Whand step by step

Alain R Marchand, may 2019

Whand is a language for those who disregard programming as a modern form of magic. It provides an intuitive way to control a setup of simple automata (typically a Skinner box). The main objectives of the language are simplicity and readability.

A Whand script consists only of object definitions where the behavior of each object is expressed once and for all. Unlike most languages, Whand focuses on effects, not on procedures. The processes needed to produce the desired behavior remain largely invisible. A number of high-level functions are also available to help programming.

All objects work in parallel. Objects can be defined in arbitrary order, so it is relatively easy to add a new object to an existing script, and parts of a script may be re-used without changes in other applications. However, it must be kept in mind that Whand is rooted in time, so combining objects requires particular attention to the relative timing of events.

Whand has only one instruction type:

object_name: *when* condition: value

when other_condition: other_value # multiple *when* clauses are allowed

…

A condition is a logical expression involving object names, operators and functions. It must boil down at any instant to *true* or *false*. A value is assigned to the defined object at the precise instant when the condition becomes *true*. How long the condition itself remains *true* does not matter, except that it will not operate again without first returning to *false*. The value assigned to the object persists until some condition for this object becomes *true* and a new value is assigned.

Keywords are *italicized* here for demo purposes, but a Whand script is plain text. When creating or editing a script, it is recommended to use Notepad++ with a Whand language configuration (see installation documentation). Notepad++ color codes keywords and provides name completion.

Step 1: Elementary script

Let us consider a simple Skinner box with a lever and a food dispenser. The lever is supposedly connected to input line 1 (*i.e. pin*(1)) and the dispenser is connected to output line 1 (*i.e. output*(1)). Input and output lines are separate, so they can have the same number. The dispenser furthermore needs a 500 ms pulse. Here is a complete Whand script for this operation:

output(1): *when pin*(1): *true*
when output(1)+500ms: *false*

output and *pin* are subscripted objects (*i.e.* lists). Each input or output element like *output*(1) is a logical objects (*i.e.* event). *output*(1) is allowed as an object name, but *pin*(1), the input, is not (it may not be given a value by the script). *true* and *false* are internally defined events (permanently true and false, respectively). 1 is a number, 500ms is a delay.

Expression *output*(1)+500ms (adding a delay to an event) creates an internal Whand event like *output*(1), but delayed by 500ms. The duration of this delayed event is the same as that of the original event. It does not matter if *pin*(1) remains *true* (the lever is pressed) for some time, because it is only when it becomes *true* (*i.e.* its onset) that the condition is fulfilled. Note that the definition of *output*(1) refers to itself, but because of the added delay it refers to its past value, not its current value, so there is no indeterminacy.

This simple script demonstrates variables of four different natures: 1) events (logical objects), such as *output*(1), *pin*(1), *true* and *false* that only take values *true* and *false*, but at various moments in time.

2) numbers, such as 1, either integers or decimal numbers, positive or negative. 3) delays (times), such as 500ms or 0.5 s, expressed in various units (ms, s, min or mn, h, day, wk). 4) lists, such as *output* and *pin*, that can be subscripted to designate individual elements. There exists a fifth nature of object, state, that will be illustrated later.

Operations between objects are designed to work if they make sense. Some operations are possible between objects of different natures. Examples of allowed operations are:

- comparing numbers between them or delays between them (yields an event);
- adding or subtracting delays between them, multiplying or dividing a delay by a number (yields a delay);
- adding a delay to an event, as in *output*(1)+500ms (yields an event), but not subtracting a delay to an event because, in general, an event cannot be anticipated;
- dividing one delay by another (yields a number), but not multiplying one delay by another;

Operations on elements of lists follow the same rules.

Step 2: Naming

It is good practice to name constants, inputs and outputs, and to add comments to make the same script more readable. The following script in long form behaves as the first one:

```
pulse_duration: when start: 500ms          # named constant
reward: when lever_pressed: true           # delivers one reward
   when reward+pulse_duration: false       # pulse to trigger the food dispenser
# naming inputs and outputs
lever_pressed: when start: pin(1)          # named input
   when change(pin(1)): pin(1)
output(1): when start: reward              # named output
   when change(reward): reward
```

Indentation is not required, but helps separate definitions.

