Syntax Color: Utilities

Version 8.12.0.11

Scott Owens

March 5, 2024

The "syntax-color" collection provides the underlying data structures and some helpful utilities for the color:text<%> class of framework.

Contents

1 Parenthesis Matching **3**

2 Lexer Contract and The Don't-Stop Structure Type **4**

3 Racket Lexer **6**

4 Default Lexer **9**

5 Module Lexer **10**

6 Scribble Lexer **13**

7 Splay Tree for Tokenization **15**

8 Interface for Clients of Syntax Color Output **17**

9 Racket S-Expression Indentation and Navigation **20**

9.1 S-Expression Navigation 20

9.2 S-Expression Indentation 21

1 Parenthesis Matching

```
(require syntax-color/paren-tree)
             package: syntax-color-lib
```

```
paren-tree% : class?
  superclass: object%
```

Parenthesis matching code built on top of `token-tree%`.

2 Lexer Contract and The Don't-Stop Structure Type

```
(require syntax-color/lexer-contract)
                  package: syntax-color-lib
```

```
lexer/c : contract?
```

Checks to be sure a lexing function is well-behaved, constrained to functions where the second return value is a symbol. For more details, see `start-colorer` in `color:text<%>`.

```
lexer*/c : contract?
```

Checks to be sure a lexing function is well-behaved. For more details, see `start-colorer` in `color:text<%>`.

It also supplies a few random arguments to the lexer and checks the results, using `option/c`'s `#:tester` argument.

Added in version 1.2 of package `syntax-color-lib`.

```
lexer*/c-without-random-testing : contract?
```

The same contract as `lexer*/c`, except without the random testing.

```
(struct dont-stop (val))
  val : any/c
```

A structure type used to indicate to the lexer that it should not allow itself to be interrupted. For more details, see `start-colorer` in `color:text<%>`.

```
(check-colorer-results-match-port-before-and-afters
 who
 type
 pos-before
 new-token-start
 new-token-end
 pos-after)
→ void?
 who : symbol?
 type : any/c
 pos-before : (or/c exact-positive-integer? #f)
 new-token-start : (or/c exact-positive-integer? #f)
 new-token-end : (or/c exact-positive-integer? #f)
 pos-after : (or/c exact-positive-integer? #f)
```

Checks that the results of a colorer make sense with respect to the positions of the port, before and after the lexer is called.

The *pos-before* argument is expected to be the third result of `port-next-location` before a lexer is called and the *pos-after* argument is expected to to be the third result of `port-next-location` after the lexer is called. The *type*, `token-start`, and `token-end` arguments should be the corresponding results from the colorer (c.f. `start-colorer`).

This function raises an error unless the following boolean expression is true:

```
(or (equal? type 'eof)
    (and (= pos-before new-token-start)
         (< new-token-start new-token-end)
         (= new-token-end pos-after)))
```

but it checks the individual parts of the expression to raise a more meaningful error message when some part is not true.

The *who* argument is used to start the error message.

Added in version 1.4 of package `syntax-color-lib`.

3 Racket Lexer

```
(require syntax-color/racket-lexer)
              package: syntax-color-lib
```

```
(racket-lexer in) → (or/c string? eof-object?)
                    symbol?
                    (or/c symbol? #f)
                    (or/c number? #f)
                    (or/c number? #f)
  in : input-port?
```

A lexer for Racket, including reader extensions (§13.7 “Reader Extension”), built specifically for `color:text<%>`.

The `racket-lexer` function returns 5 values:

- Either a string containing the matching text or the eof object. Block comments and specials currently return an empty string. This may change in the future to other string or non-string data.
- A symbol in `'(error comment sexp-comment white-space constant string no-color parenthesis hash-colon-keyword symbol eof other)`.
- A symbol in `'(|(| |)| |[| |]| |{| |}|)` or `#f`.
- A number representing the starting position of the match (or `#f` if eof).
- A number representing the ending position of the match (or `#f` if eof).

```
(racket-lexer* in offset mode)
 → (or/c string? eof-object?)
    (or/c symbol?
          (and/c (hash/c symbol? any/c) immutable?))
    (or/c symbol? #f)
    (or/c number? #f)
    (or/c number? #f)
    exact-nonnegative-integer?
    any/c
  in : input-port?
  offset : exact-nonnegative-integer?
  mode : any/c
```

Like `racket-lexer`, but uses the extended lexer protocol to track and report regions that are commented out with `#;`.

