Table des matières

1. Whand specificities 2
   1.1. Goal-driven vs. procedural languages 2
   1.2. States and transitions 4
   1.3. Timing and delays 5
   1.4. Lists and loops 6
   1.5. Summary 7
2. Whand components 8
   2.1. Principles 8
   2.2. Clauses 9
   2.3. Simplified syntax 9
   2.4. Nature of objects 9
   2.5. Events 10
   2.6. Durations and numbers 10
   2.7. States 11
   2.8. Lists 11
   2.9. Inputs, outputs and special events 12
   2.10. Operators and functions 13
3. Whand constructions 15
   3.1. Delayed events 15
   3.2. Counters 16
   3.3. Distributivity 17
   3.4. User-defined functions 19
4. Writing a script 20
   4.1. Naming 20
   4.2. Specifying the problem 20
   4.3. Pavlovian conditioning 21
   4.4. Instrumental conditioning: Ratio 22
   4.5. Instrumental conditioning: Interval 23
   4.6. Instrumental conditioning: Periods 24
   4.7. Common pitfalls 26
5. The logic of Whand 27
   5.1. Object updating 27
   5.2. Limitations of Whand 27

Whand

by Alain R. J. Marchand - February 2022

With the development of domotics, anyone may have to code. This is a daunting prospect, not only because of the intrinsic complexity of the programmed task, but also because of the programming language itself. For a neophyte, syntax and boilerplate constitute a formidable barrier. Moreover, a computer code does not exactly describe a task, but rather specifies the steps required to perform the task, so the logic of the program may not be immediately apparent. Finally, instructions need to be structured in a very specific manner, often with complicated conditional statements.

The new language Whand was designed with these hurdles in mind. Specifically directed towards non-programmers, its primary aim was to control complex on and off events in a type of Skinner box or other equipment. Whand combines simplicity with efficiency, albeit on a restricted range of applications. In particular, Whand cannot manipulate character strings or documents except for reading or writing text files. However, an interface with Python is provided to extend its functions.

A Whand script focuses on the goal rather than the means, with emphasis on logic and timing operations. The syntax is highly simplified and evokes a natural language, to make it easily readable. Variable names and formulas are not limited. There is no strict order of instructions, because Whand is declarative and parallel. The language automatically manages states, transitions and delays, and provides convenient built-in functions to manipulate lists of objects.

Other types of languages have been proposed, with graphic representations, or parallel state representations. Still, their syntax often remains obscure to a non-specialist. Whand is unique in that it completely forgoes sequential programming and features a single type of instruction throughout.

1. Whand specificities

1.1. Goal-driven vs. procedural languages

Computers deal with states and actions. Objects stored in memory have constant values, or states, over some time intervals. Standard, procedural languages focus on actions that change states, but Whand considers states as goals. A state may be for instance the fact that a light is on. This goal involves actions to switch the light on or off.

Consider for instance the following specifications and how they translate into a standard procedural language (in pseudocode).

Switch on light when daylight is less than daylight threshold.
Switch off light when daylight is more than daylight threshold.

```
Procedure control_light:
    set daylight_threshold to 5 arbitrary units
    set light_state to off
    repeat: {
        measure daylight
        if daylight < daylight_threshold and light_state is off:
            switch light on
            set light_state to on
        if daylight >= daylight_threshold and light_state is on:
            switch light off
            set light_state to off
         }
```

This code appears rather complex, even though procedures to measure daylight and to switch light on or off are not detailed. We need to explicitly set up a loop to test the level of daylight and compare it to the threshold (Figure 1.1). Indentation and/or curly brackets delimit the loop. The loop indefinitely repeats. We must also keep track of the state of the light.

In Whand, all these operations are performed implicitly, essentially in one line of code:

```
light_is_on: daylight < daylight_threshold      # define state "light_is_on"

daylight_threshold: 5                           # set threshold to 5 arbitrary units
daylight: measure 2                             # input no.2 (keyword: "measure")
output 1: light_is_on                           # output no.1 (keyword: "output")
```

As can be seen, the Whand script closely resembles the initial specifications (texts after # are comments for the reader). The goal (the light is on) is stated up-front. It depends on two causes: the level of daylight (measured) and the threshold value (5 units).

The instruction "light_is_on: daylight < daylight_threshold" is equivalent to:

```
light_is_on when daylight < daylight_threshold      # defines state "light_is_on"
   until daylight >= daylight_threshold             # with keywords "when" and "until"
```

The "when" condition results in light on, the "until" condition results in light off. "when" (and its variant "until") is the fundamental building block of a Whand script.

In Whand, boilerplate is kept to a strict minimum. There is no need to specify how the goal is to be achieved. The output 1 instruction directly links the internal variable "light_is_on" to the hardware controlling the light (line 1). The measure 2 instruction assigns name "daylight" to the input from a particular measure instrument (connected to line 2). The order of the various instructions is indifferent, so they may be arranged as best for readability.

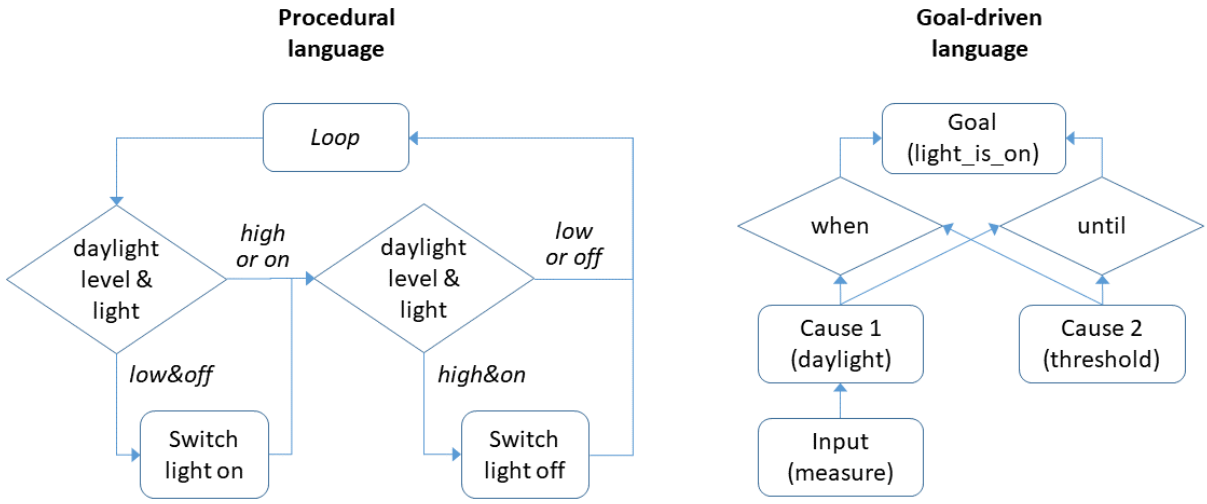

Figure 1.1. Procedural vs. goal-driven approaches. Left panel: In a procedural language, processing is sequential and follows a loop structure. The organization and order of instructions is critical. All processing takes place within a general loop which repeats indefinitely. One must test both the level of daylight and the state of the light before taking action to switch the light on or off. These actions rely on procedures which then return to the main loop. Right panel: In a goal-driven language such as Whand, the goal to switch the light on or off immediately depends on two conditions relating the two causes: daylight level and daylight threshold. The loop is implicit and the measure of daylight is permanent. The "when" condition results in light on, the "until" condition results in light off. All instructions are processed in parallel.

This simple example illustrates the profound difference in philosophy between procedural and goal-driven languages (Figure 1.1). The procedural language relies on a specific sequence of instructions. A loop structure must be explicit, whereas in Whand it is implicit. The calls to procedures light_on and

light_off constitute the actual goal, but they are somewhat buried inside the code. The state of each appliance must be tested before deciding to switch it on or off. Moreover, the code must continuously test the daylight level and cannot leave the loop. If one wants to control another appliance, the new code must be incorporated within the general loop. Of course, the loop should not stop to wait for an event because this would suspend all other processing.

The goal-driven language declares the goal (light_is_on) at the top. It then follows a top-down approach to specify causes that will result in the achievement of this goal. Each cause may be defined as a function of lower causes. The details of the implementation are left to the system, with minimal specifications. The goal is given as a state, which means that the system internally knows whether the light is on or off and whether any action is needed to change the state. No particular sequence or structure of instructions is required. Processing is parallel, so the code for another appliance can simply be added before or after the current code without any risk of interference. All of this facilitates code re-use and customization.

