Consensus B%

Programming Guide

Version-1.1

Patrick Bouchaud, 2021-01-28

Table of Contents

I. Introduction

The objective of Consensus is to allow the user to create, observe and share his own knowledge base(s) using his own syntax and terminology.

Note that the word *observe* here refers to the fact that the targeted knowledge base(s) do not contain only static information but also information about how this information changes over time, based on user-specified rules.

In other words, the objective of Consensus is to become a Behavioral Modelling Platform allowing the user to specify rules and behaviors pertaining to his own system, using his own terminology, and to visualize, track and communicate changes resulting from their application, on a frame by frame basis.

I.1. Consensus B% Programming Language

A Consensus Story is a program written in the Consensus B% (B-mod) Programming Language, allowing the user to describe his System as a series of in-on-do operations

in *Condition*
on *Event*
do *Action*

where

Condition, *Event* and *Action* are expressions related to the existence, creation or release of user-defined CNDB Entities.

I.2. Consensus B% Program Execution

The Consensus B% executable, provided with the distribution, allows the user to run his Story by typing either one of the following UNIX commands

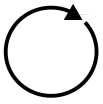
B% Story
B% -f CNDB.init Story

where

the CNDB.init file holds the list of CNDB initial conditions

The Consensus Operating System (library) manages the following operations

Load Story
Init CNDB
Run
CNDB Operations



Condition Evaluation (in)
Event Evaluation (on)
Action Registration (do)

CNDB Update & Synchronization
Exit & Deletion

The key operations as far as the user is concerned are the CNDB in-on-do operations, which are translated directly from the Consensus B% Narrative commands located in the Story file.

II. Consensus Database and CNDB Entities

A Consensus Database (CNDB) is a collection of user-defined CNDB Entities, which can be either

1. CNDB Base Entities
2. CNDB Relationship Instances

Consensus B% expressions are used to represent CNDB entities.

II.1. CNDB Base Entities

Consensus allows the user to create and reference CNDB Base entities using CNDB-unique identifiers, which can be either

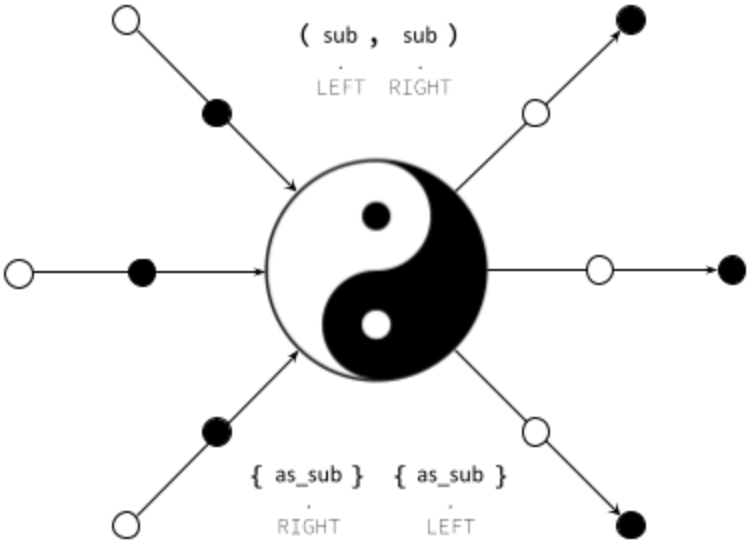
1. any combination of letters, digits and the underscore sign (_)
2. a character constant—represented by a character literal enclosed in single quotes (‘ ’)
3. one of the standalone characters ⋆ or %

II.2. CNDB Relationship Instances

Consensus allows to create and reference higher-level CNDB entities by associating any two CNDB entities together.

The resulting relationship instance is represented by enclosing the two lower-level entity representations—aka. *terms* of the representation—in parentheses, separated by a *comma*.

Technically speaking, a CNDB entity is the root of a binary tree whose leaves are CNDB Base entities. Its internal representation consists of a simple pair of pairs, the first pair representing the terms, if there are, of the entity viewed as a relationship instance, and the second the lists of instances in which the entity participates as term.



Concretely, a CNDB entity is defined as much by its constitutive entities as by its connections with others, where connections are CNDB entities in their own right.

II.3. Consensus B% Expressions

Consensus B% expressions are used to represent CNDB entities whose representation matches a specific pattern, consisting of the characters already mentioned to represent a CNDB entity, combined with special characters.