Added in version 1.2 of package `syntax-color-lib`.

```
(racket-lexer/status in) → (or/c string? eof-object?)
                           symbol?
                           (or/c symbol? #f)
                           (or/c number? #f)
                           (or/c number? #f)
                           (or/c 'datum 'open 'close 'continue)
  in : input-port?
```

Like racket-lexer, but returns an extra value. The last return value indicates whether the consumed token should count as a datum, an opening parenthesis (or similar starting token to group other tokens), a closing parenthesis (or similar), or a prefix (such as whitespace) on a datum.

```
(racket-lexer*/status in offset mode)
 → (or/c string? eof-object?)
   (or/c symbol?
         (and/c (hash/c symbol? any/c) immutable?))
   (or/c symbol? #f)
   (or/c number? #f)
   (or/c number? #f)
   exact-nonnegative-integer?
   any/c
   (or/c 'datum 'open 'close 'continue)
  in : input-port?
  offset : exact-nonnegative-integer?
  mode : any/c
```

Like racket-lexer/status, but with comment tracking like racket-lexer*.

Added in version 1.2 of package syntax-color-lib.

```
(racket-nobar-lexer/status in)
 → (or/c string? eof-object?)
   symbol?
   (or/c symbol? #f)
   (or/c number? #f)
   (or/c number? #f)
   (or/c 'datum 'open 'close 'continue)
  in : input-port?
```

Like racket-lexer/status, except it treats | as a delimiter instead of quoting syntax for a symbol. This function is used by scribble-lexer.

```
(racket-nobar-lexer*/status in offset mode)
```

```
→ (or/c string? eof-object?)
   (or/c symbol?
         (and/c (hash/c symbol? any/c) immutable?))
   (or/c symbol? #f)
   (or/c number? #f)
   (or/c number? #f)
   exact-nonnegative-integer?
   any/c
   (or/c 'datum 'open 'close 'continue)
  in : input-port?
  offset : exact-nonnegative-integer?
  mode : any/c
```

Like racket-nobar-lexer/status, but with comment tracking like racket-lexer*.

Added in version 1.2 of package syntax-color-lib.

4 Default Lexer

```
(require syntax-color/default-lexer)
                package: syntax-color-lib
```

```
(default-lexer in) → (or/c string? eof-object?)
                     symbol?
                     (or/c symbol? #f)
                     (or/c number? #f)
                     (or/c number? #f)
  in : input-port?
```

A lexer that only identifies `(`, `)`, `[`, `]`, `{`, and `}` built specifically for `color:text<%>`.

`default-lexer` returns 5 values:

- Either a string containing the matching text or the eof object. Block specials currently return an empty string. This may change in the future to other string or non-string data.
- A symbol in `'(comment white-space no-color eof)`.
- A symbol in `'(|(| |)| |[| |]| |{| |}|)` or `#f`.
- A number representing the starting position of the match (or `#f` if eof).
- A number representing the ending position of the match (or `#f` if eof).

5 Module Lexer

```
(require syntax-color/module-lexer)
```
package: syntax-color-lib

```
(module-lexer in offset mode)
 → (or/c string? eof-object?)
    symbol?
    (or/c symbol? #f)
    (or/c number? #f)
    (or/c number? #f)
    exact-nonnegative-integer?
    (or/c #f
          (-> input-port? any)
          (cons/c (-> input-port? exact-nonnegative-integer?
                      any/c
                      any)
                  any/c)
          (struct/c dont-stop
                    ((cons/c (-> input-port?
                                 exact-nonnegative-integer?
                                 any/c
                                 any)
                             any/c))))
  in : input-port?
  offset : exact-nonnegative-integer?
  mode : (or/c #f
               (-> input-port? exact-nonnegative-integer?any)
               (cons/c (-> input-port?
                           exact-nonnegative-integer?
                           any/c
                           any)
                       any/c))
```

Like `racket-lexer`, but with several differences:

- The `module-lexer` function accepts an offset and lexer mode, instead of just an input port.
- In addition to the results of `racket-lexer`, `module-lexer` returns a backup distance and a new lexer mode.
- When *mode* is `#f` (indicating the start of the stream), the lexer checks *in* for a `#lang` specification.

  If a `#lang` line is present but the specified language does not exist, the entire *in* input is consumed and colored as `'error`.

If the language exists and the language provides a `get-info` function, then it is called with `'color-lexer`. If the result is not `#f`, then it should be a lexer function for use with `color:text<%>`. The result mode is the lexer—paired with `#f` if the lexer is a procedure arity 3—so that future calls will dispatch to the language-supplied lexer.

