patterns in `x` and `y` match the source node. *See [Pattern Matching](#).*

**5.8.2 Meta-Node: `or (x, y)`**

Logical OR.

Returns **True** if either `x` or `y` evaluate to **True**. Returns **False** if both `x` and `y` evaluate to **False**.

<code>x</code>	A Boolean value
<code>y</code>	A Boolean value

**Note**

Registered as an infix operator with precedence **20** and **left** associativity.

**Pattern Matching**

Matches if at least one of the nested patterns in `x` and `y` match the source node. Both the bindings generated by the patterns `x` and `y` are established if the corresponding pattern condition matches. *See [Pattern Matching](#).*

**Note**

This pattern matches even if not all its nested patterns have matched.

**5.8.3 Meta-Node: `not (x)`**

Logical NOT.

Returns **True** if `x` evaluates to **False**.

<code>x</code>	A Boolean value
----------------	-----------------

**Pattern Matching**

Matches if the nested pattern `x` does not match. The bindings generated by `x` are not established by this pattern. *See [Pattern Matching](#).*

**Note**

Since this binding does not establish any bindings, it is treated as a constant pattern and may only appear nested inside other patterns.



## 5.9 Selection Operators

### 5.9.1 Meta-Node: `if(condition, true-value, :(false-value))`

Returns `true-value` if `condition` is [True](#) otherwise returns `false-value`.

If `false-value` is not provided, a failure is returned if `condition` evaluates to [False](#).

**condition**

The condition.

**true-value**

Value to return if `condition` is true.

**false-value (Optional)**

Value to return if `condition` is false. If not provided defaults to a failure of type [No-Value](#).

### 5.9.2 Macro-Node: `case(..(clauses))`

```
<clause> = <condition> : <value>
```

Expands to nested [if](#) expressions.

Each argument is a clause is of the form `condition : value`. The `case` expression evaluates to the value corresponding to the first clause of which the `condition` node evaluates to [True](#). The final clause may also be of the form `value`, in which case it becomes the default value, to which the `case` expression evaluates if the conditions of all the other clauses evaluate to [False](#).

`clauses`                      The clauses.

#### Example

```
case(
  a < b : a,
  b >= a : b
)

# Is equivalent to:

if(a < b, a, if(b >= a, b))
```

#### Example with default value

```
case(
  a < b : -1,
  b > a : 1,
  0
)

# Is equivalent to:

if(a < b, -1, if(b > a, 1, 0))
```

### 5.9.3 Node `True`

The value of this node represents Boolean *True*.

---

### 5.9.4 Node **False**

The value of this node represents Boolean *False*.

## 5.10 Types

### 5.10.1 Meta-Node: **int** (**x**)

Converts **x** to an **integer** value.

- If **x** is an **integer** returns **x**.
- If **x** is a **real** returns **x** with the fractional part truncated.
- If **x** is a **string**, attempts to parse an integer from **x**. Returns the parsed value if successful otherwise returns a failure of type [Invalid-Integer](#).

If **x** is neither of the above returns a failure of type [Type-Error](#).

**x**                      The value to convert to an integer.

#### Pattern Matching

Matches if the source node is an **integer**, in which case **x** is matched to the integer value. See [Pattern Matching](#).

### 5.10.2 Meta-Node: **real** (**x**)

Converts **x** to a **real** number value.

- If **x** is an **integer** or **real** returns **x**.
- If **x** is a **string**, attempts to parse a real number from **x**. Returns the parsed value if successful otherwise returns a failure of type [Invalid-Real](#).

If **x** is neither of the above returns a failure of type [Type-Error](#).

**x**                      The value to convert to a real.

#### Pattern Matching

Matches if the source node is a **real**, in which case **x** is matched to the real value. See [Pattern Matching](#).

### 5.10.3 Meta-Node: **string** (**x**)

Converts **x** to a **string**.

**x**                      The value to convert to a string.

## Pattern Matching

Matches if the source node is a **string**, in which case  $x$  is matched to the string value. See [Pattern Matching](#).

### 5.10.4 Meta-Node: `to-int (x)`

Converts  $x$  to an **integer** value.

Same as [int](#) however with the `target-node` attribute set to [int](#). As such, in the following:

```
a -> to-int (b)
```

Node  $b$  is set to the value of  $a$  converted to an integer.

$x$                       The value to convert.

### 5.10.5 Meta-Node: `to-real (x)`

Converts  $x$  to an **real** number value.

Same as [real](#) however with the `target-node` attribute set to [real](#). As such, in the following:

```
a -> to-real (b)
```

Node  $b$  is set to the value of  $a$  converted to a real number.

$x$                       The value to convert.

### 5.10.6 Meta-Node: `to-string (x)`

Converts  $x$  to an **integer** value.

Same as [string](#) however with the `target-node` attribute set to [string](#). As such, in the following:

```
a -> to-string (b)
```

Node  $b$  is set to the value of  $a$  converted to a string.

$x$                       The value to convert.

### 5.10.7 Meta-Node: `int? (x)`

Returns true if  $x$  is an **integer**.

$x$                       The value to test.

### 5.10.8 Meta-Node: `real? (x)`

Returns true if  $x$  is a **real** valued number.

$x$                       The value to test.

---

### 5.10.9 Meta-Node: `string?(x)`

Returns true if `x` is a **string**.

`x`                      The value to test.

### 5.10.10 Meta-Node: `symbol?`

Returns true if `x` is a **symbol**.

`x`                      The value to test.

### 5.10.11 Meta-Node: `char?`

Returns true if `x` is a **character**.

`x`                      The value to test.

### 5.10.12 Meta-Node: `inf?(x)`

Returns true if `x` is either positive or negative infinity.

`x`                      The value to test.

### 5.10.13 Meta-Node: `NaN?(x)`

Returns true if `x` is a **NaN** (Not a Number) value.

`x`                      The value to test.

## 5.11 Lists

Lists are represented by a special `cons` type, in which the *head* stores the first element of the list and the *tail* stores the list of remaining elements. Neither the *head* nor the *tail* are evaluated until they are actually referenced and used.

The empty list is represented by the value of the node [Empty](#).

### 5.11.1 Meta-Node: `cons(head, tail)`

Creates a list with the `head` as the first element and `tail` as the list of remaining elements.

`head`                      The first element of the list.

`tail`                      The list containing the remaining elements after the first.

## Pattern Matching

Matches if the source node is a non-empty list, in which case `head` is matched to the *head* of the list and `tail` is matched to the *tail* of the list. See [Pattern Matching](#).

### 5.11.2 Meta-Node: `head(list)`

Returns the *head* (first element) of a list.

If `list` is the empty list, returns a failure of type [Empty](#).

If `list` is not a list returns a failure value of type [Type-Error](#).

`list`                      The list.

### 5.11.3 Meta-Node: `tail(list)`

Returns the *tail*, the list containing the elements after the first element, of a list.

If `list` is the empty list, returns a failure of type [Empty](#).

If `list` is not a list returns a failure value of type [Type-Error](#).

`list`                      The list.

### 5.11.4 Meta-Node: `cons?(thing)`

Returns true if `thing` is a list of at least one element, false otherwise.

---

#### Note

Does not return true if `thing` is the empty list.

---

`list`                      The list.

### 5.11.5 Node: `Empty`

The value of this node represents the empty list.

### 5.11.6 Node: `Empty!`

The value of this node is a failure of type [Empty](#). This failure is returned when attempting to access the *head* or *tail* of an empty list.

### 5.11.7 Meta-Node: `list(...(xs))`

Creates a list with elements `xs`.

`xs`                          The list elements.

---

## Pattern Matching

Matches if the source node is a list of the same size as `xs`, in which case each argument in `xs` is matched to the corresponding list element. See [Pattern Matching](#).

### 5.11.8 Meta-Node: `list*(..(xs))`

Creates a list containing, as elements, all the arguments in `xs` excluding the last. The last argument in `xs` is treated as a list containing the remaining elements.

`xs`                      The list elements, with the last argument being the list containing the remaining elements.

## Pattern Matching

Matches if the source node is a list of at least one less elements than the number of elements in `xs`. The arguments, excluding the last, are matched to the corresponding elements in the list with the last argument being matched to the remaining list elements. See [Pattern Matching](#).

### 5.11.9 Meta-Node: `list!(..(xs))`

Creates a list containing, as elements, all the arguments in `xs`.

Unlike `list`, if at least one of `xs` fails to evaluate to a value, the failure is returned.

`xs`                      The list elements.

### 5.11.10 Meta-Node: `nth(list, n)`

Retrieves the element of a list at a particular index.