1.2. States and transitions

In the procedural approach, we keep track of the state of the light to test it before taking any action. This step is critical if, for instance, the goal is to order milk to replenish the fridge. One must send the order once and not repeat it while the fridge is still depleted and we wait for delivery. But the test will be performed on each iteration of the loop. Thus, our trigger must be the transition from full to depleted and not the depleted state itself.

In a procedural program, this can be performed within the general loop by explicitly comparing the current state to the preceding state and then updating the state, and/or by keeping track of the execution of the action, here milk order and delivery:

```
Procedure order_milk:
        set milk_threshold to 2
        set milk_order to none
        repeat: {
                measure milk_level
                if milk_level < milk_threshold and milk_order is none:
                        order milk
                        set milk order to pending
                if milk_delivery and milk_order is pending:
                        set milk_order to none
                }
```

How milk level is evaluated need not concern us here. Note that the state of milk_order must be explicitly changed: it is initialized to "none", set to "pending" as soon as the order is passed, and reset to "none" upon delivery. None of these steps may be omitted.

In Whand, it is not necessary to check the state of milk_order, because the "when" and "until" conditions only respond to changes. Specifically, they only respond to the transition of the specified expression, a logical event, from false to true, or in electronics terms, to a *rising edge*.

```
milk_threshold: 2
milk_ordered when milk_level < milk_threshold
   until milk_delivery
```

Here the expression controlling "when" is "milk_level < milk_threshold". The state "milk_ordered" will only pass an actual order on the rising edge of this expression, i.e. when it becomes true, and the state will remain true afterwards. There is no risk of passing another order until "milk_ordered" returns to false, upon milk_delivery. The transition to false does not trigger any action by itself, but only enables the occurrence of a new rising edge. Actually, since delivery will increase milk level above threshold, we could use an even shorter formulation:

```
milk_threshold: 2
milk_ordered: milk_level < milk_threshold
```

Here, there is not even a "when" condition. The term on the left "milk_ordered" continuously tracks the value of the expression on the right "milk_level < milk_threshold". Its rising edge will trigger the order. Admittedly, this shorter form is less explicit. Still, in both cases, the user can focus on the state of the goal and its transitions. There is no need for the precise arrangement of instructions characteristic of the procedural language.

1.3. Timing and delays

In the preceding example, note that there was a period of undefined duration between the order and the delivery of milk. Surely, beyond a certain time duration, further action may be needed to rapidly get milk. One often needs to compute time and delays in the context of automation. A delay is triggered by an event or condition, and then, after the specified time has elapsed, some action will be taken. For instance, the microwave oven will stop heating, or the light in the basement will switch off. These delays are often programmed into the hardware, but sometimes, more flexibility is needed, for instance if some action needs to be repeated at regular intervals.

Delays come in two flavors: *non-resetting delays* and *resetting delays*. A non-resetting delay ignores triggering events that occur during the delay. Microwave cooking is an example of non-resetting delay because the cooking duration is fixed once started (except when prematurely stopped). By contrast, a resetting delay computes time with respect to the last occurrence of the triggering event. A basement light controlled by a movement detector is an example of resetting delay because we want the light to persist for some duration, but also to remain on as long as movement occurs.

With procedural programming, the instruction flow must be carefully designed. One needs to refer to a clock that provides current time:

```
Procedure timers:
        set cooking_duration to 2 min
        set basement_light_duration to 30 s
        set microwave_state to off
        set basement_light_state to off
        repeat: {
                read clock_time
                if button_pressed and microwave_state is off:       # ignore button while on
                        start microwave
                        set microwave_state to on
                        store clock_time as microwave_t0
                if clock_time - microwave_t0 > cooking_duration and microwave_state is on:
                        stop microwave
                        set microwave_state to off
                if movement detected:                               # do not ignore movement
                        store clock_time as basement_t0
                        if basement_light_state is off:
                                switch on basement light
                                set basement_light_state to on
                if clock_time-basement_t0>basement_light_duration \
                                                        and basement_light_state is on:
                        switch off basement light
                        set basement_light_state to off
                }
```

Here, each timer needs its own start time memory t0. This value is then compared to the clock time on each iteration of the loop. The code also keeps track of the state of the microwave and of the basement light and uses them in the form of composed if statements.

On the other hand, Whand handles delays effortlessly:

```
cooking_duration: 2 min
microwave_on: when button_pressed                               # off at start by default
   until microwave_on + cooking_duration                        # non-resetting delay
basement_light_duration: 30 s
basement_light: when movement_detected                          # off at start by default
    until basement_light_duration since movement_detected       # resetting delay
```

Note that "microwave_on + cooking_duration" adds a delay to a logical event. This creates an internal event which is automatically delayed by the specified duration and then acts as a condition for the "when" or "until" clause. The "since" operator is similar except for the resetting mechanism.

1.4. Lists and loops

Whand has no explicit flow of control, so it does not provide any simple way to set up processing loops. Usually, this is not a serious drawback. First, the general loop that controls all objects is implicit in Whand. Then, loops in a procedural language are most frequently needed to repeat the same process on multiple items of the same collection (an array, a list, a set, etc.). Loops may also serve to repeat multiple processing steps on the same item or group of items, but this mostly concerns computing applications for which Whand is not really appropriate.

Consider for instance the problem of keeping track of depleted items in a stock. Let's assume the stock is represented as a list of item names, where the value associated with each item is the amount of that item in stock. We want to extract a list of items that are depleted, with less than a certain percentage remaining.

A procedural language would use a function like:

```
Function pick_if_less (stock as list, n as number):
        set selection to empty list
        for each element in stock do: {
                if value(element) < n:
                        append element to selection
                }
                        return selection
        }
```

This function uses a "for" loop to cycle through all elements in the list.

Whand does not need a loop to work with lists. Consider the following example where item "butter" has less than 40% remaining.

```
stock(flour): 50                    # stock values are fixed here for demonstration purposes
stock(butter): 25
depleted: (have stock) pick (stock<40)          # (have stock) returns list (flour, butter)
```

Formula for "depleted" uses internally defined functions "have" and "pick" to return list (butter,).

But the power of list processing in Whand is more than just predefined functions like "have" or "pick". It is also the fact that a list is considered as a single object when submitted to elementary operations. This property, which I call *distributivity*, is illustrated in the expression "stock<40" above:

Operator "<" would normally only compare numbers (or delays) to return a logical value, true or false. It would not be expected to work between a list and a number. But because of distributivity, the comparison is actually performed between each value in list "stock" and the number 40, so "stock<40" returns a list of logical values.

Fortunately, such a logical list is exactly what the "pick" operator expects as a second argument. Thus, "(have stock) pick (stock<40)" will use the list of logical values to return the list of elements in stock that fulfill the condition "element<40". A true value will let the element be included; a false value will exclude it.

Distributivity is a general property of operators in Whand, and it greatly simplifies the processing of list without requiring any loop structure. Moreover, all this is done while preserving the intuitive feel of the Whand code.

1.5. Summary

In brief, Whand simplifies coding by making implicit and automatic a number of functions that need to be explicit in a procedural language. Variables/objects not only have values or states but also transitions that can be directly exploited in conditions (transitions can also be referred to explicitly using function "begin", "end", or "change"). The permanent loop that monitors inputs in procedural languages is here entirely implicit. Delays are seamlessly integrated with events and do not require reference to a clock. Lists are not browsed using loops but instead are treated globally using distributivity and list functions. The result is a code that is compact yet readable and intuitive.

Whand is open source and available for download or inspection on:

https://github.com/MarchandAlain/Whand

2. Whand components

2.1. Principles

Whand is primarily designed to be readable. Several features contribute to this goal:

A Whand script is nothing but a set of *parallel* object definitions that are largely independent and may be provided in any order. This encourages a *top-down* approach where essential processes are spelled out before delving into details.

Whand starts with *behavior* and describes *causes*. It focuses on the behavior of each component, rather than on the detailed procedure needed to produce this behavior. Causes may be redesigned separately from effects without altering the general function of the script.

Objects cannot communicate with or act on other objects, but only *observe* one another. All dependencies upon other objects are explicitly specified in the object's *definition*. This greatly facilitates debugging.

Whand is rooted in time, with a single operator, *when*, in all instructions. There is no sequential processing or flow of control. Each instruction is executed at a specified moment, independently from other instructions.

