WHAND LANGUAGE REFERENCE MANUAL (V2)

*By Alain R. Marchand, sept. 2019 revised april 2024*

Content

[1. Introduction 3](#_Toc20424082)

[2. Lexical analysis 3](#_Toc20424083)

[2.1. Line structure 3](#_Toc20424084)

[2.1.1. Physical lines 3](#_Toc20424085)

[2.1.2. Explicit line joining 3](#_Toc20424086)

[2.1.3. Comments 4](#_Toc20424087)

[2.1.4. Packing logical lines 4](#_Toc20424088)

[2.1.5. Blank lines 4](#_Toc20424089)

[2.1.6. Whitespace before and between tokens 4](#_Toc20424090)

[2.2. Identifiers, keywords and delimiters 4](#_Toc20424091)

[2.2.1. Identifiers 4](#_Toc20424092)

[2.2.2 Keywords 4](#_Toc20424093)

[2.2.3. Operators and comparators 5](#_Toc20424094)

[2.2.4. Delimiters 5](#_Toc20424095)

[3. Data model 5](#_Toc20424096)

[3.1. Nature of objects 5](#_Toc20424097)

[3.2. Type inference 6](#_Toc20424098)

[4. Expressions 7](#_Toc20424099)

[4.1. Atoms 7](#_Toc20424100)

[4.1.1. Parenthesized forms 7](#_Toc20424101)

[4.1.2. List explicit form 7](#_Toc20424102)

[4.1.3. List subscription 7](#_Toc20424103)

[4.1.4. Subscription with a list 8](#_Toc20424104)

[4.1.5. List implicit form 8](#_Toc20424105)

[4.1.6. List distributivity 8](#_Toc20424106)

[4.2. Unary operations 8](#_Toc20424107)

[4.3. Binary arithmetic operations 8](#_Toc20424108)

[4.4. Binary logic (event) operations 9](#_Toc20424109)

[4.5. Comparisons 9](#_Toc20424110)

[5. Execution model 10](#_Toc20424111)

[5.1. Structure of a program 10](#_Toc20424112)

[5.2 Definitions 10](#_Toc20424113)

[5.2.1. Objects without definition 10](#_Toc20424114)

[5.2.1. Simplified clauses 11](#_Toc20424115)

[5.3. Execution 11](#_Toc20424116)

[5.4. Delayed events 11](#_Toc20424117)

[5.5. Updating object consequences 11](#_Toc20424118)

[5.6. Procedural programming 12](#_Toc20424119)

[6. Functions 12](#_Toc20424120)

[6.1 Built-in functions and constants 12](#_Toc20424121)

[6.1.1. Built-in constants 13](#_Toc20424122)

[6.1.2. List of built-in functions 13](#_Toc20424123)

[6.1.3. Logic functions 13](#_Toc20424124)

[6.1.4. Event functions 15](#_Toc20424125)

[6.1.5. Mathematical functions 15](#_Toc20424126)

[6.1.6. List functions 16](#_Toc20424127)

[6.1.7. Absolute time functions 18](#_Toc20424128)

[6.1.8. I/O functions and objects 18](#_Toc20424129)

[6.1.9. Miscellaneous functions and keywords 21](#_Toc20424130)

[6.2. User-defined functions 23](#_Toc20424131)

[6.2.1. Overview 23](#_Toc20424132)

[6.2.2. Function definition 23](#_Toc20424133)

[6.2.3. Accessory variables 23](#_Toc20424134)

[6.2.4. Function application 23](#_Toc20424135)

WHAND LANGUAGE REFERENCE MANUAL (V2)

*By Alain R. Marchand, sept. 2019*

This reference manual aims at being accurate and complete description of the syntax and basic mechanics of Whand. For an informal introduction to the language, see WHAND TUTORIAL. To use a particular implementation of Whand, see WHAND USER MANUAL. For a deeper understanding of the mechanics of Whand, see WHAND DEVELOPER MANUAL.

Whand is a form of functional, parallel, event-oriented, static-typed language with simplified syntax, type inference and limited operations. It focuses on the readability of programs.

Current Whand version is V2. There is no Whand V1.

# 1. Introduction

This reference manual is a description of Whand programming language, not a tutorial. Most implementation details are deferred to WHAND USER MANUAL and WHAND DEVELOPER MANUAL. Since Whand aims above all at clarity, English will be used through most of the presentation. The reader is assumed to have at least superficial acquaintance with Whand and only minimal technical knowledge.

The descriptions of lexical analysis and syntax use a simplified notation. In this manual (but not in Whand itself), italics are used to identify *keywords* and important concepts. An arrow 🡪 separates a name from its definition. Items inside square brackets [ ] correspond to sequences of characters chosen by the user, such as identifiers. Expression […] designates zero or more repetitions of the preceding item. To delimit an item, curly brackets { } may be used in this manual. Note that the Whand language never uses square brackets or curly brackets, and that it ignores italics or other formatting.

# 2. Lexical analysis

## 2.1. Line structure

A Whand program is divided into a number of logical lines that correspond to physical lines unless two or more physical lines are explicitly joined, or two or more logical lines are packed into a single physical line.

### 2.1.1. Physical lines

A physical line is a sequence of characters terminated by an end-of-line sequence: ASCII sequence CR LF (return followed by linefeed). The end of input also terminates the final physical line.

### 2.1.2. Explicit line joining

Two or more physical lines may be joined into a logical line using the backslash character (\). This character must be the last non-whitespace character before the end of the physical line.

### 2.1.3. Comments

A comment starts with a hash character (#) and ends at the end of the physical line. A comment always signifies the end of the logical line. Comments are ignored by the syntax. Hash characters should not be used outside comments.

### 2.1.4. Packing logical lines

Several logical lines belonging to the same object definition may be concatenated into a single physical line. At least one blank character (space, tab) should be used to separate the logical lines. For readability, this practice is not recommended except for very simple definitions.

Lines belonging to different object definitions cannot be packed (this includes the definitions of elements in a list of [implicit form](#_4.1.5._List_implicit)).

### 2.1.5. Blank lines

A logical line that contains only spaces, tabs and possibly a comment is ignored.

### 2.1.6. Whitespace before and between tokens

Leading whitespace (spaces and tabs) at the beginning of a logical line may be used for readability. They are ignored.

Consecutive whitespace characters are treated as one whitespace character.

The whitespace characters space and tab can be used interchangeably to separate tokens. Whitespace is needed between two tokens only if their concatenation could otherwise be interpreted as a single token.

## 2.2. Identifiers, keywords and delimiters

### 2.2.1. Identifiers

*Identifiers* (also referred to as names) are unlimited in length. Using explicit, informative names in Whand programs is highly recommended.

Identifiers cannot have more than one [definition](#_5.2_Definitions). Redefining an object yields a fatal error at compile time.

The valid characters for identifiers are the uppercase and lowercase letters A through Z, the dot ., the underscore \_ and, except for the first character, the digits 0 through 9. Case is significant. Other characters are allowed in the following cases:

Double quote delimiters " may be added around a name to specify a state object (equivalent to a string literal). The delimiters are part of the name.

A single quote ' at the end of names identifies variables within [user-defined functions](#_6.2._User-defined_functions).

Identifiers for objects of delay nature are a number followed by a time unit (ms, s, mn, min, h, day or wk).