The following table lists all special characters together with their usage and effect in B% expressions.

Character		Usage	Effect
`.`	dot	as one or more of the terms of a Consensus B% expression	represents *any*
		preceding *expression* in a B% command	causes B% to interpret `.`*expression* as (this, *expression*)
		Special case: narrative parameter	cf. section III.2. of this document
		Special case: narrative variable declaration	cf. section III.3. of this document
`%`	percent	preceding the opening parenthesis of a relationship instance expression	Allows `?` as one of the expression terms (see below) and prevents B% `do` operation from substantiating the expression
		`%?`	represents the first match found during the latest parent `in ?:` or `on ?:` command
		`%!`	represents the latest result(s) evaluated upon a `do ( |` pipeline operation (see below)
`?`	question mark	as one (only) of the terms of an *expression* whose opening parenthesis was preceded by `%`	causes B% to retain not the instance itself, but the corresponding term of the instance(s) matching *expression*
		`%?`	represents the first match found during the latest parent `in ?:` or `on ?:` command
		`*?`	translates as `?:%(( *, ? ), . )`
`!`	Exclamation mark	`%!`	represents the latest result(s) evaluated upon a `do ( |` pipeline operation (see below)
`*`	star	preceding *expression*	causes B% to interpret `*`*expression* as `%(( *,` *expression* `), ? )`
		`*?`	translates as `?:%(( *, ? ), . )`
`~`	tilde	preceding *expression*	causes B% to retain only those entities *not* matching *expression*
`:`	colon	joining *expressions*	causes B% to retain only those entities matching both *expressions*
		Special case: input/output commands	cf. section III.4. of this document
		Special case: B% literals	cf. section V.2. of this document

/	forward slash	following colon	causes B% to read the sequence of characters up to the next forward slash as a regular expression representing a CNDB Base entity identifier, allowing the metacharacters `.` `[ ]` and `[^ ]` to be used as regex features
{ }	curly brackets	as *term* of a do (operation	causes B% to evaluate *term* as the union of the sets of CNDB entities resulting from the expressions enclosed in the curly brackets, separated by *comma*
|	pipe	following *term* of a do (operation	starts B% | pipeline sub-*expression*, allowing to reference the latest evaluated term as `%!`

Note, for instance, that

- The expression `~%( .,. )` represents all CNDB Base entities (singletons)
- The expression `~%( ?, . ):~%( ., ? )` represents all highest-level relationship instances contained in the CNDB, singletons possibly included

III. Consensus B% Narratives and Operations

A Consensus Story is a collection of Consensus B% Narratives, each consisting of

1. A narrative header - optional - consisting of a colon (`:`) as first character of a Story line, optionally followed by a narrative prototype
2. A narrative body consisting of B% Narrative commands

In case a narrative header is specified containing a narrative prototype, then the narrative body will only be traversed if a CNDB entity was found during the execution of a narrative `%` command (see below) matching the narrative prototype.

Otherwise—that is, if either no narrative header or a header with no narrative prototype was specified—then the narrative body will always be traversed.

III.1. B% Narrative Commands

The table below lists all existing Consensus B% Narrative commands, together with a short description.

Each narrative command has its own starting line in the Story file. The success or failure of a narrative command conditions the execution, or not, of all the narrative commands immediately following on a line with a level of indentation superior—or equal, if the command starts with `else`—to the current one.

command		argument	description
`.`		*identifier*	Narrative variable declaration—see below Section III.3.
`%`	else `%`	(*expression*)	enables execution of all B% narratives for which a CNDB entity can be found matching both *expression* and the narrative's prototype
in	else in	`~.`	passes if the CNDB is empty
		`?:` *expression*	passes if at least one CNDB entity matches *expression*, in which case allows `%?` to represent the first matching entity as term of any narrative command immediately following on a line with a level of indentation superior to the current one.
		expression	passes if at least one CNDB entity matches *expression*
on	else on	init	passes only during the first execution cycle (frame)
		`~.`	passes only if no change occurred during the last frame
		`~`(*expression*)	passes if at least one CNDB entity matching *expression* was released during the last frame
		`?:` *expression*	passes if at least one CNDB entity matching *expression* was created during the last frame, in which case allows `%?` to represent the first match as term of any narrative command immediately following on a line with a level of indentation superior to the current one.
		expression	passes if at least one CNDB entity matching *expression* was created during the last frame

