Parse dialect

Table of Contents

1. Overview

Parse dialect is an embedded domain-specific language (DSL) of Red that allows concise processing of *input* series with grammar *rules*. Parse's common use-cases are:

- **Search:** locate specific patterns;
- **Validation**: check the conformity of input to some specification;
- **Extraction**: percolate data and aggregate values (e.g. scraping);
- **Modification**: data transformations (insertion of values, removal, and changing of matched input);
- **Language processing**: implementation of compilers, interpreters and lexical analyzers, particularly for DSLs;
- **Encoding and decoding**: conversion of data formats from one to another.

1.1. Usage

Parse is invoked through the `parse` function using a simple default syntax (see Extra functions section for more details):

```
parse <input> <rules>

<input> : any series! value except for image! and vector!
<rules> : a block! value with valid Parse dialect content (top-level rule)
```

By default, `parse` returns `logic!` value to indicate whether or not provided grammar rules succeeded in fully matching the input series.

1.2. Core principles

Parse grammar rules define patterns that are used to *consume* the input series. The basic evaluation step of Parse dialect is a rule *match*, which has one of the two outcomes:

- When a given rule matches, it *succeeds*, and Parse (optionally) *advances* past the matched portion of the input and past the matched rule;
- If there is no match, rule *fails*, while Parse *backtracks* and falls back on *alternative* rules (if any).

Input processing (*parsing*) is an indefinite application of this basic step, which is halted by either of the two terminating conditions:

- Top-level rule failure: `parse` returns `false`, signaling failed match;
- Full match of top-level rule and input exhaustion (i.e. reaching the end of the series): `parse` returns `true`, signaling succeeded match.

CAUTION	If terminating conditions are not met, Parse may enter an infinite loop.

1.3. Glossary

Parse dialect is an enhanced member of Parsing Expression Grammar (PEG) family of formal languages, differentiating itself by an extended set of features and deep integration with Red, but sharing the common meaning of core constructs and operations:

Grammar rules

Hierarchical expressions with virtually unlimited composability. Their syntax and semantics are described in Parse rules section.

Advancing

The advance of input series by a sequential match of grammar rules until top-level rule's success or failure.

Fetching

Search for a subsequent rule to apply after a successful match.

Alternating (described in PEG as *ordered choice*)

In case of a rule failure, attempt to match alternative rules following the next `|` ("pipe", "bar", "or else") word in the same block, one-by-one, until either some alternative rule succeeds or the end

of the block is reached.

Backtracking

Restoration of input and rules to their positions before the rule failure. Other changes (such as side-effects and modification of input/rules) remain.

Possessive matching

Parse rules (mainly Repetition) always try to match as much input data as possible.

2. Parsing modes

Parse offers a degree of flexibility by supporting different modes of operation.

2.1. Case-sensitivity

By default, Parse follows Red semantics and is case-insensitive. Case-sensitivity can be enabled with `/case` refinement or turned on/off with `case` keyword.

Syntax

```
case <word>

<word> : word! value
```

The value referred by word is treated as a logical flag according to standard Red semantics. Logical true enables case-sensitive mode, while logical false disables it.

2.2. Collecting mode

`collect` rule makes `parse` return a block instead of `logic!` value. Refer to Extraction section for details.

2.3. Types of input

Depending on the type of input series, some Parse rules are not applicable or behave differently.

- `any-block!`: matching by character set has no meaning and always fails;
- `any-string!`: matching by datatype or type set is not supported;
- `binary!`: matching by datatype or type set is supported for UTF-8 encoded values; such match succeeds if matched portion of the input represents one of the datatype's literal forms. Blank characters before tokens are automatically skipped.

Example

```
parse to binary! "3 words: matching by datatype" [number! set-word! 3 word!]
```

3. Parse rules

Grammar rules in Parse dialect can have several forms and usually have nested or recursive structure. Any given rule is one of the following:

- Dialect-reserved *keyword*, optionally followed by arguments and options (see below);
- Value of one of the following datatypes:
  - `datatype!` or `typeset!` that match input value by its type;
  - `bitset!`, which represents character set;
  - `word!` that refers to well-formed sub-rule;
  - `lit-word!` or `lit-path!` — convenient shortcuts for literal matching of `word!` and `path!` input values, respectively;
  - `set-word!`, used to set word to current input position;
  - `get-word!`, restores input position to which word was set previously;
  - `block!` value that contains any number of sub-rules and `|` words, which act as delimiters for alternative rules;
  - `integer!` value, serves as a counter for repetition of a rule; two subsequent `integer!` values denote a range of possible iterations;
  - `paren!` value, acts as a dialect escape mechanism by evaluating contained Red expression and resuming Parse input processing; some Parse keywords use the value returned from expression according to their specified semantics;
- Any other literal value not mentioned above, which is used as-is for direct matching of the input.