Some implementations may allow a larger set of characters for identifiers, only excluding characters serving as delimiters or operators. Special character ` (inverted quote) is not allowed anywhere since it is used internally.

### 2.2.2 Keywords

The following identifiers are used as reserved words, or keywords of the language, and cannot be used as ordinary identifiers. They consist of [special tokens](#_6.1.9._Miscellaneous_functions), [built-in constants](#_6.1.1._Built-in_constants), [built-in functions](#_6.1.2._List_of) and [input/output functions and objects](#_6.1.8._I/O_functions). Keywords must be spelled exactly as written here:

[be](#be) [exit](#exit) [hide](#hide) [include](#include)

[show](#show) [until](#until) [unused](#unused) [when](#when)

[empty](#empty) [epsilon](#epsilon) [false](#false) [true](#true)

[monday … sunday](#monday) [january … december](#january)

[absv](#absv) [add](#add) [alea](#alea) [all](#all) [any](#any)

[begin](#begin) [change](#change) [count](#count) [cumul](#cumul) [dateis](#dateis)

[dayis](#dayis) [end](#end) [find](#find) [hasvalue](#hasvalue) [have](#have)

[inter](#inter) [intg](#intg) [isin](#isin) [lasted](#lasted) [listfind](#listfind)

[logd](#logd) [match](#match) [next](#next) [occur](#occur) [order](#order)

[pick](#pick) [pointer](#pointer) [powr](#powr) [proba](#proba) [ramp](#ramp)

[sequence](#sequence) [shuffle](#shuffle) [since](#since) [sort](#sort) [sqrt](#sqrt)

[start](#start) [steps](#steps) [time](#time) [timeis](#timeis) [to](#to)

[weekis](#weekis) [within](#within)

[call](#call) [command](#command) [display](#display) [key](#key) [load](#load)

[output](#output) [pin](#pin) [print](#print) [read](#read) [store](#store)

[tell](#tell) [touch](#touch) [write](#write)

### 

### 2.2.3. Operators and comparators

Whand employs as few special characters as possible. The following tokens are [operators](#_4.3._Binary_arithmetic):

+ - \* /

The following tokens are [comparators](#_4.5._Comparisons):

< > <= >= = != and or not is isnot

### 2.2.4. Delimiters

The following characters serve as delimiters in the grammar:

( ) , :

The following printing ASCII characters have special meaning as part of other tokens or are otherwise significant:

' " # \

# 3. Data model

## 3.1. Nature of objects

Objects are the basic elements in Whand. All the information in a Whand program is represented by objects or by relations between objects. An object’s behavior is entirely specified in its [definition](#_5.2_Definitions). A Whand program consists only of a set of object definitions.

Every object has an identity, a nature and a value. An object’s identity and its nature never change. The value of objects can change or remain constant during execution.

An object’s nature (also called type) defines the possible values for objects of that nature and determines the operations that the object supports. Natures are determined at compile time using a form of [type inference](#_3.2._Type_inference). To perform type inference, Whand sometimes requires help in the form of[*be*](#be) clauses.

All conversions of nature between values must be performed explicitly using appropriate operators (e.g. dividing a delay by another delay yields a number).

There are exactly five possible natures for an object:

number: a number object can take any floating point or integer value. Numbers support usual operators (+ - \* /) and comparators (> < >= <= = !=). There is no distinction between integers and floating point numbers, e.g. 3 is equal to 3.0.

state: a state object can only take a string literal as value, enclosed or not between double quotes. States support comparison with operator *is* or *isnot*, e.g. {stop: *when* light *is* *not* green: *true*}.

event: an event object can only take value *true* or *false*. Events may alternate between true or false any number of times during execution.

The *onset* of an event is the instant it changes from value *false* to value *true*, or the instant of *start* of program execution if the event is declared as *true* at *start*. The *onset* of an event can be specifically addressed using function [*begin*](#begin).

The *offset* of an event is the instant it changes from value *true* to value *false*. It can be specifically addressed using function [*end*](#end).

Events support logical operators (*and*, *or*, *not*).

Each [*condition*](#clause) in an object declaration must be an event.

[Delayed events](#_5.5._Delayed_events) are events whose *true* or *false* value is determined by the software, as a function of elapsed time.

delay: the value of a delay object is the literal representation of a number followed by s (seconds). Delays specified in other time units, e.g. {1 day} are internally converted to seconds (i.e. 86400s).

Delays support the usual comparators (> < >= <= = !=) but only three operators (+ - /) (n.b. dividing two delays yields a number). A delay can be multiplied or divided by a number, e.g. {1.5\*12s, 12s/6}.

list: a list object is an ordered set of identifiers corresponding to objects of any nature, the list’s elements. In a [list in explicit form](#_4.1.2._List_explicit), element identifiers are separated by commas.

Lists support [list functions](#_6.1.6._List_functions) as well as any operator or function appropriate to the nature of their elements, a property called [distributivity](#_4.1.6._List_distributivity).

## 3.2. Type inference

At compile time, Whand attempts to attribute a nature (type) to all objects in the program.

Type inference first looks for objects of obvious nature, such as [built-in constants](#_6.1.1._Built-in_constants), [inputs or outputs](#_6.1.8._I/O_functions), decimal numbers, explicit delays or lists etc. It then proceeds incrementally by transferring the nature of a value to the object to which the value is assigned and vice-versa.

Other rules are also applied, based on the allowed nature of objects in expressions and functions, or the fact that an object or expression that serves as a [condition](#clause) must be an event.

This process continues until no more progress is made or a conflict occurs. Nature conflict is a fatal error. Remaining objects may tentatively be attributed state nature.

One difficulty in finding the nature of an object is [distributivity](#_4.1.6._List_distributivity), which allows applying most operations to lists.

Another difficulty is that the nature of [subscriptions](#_4.1.3._List_subscription) from a list cannot be directly determined, even when the list is in explicit form.

Whand may be unable to resolve all objects, in which case an error occurs at compile time and Whand requires information on some object’s nature. One way to help Whand determine the nature of an object is to use a [*be*](#be) clause.

Within a [user-defined function definition](#_6.2.2._Function_definition) or [accessory variable](#_6.2.3._Accessory_variables) definition, all defined objects must be virtual (prime) objects. This may require tricks to force the nature of these objects. For instance, one may use the fact that a condition must be an event (which precludes distributivity) and that condition *false* is never triggered, e.g. the following clauses indicate that my\_object' is:

- a number: {hint\_number': *when false and* (my\_object' = 0)}

- a delay: {hint\_delay': *when false* and (my\_object' = 0s)}

- an event: {hint\_event': *when false and* my\_object' }

- a state: {hint\_state': *when false* *and* (my\_object' *is* monday)}

- a list: {hint\_list': *when false* *and* (my\_object' *match empty*)}

The accessory variables [hint…'] have arbitrary names. They are used only for type inference and play no role during execution.

# 4. Expressions

This chapter explains the meaning of the elements of expressions in Whand.

## 4.1. Atoms

Atoms are the most basic elements of expressions. The simplest atoms are identifiers or literals. Forms enclosed in parentheses, are also categorized syntactically as atoms. Expressions consist of atoms separated by operators or comparators.

### 4.1.1. Parenthesized forms

A parenthesized form is an optional expression list enclosed in parentheses. Parenthesized forms are evaluated first. They are used to disambiguate expressions or simply for clarity.

