BOXES

1. Introduction

The main idea of Boxes is that a program consists of a set of small, independent units that communicate with each other via messages. I call these units 'boxes'.

So a program basically consists of a number of boxes, where each box has:

- a name
- a type specification
- one or more message responses
- a variable or a set of variables (depending on its type specification)

send:

- a send call specifies a tag followed by a parameter list (which may be empty)
- if a tag is sent, each box that specifies a message response for that tag executes the message actions associated with the tag
- if the send call specifies too many params, excess params are simply ignored

program execution is governed by a message queue.
In the beginning the message queue only contains the send call 'start' and sends it. Sending it may cause boxes to send other tags + params where each tag + params is regarded as one entry in the queue.
If all messages have been processed and the queue is empty the send call 'final' is added to the (empty) queue and sent and the appropriate messages (if any) are processed.'Final' is only added once, so execution definitely stops after all actions/messages evoked (directly or indirectly) by 'final' are processed.

<u>comments</u>: the string // marks the beginning of a line comment

2. BNF

<program> = (box <name> <typespec> <message response>+ end)+
<message response> = on <tagname> (<action> , '|')+ off
<action> = <basic action> | <specific action>

<specific action> = <array action> | <queue action> | <machine action>
<basic action> = <simple basic action> | <special basic action>

<special basic action> = cond <cond> | save <global var> <expression> | set <var> <expression> | setrange <array var> <range expression>

<cond> = <item> | <item> <relop> <item> | <item> <relop> <cond> | '(' <cond> ')'
<item> = <constant> | <varname> | <param>
<expression> = <<mathematical expression, see below>>
<range expression> == <<range set expression, see below>>

<op> = '+' | '-' | '*' | '/' | '%' | '&'
<relop> = '<=' | '>=' | '=' | '<>' | '<', | '>'
<constant> = <<integer>> | <<string>>
<param> = #<<natural number>> | #*
note that indexing is 1-based, #* = list of all parameters
<name> = <<letter sequence>>
<tagname> = <<letter sequence>>
<dirname> ==<<char sequence denoting a directory>>
<filename> = <<letter sequence denoting a filename>>
<typespec> = <structure name> <basic type name>
<basic type name> = int | string | float
<structure name> = array | queue | scalar | triple | quintuple | septuple | decuple | machine
<var> = name of a box variable
<global var> = name of a global box variable

with the usual definitions:

|	= separates alternatives
*	= 0 or more occurences
+	= at least one occurrence
[]	= optional element
()	= groups two or more elements together

and:

(a,b)	= a followed by b if not the last in a sequence

operators are enclosed in '', but these are not written in a program

3. Details

a) Calling a program

The general format for calling a BOXES program via the interpreter is as follows:

BOXES.exe <calling options> <boxes program> <program arguments>

BOXES.exe
the name of the interpreter, including the path if necessary

<calling options>
options for calling the interpreter (may be empty). Options are introduced by a minus sign. Available options:
s<n> = scope, where <n> = l -> Lexer, p =-> Lexer + Parser, x -> all (default)
l<n> = loglevel, where <n> = i -> info, w -> warn, e -> error, f -> fatal
d<n> = debug, where <n> =
    L = debug Lexer, P -> Parser, X -> Execution,
    Q -> Queue, I -> Inits, R-> resolve varnames, C-> print chaining
    M -> message calls, o -> OPG/RAG, q -> Queue Stack
    m -> machine

<boxes program>
The program to be executed, including path if necessary

<program arguments>
arguments to the boxes program (may be empty). The arguments can be accessed from the program code as global variables:
argument<n>: where <n> =1,2,3… (number of arguments)
arguments: number of arguments
note that these variables are not protected, i.e. may be overwritten by the program code.

b) The Message Queue

A queue entry has the following structure:

- a message name
- a list of box identifiers of those boxes that respond to this message
- an index to that list, initialized with -1

- a list of parameters (string)
- a type (string or int)
- a state

Queue initialization:
When the program starts, the queue is initialized with an entry having message name ‘start’ and state ‘new’. If the program was called with parameters, these are the parameters of the message, otherwise there are no parameters.

The method for executing the queue is:

```
1.) if the queue is empty then
1a) if final hasn't been executed then add entry to queue with message
name 'final' and state 'new'
1.b) else stop
2.) find the first list entry
3.) if the identifier list is empty:
        if the state is ‚new‘ then get the list of box identifiers that respond and
                set the state to ’init(ialized)’
        otherwise delete the entry and go to (1.)
4.) advance the index
        if index > last box identifier index then delete the entry and go to (1.)
5.) execute the message for the selected box identifier
6.) goto (1.)
```

c) Actions – general remarks

- all actions may be made conditional by appending .if or .ifnot. In this case, the top of the condition queue is evaluated and the action is only executed if it evaluates to true (for .if) or false (for .ifnot)
- some actions may be augmented by additional settings which are appended to the action name, separated by a dot. You may start an augmentation with a '!' introducing a variable name that is to be evaluated to determine the true augmentation.
- You may use placeholders to denote characters that would otherwise be misinterpreted. These include: @space = single space, @bar = |, @hash = #, @none = "" . @gvars returns a list of the currently existing global variables.
  Note that – depending on the action – there may also be other placeholders. These are explained in the appropriate section.
- While some action arguments are resolved (i.e. it is checked whether the argument is a valid variable and if so, its content is used rather than its name) others are not resolved. Resolved arguments are indicated by using double angle brackets.
- $<var> designates the content of global variable <var>. You may force resolving the string contained in <var> by preceding it with '!' like so: $!<var>.