NOTE Parse is consistent with Red in using loose comparison for matching of literal values.

Each rule is characterized by conditions under which it advances the input and succeeds. An overview of Parse rules (both reserved datatypes and keywords) is given in the table below.

Table 1. Overview of Parse rules.

Rule	Category	Advances	Succeeds
`case`	Parsing modes	Never	Always
`block!`	Composition	Depends	Depends
`word!`	Composition	Depends	Depends
Literal value	Direct matching	Depends	Depends
`lit-word!`	Direct matching	Depends	Depends

Rule	Category	Advances	Succeeds
`lit-path!`	Direct matching	Depends	Depends
`datatype!`	Direct matching	Depends	Depends
`typeset!`	Direct matching	Depends	Depends
`bitset!`	Direct matching	Depends	Depends
`quote`	Direct matching	Depends	Depends
`skip`	Direct matching	Depends	Depends
`none`	Direct matching	Never	Always
`end`	Direct matching	Never	Depends
`opt`	Look-ahead	Depends	Always
`not`	Look-ahead	Never	Depends
`ahead`	Look-ahead	Never	Depends
`paren!`	Expression evaluation	Never	Always
`set-word!`	Positioning	Never	Always
`get-word!`	Positioning	Depends	Always
`integer!`	Repetition	Depends	Depends
`any`	Repetition	Depends	Always
`some`	Repetition	Depends	Depends
`while`	Repetition	Depends	Always
`to`	Search	Depends	Depends
`thru`	Search	Depends	Depends
`if`	Control flow	Never	Depends
`into`	Control flow	Depends	Depends
`fail`	Control flow	Never	Never
`break`	Control flow	Never	Always
`reject`	Control flow	Never	Never
`set`	Extraction	Depends	Depends
`copy`	Extraction	Depends	Depends
`collect`	Extraction	Depends	Depends
`keep`	Extraction	Depends	Depends
`remove`	Modification	Depends	Depends
`insert`	Modification	Always	Always
`change`	Modification	Depends	Depends

NOTE All example rules given in the sections below fully match their input.

3.1. Composition

`block!` rules directly group other rules, thus providing means of combination. `word!` rules indirectly refer to other rules and provide the means of abstraction. Together, they form the basis of Parse grammar composition.

At the structural level, Parse grammar is composed of *sequences* and *alternatives*.

- A sequence of rules is a group of zero or more rules, terminated by the *end* of the sequence. Such sequence succeeds if Parse, by successively matching its sub-rules (if any), reaches the end of the sequence. In case of any sub-rule's failure, Parse backtracks to the beginning of the failed sequence.
- End of the sequence of rules is either the end of the wrapping block or an alternative *boundary* (| word).
- Alternative is an optional sequence that Parse attempts to match in case of the previous (i.e. located before the | boundary) sequence failure.

3.2. Direct matching

Rules described in this section directly match input data, and serve as basic building blocks from which more complex rules can be composed.

3.2.1. Literal value

Match by literal value succeeds and advances the input if said literal value is equal to the value at the current position.

NOTE	By default, Parse uses loose comparison for equality checking. Case-sensitive mode enforces case-sensitive comparison.

Example

```
parse [today is 5-September-2012 #"," 20.3 degrees/celsius][
    'yesterday 'was | 'today 'is 05/09/12 comma 2030e-2 ['degrees/fahrenheit |
'degrees/celsius]
]
```

NOTE	For matching literal values reserved by Parse dialect, `quote` keyword is used as an escape mechanism.

3.2.2. Datatype

Match by datatype succeeds and advances the input if the input value is of a given type.

Example

```
parse [#a 'bird /is :the word][issue! lit-word! refinement! get-word! word!]
```

NOTE	Matching by datatype is not supported for `any-string!` input; for `binary!` input, rules are described in Types of input section.

3.2.3. Type set

Match by typeset succeeds and advances the input if input value's datatype belongs to a given typeset.