Syntax, operators and functions aim at being simple and *intuitive*. Objects may be freely named to clarify their function.

```
reward when lever_pressed                    # specify causal dependencies
   until reward + pulse_duration             # specify event duration
```

In the above example, "*when* lever_pressed" and "*until* reward + pulse_duration" together constitute the *definition* of object "reward".

Whand makes entirely explicit the relationship between the behavior of a reward dispenser and its cause, a lever that must be pressed. Both "reward" and "lever_pressed" are *events*, meaning that they can take values *true* or *false* at various times. "pulse_duration" is a delay.

According to the top-down approach, "reward *when* lever_pressed" defines the main function to be performed in the script. However, we do not want the reward dispenser to remain on forever after the first lever press. It would preclude further reward deliveries or, worse, it might deliver rewards continuously. This is why the clause "*until* reward + pulse_duration" is needed to limit the duration of "reward".

The only keywords here are *when* and *until*. Names "lever_pressed" and "reward" are arbitrary and they could mean anything. The details of what exactly is a "reward" and what counts as "lever_pressed", as well as the value of "pulse_duration", are deferred to another part of the script.

Names "lever_pressed" and "reward" may be directly linked to physical inputs and outputs, or may involve more complex relationships.

```
output 2: reward                             # direct output
lever_pressed: pin 3 and (period is allowed) # input gated by a state
```

Here, the physical output #2 will be continuously controlled by the value of "reward".

The object "lever_pressed" will continuously track the value of expression "pin 3 and (period is allowed)", an input that depends both on the hardware and on a state, "period", defined elsewhere in the script.

2.2. Clauses

A Whand script specifies the behavior of various appliances, switching them ON or OFF (*true* or *false*) according to a set of clauses that may be simple or complex, based on delays, counters, states or lists. The script emphasizes the relationship between each output and input.

A Whand script is a set of object definitions. The definition of an object starts with its name followed by a set of clauses. Each definition ends where the next definition starts.

Objects are inputs, outputs or variables allowing computations. The object is characterized by its *name*, its set of *clauses*, and its *value* at any given instant.

Whand has a single instruction type: the clause

```
when condition: value
```

Keyword "when" is a sign that Whand is rooted in time. The script is not sequential. Whand will test all of the clauses all the time. An object's value changes every time one of its clauses is fulfilled, i.e. *condition* becomes *true*. At that instant, the *value* is computed and assigned to the object.

The set of clauses constitutes the definition of the object. An object can only have one definition and this definition determines all of its behavior. Objects interact only through observing one another. No object can change the value of another object.

All objects and clauses function in parallel. Definitions and clauses may be arranged in any order suitable for readability.

2.3. Simplified syntax

For convenience, some simplified syntax is allowed.

- *when* condition means "*when* condition: *true*"
- *until* condition means "*when* condition: *false*"
- name (i.e. an empty definition) means "name *when start*: *true*" (permanently *true*)
- name: value means the same as two clauses:
  name *when start*: value
  when change value: value

so the object will continuously track the value, without timing or condition. This syntax can be used to create various aliases for constants, inputs and outputs, expressions:

```
active_time: 2 min                    # alias for a constant (recommended)
mean(x'): cumul(x')(-1)/count x'      # define new user-function "mean"
```

N.b. parentheses may be replaced by spaces, except in case of ambiguity.

2.4. Nature of objects

Values are expressions based on names, operators, functions or comparators. The nature of the value determines the *nature* of the object. The nature (type) of a given object is fixed throughout a script.

There are five different natures of objects in Whand, each taking a different kind of value:

- *event*: a timeline with possible transitions between "true" and "false" values.
- *number*: a constant or variable integer or floating point number, e.g. "3.141592".
- *delay*: a duration, constant or variable, positive or negative, e.g. "-500ms".
- *state*: a constant or variable string of characters, e.g. "waiting".
- *list*: a list of names of objects of any nature, e.g. " 999, 3 min, "locked", *true* "

Conditions are expressions based on names, operators, functions or comparators. A condition must evaluate as *true* or *false* at any given moment, so a condition is necessarily an *event*.

Most of the time, Whand is able to infer the nature of objects. If an object lacks a definition and is not explicitly a number or a duration, it is considered to be a state. In particular cases, one may need to clarify the nature of an object by assigning it a value of known nature, e.g. "*when false*: *number*" (clause "*when false*" is never true, so it does not affect program execution).

2.5. Events

Events are the fundamental objects in Whand. They correspond to binary inputs, outputs or logical gates. An event can only take value true or false, that may correspond to ON or OFF for an appliance or a sensor. Events may change value repeatedly depending on their clauses.

Particular attention must be paid to the timing of events. One may visualize events by means of a graph (Figure 2.1).

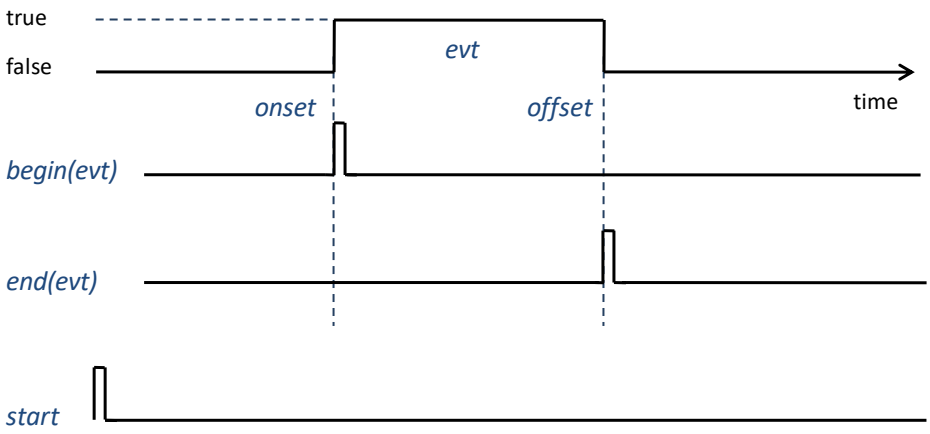

Figure 2.1. Event value (true or false), onset and offset, begin and end. Event start marks the beginning of execution of the script.

The onset or occurrence of an event is when its value changes from false to true. In a clause, the onset of the condition determines when the value is computed and assigned to the object. The value assigned to an object persists and can only change at condition onset.

The offset of an event is the change from true to false. It plays no role in a clause except allowing a later occurrence. A repeating event imperatively needs an offset clause, e.g.

```
alert when trigger: true                    # condition for "alert" onset: "trigger"
  when disable: false                       # condition for "alert" offset: "disable"
```

Or, equivalently

```
alert when trigger until disable
```

The name "alert" is arbitrary and has no particular meaning in Whand. "trigger" and "disable" are also arbitrary event names that must be defined elsewhere.

Functions *begin* and *end* create very brief events corresponding respectively to the onset and offset of an event, (Figure 2.1).

2.6. Durations and numbers

Numbers can be positive or negative, integer or floating point. Their internal representation is floating point values. When comparing numbers or durations, Whand allows for rounding errors, e.g. 1.99999995 is considered equal to 2.

Durations (delays) can be expressed in milliseconds (ms), seconds (s), minutes (min or mn), hours (h), days (day) or weeks (wk), before being internally converted to seconds, e.g.

2wk 1 day 5h 10mn 12s300 ms represent 1 314 612.3s

epsilon is a predefined internal duration. It is extremely brief, but very useful to avoid simultaneity. Simultaneity in a parallel system tends to render the order of operations unpredictable and to interfere with value updating.

Durations are quite distinct from numbers, but they can be combined provided the result is either a duration or a number. Thus, the following operations are legal in Whand:

```
1s-0.4min,      1+4.9*3,      -2.5*3.3s,      4s*12,      2mn/4,      3mn/5s   # OK
```

Conversely, the following operations will yield an error:

```
30s+1,          10s*10s,      60/1s                                    # WRONG !
```

2.7. States

States may provide a convenient temporal frame in a task, in particular to gate clauses, which otherwise would be active all the time. Unlike an event that can only take value true or false, a *state* can take several values designated by arbitrary names, e.g. "off", "standby", "choice", etc.

A variable state needs a definition (e.g. "phase"). A constant state does not need any definition. Quotes (") around a state name are optional except when the name contains spaces.

```
phase when start: standby                # "standby", "active" or "overheat" have no definition
   when on_switch: active
   when phase is in (active, overheat) and end on_switch: standby
   when phase is active and 5s since begin (phase is active): overheat
print when start or change phase: "phase is", phase
```

2.8. Lists

In Whand, a list is a collection of objects of any nature (lists of lists are allowed). A list is typically declared as a sequence of object names, separated by commas, even when composed of a single element, e.g. (one,). The current size of a list is obtained with function *count*.