d) The Basic Actions

i. branch <<tag>>

these are simply syntactic sugar for executing:

branch <<tag>>	= send.ifnot <tag> | if
branch.not <<tag>>	= send.if <tag> | ifnot

you may view this action as defining the else-branch where the if-branch means going on. see however, the handling in arrays below.

ii. break[.next]

ends the execution of the current message response

<u>if aug=next</u>
if the current message is executed by do, then not only the current message is ended but also the calling message.

iii. cond[.int|.float|.string|.mixed] <<condition>>

adds the value of condition <<condition>> to the front of the condition queue. See own chapter for expressions. If augment "int", "float", "string" or "mixed" is set, the condition is evaluated as an expression of the type the augmentation determines, otherwise it conforms to the box type.

iv. decr <var>

decrements the variable <var> by 1. Supposes var to be of integer value.

v. dir[.create|.delete|.exists] <<dir>>

<u>if aug=create</u>
creates the directory denoted by <<dir>>
adds the result to the condition queue: true = creation succeeded or directory already exists, false otherwise

<u>if aug= delete:</u>
deletes the directory denoted by <<dir>>
adds the result to the condition queue: true = deletion succeeded or directory doesn't exist, false otherwise

<u>if aug=exists</u>
check,if the directory exists
adds the result to the condition queue: true = directory exists, false otherwise

vi. do <<tagname>> <<item>>*

execute the tag with params <item>*. Note that the tag must refer to a response in the same box from which it was called. Also note that calling do isn't equivalent to a function call since the local variable context doesn't change.

vii. exists <<var>>

adds true to the condition queue, if global variable <var> exists, false otherwise

viii. file[.create|.delete|.exists] <<file>>

<u>if aug=create</u>
creates the file denoted by <file>
adds the result to the condition queue: true = creation succeeded or file already exists, false otherwise
<u>if aug= delete:</u>
deletes the file denoted by <file>
adds the result to the condition queue: true = deletion succeeded or file doesn't exist, false otherwise

if aug=exists
check,if the file exists
adds the result to the condition queue: true = file exists, false otherwise

ix. finis

delete the first entry from the condition queue.

x. if

if the first entry in the condition queue is true, then continue with next action, otherwise the message response ends.
The first entry from the condition queue is deleted.
Note than cond and if/ifnot together implement a LIFO behavior.

xi. ifnot

if the first entry in the condition queue is false, then continue with next action, otherwise the message response ends.
The first entry from the condition queue is deleted.
Note than cond and if/ifnot together implement a LIFO behavior.

xii. incr <var>

increments the variable <var> by 1. Supposes var to be of integer value.

xiii. init (<var> <<value>>)+

sets the variables to the value following the variable name. The value must be a single entity, it cannot be an expression.
The action is only executed if the response doesn't contain a repeat or if the repeat hasn't been executed yet (i.e. in the first execution of a loop).

xiv. input[.line] <var>

without aug:
sets variable var to the input read from the console, input is terminated by white space.

with aug=line
sets variable var to the input read from the console, input is terminated by eol.

xv. print[.nl] <<item>>+

without aug:

prints the items to the console

with aug = nl:
prints the item to the console, followed by newline.
This is just syntactic sugar for print <<item>>+ \n

xvi. readline <<filename>> <var>

reads the next line of file <filename> into the string variable <var>
In addition, a condition is set: false if eof was reached, true otherwise.

xvii. readvars <<filename>>

reads the file <filename> line by line. If the first entry of a line matches the current box name, the interpret the rest of the line as the list of variable contents (var,var2,..) and set the appropriate variables. The whole file is read so the last occurrence of the box name determines the content set.
In addition, a condition is set: true if the box name was found, false otherwise.

xviii. repeat

go to the beginning of the message response and start over again

xix. rv <var>

reverses the string of variable <var> char-by-char

xx. save <<var>> <<expression>>

sets global variable <var> to the result of the (infix) expression.
Note that the expression is always evaluated in “mixed” mode.
See own chapter on expressions

xxi. send <<tagname>> <<item>>*

send the tag with params <item>* to the queue

xxii. sep <<item>>

Sets the separator to item. The default separator (if nothing is set) is @none = no separator. So far this affects the following operations:

- write actions
- reading states for machine fields
- defining the rule key for machine fields

xxiii. set[.int|.float|.string|.mixed] <var> <<expression>>

sets variable <var> to the result of the (infix) expression.
If augment “int”, “float”, “string” or “mixed” is set, the expression is evaluated as an expression of the type the augmentation determines, otherwise it conforms to the box type (see however, the notes in (e)).
See own chapter on expressions

xxiv. sl <var> <<pair list>>

selects chars from var: <pair list> is a list of index ranges that are to be selected.
You may use the special symbol @ introducing a char whose first/next occurrence index is to be selected. @last selects the last index.
Note that indexing is 1-based.