Returns a failure of type [Index-Out-Bounds](#) if `n` is greater than the number of elements in `list`.

`list`                    The list.

`n`                        The index of the element to retrieved.

### 5.11.11 Meta-Node: `append(list1, list2)`

Returns a list containing the elements of `list2` appended to `list1`.

`list1`                  The initial list.

`list2`                  The list which is appended onto `list1`.

### 5.11.12 Meta-Node: `foldl'(x, f, list)`

Folds a list to a single value, starting from the first element.

The function `f` is first applied on `x` and the [head](#) of `list`. Subsequently, `f` is applied on the result of the previous application and the next element of `list`, until the end of `list` is reached.

<code>x</code>	Initial first argument to <code>f</code> .
<code>f</code>	Function of two arguments.
<code>list</code>	List to fold.

#### 5.11.13 Meta-Node: `foldl(f, list)`

Folds a list to a single value, starting from the first element.

Same as `foldl'` except the `head` of `list` is used as the initial first argument to the fold function `f`.

<code>f</code>	Function of two arguments.
<code>list</code>	List to fold.

#### 5.11.14 Meta-Node: `foldr(f, list, : (x))`

Folds a list to a single value, starting from the last element.

`f` is first applied on the last element of `list` and the value of `x`. If the `x` argument is not provided or `x` evaluates to a failure of type `No-Value`, `f` is first applied on the last two elements of `list`. Subsequently `f` is applied on the previous element of `list` and the result of the previous application, until the *head* of list `list` is reached.

If `list` only has a single element and `x` is not provided, the element is returned as is. If `l` is empty and `x` is provided, `x` is returned as is.

**`f`**  
Function of two arguments.

**`list`**  
List to fold.

**`x (Optional)`**  
Second argument to the application of `f` on the last element of `list`.

#### 5.11.15 Meta-Node: `map(f, list)`

Applies a function on each element of a list.

Returns a list containing the result of applying `f` on each element of `list` in turn.

<code>f</code>	Function of one argument.
<code>list</code>	The list.

#### 5.11.16 Meta-Node: `filter(f, list)`

Filters elements from a list.

Returns a list containing only the elements of `list` for which the function `f` returns true.

<code>f</code>	Function of one argument, which should return true if the argument should be retained in the list or false if it should be removed.
----------------	---

`list`                      The list to filter.

#### 5.11.17 Meta-Node: `every?(f, list)`

Returns true if `f` returns true for every element of `list`.

`f`                          Function of one argument.

`list`                      The list.

#### 5.11.18 Meta-Node: `some?(f, list)`

Returns true if `f` returns true for at least one element of `list`.

`f`                          Function of one argument.

`list`                      The list.

#### 5.11.19 Meta-Node: `not-any?(f, list)`

Returns true if `f` returns false for every element of `list`.

`f`                          Function of one argument.

`list`                      The list.

#### 5.11.20 Meta-Node: `not-every?(f, list)`

Returns true if `f` returns false for at least one element of `list`.

`f`                          Function of one argument.

`list`                      The list.

## 5.12 Strings

### 5.12.1 Meta-Node: `string-at(string, index)`

Returns the character at a given index in the string.

If the index is greater than the number of characters in this string, returns a failure of type [Index-Out-Bounds](#).

`string`                    The string.

`index`                    The index of the character.

---



### 5.12.2 Meta-Node: `string-concat(string, str1, str2)`

Concatenates `str2` to the end of `str1`.

<code>str1</code>	The first string.
<code>str2</code>	The string which is concatenated to <code>str1</code> .

### 5.12.3 Meta-Node: `string->list(string)`

Returns a list containing the characters in a string.

<code>string</code>	The string.
---------------------	-------------

### 5.12.4 Meta-Node: `list->string(list)`

Returns a string containing the concatenation of the elements in a list.

Each element of `list` is converted to a string and concatenated to the result string.

<code>list</code>	List of elements to concatenate.
-------------------	----------------------------------

### 5.12.5 Meta-Node: `format(string, ..(args))`

Creates a formatted string, in which placeholders are replaced by the arguments in `args`.

The sequence `%s` designates a placeholder which is to be replaced by an argument. The first placeholder is replaced by the first argument, the second with the second argument and so on. Each argument is converted to a string prior to being substituted into the result string.

The sequence `%%` designates a literal `%` character and is thus replaced with a `%`.

<code>string</code>	The format string.
<code>args</code>	The arguments to substitute into the string.

## 5.13 Dictionaries

### 5.13.1 Meta-Node: `member(dict, key)`

Retrieves the value of an entry in a dictionary.

If `dict` does not have an entry for `key` a failure, of type [No-Value](#) is returned.

<code>dict</code>	The dictionary.
<code>key</code>	The entry key.

---

## 5.14 Functions

### 5.14.1 Meta-Node: `apply(f, ..(xs))`

Applies a function on an argument list.

The argument list, on which `f` is applied consist of each argument of `xs`, excluding the last, followed by each element of the last argument of `xs`.

<code>f</code>	The function to apply.
<code>xs</code>	The arguments to apply <code>f</code> on.



#### Caution

If `f` is not a function or the last argument of `xs` is not a list, a failure of type `Type-Error`, is returned.

---

## 5.15 Introspection

The `core/introspection` module provides utility meta-nodes for introspecting the nodes comprising a program. These meta-nodes may only be used within macro nodes, during macro expansion, as runtime definitions are not available.

### 5.15.1 Meta-Node: `node?(thing)`

Returns true if `thing` is a node object.

<code>thing</code>	The thing to check whether it is a node.
--------------------	--

### 5.15.2 Meta-Node: `find-node(node, : (module))`

Looks-up a node in a module.

Returns the node object or a failure if no node is found.

#### `node`

The node to lookup, which can be any node expression.

#### `module (Optional)`

The module in which to look-up the node. Defaults to the current module, set by the last `/module` declaration that is processed.

---

#### Note

Currently there is no way to retrieve a module object, thus the `module` argument is not used. This functionality will be added in a future release.

---

### 5.15.3 Meta-Node: `get-attribute(node, attribute)`

Retrieves the value of an attribute of a node.

Returns a failure if the attribute is not set.

---

<code>node</code>	The node object.
<code>attribute</code>	The attribute identifier.

## 5.16 Pattern Matching

Pattern matching is provided by the core module in the form of bindings involving the meta-node instance, which is to be matched, as the target. The binding succeeds if the pattern matches, otherwise it fails.

A meta-node which supports pattern matching, has a `target-node` or `target-transform`, see [Section 3.7](#), such that when an instance of the meta-node appears as the target of a binding, the argument nodes are bound to the values, required in order for the meta-node to return a value that is equivalent to the value of the source node. When there are such values, the pattern is said to have *matched*. If there is no possible value for at least one argument node, all argument nodes should evaluate to failures of type [Match-Fail](#). In this case the pattern has not *matched*.

### Example

```
x -> int(y)
```

In the example, above, `y` is bound to the value of `x` if it is an integer, otherwise `y` evaluates to a failure. There is no argument which will result in `int` returning a non-integer value thus if the source node, `x`, is not an integer the argument node, `y`, is bound to a failure. Since `int` returns the value of its argument directly, when it is an integer, the argument node is simply bound to the source node.

### Example

```
x -> list(y, z)
```

In the example, above, `y` is bound to the first element of `x` and `z` is bound to the second element of `x` if `x` is a list of two elements. These bindings will result in a list, equivalent to `x`, being produced when `y` and `z` are passed as arguments to the `list` meta-node.

### 5.16.1 Nested Patterns

Patterns may be nested, that is an argument to a meta-node instance is itself a meta-node instance of which the operator meta-node supports pattern matching. When the arguments contain one or more nested patterns, the bindings to the argument nodes should only succeed if all nested patterns *match*.

### Example

```
x -> list(int(y), z)
```

The example, above, is similar to the previous example except with the additional condition that the first element of `x` should also be an integer. That is `y` is bound to the first element of `x` and `z` to the second element of `x` if `x` is a list of two elements of which the first element is an integer.

When `_` appears nested inside a pattern it matches anything and does not establish any bindings. This is used to indicate that the value for a particular argument is unimportant.

### Example

```
x -> list(_, y)
```

In the example, above, `y` is bound to the second element of `x` if it is a list of two elements. The value of the first element of `x` is ignored completely.

---

### 5.16.2 Constant Patterns

Constant patterns comprise a constant value as opposed to a node. These patterns *match* when the source node is equal, by `=`, to the constant value. Constant patterns do not result in any bindings being established however they do affect the condition of the pattern in which they are nested.



