code-printer Manual

Naoki Kaneko a.k.a puripuri2100

目次

1. 簡単な使い方 1

2. シンタックスハイライトを付ける 3

3. デザインを弄る 13

4. Syntax と Theme の作り方 20

4.1. Syntax の作り方 20

4.2. Theme の作り方 23

5. 必要なバージョンや依存など 25

6. バグ報告・修正や機能追加の提案 25

7. ライセンスとコピーライト 25

このライブラリはソースコードの表示のための機能を提供します。 ソースコードのシンタックスハイライトやコードの枠の装飾などを行い、かつユーザーが自身の手で表示部分を調節することができるようになっています。LaTeX での listings パッケージと同じような機能を持っています。

1. 簡単な使い方

satyrographos でインストールした後に、`@require: code-printer/code-printer`をプリアンブルに書くことでコマンドが使えるようになります。モジュール名は`CodePr`

inter です。

提供するコマンドは

- \inline-code
- +code-printer
- \code-printer
- +file-printer
- \file-printer

の 5 つです。この 5 つのコマンドの型はどれも [code-printer-config?; string] となっており、「シンタックスハイライトや装飾に関する設定」を省略可能なオプション引数で与え、次に、表示したいコードそのものもしくは表示したいファイルへのパスを string 型で与える形になっています。

\inline-code コマンドは行中にコードを表示する際に使用します。+code-printer と \code-printer はどちらもブロックでコードを表示する際に使います。\code-printer は inline-text 中に書けるようになっているだけで、効果は +code-printer と変わりません。+file-printer と \file-printer はファイルへのパスを与えると、その中身をコードとして +code-printer に与えた時と同じように表示します。

シンタックスハイライトや装飾についての設定を一切行わないときは非常にシンプルになります。例えば、左のようなコードを与えると右のように出力されます。

```
+code-printer
(`fn main() {
  println!("Hi!");
}`);
```

```
1 fn main() {
2   println!("Hi!");
3 }
```

見てわかる通り、+code-printer コマンドに与えられたコードが表示されていると思います。デフォルトでは行番号が付き、背景が灰色の枠付きのコード表示となっています。

2. シンタックスハイライトを付ける

code-printer ライブラリではコードの解析を行う部分と、解析結果に色付けなどをする部分を切り離しています。プログラミング言語ごとに解析用の設定を纏めて一つの定数として提供し、それとは別にテーマごとに色やフォントを変えるための設定を纏めた定数として提供しています。ユーザーはこれらを組み合わせて自分好みの表示を作成するようになっています。

解析用の設定値とテーマ用の設定値を組み合わせてコマンドに渡すための設定値を生成するには、`CodePrinter`モジュールで提供されている`make-config`を使う必要があります。`make-config`関数は二つ引数を取り、一つ目は解析用の設定値で二つ目がテーマ用の設定値です。この二つを渡して生成した設定値はそのままコマンドのオプショナル引数に渡すことで動作します。つまり、

```
+code-printer ?:(
    CodePrinter.make-config syntax theme
)(`code`);
```

のようにすることで、`syntax`で設定された通りにコードが解析され、その結果が`theme`で設定された通りに描画される、ということです。

ここでの`syntax`と`theme`に相当する設定値はユーザー自身が作成することもできますが、code-printer ライブラリの提供する機能の一つとして既にいくつかのパターンを用意しています。ここではこのライブラリで提供している設定値を紹介します。

デフォルトの`syntax`と`theme`の設定値はそれぞれ`CodePrinter.default-syntax`と`CodePrinter.default-theme`として公開されています。

プログラミング言語ごとに纏められた解析用の設定は、code-syntax.satyg ファイルによって提供されています。`@require: code-printer/code-syntax`によって読み込んでください。モジュール名は`CodeSyntax`です。

テーマ用の設定値は code-theme.satyh ファイルによって提供されています。ファイルの先頭に`@require: code-printer/code-theme`と書いて読み込んでください。モジュール名は`CodeTheme`です。

まずは解析用の設定値の紹介です。その設定値を使って解析した結果のシンタックスハイライトと一緒に載せています。

```
@require: base/int
module Demo : sig
  val hi:string -> string
end = struct
  % 挨拶をする関数
  let hi name =
    `Hi, `# ^ name
end
```

CodeSyntax.satysfi

```
open Pervasives


(* comment *)
let main () =
  print_string "Hi!\n"
```

CodeSyntax.ocaml

```
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeSyntax.rust

```
IDENTIFICATION  DIVISION.
PROGRAM-ID. HELLO.
PROCEDURE DIVISION.
MAIN SECTION.
  DISPLAY "Hello World!".
STOP RUN.
```

CodeSyntax.cobol

