Lexer

Table of Contents

1. Concept

The lexer is the part of the Red runtime library in charge of converting Red values from text representation to in-memory data structures, according to Red values syntactic rules. That process is called *loading*.

Note: the Red compiler also contains a temporary lexer implementation with limited capabilities compared to the runtime version. That lexer will be dropped once Red gets self-hosted.

2. Loading values

2.1. `transcode`

Syntax

```
transcode <input>

<input> : series to load (binary! string!).

Returns : a single value or a block of values.
```

Description

Converts an UTF-8 encoded input binary series into a Red value according to Red syntactic rules. Non-conformity with such rules will result in throwing an error. When a string series is provided, it will be internally first converted to a UTF-8 binary series before processing.

3. Scanning values

3.1. `scan`

Syntax

```
scan <input>
scan/fast <input>

<input> : series to scan (binary! string!).

Returns : recognized type (datatype!).
```

Description

Scans a string or binary buffer according to Red syntactic rules, stops after the first value is recognized and returns its datatype. No value is loaded, so no extra memory is allocated. If the scanned input is invalid, the `error!` type is returned, no exception occurs.

The `/fast` refinement trades speed for accuracy. The scanning will return the *most likely* type (not doing any deep validation) using only the fast internal prescanning engine. Very little error checking is done in such case. Cases where the guessed type can differ from the real type are shown in the following table:

Guessed type	Real type
integer!	integer! float! (1)
word!	word! lit-word! get-word! set-word!
refinement!	refinement! word! lit-word! get-word! set-word! (2)

(1) Mostly because of integer! to float! promotion rules for out of range integers.

(2) The possible words type are caused by single or multiple slash character sequences (ex: '/, /:, ://,...).

Note:

- When a string series is provided, it will be internally first converted to a UTF-8 binary series before processing.
- Scanning is much faster than loading and does not allocate any extra memory.
- Scanning is slower than prescanning, as it requires extra processing to do deep validation.

4. Processing values incrementally

4.1. `load/next`

Syntax

```
load/next <input> <var>

<input> : series to load (binary! string!).
<var>   : word to be set to the input after the first value (word!).

Returns : the first loaded value.
```

Description

Loads and returns the first Red value from an input text. The argument word is set to the remaining part of the input text after the loaded value.

4.2. `transcode/next`

Syntax

```
transcode/next <input>

<input> : series to load (binary! string!).

Returns : a block with first loaded value and rest of the input.
```

Description

Loads the first Red value from an input text. It returns a block containing:

- the first loaded value from input (any-type!).
- the remaining part of the input after the loaded value (binary! string!).

4.3. scan/next

Syntax

```
scan/next <input>

<input> : series to load (binary! string!).

Returns : a block with the type of the first value and rest of the input.
```

Description

Scans the first Red value from an input text. It returns a block containing:

- the datatype of the first value from input (datatype!).
- the remaining part of the input after the scanned value (binary! string!).

5. Instrumenting the lexer

5.1. The lexer processing pipeline

The tokenization process is split in stages, triggering events where a user-provided callback function can be invoked. The different stages are:

```
                +-> ERROR
               /
         +-> CLOSE series
        /
       +-> OPEN series
      /
-> PRESCAN token -> SCAN token -> LOAD value
     \               \             \
      +-> ERROR       +-> ERROR     +-> ERROR
```

So the lexer events are: `prescan`, `scan`, `load`, `open`, `close`, `error`.

5.2. transcode/trace

Syntax

```
transcode/trace <input> <callback>

<input>    : series to load (binary! string!).
<callback> : a callback function to process lexer events (function!).

Returns    : a single value or a block of values.
```

Description

Converts an UTF-8 encoded input binary series into a Red value according to Red syntactic rules, calling a user-provided callback function for each lexer events.

Callback function specification block:

```
func [
    event [word!]               ;-- current lexer state (see table below).
    input [string! binary!]     ;-- reference to the input series at current loading
position (can be changed).
    type  [datatype! word!]     ;-- word or datatype describing the type of token or
value currently processed.
    line  [integer!]            ;-- current input line number.
    token                       ;-- current token as an input slice (pair!) or a
loaded value.
    return: [logic!]
][
    [events]                    ;-- optional restricted event names list
    ...body...
]
```

The offset of the `input` series argument is set where the lexer stopped after detection a token's end. That offset can be modified from within the callback function. If an `error` event is ignored, the input does not advance automatically, it's up to the callback function to set the `input` series to the right position for resuming the processing. Failure to do so can result in infinite loops.

In `error` events, `type` argument contains the datatype name that was partially recognized. If the error happens on an isolated character, like on an unmatched closing delimiter `) ] }`, `type` is set to `error!` as no specific datatype is recognized in such cases.

The body block can start with an optional filtering block, for indicating which events will be triggered. This allows to reduce the number of callback calls resulting in much better processing performance.

The meaning of some arguments and return value *depends* on the event. The following table documents the possible combinations and effects:

Event	Type	Token	Return Value	Description
prescan	word! datatype!	pair!	true: scan false: drop	When a Red token has been recognized.
scan	word! datatype!	pair!	true: load false: drop	When a Red token type has been accurately recognized.
load	datatype!	<value>	true: store false: drop	When a Red token has been converted to a Red value.
open	datatype!	pair!	true: open false: drop	When a new block!, paren!, path!, map! or multiline string! is opened.
close	datatype!	pair!	true: close false: drop	When a new block!, paren!, path!, map! or multiline string! is closed.
error	datatype!	pair!	true: throw false: ignore	When a syntax error occurs.

Possible values for `type` field (word! or datatype!) in `scan` event:

```
eof comment hex error! block! paren! string! map! path! word! refinement!
issue! file! binary! char! percent! integer! float! tuple! date! pair! time!
money! tag! url! email! ref! lit-word! get-word! set-word!
```

Possible values for `type` field (datatype!) in `open` event:

```
block! paren! string!(1) map! path! lit-path! get-path!
```

Possible values for `type` field (datatype!) in `close` event:

```
block! paren! string!(1) map! path! lit-path! get-path! set-path!
```

(1): only for strings delimited by brackets.

Notes:

- If `false` is returned on a `prescan` event, the corresponding `scan` and `load` events will be skipped.
- If `false` is returned on a `scan` event, the corresponding `load` event will be skipped.
- If an `open` event is dropped, the corresponding `close` event should also be dropped.

See examples at https://github.com/red/code/tree/master/Scripts/lexer

6. Values Literal Syntax

6.1. Raw Strings

Strings in Red have special rules for some characters, like using ^ character as escaping mechanism or bracketed strings having to balance nested curly brackets. Raw strings format provides a way to input literal strings without any special treatment of its content.

Syntax

```
%{...}%
%%{...}%%
%%%{...}%%%
...
```

Any number of % character can be used in order to make the ending sequence not collide with string's content. The leading count of % must match the trailing count, otherwise a syntax error will occur on loading.

^ is processed as a regular character. Curly braces can be used without any escaping or balancing constraint.