Syntax Color: Utilities

Version 4.1.1

Scott Owens

October 5, 2008

The `"syntax-color"` collection provides the underlying data structures and some helpful utilities for the `color:text%` class of the §"**Framework**: PLT GUI Application Framework".

1 Parenthesis Matching

```
(require syntax-color/paren-tree)
```

---

paren-tree% : class?
  superclass: object%

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

2 Scheme Lexer

```
(require syntax-color/scheme-lexer)
```

```
(scheme-lexer in) → (or/c string? eof-object?)
                    symbol?
                    (or/c symbol? false/c)
                    (or/c number? false/c)
                    (or/c number? false/c)
  in : input-port?
```

A lexer for Scheme, including reader extensions (§12.9 “Reader Extension”), built specifically for `color:text%`.

The `scheme-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 other symbol 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).

3 Default lexer

```
(require syntax-color/default-lexer)
```

```
(default-lexer in) → (or/c string? eof-object?)
                     symbol?
                     (or/c symbol? false/c)
                     (or/c number? false/c)
                     (or/c number? false/c)
  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).

4 Splay Tree for Tokenization

```
(require syntax-color/token-tree)
```

```
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? false/c)
```

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? false/c)
  n : node?
(node-right n) → (or/c node? false/c)
  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.