Example

```
banner: [
                |
              [_]
             [___]
            [_____]
    Red programming language
    https://www.red-lang.org
]

parse banner [default! series! any-block! any-list! all-word! any-word! any-type! any-
string!]
```

NOTE	Matching by typeset is not supported for `any-string!` input; for `binary!` input, rules are described in Types of input section.

3.2.4. Character set

If the input series is of type `any-string!` or `binary!` and input value represents a Unicode Code Point (UCP) that belongs to a given character set, match succeeds and advances the input. In all other cases match fails.

Refer to `bitset!` datatype documentation for the details on character set creation.

Example

```
animal: charset [#"🐈" #"^(1F418)" 128007]
follow: charset "🐈🐘🐇"

parse "🐈 the white 🐇" [follow " the white " animal]
```

NOTE	Lowercase/uppercase variants of the same character have different UCPs. It follows that match by character set is case-sensitive, regardless of the parsing mode.

NOTE	For `binary!` input, only UCPs up to `255` are meaningful, since Parsing in this mode is byte-granular.

3.2.5. `quote`

Acts as an escape mechanism from Parse semantics by literally matching the value that follows it. This rule succeeds and advances the input if match by literal value succeeds.

Syntax

```
quote <value>

<value> : literal value to match
```

Example

```
parse [[integer!] matches 20][quote [integer!] quote matches quote 20]
```

3.2.6. `skip`

Matches any value and advances the input. Fails only if the input position is at the tail (since there is no value to match).

Example

```
parse <□> [skip | the beat]
```

3.2.7. `none`

No-op or catch-all rule, always matches and never advances the input.

Example

```
parse reduce [none none][none #[none] ['none | none] none! none]
```

3.2.8. `end`

Succeeds only if the input position is at the tail and never advances the input (since there is no more input to advance).

Example

```
parse [(□□□□□)][end | skip [skip | end]]
```

3.3. Look-ahead

Look-ahead rules offer more fine-grained control over matching, backtracking and input advancing.

3.3.1. opt

Optionally matches a given rule, which either does or does not advance the input. Always succeeds regardless of the match.

Syntax

```
opt <rule>

<rule> : Parse rule (option) to match
```

Example

```
parse "maybe" [opt "or" "may" opt [#"b" #"e"] opt "not"]
```

3.3.2. not

Invertor, succeeds if a given rule fails and vice versa. Never advances the input, regardless of the match.

Syntax

```
not <rule>

<rule> : Parse rule to invert
```

Example

```
parse [panama][not 'man not ['plan | 'canal] not word! | skip]
```

3.3.3. ahead

Preemptively matches a given rule. Fails in case of a rule failure, otherwise succeeds without advancing the input.

Syntax

```
ahead <rule>

<rule> : Parse rule to look ahead
```

Example

```
parse [great times ahead][ahead ['great 'times] 'great ahead ['times ahead word!
'ahead] 'times skip]
```

3.4. Expression evaluation

`paren!` rule contains arbitrary Red expression that will be evaluated upon match. This rule always succeeds but does not advance the input.

Example

```
parse [(did it match?)][
    block! (not matched)
    | (probe 'backtracked) quote (did it match?) (probe 'matched!)
]
```

3.5. Positioning

It is possible to mark the current Parse input position, or to rewind/fast-forward to a position in the same input series.

3.5.1. Marking

`set-word!` rule sets word to the current series position. It always succeeds and never advances the input.

Example

```
check: quote (probe reduce [start :failed before after current end])
match: [before: 'this none after:]

parse [match this input][
    start: quote [false start] failed:
    | ahead [skip match] current: ['match 'this 'input] end: check
]
```

3.5.2. Restoring

`get-word!` rule sets the input position to the one referred by word. It always succeeds and either

advances forward, stays put or resets back, depending on where the marker is located relatively to the current input position.

Example

```
phrase: "and so on and so forth, 'til it gets boring"
goes: skip find phrase comma 2
end: tail phrase

parse phrase [again: "and" :again ['it | :goes] "until the" | :end]
```

NOTE	Restoring position to series other than the input one is forbidden.

3.6. Repetition

Rules described below act as loops or iterators by matching either a specified number of times or until failure.

NOTE	Repetition rules have possessive behavior, and will match as much input as possible.

3.6.1. Iteration count

Matches a given rule specified number of times. If range syntax is used, any number of matches in this range is accepted as successful.