#### Important

Constant patterns may only be used when nested inside a non-constant pattern.

---

Constant values include any literal constants, such as numbers, strings as well as character literals, produced by the `c` macro, and literal symbols, produced by the `'` macro.

#### Example

```
x -> list(1, y, z)
```

In the example, above, `y` is bound to the second element of `x` and `z` to the third element of `x` if `x` is a list of three elements of which the first element is equal to 1.

The following are examples of invalid uses of constant patterns:

#### Examples: Invalid use of Constant Patterns

```
# Invalid as the pattern is not nested
x -> 1

# Invalid as at least one argument should not be a constant.
x -> list(1, 2)
```



#### Caution

Functor nodes, of which the arguments are all constants, such as `1 + 1`, are only treated as constant patterns if the meta-node supports pattern matching. In this case the `+` meta-node does not support pattern matching, thus `1 + 1` is currently not treated as a constant pattern.

---

### 5.16.3 Matchers

The `matcher` node attribute stores a meta-node which is called to construct the pattern for a given list of arguments. The *matcher* meta-node is called with two arguments: the place to be matched, which should become the *source* node of any bindings established by the pattern, and the pattern *functor* expression itself (including the operator). The meta-node should return a `Pattern` object, which is a dictionary containing the following entries:

#### **condition**

The node expression which evaluates to true if the pattern matches. This should include the conditions of the argument nodes if they are patterns themselves.

#### **bindings**

List of bindings established by the pattern. If there are no bindings established by the pattern, then this entry should be set to the empty list, see [Empty](#).



#### Important

The bindings should not be conditioned on `condition` as they will be conditioned later when the node declarations for the entire pattern (including the parent patterns) is constructed. See [Conditional Bindings](#).

---

**Tip**

Pattern objects may be created with the [Pattern](#) meta-node.

All bindings, established by a pattern, should be established in an explicit context with identifier `match`, which is activated only on failures with type [Match-Fail](#). This allows multiple patterns to be specified on a single node, with the node being set to the value corresponding to the binding of the first pattern that *matches*.

**Example: Multiple Patterns**

```
x -> int(y)
x -> list(int(y))
x -> list("x", int(y))
```

The example above contains multiple patterns involving a single node `y`.

`y` is bound to:

1. the value of `x` if it is an integer, or
2. the first element of `x` if it is a list of one element, which is an integer, or
3. the second element of `x` if it is a list of two elements, with the first element being the string value `"x"` and the second element being an integer.

The following meta-nodes in the `core` module all have a matcher and may thus appear within patterns.

- `'`
- `c`
- [fails?](#)
- [and](#)
- [or](#)
- [not](#)
- [int](#)
- [real](#)
- [string](#)
- [cons](#)
- [list](#)
- [list\\*](#)

**5.17 Module: core/patterns**

This module contains utilities for creating and processing patterns.

**5.17.1 Meta-Node: Pattern(condition, : (binding))**

Creates a `Pattern` object. See [Matchers](#)

**condition**

The node expression which evaluates to true if the pattern matches.

**binding (Optional)**

List of binding expressions of the bindings established by the pattern. Defaults to the empty list if not provided.

**5.17.2 Meta-Node: `get-matcher (node)`**

Returns the *matcher* function, stored in the `matcher` attribute of a node.

Returns a failure if the node's `matcher` attribute is not set.

`node`                      The node object of which to retrieve the *matcher*.

**5.17.3 Meta-Node: `make-pattern (place, pattern)`**

Creates the `Pattern` object for a pattern expression.

**Note**

Can be used for any pattern, including constant patterns.

**`place`**

The place which should be matched to the pattern, i.e. the source node of the bindings established by the pattern.

**`pattern`**

The pattern expression.

**Note**

If `pattern` is a functor expression of which the operator is not a meta-node with a `matcher`, a `Pattern` with a single binding `place -> pattern`, and no condition is returned.

**5.17.4 Meta-Node: `combine-conditions (c1, c2)`**

Returns an expression which is the conjunction of two expressions, by [and](#).

**`c1`**

The first condition, on the left hand side of the `and`.

**`c2`**

The second condition, on the right hand side of the `and`.

If `c1` evaluates to a failure, returns `c2`. If `c2` evaluates to a failure, returns `c1`.

**Tip**

This is useful for creating a condition which combines the conditions of multiple argument nodes.

**5.17.5 Meta-Node: `conditionalize-bindings(condition, bindings)`**

Returns a list where each binding in `bindings` is conditioned on `condition`. See [Conditional Bindings](#).

If `condition` evaluates to a failure, a condition of `True` is assumed, thus the bindings are not conditioned.

**`condition`**

The condition on which to condition the bindings.

**`bindings`**

List of bindings to condition.

### 5.17.6 Failure Type Node: Match-Fail

Failure type indicating that a pattern failed to match.

Node `Match-Fail!` is bound to a failure of this type.

### 5.17.7 Meta-Node: fail-match(condition)

If `condition` evaluates to false or to a failure, returns a failure of type [Match-Fail](#), otherwise returns `true`.

**condition**

The pattern condition.

### 5.17.8 Meta-Node: make-match-bind(src, target)

Generates a binding `src -> target`, with `target` in the `match` context which is activated on failures of type [Match-Fail](#).

**src**

The source of the binding.

**target**

The target of the binding.

### 5.17.9 Meta-Node: make-pattern-declarations(pattern)

Creates the node declarations implementing a pattern.

Returns a single node declaration.

**pattern**

The `Pattern` object for which to create the declarations.

---

#### Tip

The declaration returned by this meta-node is a suitable return value for a `target-transform` function. See [Section 3.7](#).

---

## 5.18 Operator Table

Operator	Precedence	Associativity
<code>.</code>	1000	left
<code>when</code>	850	left
<code>@</code>	800	left
<code>=&gt;</code>	750	left
<code>::</code>	700	left
<code>*</code>	200	left
<code>/</code>	200	left
<code>+</code>	100	left
<code>-</code>	100	left
<code>&lt;</code>	50	left
<code>&lt;=</code>	50	left
<code>&gt;</code>	50	left
<code>&gt;=</code>	50	left
<code>=</code>	50	left

---

Operator	Precedence	Associativity
!=	50	left
and	25	left
or	20	left
!-	15	right
->	10	right
<-	10	left
:	5	right

## 6 Optimizations

This section details functionality for controlling and fine tuning the optimizations performed by the Tridash compiler.

### 6.1 Coalescing

A node is compiled to a runtime node object definition, which stores the details of the node's observers, dependencies, contexts and various other attributes. However, for efficiency, both in terms of speed and code size, not every single node is actually compiled to a runtime node object. When possible successive nodes are coalesced (merged) into a single node, with a value function that combines the functions of the coalesced nodes.

Node coalescing does not affect the behaviour of a program however it can effect which nodes are actually available when interfacing with other languages.



#### Important

Input Nodes, *see Section 2.8*, are never coalesced into other nodes, as they provide the interface by which a program receives its inputs.

#### Example

```
a + b -> x
x -> y
y + z -> output
```

In this example, assuming `a`, `b` and `z` are input nodes and `output` is an output node, i.e. it has no observers, the following intermediate nodes can all be coalesced into the node `output`:

- `a + b`
- `x`
- `y`
- `y + z`

As a result, only the following nodes remain:

- `a`
- `b`
- `z`
- `output`

Node `output` has the dependencies: `a`, `b` and `z` with the following combined value function:



```
# y + z
# x + z ❶
# a + b + z ❷
a + b + z
```

- ❶ y is substituted with its dependency x
- ❷ x is substituted with its dependency a + b

When interfacing with Tridash, from another language, only the values of the nodes remaining after coalescing can be observed. The only nodes guaranteed to never be coalesced into other nodes are input nodes, which are explicitly marked as such, and output nodes, that is nodes with no observers.

To allow the value of a node, which is neither an input nor an output node, to be observed, the value of the node's `coalescable` attribute can be set to false. This guarantees that the node will never be coalesced into another node, and hence a runtime node object will be created for it.

### Example

```
a + b -> x
x -> y
y + z -> output

/attribute(x, coalescable, False) ❶
```

- ❶ Set value of x's `coalescable` attribute to false

In this example, x is guaranteed to never be coalesced into another node and is thus available for interfacing with other languages.

The value function of `output` thus becomes `x + z`, assuming node `y` is coalesced into `x`.



#### Caution

Preventing a node from being coalesced, by setting the `coalescable` attribute to false, may negatively impact the performance of the program.