Function *ramp* generates a list of consecutive integers starting at 1, e.g. "*ramp* 4" is list (1,2,3,4).

List elements may be successively retrieved using function *next*:

```
random_value: 3,5,2,9,4,1,6,8,7                    # declare a list
current_value when start: next random_value        # initialize current value to 3
   when get_a_value: next random_value             # get next value from list
```

Lists are cyclic, so here *next* may retrieve the sequence 3,5,2,9,4,1,6,8,7,3,5,2,9,4,1,6…

List elements may be addressed by index, e.g. "random_value(4)" is 9. Negative indices work backwards, e.g. "random_value(-1)" is 7 and "random_value(-3)" is 6. There is no element with index 0. The index of a list of N elements is cyclic. After index N, index N+1 corresponds again to the first element, etc.

A list may be defined element-by-element, rather than as a single sequence. Indexing information with a name is often more convenient than with a number.

```
# Three-way light switch
light_on: count pick(switch) is in (1,3)          # count switches that are on
switch(front): pin 1                              # define one element (input)
switch(middle): pin 2                             # define another element
switch(back): pin 3                               # switch list contains three elements
output(9): light_on                               # link event to hardware
```

The light is off if 0 or 2 switches are on. "pick(switch)" extracts from list "switch" the logical elements that are true (on). "front", "middle" and "back" are arbitrary names (constant states).

A new list may be built by adding or selecting elements (*add* or *pick*). A sub-list may be extracted by a list or a *ramp* of indices. *empty* represents an empty list (,) with 0 elements.

A list may be read from a text file with *load*.

2.9. Inputs, outputs and special events

Inputs and outputs are special events that allow the program to interact with hardware and the physical world.

pin designates a standard input, e.g. "*pin*(3)" or "*pin* 3". The number of available pins depends on hardware. The state of the corresponding hardware line, ON or OFF, continuously determines the value, *true* or *false*, of the *pin*. Pins are controlled externally and cannot have a definition. They are used directly when needed as values or conditions.

output designates a standard output, e.g. "*output*(4)" or "*output* 4". The number of available outputs depends on hardware. An output must have a definition that links it to a formula or an object. The computed value of this object, *true* or *false*, continuously determines the state of the corresponding hardware line, ON or OFF.

For clarity, it is recommended to use inputs and outputs through aliases.

```
output 2: light          # definition and alias for an output (keyword on the left)
                         # the behavior of object "light" must be defined elsewhere
button: pin 5            # alias for an input (keyword pin on the right)
                         # object "button" can be used anywhere in place of "pin 5"
```

Useful pieces of script may be saved to file and retrieved elsewhere with instruction *include*.

```
include "io_connections.txt"      # file of standard aliases for inputs and outputs
include "common_functions.txt"    # additional functions designed by user
```

start is a very brief special event, internally generated. It marks the beginning of execution of the Whand script, and cannot be redefined. *start* serves as a condition to initialize objects and as a time reference.

exit is recognized as a special event. When *exit* becomes *true*, it terminates the execution of the script, e.g. "*exit when start* + 30 min" determines a 30 min session. Object *exit* must have a definition, often with multiple clauses, and the first condition to occur terminates execution.

```
exit when start + 30 min          # maximum duration or
  when count trial = 50           # maximum number of trials
```

print is recognized as a special event to display information on the screen. Only one definition of object *print* is allowed, so all prints must be declared in the same block of script, each with its own condition, e.g.

```
print: "Hello World! "                          # same as    print when start: "Hello World!"
  when start + 1h: "One hour has elapsed"
  when exit: "Bye!"
  when alert: "Please check equipment!"
```

N.b. two prints may not occur simultaneously, e.g. at *start*. Like any object, *print* can only have one value at any given instant. If necessary, delay print by *epsilon*.

store is recognized as a special event, and will write to a disk file the content of a list, each element on one line. All store actions to a given file must be declared in the same block of script.

```
store("some_file.txt") when start: empty                    # erase the file before writing
   when exit: ("number of occurrences", some_counter)       # append data to file (2 lines)
```

load is recognized as a special event, and will read from a disk file the content of a list, each element from one line.

```
some_list when start: load ("some_file.txt")
```

display is recognized as a special event, and will display on screen an image read from file.

```
display(frame) when start until start+30s                   # frame is a list of 3 elements
frame(1): "some_image.gif"                                  # file containing an image
frame(2): position_X
frame(3): position_Y
```

show is recognized as a special event, e.g. "*show count*(alert)". If it appears in the script, a dashboard with a selection of objects is displayed. The user may then monitor the value of the objects and interact with them.

Summary of output and input commands (depending on hardware configuration):

Output name	expects	function
output	event	output to appropriate hardware
command	number	output to appropriate hardware
write	text (state)	output to appropriate hardware
print	list	output to screen
store	list	output to file (append)
Input name	**returns**	**function**
pin	event function	read appropriate hardware
key	event function	read the keyboard
measure	number function	read appropriate hardware
read	text (state) function	read appropriate hardware
load	list function	read a file
call	any object	interface with Python code

2.10. Operators and functions

Comparison operators:

The result of a comparison is an event (true/false) or a list of events if distributivity applies.

= , !=	equal or not, apply to events, numbers or delays within rounding errors.
> , >= , < , <=	apply to numbers or delays.
is, *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, element-by-element, by value, and yields a single event.
is 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, yield an intersection or union event.
not	yields an event of opposite value.
all	in a list of events, tests whether all of these events are true.
any	in a list of events, tests whether at least one of these events is true.
start	function without arguments. It becomes briefly true when execution begins.

Time operators and functions:

since	combines a delay and an event, defining a resetting delay.
to	yields the delay between the onset of two events.
inter	function yields the delay between the last occurrences of a given event.
lasted	yields the delay since onset if event is still true. Avoid whenever possible.

List operators:

empty	yields an empty list. If stored, erases the content of the file.
add	pastes together two lists, or inserts an element left or right of the list.
pick	selects elements from a list, using a second list as a mask.
sort	yields a sorted list, using a second list as a key.
find	yields the position of the first element equal to the value, or 0 if not found.
listfind	yields the position of a sublist in a list.

Math functions:

sqrt, *intg*, *absv*, *logd*, *powr* square root, integer rounding, absolute value, log of base 10, power.

List functions:

count	yields the number of onsets of an event, or the number of elements in a list.
ramp	yields a list of integers from 1 to n.
cumul	yields a cumulative list of numbers or delays.
steps	yields list of differences between successive elements (reciprocal of *cumul*).
have	yields list of indices of the list.

Volatile functions:

Volatile functions give a variable result and cannot be employed as conditions or in expressions.

*nex*t	cycles through elements in a list.
pointer	yields current position in the list accessed with *next*.
alea	yields a random floating-point number between 0 and 1.
proba	yields a random event true with the specified probability.
shuffle	yields a shuffled list.

Sequencing functions:

order	check that events occur in the order given in a list. Ignores repeats.
sequence	check that events occur in the order given in a list. Sensitive to repeats.

Absolute time functions:

time	gives the instant of last onset of an event, a delay measured from 0h.

it is (or *itis*)	yields an event triggered each time the specified time of day, date or weekday occurs. The duration of the event will be up to 1 s for time (e.g. "14h08min") and up to 1 day for date (e.g. "may 15") or weekday (e.g. "tuesday").

Current time (in s from midnight) would be "*time start* + *lasted not start*".

Miscellaneous:

old	refers to the value of an object at the beginning of the current time step.
change	yields a very brief event when object value changes.
begin, end	yield a very brief event at onset or offset of events.
occur	yields a list of recent occurrences of an event (list of times since start).
text	transforms any object into a string. No formatting available yet.
hasvalue	true if argument has a valid value. false for object not initialized, element extracted from an empty list…).

3. Whand constructions

3.1. Delayed events

Delays are typically used to trigger a clause. Delayed event are internally generated and triggered.

Non-resetting delay: A *non-resetting delay* is created by adding a duration to an event, e.g. "*start* + 30 min". It is appropriate when the delay needs not be prolonged, shortened or restarted (Figure 3.1). A delayed event cannot be cancelled or modified after the onset of the original event.

The delayed event occurs the specified time after the onset of the original event. The offset of the delayed event is similarly delayed, so the delayed event has the same duration as the original event.