Syntax

```
<count> <rule>
<count> <count> <rule>

<count> : non-negative integer! value or word! referring to such value
<rule>  : Parse rule to match a specified number of times
```

NOTE	When using range syntax, 1st integer (lower bound) must be less than or equal to 2nd integer (upper bound).

Example

```
tuple:  [2 word!]
triple: [3 skip]
THX:    1138

parse [G A T T A C A][2 3 tuple triple | 0 thx [triple tuple] 1 tuple 0 triple]
```

3.6.2. Recursion

Parse rules can be recursively composed. Recursion level is limited by Parse's internal stack depth.

Example

```
ping: [none pong]
pong: [skip ping | end]

parse https://google.com ping
```

3.6.3. any

Matches given rule zero or more times (Kleene star), stops if the match failed or if input did not advance. Always succeeds.

Syntax

```
any <rule>

<rule> : Parse rule to match zero or more times
```

Example

```
letter: charset [#"a" - #"z" #"A" - #"Z"]
digit:  charset [#"0" - #"9"]

parse "Wow, 20 horses at 12,000 RPM!" [
    any "Twin ceramic rotor drives on each wheel!"
    "Wow" any [
        comma any space any digit
        space any letter any [not comma skip]
    ]
]
```

3.6.4. some

Matches given rule one or more times (Kleene plus), stops if the match failed or if input did not advance. Succeeds if the rule matched at least once.

Syntax

```
some <rule>

<rule> : Parse rule to match one or more times
```

Example

```
parse [
    skidamarink a dink a dink
    skidamarink a doo
][
    some [
        some none 'skidamarink
        [some ['a 'dink] | 'a 'doo]
    ]
]
```

3.6.5. `while`

Repeatedly matches a given rule, stopping only after the rule's failure. Always succeeds.

CAUTION	If the rule does not fail, `while` stuck in an infinite loop.

Syntax

```
while <rule>

<rule> : Parse rule to match repeatedly
```

Example

```
parse [throw for a loop][
    while [word! | (print "failed and backtracked on matching the end") [not end]
:explicit failure]
    | [while none] :infinite loop
]
```

3.7. Search

Search rules seek specified pattern by advancing the input until a match is found.

3.7.1. `to`

Repeatedly attempts to match a given rule until its a full match. If said rule fails, the input is advanced by one element, which counts as a partial match. In case of a full match, the input position is placed at the head of the matched portion. Succeeds if rule match succeeded.

Syntax

```
to <rule>

<rule> : Parse rule (pattern to put input position at)
```

Example

```
matrix: #{
    416C6C20492073656520697320626C6F6E6465
    2C206272756E657474652C201337526564C0DE
}

parse matrix [
    to #{FACEFEED}
    | to #{1337} #{1337} start: to #{C0DE} end: (print to string! copy/part start end)
2 skip
]
```

3.7.2. thru

Repeatedly attempts to match a given rule until its a full match. If said rule fails, the input is advanced by one element, which counts as a partial match. In case of a full match, the input position is placed at the tail of the matched portion. Succeeds if rule match succeeded.

Syntax

```
thru <rule>

<rule> : Parse rule (pattern to advance thru)
```

Example

```
parse 'per/aspera/ad/astra [thru 'aspera ad: to 'astra thru end (probe ad)]
```

3.8. Control flow

Control flow rules direct execution of Parse with loop (Repetition) breaking, change of input, early exiting and conditional matching.

3.8.1. if

Conditional match, succeeds if a given Red expression evaluates to true. Never advances the input.

Syntax

```
if <expression>

<expression> : paren! expression
```

Example

```
parse [4 8 15 16 23 42][
    some [mark: skip if (any [even? probe mark/1 find [15 23] first mark])]
]
```

3.8.2. `into`

If value at the current input position has datatype supported by Parse, `into` temporarily switches input to this value and matches it with a given rule. Once the match is finished, the input is restored and parsing continues past the matched value.

Syntax

```
into <rule>

<rule> : block! rule or word! that refers to such rule
```

Example

```
rule: [some [word! | into rule]]

parse [we [need [to [go [deeper]]]]] rule
```

3.8.3. `fail`

Forces enclosing rule to instantly fail if placed at the end of it. Never succeeds or advances the input.