xxv. sr[.desc] <var> <<string>>

sorts the string chars (desc = descending, otherwise ascending) and saves the result in <var>. This action is only available for type string.

xxvi. stop

stop program execution immediately

xxvii. tl[.once] <var> <<pair list>>

without aug:
this action works similar to translate, but on a char-by-char basis:
it loops over all chars of variable <var> applying the translation to each char.
You may use the special symbol @- as the second part of a pair to indicate that the char should be deleted.

with aug=once:
translation is done for each char but stops after the first translation has been found and applied

xxviii. translate <var> <<pair list>>

the first argument is a variable name, followed by a list of pairs:
these are checked in sequential order until a match is found or the sequence is exhausted:
A match is found, if the variable is equal to the first part of a pair. Then the variable is set to the second part and evaluation ends.

if no match is found, the variable isn’t changed

xxix. write[.line|.<n>|.vars] <<filename>> [<<list>>]

appends the content of the list to file <filename>

if aug1 == line

writes the list to the file as a single line, separated by the separator.

if aug1 == <n>
write lines of list elements, where n list elements make up one line, separated by the separator.

if aug1 == vars
writes a line containing the box name and the values of all scalar variables (var1, var2,...) to the file. If a list is given, it is ignored.

otherwise write the list content to the file with one list item per line

xxx. !<<var>>

the variable is evaluated. Its value should be a valid action name. If so, this is executed.

e) The Box

While basic actions and messaging features as described above (BNF) are the same for all boxes, they differ in the number of variables and may understand specific actions depending on their type. Note that underscores are not written in the code, but just serve to more easily scan this text.

i. scalar int, scalar string and scalar float

This type only has one variable that may be referred to by var.

ii. triple int, triple string and triple float

This type has three variables that may be referred to by var1, var2, var3.

iii. quintuple int, quintuple string and quintuple float

This type has five variables that may be referred to by var1, var2, var3, var4, var5.

iv. septuple int, septuple string and septuple float

This type has seven variables that may be referred to by var1, var2, var3, var4, var5, var6, var7.

v. decuple int, decuple string and decuple float

This type has ten variables that may be referred to by var1, var2, var3, var4, var5, var6, var7, var8, var9, var10.

vi. array int, array string and array float

These types contain:

- a vector (of arbitrary length) that may be referred to by vec.
- indexes to that vector that may be referred to by idx, idy, idz
- the vector length that may be referred to by len (read only)
- the number of columns that may be referred to by collen. (may only be set using impose)
- the number of rows that may be referred to by rowlen.(may only be set using impose)
- the vector elements that may be referred to by vecx meaning vec[idx]

or vecn meaning vec[idn] where idn is the one-dimensional index corresponding to (idx,idy,idz)

- you may also refer to the n-th array element as @<n>, <n>=1,2,.. You may use @!var to force evaluation of var and use the result as the index. This usage is , however, restricted to read only.
- additional variables var1, var2, var3, var4, var5
- you may use vec as a variable name in cases, where a list is expected as a parameter to an action. i.e. like so: *set var vec,* where vec represents an expression.

note that indexing is 1-based

vii. machine field int, machine field string and machine field float

These types contain:

- a 2-dimensional field (of arbitrary length) containing a type and a value per element.
- indexes to that field that may be referred to by row and col.
- the field length that may be referred to by len (read only)
- the number of columns that may be referred to by collen. (may only be set using impose)
- the number of rows that may be referred to by rowlen.(will be automatically derived from collen and len)
- the field element values that may be referred to by fieldn.value meaning (field[row,col].).value
- the field element types that may be referred to by fieldn.type meaning (field[row,col].).type
- additional variables var1, var2, var3, var4, var5
- You may use field.type or field.value as a variable name in cases, where a list is expected as a parameter to an action. i.e. like so: write *field.type*.

note that indexing is 1-based

viii. machine net int, machine net string and machine net float

These types contain:

- a list (of arbitrary length) of states containing a value per element.
- the number of states may be referred to by len (read only)
- the state value may be referred to by svalue.<<name>> (read only)
- additional variables var1, var2, var3, var4, var5
- You may use states.name, states.value or output as a variable name in cases, where a list is expected as a parameter to an action. i.e. like so: write *states.name*. (read only)

- the n-th item of the input list may be referred to by input.<<n>> (read only)

note that indexing is 1-based

ix. queue array int, queue array string and queue array float

These type contains:

- all variables that an array contains
- the queue length that may be referred to by qlen (read only)
- two more arrays vec2 and vec3. Their lengths may be referred to as len2 and len3, resp. Their elements may be accessed by vec2@<n> and vec3@<n>

note that indexing is 1-based

f) The Array Box

See above. May be of type string, int or float. Note that – if not stated other-wise – most actions are ignorant of the dimension set by impose and assume dimension = 1. Arrays understand the following specific actions:

i. append[.new] <vec> <<list>>

without aug:

appends the list elements to the array <vec>

with aug=new:

appends only the list elements to the array <vec> which aren't already present in <vec>. Using .new consequently to fill <vec> thus guarantees uniqueness.

ii. arrange <<specification>>

This changes the layout of the array (i.e. the contents of the array elements). The action is only available for arrays of type string.