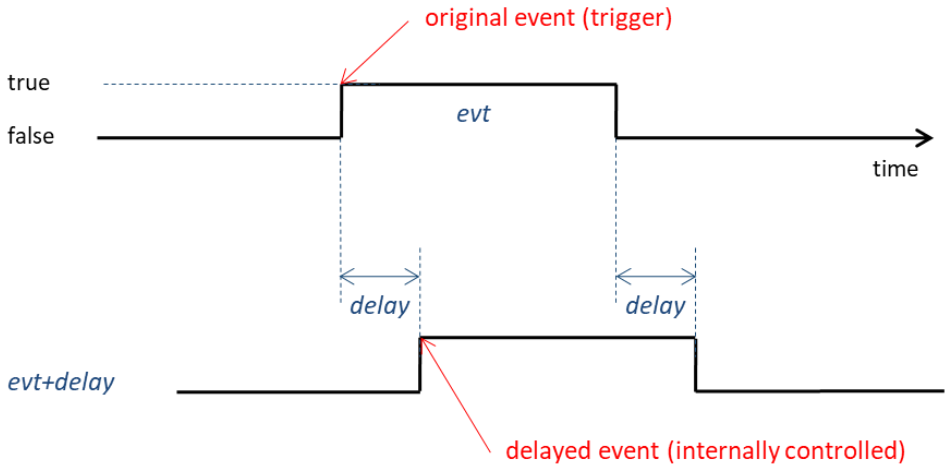

Figure 3.1. A non-resetting delay.

An event may refer to its delayed version(s) in its definition.

```
clock_tick when (start or clock_tick + 1s) until clock_tick + 50ms          # repeating ticks
```

The following structure may be applied to various periodic event.

```
# Repetitive warning sound with enable signal (non-resetting delay)
sound_duration: 1 s                                  # duration of each sound pulse
sound_period: 3.5 s                                  # interval from one pulse to the next

sound: when enable_signal
   when (sound + sound_period) and enable_signal     # auto-restart gated by enable_signal
   until sound + sound_duration
   until not enable_signal                           #    or      until end enable_signal

output 1: sound
```

Here, the delayed expression "sound + sound_period" would automatically restart the sound after each period. It is however gated by the enable signal.

Resetting delay: A *resetting delay* is created with function *since*. It is appropriate when the delay needs to be modified or restarted. The delayed event occurs the specified time after the *offset* of the original event, unless "*since begin*" is used.

If the original event re-occurs before the end of the delay, the delayed event is cancelled and the delay starts again (resets) at event offset. After the delay has fully elapsed, the delayed event remains true as long as the original event does not re-occur (indefinite duration).

```
# Cat flap with movement detector (resetting delay)

safe_time: 3s                                        # time after last movement

open when movement
   until safe_time since movement                    # resetting delay

# inputs and outputs
movement: pin(1)                                     # from movement detector
output(1): open                                      # to flap controller
```

Here, the script controls an automated cat flap that opens when movement is detected (Figure 3.2). Of course, the door should not close on the cat, so the open time cannot be fixed. The flap only closes when there has been no movement in the last "safe-time". The delay in "*since* movement" runs from the *end* of "movement".

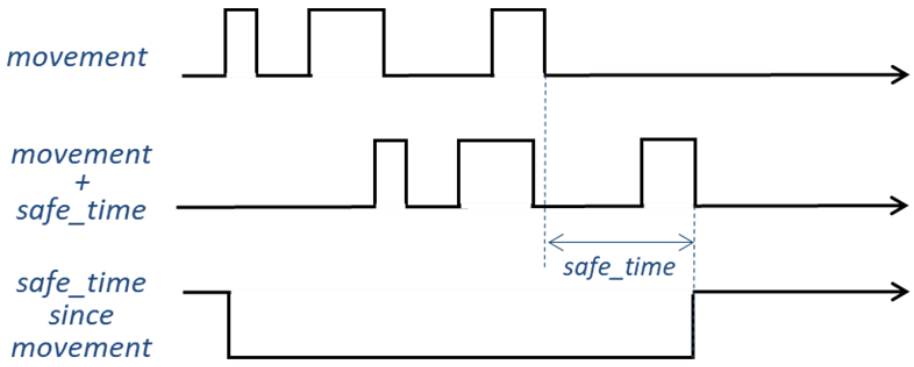

Figure 3.2. Non-resetting (movement + safe_time) and resetting delays (safe_time since movement). Both are generated in response to movement event. A non-resetting delay may result in breaks inappropriate for a cat flap.

3.2. Counters

The easiest way to count events is with function *count*. It immediately provides the total number of occurrences of an event. Combined with delays, function *count* can be extremely powerful:

```
frequency_per_minute: count contact – count(contact + 1mn)
```

Here, delayed events "contact + 1 min" are not counted until one minute after contact events. This simple expression continuously calculates the number of events that occurred in the last minute, an operation that would require elaborate computations with a procedural language.

Function *count* automatically keeps the total of all occurrences of an event since the start of execution. A disadvantage is that it is never reset to zero.

It is also possible to set up a resetting counter:

```
bottles_in_pack: when start: 0
   when add_bottle: old + 1
   when reset_counter: 0
reset_counter when (bottles_in_pack = 6) + epsilon      # allow brief delay before resetting
```

Function *old* is absolutely required to increment a counter, to add a value to a list, etc. *old* refers here to the value of "bottles_in_pack" before updating. On a given time step, "bottles_in_pack" should never be set to "bottles_in_pack + 1". It would create an infinite updating loop.

Finally, one may want to use a counter to trigger some event at intervals.

```
ratio_list: 5,3,7,8,2,6,4, 8,5,2,4,7,6,3, 6,5,7,3,4   # pseudo-rendom sequence
reward when count press is in cumul ratio_list   # when total of presses equals 5, 8, 15, 23, 25…
   until pellet + 500ms                          # standard pulse for food dispenser
```

This procedure is simpler than using a resetting counter. Here, "cumul(ratio_list)" is the cumulative list 5,8,15,23,25,31,35...

3.3. Distributivity

Lists in Whand have a very useful property called distributivity. A list can be used as a vector to perform an operation element by element, eliminating the need to explicitly loop over list elements. Any operation which normally applies to single values may be applied to a list. It will yield a list of results instead of a single result.

If the operation is applied to two lists of equal size, elements at similar positions in the two lists are combined to yield a list of results of the same size.

```
test: (1, 8, 3, 6) > (2, 5, 1, 7)          #  yields      (false, true, true, false)
all test = true                            #  yields      false
any test = true                            #  yields      true
test match (false, true, true, false)      #  yields      true
```

Distributivity has many applications:

Conversion between lists of numbers and delays:

```
interval_list: (5,7,3,9,2,6,8,1,4,7,5,9,3,6,4,1,8,2) * 15s          #  yields a list of durations
```

Slicing, extension and reordering:

A list may be used to index another list. When the list is exhausted, it is recycled from the start.

```
(a, b, c, d, e)(-2, -3, -4)                                  #  yields      (d, c, b)
(1,)(ramp 8)                                                 #  yields      (1, 1, 1, 1, 1, 1, 1, 1)
ratio_list: (5,7,3,9,2,6,8,1,4,7,5,9,3,6,4,1,8,2)(ramp max_rewards)      #  extend list
```

Burst of delayed events:

Adding the list of delays to event "trigger" yields the list of delayed events: trigger+0ms, trigger+200ms, trigger+400ms. Function *any* will become *true* every time one of these delayed events occurs. It is required because the *when* condition needs a single event, not a list.

```
burst when any(trigger + (ramp(3)-1)*200ms)      # delayed events at following instants:
until burst + 50ms                               # trigger, trigger+200ms, trigger+400ms
```

Advanced formulas:

```
L: 2,3,2,1,3,4,2,2,4                             # list with duplicates
no_duplicates: L pick ((L find L) = ramp(count L))   # list without duplicates  (2,3,1,4)
```

Here, "L *find* L" is a distributive use of *find*. It gives the first position of each element as a list (1,2,1,4,2,6,1,1,6). Comparing it to (1,2,3,4,5,6,7,8,9) from "*ramp*(*count* L)" yields (true, true, false, true, false, true, false, false, false). Finally, *pick* extracts elements of L corresponding to the positions of true values (2,3,1,4).

Queues, stacks and circular lists:

The following examples illustrate the use of particular data structures: queues, stacks and circular lists. The events used to trigger storing ("push", "pile", "new") or retrieval ("pull", "pop") are assumed to be specified elsewhere.