It does not matter if names are given before or after being used. For Whand, the order of instructions is indifferent.

change is an internally defined function (an event) which becomes briefly true when the value of its argument changes. The combination of *start* and *change* makes sure that the value of lever_pressed tracks that of *pin*(1), and that the value of *output*(1) tracks that of reward. 500ms is a constant so the value of pulse_duration can persist throughout the session.

Names may be freely chosen but should be as explicit as possible. They must not contain operators or spaces, unless name is enclosed in quotes. Names with subscripts (*e.g. pin*(1)) designate list elements. Upper and lower case characters in names are considered different (all keywords are lower case).

A name can only be declared once but it can be used any number of times as a value, inside values or in conditions. This is also true for lists declared as a whole, but individual list elements may be separately defined (*e.g. output*(1), *output*(2)). Again, *output* must be used on the left side (names) and *pin* must be used on the right side (values or conditions) of definitions.

Step 3: Short forms

Whand allows shorter forms for common structures. The following script is strictly equivalent to the previous ones, but more intuitive:

```
pulse_duration: 500ms                      # when start not needed here
reward when lever_pressed                  # ':' omitted before when; true is implicit
   until reward+pulse_duration             # until replaces when … false
```

```
lever_pressed: pin 1                    # when start and when change not needed
output 1: reward                        # when start and when change not needed
```

Note that several of these instruction actually omitted the *when* keyword. Omitting *when* means that the value of the term on the left will always track the value of the term on the right.

Brackets may generally be omitted if there is no ambiguity (*pin* 1, *output* 1).

Several *when* or *until* clauses may also be placed on the same line, possibly at the cost of legibility, *e.g.* 'reward *when* lever_pressed *until* reward+pulse_duration'.

Step 4: Exit condition

We can add some more meat to the script:

```
pulse_duration: 500ms                   # named constants
max_time: 20min                         # time units are ms, s, mn/min, h, day, wk
max_rewards: 30
                                        # empty lines are ignored
# body of the script – what it really does
exit when (count(reward)=max_rewards)+10s
  when start+max_time                   # whichever condition comes first
houselight                              # will be true from the start, true is implicit
reward when lever_pressed               # order of instructions is indifferent
  until reward+pulse_duration

lever_pressed: pin 1                    # named inputs and outputs
output 1: reward
output 2: houselight                    # lighting inside the Skinner box

controlpanel                            # display interactive panel
show max_time, count(lever_pressed ), count(reward)
```

exit is a special object (an event) that stops execution when it becomes *true*. It is the responsibility of the user to define *exit* conditions.

count is an internally defined function (a number) which tracks the number of onsets of an event up to current time. Formula '(*count*(reward)=max_rewards)+10s' is the sum of an event (*count*(reward)=max_rewards) and a delay (10s). Function *count* can also count elements in a list.

Instruction line *controlpanel* (a special event) anywhere in the script can be used to test a script in simulation. Variables can then be displayed on the control panel using one or several *show* instructions.

Step 5: Include

Some parts of a script can be directly re-used after being saved to file. So, the following could be saved to file "connections.txt":

```
pulse_duration: 500 ms                  # space allowed before time unit
lever_pressed: pin 1                    # named inputs and outputs
output 1: reward
output 2: houselight                    # lighting inside the Skinner box
controlpanel                            # display interactive panel
show count(lever_pressed ), count(reward)
```

and then be replaced in the script with:

```
include("connections.txt")
```

Step 6: Incrementing a counter

Suppose we do not want to reward every lever press, but to increase gradually the difficulty, by requiring first 1 press, then 4, 10, 13 etc. (progressive ratio). We will make a counter for presses.

```
include("connections.txt")                    # instruction may be placed anywhere
max_time: 20 mn
max_rewards: 30
exit when (count reward=max_rewards)+10s      # count applies to reward
   when start+max_time
houselight
reward when counter=ratio                     # use a resetting counter
   until reward+pulse_duration
counter when start: 0                         # initialize counter
   when : lever_pressed: old+1                # increment count
   when (counter=ratio)+epsilon: 0            # reset after a brief delay
ratio when start: 1                           # initialize ratio
   when (counter=ratio)+2epsilon: old+3       # increment ratio after a brief delay
```

This script highlights one main difficulty of Whand: circular dependencies. Updating of variables is instantaneous, so assigning a value of counter+1 to counter when a condition is met would make the value of counter indeterminate. Infinite update loops can be avoided using *old* and *epsilon*.

old (internally defined function) is a copy of the value of the object, here counter or ratio, made at the beginning of the current time step. This copy only changes at the end of the current time step so infinite updating is prevented. The form *old*(expression) can also be used, so no more than one *old* function is needed when specifying a value.