So far the following specifications are recognized:

grid to rows

Joins all array elements belonging to the same row into one string. After the action, the array consists of these row strings. collen must have been set by using impose.

rows to grid

Splits all array elements belonging to the same row. After the action, the array is in grid form. Note that rows may have different lengths. Thus a subsequent grid to rows command may not yield the original arrangement.

grid to cols

Joins all array elements belonging to the same col into one string. After the action, the array consists of these col strings. collen must have been set by using impose.

cols to grid

Splits all array elements belonging to the same column. After the action, the array is in grid form. Note that columns may have different lengths. Thus a subsequent grid to cols command may not yield the original arrangement.

<u>cols to rows</u>

Assumes that the array elements are columns. Splits every column and rearranges the array so that the array now contains a list of rows.

<u>rows to cols</u>

Assumes that the array elements are rows. Splits every row and rearranges the array so that the array now contains a list of columns.

Note:

- If the array is in row form, using cols to grid transposes the array.
- If the array is in col form, using rows to grid transposes the array.
- The sequence: arrange grid to rows | reverse | arrange cols to grid rotates the array 90° clockwise.

iii. branch[.not][.inc][.inclt] <<tag>>

<u>without inc:</u>
see basic actions

<u>with inc:</u>
```
branch.inc <tag>       = incindex | send.ifnot <tag> | if
branch.not.inc <tag>   = incindex | send.if <tag> | ifnot
```

<u>with inclt:</u>
```
branch.inclt <tag>     = incindex.lt | send.ifnot <tag> | if
branch.not.inclt <tag> = incindex.lt | send.if <tag> | ifnot
```

will usually be used in a sequence ending with repeat

You may view this action as defining the else-branch where the if-branch means going on. We could have renamed it to ‘else’, however this feels awkward if augmentation is used, since we do not expect an else-statement to change a variable.

iv. change <vec> <<pair list>>

This is an array version of translate with some special features:

For each array element of the array <vec> the list of pairs is checked in sequential order until a match is found or the sequence is exhausted:
A match is found, if the array element is equal to the first part of a pair. Then the array element is set to the second part and evaluation ends.
if no match is found, the array element isn't changed.
Besides the standard placeholders you may use some additional placeholders:
<u>first part:</u>
@<n> where n is an integer: to refer to a specific index (one-based)

<u>second part:</u>
@- delete the array element, @+: duplicate the array element, @<n>: set the array element to array element n

v. clear <vec>

deletes the contents of vector <vec>

vi. collect[.int|.float|.string|.mixed] <var> <vec> <<expression>>

The expression is applied to every value of the array <vec> and the variable is set to the result of the expression. You may refer to the value of the current item by using @curr, the previous item by using @prev and the next item by using @succ. Note: if you use @prev the first value of the original list will not be evaluated (for @curr). If you use @succ the last value of the original list will not be evaluated for @curr. The dimension is ignored.

vii. compact[.row|.col] <var> <vec> [<<list>>]

<u>without aug:</u>
join the array elements of <vec> together. if a list is present, then separate array elements by a list element (array elements 1 and 2 by list element 1 , array elements 2 and 3 by list element 2 and so forth, until the list is exhausted. Then start over again.)
The result is saved in <var>.

<u>if aug == "row"</u>
The row selected by the current value of idy is joined following the rules described above.

<u>if aug == "col"</u>
The column selected by the current value of idx is joined following the rules described above.

viii. delete[.row|.at] <vec> [<index>]

<index> must only be given if augmentation 'at' is used.

if aug = at:

Delete the vector element in array <vec> given by the index <index>.

if dimension = 1:

Delete the vector element in array <vec> given by the current index idx.

if dimension = 2:

Delete the column given by the current index idx. After deletion, collen and idx are both decremented by 1.

if dimension = 2 and aug=row
Delete the row given by the current index idy. After deletion, idy is decremented by 1.

ix. find[.idx] <var> <vec> <<value>>

without aug:
set var equal to the index of the first occurrence of <value> in array <vec>, 0 if not found.

if aug = idx is set:
set var equal <vec>[<value>] (1-based)
(If <vec> = vec, this is just syntactic sugar for set idx <value> | set var vecx, however, the index idx isn’t actually updated)

x. impose [<<cols>> [<<rows>>]]

defines the current array as having a two- or three-dimensional structure with <cols> = collen = number of columns per row.

If present <rows> = rowlen = number of rows per plane. All current data is preserved.

Second dimension is indexed by idy, third by idz.