A queue is a memory structure where the first element retrieved is the first that was stored.

```
# queue: an item is stored on event "push" and retrieved on event "pull"
item when modify: new_value                      # item is either computed
   when pull: queue(1)                           # or retrieved as first element
queue when start: empty
   when push: old add item                       # store item as last element
   when pull + epsilon: old(1 + ramp(count old - 1))   # forget item after delay
```

A stack is a memory structure where the first element retrieved is the last that was stored.

```
# stack: an item is stored on event "pile" and retrieved on event "pop"
item when modify: new_value                      # item is either computed
   when pop: queue(-1)                           # or retrieved as last element
stack when start: empty
   when pile: old add item                       # store item as last element
   when pop + epsilon: old(ramp(count old - 1))  # forget item after delay
```

A circular list is similar to a queue, but of limited capacity. Adding a new element removes the oldest one from the list.

```
# circular list: an item is stored on event "new" and oldest item is forgotten
item when new: new_value                         # compute item
circular when start: empty
   when new + epsilon: old(1 + ramp(count old - 1)) add item # store item
```

3.4. User-defined functions

Whand allows you to define your own functions to use them in multiple places in a script. Common user-defined functions may be recorded in a text file for *include*.

A user-defined function is defined like a list element, except that the index between parentheses is an argument or a list of arguments whose names end with a prime ('). These virtual arguments are placeholders for real arguments. They are never defined and have no actual value. Similarly, ancillary objects that participate in the function definition are virtual, with names ending with a prime.

The following function computes a percentage of occurrence of two events without dividing by zero at start (not a fatal error, but Whand would give a warning).

```
percent(x', y'): 100*count x' / nonzero'                    # user-function definition
nonzero': when start: 1                                     # ancillary object immediately
   when change (count x'+count y'): count x'+count y'       # follows function definition
```

To call the function, we may for instance write "percent(success, fail)" where "success" and "fail" are real defined events. The function may be used multiple times with various real arguments.

Different user-defined functions may use the same virtual argument names and ancillary objects, so they may be stored in a file and included without interference.

User-defined functions are not recursive. They may build on other user-defined functions, but cannot call themselves directly or indirectly.

```
mean(X'): (cumul X')(-1) / count X'                         # X' is a virtual argument (a list)
variance(X'): mean(X'*X') - mean(X')*mean(X')               # variance definition builds on mean
                                  # same name X' is used in different functions without interference
```

In this definition of "mean", elements of X' may be numbers or delays, but in "variance", elements of X' must be numbers (multiplying delays is not allowed).

Some user-defined functions:

```
total(L'): (cumul L')(-1)                                   # sum of a list of numbers or durations
mini(L'): (sort L')(1)                                      # smallest element in a list
maxi(L'): (sort L')(-1)                                     # largest element in a list
between(e', f'): when e' until f'                           # duration of e' does not matter
trigger(e', intervals'): any(begin e' + cumul intervals')   # brief events following a list of delays
frequency(e', span'): count e' – count(e'+span')            # frequency of e' in a running window
lasttime(e'): time start + (occur e')(-1)*1s                # absolute time of last event onset
series(n', m'): (m',)(ramp n')                              # create list with n' times element m'
repeat(n', L'): L'(ramp(n'*count L'))                       # create list with L' repeated n' times
reverse(L'): L'(-ramp count L')                             # reverse list order
insert(L', here', x'): L'(ramp(here'-1)) add x' add L'(here'-1+ramp(count L' -here'+1))
replace(L', here', x'): L'(ramp(here'-1)) add x' add L'(here'+ramp(count L' -here'))
```

Function *old* can be used to refer to a prior value to allow in-place modification.

```
noduplicate(L'): old(L' pick ((L' find L')=ramp count L'))   # remove duplicates
          # because of "old", it is possible to assign the value noduplicate(mylist) to mylist itself
```

Keyword *old* without argument refers to the prior value of the function being defined.

```
# All-purpose clock or burst function
tick(gate', period', duration'): when gate'          # start tick with gate'
   until old + duration'                             # duration of each tick
   when gate' and begin old + period'                # restart tick after period'
   until end gate'                                   # stop when gate' is off
```

4. Writing a script

Here are a few tips on how to start programming in Whand. It is possible to edit a Whand script like any text file (e.g. with Notepad), but it is recommended to install Notepad++ with the Whand language extension. It adds helpful functions such as syntactic coloration, indenting, auto-completion and name highlighting.

There is no flow of instructions in Whand, no if, no loops. Instead, processing is fully parallel and instructions may be given in any arbitrary order. However, grouping them according to function greatly increases readability. In particular, defining at the beginning of the script aliases for parameters such as session duration, etc. helps when later trying to adapt the program.

In the following examples, the script will be created piecewise. It is not a problem since new object definitions may be added to a script without interfering with the existing objects (but avoid duplicating names). Each object functions autonomously. With instruction *include*, you can re-use common pieces of script save to file.

The top-down approach allows you to create names as you go, and to refine their function later. It is good practice to keep track of names that have not yet been defined.

4.1. Naming

A Whand program is made of objects, each designated by a unique name. Use explicit names for events, constants or variables. This makes the program easier to understand and to modify.

Each name used must have a definition somewhere (except for constant states that are nothing but names). Each name can be used in expressions anywhere, both before and after it is defined, but it can only be defined once.

You can choose almost any name for your variables, with the following constraints:

- Keywords that act as outputs, e.g. *exit*, *output*, *store*, *print*… need a definition
- Predefined Whand functions and other keywords cannot be redefined
- A name may contain a subscript, e.g. color(hat), *output*(1)
- Names cannot start with a number (because 2x means 2*x)
- Use of spaces in names is discouraged, they are interpreted as parentheses
- State names between quotes may contain spaces, e.g. “an example”
- Underscores are allowed, but some special characters such as § = : are forbidden.
- Names are case-sensitive, so keep to lower case as much as possible.

4.2. Specifying the problem

First, you need to clearly specify the task. A Whand script aims at controlling events in time. The examples below are taken from experiments with Skinner boxes.

A simple task may be to deliver particular events at specified times (e.g. in Pavlovian conditioning).

A slightly more complex question is to link outputs to inputs according to some contingencies so that an input event (action) will result in some output event (e.g. in freely-paced instrumental conditioning).

The contingencies may involve counting inputs (e.g. in ratio schedules) or adding delays (e.g. in interval schedules).

Finally, one may want to define periods where the contingencies between events change.

4.3. Pavlovian conditioning

For Pavlovian conditioning, you need a conditioned stimulus and an unconditioned stimulus that occur in a specific temporal sequence. They correspond to *events* and may be named for instance "tone" (or "stimulus", "CS" etc.) and "reward" (or "shock", "US" etc.).

Each of these events will need to be linked to a specific hardware output.

```
# I/O - hardware-dependent - may be replaced with: include("scripts/connections.txt")
output 1: tone
output 2: reward
output 3: houselight
```

These connections may also be specified in a standard file for the specific setup and retrieved with an *include* statement.

The durations of CS and US must be specified somewhere (typically at the beginning of the script)

```
# Parameters                                          n.b.  titles are only for clarity
session_duration: 11mn
tone_duration: 10s
reward_duration: 500ms
```

The relative timing of CS and US needs to be specified, e.g. for paired CS-US, the US comes at the end of the CS

```
# Procedure (US)
reward: when end tone
   until reward+reward_duration              # optional indentation to help separate definitions
```

or the CS and US may instead co-terminate

```
# Procedure (US)
reward: when tone+(tone_duration-reward_duration)   # n.b.  total delay must be positive
   until reward+reward_duration
```

Important: always specify a condition (*until*) to terminate an event that may occur more than once.

You also need some sort of delay list to specify when the CS will occur (and another list for the US if it is unpaired), e.g.

```
# Procedure (CS)
tone: when any(start+(3mn, 5mn, 6mn30s, 10mn))
   until tone+tone_duration
```

or equivalently:

```
# Procedure (CS)
tone: when any(start+moment_list)
   until tone+tone_duration
moment_list: cumul(3, 2, 1.5, 3.5)*1mn
```

The formula "*any*(*start*+moment_list)" adds a list of delays to the internal event named *start*, resulting in a list of delayed events. Keyword *any* detects whenever one of the events in the list becomes true. Each event is of the same duration as the initial event *start*, i.e. very brief. However, they only determine the onset of the tone. The duration of the tone is specified by *until*.