Similarly, *epsilon* (a very brief delay, internally defined) avoids a circular definition between counter and ratio during resetting. It takes the place of an actual delay (*e.g.* 1 ms) but *epsilon* is considered to be always shorter. Multiples of *epsilon* are allowed (*e.g.* 3*epsilon*>2*epsilon*>*epsilon*).

Note that the same condition 'counter=ratio' is used to trigger a reward, reset counter and reset ratio. The counter is only reset after a short delay, not at the instant of reward, and the ratio is incremented after another short delay. This is preferable to having ratio and/or counter depend on the reward and creating a circular dependency.

Step 7: Lists

Here is a similar script that delivers a reward after a variable number of presses, according to a predefined list named ratio_list.

```
max_time: 20 mn
max_rewards: 30
exit when (count reward=max_rewards)+10s
   when start+max_time
houselight
reward when counter= ratio                    # compare counter to list element
   until reward+pulse_duration
counter when start: 0                         # make a resetting counter
   when : lever_pressed: old+1
   when (counter=ratio)+epsilon: 0
ratio when start: next ratio_list             # initialize ratio
   when (counter=ratio)+2epsilon: next ratio_list        # extract one list element
include("connections.txt")                    # instruction may be placed anywhere
ratio_list: 5,7,3,9,2,6,8,1,4,7,5,9,3,6,4,1,8,2
```

This script illustrates the use of a list with *next*. *next* extracts each time a new element from the list, starting with the first element. A list may contain numbers, delays or any other type of value. A list is considered to be circular, so after the last sequence …1,8,2 it starts again at 5,7,3…

next should be used in its simple form *'next* ratio_list', and never in an expression or a condition.

Step 8: Manipulating lists

There is a smarter way to implement the same task using list functions. We can compare the total number of presses at any moment with a predefined list computed from the ratio list.

```
max_time: 20 mn
max_rewards: 30
exit when (count reward=max_rewards)+10s
   when start+max_time
houselight
reward when count lever_pressed is in cumul ratio_list   # everything happens here
   until reward+pulse_duration
ratio_list: 5,7,3,9,2,6,8,1,4,7,5,9,3,6,4,1,8,2 \
        5,7,3,9,2,6,8,1,4,7,5,9,3,6,4,1,8,2 \
        5,7,3,9,2,6,8,1,4,7,5,9,3,6,4,1,8,2 \
        5,7,3                                  # list continued on several lines
include("connections.txt")                     # instruction may be placed anywhere
```

Backslash '\' can be used to continue an instruction beyond the end-of-line.

is in is a simple way to determine whether a value is present anywhere in a list. The result is a logical value (event). List function *find* may be used to determine the actual position of a value in a list.

The script demonstrates one use of *cumul* (list function, internally defined). It applies to a list of numbers or of delays and returns a cumulative list (sums of all elements to the left of each position), here 5,12,15,24,26,32,40… The cumulative list *cumul*(ratio_list) has the same length as ratio_list, and would cycle back to 5,12,15… if exhausted, which would not work. For this reason, we need a full list containing max_rewards elements.

Step 9: Distributivity

Lists in Whand have a very useful property called distributivity: any operation which normally applies to single values will produce a list of values when applied to a list. Distributivity has many applications, for instance to extend a list. The following definition of ratio_list behaves just as the previous one:

```
ratio_list: (5,7,3,9,2,6,8,1,4,7,5,9,3,6,4,1,8,2)(ramp max_rewards)        # list extension
```

Function *ramp* (internally defined) generates a list of increasing numbers (1,2,3… max_rewards).