If the language is specified but it provides no `get-info` or `'color-lexer` result, then `racket-lexer` is returned as the mode.

- When *mode* is a lexer procedure, the lexer is applied to *in*. The lexer's results are returned, plus the lexer again as the mode; if the lexer produces a hash-table attribute result, however, the `'type` value is extracted and returned in place of the hash table.
- When *mode* is a pair, then the lexer procedure in the `car` is applied to *in*, *offset*, and the mode in the `cdr`. The lexer's results are returned, except that its mode result is paired back with the lexer procedure.

```
(module-lexer* in offset mode)
 → (or/c string? eof-object?)
    (or/c symbol?
          (and/c (hash/c symbol? any/c) immutable?))
    (or/c symbol? #f)
    (or/c number? #f)
    (or/c number? #f)
    exact-nonnegative-integer?
    (or/c #f
          (-> input-port? any)
          (cons/c (-> input-port?
                      exact-nonnegative-integer?
                      any/c
                      any)
                  any/c)
          (struct/c dont-stop
                    ((cons/c (-> input-port?
                                 exact-nonnegative-integer?
                                 any/c
                                 any)
                             any/c))))
  in : input-port?
  offset : exact-nonnegative-integer?
  mode : (or/c #f
               (-> input-port? any)
               (cons/c (-> input-port?
                           exact-nonnegative-integer?
                           any/c
                           any)
                       any/c))
```

Like `module-lexer`, except that the attribute result propagated from a language-specific lexer can be a hash table.

Added in version 1.2 of package `syntax-color-lib`.

6 Scribble Lexer

```
(require syntax-color/scribble-lexer)
                package: syntax-color-lib
```

```
(scribble-lexer in offset mode)
 → (or/c string? eof-object?)
    (or/c symbol?
          (and/c (hash/c symbol? any/c) immutable?))
    (or/c symbol? #f)
    (or/c number? #f)
    (or/c number? #f)
    exact-nonnegative-integer?
    any/c
  in : input-port?
  offset : exact-nonnegative-integer?
  mode : any/c
```

Like racket-lexer*, but for Racket extended with Scribble's @ notation (see §2 "@ Syntax").

Changed in version 1.2 of package syntax-color-lib: Changed to be like racket-lexer* instead of racket-lexer.

```
(scribble-inside-lexer in offset mode)
 → (or/c string? eof-object?)
    (or/c symbol?
          (and/c (hash/c symbol? any/c) immutable?))
    (or/c symbol? #f)
    (or/c number? #f)
    (or/c number? #f)
    exact-nonnegative-integer?
    any/c
  in : input-port?
  offset : exact-nonnegative-integer?
  mode : any/c
```

Like scribble-lexer, but starting in "text" mode instead of Racket mode.

Changed in version 1.2 of package syntax-color-lib: Changed to be like racket-lexer* instead of racket-lexer.

```
(make-scribble-lexer [#:command-char at]) → lexer/c
  at : (and/c char? (not/c (or/c #\] #\[))) = #\@
```

Produces a lexer like scribble-lexer, but using *at* in place of @.

Added in version 1.1 of package `syntax-color-lib`.
Changed in version 1.2: Changed like `scribble-lexer`.

```
(make-scribble-inside-lexer [#:command-char at]) → lexer/c
  at : (and/c char? (not/c (or/c #\] #\[))) = #\@
```

Produces a lexer function like `scribble-inside-lexer`, but using *at* in place of `@`.

Added in version 1.1 of package `syntax-color-lib`.
Changed in version 1.2: Changed like `scribble-lexer`.

7 Splay Tree for Tokenization

```
(require syntax-color/token-tree)
             package: syntax-color-lib
```

```
token-tree% : class?
  superclass: object%
```

A splay-tree class specifically geared for the task of on-the-fly tokenization. Instead of keying nodes on values, each node has a length, and they are found by finding a node that follows a certain total length of preceding nodes.

