RS-COLORS

A Color Data Type for Common Lisp

Ralph Schleicher

This is edition 1, last updated 2020-02-08, of RS-COLORS – *A Color Data Type for Common Lisp*, for RS-COLORS version 1.0.

Please report any errors in this manual to `rs@ralph-schleicher.de`.

Table of Contents

References

[sRGB] http://www.w3.org/Graphics/Color/sRGB.html
http://www.color.org/chardata/rgb/srgb.xalter

[Adobe RGB]
Adobe RGB (1998) Color Image Encoding, Version 2005-05
http://www.color.org/chardata/rgb/adobergb.xalter

[Xcms] Valerie Quercia, Tim O'Reilly: X Window System User's Guide, for X11 Release 5

[HTML] http://www.w3.org/TR/1999/REC-html401-19991224

[CSS3] https://www.w3.org/TR/css-color-3

1 Introduction

By using the RS-COLORS library you can create and manipulate colors, or better color objects, in Common Lisp. A color is either associated with a color model or a color space. The implemented color models and color spaces are:

- generic RGB
- generic HSV/HSB
- generic HSL
- generic CMY
- generic CMYK

and

- CIE RGB
- CIE XYZ
- CIE xyY
- CIE L*u*v*
- CIE L*a*b*
- CIE L*C*h
- sRGB
- Adobe RGB

In Chapter 3 User's Guide, we demonstrate the most important concepts provided by the RS-COLORS library. The details are documented in Chapter 4 Reference Manual.

2 Installation

The RS-COLORS library is available in Quicklisp (`https://www.quicklisp.org`). Thus,

```
(ql:quickload :rs-colors)
(use-package :rs-colors)
```

should do it.

3 User's Guide

3.1 The Color Data Type

First of all, there is not *one* color data type. Instead, every color is an instance of a particular color class. All color classes are sub-classes of the abstract `color-object` class. The built-in color classes are listed in the following tables.

Color Classes for Color Models

`generic-rgb-color`
: Mathematical description of the RGB color model.

`generic-hsv-color`
: Mathematical description of the HSV color space. The HSV color space is a different representation of the RGB color model.

`generic-hsl-color`
: Mathematical description of the HSL color space. The HSL color space is a different representation of the RGB color model.

`generic-cmy-color`
: Mathematical description of the CMY color model.

`generic-cmyk-color`
: Mathematical description of the CMYK color model.

Color Classes for Absolute Color Spaces

`ciergb-color`
: The CIE RGB color space.

`ciexyz-color`
: The CIE XYZ color space.

`ciexyy-color`
: The CIE xyY color space.

`cieluv-color`
: The CIE L*u*v* color space.

`cielab-color`
: The CIE L*a*b* color space.

`cielch-color`
: The CIE L*C*h color space.

Color Classes for Device Dependent Color Spaces

`srgb-color`
: The sRGB color space.

`adobe-rgb-color`
: The Adobe RGB color space.

3.2 Creating Color Objects

Colors are instantiated by calling a constructor function. Constructor arguments are usually the color coordinates in the respective color space. To create, for example, a color in the sRGB color space, say

```
(make-srgb-color 252/255 175/255 62/255)
 ⇒ #<SRGB-COLOR (84/85 35/51 62/255)>
```

Many color coordinates have to be expressed as intensity values, that is values in the range from zero to one inclusive. That's the reason why the sRGB color coordinates in the above example are specified as rational numbers.

Some constructors accept a `:byte-size` keyword argument. This is useful if the scale factor is equal for all color coordinates. With that we can rewrite the above example as

```
(make-srgb-color 252 175 62 :byte-size 8)
 ⇒ #<SRGB-COLOR (84/85 35/51 62/255)>
```

As you can see, the resulting color coordinates are equal. Another common case is to encode the color coordinates in a single integral number. Again, the `:byte-size` keyword argument specifies how many bits are used to encode a single color coordinate. Thus,

```
(make-srgb-color-from-number #XFCAF3E :byte-size 8)
 ⇒ #<SRGB-COLOR (84/85 35/51 62/255)>
```

results in the same color as before.

The built-in constructors are listed in the following table.

`make-generic-rgb-color`
`make-generic-rgb-color-from-number`
: Create a generic RGB color object.

`make-generic-hsv-color`
: Create a generic HSV color object.

`make-generic-hsl-color`
: Create a generic HSL color object.

`make-generic-cmy-color`
`make-generic-cmy-color-from-number`
: Create a generic CMY color object.

`make-generic-cmyk-color`
`make-generic-cmyk-color-from-number`
: Create a generic CMYK color object.

`make-ciergb-color`
: Create a CIE RGB color object.

`make-ciexyz-color`
: Create a CIE XYZ color object.

`make-ciexyy-color`
: Create a CIE xyY color object.

`make-cieluv-color`
: Create a CIE L*u*v* color object.

`make-cielab-color`
: Create a CIE L*a*b* color object.

`make-cielch-color`
: Create a CIE L*C*h color object.

`make-srgb-color`
`make-srgb-color-from-number`
: Create a sRGB color object.

`make-adobe-rgb-color`
`make-adobe-rgb-color-from-number`
: Create an Adobe RGB color object.

3.3 Color Coordinates

Use the `color-coordinates` function to get the color coordinates of a color.

```
(let ((color (make-srgb-color 252 175 62 :byte-size 8)))
  ;; We know that color is an RGB color.
  (multiple-value-bind (r g b)
      (color-coordinates color)
    (list r g b)))
 ⇒ (84/85 35/51 62/255)
```

A more practical way to get the color coordinates of a color is described in Section 3.5 Color Conversion.

3.4 White Point

A device dependent color space usually has a *white point*. If so, the `white-point` function returns a color object of this white point.

3.5 Color Conversion

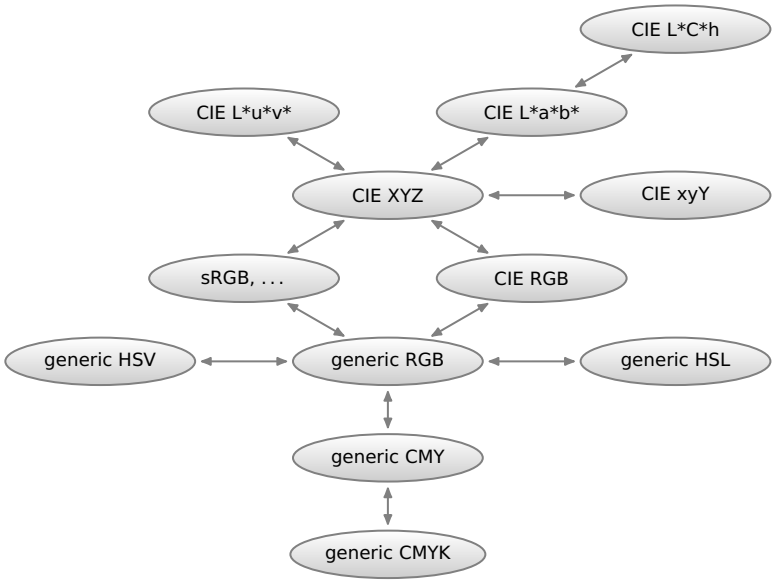

Figure 3.1: Color Conversions

Figure 3.1 depicts the implemented color conversions. The nice thing about RS-COLORS is that all these color conversions can be performed with the `change-class` function.

```
(let ((color (make-srgb-color 252 175 62 :byte-size 8)))
  (values (change-class color 'generic-cmyk-color) color))
 ⇒ #<GENERIC-CMYK-COLOR (0 11/36 95/126 1/85)>
 ⇒ #<GENERIC-CMYK-COLOR (0 11/36 95/126 1/85)>
```

If you wish to keep the original color object unchanged, use the `coerce-color` function.

```
(let ((color (make-srgb-color 252 175 62 :byte-size 8)))
  (values (coerce-color color 'generic-cmyk-color) color))
⇒ #<GENERIC-CMYK-COLOR (0 11/36 95/126 1/85)>
⇒ #<SRGB-COLOR (84/85 35/51 62/255)>
```

The `coerce-color` function only creates a copy of the color if the color object is not already of the correct type.

If you only need the color coordinates, then you can call one of the following functions to get them.