#### Caution

A node with the `coalescable` attribute set to false, can still be removed if it is not reachable from any input node, and thus its value can never change from its initial value. To prevent the node from being removed, the `removable` attribute should also be set to false.

#### Note

The `coalescable` and `removable` attributes have no effect within meta-nodes.

## 7 Interfacing with Other Languages

The languages that Tridash can interface to, and the foreign function interface itself, depend on the compilation target (the compiler backend).

The current version of Tridash has two compilation targets *JavaScript*, and 32-bit *WebAssembly* (`wasm32`).

## 7.1 Calling External Functions

To be able to call an externally defined (defined outside Tridash code) function from Tridash, a meta-node without a definition has to be declared. This is achieved using the `/external` special operator. *See Section 3.4 for the operator's syntax.* Once an *external meta-node* is declared, it can be used, in Tridash code, as though it is a regular meta-node.

The external meta-node has to be linked to an externally defined function, which implements the functionality of the meta-node, for each backend. This is achieved via backend specific attributes. For the *JavaScript* and *WebAssembly* backends the attributes are `js-name` and `wasm-name`, respectively.

## 7.2 Accessing and Setting Node Values

In order to access the value of a node from outside Tridash code, the node must be given a public identifier. This is done by setting the node's `public-name` attribute to a string identifier.

### Example: Setting Node Public Identifier

```
# Set public identifier of `x` to `node_x`  
  
/attribute(x, public-name, "node_x")
```

Node `x` is now accessible outside Tridash code by the `node_x` identifier.



#### Caution

Currently, it is not checked whether multiple nodes are given the same public identifier.

---

### 7.2.1 Runtime Module Objects

A Tridash application is compiled to a single runtime module object. When targeting the *JavaScript* or *WebAssembly* backends, this is a JavaScript object with at least the following property:

#### **nodes**

An object containing references to the runtime node objects of all publicly accessible nodes. This object contains a property for each publicly accessible node, with the property identifier being the node's public identifier.

The module object also provides the following member function:

#### **set\_values(values)**

Set the values of multiple nodes simultaneously.

`values` is an array with each element being an array of the form `[node, value]` where `node` is the node's runtime node object and `value` is the value to which the node should be set.

### 7.2.2 Runtime Node Objects

The runtime node object provides three methods:

#### **get\_value()**

Returns the value of a node.

#### **set\_value(value)**

Set the value of the node to `value`.

---

**watch(f)**

Call the function `f` whenever the value of the node changes. The function is passed the new value of the node as its only argument.

**Important**

In order to be able to set the value of a node, it must be marked as an input node, by setting its `input` attribute to `true`. See [Section 2.8](#).

---

**Caution**

Unless a node is an *input* node or an *output* node, that is a node with no observers, there is no guarantee that a runtime node object is created for it, even if it is given a public identifier, due to node coalescing. To ensure that a runtime node object is created for a node, for which a public identifier is assigned, its `coalescable` and `removable` attributes should be set to `false`. See [Section 6.1](#) for more information.

---

## 7.3 JavaScript Backend

This section details the foreign function interface of the JavaScript backend.

### 7.3.1 Value Types

Most Tridash primitive value types correspond directly to JavaScript primitives:

- **Integers** are represented by JavaScript integers.
- **Reals** are represented by JavaScript floating point numbers.
- **Booleans** are represented by JavaScript Booleans.
- **Strings** are represented by JavaScript strings.
- **Subnode dictionaries** are represented by JavaScript objects with a property for each subnode. The identifier of each property is the corresponding subnode identifier.

The remaining value types are represented by instances of classes in the runtime library, which is contained in the `Tridash` module object.

- **Characters** are represented by instances of the `Tridash.Char` class, which has one property:

<code>chr</code>	A string containing just the character itself.
------------------	--

- **Lists** are either represented by JavaScript arrays or an internal linked list node class.

The `Tridash.head` and `Tridash.tail` functions return the *head* and *tail* of a list respectively. These functions will return a type error failure value if the argument is not a linked list node or array.

### 7.3.2 Failure Values

Failure values are represented by an instance of the `Tridash.Fail` class, which has one property

<code>type</code>	The failure type. This property is <code>null</code> if the failure does not have a type.
-------------------	---

---

Instances of this class are thrown as JavaScript exceptions rather than being returned directly. Generally failure values are represented by *thunks*, see the next section on lazy evaluation, which, when evaluated, throw a `Tridash.Fail` exception.

---

**Tip**

Use the `Tridash.fail` function, which takes one argument, the failure type (defaulting to `null`), to create failure values, which are wrapped in thunks.

---

### 7.3.3 Builtin Failure Types

The following functions return a JavaScript object, which serves to identify a builtin failure type.

**Tridash.Empty()**

Returns the object representing the empty list.

**Tridash.NoValue()**

Returns the object representing the [No Value](#) failure type.

**Tridash.TypeError()**

Returns the object representing the [Type-Error](#) failure type.

**Tridash.IndexOutOfBounds()**

Returns the object representing the [Index-Out-Bounds](#) failure type.

**Tridash.InvalidInteger()**

Returns the object representing the [Invalid-Integer](#) failure type.

**Tridash.InvalidReal()**

Returns the object representing the [Invalid-Real](#) failure type.

**Tridash.ArityError()**

Returns the object representing the [Arity-Error](#) failure type.

### 7.3.4 Lazy Evaluation

Tridash expressions may be evaluated lazily, that is they are only evaluated at the point where they are first used. Lazily evaluated expressions are wrapped in *thunk* objects, which are instances of the `Tridash.Thunk` class.

Thunks are created using the `Tridash.Thunk` constructor which takes one argument, the *thunk computation* function. The thunk computation function should be a function of no arguments which computes, and returns, the thunk's result value.

The `Tridash.resolve` function, of one argument, computes the value of a `Tridash.Thunk`, passed as the argument, if it has not been computed already, and returns the resulting value. If the argument is not a `Tridash.Thunk` object it is returned directly.

**Important**

`Tridash.resolve` repeatedly computes the value of a thunk, which is the result of computing the value of another thunk until the result is not a thunk. This means `Tridash.resolve` always returns an immediate value, not a thunk object.

---

#### Example: Resolving node value and handling failures

```
try {
  value = Tridash.resolve(value);

  // Use resolved value
  ...
}
```

```

}
catch (e) {
  if (e instanceof Tridash.Fail) {
    // Handle failure
    ...
  }
}

```

### 7.3.5 Linking Meta-Nodes to External Functions

In the *JavaScript* backend, the name of the JavaScript function, which provides the implementation of an *external meta-node* is given by the value of the node's `js-name` attribute. This attribute must be a string which names a function that is globally accessible to the generated JavaScript code, at the time it is run.

#### Example: Linking external meta-node to JavaScript function

```

## Meta-Node of two arguments
/external(fn, a, b)

## Link to JavaScript 'my_func' function
/attribute(fn, js-name, "my_func")

```

The function is called passing in the values of the argument nodes as arguments. If the value for an optional argument is not provided to the meta-node, in the Tridash source, the default value is passed to the function. If the optional argument and the arguments following it do not have default values, they are omitted in the generated function call.

#### Example: External meta-node with optional arguments with default value

```

## Meta-Node with 1 required and 1 optional argument
/external(fn, a, b : 1)

## Link to JavaScript 'my_func' function
/attribute(fn, js-name, "my_func")

## Instance with optional argument omitted
fn(x)

```

This example compiles to the following call to `my_func`:

```
my_func(x, 1)
```

#### Example: External meta-node with optional arguments without default value

```

## Meta-Node with 1 required and 1 optional argument
/external(fn, a, :(b))

## Link to JavaScript 'my_func' function
/attribute(fn, js-name, "my_func")

## Instance with optional argument omitted
fn(x)

```

This example compiles to the following call to `my_func`:

```
my_func(x)
```

Notice that no value is provided for the `b` argument since it was omitted, in the meta-node instance `fn(x)` and it does not have a default value.

Rest arguments are accumulated into a single JavaScript array which is passed as the last argument to the function. If the rest argument list is empty, the argument is omitted entirely in the generated function call.

#### Example: External meta-node with rest arguments

```
## Meta-Node with 1 required and 1 rest argument
/external(fn, x, ..(xs))

## Link to JavaScript 'my_func' function
/attribute(fn, js-name, "my_func")

## Instance 1: 3 rest arguments
fn(a, b, c, d)

## Instance 2: no rest arguments
```

Instance 1 is compiled to the following call to `my_func`:

```
my_func(a, [b, c, d])
```

Instance 2 is compiled to the following call to `my_func`:

```
my_func(a)
```

Notice no value is passed for the second argument which corresponds to the rest argument.