A blank separator may be used in place of a parenthesized form containing a single atom, e.g. {*pin* 1} instead of {*pin*(1)}.

### 4.1.2. List explicit form

A list explicit form is a possibly empty series of expressions, separated by commas.

List explicit form 🡪 [list identifier],[…]

A list with zero elements is defined by parentheses enclosing a single comma {(,)} or by keyword [*empty*](#empty).

### 4.1.3. List subscription

A subscription selects an element from a list. Each element is referred to by an index between parentheses.

subscription 🡪 [list identifier]([index])

A list in explicit form has integer indices, e.g. {my\_list(3)}. Indices run from 1 to the size (n) of the list. Index 1 refers to the first element in the list.

Positive indices cycle, meaning that indices n+1 and above again refer to elements 1 and above, so does index 2n+1, etc.

Index 0 does not refer to any element in the list. It may however be used in a [list in implicit form](#_4.1.5._List_implicit).

Index -1, like index n, refers to the last element in the list. Negative indices, down to (-n), refer to elements listed backward from the last element.

Negative indices do not cycle, so referring to element of index below –n does not yield any value, a condition that may be tested using function *hasvalue*.

### 4.1.4. Subscription with a list

A list can be used as an index due to [distributivity](#_4.1.6._List_distributivity). In that case, the result of the subscription is not a single element but a list. The result list has the same size as the index list. It may be larger than the source list, e.g. {(5,6)(1,2,2,1)} yields {5,6,6,5} and {(4,)(*ramp* 7)} yields {4,4,4,4,4,4,4}.

### 4.1.5. List implicit form

A list in implicit form has literals as indices. Each literal index is a Whand identifier for an object. A list in implicit form is defined element by element, e.g. in {color(light): green}, {color} is the list identifier, {light} is the index, and {green} is all or part of the definition of one element {color(light)}.

A list in implicit form is a form of dictionary. It can be considered as conferring an attribute to the object identified by its index, e.g. object {light} has attribute {color}. Because attribute {color} is a list, several objects can have a common attribute {color}, while the value of the attribute, e.g. {green} can be different for each object (see also function [*have*](#have)).

The elements in an implicit list are ordered according to the order of appearance of their [definition](#_5.2_Definitions).

In this particular case, the order of object definition has consequences on list order.

Each element in an implicit list has its own definition. It cannot have more than one definition.

### 4.1.6. List distributivity

Expressions can involve operations between lists and objects of any nature. If the operator is not a list operator, distributivity will apply. The result of the operation will be a list of results obtained by applying the operator to each of the elements, e.g. {(1,2,3)\*2s} yields {2s,4s,6s}. When a non-list operator is applied between two lists, the result is obtained by applying the operator to the pair of elements that are in the same position in the two lists e.g. {(1s,2s,3s)\*(4,5,6)} yields {4s,10s,18s}.

Operations or functions that normally return a list do not support distributivity. Operations or functions that expect a list argument do not support distributivity on this argument.

## 4.2. Unary operations

The unary - (minus) operator yields the negation of its numeric argument.

The unary *not* operator yields the negation of its event argument.

If the argument has no value, the result has no value, a condition that may be tested using function [*hasvalue*](#hasvalue).

## 4.3. Binary arithmetic operations

The binary arithmetic operations have the conventional priority levels. Any arithmetic operation can be applied to a list, provided arguments are compatible, due to distributivity.

If one of the arguments has no value, the result has no value, a condition that may be tested using function [*hasvalue*](#hasvalue).

The \* (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be an integer and the other must be a delay. The \* operator may be omitted if the identifier of the first argument is a numerical value and the first character of the identifier of the second argument is not a number, e.g. {2.3length} instead of {2.3\*length}.

The / (division) operator yields the quotient of its arguments. The arguments must either both be numbers, or both be delays. All numbers and delay values are treated as floating point numbers. The result is always a number. Division by zero does not yield any value, a condition that may be tested using function [*hasvalue*](#hasvalue).

The - (subtraction) operator yields the difference of its arguments. The arguments must either both be numbers or both delays.

The + (addition) operator yields the sum of its arguments in the case of numbers or delays. The arguments must either both be numbers or both delays (but see below [logic addition](#_4.4._Binary_logic)).

## 4.4. Binary logic (event) operations

The + (logic addition) operator is also allowed when the first argument is an event and the second argument a delay. In that case, the result is a [delayed event](#_5.5._Delayed_events).

Other logic operators are *and* and *or*. They have lower priority than arithmetic operations or logic addition.

The and operator yields an event that is the intersection of its arguments, i.e. it is *true* during each period when both arguments are *true*. If one of the arguments has no value, the result has no value, a condition that may be tested using function [*hasvalue*](#hasvalue).

The or operator yields an event that is the union of its arguments. , i.e. it is *true* during each period when at least one of its arguments is *true*. If one of the arguments has no value, the result is the value of the other argument (lazy or). If none of the arguments has value, the result has no value, a condition that may be tested using function [*hasvalue*](#hasvalue).

## 4.5. Comparisons

All comparison operations in Whand have the same priority, which is lower than that of any arithmetic or logic operation.

If one of the arguments has no value, the result has no value, a condition that may be tested using function [*hasvalue*](#hasvalue). The result of the comparison is then treated as *false*. However, the opposite (*not*) of the result also has no value and it is also treated as *false*.

Comparators <, >, =, >=, <=, and != compare the values of two objects. The objects need to have the same nature.

The following list describes the comparison behavior of objects according to their nature.

Events can be compared using = or !=, which test whether they are both *true* or both *false*.

Numbers and delays (in seconds) compare mathematically within a level of precision set by parameter FloatPrecision (e.g. 5.0E-8).

is, isnot: States can only be compared using operator *is* or *isnot* (also spelled {*is not*}). Some implementations also allow = and != between states.

List matching: Lists can be compared with operator *match*, which yields an event, or using distributivity, which yields a list of events the same size as the longest of the argument lists.

# 5. Execution model

## 5.1. Structure of a program

A Whand program is constructed from object definitions. All the object identifiers that appear in a program are visible to all objects in the program. However, objects that are implemented as part of user-defined functions are not visible to other objects in the program unless special naming conventions are used (see [user-defined functions](#_6.2._User-defined_functions)).

## 5.2 Definitions

An object definition starts with an identifier, followed by one or more *when* clauses.

Keyword *when* is the fundamental building block of Whand. Whatever its actual form, each of the clauses in an object definition is exactly equivalent to one (or two) *when* clause(s). In this sense, Whand may be considered to have only one instruction type.

In Whand, the order of object definitions is irrelevant and does not affect [execution](#_5.3._Execution), except in specific cases (list in [implicit form](#_4.1.5._List_implicit), [accessory variables](#_6.2.3._Accessory_variables),[*show*](#show))

Within an object definition, the order of clauses is irrelevant (except for some cases of [simplified clauses](#_5.2.1._Simplified_clauses)).

object definition🡪 [object identifier]: *clause*

[…]

clause 🡪 *when* [condition]: [value]

The object identifier must be unique and not redefined elsewhere in the program. The first clause must be placed on the same logical line as the object identifier.

Indentation of clauses is not required, but may contribute to readability*,* e.g.