FIXME: many methods are not yet documented.

```
(new token-tree% [len len] [data data])
 → (is-a?/c token-tree%)
  len : (or/c exact-nonnegative-integer? fasle/c)
  data : any/c
```

Creates a token tree with a single element.

```
(send a-token-tree get-root) → (or/c node? #f)
```

Returns the root node in the tree.

```
(send a-token-tree search! key-position) → void?
  key-position : natural-number/c
```

Splays, setting the root node to be the closest node to offset *key-position* (i.e., making the total length of the left tree at least *key-position*, if possible).

```
(node? v) → boolean?
  v : any/c
(node-token-length n) → natural-number/c
  n : node?
(node-token-data n) → any/c
  n : node?
(node-left-subtree-length n) → natural-number/c
  n : node?
(node-left n) → (or/c node? #f)
  n : node?
(node-right n) → (or/c node? #f)
  n : node?
```

Functions for working with nodes in a token-tree%.

```
(insert-first! tree1 tree2) → void?
  tree1 : (is-a?/c token-tree%)
  tree2 : (is-a?/c token-tree%)
```

Inserts *tree1* into *tree2* as the first thing, setting *tree2*'s root to #f.

```
(insert-last! tree1 tree2) → void?
  tree1 : (is-a?/c token-tree%)
  tree2 : (is-a?/c token-tree%)
```

Inserts *tree1* into *tree2* as the last thing, setting *tree2*'s root to #f.

```
(insert-last-spec! tree n v) → void?
  tree : (is-a?/c token-tree%)
  n : natural-number/c
  v : any/c
```

Same as

```
(insert-last! tree
              (new token-tree%
                   [length n]
                   [data v]))
```

This optimization is important for the colorer.

8 Interface for Clients of Syntax Color Output

```
(require syntax-color/color-textoid)
                package: syntax-color-lib
```

Added in version 1.3 of package `syntax-color-lib`.

```
color-textoid<%> : interface?
```

The `color-textoid<%>` interface matches a subset of the `color:text<%>` interface. It specifies methods that indentation and expression-navigation functions can use so that they work either on actual `color:text<%>` objects or in other environments that use the `start-colorer` in `color:text<%>` protocol without `racket/gui`.

```
(send a-color-textoid get-text [start end]) → string?
  start : exact-nonnegative-integer? = 0
  end : (or/c exact-nonnegative-integer? 'eof) = 'eof
```

Like `get-text` in `text%`.

```
(send a-color-textoid get-character start) → char?
  start : exact-nonnegative-integer?
```

Like `get-character` in `text%`.

```
(send a-color-textoid last-position)
 → exact-nonnegative-integer?
```

Like `last-position` in `text%`.

```
(send a-color-textoid position-paragraph  start
                                          [at-eol?])
 → exact-nonnegative-integer?
  start : exact-nonnegative-integer?
  at-eol? : any/c = #f
```

Like `position-paragraph` in `text%`.

```
(send a-color-textoid paragraph-start-position  paragraph
                                                [visible?])
 → exact-nonnegative-integer?
  paragraph : exact-nonnegative-integer?
  visible? : any/c = #t
```

Like `paragraph-start-position` in `text%`.

```
(send a-color-textoid paragraph-end-position  paragraph
                                              [visible?])
 → exact-nonnegative-integer?
  paragraph : exact-nonnegative-integer?
  visible? : any/c = #t
```

Like paragraph-end-position in text%.

```
(send a-color-textoid skip-whitespace position
                                     direction
                                     comments?)
 → exact-nonnegative-integer?
  position : exact-nonnegative-integer?
  direction : (or/c 'forward 'backward)
  comments? : boolean?
```

Like skip-whitespace in color:text<%>.

```
(send a-color-textoid backward-match position
                                     cutoff)
 → (or/c exact-nonnegative-integer? #f)
  position : exact-nonnegative-integer?
  cutoff : exact-nonnegative-integer?
```

Like backward-match in color:text<%>.

```
(send a-color-textoid backward-containing-sexp position
                                               cutoff)
 → (or/c exact-nonnegative-integer? #f)
  position : exact-nonnegative-integer?
  cutoff : exact-nonnegative-integer?
```

Like backward-containing-sexp in color:text<%>.

```
(send a-color-textoid forward-match position
                                    cutoff)
 → (or/c exact-nonnegative-integer? #f)
  position : exact-nonnegative-integer?
  cutoff : exact-nonnegative-integer?
```

Like forward-match in color:text<%>.

```
(send a-color-textoid classify-position position)
 → (or/c symbol? #f)
  position : exact-nonnegative-integer?
```

Like classify-position in color:text<%>.

```
(send a-color-textoid classify-position* position)
 → (or/c (and/c (hash/c symbol? any/c) immutable?) #f)
  position : exact-nonnegative-integer?
```

Like classify-position* in color:text<%>.

```
(send a-color-textoid get-token-range position)
 → (or/c #f exact-nonnegative-integer?)
    (or/c #f exact-nonnegative-integer?)
  position : exact-nonnegative-integer?
```

Like get-token-range in color:text<%>.

```
(send a-color-textoid get-backward-navigation-limit start)
 → exact-integer?
  start : exact-integer?
```

Like get-backward-navigation-limit in color:text<%>.

```
(send a-color-textoid get-regions)
 → (listof (list/c exact-nonnegative-integer? (or/c exact-nonnegative-integer? 'end)))
```

Like get-regions in color:text<%>.