`generic-rgb-color-coordinates`
: Return the generic RGB color space coordinates of a color.

`generic-hsv-color-coordinates`
: Return the generic HSV color space coordinates of a color.

`generic-hsl-color-coordinates`
: Return the generic HSL color space coordinates of a color.

`generic-cmy-color-coordinates`
: Return the generic CMY color space coordinates of a color.

`generic-cmyk-color-coordinates`
: Return the generic CMYK color space coordinates of a color.

`ciergb-color-coordinates`
: Return the CIE RGB color space coordinates of a color.

`ciexyz-color-coordinates`
: Return the CIE XYZ color space coordinates of a color.

`ciexyy-color-coordinates`
: Return the CIE xyY color space coordinates of a color.

`cieluv-color-coordinates`
: Return the CIE L*u*v* color space coordinates of a color.

`cielab-color-coordinates`
: Return the CIE L*a*b* color space coordinates of a color.

`cielch-color-coordinates`
: Return the CIE L*C*h color space coordinates of a color.

`srgb-color-coordinates`
: Return the sRGB color space coordinates of a color.

`adobe-rgb-color-coordinates`
: Return the Adobe RGB color space coordinates of a color.

Let's repeat our example:

```
(let ((color (make-srgb-color 252 175 62 :byte-size 8)))
  (multiple-value-bind (c m y k)
      (generic-cmyk-color-coordinates color)
    (values (list c m y k) color)))
⇒ (0 11/36 95/126 1/85)
⇒ #<SRGB-COLOR (84/85 35/51 62/255)>
```

All these color conversions only work if the color conversion path is unambiguous. The crux is the conversion from the CIE XYZ color space to the generic RGB color space and

vice versa. If you want to go that way, then you have to specify the intermediate color space.

```
(let ((color (make-generic-hsl-color 30 1 1/2)))
  (multiple-value-bind (L* C* h)
      (cielch-color-coordinates
       (if (typep color 'generic-color-object)
           (coerce-color color 'srgb-color)
         color))
    (values (list L* C* h) color)))
 ⇒ (66.9542661879145 8d0 85.58632577011882d0 59.7885844150978d0)
 ⇒ #<GENERIC-HSL-COLOR (30 1 1/2)>
```

Or destructively:

```
(let ((color (make-generic-hsl-color 30 1 1/2)))
  (change-class (change-class color 'srgb-color) 'cielch-color))
 ⇒ #<CIELCH-COLOR (66.95426... 85.58632... 59.78858...)>
```

3.6 External Representations

There are some functions for reading and printing colors in different formats. The RS-COLORS library supports Xcms notation, numerical HTML notation, and CSS3 functional notation.

All functions adhere to the standards set forth by the `read`, `print`, and `format` functions. Thus, they interoperate well with the stream concepts of Common Lisp.

```
(with-input-from-string (input "RGB:4e/9a/06")
  (with-output-to-string (output)
    (print-color-html (read-color-xcms-rgb input) output)))
 ⇒ "#4E9A06"
```

```
(let ((color (make-srgb-color 78 154 6 :byte-size 8)))
  (format nil "background: ~?;" color-formatter-css3-rgb (list color)))
 ⇒ "background: rgb(78, 154, 6);"
```

See Section 4.9 Input and Output, for more details about the supported formats.

3.7 Color Dictionaries

The RS-COLORS library provides several packages with predefined named colors. The recommended way to use these packages is to load it, for example, with QuickLisp. Then you should refer to the named colors via the package prefix to avoid name clashes.

```
;; Load a color dictionary.
(ql:quickload :rs-colors-html)
```

```
;; Use a named color.
(format nil color-formatter-css3-rgb html-color:green)
 ⇒ "rgb(0, 128, 0)"
```

Below is a list of all package prefixes together with their meaning.

`rs-colors-x11`, `x11-color`
: X11 color names, '`(ql:quickload :rs-colors-x11)`'.
  See `https://en.wikipedia.org/wiki/X11_color_names`.

`rs-colors-html`, `html-color`
: HTML basic colors, '`(ql:quickload :rs-colors-html)`'.
  See `https://www.w3.org/TR/css3-color/#html4`.

`rs-colors-svg`, `svg-color`
: SVG color names, ‘`(ql:quickload :rs-colors-svg)`’.
See `https://www.w3.org/TR/css3-color/#svg-color`.

`rs-colors-ral`, `ral-color`
: RAL Classic color names, ‘`(ql:quickload :rs-colors-ral)`’.
See `https://www.ral-farben.de`.

`rs-colors-ral-design`, `ral-design-color`
: RAL Design color names, ‘`(ql:quickload :rs-colors-ral-design)`’.
See `https://www.ral-farben.de`.

`rs-colors-tango`, `tango-color`
: Tango desktop project colors, ‘`(ql:quickload :rs-colors-tango)`’.
See `http://tango.freedesktop.org/Tango_Icon_Theme_Guidelines`.

`rs-colors-material-io`, `material-io-color`
: Material design color palette, ‘`(ql:quickload :rs-colors-material-io)`’.
See `https://material.io/guidelines/style/color.html`.

4 Reference Manual

4.1 Color Predicates

Use the `colorp` function to check whether or not an object is a color. This covers all color classes documented in this manual.

`colorp` *object* [Function]
Return true if *object* is a color object.

4.2 Abstract Color Classes

The color classes documented in this section are merely used as superclasses.

`color-object` [Class]
Base class for a color.

Class Precedence List
`color-object`, `standard-object`, `t`.

`rgb-color-object` [Class]
Color class for a RGB color space.

Color coordinates are the normalized intensities of the red, green, and blue primary. Values are real numbers in the closed interval $[0, 1]$.

Class Precedence List
`rgb-color-object`, `color-object`, ...

`hsv-color-object` [Class]
Color class for a HSV/HSB color space.

Color coordinates are hue, saturation, and value (brightness). Hue is a real number in the half-closed interval $[0, 360)$. Saturation and value are real numbers in the closed interval $[0, 1]$.

Class Precedence List
`hsv-color-object`, `color-object`, ...

`hsl-color-object` [Class]
Color class for a HSL color space.

Color coordinates are hue, saturation, and lightness. Hue is a real number in the half-closed interval $[0, 360)$. Saturation and lightness are real numbers in the closed interval $[0, 1]$.

Class Precedence List
`hsl-color-object`, `color-object`, ...

`cmy-color-object` [Class]
Color class for a CMY color space.

Color coordinates are the normalized intensities of the cyan, magenta, and yellow primary. Values are real numbers in the closed interval $[0, 1]$.

Class Precedence List
`cmy-color-object`, `color-object`, ...

`cmyk-color-object` [Class]

Color class for a CMYK color space.

Color coordinates are the normalized intensities of the cyan, magenta, yellow, and black (key) primary. Values are real numbers in the closed interval $[0, 1]$.

Class Precedence List

`cmyk-color-object`, `color-object`, ...

`generic-color-object` [Class]

Color class for the mathematical model of a color space.

Class Precedence List

`generic-color-object`, `color-object`, ...

4.3 Generic Color Spaces (Color Models)

A generic color space implements a color model. Two color models are in widespread use with computers:

- The additive RGB color model with the primary colors red, green, and blue.
- The subtractive CMY color model with the primary colors cyan, magenta, and yellow.

The RGB color model is the usual color model for computer displays. If the color intensity of all primary colors is zero, that means "off", the display appears "black". Otherwise, if the color intensity of all primary colors is one, that means "on", the display appears "white".

The CMY color model is the usual color model for paper printers. If the color intensity of all primary colors is zero, that means "off", the paper appears "white". Otherwise, if the color intensity of all primary colors is one, that means "on", the paper appears "black".

Theoretically, a RGB tuple (R, G, B) and a CMY tuple (C, M, Y) are related to each other via the simple equations

$$\begin{aligned} C &= 1 - R \\ M &= 1 - G \\ Y &= 1 - B \end{aligned}$$

and

$$\begin{aligned} R &= 1 - C \\ G &= 1 - M \\ B &= 1 - Y \end{aligned}$$

The CMYK color model is an extension of the CMY color model to save ink. Theoretically, a CMY tuple (C, M, Y) and a CMYK quadruple (c, m, y, k) can be related to each other via the equations

$$\begin{aligned} k &= \min(C, M, Y) \\ c &= \frac{C - k}{1 - k} \\ m &= \frac{M - k}{1 - k} \\ y &= \frac{Y - k}{1 - k} \end{aligned}$$

and

$$C = \min(1, c \cdot (1 - k) + k)$$
$$M = \min(1, m \cdot (1 - k) + k)$$
$$Y = \min(1, y \cdot (1 - k) + k)$$

4.3.1 Generic RGB Color Space

The generic RGB color space is a mathematical description of the RGB color model. It is not associated with a particular device.