Each argument, passed to an external function, may either be an immediate value or a `Tridash.Thunk` object, in the case of an argument which is lazily evaluated. The `Tridash.resolve` function should be called on each argument, of which the value is actually used, to ensure that the immediate value is obtained.

The function should always return a value, which becomes the return value of the meta-node. This can either be an immediate Tridash value, of one of the types specified in [Value Types](#), or a `Tridash.Thunk` object.

Any failure value exception, thrown inside the function either by the function itself or while evaluating a *thunk*, should be caught and wrapped in a `Tridash.Thunk` object which is returned from the function.

#### Example: Handling failures in external meta-node functions

```
function my_func(a) {
  try {
    // Resolve value of the `a` argument
    // The result may be a failure value
    a = Tridash.resolve(a);

    // Do something with `a`
    ...
  }
  catch (e) {
    // Handle failure value exception
    if (e instanceof Tridash.Fail) {
      // Wrap failure in thunk object and return
      return new Tridash.Thunk(() => { throw e });
    }

    // Rethrow other exceptions
    throw e;
  }
}
```

### 7.3.6 Runtime Module Object

By default, when targeting the JavaScript backend, a standalone JavaScript file is produced, which assigns the `nodes` and `set_values` properties of the runtime module object to the `exports` object. This file can be loaded by a JavaScript runtime which supports the `require` function, such as *Node.js*. The properties of the module object can then be accessed as properties of the object returned by `require`.

#### Example: JavaScript Module Object Usage

```
# Load the compiled JavaScript module. Substitute `module.js` with the
# path to the generated JavaScript file.

const module = require('module.js');
const node_x = module.nodes.node_x;

// Retrieve value of node_x
const value = node_x.get_value();

...

// Set value of node_x to integer 10
node_x.set_value(10);
```

The `get_value` method of the runtime node object takes a single optional argument, which if *true* (the default), the value of the node is fully evaluated if it is a *think* object, representing a lazily evaluated value. If the argument is *false*, *think* objects are not evaluated but are returned directly.

## 7.4 WebAssembly Backend

This section details calling JavaScript functions when targeting WebAssembly.

### 7.4.1 Memory

Tridash value types are represented by objects stored in the heap of the compiled WebAssembly module's memory object. The heap is managed by a tracing garbage collector, which is run whenever a memory allocation requests more space than is available.

In order for a Tridash value to be passed to a JavaScript function, and vice versa, it has to be converted (*marshalled*) to an equivalent representation using JavaScript types. This functionality is provided by the `Tridash.Marshaller` class in the JavaScript component of the runtime library, which is encapsulated in the module object `Tridash`.

### 7.4.2 Value Types

Tridash value types, when converted to JavaScript value types, are represented by the following:

- **Integers** are represented by JavaScript integers.
- **Reals** are represented by JavaScript floating point numbers.
- **Booleans** are represented by JavaScript Booleans.
- **Strings** are represented by JavaScript strings.
- **Arrays** are represented by JavaScript arrays.
- **Subnode dictionaries** are represented by an instances of the `Tridash.Marshaller.Object` class. The `subnodes` property contains an object with a property for each subnode. The identifier of each property is the corresponding subnode identifier.
- **Characters** are represented by instances of the `Tridash.Marshaller.Char` class, which has a single property:

code                      The character code.

- **Symbols** are represented by instances of the `Tridash.Marshaller.Symbol`, which has a single property:

`name`                      The symbol name as a string.

- **Linked list nodes** are represented by instances of the `Tridash.Marshaller.ListNode` class which the following properties:

`head`                      The element stored in the node

`tail`                      The next node in the linked list

- **Failure values** are represented by instances of the `Tridash.Marshaller.Fail` class, which has a single property:

`type`                      The failure type.

- **Raw node references** are represented by instances of the `Tridash.Marshaller.Node` class, which has a single property.

`id`                      The node ID.

### 7.4.3 Builtin Failure Types

The following properties of the `Marshaller.FailTypes` represent builtin failure types as JavaScript Values:

#### **NoValue**

Represents the [No Value](#) failure type.

#### **TypeError**

Represents the [Type-Error](#) failure type.

#### **InvalidInteger**

Represents the [Invalid-Integer](#) failure type.

#### **InvalidReal**

Represents the [Invalid-Real](#) failure type.

#### **ArityError**

Represents the [Arity-Error](#) failure type.

#### **IndexOutOfBounds**

Represents the [Index-Out-Bounds](#) failure type.

#### **Empty**

Represents the empty list.

### 7.4.4 Marshaller

The `Tridash.Marshaller` class is responsible for converting Tridash values stored in the WebAssembly heap to equivalent JavaScript values and vice versa. An instance of this class is automatically created when loading the WebAssembly module.

#### Methods



**to\_tridash(value)**

Convert a JavaScript value (*value*), of a type listed in [Value Types](#), returning a pointer (integer offset) to the Tridash value with the WebAssembly memory heap.

*value* may also be an instance of the `Marshaller.TridashValue` class, which represents a value stored on the WebAssembly heap, at the location given by the `ptr` property. This is useful to create arrays or list objects which contain references to existing Tridash values.

**Note**

Subnode dictionaries (`Tridash.Marshaller.Object` instances) and raw node references (`Tridash.Marshaller.Node` instances) can only be converted to Tridash values if they were obtained from a Tridash to JavaScript value conversion.

**to\_js(pointer)**

Convert a Tridash value, stored at location *pointer* within the WebAssembly heap, to a JavaScript value.

**stack\_push(pointer)**

Push a pointer to a Tridash value onto the GC root set stack.

**stack\_pop()**

Pop and return a pointer, to a Tridash value, from the GC root set stack.

**7.4.5 Lazy Evaluation and Garbage Collection**

A pointer may point to a thunk object, which represents a Tridash value which has not been computed yet. When converting a Tridash value to a JavaScript value, all thunk objects are computed. As a result of this each conversion from Tridash to JavaScript may trigger a garbage collection cycle. Likewise, converting a JavaScript value to a Tridash value may also trigger a garbage collection since a new object is created on the heap.

**Caution**

Converting an infinite list or a cyclic object will result in an infinite loop, due to all lazily evaluated values being computed on conversion.

When converting multiple values, of which the conversion of the first value, *a*, results in a garbage collection cycle being run, the memory held by the second value, *b*, may be reclaimed if it is not referenced by an object in the root set. Furthermore, the pointer to *b* is no longer valid due to the objects being copied to the new heap.

To overcome this problem, before converting a Tridash value to JavaScript, all pointers to other Tridash values should be pushed onto the root set stack, using the `stack_push` method of `Tridash.Marshaller`. After the conversion the updated pointer values should be popped off the root set stack using the `stack_pop` method of `Tridash.Marshaller`.

**Important**

Every call to `stack_push` must be balanced by a call to `stack_pop`.

**Example: Converting Multiple Tridash Values to JS Values**

```
/// Convert the Tridash values, pointer to by `a`, `b` and `c`.

// Push pointer `c` to root set stack
marshaller.stack_push(c);

// Push pointer `b` to root set stack
```

```
marshaller.stack_push(b);

// Convert Tridash value `a` to JS value
a = marshaller.to_js(a);

// Pop updated pointer `b` from stack
b = marshaller.stack_pop();

// Convert Tridash value `b` to JS value
b = marshaller.to_js(b);

// Pop updated pointer `c` from stack
c = marshaller.stack_pop();

// Convert Tridash value `c` to JS value
c = marshaller.to_js(c);
```

#### 7.4.6 Linking Meta-Nodes to External Functions

The name of the JavaScript function, which provides the implementation of an *external meta-node* is given by the value of the node's `wasm-name` attribute. The value of this attribute may either be a symbol or string which names a global JavaScript function, or an expression of the form `module.name` where `module` is the name of a globally accessible object, and `name` is the name of the property which contains the function that implements the meta-node.

##### Example: Linking external meta-node to JavaScript function

```
## Meta-Node of two arguments
/external(f, a, b)
/external(g, a, b)

## Link to JavaScript 'my-func' function
/attribute(f, wasm-name, "my-func")

## Link to JavaScript `func` function that is a property
## of the global object `my_lib`
/attribute(g, wasm-name, my_lib.func)
```

The function is called with the arguments being the pointers to the values of the argument nodes, within the module's memory object.

If the value for an optional argument is not provided to the meta-node, in the Tridash source, and there is no default value, the *null* pointer (0) is passed to the function.