If impose has no arguments, the array is reset to 1-dimensional

xi. incindex[.lt]

increments the index variable(s) of the array if dimension = 1:

increments idx, then checks whether idx <= len and adds the appropriate condition queue entry.

if dimension = 1 and aug=lt

increments idx, checks whether it is < len after the update and adds the appropriate condition queue entry.

if dimension = 2:

aug is ignored.
Increments idx, if this is larger than collen, then reset idx to 1 and increment idy. Finally checks whether idy is <= rowlen after the update and adds the appropriate condition queue entry.

xii. insert[.row] <vec> <<list>>

if dimension = 1:

Insert the list elements in array <vec> before the index given by the current index idx. After insertion, idx is incremented by the length of <list>.

if dimension = 2:

Insert a new column before the column given by the current index idx. After insertion, collen and idx are both incremented by 1. If <list> doesn't contain enough elements to make up a new column, the list elements are reused i.e. a b c a b c ...

if dimension = 2 and aug=row
Insert a new row before the row given by the current index idy. After insertion, idy is incremented by 1. If <list> doesn't contain enough elements to make up a new column, the list elements are reused i.e. a b c a b c … If it contains to many elements, excess elements are ignored.

xiii. intersect <vec> <<list>>

intersects the list vec with <list> and saves the result in vec. Note that multiple occurences in vec will be retained but not in list, thus:
intersect {a a a} {a} results in {a a a} while
intersect {a} {a a a} results in {a}

xiv. map <vec> <<expression>>

The expression is applied to every value of the array <vec> and the result of the expression is used to form a new result list. You may refer to the value of the current item by using @curr, the previous item by using @prev nd the next item by using @succ. Note: using @prev will result in a shortened list, since the first value of the original list will not be evaluated. Using @succ will also result in a shortened list, since the last value of the original list will not be evaluated.

xv. minus <vec> <<list>>

subtracts the list <list> from vec and saves the result in vec. Note that a single occurence in list suffices to eliminate all occurences of this item in vec, thus:
minus {a a a b} {a} and minus {a b} {a} both result in {b}.

xvi. partition[.<n>[.keep|.reverse]] <vec> <<var>> [<<list>>]

the string contained in var is partitioned into <vec> as a 1-dim. array:

1.) if the current string starts with one of the strings in the list, this string becomes the next array element and the current string is reduced by that string.

2.) If nothing is found:

- goon until the unrecognized part has <n> chars. Then the string is reduced by <n> chars (n defaults to 1). If n=0,it is reduced by the whole string that cannot be recognized
- If option keep is specified this string also becomes an array element, otherwise it is dropped (may be specified by option drop). If option reverse is specified, only unknown strings are saved as elements, known strings are dropped.

3.) start over with 1.) until the current string is empty

xvii. read <<filename>>

reads the contents of the textfile into the array as a list of lines

xviii. reverse <vec>

reverses the array <vec>

xix. select[.idx] <vec> <<list>>

Constructs a new array <vec> that only contains the elements that are present in the list (i.e. works similar to an intersection of vec with the list supplied as parameters). Note that having the same value more than once in the <list> will not affect the result while same values in the array will be preserved if their value is in the <list>.

<u>if aug = idx is set:</u>
The parameter list is regarded as a list of indexes that are to be selected from the array vec to form the new array – if the index is present in the old array. Note that in this case having the same index more than once in the list will also result in having the appropriate values more than once in the result.

xx. setrange <vec> <<range expression>>

The expression is evaluated as a range expression (see below) and its result saved in array <vec>

xxi. sort[.desc] <vec>

sort the array <vec>. desc = descending, otherwise ascending.

xxii. localchange <vec> <<list of change rules>>

This action presupposes that the array is 2-dimensional and impose has been set accordingly.
Then the change rules are applied to every array item to determine new values of its surroundings and used to construct a new array. Thus the rules always see the same array when they are applied. Finally the new array replaces the old array. Rules are separated by ':'. Each rule has the following structure
<array element to be updated> <condition> => <expression>
Each rule applies to the square surrounding the current element. You may use the following placeholders to refer to elements relative to the current element:
@ul = upper left, @um = upper middle, @ur upper right, @cl = center left, @cm = center middle (the current element), @cr = center right, @dl = down left, @dm = down middle, @dr = down right.
The square layout thus is:
@ul @um @ur
@cl @cm @cr
@dl @dm @dr

For instance:
@ul @cm == 1 => 0 would mean that the upper left is set to 0 if the center element is 1.
You may define an arbitrary list of rules. If the element to which the rule applies doesn’t exist (i.e. if cm refers to the first array element there is no upper left) it is ignored.

Notes
1) Even if you only define one rule per position, due to the nature of the process each element may be updated up to 9 times. The last update will prevail. The same is true if more than one rule is defined for the same position. i.e.:
@ul @cm > 5 => 7 : @ul @cm > 3 => 6
will result in @ul = 6 if @cm>3

2) You may view this action as a generalized spread/blur:
spreading means that the central value influences its surrounding values
blur means the the surrounding values influence the central value.

xxiii. localchange.plus <vec> <<center value>> <<old value>> <<new value>>

This action presupposes that the array is 2-dimensional and impose has been set accordingly. It is a special stripped down version of localchange: There is only one rule: if the center value is <<center value>> and the old value in the positions @um @cl @cr @dm are <<old value>> then they are updated to <<new value>>.

xxiv. with(.row|.col)[.int|float|.string|.mixed] <var> <vec> <<index1>> <<index2>> <<junctor>> <<expr>>

if aug1 == row
This generates an expression by resolving expr for each column of the rows given by <index1> and <index2> and concatenating it to one compound expression (using the junctor between two successive expressions) that is finally evaluated and its value assigned to <var>. You may refer to the value of the current item of the first row by using @first and the current item of the second row by using @second. It is implied that impose has been set.