Elements are extracted from a list using the subscript as an index, so ratio_list(1) would be 5, ratio_list(4) would be 9. Because of distributivity, (5,7,3,9,2,6,8,1,4,7,5,9,3,6,4,1,8,2)(*ramp* max_rewards) returns a list using each element of *ramp* max_rewards as an index to access one element of the initial list. Thus, the new list starts just as the original one (ratio_list(1), ratio_list(2), ratio_list(3)…). However, when the index exceeds the length of the original list, the first elements are extracted again in a circular way. Thus, the new list can be much longer than the original one.

Another use of distributivity is to convert numbers to delays, and vice-versa:

```
interval_list: (5,7,3,9,2,6,8,1,4,7,5,9,3,6,4,1,8,2)*15s
```

Multiplying a number by a delay (*e.g.* 1s) converts it to a delay. Multiplying a list of numbers by a delay (here 15s) converts all numbers to delays through distributivity.

Step 10: Intervals

Suppose we want to distribute rewards at specified times, irrespective of the subject's activity.

```
include("connections.txt")                          # even if lever is not used
max_rewards: 30
exit when (count reward=max_rewards)+10s
houselight
reward when any(start+cumul interval_list)          # multiple triggers using list
   until reward+pulse_duration
interval_list: (5,7,3,9,2,6,8,1,4,7,5,9,3,6,4,1,8,2)*15s    # convert numbers to delays
```

This script illustrates the power of list operations. Whand allows long formulas such as *any*(*start*+*cumul* interval_list). Condition *any*(*start*+*cumul* interval_list) is an event that will become *true* on each of the specified moments. Note that 'reward' must be restored to *false* after pulse_duration for the next trigger to be effective.

cumul interval_list is a list of times (function *cumul* works both on numbers and on delays);

(*start*+*cumul* interval_list) uses distributivity to convert a brief event (*start*) into a list of delayed events by adding a list of delays to this single event.

any(*start*+*cumul* interval_list) converts the list of delayed events to a single event (a condition cannot be a list). This expression is *true* whenever at least one of the delayed events in the list is *true*.

Step 11: Lists of lists

Suppose we want to distribute rewards at various times during stimuli. We can use a list of lists.

```
include("connections.txt")                          # even if lever is not used
max_rewards: 30
exit when (count reward=max_rewards)+10s
houselight

stimulus_duration: 30s
stimulus_list: 5min, 10min, 20min, 25 min           # instants, not intervals
stimulus: when any(start+stimulus_list)             # multiple triggers using list, no cumul
   until stimulus+stimulus_duration

reward when any(stimulus+intervals*1s)              # multiple triggers using list, no cumul
   until reward+pulse_duration
intervals: when start: next interval_list           # extract an element which is a list
   when end stimulus: next interval_list            # not before stimulus is finished
interval_list: (9,22), (7,19), (15,27), (3,14)      # always less than stimulus duration
```

Step 12: Print and file operations

Up to now, we have defined lists explicitly in the script. However, it is often convenient to read a long list from a file. Function *load* reads a text file into a list, not necessarily at *start*, e.g. 'mylist: *when start*: *load*(myfile.txt)'. A path may be specified before the file name, *e.g.* '*load*(mypath\myfile.txt)' or '*load*("mypath\myfile.txt")'. Function *load* expects values stored one per line. If several values, separated by commas, are present on the same line, they will be considered together as a list element of the larger list. Function *load* should only be used in values, never in conditions, nor in a clause without *when*.

A special list object (*store*) performs the opposite of *load*, *e.g. 'store*(my_file.txt): when condition: value'. It appends values at specified moments to the specified file. Multiple clauses and values are allowed. To erase the file, *store* the value '*empty*'. To save an empty list to file, *store* the value '(,)' or the name of the empty list. To save a list on a single line, *store* the value '*text*(list_name)'.

print is the same as *store*(screen), *e.g.* '*print*: *when start*+1s: "Hello World!"'. It prints values to screen at specified moments. Multiple clauses and values are allowed, but only one *print* definition.