---

##### Note

The `to_js` method of the `Tridash.Marshaller` class, returns the JavaScript `null` value when passed the *null* pointer.

---

##### Example: External meta-node with optional arguments without default value

```
## Meta-Node with 1 required and 1 optional argument
/external(fn, a, :(b))

## Link to JavaScript 'my_func' function
/attribute(fn, wasm-name, "my_func")

## Instance with optional argument omitted
fn(x)
```

---

In this example, `my_func` will be called with the pointer to the value of `x` as the first argument and the *null* pointer (`0`) as the second argument.

The function should return its result as a pointer to a Tridash value, within the WebAssembly memory object.

#### 7.4.7 Runtime Module Object

By default, when targeting the WebAssembly backend, a JavaScript file is produced, along with the `.wasm` file containing the WebAssembly module, which contains a script that loads the module. A *promise*, which resolves to the runtime module object containing the `nodes` property and `set_values` member function, is assigned to the `module` property of the `exports` object. This file can be loaded by a JavaScript runtime which supports the `require` function, such as *Node.js*. The properties of the module object can then be accessed as properties of the object returned by `require`.

---

**Note**

A promise to the module object is assigned, rather than the module object itself, since WebAssembly modules are loaded asynchronously.

---

The runtime module object also contains the following additional fields:

**module**

`WebAssembly.Instance` object of the compiled Tridash module.

**runtime**

`WebAssembly.Instance` object of the runtime library module.

**memory**

The WebAssembly memory object.

**marshaller**

The `Tridash.Marshaller` object for marshalling values to and from the compiled module's heap memory.

#### Example: WebAssembly Module Object Usage

```
# Load the compiled Wasm loader script. Substitute `module.js` with
# the path to the generated JavaScript file.

const module = require('module.js')

module.then((mod) => {
  const marshaller = mod.marshaller;
  const node_x = mod.nodes.node_x;

  // Retrieve value of node_x
  const value = node_x.get_value();

  ...

  // Set value of node_x to integer 10
  node_x.set_value(10);
});
```

The `set_value`, and `get_value` methods, of the runtime node object, automatically marshal values to and from the WebAssembly module's heap.

The `get_value` method, of the runtime node object, takes a single optional argument, which if *false* returns a raw pointer to the Tridash value, of the node, rather than converting it to a JavaScript value. By default this argument is *true*.

---

## 8 Compiler Options

The Tridash compiler (`tridashc`) accepts a number of options which affect the compilation of certain files and the output code that is generated.

*View the man page for `tridashc` or run the command `tridashc -h` for details of how these options are specified.*

### 8.1 Compilation Targets

The compilation target (or compiler backend) determines the type of output file that is generated. The compilation target is specified using the `-t` command-line option or the `backend` output option of the build configuration file, *View the `tridashc` man-page for more information.*

The current version of Tridash has the following targets:

- `javascript`
- `wasm32` (32-bit WebAssembly)

### 8.2 Output Options

The output options control various aspects of the output code that is generated. The options accepted depends on the compilation target.

#### 8.2.1 JavaScript Backend

The JavaScript backend accepts the following options:

**`indented`**

If set to *true*, formatted JavaScript code with line breaks and indentation is generated. If *false* no line breaks or non-syntactic spaces are added.

**`runtime-path`**

Path to the runtime library. If not provided, the runtime library in the Tridash installation directory is used.

**`runtime-linkage`**

Controls how the runtime library is linked with the generated code. This only has an effect when the output is an HTML file, otherwise the runtime library is not linked to the output and has to be loaded manually.

It can be one of the following values:

**`static`**

Embed the runtime library directly inside the HTML file.

**`dynamic`**

Insert a reference to the runtime library, located at the URL given by `runtime-path`, inside the HTML file.

**`none`**

Do not link the runtime library. This should be used if the runtime library is linked manually by a script tag in the HTML source.

**`type`**

Type of output that is generated. Can be one of the following:

**`html`**

Generate an HTML file with the compiled Tridash code embedded in it. The HTML source, which is referenced by the node with identifier given by the `main-ui` option, is used as the template for the HTML output file.

---

**js**

Generate a JavaScript file containing just the compiled Tridash code. The runtime library is not loaded, thus has to be loaded manually, and stored in the global `Tridash` object, prior to loading the generated JavaScript file. This is the default.

**main-ui**

The identifier of the node, which references the contents of an HTML file, to use as the template for the output, when the output type is an HTML file.

**module-name**

The identifier of the JavaScript variable, by which the runtime module object is referenced.

If empty (the default) and the output type is a JavaScript file, the properties of the runtime module object are assigned to the `exports` object. The file can then be loaded using the `require` function, in a JavaScript runtime where it is supported. When the output type is an HTML file, the runtime module object is not accessible.

When given a value, the runtime module object is assigned to the global variable with the identifier given by the value. The JavaScript file can then be loaded using an HTML script tag.

## 8.2.2 WebAssembly Backend

When targeting WebAssembly (`wasm32`), two output files are generated:

- A WebAssembly `.wasm` containing the compiled Tridash code.
- A JavaScript file which contains a script that loads the WebAssembly module, as well as the runtime library module.

To use the compiled module, only the loader script has to be loaded.

The following output options are accepted.

**indented**

If set to *true*, formatted JavaScript code with line breaks and indentation is generated. If *false* no line breaks or non-syntactic spaces are added.

**type**

Type of output that is generated.

By default a separate WebAssembly (`.wasm`) and JavaScript file are generated. The module loader library has to be loaded manually.

If set to `html`, generate an HTML file with the compiled Tridash code embedded in it. The HTML source, which is referenced by the HTML node with identifier given by the `main-ui` option, is used as the template for the HTML output file.

**main-ui**

The identifier of the node, which references the contents of an HTML file, to use as the template for the output, when the output type is an HTML file.

**module-name**

The identifier of the JavaScript variable, through which, the runtime module object is referenced.

If empty (the default) and the output type is not `html`, the properties of the runtime module object are assigned to the `exports` object. The loader script can then be loaded using the `require` function, in a JavaScript runtime where it is supported. When the output type is an HTML file, the runtime module object is not accessible.

When given a value, the runtime module object is assigned to the global variable with the identifier given by the value. The loader script file can then be loaded using an HTML script tag.

**linkage**

Control how the WebAssembly module is linked to the loader script

This can be one of the following:

**local**

The loader script loads the WebAssembly modules from the local file system. This requires a JavaScript runtime which provides the `require` function and the `fs` module. The paths to the compiled module and runtime library are given by the `module-path` and `runtime-path` options, respectively.

**remote**

The loader script loads the WebAssembly modules from a remote location. The URLs to the compiled module and runtime library are given by the `module-path` and `runtime-path` options, respectively.

**embed**

The compiled Tridash module and runtime library are embedded directly in the loader script. The paths to the compiled module and runtime library, which need to be accessible to the compiler, are given by the `module-path` and `runtime-path` options, respectively.

**runtime-path**

Path to the runtime library. If omitted, the path to the runtime library in the Tridash installation is used.

**module-path**

Path to the compiled Tridash module. If omitted, the path to the output file is used.

**stack-size**

The amount of space to reserve for the GC root set stack. The default stack size is 64 KB.

**Caution**

The root set stack does not expand after the space is reserved. If more elements are pushed onto it than there is space, a memory access out of bounds exception will be thrown by the WebAssembly runtime.