Color coordinates are the normalized intensities of the red, green, and blue primary. Values are real numbers in the closed interval $[0, 1]$. There is no white point.

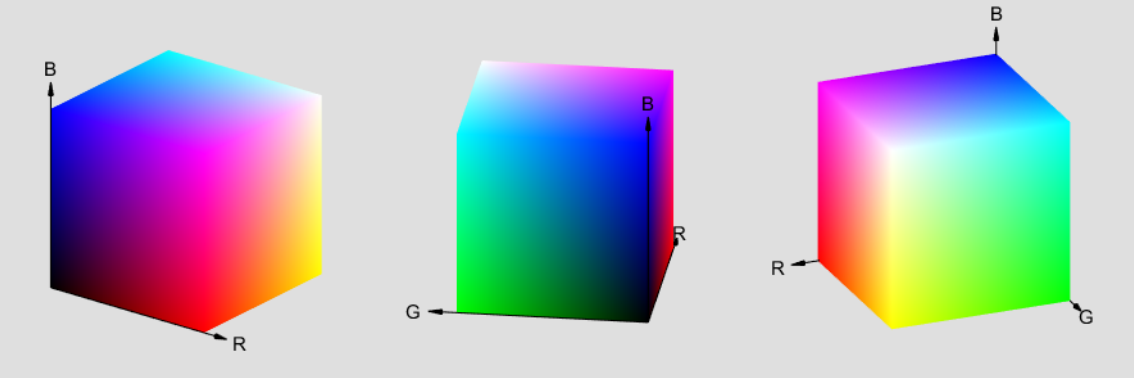

Figure 4.1: RGB Color Space with sRGB Colors

`generic-rgb-color` [Class]

Color class for the generic RGB color space.

Class Precedence List

`generic-rgb-color`, `rgb-color-object`, `generic-color-object`, `color-object`, ...

`make-generic-rgb-color` *red green blue* &key *byte-size* [Function]

Create a new color in the generic RGB color space.

- First argument *red* is the normalized intensity of the red primary.
- Second argument *green* is the normalized intensity of the green primary.
- Third argument *blue* is the normalized intensity of the blue primary.

Arguments *red*, *green*, and *blue* have to be real numbers in the closed interval $[0, 1]$.

Keyword argument *byte-size* is the number of bits used to represent a primary. If specified, arguments *red*, *green*, and *blue* have to be integral numbers in the range from 0 to $2^n - 1$ where n is the number of bits. If so, arguments *red*, *green*, and *blue* are scaled accordingly.

Example:

```
(make-generic-rgb-color 252/255 175/255 62/255)
 ⇒ #<GENERIC-RGB-COLOR (84/85 35/51 62/255)>

(make-generic-rgb-color 252 175 62 :byte-size 8)
 ⇒ #<GENERIC-RGB-COLOR (84/85 35/51 62/255)>
```

`make-generic-rgb-color-from-number` *value* &key *byte-size* [Function]
Create a new color in the generic RGB color space.

- Argument *value* is the numerical value of the encoded RGB color coordinates.

Keyword argument *byte-size* is the number of bits used to represent a primary. Default is eight bit (one byte). Argument *value* has to be an integral number in the range from 0 to $2^{3n} - 1$ where n is the number of bits per primary. The most significant bits denote the intensity of the red primary.

Example:

```
(make-generic-rgb-color-from-number #XFCAF3E)
 ⇒ #<GENERIC-RGB-COLOR (84/85 35/51 62/255)>
```

`generic-rgb-color-coordinates` *color* [Generic Function]
Return the generic RGB color space coordinates of the color.

- Argument *color* is a color object.

Values are the normalized intensities of the red, green, and blue primary.

4.3.2 Generic HSV Color Space

The HSV color space is a different representation of the RGB color model. The HSV color space is also called HSB color space. The generic HSV color space is not associated with a particular device.

Color coordinates are hue, saturation, and value (brightness). Hue is measured in degree angle and it is a real number in the half-closed interval $[0, 360)$. Saturation and value are real numbers in the closed interval $[0, 1]$. There is no white point.

The HSV color space is a cylindrical coordinate system but it is usually visualized as a cone as depicted in Figure 4.2. $H = 0°$ is red, $H = 120°$ is green, and $H = 240°$ is blue. $S = 0$ is the centerline of the cone with the gray levels from black ($V = 0$) to white ($V = 1$). $S = 1$ defines the surface of the cone.

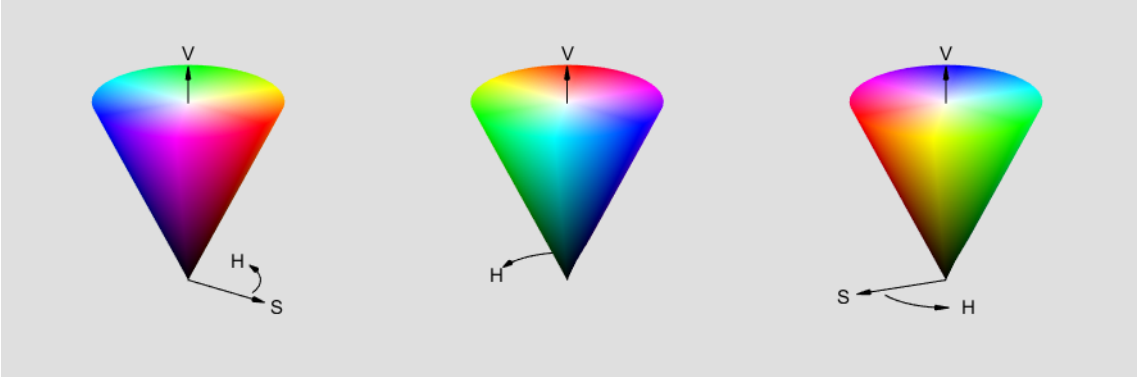

Figure 4.2: HSV Color Space with sRGB Colors

`generic-hsv-color` [Class]
Color class for the generic HSV color space.

Class Precedence List
`generic-hsv-color`, `hsv-color-object`, `generic-color-object`, `color-object`, ...

`make-generic-hsv-color` *hue saturation value* [Function]

Create a new color in the generic HSV color space.

- First argument *hue* is the angle of the RGB color wheel in degree.
- Second argument *saturation* is the saturation.
- Third argument *value* is the brightness.

Argument *hue* has to be a real number. It's value is reduced to the half-closed interval $[0, 360)$. Arguments *saturation* and *value* have to be real numbers in the closed interval $[0, 1]$.

`generic-hsv-color-coordinates` *color* [Generic Function]

Return the generic HSV color space coordinates of the color.

- Argument *color* is a color object.

Values are the hue, saturation, and value (brightness).

4.3.3 Generic HSL Color Space

The HSL color space is a different representation of the RGB color model. The generic HSL color space is not associated with a particular device.

Color coordinates are hue, saturation, and lightness. Hue is measured in degree angle and it is a real number in the half-closed interval $[0, 360)$. Saturation and lightness are real numbers in the closed interval $[0, 1]$. There is no white point.

The HSL color space is a cylindrical coordinate system but it is usually visualized as a bicone as depicted in Figure 4.3. $H = 0°$ is red, $H = 120°$ is green, and $H = 240°$ is blue. $S = 0$ is the centerline of the bicone with the gray levels from black ($L = 0$) to white ($L = 1$). $S = 1$ defines the surface of the bicone.