Example

```
parse foo@bar.baz [["quux" | some fail | "foo"] "@" [fail] | thru "bar.baz"]
```

3.8.4. `break`

Forces enclosing `block!` rule to instantly succeed. Breaks the matching loop if used at the top-level of a repetition rule. Always succeeds and never advances the input.

Example

```
parse [break away from everything][some [break] 0 1 [break] [2 [break] | 3 word!
[break] skip]]
```

3.8.5. reject

Forces enclosing block! rule to instantly fail. Breaks the matching loop if used at the top-level of a repetition rule. Never succeeds or advances the input.

Example

```
parse quote (I made a choice that I regret) [
    any [reject now] some [5 word! what: reject I see] is
    | :what 'I [[reject get] | skip]
]
```

3.9. Extraction

Extraction rules copy out matched values from the input series.

3.9.1. set

Sets a given word to the first value in a matched portion of the input.

> **NOTE** Word is set to none if the matched rule did not advance the input position.

> **NOTE** For binary! input, word is set to integer! value between 0 and 255.

Syntax

```
set <word> <rule>

<word> : word! value to set
<rule> : Parse rule
```

Example

```
parse "□□□" [set hole ahead [2 skip] set donut [to end]]
```

3.9.2. copy

Sets a given word to a copy of a matched portion of the input.

NOTE	If the matched rule did not advance the input, word is set to an empty series of the same type as input.

Syntax

```
copy <word> <rule>

<word> : word! value to set
<rule> : Parse rule
```

Example

```
parse [Huston do you copy?][2 word! copy Huston [2 word!] copy we opt "have a
problem"]
```

3.9.3. `collect`

Collects values matched by rules that are marked with `keep` keyword. Succeeds if a given rule succeeds, advancing past the matched input portion.

`keep` rule succeeds if provided rule succeeds, inserting matched values into a block allocated by `collect` rule in which it resides.

NOTE	Usage of `keep` keyword without wrapping `collect` is forbidden.

Syntax

```
collect <rule>
collect set <word> <rule>
collect into <word> <rule>
collect after <word> <rule>

<word> : word! value
<rule> : Parse rule
```

By default, values are inserted at the tail of a block. This behavior can be changed with the options described below.

Table 2. `collect` options.

Option	Description
`set`	Sets a given word to a block of collected values.
`into`	Inserts collected values into a series referred by a word, resets series' index to the head.
`after`	Inserts collected values into a series referred by a word, moves series' index past the insertion.

- If `collect` is used without `into` or `after` option in any of the rules, `parse` function will return a block of collected values (see Parsing modes); if top-level `collect` is used with `set` option, `parse` will return `logic!` value as usual.
- First use of `collect` allocates a new block that is returned by `parse` function, any subsequent `collect` allocates at the tail of its predecessor's block; with `into` or `after` option, `collect` reuses provided series buffer rather than allocating a new block.

Syntax for `keep`:

```
keep <rule>
keep pick <rule>
keep <expression>
keep pick <expression>

<rule>       : Parse rule
<expression> : paren! expression
```

- If matched rule did not advance the input, `keep` does not keep anything.
- If rule matched a single value, this value is kept; if `keep` is followed by a `copy` rule, then matched value is enclosed into a series of the same type as input.
- If rule matched multiple values, they are grouped into a series of the same type as input; with `pick` option, values are not grouped but kept one-by-one.
- If `keep` is used with `paren!` expression, result of its evaluation is kept as-is.

Example

```
fruit: charset [#"^(1F346)" - #"^(1F353)"]
plate: "tropical stuff: 🍆🍌 and other healthy food: 🥦🥒🥬"

parse plate [
    collect [
        keep (quote fruits:) collect [some [keep fruit | skip] fail]
        | keep (quote vegetables:) collect [to [#"🥒" | "Pickle Rick!"] keep pick [to
end]]
    ]
]
```

3.10. Modification

Parse can modify its input series by inserting new values and removing/changing a matched portion of the input.

3.10.1. `remove`

Either removes a portion of the input matched by a given rule or removes input between the

current position and the marked one; after that, it succeeds and retains the input position after removal.

NOTE	Removal of values is a forward-consuming operation. In other words, it counts as a match, despite the absence of input advancement.

Syntax

```
remove <rule>
remove <word>

<rule> : Parse rule
<word> : input postion
```

Example

```
parse [remove me <and me also> "but leave me be"][some [remove word!] mark: to string!
remove mark skip]
```

3.10.2. `insert`

Inserts literal value or result of expression evaluation at the current position. Always succeeds and advances the input past the insertion.