do	else do	exit	causes B% to terminate execution at the end of the current frame
		~(*expression*)	releases all CNDB entities matching *expression*
		expression	instantiates CNDB entities as required and as long as feasible toward substantiating *expression* Special case: if *expression* is of the form ((`*`, *variable*), *value*) then pre-existing ((`*`, *variable*), `.`) relationship instances not matching *expression* are released prior to instantiation. The same applies to sub-expressions.
		expression **:** *format* `<`	reads input from stdin according to *format*, then if input is EOF releases (`*`, *expression*) otherwise instantiates ((`*`, *expression*), input)
		`>` *format* **:** *expression*	writes *expression* evaluation results to stdout according to *format*
else			passes if the previous `in` or `on` command at the same indentation level was executed and failed

III.2. B% Narrative Prototypes and Narrative Parameters

Each CNDB entity can have a story of its own, which we call narrative.

Narratives are introduced by a colon (`:`) as the first character of a Story line, followed by the narrative's prototype, the latter consisting of an expression of the form normally used to represent CNDB entities, but where terms of the form **.***identifier* take on a special role, in addition to representing *any*.

The *identifier* part of such terms—where *identifier* can be any combination of letters, digits and the underscore symbol (_)—is called a B% Narrative Parameter.

These can be used as terms of any expression in the narrative body, where they represent the corresponding term of the entity matching the prototype of the narrative currently under execution. The entity itself is represented by the built-in B% Narrative Parameter `this`.

III.3. B% Narrative Variables

B% Narrative Variables are declared using the B% Narrative command **.***identifier*—or several, separated by *space*, on the same line—anywhere in a B% Narrative body.

Provided the declaration command is executed, the *identifier* part of the declaration—where *identifier* can be any combination of letters, digits and the underscore symbol (_)— can be used as term of any subsequent expression in the narrative body, where it represents the relationship instance (`this`, *identifier*) which was created, if it didn't exist, upon the declaration command's execution.

Otherwise—that is, if the declaration command is not executed—then *identifier* represents the corresponding CNDB Base entity in the course of that frame.

Note that **.***identifier* used as term of an expression translates directly into (`this`, *identifier*) which, if *identifier* was declared as a narrative variable, would probably not achieve the desired effect.

III.4. B% Narrative Input and Output Commands

III.4.1. B% Narrative Input Commands

B% Narrative Input commands are commands of the form

do *expression* **:** *format* <

where

format - optional - is a sequence of characters enclosed in double quotes (“”) specifying the characters which are expected to be read from the input stream, so that

1. If input does not match the format specified at current format position, then:

   if input is EOF, then the (⋆, *expression*) relationship instance is released. Otherwise, if input is not EOF, then the input command ends, with the last input character put back into the stream.

2. Otherwise, if input does match the format specified at current format position, then:

   if the characters at the current format position do not represent a format specifier—starting with the % character—then input is discarded and the command continues, with the current format position moved forward. Otherwise, if the characters at the current format position do represent a format specifier, then the ((⋆, *expression*), input) relationship instance is created according to the format specifier, and the command continues, with the current format position moved forward.

   The following table lists the format specifiers currently supported in the B% Input Command, together with expectation.

specifier	expectation
%_	instantiate the CNDB base entity or relationship instance represented by the next sequence of characters read from stdin—discarding leading unquoted separators, as well as both leading and interspersed # … cr sequences.
%c	instantiate the single character entity corresponding to the next byte read from stdin.
%%	instantiate the single character entity % provided the corresponding byte is read from stdin.

   Version-1.1 restriction: only the first format specifier encountered in *format* will actually be taken into account—others will be ignored—as only one *expression* is allowed in the command.

If no *format* is specified, then the command defaults to the %_ format specifier.

III.4.2. B% Narrative Output Commands

B% Narrative Output commands are commands of the form

do > *format* **:** *expression*

where

format - optional - is a sequence of characters enclosed in double quotes (“”) specifying the characters to be written to stdout, either as such—including the customary backslash \ usage—or, if the characters represent a format specifier—starting with the % character—according to the formatted representation of the CNDB entities resulting from evaluating *expression*.