```
#include <stdio.h>
main ()
{
  printf("Hello World!")
}
```

CodeSyntax.c

```
#include <iostream>
int main(){
  std::cout << "Hi!";
  return 0;
}
```

CodeSyntax.cpp

```csharp
using System;
namespace Demo {
  class Demo {
    static void Main() {
      Console.
        WriteLine("Hi!");
    }
  }
}
```

CodeSyntax.csharp

```d
import std.stdio;


void main() {
  writeln("Hello World!");
}
```

CodeSyntax.d

```erlang
%% comment
hello() ->
  io:fwrite("Hi")
```

CodeSyntax.erlang

```fsharp
// F#
(*
  block comment
*)
let x = 2 + 3
printfn "%d" x
```

CodeSyntax.fsharp

```fortran
! comment
program hello
  write(*,*) "Hi!"
  stop
end program hello
```

CodeSyntax.fortran

```go
package main
import "fmt"
func main() {
  fmt.Printf("Hi!")
}
```

CodeSyntax.go

```haskell
{-
  block comment
-}
main =
  let x = 2 + 3 in
  print x
```

CodeSyntax.haskell

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" ↵
     ↳ >
  </head>
  <body>
    <!--
      comment
    -->
    <p>Hi!</p>
  </body>
</html>
```

CodeSyntax.html

```java
class Demo {
  public static void
    main(String[ ] args) {
    int x;
    x = 2 + 3;
    System.out.println(x);
  }
}
```

CodeSyntax.java

```javascript
// comment
var x = 2 + 3;
console.log(x);
```

CodeSyntax.javascript

```json
{
  "a" : "hoge",
  "b" : [12, 34]
}
```

CodeSyntax.json

```
// comment
{
  a : "hoge",
  b : [12, 34],
  c : {"あいう":'aiu'},
}
```

CodeSyntax.json5

```julia
# comment
println("Hi!")
```

CodeSyntax.julia

```kotlin
// comment
fun main() {
  println("Hi!")
}
```

CodeSyntax.kotlin

```lua
-- comment
print("Hello World!")
```

CodeSyntax.lua

```lisp
; comment
(format t "Hello World!")
```

CodeSyntax.lisp

```nim
# comment
let x = 2 + 3
echo x
```

CodeSyntax.nim

```makefile
all:
  satysfi doc/code-printe ↵
   ↳ r-ja.saty
```

CodeSyntax.makefile

```shell
echo "Hello World!"
```

CodeSyntax.shell

```perl
# comment
print "Hello World!";
```

CodeSyntax.perl

```prolog
desc(X, Y) :- child(X, Y).
```

CodeSyntax.prolog

```php
<?php
  echo "<p>Hi!</p>";
?>
```

CodeSyntax.php

```python
# commnet
print('Hello World!')
```

CodeSyntax.python

```r
# comment
print("Hello World!")
```

CodeSyntax.r

```ruby
# comment
puts "Hello World!"
```

CodeSyntax.ruby

```scala
// comment
object Demo {
  def main() = {
    println("Hi!")
  }
}
```

CodeSyntax.scala

```swift
// comment
print("Hello World!")
```

CodeSyntax.swift

```tex
\documentclass{article}
\begin{document}
  % comment
  Hello {\LaTeX}!
\end{document}
```

CodeSyntax.tex

```toml
key = "key"
int = 4

[table]
  array = [-1.5, +99.5]
```

CodeSyntax.toml

```typescript
// comment
console.log("Hello World! ↵
 ↳ ");
```

CodeSyntax.typescript

```visualbasic
Module Demo
  Sub Main()
    Console.
      WriteLine("Hi!")
  End Sub
End Module
```

CodeSyntax.visualbasic

```xml
<?xml version="1.0"?>
<document>
  <!-- comment -->
  <P>
    Hi!
  </P>
</document>
```

CodeSyntax.xml

現在提供している解析用の設定値は以上になります。

次はテーマ用の設定値の紹介です。

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

`CodeTheme.basic-light`

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

`CodeTheme.basic-dark`

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

`CodeTheme.basic-font`

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

`CodeTheme.gruvbox-light`

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

`CodeTheme.gruvbox-dark`

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

`CodeTheme.dracula`

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.iceberg-light

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.iceberg-dark

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.tokyo-night

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.tokyo-night-strom

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.tokyo-night-light

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.ayu-dark

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.ayu-mirage

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.ayu-light

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.spacegray-eighties

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.spacegray-mocha

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

Cod

eTheme.spacegray-ocean-dark

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

Cod

eTheme.spacegray-ocean-light

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.night-owl

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.light-owl

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.winteriscoming-dark-blue

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.winteriscoming-dark-black

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.winteriscoming-light

```rust
fn main () {
  let s = "Hi!";
  let mut cl = s.chars();
  let head = cl.nth(0);
  // head == Some('H')
  println!("{:?}", head);
}
```