Syntax

```
insert <value>
insert <expression>

insert only <value>
insert only <expression>

<value>      : literal value
<expression> : paren! expression
```

If literal value is a `word!`, value referred by it will be used. `only` option enforces `insert/only` semantics.

Example

```
parse [assembly][insert [some] skip insert (load "required") insert only [□□ □□]]
```

3.10.3. `change`

Changes matched portion on the input to a literal value or a result of expression evaluation. In

addition to that, it can change a portion of the input between the current position and the marked one. After the change, it succeeds and advances the input past the modified portion.

Syntax

```
change <rule> <value>
change <rule> <expression>

change <word> <value>
change <word> <expression>

change only <rule> <value>
change only <rule> <expression>
change only <word> <value>
change only <word> <expression>

<rule>       : Parse rule
<word>       : input position
<value>      : literal value
<expression> : paren! expression
```

If literal value is a `word!`, value referred by it will be used. `only` option enforces `change/only` semantics.

Example

```
parse [some things never change][
    change none (quote and) 2 skip mark: to end change only mark [do]
]
```

4. Parse events

Parse dialect is implemented as a pushdown automaton; at each state transition, it emits an *event* (`word!` value) that notifies the user about the parsing process. Interaction with events and internal Parse state is achieved via `/trace` refinement and callback function (see next section).

The list of all events with conditions under which they occur is given below.

Table 3. List of Parse events.

Event	Description
`push`	After a rule is pushed on the stack.
`pop`	Before rule is popped from the stack.
`fetch`	Before a new rule is fetched.
`match`	After a value has matched.
`iterate`	After the beginning of a new iteration pass (see Repetition).

Event	Description
`paren`	After evaluation of `paren!` expression.
`end`	After reaching the end of the input.

5. Extra functions

The entry point for Parse dialect is a `parse` native that accepts input series with a block of rules and supports additional refinements.

Table 4. `parse` refinements.

Refinement	Description
`/case`	Enable case-sensitive mode.
`/part`	Limit parsing up to specified length or input position.
`/trace`	Interact with event-based Parse API via provided *callback*.

Callback function (`function!` value) with the following specification must be provided when `/trace` refinement is used.

Table 5. Callback function specification.

Argument	Type	Description
`event`	`word!`	One of the Parse events.
`match?`	`logic!`	Result of the last match.
`rule`	`block!`	Current rule at current position.
`input`	`series!`	Input series at current position.
`stack`	`block!`	Internal Parse rules stack.

Callback function must return `logic!` value to indicate if parsing should be resumed (`true`) or not (`false`).

Default `on-parse-event` callback and its `parse-trace` wrapper are provided for debugging purposes.

6. Implementation notes

Some design and implementation facets of Parse are briefly covered in this section.

6.1. Loose comparison

As was mentioned previously, Parse uses loose comparison for matching literal values, which is consistent with Red.

Example

```
parse [I'm 100% <sure>][quote :I'M 1.0 "sure"]
```

6.2. Flat rule format

To some extent, Parse supports *flat* rules format, where rules are written linearly as variable-arity expressions rather than using nested blocks.

Example

```
parse [on the count of three 1 2 3][collect set stash keep pick to ahead some 1 3
integer! remove any skip]
```

6.3. Open issues

Pending bugs and design inconsistencies relevant to Parse are listed below.

Table 6. Pending issues.

Affected rules	Description	Tickets
`change <position> <expression>`	`word!` values are not used literally.	#4200
`remove <position>`	The case where position comes after the current one is not handled.	#4199
`keep pick <expression>`	Semantics is undefined.	#4198
`collect into`	Incorrect handling of series buffer.	#4197
`into`	It is possible to match series not supported by Parse.	#4194
`break`, `reject`	Preemptive break of Repetition rules.	#4193
`insert <word>`	The rule is not handled properly.	#4153
`path!`, `remove`, `insert`, `change`	Usage of `path!` literal value inside rules is forbidden, `path!` values are handled inconsistently by Modification rules.	#4101, #3528
`fail`, `break`, `reject`	Design of some Control flow rules is not finalized.	#3478, #3398
`lit-word!`, `lit-path!`	Case-sensitive comparison is not handled properly.	#3029