The following table lists the format specifiers currently supported in the B% Output Command, together with expectation.

specifier	expectation
`%_`	output the B% representation of the CNDB entities resulting from evaluating *expression*, using single quotes as needed in case of single character entities.
`%s`	output the B% representation of the CNDB entities resulting from evaluating *expression* either 1. If the evaluation results in a single character entities: without single quotes 2. If the evaluation results in a single relationship instance: by preceding the opening `(` of the relationship instance with backslash: `\(` 3. if the evaluation yields multiple results: by preceding the opening `{` of the group of results with backslash: `\{`
`%%`	output `%`

Version-1.1 restriction: only the first format specifier encountered in *format* will actually be taken into account—others will be ignored—as only one *expression* is allowed in the command.

If no *format* is specified, then the command defaults to the `%_` format specifier.

III.4.3. Special Case and Useful Examples

- The B% command `do >:` outputs a carriage return (*cr*)
- The command `do >: ~%(.,.)` outputs all CNDB Base entities
- The command `do >: ~%(?,.): ~%(.,?)` outputs only the highest-level relationship instances

The latter command, piped into a CNDB.init file, allows to backup and restore the entirety of the CNDB at that time.

IV. B% Programming Examples

IV.1. First Steps

Using an ASCII editor of your choice, open a new file, which we shall name Story, and type in:

```
1  on init
2          do > “hello, world\n”
3  else do exit
```

where

Lines 1 and 3 begin with the text—not numbers.
Line 2 begins with a *tab*, followed by the text.

Then, from a UNIX command shell with the PATH environment variable set to include the location of your B% executable, and the current directory set to the directory where your Story file is located, run:

B% -p Story

This will output either the program you have just written, stripped of all extraneous space or comments, or, if an error was found, an error message indicating the line and column of the error. When no error remains, run:

B% Story

Congratulations! You have successfully created and run your first Consensus Story.

IV.2. Comments and other B% Preprocessing Commands

- Single line comments start with two forward slashes (//)
- Multi-line comments start with /* and end with */
- Outside of comments, a single backslash (\) at the end of a line allows its continuation to the first character other than space (*sp*) or tab (*ht*) on the next line.
- + resp. − signs as first characters on a line insert resp. remove the equivalent number of leading tabs (*ht*) from this line onwards.
- Lines starting with the pound (#) sign are ignored at present, but their usage in a Story file should be reserved for future versions.

IV.3. The hello_world.story

The file *hello_world.story* in the Release/Examples/1_Schematize directory of the distribution contains a simplified version of the *yak.story* and *schematize* programs located in the same directory.

This example provides a comprehensive illustration of the range of features presented so far defining the scope of the Consensus B% Version-1.1 release. It is also intended to facilitate the reader’s understanding of the other two, more advanced programs.

A complete listing of the *hello_world.story* program is included here, starting on the next page.

```
: // base narrative
        .carry
        on init
                do ( Schema, (h,(e,(l,(l,(o,(',',(' ',(w,(o,(r,(l,(d,'\0')))))))))))) )
                do ( *, input ) // required to catch EOF first frame
                do ((*,record), (record,*))
                do INPUT
        else in INPUT
                %( schema, . )
                on INPUT
                        do ( schema, %( Schema, ? ) )
                else in ( schema, . )
                        in ( schema, . ): %( ?, (COMPLETE,.))
                                do ~( INPUT )
                        else in ( schema, . ): ~%( ?, READY )
                        else // all schemas ready
                                on ( ., READY )
                                        in *carry
                                                do ((*,input), *carry )
                                                do ~( *, carry )
                                        else do input:"%c"<
                                on ((*,input), . )
                                        do ((*,record), (*record,*input))
                                on ~( *, input )
                                        do ((*,record), (*record,EOF))
                else // all schemas failed
                        do ~( INPUT )
        else on ~( INPUT )
                in (*,input): ~%(?,.) // nop
                        do exit
                else in ( schema, . ): %( ?, (COMPLETE,']'))
                else in *record: ~((record,*), . ) // not first input
                        do ((*,carry), %((.,?):*record))
                        do ((*,record), %((?,.):*record))
                do OUTPUT
        else in OUTPUT
                .f
                on OUTPUT
                        do ((*,f), %((record,*), . ))
                        in *carry // trim record
                                do ~( *record, . )
                        in ( schema, . )
                                do >" *** "
                else
                        in ?: %((.,?): *f ): ~EOF
                                do >"%s": %?
                        in ?: ( *f, . ) // next frame
                                do ((*,f), %? )
                        else
                                in ( schema, . )
                                        do >" *** "
                                do ~( OUTPUT )
        else on ~( OUTPUT )
                in *record:(.,EOF)
                        do exit
```

```
                else
                        do ~( record )
                        do ~( schema, . )
                        do ~( (*,input), . )
        else on ~( record )
                do ((*,record), (record,*))
                do INPUT

: ( schema, .start )
        .position .event
        on this
                do ((*,position), start )
        else in .READY
                on ((*,record), . )
                        do ~( .READY )
                        do ((*,event), %((.,?): *record ))
        else on ((*,event), . )
                in *position: '\0'
                        do .( COMPLETE, '[' ) // event not consumed
                else in ?: %((?,.): *position )
                        in %?: ( '\\', . )
                                in %?: ( ., w )
                                        in *event: /[0-9A-Za-z_]/
                                                do .CHECK
                                                do .READY
                                        else in .CHECK
                                                do ~( .CHECK )
                                                do ((*,position), %((.,?):*position))
                                                do ((*,event), *event )
                                        else do ~( this )
                                else in *event: %((.,?): %? )
                                        do ((*,position), %((.,?):*position))
                                else do ~( this )
                        else in %?:' '
                                in *event: /[ \t]/
                                        do .READY
                                else
                                        do ((*,position), %((.,?):*position))
                                        do ((*,event), *event )
                        else in *event: %?
                                do ((*,position), %((.,?):*position))
                        else do ~( this )
                else // *position is a base entity (singleton) other than '\0'
                        do >"Error: %_-terminated schema not supported\n": *position
                        do ~( this )
        else on ((*,position), . )
                in ((*,position), '\0' )
                        do .( COMPLETE, ']' ) // event is consumed
                else do .READY