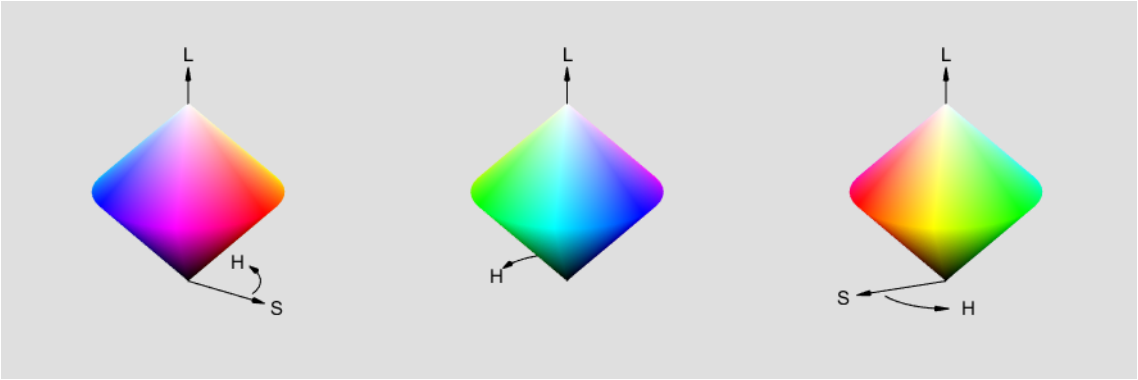

Figure 4.3: HSL Color Space with sRGB Colors

`generic-hsl-color` [Class]

Color class for the generic HSL color space.

Class Precedence List

`generic-hsl-color`, `hsl-color-object`, `generic-color-object`, `color-object`, ...

`make-generic-hsl-color` *hue saturation lightness* [Function]

Create a new color in the generic HSL color space.

- First argument *hue* is the angle of the RGB color wheel in degree.
- Second argument *saturation* is the saturation.

- Third argument *lightness* is the lightness.

Argument *hue* has to be a real number. It's value is reduced to the half-closed interval $[0, 360)$. Arguments *saturation* and *lightness* have to be real numbers in the closed interval $[0, 1]$.

`generic-hsl-color-coordinates` *color* [Generic Function]

Return the generic HSL color space coordinates of the color.

- Argument *color* is a color object.

Values are the hue, saturation, and lightness.

4.3.4 Generic CMY Color Space

The generic CMY color space is a mathematical description of the CMY color model. It is not associated with a particular device.

Color coordinates are the normalized intensities of the cyan, magenta, and yellow primary. Values are real numbers in the closed interval $[0, 1]$. There is no white point.

`generic-cmy-color` [Class]

Color class for the generic CMY color space.

Class Precedence List

`generic-cmy-color`, `cmy-color-object`, `generic-color-object`, `color-object`, ...

`make-generic-cmy-color` *cyan magenta yellow* &key *byte-size* [Function]

Create a new color in the generic CMY color space.

- First argument *cyan* is the normalized intensity of the cyan primary.
- Second argument *magenta* is the normalized intensity of the magenta primary.
- Third argument *yellow* is the normalized intensity of the yellow primary.

Arguments *cyan*, *magenta*, and *yellow* have to be real numbers in the closed interval $[0, 1]$.

Keyword argument *byte-size* is the number of bits used to represent a primary. If specified, arguments *cyan*, *magenta*, and *yellow* have to be integral numbers in the range from 0 to $2^n - 1$ where n is the number of bits. If so, arguments *cyan*, *magenta*, and *yellow* are scaled accordingly.

Example:

```
(make-generic-cmy-color 3/255 80/255 193/255)
 ⇒ #<GENERIC-CMY-COLOR (1/85 16/51 193/255)>

(make-generic-cmy-color 3 80 193 :byte-size 8)
 ⇒ #<GENERIC-CMY-COLOR (1/85 16/51 193/255)>
```

`make-generic-cmy-color-from-number` *value* &key *byte-size* [Function]

Create a new color in the generic CMY color space.

- Argument *value* is the numerical value of the encoded CMY color coordinates.

Keyword argument *byte-size* is the number of bits used to represent a primary. Default is eight bit (one byte). Argument *value* has to be an integral number in the range from 0 to $2^{3n} - 1$ where n is the number of bits per primary. The most significant bits denote the intensity of the cyan primary.

Example:

```
(make-generic-cmy-color-from-number #X0350C1)
 ⇒ #<GENERIC-CMY-COLOR (1/85 16/51 193/255)>
```

`generic-cmy-color-coordinates` *color* [Generic Function]

Return the generic CMY color space coordinates of the color.

- Argument *color* is a color object.

Values are the normalized intensities of the cyan, magenta, and yellow primary.

4.3.5 Generic CMYK Color Space

The generic CMYK color space is a mathematical description of the CMYK color model. It is not associated with a particular device.

Color coordinates are the normalized intensities of the cyan, magenta, yellow, and black (key) primary. Values are real numbers in the closed interval $[0, 1]$. There is no white point.

`generic-cmyk-color` [Class]

Color class for the generic CMYK color space.

Class Precedence List

`generic-cmyk-color`, `cmyk-color-object`, `generic-color-object`, `color-object`, ...

`make-generic-cmyk-color` *cyan magenta yellow black* &key *byte-size* [Function]

Create a new color in the generic CMYK color space.

- First argument *cyan* is the normalized intensity of the cyan primary.
- Second argument *magenta* is the normalized intensity of the magenta primary.
- Third argument *yellow* is the normalized intensity of the yellow primary.
- Fourth argument *black* is the normalized intensity of the black primary.

Arguments *cyan*, *magenta*, *yellow*, and *black* have to be real numbers in the closed interval $[0, 1]$. If *black* is zero, *cyan*, *magenta*, and *yellow* are converted from CMY color coordinates to CMYK color coordinates.

Keyword argument *byte-size* is the number of bits used to represent a primary. If specified, arguments *cyan*, *magenta*, *yellow*, and *black* have to be integral numbers in the range from 0 to $2^n - 1$ where n is the number of bits. If so, arguments *cyan*, *magenta*, *yellow*, and *black* are scaled accordingly.

Example:

```
(make-generic-cmyk-color 3/255 80/255 193/255 0)
 ⇒ #<GENERIC-CMYK-COLOR (0 11/36 95/126 1/85)>

(make-generic-cmyk-color 3 80 193 :byte-size 8)
 ⇒ #<GENERIC-CMYK-COLOR (0 11/36 95/126 1/85)>
```

`make-generic-cmyk-color-from-number` *value* &key *byte-size* [Function]

Create a new color in the generic CMYK color space.

- Argument *value* is the numerical value of the encoded CMYK color coordinates.

Keyword argument *byte-size* is the number of bits used to represent a primary. Default is eight bit (one byte). Argument *value* has to be an integral number in the range from 0 to $2^{4n} - 1$ where n is the number of bits per primary. The most significant bits denote the intensity of the cyan primary.

Example:

```
(make-generic-cmyk-color-from-number #X0350C100)
 ⇒ #<GENERIC-CMYK-COLOR (0 11/36 95/126 1/85)>
```

`generic-cmyk-color-coordinates` *color* [Generic Function]

Return the generic CMYK color space coordinates of the color.

- Argument *color* is a color object.

Values are the normalized intensities of the cyan, magenta, yellow, and black primary.

4.4 CIE Color Spaces

All CIE color spaces are absolute color spaces, that means they are device independent.

4.4.1 CIE RGB Color Space

The CIE RGB color space is the origin of all CIE color spaces.

Color coordinates are the normalized intensities of the red, green, and blue primary. Values are real numbers in the closed interval $[0, 1]$. The white point of the CIE RGB color space is the CIE standard illuminant E[1].

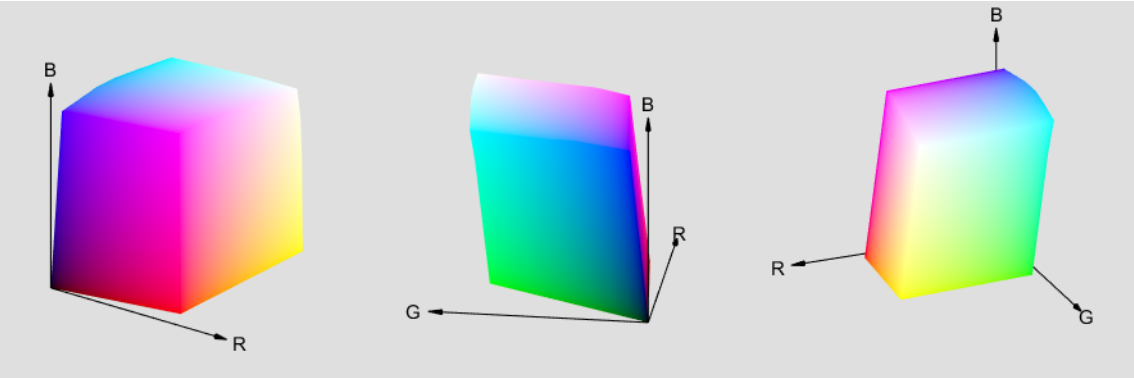

Figure 4.4: CIE RGB Color Space with sRGB Colors

`ciergb-color` [Class]

Color class for the CIE RGB color space.