{pulse: *when* trigger: *true*

*when* stop: *false*}

Several when clauses may be placed on the same physical line, although this is not usually recommended for readability, e.g. {pulse: *when* trigger: *true when* stop: *false*}, or equivalently {pulse *when* trigger *until* stop}.

The *condition* must be an object of [*event*](#event) nature, including logic expressions that yield a single event result and not a list.

The *value* must be compatible with the object’s nature. It is also part of the information used by Whand at compile time to infer the nature of the object (see [type inference](#_3.2._Type_inference)).

Within the definition of a given object, the *condition* of each *when* clause is expected to be different from the condition of the other clauses. The *value* of each clause may be the same or different.

### 5.2.1. Objects without definition

Not all objects need to have a definition. Objects without a definition have a constant value. They may be numbers, delays, states, or lists in explicit form.

A number without a definition is the standard character representation of a numerical value, either integer or floating-point, e.g. {12345.678}. No spacing character other than a single decimal point is allowed within a number. Scientific notation, e.g. {6.02E+23} is not understood by Whand. It will be parsed instead as {(6.02\*E) + 23}.

A delay without a definition is a number without a definition followed by a time unit. Allowed time units are {ms, s, mn, min, h, day, wk}. Several delays without a definition can be concatenated into a single value, e.g. {1 wk 3day 12 h14mn 5 s} is understood as 908045s.

A state without a definition takes for value its identifier (literal representation).

[Built-in constants](#_6.1.1._Built-in_constants) (e.g. *epsilon*, *false*, *empty*) should not be defined in a program.

### 5.2.1. Simplified clauses

In the interest of readability, some clauses may be simplified according to the following equivalence rules. In particular, the column {:} between [object identifier] and *when* may be omitted.

Each of the following examples is a single physical program line. In each definition, only the first clause is preceded by [object identifier].

[object identifier] *when* [condition]: [value] 🡪 [object identifier]: *when* [condition]: [value]

[object identifier] *when* [condition] 🡪 [object identifier] *when* [condition]: *true*

[object identifier] *until* [condition] 🡪 [object identifier] *when* [condition]: *false*

[object identifier]: [value] 🡪 [object identifier]: *when* *start*: [value]

*when* *change*[value]: [value]

[object identifier] 🡪 [object identifier]: *when* *start*: *true*

For syntactic reasons, if a simplified clause does not contain either *when*, *until* or [*be*](#be), it must be the first clause in an object’s definition. In this particular case, the order of clauses has consequences.

## 5.3. Execution

A Whand program only consists of object definitions.

All objects in a Whand program function in parallel.

The clauses in each object definition fully define the behavior of this object.

when clauses: All [clauses](#clause) can be considered as being [*when* clauses](#when), with a *condition* and a *value*. During execution of a Whand program, the software monitors the clauses of all objects in parallel and [updates](#_5.5._Updating_object) them as needed.

The *when* keyword means that at the instant any *condition* in an object’s definition takes the value *true*, the corresponding clause gets activated, so that the current *value* of this clause is computed and affected to the object. This value of the object persists until the activation of a clause affects another value to the object.

Before the same clause can be activated again, its *condition* must first return to value *false*.

## 5.4. Delayed events

A delayed event is the result of the addition {+} of an event (the initial event) and a delay. The onset of the delayed event will occur after the specified delay with respect to the onset of the original event. The offset of the delayed event will occur after the specified delay with respect to the offset of the original event.

The onset and offset of the delayed event are controlled by the software. They are computed at the instants of onset and offset of the initial event, respectively. Modifying the delay of a delayed event after the onset of the original event has no effect on the delayed event’s onset.

Modifying the delay of a delayed event after the offset of the original event has no effect on the delayed event.

A delayed event cannot be cancelled after the onset of the original event. Function *since* can be an alternative to delayed events if cancellation is needed.

## 5.5. Updating object consequences

At regular intervals, specified by parameter Timestep, as well as after an input or a delayed event has been triggered, an updating process occurs. The conditions of all objects that refer to the triggered object are checked. If a condition is found to be *true*, the appropriate value is computed and affected to the object.

During a single time step, this process continues iteratively until all affected objects have been updated.

Whand does not guarantee any order of evaluation within a time step. This is especially true for expressions, because expressions are internally broken down into component objects that may be updated in any order. This may lead to the problem called a *race condition* if a clause relies on an expression that has been only partly updated.

Race conditions are one of the main difficulty in Whand. They are likely to occur at the instant an object changes value. Race conditions may lead to erratic behaviors, even with apparently simple objects, e.g. {*when* x=1 *and* f(x)=3} may cause a race condition if (x=1) has been updated but (f(x)=3) has not yet been updated at the moment the expression is evaluated. In this case, it is possible to avoid the race condition using {*when* (x=1)+*epsilon* *and* f(x)=3}. The [*epsilon*](#epsilon) delay is negligible in execution time, but it guarantees that f(x) has been properly updated when the expression is evaluated.

Function [*old*](#old) is required to prevent an infinite updating loop when an object is modified according to its previous value, e.g. when incrementing or decrementing a value, or when adding ([*add*](#add)) or removing ([*pick*](#pick)) elements from a list.

## 5.6. Procedural programming

Whand is oriented toward the parallel execution of objects. It cares about timing, not sequences. There is no explicit flow of control nor conditional statements, since all clauses are evaluated at all times.

An equivalent of loops on lists is provided by [distributivity](#_4.1.6._List_distributivity). This powerful property allows for instance to multiply all elements of a list L by a number or a delay x in a single operation {L\*x}.

If a group of operations need to be performed sequentially, one way is to delay each of them by a certain multiple of [*epsilon*](#epsilon).

Another possibility is to attribute an execution time to each operation. One might then create a kind of clock event to synchronize them, e.g.

clock: *when* trigger

*when* clock+period and *count* clock<max\_number

*until* clock+*epsilon*

Each operation may then be performed when *count*(clock) reaches a particular value.

Other solutions may involve cycling through a list at each clock onset.

# 6. Functions

## 6.1 Built-in functions and constants

Built-in functions are keywords. They should be typed exactly as in the table below. Built-in functions should never be redefined in the program.

Some functions have zero arguments (built-in constants). They never change value during execution.

If a function has one argument, argument is placed after the function identifier, usually between parentheses, e.g. {*count*(my\_list)}, or equivalently {*count* my\_list}. For function [*old*](#old), argument is optional.

If a function has two arguments, first argument is placed before the function identifier and second argument is placed after, e.g. {4 *isin* (8,4,2)}. Parentheses may be needed for disambiguation. For functions [*pick*](#pick) and [*sort*](#sort), first argument is optional.

### 6.1.1. Built-in constants

[empty](#empty) [epsilon](#epsilon) [false](#false) [true](#true)

[monday … sunday](#monday) [january … december](#january)

epsilon: Constant of delay nature. The duration of *epsilon* is assumed to be very brief, shorter than any real delay used in a program. *epsilon* and multiples of *epsilon* are used to sequence operations and to avoid simultaneous value transitions that might result in [race conditions](#_5.5._Updating_object). The actual duration of *epsilon* is set by parameter Epsilon\_time.

empty: constant of list nature. A list containing zero element. Useful to initialize an incrementing list. Using function *empty* as value in a *store* operation resets and empties the output file so that it can be rewritten.

false: Constant of event nature.

true: Constant of event nature.

monday, tuesday … sunday, january, february … december: Constants of state nature. Can be used with [absolute-time functions](#_6.1.7._Absolute_time) *dayis* and *weekis*, e.g. {*dayis*(*friday*)} or {*day is* *friday*} to yield an event.