Example: Let vec be an array with three rows and cols like so:
1 2 3
4 5 6

7 8 9
then with.row var2 2 3 and (@first <= @ second)
generates the following expression:
(2 <= 3) and (5 <= 6) and (8 <= 9)
which evaluates to 1 so var2 is set to 1.

if aug1 == col
This generates an expression by resolving expr for each row of the columns given by <index1> and <index2> and concatenating it to one compound expression that is finally evaluated and its value assigned to <var>. You may refer to the value of the current item of the first column by using @first and the current item of the second column row by using @second. It is implied that impose has been set.

Example: Let <vec> be an array with three rows and cols like so:
1 2 3
4 5 6
7 8 9
then with.col var2 1 3 or (@first == @ second)
generates the following expression:
(1 == 7) or (2 == 8) or (3 == 9)
which evaluates to 0 so var2 is set to 0.

You may define the type of the expression evaluation by specifying int, float, string or mixed

xxv. write[.line|.<n>] <<filename>> [<list>]

without aug

write the list content to the file with one list item per line

if aug1 == line

writes the list to the file as a single line, separated by the separator.

if aug1 == <n>
write lines of list elements, where n list elements make up one line, separated by the separator.

Notes/Hints:

- action partition is currently only implemented for arrays of type string. It works as a generalized split function.
- resizing an array using C++ functionalities would result in (possible) data loss. We therefore implemented impose, because this technique

preserves the data (but may be slower)

- You may convert a string to an expression by using partition:
  partition.0.keep var < > & + - * / % ^ <= >= <> == ( ) or and not
  Thus you can write your own eval function (sort of), see eval.box.

g) The Machine Field Box

This is one of the structures that can be used to represent a state machine.

The state structure is actually a (rectangular) field of 'states' where only transitions between adjacent states are possible. Since the number of states is usually very large, not every state is modelled individually but a (small) set of types is defined and all states are characterized by the type they have.

Thus we may define a set of rules like so:

move rule: T I rel V → M O

this rule applies for the moves from state to state (state changes)

change on entry rule: T I rel V → op N

this rule applies on entry of a state (value change)

so far we do not model value changes that occur on exit of a state

where:

T	= type
I	= current input
rel	= relational operator for checking the value V with == equal, != unequal, =~ contains (only string), !~ not contains (only string)
V	= the value the current state has (@any if all values are allowed)
op	= the operator regarding the new value: = set, + add to old value, - = subtract from old value, & append to old value (only string)
N	= the new value (@same if no change)
M	= the direction the next state has relative to the current state
O	= output

It is allowed to have more than one rule with the same T I rel V combination.

This means, that every state is characterized by the following values:

- a position (i.e. a row (integer) and a column (integer))

- a type
- a value

Last but not least, we have to define a starting position where the execution begins and input to the machine

To implement this type of machine, the following actions are defined:

i. addmoverule <<type>> <<input>> <<relop>> <<currval>> <<dir>> <<output>>

A rule as defined above.
There may be multiple rules with the same T I oprel V combination, leading to a non-deterministic behavior (i.e. having multiple states at once).

ii. addentryrule <<type>> <<input>> <<relop>> <<currval>> <<op>> <<newval>>

A rule as defined above.
There should be only one (or no) rule per T I combination. Defining more than one rule is an error.

iii. addstates (<<type>> <<value>>)+

A sequences of states as defined above. The position is derived automatically since states are to be added in sequential order.

similar to arrays, this box also understands the following actions:

iv. branch[.not][.inc] <<tag>>

<u>without inc:</u>
see basic actions

<u>with inc:</u>

```
branch.inc <tag>       = incindex | send.ifnot <tag> | if
branch.not.inc <tag>   = incindex | send.if <tag> | ifnot
```

will usually be used in a sequence ending with repeat

You may view this action as defining the else-branch where the if-branch means going on. We could have renamed it to ‘else’, however this feels awkward if augmentation is used, since we do not expect an else-statement to change a variable.

v. clear(.field|.rules)

if aug1 = field : deletes the contents of the field

if aug1 = rules : deletes the contents of the rules

vi. impose <<cols>>

Defines the two-dimensional structure of the machine field with <cols> = collen = number of columns per row. Note that machine fields are always 2-dimensional. The first dimension is indexed by col, the second by row. You may not set rowlen but retrieve its value which is automatically derived from collen and total length

vii. incindex

Increments col, if this is larger than collen, then col is reset to 1 and row is increment. Finally checks whether row is <= rowlen after the update and adds the appropriate condition queue entry.

viii. read <<filename>>

reads the contents of the textfile into the field list as a list of state-defining pairs (type – value). Lines are split using the defined separator.

ix. runmachine[.refeed] <<startrow>> <<startcol>> <<input list>>

without aug
input is used (normal operation) to determine the next rule(s)

if aug1 == refeed

output is refed as input, thus only the first input element will be used (as a start)

Runs the machine with the specified parameters.

x. write[.line|.<n>] <<filename>> [<list>]

without aug

write the list content to the file with one list item per line

if aug1 == line

writes the list to the file as a single line, separated by the separator.

if aug1 == <n>

write lines of list elements, where n list elements make up one line, separated by the separator.

h) The Machine Net Box

This is one of the structures that can be used to represent a state machine.