You also need to declare constant events such as house light

```
# Context
houselight                                    # always ON
```

Finally, you need to control session duration

```
# End of experiment
exit: when start+session_duration
```

N.b. the *exit* condition needs not be placed at the end of the script. The separate script pieces may be assembled in any order.

4.4. Instrumental conditioning: Ratio

Here, the delivery of a reward is contingent on an action from the subject, typically one or more presses on a lever. The lever is assumed to be linked to an input (in a separate connection file).

This input can be linked to an output through a simple structure

```
reward: when lever_pressed until reward+reward_duration      # clauses may be on same line
```

Sometimes a more complicated structure may be needed

```
reward: when lever_extended and begin lever_pressed until reward+reward_duration
```

This structure guarantees that the press does not begin before the lever is available ("lever_extended", to be defined and linked to an output). This is because "begin lever_pressed" is a very brief event even if the lever continues to be pressed. It can coincide with "lever_extended" only if "lever_extended" is already true when the lever gets pressed.

A ratio schedule is in effect if the subject needs to repeat an action several times (ratio) to obtain each reward. This ratio may be constant or variable around a mean value. A ratio schedule can support high response rates with a limited number of rewards per session.

The script is quite similar to the Pavlovian experiment, but it involves a different reward procedure (see 3.2. Counters).

```
# Variable ratio in a Skinner box. After a mean of 5 lever presses, a reward is delivered
# Parameters
max_rewards: 40
max_session_duration: 45min
reward_duration: 500ms                          # standard pulse duration
ratio_list: 5,3,7,8,2,6,4, 8,5,2,4,7,6,3, 6,5,7,3,4,2,8, \
            3,4,5,8,6,2,7, 4,3,8,7,2,5,6, 7,5,4,6,3   # pseudo-random sequence
# Procedure
reward when count press is in cumul ratio_list  # when total of presses equals 5, 8, 15, 23, 25…
   until reward + reward_duration               # standard pulse for food dispenser
# End of experiment
exit when count reward = max_rewards            # conditions to terminate the session
   when start + max_session_duration            # whichever happens first
# Context
houselight                                      # house light and lever are on
lever_extended                                  # for whole session
# Hardware configuration
include "scripts/connections.txt“               # define I/O events: press, reward,
                                                # houselight, lever_extended
```

It is also possible to use a resetting counter for the procedure, to the same effect

```
# Procedure
reward when counter=ratio until reward+reward_duration
counter when start: 0
   when press: old+1                            # needs 'old' to increment counter
   when reward: 0                               # immediate resetting
ratio when start or reward: next ratio_list
```

Here, there is a causal loop "counter" → "reward"→ "counter" & "ratio", but correct updating is still possible. Note that the timing of the various events is often critical and may require delays to avoid unwanted effects, e.g. a comparison performed before both terms are properly updated.

4.5. Instrumental conditioning: Interval

Another protocol, called an interval schedule requires providing a reward during an active period only, i.e. after an inactive interval following the last reward. Here, timing becomes critical.

A fixed interval schedule is relatively easy to set up with function *since*. It guarantees that the duration of the inactive interval has elapsed and can remain true for an unlimited period until the next reward occurs. The onset (*begin*) of the reward, not the offset, must be used to initiate the delay.

To be rewarded, a press must begin after, not before, the end of the inactive interval.

```
# Fixed interval in a Skinner box. After a inactive delay, a reward is delivered on the first press
# Parameters
inactive_interval: 20s
max_rewards: 40
max_session_duration: 45min
reward_duration: 500ms                    # standard pulse duration
# Procedure
reward when press and count reward = 0    # first reward without inactive period
   when begin press and inactive_interval since begin reward
   until reward + reward_duration         # standard pulse for food dispenser
# End of experiment
exit when count reward = max_rewards      # conditions to terminate the session
   when start + max_session_duration      # whichever happens first
# Context
houselight                                # house light and lever are on
lever_extended                            # for whole session
# Hardware configuration
include "scripts/connections.txt“         # define I/O events: press, reward,
                                          # houselight, lever_extended
```

A variable interval schedule is very similar, but finding the right instant to change the duration of the interval is tricky. A *since* delay starts at the end of the triggering event, here the onset of a reward, and the delay cannot be shortened after its initiation. At the exact onset of reward, the *since* event is being reset, so its value is not guaranteed. The delay must therefore be triggered after "*begin* reward+*epsilon*", and its duration must be set just before, at the onset of reward.

```
inactive_interval_list: 1s*(30,45,15,20,40,27,34)   # etc.
inactive_interval when reward: next inactive_interval_list # no value needed at start
# Procedure
reward when begin press and count reward = 0 # first reward without inactive period
   when begin press and inactive_interval since (begin reward + epsilon)
   until reward + reward_duration         # standard pulse for food dispenser
```

4.6. Instrumental conditioning: Periods

Your task may involve different periods where conditions change. State names can be used to differentiate periods. Suppose we want a task made of separate choice trials where the subject must systematically alternate between two targets. Choice time is limited to 5s. A warning signal is given before each trial.

First, we need a simple state structure to define which target is correct, left or right lever. Which target is correct at start may be defined in the parameters

```
first _target: left                       # select parameter: left or right
```

Here, "left" and "right" are just names, i.e. state constants.

The target is supposed to change after each response

```
# define target
target: when start: first_target
   when response and old is left: right          # switch target after any choice
   when response and old is right: left          # 'old' is current value of 'target'
```

A response can be a press on the left or right lever. It is preferable to consider only the onset of the response

```
response: begin left_choice or begin right_choice   # brief event, whatever the choice
```

We now need to define the different periods in the task. The transitions between periods depend on delays and on the subject's response

```
# define periods in the task
trial_stage: when start: pre_trial                                  # state names are arbitrary
   when (trial_stage is pre_trial)+pre_trial_duration: warning      # fixed delay
   when (trial_stage is warning)+signal_duration: choice            # fixed delay
   when response and old is choice: pre_trial                       # response: wait for next trial
   when time_limit since (trial_stage is choice): pre_trial         # resetting delay: no response
```

We use a resetting delay for the time limit when there is no response, because a trial where the subject responds is shorter, and a non-resetting delay would have consequences on the next trial.

Here, *old* refers to the current value of "trial_stage" before it changes. It avoids testing a state and changing it during the same time step. It is not a problem when "trial_stage" is delayed by "pre_trial_duration" or "signal_duration" because then "trial_stage" refers to a past value.

Now we need to reward responses on the target lever only

```
# reward conditions
reward when trial_stage is choice and correct_choice
   until reward + reward_duration

correct_choice when target is left and begin left_choice          # define correct choice
   when target is right and begin right_choice
   until correct_choice + epsilon                                 # brief event
```

Let's add in a few events to complete the context of the task

```
# Context
signal: trial_stage is warning                                    # for the duration of the stage
present_levers: trial_stage is choice                             # levers retracted otherwise
houselight                                                        # ON for the whole session

exit when (count reward=max_rewards)+10s                          # termination condition
   when start + max_session_duration
```

Note that several objects still have no definition. These are inputs, outputs and parameters.

```
# I/O - may be recorded in a connection file
left_ choice: pin 1
right_ choice: pin 2
output 1: reward
output 2: houselight
output 3: present_levers                    # out when true, retracted when false
output 4: signal
```

```
# Parameters
first _target: left                         # select parameter: left or right
reward_duration: 500 ms                     # standard pulse for reward dispenser
max_rewards: 30
max_session_duration: 30 mn
pre_trial_duration: 10s                     # from the end of a trial to the next trial
signal_duration: 2s                         # warning period
time_limit: 5s                              # maximum time allowed for a response
```

Of course, parameters are best placed at the beginning of the script for easy modifications.

4.7. Common pitfalls

A common error is to use a *when* condition without *until* so the event never ends. Always check the *until* condition of events. A clause without *when* or *until* ends automatically as soon as the condition ceases to be fulfilled.

Another error is to place a *when, until* or *be* clause before a clause without *when*. This error may not be detected by Whand, with serious consequences. Keyword *be* is equivalent to "*when false*:" and simply indicates an object's nature. It is best located as the last clause of a definition.

Operator *or* is not recommended as part of a condition. An expression such as "*when* A *or* B" may skip the onset of A or B if the other event is still *true*. For this reason, it is recommended to use separate conditions "*when* A", "*when* B"… unless events can never overlap.