Step 13: Variable intervals

Suppose we want to space reward distributions while still requiring lever presses. The lever becomes inactive for a variable interval following each reward. The first lever press after the inactive interval is rewarded. The subject is not informed when the inactive interval ends.

```
include("connections.txt")
max_time: 20 mn
max_rewards: 30
exit when (count reward=max_rewards)+10s
   when start+max_time
houselight
reward when active and begin lever_pressed          # no lever press in advance
   until reward+pulse_duration
interval when start: 0
   when end active: next interval_list              # before computing new interval
active: when start                                  # allow reward at start
   until reward+epsilon                             # ends just after reward
   when interval since(end(active)+epsilon)         # starts again after interval
interval_list: (5,7,3,9,2,6,8,1,4,7,5,9,3,6,4,1,8,2)*3s   # convert numbers to delays
```

Functions *begin* and *end* are brief events indicating the onset and offset of an event, respectively.

This script illustrates another difficulty of Whand: variable delays require particular care. In this example, the circular dependency between 'active', 'reward' and 'interval' cannot be avoided. 'active' must end only after 'reward' is activated. Furthermore, 'interval' determines the next occurrence of 'active' and must be changed after each reward but before computing the next occurrence of 'active'. Here, 'active' becomes false just after 'reward' (reward+*epsilon*), and this is also exactly when 'interval' is updated (*end* active). Then the next occurrence of 'active' is computed after 'interval' has been updated (*end*(active)+*epsilon*).

Operator *since* counts time from the offset of an event (not from its onset). Formula 'interval *since*(*end*(active)+*epsilon*)' becomes *true* after 'interval' and remains *true* until the next event onset (*end*(active)+*epsilon*). Its duration can be much longer than the event itself. Thus '*start*+1mn' would be a brief event, but '1mn *since start*' would be *true* for the rest of the session.

The *since* construction should be preferred to adding a delay if delays change or if there are multiple triggering events. This is because each change in event or delay results in the creation of a delayed event that may not be desired. With *since*, the delay is reset whenever a new event occurs. Only the last event before the delay expires gives rise to a delayed event, so there is less risk of interference.

Note that a condition such as (active *and* lever_pressed) would be symmetrical, but (active *and begin* lever_pressed) is asymmetrical because of *begin*. This means that the condition will not evaluate to *true* if lever starts to be pressed before the active period.

Step 14: States

File names, text between "quotes" and all objects without definition are considered to be states. State names can be used to differentiate periods during a session. Suppose we want a task made of

separate trials where the subject must systematically alternate between two levers. Choice time is limited to 5s:

```
pulse_duration: 500 ms
max_rewards: 30
max_time: 30 mn
pre_trial_duration: 10s
signal_duration: 2s
choice_duration: 5s
exit when (count reward=max_rewards)+10s
   when start+max_time
houselight

target: when start: left                                          # two states: left or right
   when (target is left)+epsilon and begin left_lever: right      # do not end state too soon
   when (target is left)+epsilon and begin right_lever: right
   when (target is right)+epsilon and begin right_lever: left     # switch whatever the choice
   when (target is right)+epsilon and begin left_lever: left

trial_stage: when start: pre_trial                                # several states
   when (trial_stage is pre_trial)+pre_trial_duration: warning
   when (trial_stage is warning)+signal_duration: choice
   when (trial_stage is choice)+choice_duration: pre_trial        # if no response
   when (trial_stage is choice)+epsilon and begin left_lever: pre_trial     # delay before change
   when (trial_stage is choice)+epsilon and begin right_lever: pre_trial    # after any choice

signal: trial_stage is warning                                    # for the duration of the stage
present_levers: trial_stage is choice                             # for the duration of the stage
reward when trial_stage is choice and correct_action
   until reward+pulse_duration
correct_action: when target is left and begin left_lever
   when target is right and begin right_lever
   until correct_action +epsilon                                  # brief event

left_lever: pin 1                                                 # or put in connection file
right_lever: pin 2
output 1: reward
output 2: houselight
output 3: present_levers                                          # levers retract when false
output 4: signal
```

States are just names. Their value can be tested with *is*. You cannot test a state and change it during the same time step. This is why a delay like signal_duration or *epsilon* is needed.

Step 15: Natures

Specifying the nature of objects is not usually necessary. If Whand asks for it, one may use the *be* instruction, which actually means '*when false*:' Instruction '*when false*:' is never executed, but disambiguates object nature. It should be followed by an object of obvious nature such as 'number', 'delay', 'event', 'state' or 'list' (or 0, 1s, *false, ""*, *empty,* etc.).

```
object1: L(5)
      be number
```

The *be* instruction can also be reversed (only once for each nature) as follows:

event: *be* object2 *be* object3

The latter is particularly useful for objects that have no definition, such as subscripted list elements.