Class Precedence List

`ciergb-color`, `rgb-color-object`, `color-object`, ...

`make-ciergb-color` *red green blue* [Function]

Create a new color in the CIE RGB color space.

- First argument *red* is the normalized intensity of the red primary.
- Second argument *green* is the normalized intensity of the green primary.
- Third argument *blue* is the normalized intensity of the blue primary.

Arguments *red*, *green*, and *blue* have to be real numbers in the closed interval $[0, 1]$.

`ciergb-color-coordinates` *color* [Generic Function]

Return the CIE RGB color space coordinates of the color.

- Argument *color* is a color object.

Values are the normalized intensities of the red, green, and blue primary.

[1] You can easily check this if you convert CIE RGB white into the CIE xyY color space:

```
(change-class (make-ciergb-color 1 1 1) 'ciexyy-color)
 ⇒ #<CIEXYY-COLOR (1/3 1/3 1)>
```

4.4.2 CIE XYZ Color Space

The CIE XYZ color space is a linear transformation of the CIE RGB color space. The CIE XYZ color space covers all colors an average person can experience. Many other color spaces are defined against the CIE XYZ color space.

Color coordinates are the X, Y, and Z tristimulus values. The CIE XYZ color space has no explicit white point.

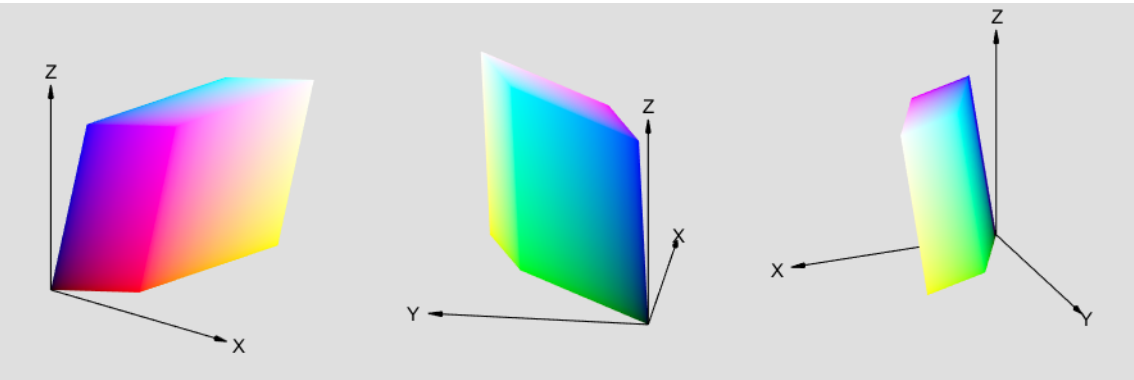

Figure 4.5: CIE XYZ Color Space with sRGB Colors

`ciexyz-color` [Class]

Color class for the CIE XYZ color space.

Class Precedence List

`ciexyz-color`, `color-object`, ...

`make-ciexyz-color` *x y z* [Function]

Create a new color in the CIE XYZ color space.

- First argument x is the X tristimulus value.
- Second argument y is the Y tristimulus value.
- Third argument z is the Z tristimulus value.

Arguments x, y, and z have to be non-negative real numbers.

`ciexyz-color-coordinates` *color* [Generic Function]

Return the CIE XYZ color space coordinates of the color.

- Argument *color* is a color object.

Values are the X, Y, and Z tristimulus values.

Objects of the `ciexyz-color` class can be instantiated with absolute and normalized color coordinates. However, if you want to convert colors from CIE XYZ color space to CIE RGB color space (or any other RGB color space), the CIE XYZ color coordinates have to be normalized color coordinates. See the `normalize-color` and `absolute-color`, for how to convert from absolute color coordinates to normalized color coordinates and vice versa.

4.4.3 CIE xyY Color Space

The CIE xyY color space uses the x and y chromaticity coordinates of the CIE XYZ color space. That is,

$$x = \frac{X}{X+Y+Z}$$
$$y = \frac{Y}{X+Y+Z}$$
$$z = \frac{Z}{X+Y+Z}$$

Simple arithmetic results in the following relations:

$$\frac{Y}{y} = X+Y+Z$$
$$1 = x+y+z$$

Therefore, the inverse transformation is

$$X = x \cdot \frac{Y}{y}$$
$$Y = y \cdot \frac{Y}{y} = Y$$
$$Z = z \cdot \frac{Y}{y} = (1-x-y) \cdot \frac{Y}{y}$$

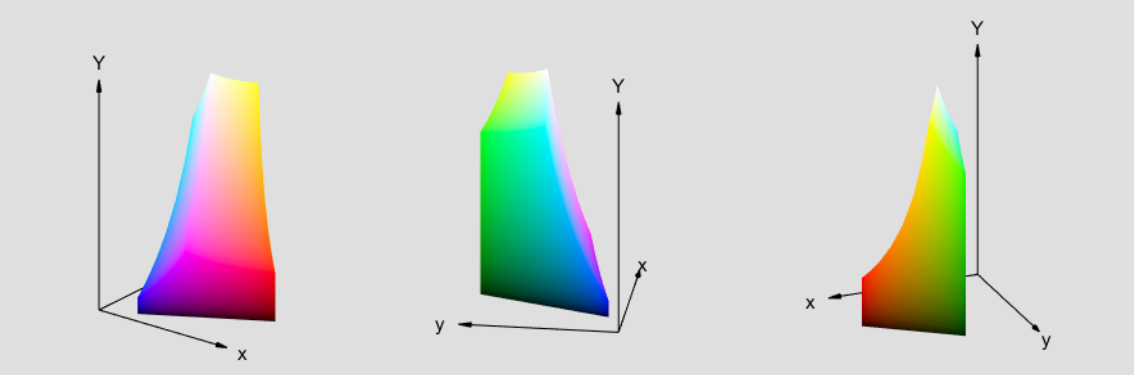

Figure 4.6: CIE xyY Color Space with sRGB Colors

`ciexyy-color` [Class]

Color class for the CIE xyY color space.

Class Precedence List

`ciexyy-color`, `color-object`, ...

`make-ciexyy-color` *x* y* y* [Function]

Create a new color in the CIE xyY color space.

- Arguments *x** and *y** are the x and y chromaticity coordinates of the CIE XYZ color space.
- Third argument y is the luminance, that is the Y tristimulus value of the CIE XYZ color space.

`ciexyy-color-coordinates` *color* [Generic Function]

Return the CIE xyY color space coordinates of the color.

- Argument *color* is a color object.

Values are the x and y chromaticity coordinates and the luminance.

4.4.4 CIE L*u*v* Color Space

The CIE L*u*v* color space is a non-linear transformation of the CIE XYZ color space. The CIE L*u*v* color space is more perceptually uniform than the CIE XYZ color space.

Color coordinates are lightness and two chromaticity coordinates. Lightness L^* is in the range from 0 to 100. However, values greater than 100 are accepted, too. The two chromaticity coordinates u^* and v^* are usually in the range from -100 to $+100$.

CIE L*u*v* color coordinates are always relative to a white point. This is either the white point of the color space you are converting from or CIE standard illuminant D50.