```

IV.4. Story Analysis

All three programs perform the same task, which is to recognize pattern-matching sequences from an input stream of characters. But, whereas the *yak.story* and *schematize* examples allow the user to specify a full formal grammar as a collection of named Rules, each Rule made of Schemas which in turn—using the (`%`, *identifier*) relationship instance—may reference Rules, the *hello_world.story* example only allows the use of Schemas, without any cross-referencing possibility.

The key to understanding this story—or, for that matter, any Consensus Story—is to remember that, although written linearly, the code actually represents the possible application's *states* and their transitions.

Not all states are active at the same time, nor do they necessarily occur in the written order—although we do our best to arrange them this way. Transitions from one state to another are conditioned by the existence, creation or release of CNDB entities, the exact same way our own state, e.g. as a reader, is conditioned by the existence, creation or release of synapses in our brains.

The story here is composed of two parts.

The first part describes the conditions, events and actions of the base narrative, whose purpose is to read input from stdin and output it to stdout either verbatim or, if a sequence has been found matching the pattern specified in association with the keyword Schema, with the matching sequence enclosed in `***` .

> Note how the command `do ((*,record), (*record,*input))` allows to grow the `record` list from its initial `(record,*)` value.
>
> Section VI. of this document showcases a complete visual representation of this feature, which allows the list not only to be grown but also to be navigated back and forth, and trimmed.
>
> This is illustrated by the operations related to the `carry` value which is set upon an incomplete match, allowing for example the input sequence `hellohello, world` to translate correctly into the output `hello *** hello, world ***`

The second part describes the conditions, events and actions pertaining to the schema thread narrative, where

- a schema thread is a relationship instance of the form `( schema,` *pattern* `)` derived, in the base narrative, from the `( Schema,` *pattern* `)` relationship instance created upon *`init`*
- The purpose of schema thread instances is to recognize sequences matching their respective *patterns* in the on-going record operations.

The example features a single Schema-pattern association, targeting the input sequence "hello, world".

> Note how the space (*sp*) character used in the pattern allows in fact multiple consecutive spaces to be entered by the user, where space here is interpreted as any combination of space (*sp*) and tab (*ht*) characters—including none.

The interested reader will benefit from adding more or substituting the pattern expression with others, if only to validate their understanding of the schema thread narrative operations. The CNDB.init file and B% Literal expressions, described in the next section, provide additional possibilities, namely to externalize the pattern definition and to simplify its representation.

V. Consensus CNDB.init file and B% Literals

V.1. Consensus CNDB.init file

Consensus allows the user to specify a CNDB.init file to be loaded at launch time, provided the program is launched using the UNIX command:

B% -f *CNDB.init* Story

where

the *CNDB.init* file—of any user-chosen name—holds the list of CNDB initial conditions.