Because *be* is another form for a *when* instruction, several *be* instructions may be placed on the same line. *be* should never be placed before an instruction where *when* has been omitted.

Step 16: More list functions

Suppose we want to define a performance threshold in the previous task. For instance, over a number of successive trials (window) there should be at least a certain number (criterion) of correct choices (rewards). We can use advanced list functions to create a cyclic list of events (corresponding to each trial) and counts the number of *true* events (correct choices) in this cyclic list.
The following instructions may be added:

```
window: 30                                  # nb. of successive trials for criterion
criterion: 21                               # good trials required in window
performance: when count(cycle pick cycle)>=criterion
   until performance+epsilon
cycle: when start: empty                    # cyclic list: start with an empty list
   when (end present_levers)+epsilon and old(count(cycle)<window): \
                              reward add old cycle
   when (end present_levers)+epsilon and old(count(cycle)>=window): \
                              reward add old(cycle(ramp(window-1)))
   when performance+epsilon: empty          # reset not too soon after criterion is reached
```

The end of a trial (when a choice has been made) is indicated by '*end* present_levers'. If the choice is correct, then 'reward' becomes *true* just after this event. If the choice is incorrect, then 'reward' remains *false*. Thus 'reward' is the event we want to put in the cyclic list.
We want the cyclic list to contain at least 'criterion' *true* values. Formula 'cycle *pick* cycle' extracts a list that contains only the *true* values from 'cycle' so we can *count* the number of values in this list and compare this number to 'criterion'. The first argument ('cycle') of *pick* designates the list from which to extract values, and the second argument (here also 'cycle') designates a mask, *i.e.* a list of events (*true*/*false*) indicating whether or not the corresponding value should be extracted. There is actually a shorter form for 'cycle *pick* cycle', where the two arguments are the same: '*pick* cycle'.
The cyclic list ('cycle') is constructed after each trial by *add*-ing the 'reward' (*true*/*false*) of the trial at the beginning of the list (*add* can work on both ends with lists and elements). If the size of the list is less than 'window', then the new 'reward' event is simply *add*-ed. Otherwise, the size of the list is limited by 'cycle(*ramp*(window-1))' which selects only the first 'window-1' elements of 'cycle'. Function *old* is necessary because we are modifying list 'cycle' itself during the current time step.

Step 17: Sequencing

There is no explicit loop structure in Whand. States, list operations and distributivity take care of a number of problems, but some operations need to be performed in a strict sequence. It that case, we can build a clock that delivers impulses to trigger each action. Here is a theoretical example:

```
action1_duration: 120ms
action2_duration: 350ms
action5_duration: 80ms
clock: when start                           # a brief pulse every 200 ms
   when clock+200ms
   until clock+epsilon
step: (ramp 5)(count clock)                 # runs numbers 1..5 through a cycle in 1s
action1: when step=1
   until action1+ action1_duration
```

```
action2: when step=2
   until action2+ action2_duration          # note that action2 lasts more than one step
action5: when step=5
   until action5+ action5_duration
output 1: action1
output 2: action2
output 3: action5
```

Another way would be to use the beginning of each action to trigger the next one, but this can sometimes get confusing.

Note that actual clock accuracy is about 25ms, but specifying shorter delays still defines the order in which operations will be performed.

Step 18: Operators and functions

Operators usually take two arguments, one on each side of the operator.

Comparison operators:

The result of a comparison is an event ('true'/'false') or a list of events if distributivity applies.
'=' and '!=' determine whether numbers or delays are equal, within an accuracy limit. They also apply to events ('true' or 'false').
'<', '<=', '>', '>=' compare numbers or delays.
'*is*' and '*is not*' (also '*isnot*') compare states while ignoring quotes. Also apply to attribute lists, e.g. 'color(hat): green' may be tested as 'hat *is* green'.
'*match*' compares two lists one to one, according to the current value of each list element. The result is a single event (distributivity does not apply).
'*in*' tests whether a value (of any nature) is present in a list.
'*within*' tests whether a sublist is part of a list.