Whenever possible, prefer functions begin or end to function change. In particular, change is always *false* at *start*, whereas *begin* detects events that are *true* at *start*. To force a change at start, use "*when start+epsilon*". Condition "*when change* some_object+some_delay" may also snowball in bizarre ways if several such conditions are used to modify the object itself. Using "*when* some_delay *since change* some_object" instead avoids this problem. Note that events that are not initialized at start may behave strangely with function *since*.

Beware of any expression involving an event in transition (onset or offset) because updating is never quite immediate (except for *begin* A, *end* A, *occur* A or *change* A). For instance, "*when* A *and count* A = n" will not work properly because the count is not updated instantly when A occurs. Instead, when A occurs, *count* A will have an ambiguous value. A consequence is that both *count* A = n-1 and *count* A = n may be recognized as true during the same time step, which is clearly incorrect.

In general, this type of problem can be circumvented by using *epsilon* to prevent simultaneous events. Thus, "*when* A+*epsilon and count* A = n" leaves time for updating the count before testing it. Similarly, function *old* should be used to make sure a value does not change during updating.

Causal loops between objects (without delays) are also particularly prone to transition problems.

5. The logic of Whand

5.1. Object updating

Before *start*, all objects are initialized as specified in their definition, and a round of updating is performed.

During updating, Whand determines which objects have changed value and uses the clauses in their definitions to determine consequences, i.e. objects that observe and depend upon each event. The clauses in each object's definition are tested. If a condition is found to become *true*, the current value of whatever object or formula is specified in the right part of the clause is computed and assigned to the object. If the value has changed, updating is extended to further consequences. All the consequences of the triggering event are updated until a stable state is reached.

Execution begins with the *start* event and ends with the *exit* event. It mimics parallel processing. Whand monitors the system clock and adjusts its current time step by step (e.g. in 50 ms steps). When current time changes, Whand scans internal events, external inputs and delayed events (delayed onsets or offsets). When any of these occurs on the current time step, updating is triggered.

Delayed events may be used to impose a sequential order during a single time step. The duration of events such as *start*, *begin*, *end* or *change* as well as duration *epsilon* are negligible compared to a time step, so "x+2**epsilon*" will be updated after "x+*epsilon*", and "x+0.1ms" will be updated after "x+100**epsilon*", but their timing will not be exact within the current time step.

If a delayed event is expected before the next time step, current time advances to the time of this delayed event and the event is updated. If no delayed event is expected, current time advances to the next time step according to the system clock.

Within a single time step, the order in which objects are updated is not guaranteed, so conditions involving objects that simultaneously change value may be unreliable.

5.2. Limitations of Whand

Whand primarily aims at controlling appliances, with states that are "on" or "off". It is comfortable with timing, and can perform simple computations, but it is not really intended for complex algorithms.

Parallelism in Whand encourages a top-down approach. But full parallelism also has constraints: All expressions and conditions are evaluated at all times, even when their value has no meaning or cannot be computed. For this reason, most runtime errors in Whand are not fatal, but mainly result in a warning message.

When a value changes, updating becomes critical. Since objects and expressions may be updated and evaluated in any order, some care must be taken not to test a value or expression just when it undergoes a transition, but only after an infinitesimal duration named *epsilon*. Similarly, referring to an object undergoing a transition requires function *old*.

Procedural operations: FOR loop

In some cases, computation must proceed according to a sequential order. This can be done in Whand but usually at the cost of complicated expressions.

For instance, assume we want to count items of different types in a list, i.e. from list (B,A,C,A,D,D,B,D,A,D) extract item types (A,B,C,D) and counts (3,2,1,4). For a single element x, we could use distributivity with "*count*(L *pick* (L=x))". However, this formula does not scale to all elements x of L because this would involve double distributivity. Instead, "*count*(L *pick* (L=L))" is just "*count* L", because distributivity between two lists operates between elements of corresponding position. It yields a list of the same size, and not a product of the two lists.

In this particular case, it is possible to count the various types of items using a contrived combination of list functions and distributivity

```
# extract counts of repeated items in a list
L: B,A,C,A,D,D,B,D,A,D
types: sort(L pick ((L find L) = ramp count L))                    # extract unique items A,B,C,D
counts: steps(((sort L find types)-1) add count L)(1+ramp count types)   # get all counts
```

This code is quite obscure and challenges all but the boldest reader (hints: "sort" sorts a list in ascending order; "L find x" lists the first position of item x in list L; "steps" computes the differences between pairs of consecutive values in a list. "add" appends a new element to a list; "(1+ramp count types)" indexes list "steps(…)" with list (2,3,4,5) to extract a sublist).

However, it is also possible to mimic a procedural loop structure in Whand. To build the result list element by element, we create a loop with a clock and index, so that conditions depend on this index. In addition, we must wait for the completion of each operation before using its result. We still use some list functions and distributivity (distr.)

```
# extract counts of repeated items in a list
L: B,A,C,A,D,D,B,D,A,D
wait1: 2*(count L + 2)*epsilon                                  # duration of unique items loop
# loop to extract unique items A,B,C,D
list_indices: ramp count L                                      # 1,2,3,…,10
clock1: any(start+list_indices*2*epsilon)                       # 10 events spaced by 2*epsilon (distr.)
L_index: count clock1                                           # index incremented on clock1
types when start: empty                                         # initialize types list as empty
   when clock1+epsilon and not(L(L_index) is in types): old add L(L_index)      # build list
   when (L_index=count L)+2*epsilon: sort old                   # finally sort types list
# loop to count items by type
type_indices: ramp count types                                  # 1,2,3,4
clock2: any(start+wait1+type_indices*2*epsilon)                 # 4 events spaced by 2*epsilon (distr.)
type_index: count clock2                                        # index incremented on clock2
counts when start: empty                                        # initialize counts list as empty
   when clock2+epsilon: old add count(L pick (L=types(type_index)))      # build list (distr.)
```

The clock and index method is a general way to set up a procedural loop. The timing of each operation must be specified by the clocks, and clock2 must start after clock1 has terminated (thus "wait1"). Unit time *epsilon* (also 2**epsilon*, etc.) allows sequencing while being short enough to run the code as fast as possible. Keyword *old* is required to designate the current object (here types or counts) before it changes value.

Procedural operations: WHILE loop

If the number of iterations in the loop is not known in advance, it may be easier to use states for sequencing. A state may summarize all the flow of control. The following example involves two nested loops of unknown length and tests whether a computed sequence finally ends with 1.

One must be very careful to allot a time step, and therefore a state, for every transition so that no value is used at the time of its transition. For instance, when state s is "get_number", number "initial" is retrieved from "number_list". It is then used to initialize "prior" and "result" when state s is "prepare". If both operations were simultaneous, the old value of "initial" might trump the new value.

Similarly, the final values of peak and length are added to their list when state s is "save", i.e. only after they have been fully updated. Also "prior" should not change during the compute state.

We touch here the limits of the language.

```
# Test of Collatz conjecture for a list of numbers
number_list: 15, 127                                    # select list of numbers here
step: epsilon                                           # select time step here
exit when s is finish

# state for sequencing
s when start: prepare
   when (s is prepare) + step: compute
   when (s is compute) + step: update
   when (s is update) + step and prior!=1: compute      # inner loop: sequence
   when (s is update) + step and prior=1: save          # end of sequence
   when (s is save) and initial=number_list(-1): finish # last number has been processed
   when (s is save)+ step: get_number                   # change initial number
   when (s is get_number) + step: prepare               # outer loop: initial numbers

# initialize run
initial when start: next number_list                    # number we want to test
   when s is get_number: next number_list               # retrieve initial number from list

# compute sequence - do not change prior during compute
prior when s is update: result                          # compute result then update
   when s is prepare: initial                           # after number has been retrieved
result when s is compute and prior-2*intg(prior/2)=0: prior/2   #  do not change prior now
   when s is compute and prior-2*intg(prior/2)>0: 3*prior+1     #  do not change prior now
   when s is prepare: initial                           # after number has been retrieved

# length and max value
length: when s is prepare: 0                            # resetting counter
   when s is compute: old+1
peak when s is prepare: initial                         # after number has been retrieved
   when s is update: (sort((old, result)))(-1)          # select larger from (peak, result)

# save one result
peak_list when start: empty
   when s is save: old add peak                         # add new result to list
length_list when start: empty
   when s is save: old add length

# display
show length, s, initial, prior, peak_list, length_list
```

In addition, current Whand architecture does not favour processing speed. For applications that require controlling events with better than about 10 ms accuracy on a single setup, one should consider more conventional languages. For the same reason, large or complex applications are not encouraged. Whand may be very awkward outside its intended range of applications.