9 Racket S-Expression Indentation and Navigation

The "syntax-color" collection provides Racket indentation and navigation functions that take advantage of the token categories and parenthesis information produced by a coloring lexer. They can work with any object that implements color-textoid<%>, which is extended by color:text<%>.

Added in version 1.3.

9.1 S-Expression Navigation

```
(require syntax-color/racket-navigation)
                    package: syntax-color-lib
```

```
(racket-forward-sexp text pos)
 → (or/c #f exact-nonnegative-integer?)
  text : (is-a?/c color-textoid<%>)
  pos : exact-nonnegative-integer?
(racket-backward-sexp text pos)
 → (or/c #f exact-nonnegative-integer?)
  text : (is-a?/c color-textoid<%>)
  pos : exact-nonnegative-integer?
(racket-up-sexp text pos) → (or/c #f exact-nonnegative-integer?)
  text : (is-a?/c color-textoid<%>)
  pos : exact-nonnegative-integer?
(racket-down-sexp text pos)
 → (or/c #f exact-nonnegative-integer?)
  text : (is-a?/c color-textoid<%>)
  pos : exact-nonnegative-integer?
```

Each of these functions takes a position *pos* within *text* and returns a position corresponding to S-expression movement. The result is #f if no movement in the corresponding direction is possible.

```
(racket-stick-to-next-sexp? text pos) → boolean?
  text : (is-a?/c color-textoid<%>)
  pos : exact-nonnegative-integer?
```

Returns whether the content at *pos* in *text* corresponds to a token that should "stick" to the following parenthesized sequence for navigation purposes. For example, the result is #t when the token corresponds to ', `, or #'.

9.2 S-Expression Indentation

```
(require syntax-color/racket-indentation)
                    package: syntax-color-lib
```

```
(racket-amount-to-indent  text
                          pos
                         [#:head-sexp-type head-sexp-type]
                          #:graphical-width graphical-width)
 → (or/c #f exact-nonnegative-integer?)
  text : (is-a?/c color-textoid<%>)
  pos : exact-nonnegative-integer?
  head-sexp-type : (string?
                    . -> .
                    (or/c #f 'lambda 'define 'begin 'for/fold 'other))
                 = (racket-tabify-table->head-sexp-type
                    racket-tabify-default-table)
  graphical-width : (or/c #f (-> (is-a?/c color-textoid<%>)
                                 exact-nonnegative-integer?
                                 exact-nonnegative-integer?
                                 (or/c #f exact-nonnegative-integer?)))
```

Returns an amount of indentation to use for the line in *text* that contains the position *pos*. The result may be more or less than the current amount of indentation on the line.

The *head-sexp-type* function is used to map identifiers at the start of an S-expression to the indentation rule that the identifier should use. See compute-racket-amount-to-indent in racket:text<%> for more information.

The *graphical-width* function is used to get the graphical width (distance between the "x" coordinates) of content in *text* between a start and end position. If *graphical-width* returns #f, then characters in *text* are assumed to be all the same width. If *graphical-width* is #f, it is treated the same as if it had been (λ (t start end) #f).

Changed in version 1.5 of package syntax-color-lib: Allow *graphical-width* to return #f.

```
(racket-tabify-table->head-sexp-type spec)
 → (string? . -> . (or/c #f 'lambda 'define 'begin 'for/fold 'other))
  spec : (list/c (hash/c symbol? (or/c 'lambda 'define 'begin 'for/fold))
                 (or/c #f regexp?)
                 (or/c #f regexp?)
                 (or/c #f regexp?)
                 (or/c #f regexp?))
```

Converts a serializable representation *spec* of an indentation configuration to a function suitable for use with racket-amount-to-indent.

The first element of *spec* maps individual symbols to indentation styles. The remaining elements provide patterns to recognize identifiers with the corresponding style, in the order `'lambda`, `'define`, `'begin`, and `'for/fold`.

```
racket-tabify-default-table
 : (list/c (hash/c symbol? (or/c 'lambda 'define 'begin 'for/fold))
           (or/c #f regexp?)
           (or/c #f regexp?)
           (or/c #f regexp?)
           (or/c #f regexp?))
```

A default configuration suitable as an argument to `racket-tabify-table->head-sexp-type`.