### 6.1.2. List of built-in functions

[absv](#absv) [add](#add) [alea](#alea) [all](#all) [any](#any)

[begin](#begin) [change](#change) [count](#count) [cumul](#cumul) [dateis](#dateis)

[dayis](#dayis) [end](#end) [find](#find) [hasvalue](#hasvalue) [have](#have)

[inter](#inter) [intg](#intg) [isin](#isin) [lasted](#lasted) [listfind](#listfind)

[logd](#logd) [match](#match) [next](#next) [occur](#occur) [order](#order)

[pick](#pick) [pointer](#pointer) [powr](#powr) [proba](#proba) [ramp](#ramp)

[sequence](#sequence) [shuffle](#shuffle) [since](#since) [sort](#sort) [sqrt](#sqrt)

[start](#start) [steps](#steps) [time](#time) [timeis](#timeis) [to](#to)

[weekis](#weekis) [within](#within)

### 6.1.3. Logic functions

These functions return an event, or a list of events if argument is a list.

[Absolute time functions](#_6.1.7._Absolute_time) *dayis*, *dateis*, *timeis*, *weekis* are described separately, although they return an event.

all: This function takes one argument which must be a list of events, e.g. {*all*(my\_check\_list)}. Returns an event which is *true* when all the elements of the list are *true*, and *false* otherwise.

any: This function takes one argument which must be a list of events, e.g. {*any*(my\_triggers)}. Returns an event which is *true* when at least one of the elements of the list is *true*, and *false* otherwise.

begin: (onset) This function takes one argument which must be of event nature. It returns a very brief event (shorter than [*epsilon*](#epsilon)) at the [onset](#event) of its argument. A list of events is also allowed as argument due to distributivity. In that case, the function returns a list of *begin* events.

The real duration of *begin* events is set by parameter Glitch\_time.

end: (offset) This function takes one argument which must be of event nature. It returns a very brief event (shorter than [*epsilon*](#epsilon)) at the [offset](#event) of its argument. A list of events is also allowed as argument due to distributivity. In that case, the function returns a list of *end* events.

The real duration of *end* events is set by parameter Glitch\_time.

change: This function takes one argument of any nature. It returns a very brief event (shorter than *epsilon*, *begin* or *end*) whenever its argument changes value. When applied to a list, *change* will return *true* any time the list of elements changes, and any time the value of one of the elements in the list changes.

The real duration of *change* events is set by parameter Change\_time.

hasvalue: This function takes one argument of any nature. It returns an event which is *true* if the value of the argument can be computed, *false* otherwise.

isin: Function *isin* (also spelled {*is in*}) takes two arguments. First argument may be of any nature. Second argument must be a list. Returns an event which is *true* if the identifier of the first argument can be found in the list, *false* otherwise. If first argument is a list, the result is a list of events, due to distributivity.

match: Function *match* takes two arguments which must be of list nature. Returns an event which is *true* if at each position in the two lists, the element of the first list is equal to the element in the second list. Result is *false* otherwise, or if lists are of different sizes.

order: Function *order* takes two arguments which must be of list nature. Returns an event which is *true* if all elements of the second argument can be found in the same order but not necessarily as consecutive elements in the first argument, *false* otherwise. Repetitions of an element in the first argument are allowed even if the element is not repeated in the second argument (see also *sequence*, *within*).

proba: This function takes one argument of number nature. It returns a random event. Result is *true* with a probability given by argument, *false* otherwise. If argument is zero or negative, result is always *false*. If argument is greater than 1.0, result is always *true*.

Because function *proba* yields a different value each time it is computed, it should not be used in a condition nor in an expression, but only as alone in the value of a clause.

sequence: Function *sequence* takes two arguments which must be of list nature. It returns an event which is *true* if all elements of the second argument can be found in the same order, without repetitions, but not necessarily as consecutive elements in the first argument, *false* otherwise (see also *order*, *within*).

since: Function *since* takes two arguments. First argument must be a delay, second argument must be an event, e.g. {1 mn *since* active}. A list is also allowed as first or second argument due to distributivity. In that case, the function returns a list of *since* events.

Function *since* returns an event that becomes *true* when a time interval equal to first argument has elapsed since the last *offset* of second argument, and that becomes *false* at the *onset* of second argument. If second argument becomes *true* before the time interval has elapsed, function *since* remains *false*, and thereby avoids generating unwanted multiple [delayed events](#_5.5._Delayed_events).

n.b. Also unlike delayed events, the duration of *since* can be much longer than the duration of its second argument, e.g. {15 s *since start*} will remain *true* from 15 seconds after the start of execution to the end of execution.

Function *since* may be used to detect the absence of a particular event.

start: takes no argument. Returns a brief event of the same duration as *begin* or *end* when execution begins.

The real duration of *start* event is set by parameter Glitch\_time.

within: Function *within* takes two arguments which must be of list nature. Returns an event which is *true* if all elements of the first argument can be found in the same order and as consecutive elements in the second argument, *false* otherwise (see also *order*, *sequence*).

### 6.1.4. Event functions

These functions take events as argument, or lists of events if the result is not normally a list.

count: Whenapplied to an event, *count* returns the number of [onsets](#begin) (occurrences) of this event. Function [*count*](#count_list) can also be a [list function](#_6.1.6._List_functions).

inter: Function *inter* takes one argument of event nature, e.g. {*inter*(pulse)}. It returns a delay corresponding to the time between the last two onsets of argument. If argument has less than two onsets, function does not yield any result value, a condition that may be tested using function [*hasvalue*](#hasvalue).

lasted: Function *lasted* takes one argument of event nature, e.g. {*lasted*(pulse)}. It returns a delay corresponding to the time elapsed since the last onset of argument, or since the start of execution if argument is *true* at *start*. If argument is not *true*, and in particular after event offset, function returns 0s. Function *lasted* may be used to calculate the current duration of an ongoing event.

To test for a specific event duration, function [*since*](#since) is preferable to function *lasted*.

occur: Function *occur* takes one argument which must be of event nature, e.g. {*occur*(pulse)}. It returns a list of numbers corresponding to the instant of recent [onsets](#begin) of argument, computed as seconds since the start of execution. If argument has no onset, function yields an empty list.

The absence of onsets can also be tested with {*count*(my\_argument)=0}.

The maximum size of *occur* list is set by parameter Max\_occur\_list. If the event has more that this number of onsets, only the more recent ones are returned.

to: Function *to* takes two arguments of event nature. It returns a delay corresponding to the time between the last onset of the first argument and the last onset of the second argument, e.g. {*begin* x *to end* x}. If second argument has no onset, function does not yield any result value, a condition that may be tested using function [*hasvalue*](#hasvalue). Otherwise, the result is negative during the interval between the onsets of the two events, as the last onset of second argument is taken into account.