---

## 9 Index

- 
- !
  - Core Module
  - Macro, [36](#)
- !!
  - Core Module
  - Meta-Node, [36](#)
- !-
  - Core Module
  - Meta-Node, [36](#)
- !=
  - Core Module
  - Meta-Node, [40](#)
- ,
- Core Module
  - Macro, [32](#)
- \*
  - Core Module
  - Meta-Node, [38](#)
- +
- Core Module
  - Meta-Node, [37](#)
- - Core Module
  - Meta-Node, [38](#)
- >
  - Core Module
  - Macro, [33](#)
  - Special Operator
  - Bindings, [6](#)
- .
- Special Operator
  - Subnodes, [15](#)
- ..
  - Core Module
  - Macro, [35](#)
  - Special Operator
  - Outer Node References, [22](#)
- /
  - Core Module
  - Meta-Node, [38](#)
- /attribute
  - Special Operator
  - Attributes, [14](#)
- /context
  - Special Operator
  - Contexts, [11](#)
- /export
  - Special Operator
  - Modules, [30](#)
- /external
  - Special Operator
  - External Meta-Nodes, [23](#)
- /import
  - Special Operator
  - Modules, [29](#)
- /in
  - Special Operator
  - Modules, [30](#)
- /module
  - Special Operator
  - Modules, [27](#)
- /operator
  - Special Operators
  - Infix Operators, [32](#)
- /quote
  - Special Operator
  - Literal Symbols, [25](#)
- /state
  - Special Operator
  - Node States, [15](#)
- /use
  - Special Operator
  - Modules, [27](#)
- /use-as
  - Special Operator
  - Modules, [28](#)
- :
- Core Module
  - Macro, [34](#)
  - Special Operator
  - Meta-Node Definition, [18](#)
- ::
  - Core Module
  - Macro, [34](#)
- <
  - Core Module
  - Meta-Node, [39](#)
- <-
  - Core Module
  - Macro, [33](#)
- <=
  - Core Module
  - Meta-Node, [39](#)
- =
  - Core Module
  - Meta-Node, [40](#)
- >
  - Core Module
  - Meta-Node, [39](#)
- >=
  - Core Module
  - Meta-Node, [39](#)
- ?
  - Core Module
  - Meta-Node, [35](#)

**@**

- Core Module Macro, [34](#)

**%**

- Core Module Meta-Node, [38](#)

**&**

- Core Module Macro, [33](#)
- Special Operator Node References, [25](#)

**A**

- Accessing Node Values Foreign Function Interface, [59](#)

**and**

- Core Module Meta-Node, [40](#)

**append**

- Core Module Meta-Node, [47](#)

**apply**

- Core Module Meta-Node, [51](#)

**Arithmetic**

- Core Module, [37](#)

**Arity-Error**

- Core Module Failure Type, [37](#)
- Node, [37](#)

**Arity-Error!**

- Core Module Node, [37](#)

**Atoms**

- Syntax Nodes, [1](#)

**Attribute**

- Coalescing, [57](#)
- Instance as Target, [26](#)
- Macro Nodes, [24](#)
- Pattern Matching, [53](#)

**Attributes, [14](#)**

- Nodes, [14](#)

**B****Bind Declarations**

- Nodes Bindings, [6](#)

**Bindings, [6–10](#), [12](#), [15](#)**

- Core Module Macros, [33](#)
- Nodes, [5](#)

**Builtin Failure Types**

- Core Module Failures, [37](#)
- JavaScript Backend Foreign Function Interface, [61](#)

**WebAssembly Backend**

- Foreign Function Interface, [65](#)

**C****c**

- Core Module Macro, [33](#)

**case**

- Core Module Macro, [42](#)

**catch**

- Core Module Meta-Node, [36](#)

**char?**

- Core Module Meta-Node, [45](#)

**coalescable**

- Attribute Coalescing, [57](#)

**Coalescing, [57](#)**

- Optimizations, [57](#)

**combine-conditions**

- Core Module Meta-Node, [55](#)

**Comments**

- Syntax, [1](#)

**Comparison**

- Core Module, [39](#)

**Conditional Bindings**

- Nodes Bindings, [12](#)

**conditionalize-bindings**

- Core Module Meta-Node, [55](#)

**cons**

- Core Module Meta-Node, [45](#)

**cons?**

- Core Module Meta-Node, [46](#)

**Constant Patterns**

- Core Module Pattern Matching, [53](#)

**Contexts, [11](#)**

- Nodes Bindings, [8](#)

**core**

- module Standard Library, [32](#)
- Core Module, [37](#), [39](#), [40](#), [42](#), [43](#), [45](#), [49](#), [51](#), [52](#), [56](#)
- Failure Type, [37](#), [46](#), [56](#)
- Failures, [37](#)
- Macro, [32–36](#), [42](#)
- Macros, [32–34](#)
- Meta-Node, [35–51](#), [54–56](#)
- Node, [37](#), [42](#), [43](#), [46](#), [56](#)
- Pattern Matching, [52–54](#)



Standard Library, [32](#)  
 Utilities, [35](#)

## D

Decimal Syntax  
 Syntax  
 Reals, [4](#)  
 Definitions  
 Core Module  
 Macros, [34](#)  
 Dependency  
 Nodes, [5](#)  
 Dictionaries  
 Value Type, [15](#)  
 Direct References  
 Modules, [30](#)

## E

Empty  
 Core Module  
 Failure Type, [46](#)  
 Node, [46](#)  
 Empty!  
 Core Module  
 Node, [46](#)  
 Escape Sequences  
 Syntax  
 Strings, [4](#)  
 Evaluation  
 Nodes, [8](#)  
 every?  
 Core Module  
 Meta-Node, [49](#)  
 Explicit Contexts  
 Nodes  
 Contexts, [11](#)  
 Exponent Syntax  
 Syntax  
 Reals, [4](#)  
 Exporting Nodes  
 Modules, [30](#)  
 External Functions  
 Foreign Function Interface, [59](#)  
 JavaScript Backend  
 Foreign Function Interface, [62](#)  
 WebAssembly Backend  
 Foreign Function Interface, [67](#)  
 External Meta-Nodes, [23](#)  
 Meta-Nodes, [23](#)

## F

fail  
 Core Module  
 Meta-Node, [35](#)  
 fail-match  
 Core Module  
 Meta-Node, [56](#)

fail-type  
 Core Module  
 Meta-Node, [35](#)  
 fail-type?  
 Core Module  
 Meta-Node, [36](#)  
 fails?  
 Core Module  
 Meta-Node, [35](#)  
 Failure Type, [37](#), [46](#), [56](#)  
 Failures  
 Value Type, [13](#)  
 Failure Values  
 JavaScript Backend  
 Foreign Function Interface, [60](#)  
 Failures, [37](#)  
 Core Module  
 Utilities, [35](#)  
 Value Type, [11](#), [13](#)  
 False  
 Core Module  
 Node, [43](#)  
 filter  
 Core Module  
 Meta-Node, [48](#)  
 find-node  
 Core Module  
 Meta-Node, [51](#)  
 foldl  
 Core Module  
 Meta-Node, [48](#)  
 foldl'  
 Core Module  
 Meta-Node, [47](#)  
 foldr  
 Core Module  
 Meta-Node, [48](#)  
 Foreign Function Interface, [58–68](#)  
 format  
 Core Module  
 Meta-Node, [50](#)  
 Functions  
 Meta-Nodes  
 Semantics, [17](#)  
 Functor Patterns  
 Core Module  
 Pattern Matching, [52](#)  
 Functors, [2](#)  
 Syntax  
 Nodes, [2](#)

## G

Garbage Collection  
 WebAssembly Backend  
 Foreign Function Interface, [66](#)  
 get-attribute  
 Core Module

- Meta-Node, [51](#)
- get-matcher
  - Core Module
  - Meta-Node, [55](#)
- Grammar
  - Syntax, [1](#)
- H**
- head
  - Core Module
  - Meta-Node, [46](#)
- Higher-Order Meta-Nodes
  - Meta-Nodes, [24](#)
- I**
- Identifiers
  - Syntax
  - Nodes, [1](#)
- if
  - Core Module
  - Meta-Node, [42](#)
- Importing Nodes
  - Modules, [29](#)
- Index-Out-Bounds
  - Core Module
  - Failure Type, [37](#)
  - Node, [37](#)
- Index-Out-Bounds!
  - Core Module
  - Node, [37](#)
- inf?
  - Core Module
  - Meta-Node, [45](#)
- Infix Operators, [2](#), [32](#)
  - Syntax
  - Functors, [2](#)
- Initial Values
  - Nodes
  - Bindings, [10](#)
- Input Nodes
  - Nodes, [13](#)
- Instance as Target, [26](#)
  - Meta-Nodes, [26](#)
- int
  - Core Module
  - Meta-Node, [43](#)
- int?
  - Core Module
  - Meta-Node, [44](#)
- Integers
  - Syntax
  - Numbers, [3](#)
- Introspection Utilities
  - Core Module, [51](#)
- Invalid-Integer
  - Core Module
  - Failure Type, [37](#)