The state machine is set up by defining state/value pairs (initially) and change rules like so:

C [I rel V] → S op N [O]

where:

C	= current state
I	= should be one item of the current input list
rel	= relational operator for checking I with == equal, != unequal, =~ contains (only string), !~ not contains (only string), < less , > greater, <= less equal, >= greater equal
V	=a value (fixed)
S	= the new state (may be the same as C)

op = the operator regarding the new value of the new state (*): = set, + add to old value, - = subtract from old value, & append to old value (only string)

N	= the value change (fixed)
O	= output (may be a variable etc.)

There must only be one rule with the same C I rel V combination.

This means, that every state is characterized by the following values:

- a name
- a value

Last but not least, we have to define a starting position where the execution begins and input to the machine. Usually there will be more than one run on the machine

(*) Note that this means that the initial state will never change its value. We can easily work around this by defining a new state that precedes the initial state.

To implement this type of machine, the following actions are defined:

i. addrule[.else] <<current state>> [<<input item>> <<relop>> <<value>>] <<new state>> <<op>> <<changeval>> [<<output>>]

A rule as defined above.
If more than one rule applies to the current situation, these rules are evaluated in order (of their definition) and the first matching rule is used. If the output is missing then no output is generated.

<u>if aug == else</u>
input item, relop, and value are not supplied: this is a rule that always evaluates to true (else rule)

<u>no aug</u>
the condition input relop value is evaluated, if it evaluates to true, then the second part of the rule is executed. (Transiting to state newstate and updating the new states value using the operation <state value> <op> <change val>

ii. addstates (<<name>> <<value>>)+
A sequences of states as defined above. The position is derived automatically since states are to be added in sequential order.

this box also understands the following actions:

iii. clear(.states|.rules|.output)

if aug1 = states : deletes the states

if aug1 = rules : deletes the rules

if aug1 = output : deletes the output

iv. runmachine <<startstate>> <<input list>>

Runs the machine with the specified parameters.

v. write[.line|.<n>] <<filename>> [<list>]
<u>without aug</u>
write the list content to the file with one list item per line

if aug1 == line

writes the list to the file as a single line, separated by the separator.

if aug1 == <n>
write lines of list elements, where n list elements make up one line, sepa-
rated by the separator.

i) The Queue Array Box

This structure represents a queue where each element of the queue is an array. You may use all the actions that are allowed for arrays, referring to the standard array called vec (see description of Array). This Box also has two more arrays that may be referred to as <u>vec2</u>, and <u>vec3</u>.

Additionally, the structure provides actions that add, set or delete elements (=arrays) to/from the queue:

i. qins <name> <<index>>

sets the contents of array with name <name> as element list no. index in the queue. If the index already exists, the corresponding and all following elements are shifted (index increases by 1)

ii. qadd <name>

adds the the contents of array with name <name> as a new element list to the end of the queue. This is syntactic sugar for qins @new <<array>>.

iii. qset <name> <<index>>

sets the contents of array with name <name> as element list no. index in the queue. The element must already exist and is overwritten

iv. qdel <<index>>

deletes element list no. index from the queue

v. qget <name> <<index>>

sets the array vector with name <name> to the element list no. index of the queue .

For qins, qset, qdel, qget you may use @last to denote the index of the last element of the queue. For qins @new may denote the index @last + 1.

vi. clear(.q)

<u>if aug1 = q :</u>

deletes the queue

<u>without aug:</u>

see array description

vii. copy <name1> <name2>
copies the contents of name2 to name1: name1 = name2, where both are arrays

viii. swap <name1> <name2>
swaps the contents of the arrays denoted by name1 and name2.

<u>Note</u> that the queue is 1-based.

j) The Expression

Boxes understands the following operators:

Opera-tor	**Type**	**type**	**Meaning**
+	infix	numeric	addition
-	infix	numeric	subtraction
*	infix	numeric	multiplication
/	infix	numeric	(integer) division
%	infix	numeric	modulus: a % b = a mod b
==	infix	all	test for equality
<>	infix	all	test for inequality
<=	infix	all	test for less or equal
>=	infix	all	test for greater or equal
<	infix	all	test for less
>	infix	all	test for greater
or	infix	all	logical or
and	infix	all	logical and
not	prefix	all	logical not
(	prefix	all	parenthesis: priority for enclosed term
)	postfix	all	parenthesis: priority for enclosed term
&	infix	string	concatenate two strings
max	infix	all	maximum of two values
min	infix	all	minimum of two values
abs	prefix	numeric	absolute value
size	prefix	string	size of a string
lower	prefix	string	convert string to lower
upper	prefix	string	convert string to upper
first	prefix	string	return first char of a string as string
last	prefix	string	return last char of a string as string
head	infix	string	<string> head <n> returns the first <n> chars of <string>
tail	infix	string	<string> tail <n> returns the last <n> chars of <string>
sfind	infix	string	<string> sfind <sub> returns the position of <sub> in <string>, 0 if not found, 1-based
trim	infix	string	<string> trim <char> trim string, deleting leading and trailing <char>s
matchl	infix	string	(match from left) <s1> matchl <s2>: result is “1”, if l = length of s2 and the first l chars of both strings match, else “0”
ascii	prefix	string	get the ascii value of the first string char

note that relational operators yield the result 1 if true, otherwise 0

Whether the expression to be evaluated is regarded as string or numeric is determined by the box type specification. However, if array indexes are set using action ‘set’ they are recognized as being of type int.