### 6.1.5. Mathematical functions

Unless otherwise specified, mathematical functions take a number or a delay as argument. Argument may also be a list of numbers or delays due to distributivity. In that case, the function returns a list of results. Distributivity does not apply to a function which normally returns a list, e.g. *ramp*, *alea*.

absv: Returns the absolute value, e.g. {absv(-1.5min)} yields {90.0s}.

alea: Returns a list of floating point random values taken from a uniform distribution in the range [0.0, 1.0). The argument must be a number indicating the size of the list. e.g. {alea(3)} could yield the list {0.219, 0.854, 0.477}. Distributivity does not apply to *alea*, i.e. the argument of *alea* cannot be a list.

Because function *alea* yields a different value each time it is computed, it should not be used in a condition nor in an expression, but only as alone in the value of a clause.

cumul: This function takes one argument which must be a list of numbers or a list of delays. It returns a list of the same size as argument, where each element is equal to the sum of all elements in argument up to the position of the current element, e.g. {*cumul*(1,4,7)} yields {1,5,12}.

intg: Returns the integer truncation towards zero of a number or a delay. If argument is a delay, the resulting delay is an integer number of seconds, e.g. {intg(-3.75s)} yields {-3.0s}.

logd: Returns the decimal logarithm of a number or a delay. A delay argument is first converted to seconds. Zero or negative arguments yield a warning and no result value, a condition that may be tested using function [*hasvalue*](#hasvalue).

powr: This function takes two arguments which must be numbers, e.g. {1.5 powr 4.5}. It returns the power of a number. A negative first argument with a fractional second argument yield a warning and no result value, a condition that may be tested using function *hasvalue*.

ramp: This function takes one argument which must be a number. It returns the list of ascending consecutive integers smaller or equal to argument, starting from 1, e.g. {*ramp* 5} returns {1,2,3,4,5}. Distributivity does not apply to *ramp*, i.e. the argument of *ramp* cannot be a list.

sqrt: Returns the square root of a number or a delay. A delay argument is first converted to seconds. Negative arguments yield a warning and no result value, a condition that may be tested using function [*hasvalue*](#hasvalue).

steps: This function takes one argument which must be a list of numbers or a list of delays. It is the inverse of function *cumul*. It returns a list of the same size as argument, where each element at position i is equal to the difference between element i and element i-1 in argument (or element 1 if i=1), e.g. {*steps*(3,10,15)} yields {3,7,5}.

### 6.1.6. List functions

The following functions expect at least one list argument.

The following functions are described elsewhere, although they work on lists:

Functions *cumul* and *steps* are considered [mathematical functions](#_6.1.5._Mathematical_functions).

Functions *all*, *any*, *order* and *sequence* are considered [logic functions](#_6.1.3._Logic_functions).

Functions that normally return a list do not support distributivity.

add: This function takes two arguments. At least one argument must be of list nature. If both arguments are lists, of sizes m and n, returns a list of size m+n composed of the m elements of first argument and the n elements of second argument concatenated, e.g. {(1,2) *add* (3,4,5)} yields {1,2,3,4,5}.

If first argument is a list, but second argument is not a list, returns a list of size m+1 with second argument added as a new element after the m elements of the first argument, e.g. {(1,2) *add* 3} yields {1,2,3}.

If first argument is not a list, and second argument is a list, returns a list of size n+1 with first argument added as a new element at the beginning of the second argument, e.g. {4 *add* (5,6,7)} yields {4,5,6,7}.

To add a list as a single element, the list must first be embedded into a one-element list, e.g. {(1,2) *add* ((3,4,5),)} yields {1,2,(3,4,5)} where the third element is list {(3,4,5)}.

count: This function takes one argument which must be a list of any nature, e.g. {*count*(my\_list)}. Returns a number equal to the number of elements (size) of the list.

Function [*count*](#count) can also be an [event function](#_6.1.4._Event_functions).

find: This function takes two arguments. The first argument must be of list nature. The second argument, of any nature except a list, is the target. Returns a number giving the position of the target if present as an element in the first argument, or 0 if it is not present.

If a list is given as second argument, the result is a list of numbers, due to [distributivity](#_4.1.6._List_distributivity).

If a list as a whole is the target, see functions [*listfind*](#listfind) and [*within*](#within).

have: This function takes one argument which must be a list of any nature, defined in [implicit form](#_4.1.5._List_implicit). Returns the list of objects that are used as indices in the list definition, e.g. if definition is {color(light): green

color(stop): red} then {*have* color} yields {light, stop}

listfind: This function takes two arguments. Both arguments must be of list nature. The second argument is the target. Returns the position of the target if present *as an element* in the first argument, or 0 if it is not present.

next: Extracts a single element from a list, sequentially. This function takes one argument which must be a list of any nature. The first time the function is called, it returns the first element in the list. Each time the function is called, it returns the next element in the list. After the end of the list is reached, *next* cycles through the list, i.e. it again returns the first element, then the second… The position of the last element extracted can be obtained using function [*pointer*](#pointer).

Because function *next* yields a different value each time it is computed, it should not be used in a condition nor in an expression, but only as alone in the value of a clause.

pick: (extract elements from a list). This function normally takes two arguments of list nature. The second argument must be a list of events. If two arguments are present, function *pick* returns a sublist of first argument which only contains elements in the same position in first argument as *true* elements in second argument. Thus, second argument acts as a mask to select elements in the list, e.g. if list L is {2,3,4,5}, then {L pick(L>3)} returns {4,5}. Note that {(L>3)} is computed as list {*false, false, true, true*} due to distributivity.

Function *pick* can also take a single argument if argument is a list of events. In this case, it returns the list of *true* elements in the list.

pointer: This function takes one argument which must be a list of any nature that was accessed using [*next*](#next). Function *pointer* is a consequence of function *next*. It returns the position of the last element extracted from the list using *next*.

shuffle: This function takes one argument which must be a list of any nature. It returns a list composed of the same elements in randomized order.

Because function *shuffle* yields a different value each time it is computed, it should not be used in a condition nor in an expression, but only as alone in the value of a clause.

sort: This function normally takes two arguments of list nature. The second argument must be a list of numbers, delays, states or events. If two arguments are present, function *sort* returns a sorted list of the elements of first argument, using second argument as a sorting key in ascending order, e.g. if list L is {1,2,3,4} and list K is {10,9,8,7}, then {L *sort* K} yields {4,3,2,1}. For events, *false* comes before *true*.

Function *sort* can also take a single argument if argument is a list of numbers, delays, states or events. In this case, it returns a list of the elements sorted in ascending order.

### 6.1.7. Absolute time functions

Timing in Whand is usually relative, with respect to start of execution or some other event. It is usually the result of [delayed events](#_5.4._Delayed_events) or of function [*since*](#since). The following functions instead refer to the time of day or the date.

itis: Function *itis* (also spelled {*it is*}) expects an argument which may be either

* a delay, e.g. 16h03mn. When argument matches current time since midnight, function returns *true*. Otherwise, function returns *false*. Delays exceeding 23h59mn59s yield a runtime error.
* a state from list {monday, tuesday, wednesday, thursday, friday, saturday, sunday}. If argument matches current day of week, function returns *true*. Otherwise function returns *false*.
* a state composed of a month name {january…december} followed by a number, e.g. {may 23} (spacing is optional). If argument matches current date, function returns *true*. Otherwise function returns *false*. Numbers less than 0 or greater than month length yield a runtime error.