Logical operators and functions:

'*and*', '*or*' combine two events and yield an event with value *true* or *false*.
Functions '*not*', '*all*' and '*any*' take a single argument after the operator.
'*not*' applies to one event and yields an event of opposite value.
'*all*' applies to one list of events and tests whether all of these events are *true*.
'*any*' applies to one list of events and tests whether at least one of these events is *true*.
'*start*' is a function without arguments. It becomes briefly *true* when execution begins.

Time operators and functions:

'*to*' yields the delay between the onset of two events.
'*since*' is employed between a delay and an event. It yields 'true' when the event has terminated (off) for at least the specified delay, e.g. '30 min since start'. Use 'since not' to detect that an event has remained 'true' for at least the specified delay (you don't want to use 'since begin' which is not reset at event offset but only on the next occurrence of this event).
Function '*inter*' yields the delay between the last occurrences of a given event.

List operators:

'*add*' glues together two lists, or inserts an element left or right of the list.
'*pick*' applies to two lists, the second of which is a list of events. It yields the list of elements in the first list that have the same position as *true* elements in the second list, e.g. 'L *pick* (L>3)'. The form *pick* L is allowed if the two lists are the same, and it yields the list of *true* elements from L.
'*sort*' applies to one list or two lists. '*sort* L' creates a new list sorted in ascending order. The second list, if any, is used as a sorting key, e.g. 'L *sort* (-L)' sorts L in descending order.

'*find*' is employed between a list and a single value (or a list for distributivity). It yields the position of the first element equal to the value, or 0 if the value cannot be found.
'*listfind*' is employed between a list of lists and a second list. It yields the position of the first position of the second list if it is an element of the list of lists, or 0 if it cannot be found.

Functions take a single argument after the function name. Volatile functions are functions that give a variable result when applied to a constant argument (*e.g. alea, next*). Volatile functions cannot be employed in expressions or as conditions.

Math functions:

'*sqrt*', '*intg*', '*absv*', '*logd*', correspond to square root, integer rounding, absolute value, and log of base 10, respectively.
'*alea*' yields a random number between 0 and 1 (volatile function).
'*proba*' yields a random event which is *true* with the specified probability (volatile function).

List functions:

'*nex*t' cycles through elements in a list (volatile function).
'*count*' yields the number of onsets of an event, or the number of elements in a list.
'*ramp*' applied to a number n yields a list of integers from 1 to n.
'*cumul*' applied to numbers or delays yields a cumulative list. First element is unchanged. Last element is the sum of the entire list.
'*steps*' is the reciprocal of cumul. It yields the list of differences between successive elements.
'*pointer*' applied to a list accessed with '*next*' yields the current position in the list.
'*have*' applied to an attribute list, yields the list of indices of the list. So if 'color(hat): green', then *have*(color) will be a list containing 'hat'.
'*shuffle*' applied to a list creates a shuffled list (volatile function).

Sequencing functions:

'*order*' and '*sequence*' apply to a list of events and become 'true' when events have occurred in the order given in that list. '*order*' ignores repeats while '*sequence*' is sensitive to repeats.

Absolute time functions:

'*time*' gives the instant of last onset of an event. The result is a delay measured from 0h.
Functions '*time is*' (or '*timeis*'), 'day is' (or '*dayis*'), 'date is' (or '*dateis*'), '*week is*' (or '*weekis*') may be used to create an event that will be triggered when the specified time of day, day or week occurs. The trigger will occur only once unless explicitly re-triggered. The duration of the event will be extremely brief for *timeis*, up to 24h for *dayis* or *dateis* (starting at 0h), 7 days for *weekis* (starting on *monday* or on *january* 1), and may be shorter if the specified period has already started.
Note that there is no function referring to the current time ('now'). Whand only considers times where events occur (including delayed events).

Miscellaneous functions:

'*text*' transforms any object into a string. Future implementations may provide 'as' for formatting.
'*old*' refers to the value of an object just before the current time step.
'*change*' applies to any object to yield a very brief event when object value changes.
'*begin*' and '*end*' apply to events to yield a very brief event at onset or offset.
'*occur*' applied to an event yields a list of occurrences of this event (nb. of seconds since start).
'*lasted*' applied to a *true* event yields the delay since event onset. If possible, use '*since*' instead.
'*hasvalue*' returns a true event if the argument has a valid value, otherwise false (e.g. object not initialized, element extracted from an empty list…).

