Syntax Color: Utilities

Version 7.3

Scott Owens

May 13, 2019

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

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 & 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. For more details, see `start-colorer` in `color:text<%>`.

```
(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<%>`.

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/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-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`.

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? any/c any) any/c))
  in : input-port?
  offset : exact-nonnegative-integer?
  mode : (or/c #f
               (-> input-port? any)
               (cons/c (-> input-port? 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.
- 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.

6 Scribble Lexer

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

```
(scribble-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?
                                  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").

```
(scribble-inside-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?
    any/c
  in : input-port?
  offset : exact-nonnegative-integer?
  mode : any/c
```

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

```
(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.

```
(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.

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.