Arguments that do not match the above templates yield a runtime error.

time: Function time expects an argument of event nature. Returns a delay representing the time of the last onset of argument with respect to midnight. If argument is true at start, returns the time of the start of execution. If argument has no onset, function does not return any value, a condition that may be tested using function *hasvalue*.

This condition can also be tested with {*count* [argument] = 0}.

### 6.1.8. I/O functions and objects

The following keywords perform input and output operations. Input keywords are built-in functions and cannot be redefined. Output keywords are built-in objects of nature [list in implicit form](#_4.1.5._List_implicit), and they must have a definition, with any *number* of when clauses.

The most basic I/O functions are single bit (event) controls provided by *pin* (input function) and *output* (output object), e.g. {*output* 2 *when* ok *and pin* 3}.

It is good practice to name inputs or outputs, e.g. {button: *pin* 1}, {*output*(4): light}. Inputs must appear as values (right of the clause) and outputs as defined names (left).

[call](#call) [command](#command) [display](#display) [key](#key) [load](#load)

[output](#output) [pin](#pin) [print](#print) [read](#read) [store](#store)

[tell](#tell) [touch](#touch) [write](#write)

call: (input function), e.g. {*call*("arct(X)"} provides an entry point to interface Whand with Python. When the value of this object is computed, it passes control to a Python routine (here named {arct}), which must be defined in whand\_io.py:

call 🡪 *call*("[routine name]([argument list])")

[argument list] is a Whand list which must correspond to the list of arguments in the routine.

The Python routine may return a value of any nature to Whand.

The Python routine can have side effects (e.g. work as a special output) besides computing a value.

Execution of all parallel Whand programs is suspended until the routine terminates and returns control to Whand, so time in the routine must be kept to a minimum.

command: (output object), e.g. {*command*(2)} can be used to output a floating point value to an analog hardware channel identified by [channel identifier]. [channel identifier], a number, must have an integer value.

command 🡪 *command*([channel identifier]) *when* [condition]: [number]

[…]

Each time [condition] becomes *true*, [number] is computed and sent to channel.

A program must provide command definitions for all identified analog hardware channels in use.

display: (output object), e.g. {*display*(image.jpg, 0, 0)} is used to display an image on screen. While the display object is *true*, the image taken from file [file identifier] is displayed at the specified coordinates [x coord], [y coord] on screen.

display 🡪 *display*([file identifier], [x coord], [y coord]) *when* [condition] *until* [condition off]

[…]

Argument [file identifier] must be of state nature and a valid image file name. Path separator is / (slash). Double quotes around the argument are ignored.

[x coord] and [y coord] must be of number nature.

The same image may be displayed multiple times and several identical or different images may be displayed at overlapping times and/or coordinates. More recent images will overlay older ones that are still displayed on the screen.

key: (input function), e.g. {*key*("s")} when computed, returns *true* when the specified keyboard key is depressed, *false* otherwise. The argument must be of state nature and consist of a single character (here: {s}). Double quotes around the argument are ignored.

load: (input function), e.g. {*load*(“myfile.txt”)} when computed, reads a list from file identified by [file identifier]. The returned result is of list nature.

load 🡪 *load* ([file identifier])

Argument [file identifier] must be a valid file name. Path separator is / (slash). Double quotes around the argument are ignored.

During the load operation, the file is opened, then the entire file is read at once, then the file is closed. Each line in the file is loaded as an element in the result list. Each line containing a list will be loaded as one element of list nature in the result list.

output: (output object), e.g. {*output*(2)} can be used to output an event value (one bit) to a hardware channel identified by [channel identifier]. [channel identifier], a number, must have an integer value.

output 🡪 *output*([channel identifier]) *when* [condition]: [event]

[…]

Each time [condition] becomes *true*, the value of the event (*true*/*false*) is computed and sent to channel as 0 (*false*) or 1 (*true*).

pin: (input function), e.g. {*pin*(3)} when computed, reads the status (one bit) of a hardware channel identified by [channel identifier].

pin 🡪 *pin*([channel identifier])

The argument [channel identifier], a number, must have an integer value. Function returns *true* when the specified channel has value 1, *false* if it has value 0.

print: (output function) equivalently {*tell*} displays data on the command console.

print 🡪 [object name] *when* [condition]: *print* [list]

Each time [condition] becomes *true*, the list of values of all elements in [list] is displayed, with a space between elements. If the value is an object identifier (a name), the value of this object is displayed instead.

This function can be used any number of times within the definition of any object. Calling it does not change the value of the object.

print: (output object) equivalently {*store*(screen)} displays data on the command console.

print 🡪 *print* *when* [condition]: [list]

[…]

Each time [condition] becomes *true*, the list of values of all elements in [list] is displayed, with a space between elements. If the value is an object identifier (a name), the value of this object is displayed instead.

Only one *print* object is allowed in a program, so all messages to be printed must be included in the definition of the *print* object.

read: (input function), e.g. {*read*(3)} when computed, reads a string of characters from a hardware channel identified by [channel identifier]. The returned result is of state nature.

read 🡪 *read*([channel identifier])

The argument [channel identifier], a number, must have an integer value.

store: (output object) e.g. {*store*(“myfile.txt”)} records data on file.

store 🡪 *store*([file identifier]) *when* [condition]: [list]

[…]

Argument [file identifier] must be a valid file name. Path separator is / (slash). Double quotes around the argument are ignored.

Each time [condition] becomes *true*, the values of all elements in [list] are *appended* to the file, separated by an end-of-line (i.e. one element per line). If the value is an object identifier (a name), the value of this object is displayed instead.

File is erased in order to be rewritten when keyword *empty* is used as [list].

If an element is itself a list, it will be stored on a single line in the form of *identifiers* separated by commas. In that case, to compute the values of elements themselves, use function [*text*](#text) on the list.

touch: (input function), e.g. {*touch*(image.jpg, 0, 0)} is linked to function *display* with a touch screen.

touch 🡪 *touch*([file identifier], [x coord], [y coord])

Argument [file identifier] must be of state nature and a valid image file name. Double quotes around the argument are ignored.

When computed, function *touch* determines whether there is a screen contact on the image displayed at the specified coordinates and returns a list of three elements: [x coord], [y coord], [touch event].

[touch event] is *true* when there is a contact on the screen *in the zone* delimited by image, *false* otherwise. [x coord] and [y coord] correspond to the x, y coordinates of contact, even if contact is outside the specified zone. If there is no contact, [x coord] and [y coord] have no value and [touch event] is *false*.

write: (output object), e.g. {*write*(2)} can be used to output a string of characters (state) to a hardware channel identified by [channel identifier]. [channel identifier], a number, must have an integer value.

write 🡪 *write*([channel identifier]) *when* [condition]: [state]

[…]

Each time [condition] becomes *true*, [state] is computed and sent to channel.

### 6.1.9. Miscellaneous functions and keywords

The following tokens, objects or functions have effects at compile time or at run time.

[be](#be) [exit](#exit) [include](#include) [lastchange](#lastchange)

[old](#old) [show](#show) [text](#text) [until](#until) [unused](#unused)

[when](#when)

be: (token) is used in a clause to specify the [nature](#_3.1._Nature_of) of an object, if its nature could not be determined by [type inference](#_3.2._Type_inference).

be 🡪 *be* [nature specifier]

This form is exactly equivalent to {*when false*: [nature specifier]}.