Special objects:

Special objects are not functions but events. They allow an interaction with the program:

'*exit*' terminates execution when *true*
'*controlpanel*' displays interactive panel when *true*
'*show*' e.g. '*show* a, b, c' displays variables a, b and c on the interactive panel. Several instruction *show* with different lists of variables are allowed.
'*hide*' prevents the display of variables on the interactive panel. Used with interactive inputs.
'*unused*' indicates that some inputs or outputs are not active (checks script compatibility with configuration include file)

Input functions and output objects:

Output objects are not functions. They allow an interaction with the outside world:

'*output*' event output to control the appropriate hardware
'*command*' number output to control the appropriate hardware
'*write*' text (state) output to control the appropriate hardware
'*store*' list output to file
'*print*' list output to screen
'*pin*' event function to read the appropriate hardware
'*key*' event function to read the keyboard
'*measure*' number function to read the appropriate hardware
'*read*' text (state) function to read the appropriate hardware
'*load*' list function to read a file

Step 19: User-defined functions

In addition to internally defined functions, new functions may be designed by the user. Frequently used new functions may be included in a file.

A user-defined function is identified by its virtual (ghost) arguments denoted by a prime ' at the end of their names. It may also use local ghost variables (also identified with a prime), the definition of which must immediately follow that of the function. Like an internally defined function, a user-defined function may then be called at various places with various real arguments. Names used as arguments in each new function definition are independent from the same names used anywhere else. To refer to the value of the function itself, use either its full name with arguments, or *old* (without name or arguments).

Here are some examples of user-defined functions.

repeat(n', m'): (m',)(*ramp* n') # creates a list with n' times element m' using list extension

This function may be called for instance as 'repeat(5,1)', yielding (1,1,1,1,1).

mini(L'): (*sort* L')(1) # smallest element in a list

This function may be called for instance as 'mini(5,9,3,7)', yielding 3.

maxi(L'): (*sort* L')(-1) # largest element in a list

Indices -1, -2…(-*count* L') can be used to designate elements in a list, starting from the end. These backward indices do not cycle, so L(-*count* L'-1) is not defined.

reverse(L'): L *sort* (-*ramp*(*count* L')) # reverse list order (not sorting)

ramp(*count* L') is list (1,2… *count* L'). Sorting will put the last element first, because it corresponds to the lowest sorting key -*count* L, then the one before last, etc.

mean(L'): (*cumul* L')(-1)/*count* L' # mean of a list of numbers or durations

(*cumul* L')(-1) is the last element, same as (*cumul* L)(*count* L), *i.e.* the sum of all elements.

frequency(e', span'): *count* e'–*count*(e'+span') # counts event e' in a running window span'

This remarkable formula performs an apparently complex operation. Counting occurrences of a delayed event e'+span', where span' is a delay, will miss all events that have occurred less than span' time before current time, because the delayed events will not yet have occurred. Subtracting this count from *count* e' yields the number of such events, an estimate of the instantaneous frequency of event e'.

unique(L'): *old*(L' *pick* ((L' *find* L')=*ramp*(*count* L'))) # make a list from L' without duplicates

This is another example of powerful Whand formulas. (L' *find* L') uses distributivity to obtain the position of the *first* occurrence of each element in the list. If a position in this list does not match *ramp*(*count* L'), it means that the element has already occurred in the list, and the corresponding logical value will be *false*. So the formula picks only the first occurrence of each element.

old is required in case the result is intended to replace the original list. A single *old* is sufficient.

replace(L', element', here'): L(*ramp*(here'-1)) *add* element' *add* L(here'+*ramp*(*count* L'-here'))

insert(L', element', here'): L(*ramp*(here'-1)) *add* element' *add* L(here'-1+*ramp*(*count* L'-here'+1))

The examples given above are clauses without *when*, but one or several *when* clauses are allowed:

timing(trigger', stop'): *when start or* trigger': 0s
when stop': trigger' *to* stop'

This function yields 0s until the 'stop' event occurs, then gives the delay between trigger' and stop'. It is reset by trigger' and can be used to time successive periods.