The Consensus Init CNDB operation consists of reading the entirety of the user-specified CNDB.init file, and of instantiating the CNDB base entities and relationship instances represented by the input, considering that:

- # … cr sequences and unquoted separators—other than those participating in the representation of a valid relationship instance—are discarded on the way.
- （**:** allows the sequence of characters up to the corresponding ） to be read as one B% Literal.

V.2. B% Literals

CNDB Literals are CNDB entities of the form

(*term* [, (*term* …)])

where

- *term* can represent either
    1. a single character entity
    2. a relationship instance of the form (‘\\’, *identifier*)
       where
       identifier represents a single character entity
    3. A relationship instance of the form (%, *identifier*)
       where
       identifier represents a CNDB base entity
- The square brackets and ellipsis symbol here are not part of the representation but signify, respectively: that the sequence of characters they enclose is optional, and that a sequence of the form enclosed in the square brackets is allowed to take place—including, or not, a sequence of the form enclosed in the square brackets—and so on.

B% Literals are sequences of characters enclosed between （**:** and the corresponding ） whose purpose is to facilitate the external representation of CNDB Literals by eliminating the commas and parentheses not participating directly in the terms.

The following table describes the correspondence between the characters located after the starting （**:** and before the corresponding ） of a B% Literal expression, and the *terms* of the resulting CNDB Literal entity.

characters	condition	term
:	: is the last character before the closing)	‘\0’
:	: is not the last character before the closing)	‘:’
c	*c* is any character except) \%	‘*c*’
\)		‘)’
\w		(‘\\’, w)
\0		(‘\\’, 0)
sp	*sp* is the space character aka. \x20	(‘\\’, ‘*sp*’)
\x*FF*	*F* is a hexadecimal digit	‘\x*FF*’
\c	*c* is any character except) w 0 *sp* x	‘*c*’
%%		(%,%)
%*identifier*	*identifier* is any combination of letters, digits and the underscore symbol (_)	(%, *identifier*)
%*identifier*’	*identifier* is any combination of letters, digits and the underscore symbol (_)	(%, *identifier*)

V.3. Example Usage

Now that we have introduced the CNDB Literals and B% Literal expressions, we can replace the command

```
do ( Schema, (h,(e,(l,(l,(o,(',',(' ',(w,(o,(r,(l,(d,'\0'))))))))))) )
```

located in the *hello_world.story* example, by the B% expression

```
( Schema, (:hello, world:))
```

located in a CNDB.init file used with that program.

VI. Lists and Visualization

Consensus allows to implement lists using the following operations:

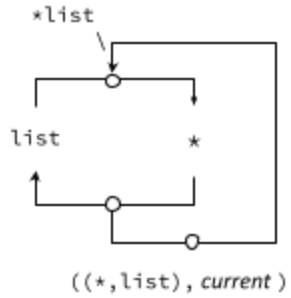
1. **Init**

   Command

   ```
   do ((*,list), (list,*))
   ```

   Description

   Initializes `list` to `(list,*)`



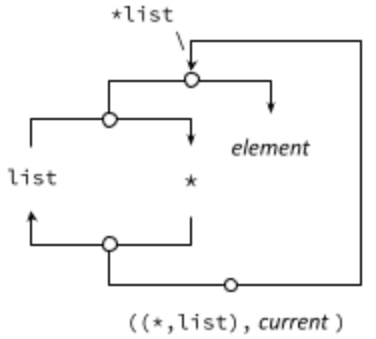
2. **Append**

   Command

   ```
   do ((*,list), (*list, element ))
   ```

   Description

   Appends *element* to `list` by creating the `(*list, element )` record—which then becomes the new `list`'s current value.



3. **Release**

   Command

   ```
   do ~( list )
   ```

   Description

   Releases `list` and all associated records

So that at any time,

`%((list,*), . )` is the first actual record
`%(((list,*), . ) . )` is the second record
etc.

`*list` is the current record
`%((?,.):*list)` is the previous record
etc.

`%((list,*), ? )` is the first actual element
`%(((list,*), . ), ? )` is the second element

`%((.,?):*list)` is the current record element

Last, but not least, we can set any list record as the right term of a relationship instance of our own choosing, the way the *current* record is—temporarily—set, without impairing the operability of the list. This capability is key in allowing the *yak.story* and *schematize* examples to segmentize the input stream.

More generally, we see here the importance of visualizing relationships in the conception and design of Consensus applications. Visualization is both a long term goal and a field of study in the Consensus Platform development.