The clause {*when false*} never becomes *true*, so it has no effect on execution. However, [nature specifier] is used by type inference to assign a nature to the object. [nature specifier] may be any object of unambiguous nature, e.g. {0}, {1s}, {*monday*}}, {*true*}, {*empty*}, or an element of list {number, delay, state, event, list}.

Identifiers {[number](#number)}, {[delay](#delay)}, {[state](#state)}, {[event](#event)} and {[list](#list)} are not keywords and may be used as ordinary objects, but their nature is built-in and corresponds to their name.

Another way to use *be* is to reverse the order of items. With this form, several *be* clauses are allowed in the same definition, which may be for instance the definition of {number}, {delay}, {state}, {event} or {list}. The object [nature specifier] can only have one definition in a program, but other objects of the same nature may also be given a definition.

[nature specifier] *be* [object identifier]

[…]

This is particularly useful when object [object identifier] cannot have a definition, for instance when it is the result of [subscription](#_4.1.4._Subscription_with) on a [list in explicit form](#_4.1.2._List_explicit), e.g. {number *be* my\_list(1) *be* my\_list(2)} indicates that the first two elements of list {my\_list} are numbers.

Within a [user-defined function definition](#_6.2.2._Function_definition) or [accessory variable](#_6.2.3._Accessory_variables) definition, all defined objects in a *be* clause must be virtual (prime) objects. This sometimes requires tricks (see [type inference](#_3.2._Type_inference)).

A *be* clause should never be placed before a clause that does not contain *when*, *until* or *be*.

exit: (output object). When *exit* is *true*, program execution stops, after all operations occurring at the same instant have been performed.

exit 🡪 *exit* *when* [condition]

[…]

The definition of *exit* can contain any number of *when* clauses. The clause that becomes *true* first will stop execution.

include: (compile instruction). At compile time, reads a program text from file identified by [file identifier]. The returned text is included without modification, in place of the *include* line. The text of the *include* file should consist of valid Whand definitions.

include 🡪 *include* ([file identifier])

Argument [file identifier] must be a valid file name. Path separator is / (slash). Double quotes around the argument are ignored.

Unlike all object definitions, instruction *include* modifies the current program. It can be used for instance to provide definitions for standard objects, inputs, outputs or [user-defined functions](#_6.2._User-defined_functions). Text in an *include* file can be shared by multiple programs.

lastchange: Function *lastchange* expects as argument an object of any nature. It returns a number corresponding to the last time the value of the argument has changed (in seconds since *start*). If the value has not changed, no value is returned, a condition that may be tested using function *hasvalue*.

For lists, only changes in the list (identity) of elements are taken into account, and not changes in the value of elements.

old: Function *old* expects as argument an object of any nature. If *old* concerns the current object definition, argument may be omitted. Function *old* returns the value of the argument observed at the beginning of the current time step. Due to the updating process, function *old* is mandatory when objects change value depending on their previous value, e.g. incrementation: {counter *when start*: 0 *when* some\_event: *old*+1}.

show:(output object). The presence of *show* results in the display of the interactive panel. *show* specifies a list of objects that should appear in the interactive panel. {*show*(*all*)} arranges for all defined objects to appear in the interactive panel, except those indicated by *hide*. When several scripts run in parallel, the interactive panel is always displayed. The content of the interactive panel is saved on *exit*.

show 🡪 *show*([object],[…])

Comma is optional if subscript list contains only one [object] identifier.

Any number of *show* definitions are allowed, as long as they are subscripted with different [object] identifiers.

In this particular case, the order of object definitions has consequences (on display and saved output).

text: Function *text* expects as argument an object of any nature. It returns and object of state nature which is the literal representation of its argument. If argument is a list, the list of values of every element in the list is returned, instead of the list of element identifiers.

Function *text* may be used with *store*, to pack a list into a single line on file, instead of storing one element per line.

until: (token) is used to replace *when* in a clause while specifying the value as *false*.

until🡪 *until* [condition]

This form is exactly equivalent to {*when* [condition]: *false*}.

unused:(output object). Specifies a list of input or output objects that should not be active in the program. The list should only consist of pin or output identifiers, e.g. {*pin*(4), *output*(3)}

This instruction may be used for instance to verify that the program is compatible with the *include* file where inputs and/or outputs are defined.

unused 🡪 *unused* ([I/O identifier],[…])

Comma is optional if subscript list contains only one [object] identifier.

Any number of *unused* definitions are allowed, as long as they are subscripted with different I/O identifiers.

## 6.2. User-defined functions

### 6.2.1. Overview

User-defined functions are used to create multiple objects with similar definitions. They are composed of two parts, *function definition* and *function applications*. The function itself is ultimately a list of function applications, in [implicit form](#_4.1.5._List_implicit).

The function definition is not an effective component of the program, but only serves as a template for the definitions of function applications. The multiple function applications differ only by their arguments.

In its definition, a user-defined function may apply other user-defined functions, but recursivity (a function applying itself) is not allowed.

### 6.2.2. Function definition

A user-defined function definition has the same syntax as the definition of an element from a list in [implicit form](#_4.1.5._List_implicit). Any number of *when* clauses is allowed. The distinctive feature of a user-defined function is that argument names end with a prime {'} (ASCII code 39 or hex27, also called single quote or apostrophe).

Function definition 🡪 [function identifier]([virtual argument list]): when [condition]: [value]

[…]

[virtual argument list] may be a single argument, with or without comma, or a list in explicit form. The identifier of each argument in the list *must* end with a prime, e.g. {sum(x',y',z'): x'+y'+z'}.

Virtual arguments should not have a definition.

Virtual argument identifiers are local to the function, which means that the same identifier may be used in several function definitions without interference. Virtual arguments are placeholders, to be replaced at compile time by real arguments in each function application.

### 6.2.3. Accessory variables

A user-defined function may refer to accessory variables. The identifier of each accessory variable *must* end with a prime. All accessory variables *must* have a definition placed immediately after the function definition, e.g.

{# User-defined function random\_pair: returns two different integers in range [1;N']

random\_pair(N'): L'(1), L'(2) # N' is the virtual argument

L': *when* recompute: *shuffle*(source') # definition of accessory variable L'

Source': *when start*: *ramp*(N') # definition of accessory variable source'

number\_hint' *be ramp*(L'(1)+L'(2)) # trick to tell Whand L'(1) and L'(2) are numbers:

# built-in function *ramp* is not distributive and takes only an argument of number nature

# n.b. here, event recompute must be defined outside the function definition}

Accessory variables are virtual and local to the function, so the same accessory variable identifier may be used in several function definitions without interference.

In this particular case, the order of object definitions has consequences. No real object definition should be placed between the definition of a user-defined function and the definition of its accessory variables.

### 6.2.4. Function application

A user-defined function is used as a value through its application in the same way as a built-in function.

application 🡪[function identifier]([real argument list])

Real argument list provided for each application must consist of existing objects of the same nature as the virtual objects specified in the function definition. Virtual and real arguments must correspond one-to-one.

At compile time, Whand automatically creates a definition for each function application, using the template given by function definition and the real arguments, e.g. {sum(4,5,9)}.

In the definition of each application, the real arguments replace the virtual arguments given in the function definition. A new, independent set of real accessory variables is created for each application. These variables are not visible or accessible by the program, except through special naming conventions.