k) The Range Expression

Boxes can do calculations involving range expressions. Note that ranges behave similar to sets and the operators for ranges thus are more like set operators. In contrast to ‘conventional’ arithmetics, the result of such an expression will therefore be a list of ranges, not a single value. So far, the operators listed below have been implemented. Note that r1 and r2 here denote ranges, while s denotes a single value. Note that the result is set to invalid if the number of result ranges is 0.
Ranges are denoted by <n>,<m> where both <n> and <m> are numbers and <n> <= <m>. The invalid range is denoted by .,. All operations on invalid ranges also yield .,. as their result. If the result contains invalid and valid ranges the invalid ranges are dropped.

Operator	**Type**	**usage**	**Results [ranges]**	**Meaning**
+	infix	r1 + r2	1 - 2	set addition (merge), if both ranges overlap, the result is a single range, otherwise 2 ranges
-	infix	r1 - r2	0-2	set subtraction: all values inside r1 but not in r2
&	infix	r1 & r2	0 – 1	set intersection: the range of values inside both r1 and r2
==	infix	r1 == s	0 - 1	cut operation: results is the range s..s, if s falls inside r1, invalid range otherwise
<>	infix	r1 <> s	1 - 2	cut operation: if s falls inside r1, the range is split in two: min..s-1 s+1..max otherwise r1 is not changed
<=	infix	r1 <= s	0 - 1	cut operation: if s falls inside r1, the range is cut: min..s otherwise unchanged or invalid
>=	infix	r1 >= s	0 - 1	cut operation: if s falls inside r1, the range is cut: s..max otherwise unchanged or invalid
<	infix	r1 < s	0 - 1	cut operation: if s falls inside r1, the range is cut: min..s-1 otherwise unchanged or invalid
>	infix	r1 > s	0 - 1	cut operation: if s falls inside r1, the range is cut: s+1..max otherwise unchanged or invalid
:	postfix	all	none	separator: separates different operations from each other
size	prefix	size r1	none	the size of the range, result is a single value

You may prefix any operator – except ‘:’ - by ~ to reverse the role of the left and right operands.

4. List of Placeholders

a) @ placeholders

placeholder	occurs in	meaning
@space	all actions	space
@bar	all actions	'|'
@gvars	all actions	list of currently defined global variables
@hash	all actions	'#'
@none	all actions	empty string
@last	sl	last index of a string
@last	qins, qset, qdel, qget	queue array: index of last queue element
@<n>	sl	nth index of a string
@<n>	arrays	nth array element
@<n>	change	first part:array index second part: set the array element to array element n
@-	tl	character deletion
@-	change	delete array element
@+	change	duplicate array element
vec2@<n>	arrays	nth array element of array vec2
vec3@<n>	arrays	nth array element of array vec3
@curr	collect, map	current array element
@prev	collect, map	preceding array element
@succ	collect, map	following array element
@ul	localchange	upper left
@um	localchange	upper middle
@ur	localchange	upper right
@cl	localchange	center left
@cm	localchange	center
@cr	localchange	center right
@dl	localchange	down left
@dm	localchange	down middle
@dr	localchange	down right
@first	with	first row or col
@second	with	second row or col
@any	machine field	any value
@same	machine field	same value (i.e. unchanged)
@new	qins	queue array: last index + 1

b) other placeholders

placeholder	occurs in	meaning
#<n>	all actions	nth message parameter
#*	all actions	list of all message parameters
$<name>	all actions	content of global variable

5. Decisions

1. global variables: ~~not necessary to implement such a mechanism because this can be done by setting/getting vars of a box.~~ We allow this now.
2. introduce n-tuple: this would be analogous to a map. As a result, one would not be forced anymore to subdivide the code into smaller parts. I find this undesirable because it leads to confusing code.
3. introduce if/ifnot for all actions. ~~In this case, one would be tempted to write long chains of conditional actions to avoid sending messages. This however would also easily lead to confusion.~~ We allow this now and leave it up to the users, which style they prefer.
4. define sequences of code globally and use them like a single action. Doesn’t seem necessary so far.

6. Possible Road-map

- a mechanism to pass a whole vector by reference as one param to a message. Thus if the only send param is vec, then use bqe.set-Params(vec)
- boxname as a general read-only variable
- some mechanism to generate boxes dynamically (a kind of template with parameters?)
- extend action set so it can be used for arrays and queues
- send.before