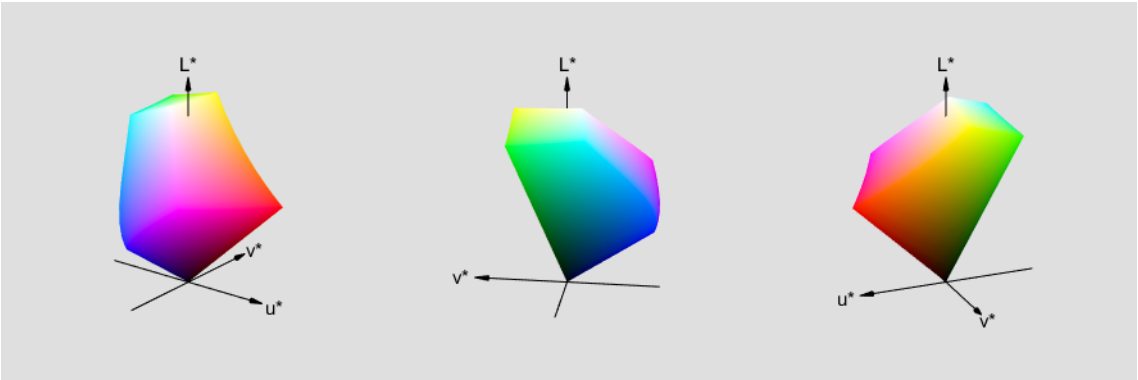

Figure 4.7: CIE L*u*v* Color Space with sRGB Colors

`*cieluv-default-white-point*` [Variable]
The default white point for colors in the CIE L*u*v* color space. Default value is the CIE 1931 D50 standard illuminant.

`cieluv-color` [Class]
Color class for the CIE L*u*v* color space.

Class Precedence List
`cieluv-color`, `color-object`, ...

`make-cieluv-color` *L* u* v** &optional *white-point* [Function]
Create a new color in the CIE L*u*v* color space.

- First argument L^* is the lightness.
- Second argument u^* is the first chromaticity coordinate.
- Third argument v^* is the second chromaticity coordinate.
- Optional fourth argument *white-point* is the white point. Default is the value of `*cieluv-default-white-point*`.

`cieluv-color-coordinates` *color* [Generic Function]
Return the CIE L*u*v* color space coordinates of the color.

- Argument *color* is a color object.

Values are the lightness and the two chromaticity coordinates.

4.4.5 CIE L*a*b* Color Space

The CIE L*a*b* color space is a non-linear transformation of the CIE XYZ color space. The CIE L*a*b* color space is more perceptually uniform than the CIE XYZ color space.

Color coordinates are lightness and two chromaticity coordinates. Lightness L^* is in the range from 0 to 100. However, values greater than 100 are accepted, too. The two

chromaticity coordinates a^* and b^* are usually in the range from -250 to $+250$ and from -100 to $+100$ respectively.

CIE L*a*b* color coordinates are always relative to a white point. This is either the white point of the color space you are converting from or CIE standard illuminant D50.

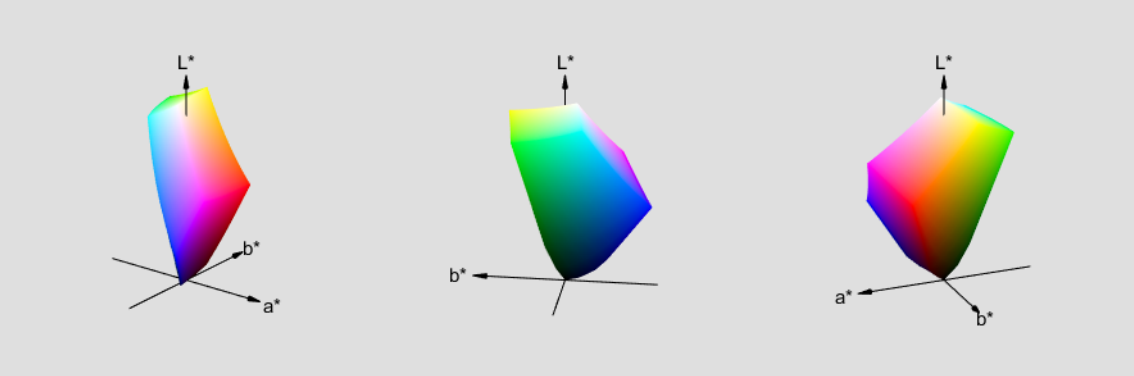

Figure 4.8: CIE L*a*b* Color Space with sRGB Colors

`*cielab-default-white-point*` [Variable]
The default white point for colors in the CIE L*a*b* color space. Default value is the CIE 1931 D50 standard illuminant.

`cielab-color` [Class]
Color class for the CIE L*a*b* color space.

Class Precedence List
`cielab-color`, `color-object`, ...

`make-cielab-color` *L* a* b** &optional *white-point* [Function]
Create a new color in the CIE L*a*b* color space.

- First argument L^* is the lightness.
- Second argument a^* is the first chromaticity coordinate.
- Third argument b^* is the second chromaticity coordinate.
- Optional fourth argument *white-point* is the white point. Default is the value of `*cielab-default-white-point*`.

`cielab-color-coordinates` *color* [Generic Function]
Return the CIE L*a*b* color space coordinates of the color.

- Argument *color* is a color object.

Values are the lightness and the two chromaticity coordinates.

4.4.6 CIE L*C*h Color Space

The CIE L*C*h color space is the transformation of the CIE L*a*b* color space from a Cartesian coordinate system into a cylindrical coordinate system.

Color coordinates are lightness, chroma, and hue. Lightness L^* is equal to the lightness of the CIE L*a*b* color space. Chroma C^* and hue h are the polar coordinates, i.e. radius and angle, of a color in the (a^*, b^*) plane.

Hue is measured in degree angle; $h = 0°$ is the positive a^*-axis (red), $h = 90°$ is the positive b^*-axis (yellow), $h = 180°$ is the negative a^*-axis (green), and $h = 270°$ is the negative b^*-axis (blue).

CIE L*C*h color coordinates are always relative to a white point. This is either the white point of the color space you are converting from or CIE standard illuminant D50.

`*cielch-default-white-point*` [Variable]
The default white point for colors in the CIE L*C*h* color space. Default value is the CIE 1931 D50 standard illuminant.

`cielch-color` [Class]
Color class for the CIE L*C*h color space.

Class Precedence List
`cielch-color`, `color-object`, ...

`make-cielch-color` *L* C* h* &optional *white-point* [Function]
Create a new color in the CIE L*C*h color space.

- First argument *L** is the lightness.
- Second argument *C** is the chroma.
- Third argument *h* is the hue.
- Optional fourth argument *white-point* is the white point. Default is the value of `*cielch-default-white-point*`.

Arguments *L** and *C** have to be non-negative real numbers. Argument *h* has to be a real number. It's value is reduced to the half-closed interval $[0, 360)$.

`cielch-color-coordinates` *color* [Generic Function]
Return the CIE L*C*h color space coordinates of the color.

- Argument *color* is a color object.

Values are the lightness, chroma, and hue.

4.4.7 CIE Standard Illuminants

`cie-1931-white-point-a` [Constant]
`cie-1931-white-point-b` [Constant]
`cie-1931-white-point-c` [Constant]
`cie-1931-white-point-d50` [Constant]
`cie-1931-white-point-d55` [Constant]
`cie-1931-white-point-d65` [Constant]
`cie-1931-white-point-d75` [Constant]
`cie-1931-white-point-e` [Constant]
`cie-1931-white-point-f1` [Constant]
`cie-1931-white-point-f2` [Constant]
`cie-1931-white-point-f3` [Constant]
`cie-1931-white-point-f4` [Constant]
`cie-1931-white-point-f5` [Constant]
`cie-1931-white-point-f6` [Constant]
`cie-1931-white-point-f7` [Constant]
`cie-1931-white-point-f8` [Constant]
`cie-1931-white-point-f9` [Constant]
`cie-1931-white-point-f10` [Constant]
`cie-1931-white-point-f11` [Constant]
`cie-1931-white-point-f12` [Constant]
White points of the CIE standard illuminants for a 2° field of view (2° standard observer).