- Node, [37](#)
- Invalid-Integer!
  - Core Module
  - Node, [37](#)
- Invalid-Real
  - Core Module
  - Failure Type, [37](#)
  - Node, [37](#)
- Invalid-Real!
  - Core Module
  - Node, [37](#)
- J**
- JavaScript Backend
  - Foreign Function Interface, [60–63](#)
- L**
- Lazy Evaluation
  - JavaScript Backend
  - Foreign Function Interface, [61](#)
  - WebAssembly Backend
  - Foreign Function Interface, [66](#)
- list
  - Core Module
  - Meta-Node, [46](#)
- list!
  - Core Module
  - Meta-Node, [47](#)
- list\*
  - Core Module
  - Meta-Node, [47](#)
- list->string
  - Core Module
  - Meta-Node, [50](#)
- Lists
  - Core Module, [45](#)
- Literal Bindings
  - Nodes
  - Bindings, [10](#)
- Literal Symbols, [25](#)
  - Macro-Nodes, [25](#)
- Literals, [3](#), [4](#)
  - Core Module
  - Macros, [32](#)
  - Syntax, [3](#)
- Local Nodes
  - Meta-Nodes
  - Meta-Node Definition, [20](#)
- Logical Operators
  - Core Module, [40](#)
- M**
- Macro, [32–36](#), [42](#)
- macro
  - Attribute
  - Macro Nodes, [24](#)
- Macro Nodes, [24](#), [25](#)

- Meta-Nodes, [24](#)
- Macro-Nodes, [25](#)
- Macros, [32–34](#)
- make-match-bind
  - Core Module
  - Meta-Node, [56](#)
- make-pattern
  - Core Module
  - Meta-Node, [55](#)
- make-pattern-declarations
  - Core Module
  - Meta-Node, [56](#)
- map
  - Core Module
  - Meta-Node, [48](#)
- Marshaller
  - WebAssembly Backend
  - Foreign Function Interface, [65](#)
- Match-Fail
  - Core Module
  - Failure Type, [56](#)
  - Node, [56](#)
- Match-Fail!
  - Core Module
  - Node, [56](#)
- matcher
  - Attribute
  - Pattern Matching, [53](#)
- Memory
  - WebAssembly Backend
  - Foreign Function Interface, [64](#)
- Meta-Node, [35–51](#), [54–56](#)
- Meta-Node Definition, [18–22](#)
  - Meta-Nodes, [18](#)
- Meta-Node References
  - Meta-Nodes, [24](#)
- Meta-Nodes, [18](#), [22–24](#), [26](#)
  - Macro Nodes, [24](#)
  - Meta-Node Definition, [19–22](#)
  - Nodes
  - Semantics, [17](#)
  - Semantics, [17](#)
- module
  - Standard Library, [32](#)
- Module Creation
  - Modules, [27](#)
- Module Pseudo-Nodes
  - Modules, [27](#)
- Modules, [27–30](#)
  - Infix Operators, [32](#)
  - Operators, [31](#)
- N**
- NaN?
  - Core Module
  - Meta-Node, [45](#)
- Nested Meta-Nodes
  - Meta-Nodes
    - Meta-Node Definition, [22](#)
  - Nested Patterns
    - Core Module
    - Pattern Matching, [52](#)
  - No-Value
    - Core Module
    - Failure Type, [37](#)
    - Node, [37](#)
  - No-Value!
    - Core Module
    - Node, [37](#)
  - Node, [37](#), [42](#), [43](#), [46](#), [56](#)
  - Node Creation
    - Nodes, [5](#)
  - Node Expression Representation
    - Meta-Nodes
      - Macro Nodes, [24](#)
  - Node Lists
    - Syntax
      - Nodes, [3](#)
  - Node References, [25](#)
    - Macro Nodes, [25](#)
  - Node States, [15](#)
    - Nodes
      - Bindings, [15](#)
  - node?
    - Core Module
    - Meta-Node, [51](#)
  - Nodes, [1–3](#), [5](#), [8](#), [13–15](#)
    - Bindings, [6–10](#), [12](#), [15](#)
    - Contexts, [11](#)
    - Semantics, [5](#), [17](#)
    - Syntax, [1](#)
  - not
    - Core Module
    - Meta-Node, [41](#)
  - not-any?
    - Core Module
    - Meta-Node, [49](#)
  - not-every?
    - Core Module
    - Meta-Node, [49](#)
  - nth
    - Core Module
    - Meta-Node, [47](#)
  - Numbers, [3](#), [4](#)
    - Syntax
      - Literals, [3](#)
- O**
- Observer
  - Nodes, [5](#)
- Operator Associativity
  - Syntax
    - Infix Operators, [2](#)
- Operator Precedence

- Syntax
  - Infix Operators, [2](#)
- Operator Table
  - Core Module, [56](#)
  - Modules
    - Operators, [31](#)
- Operators, [31](#)
- Optimizations, [57](#)
- Optional Arguments
  - Meta-Nodes
    - Meta-Node Definition, [19](#)
- or
  - Core Module
    - Meta-Node, [41](#)
- Outer Node References, [22](#)
  - Meta-Nodes, [22](#)
- P**
- Pattern
  - Core Module
    - Meta-Node, [54](#)
- Pattern Matching, [52–54](#)
  - Core Module, [52](#)
- Pattern Matching Utilities
  - Core Module
    - Pattern Matching, [54](#)
- Pattern Object
  - Core Module
    - Pattern Matching, [53](#)
- R**
- real
  - Core Module
    - Meta-Node, [43](#)
- real?
  - Core Module
    - Meta-Node, [44](#)
- Reals, [4](#)
  - Syntax
    - Numbers, [4](#)
- Recursive Meta-Nodes
  - Meta-Nodes, [22](#)
- Registering Infix Operators
  - Modules
    - Infix Operators, [32](#)
- Rest Arguments
  - Meta-Nodes
    - Meta-Node Definition, [19](#)
- Runtime Module Object
  - JavaScript Backend
    - Foreign Function Interface, [63](#)
  - WebAssembly Backend
    - Foreign Function Interface, [68](#)
- Runtime Module Objects
  - Foreign Function Interface, [59](#)
- Runtime Node Objects
  - Foreign Function Interface, [59](#)

- S**
- Selection Operators
  - Core Module, [42](#)
- Self Node
  - Meta-Nodes
    - Meta-Node Definition, [21](#)
- Semantics, [5, 17](#)
- some?
  - Core Module
    - Meta-Node, [49](#)
- Special Operator
  - Attributes, [14](#)
  - Bindings, [6](#)
  - Contexts, [11](#)
  - External Meta-Nodes, [23](#)
  - Literal Symbols, [25](#)
  - Meta-Node Definition, [18](#)
  - Modules, [27–30](#)
  - Node References, [25](#)
  - Node States, [15](#)
  - Outer Node References, [22](#)
  - Subnodes, [15](#)
- Special Operators
  - Infix Operators, [32](#)
- Stack Usage
  - Meta-Nodes, [22](#)
- Standard Library, [32](#)
- string
  - Core Module
    - Meta-Node, [43](#)
- string->list
  - Core Module
    - Meta-Node, [50](#)
- string-at
  - Core Module
    - Meta-Node, [49](#)
- string-concat
  - Core Module
    - Meta-Node, [50](#)
- string?
  - Core Module
    - Meta-Node, [45](#)
- Strings, [4](#)
  - Core Module, [49](#)
- Syntax
  - Literals, [4](#)
- Subnodes, [15](#)
  - Nodes, [15](#)
- symbol?
  - Core Module
    - Meta-Node, [45](#)
- Symbols
  - Syntax
    - Nodes, [1](#)
- Syntax, [1, 3](#)
  - Functors, [2](#)
  - Infix Operators, [2](#)

- Literals, [3, 4](#)
- Nodes, [1–3](#)
- Numbers, [3, 4](#)
- Reals, [4](#)
- Strings, [4](#)

## T

tail

- Core Module
- Meta-Node, [46](#)

Target Node Transform

- Instance as Target
- Meta-Nodes, [26](#)

target-node

- Attribute
- Instance as Target, [26](#)

target-transform

- Attribute
- Instance as Target, [26](#)

to-int

- Core Module
- Meta-Node, [44](#)

to-real

- Core Module
- Meta-Node, [44](#)

to-string

- Core Module
- Meta-Node, [44](#)

True

- Core Module
- Node, [42](#)

Two-Way Bindings

- Nodes
- Bindings, [9](#)

Type Checks

- Core Module, [43](#)

Type Conversions

- Core Module, [43](#)

Type-Error

- Core Module
- Failure Type, [37](#)
- Node, [37](#)

Type-Error!

- Core Module
- Node, [37](#)

## U

Utilities, [35](#)

## V

Value Conversions

- WebAssembly Backend
- Foreign Function Interface, [65](#)

Value Propagation

- Nodes
- Bindings, [7](#)

Value Type, [11, 13, 15](#)

Value Types

- JavaScript Backend
- Foreign Function Interface, [60](#)
- WebAssembly Backend
- Foreign Function Interface, [64](#)

## W

WebAssembly Backend

- Foreign Function Interface, [64–68](#)