`cie-1964-white-point-a` [Constant]
`cie-1964-white-point-b` [Constant]
`cie-1964-white-point-c` [Constant]
`cie-1964-white-point-d50` [Constant]
`cie-1964-white-point-d55` [Constant]
`cie-1964-white-point-d65` [Constant]
`cie-1964-white-point-d75` [Constant]
`cie-1964-white-point-e` [Constant]
`cie-1964-white-point-f1` [Constant]
`cie-1964-white-point-f2` [Constant]
`cie-1964-white-point-f3` [Constant]
`cie-1964-white-point-f4` [Constant]
`cie-1964-white-point-f5` [Constant]
`cie-1964-white-point-f6` [Constant]
`cie-1964-white-point-f7` [Constant]
`cie-1964-white-point-f8` [Constant]
`cie-1964-white-point-f9` [Constant]
`cie-1964-white-point-f10` [Constant]
`cie-1964-white-point-f11` [Constant]
`cie-1964-white-point-f12` [Constant]

White points of the CIE standard illuminants for a 10° field of view (10° standard observer).

4.5 RGB Color Spaces

4.5.1 sRGB Color Space

`srgb-color` [Class]

Color class for the sRGB color space.

Class Precedence List

`srgb-color`, `rgb-color-object`, `color-object`, ...

`make-srgb-color` *red green blue* &key *byte-size* [Function]

Create a new color in the sRGB color space.

- First argument *red* is the normalized intensity of the red primary.
- Second argument *green* is the normalized intensity of the green primary.
- Third argument *blue* is the normalized intensity of the blue primary.

Arguments *red*, *green*, and *blue* have to be normalized intensity values in the closed interval $[0, 1]$.

Keyword argument *byte-size* is the number of bits used to represent a primary. If specified, arguments *red*, *green*, and *blue* have to be integral numbers in the range from 0 to $2^n - 1$ where n is the number of bits. If so, arguments *red*, *green*, and *blue* are scaled accordingly.

Example:

```
(make-srgb-color 252/255 175/255 62/255)
 ⇒ #<SRGB-COLOR (84/85 35/51 62/255)>

(make-srgb-color 252 175 62 :byte-size 8)
 ⇒ #<SRGB-COLOR (84/85 35/51 62/255)>
```

`make-srgb-color-from-number` *value* &key *byte-size* [Function]
Create a new color in the sRGB color space.

- Argument *value* is the numerical value of the encoded RGB color coordinates.

Keyword argument *byte-size* is the number of bits used to represent a primary. Default is eight bit (one byte). Argument *value* has to be an integral number in the range from 0 to $2^{3n} - 1$ where n is the number of bits per primary. The most significant bits denote the intensity of the red primary.

Example:

```
(make-srgb-color-from-number #XFCAF3E)
 ⇒ #<SRGB-COLOR (84/85 35/51 62/255)>
```

`srgb-color-coordinates` *color* [Generic Function]
Return the sRGB color space coordinates of the color.

- Argument *color* is a color object.

Values are the normalized intensities of the red, green, and blue primary.

4.5.2 Adobe RGB Color Space

`adobe-rgb-color` [Class]
Color class for the Adobe RGB color space.

Class Precedence List
`adobe-rgb-color`, `rgb-color-object`, `color-object`, ...

`make-adobe-rgb-color` *red green blue* &key *byte-size* [Function]
Create a new color in the Adobe RGB color space.

- First argument *red* is the normalized intensity of the red primary.
- Second argument *green* is the normalized intensity of the green primary.
- Third argument *blue* is the normalized intensity of the blue primary.

Arguments *red*, *green*, and *blue* have to be normalized intensity values in the closed interval $[0, 1]$.

Keyword argument *byte-size* is the number of bits used to represent a primary. If specified, arguments *red*, *green*, and *blue* have to be integral numbers in the range from 0 to $2^n - 1$ where n is the number of bits. If so, arguments *red*, *green*, and *blue* are scaled accordingly.

Example:

```
(make-adobe-rgb-color 252/255 175/255 62/255)
 ⇒ #<ADOBE-RGB-COLOR (84/85 35/51 62/255)>

(make-adobe-rgb-color 252 175 62 :byte-size 8)
 ⇒ #<ADOBE-RGB-COLOR (84/85 35/51 62/255)>
```

`make-adobe-rgb-color-from-number` *value* &key *byte-size* [Function]
Create a new color in the Adobe RGB color space.

- Argument *value* is the numerical value of the encoded RGB color coordinates.

Keyword argument *byte-size* is the number of bits used to represent a primary. Default is eight bit (one byte). Argument *value* has to be an integral number in the range from 0 to $2^{3n} - 1$ where n is the number of bits per primary. The most significant bits denote the intensity of the red primary.

Example:

```
(make-adobe-rgb-color-from-number #XFCAF3E)
 ⇒ #<ADOBE-RGB-COLOR (84/85 35/51 62/255)>
```

`adobe-rgb-color-coordinates` *color* [Generic Function]
Return the Adobe RGB color space coordinates of the color.

- Argument *color* is a color object.

Values are the normalized intensities of the red, green, and blue primary.

4.6 Color Properties

`color-coordinates` *color* [Generic Function]
Return the color space coordinates of the color as multiple values.

- Argument *color* is a color object.

`white-point` *color* [Generic Function]
Return the white point of the color.

- Argument *color* is a color object.

Value is the color object of the color's white point, or nil if the white point is not defined or if multiple white points exist.

4.7 Color Conversions

`change-class` *color color-type* [Generic Function]
Change the class of the color object.

- First argument *color* is a color object.
- Second argument *color-type* is a color data type.

The `change-class` function destructively modifies *color* by converting it's color coordinates into the color space denoted by *color-type*.

Example:

```
(let ((red (make-srgb-color 1 0 0)))
  (change-class red 'ciexyy-color)
  red)
 ⇒ #<CIEXYY-COLOR (0.64d0 0.33d0 ...)>
```

`coerce-color` *color color-type* [Function]
Coerce the color object into the specified color type.

- First argument *color* is a color object.
- Second argument *color-type* is a color data type.

If argument *color* is already a color of the requested color data type, return *color* as is (no conversion). Otherwise, return a new color with the color coordinates of *color* converted into the color space denoted by *color-type*.

`copy-color` *color* [Generic Function]
Return a shallow copy of the color.

- Argument *color* is a color object.

Value is a color object with the same color coordinates as *color*.

4.8 Color Differences

`cie76` *first-color second-color* [Function]
Calculate the CIE76 color difference between two colors.

Value is the Euclidean distance between the two colors in the CIE L*a*b* color space. The CIE76 color difference is symmetric, i.e. CIE76(a,b) is equal to CIE76(b,a).

`cie94` *reference other* &optional *textile lightness chroma hue* [Function]
Calculate the CIE94 color difference between two colors.

- First argument *reference* is the reference color.
- Second argument *other* is the other color.
- If optional third argument *textile* is non-null, use parameters for calculating the color difference for textiles. Default is to calculate the color difference for graphic arts.
- Optional fourth to sixth argument *lightness*, *chroma*, and *HUE* are the weighting factors for differences in lightness, chroma, and hue respectively. Higher value means less weight. Default is one for all weighting factors (if *textile* is true, the default for *lightness* is two).

The CIE94 color difference is asymmetric, i.e. CIE94(a,b) is not equal to CIE94(b,a).

4.9 Input and Output

4.9.1 Xcms Formats

The syntax of a Xcms (X Color Management System) color is

prefix:*first*/*second*/*third*

The *prefix* part specifies the color space or format and *first*, *second*, and *third* are the color coordinates. Below is a table with all color spaces and corresponding Xcms prefix.

Color Space	Prefix	R5
CIE RGB	`CIERGB`	
CIE XYZ	`CIEXYZ`	•
CIE xyY	`CIExyY`	•
CIE L*u*v*	`CIELuv`	•
CIE L*a*b*	`CIELab`	•
CIE L*C*h	`CIELCh`	
generic RGB	`RGBi`	•
generic RGB	`RGB`	•

The Xcms prefix is case insensitive.

`print-color-xcms-ciergb` *color* &optional *stream* [Function]
`print-color-xcms-ciexyz` *color* &optional *stream* [Function]
`print-color-xcms-ciexyy` *color* &optional *stream* [Function]
`print-color-xcms-cieluv` *color* &optional *stream* [Function]
`print-color-xcms-cielab` *color* &optional *stream* [Function]
`print-color-xcms-cielch` *color* &optional *stream* [Function]
`print-color-xcms-rgbi` *color* &optional *stream* [Function]
`print-color-xcms-rgb` *color* &optional *stream* [Function]
Print a color in Xcms notation.

- First argument *color* is a color object.

- Optional second argument *stream* is an output stream. Default is to print to `*standard-output*`.

Value is the color object.

The RGBi and RGB formats are device dependent. Thus, you can only print colors in these formats if the color conversion path to the generic RGB color space is unambiguous.

The RGB format prints the color coordinates as hexadecimal numbers. If all RGB color intensities are multiples of 1/255, it uses two digits. Otherwise, it uses four digits with a precision of 1/65535.

Example:

```
(let ((color (make-srgb-color 78 154 6 :byte-size 8)))
  (with-output-to-string (stream)
    (print-color-xcms-ciexyz color stream)))
 ⇒ "CIEXYZ:0.14729778/0.24743336/0.04172078"

(let ((color (make-srgb-color-from-number #X4E9A06)))
  (with-output-to-string (stream)
    (print-color-xcms-rgb color stream)))
 ⇒ "RGB:4e/9a/06"
```

`color-formatter-xcms-ciergb` [Constant]
`color-formatter-xcms-ciexyz` [Constant]
`color-formatter-xcms-ciexyy` [Constant]
`color-formatter-xcms-cieluv` [Constant]
`color-formatter-xcms-cielab` [Constant]
`color-formatter-xcms-cielch` [Constant]
`color-formatter-xcms-rgbi` [Constant]
`color-formatter-xcms-rgb` [Constant]

A format function for printing a color in Xcms notation.

Value is a function which has a behavior equivalent to a function returned by the `formatter` macro.

Example:

```
(let ((color (make-srgb-color 78 154 6 :byte-size 8)))
  (format nil color-formatter-xcms-ciexyz color))
 ⇒ "CIEXYZ:0.14729778/0.24743336/0.04172078"
```

`read-color-xcms-ciergb` &optional *stream* [Function]
`read-color-xcms-ciexyz` &optional *stream* [Function]
`read-color-xcms-ciexyy` &optional *stream* [Function]
`read-color-xcms-cieluv` &optional *stream* [Function]
`read-color-xcms-cielab` &optional *stream* [Function]
`read-color-xcms-cielch` &optional *stream* [Function]
`read-color-xcms-rgbi` &optional *stream* [Function]
`read-color-xcms-rgb` &optional *stream* [Function]
`read-color-xcms` &optional *stream* [Function]

Read a color in Xcms notation.

- Optional argument *stream* is an input stream. Default is to read from `*standard-input*`.

Value is a color object in the respective color space.

The functions expect the respective Xcms prefix followed by a colon and the three color coordinates separated by a slash character. Leading or trailing whitespace is not ignored. Reading stops immediately after the last color coordinate. An error is signaled if a parse error occurs or if the file ends prematurely.

The `read-color-xcms` function is slightly different since it first reads the Xcms prefix and then dispatches on it.

Example:

```
(with-input-from-string (stream "RGB:4e/9a/06 junk")
  (change-class (read-color-xcms stream) 'srgb-color))
 ⇒ #<SRGB-COLOR (26/85 154/255 2/85)>
```

4.9.2 HTML Format

A HTML color value is either a hexadecimal number prefixed by a hash mark or a keyword. This section only covers numerical color values.

`print-color-html` *color* &optional *stream* [Function]

Print a numerical HTML color value, that is a hexadecimal number prefixed by a hash mark.

- First argument *color* is a color object.
- Optional second argument *stream* is an output stream. Default is to print to `*standard-output*`.

Value is the color object.

Example:

```
(let ((color (make-srgb-color-from-number #X4E9A06)))
  (with-output-to-string (stream)
    (print-color-html color stream)))
 ⇒ "#4E9A06"
```

`color-formatter-html` [Constant]

A format function for printing a numerical HTML color value.

Value is a function which has a behavior equivalent to a function returned by the `formatter` macro.

Example:

```
(let ((color (make-srgb-color-from-number #X4E9A06)))
  (format nil color-formatter-html color))
 ⇒ "#4E9A06"
```

`read-color-html` &optional *stream* [Function]

Read a numerical HTML color value, that is a hexadecimal number prefixed by a hash mark.

- Optional argument *stream* is an input stream. Default is to read from `*standard-input*`.

Value is a color object in the sRGB color space.

The function expects a hash mark followed by a sequence of hexadecimal digits. The number of hexadecimal digits has to be a multiple of three. Leading or trailing whitespace is not ignored. Reading stops at the first non-hexadecimal digit character. An error is signaled if a parse error occurs or if the file ends prematurely.

Example:

```
(with-input-from-string (stream "#4E9A06 junk")
  (read-color-html stream))
 ⇒ #<SRGB-COLOR (26/85 154/255 2/85)>
```

4.9.3 CSS3 Formats

A CSS3 (cascading style sheets, level 3) color is either a HTML color value or a color value in functional notation. The later has the form

`rgb(`*red*`, `*green*`, `*blue*`)`

where *red*, *green*, and *blue* are either integers in the range from 0 to 255 or percentage values in the range from 0 % to 100 %. CSS3 also supports a functional notation for RGB colors represented in the HSL color space. The format is

`hsl(`*hue*`, `*saturation*`, `*lightness*`)`

where *hue* is the angle in degree and *saturation* and *lightness* are percentage values in the range from 0 % to 100 %.

`print-color-css3-rgb` *color* &optional *stream* [Function]
`print-color-css3-hsl` *color* &optional *stream* [Function]

Print a color in CSS3 functional notation.

- First argument *color* is a color object.
- Optional second argument *stream* is an output stream. Default is to print to `*standard-output*`.

Value is the color object.

If all sRGB color intensities are multiples of 1/255, the *red*, *green*, and *blue* color intensities are printed as integers. Otherwise, they are printed as percentage values.

Example:

```
(let ((color (make-srgb-color-from-number #X4E9A06)))
  (with-output-to-string (stream)
    (print-color-css3-rgb color stream)))
 ⇒ "rgb(78, 154, 6)"

(let ((color (make-srgb-color 0.3D0 0.6D0 0.1D0)))
  (with-output-to-string (stream)
    (print-color-css3-rgb color stream)))
 ⇒ "rgb(30.0%, 60.0%, 10.0%)"
```

`color-formatter-css3-rgb` [Constant]
`color-formatter-css3-hsl` [Constant]

A format function for printing a color in CSS3 functional notation.

Value is a function which has a behavior equivalent to a function returned by the `formatter` macro.

Example:

```
(let ((color (make-srgb-color-from-number #X4E9A06)))
  (format nil color-formatter-css3-rgb color))
 ⇒ "rgb(78, 154, 6)"
```

`read-color-css3-rgb` &optional *stream* [Function]
`read-color-css3-hsl` &optional *stream* [Function]

Read a color in CSS3 functional notation.

- Optional argument *stream* is an input stream. Default is to read from `*standard-input*`.

Value is a color object in the sRGB color space.

The functions expect a character sequence of the form `rgb(`*red*`,`*green*`,`*blue*`)` or `hsl(`*hue*`,`*saturation*`,`*lightness*`)` respectively. Leading or trailing whitespace is not

ignored but whitespace characters are allowed around the numerical values. Reading stops after the closing parenthesis. An error is signaled if a parse error occurs or if the file ends prematurely.

Example:

```
(with-input-from-string (stream "rgb(78, 154, 6) junk")
  (read-color-css3-rgb stream))
 ⇒ #<SRGB-COLOR (26/85 154/255 2/85)>

(with-input-from-string (stream "rgb(30%, 60%, 10%) junk")
  (read-color-css3-rgb stream))
 ⇒ #<SRGB-COLOR (0.3d0 0.6d0 0.1d0)>
```

`read-color-css3` &optional *stream* [Function]

Read a CSS3 color value, i.e. either a numerical HTML color value or a color in RGB or HSL functional notation.

- Optional argument *stream* is an input stream. Default is to read from `*standard-input*`.

Value is a color object in the sRGB color space.

See the `read-color-html`, `read-color-css3-rgb`, and `read-color-css3-hsl` function for more details.

4.10 Miscellaneous

`normalize-color` *color* &key *black white* [Generic Function]

Convert from absolute color coordinates to normalized color coordinates.

`absolute-color` *color* &key *black white* [Generic Function]

Convert from normalized color coordinates to absolute color coordinates.

Symbol Index

*

A

C

G

H

M

N

P

R

S

W

Concept Index