CodeTheme.one-light

現在提供しているテーマ用の設定値は以上になります。

3. デザインを弄る

CodePrinterモジュールでは、コードブロックのデザインを変更する機能を提供しています。

変更できる項目は以下のようになっています。

- フォントサイズの指定（フォントサイズを変更するテーマを使用した場合は上書きされます）
- 行番号の出力結果の指定
- 行ごとに色を変えるかどうかの選択と、変更先の色の指定
- 改行されたときに、その場所に出力するものの指定
- タブ文字をどの大きさで表すかの指定
- inline-codeコマンドの出力結果の装飾
- inline-codeコマンドの出力結果の周囲の余白
- コードブロックの装飾
- コードブロック周囲の余白

フォントサイズの指定はset-basic-font-sizeという関数で行います。一つ目の引数の値がフォントサイズを表すlength型です。

```
+code-printer ?:(
  CodePrinter.make-config syntax theme
  |> CodePrinter.set-basic-font-size 10pt;
) (`code`);
```

のようにして使います。

行番号の出力結果の変更はset-number-funという関数で行います。この関数は一つ目の引数が「コードを組むときに使う context データと、行数を表す整数値を受け取って、行番号を表す inline-boxes 型のデータを返す関数」となっています。ですので、一つ目の引数の型はcontext -> int -> inline-boxesとなっています。そのこと除けばをフォントサイズ設定用の関数と使い方は同じです。例えば、

```
+code-printer ?:(
  CodePrinter.make-config
    CodeSyntax.satysfi
    CodeTheme.basic-light
  |> CodePrinter.set-number-fun (fun ctx i -> (
    let ctx = set-text-color Color.black ctx in
    let number-it = i |> arabic |> embed-string in
    read-inline ctx {#number-it;行目}
  ))
) (`let x = 1 in
x * 2`);
```

と設定すると、

```
1行目 let x = 1 in
2行目 x * 2
```

という風に出力されます。code-design.satyh ファイル（提供しているモジュールの名称は`CodeDesign`）では行番号出力結果を変更するための`set-number-fun`の一つ目の引数に渡すことができる関数をいくつか提供しています。それは

- `number-fun-null`：何も出力しないための関数
- `number`：行番号のみを出力する関数

の 2 つです。

このライブラリでは「行ごとに色を変えるかどうかの選択と、変更先の色の指定」ができるようになっています。 この指定は`set-line-background-color`という関数で行います。「特定の行数を目立たせたい」・「偶奇で背景色を変えたい」という時に役に立ちます。

`set-line-background-color`関数の一つ目の引数には「行番号を表す整数値を受け取って`color option`を返す関数」を渡します。この関数の方は`int -> color option`です。色を変更したい行番号が来た時には`Some(Color.red)`のように、値を返します。このとき指定した色によって自動的にその行の背景色が変更されます。また、「色を変更しない」という指定をするときには`None`を返します。これを返すと背景色は変更

されません。 `set-line-background-color` の使い方はフォントサイズ設定用の関数と同じです。

デフォルトでは「行番号に関わらず `None` が指定され、背景色は変更されない」という状態になっています。

例えば、特定の関数の使用部分を強調したいときに、以下の様にコードを書くと、

```
+code-printer ?:(
  CodePrinter.make-config
    CodeSyntax.rust CodeTheme.basic-light
  |> CodePrinter.set-line-background-color (fun i -> (
    if i == 7 then
      Some(Color.yellow)
    else
      None
  ))
) (`fn f(i: usize) -> usize {
  i + 5
}

fn main() {
  let x = 5;
  let y = f(5);
  println!("{}", y);
}`);
```

以下の様に出力され、7 行目の背景が黄色になり、その他の行では元から設定されている背景色のままでいることがわかります。

```
1 fn f(i: usize) -> usize {
2   i + 5
3 }
4
5 fn main() {
```

```
6     let x = 5;
7     let y = f(5);
8     println!("{}", y);
9   }
```

「改行されたときに、その場所に出力するものの指定」を行うことができます。改行箇所の直前と直後に入れる inline-boxes をそれぞれ指定できます。そのための関数は`set-line-break-mark`で、型は`(context -> length -> (inline-boxes * inline-boxes)) -> code-printer-config -> code-printer-config`です。一つ目の引数に与える関数は「context と、一文字分の横幅を基に、改行前後に入れる inline-boxes をペアで生成する」というものになります。インデントは自動で入るため、その分は考えないでください。

デフォルトでは矢印のグラフィックを改行直前と改行直後に入れています。

これを、「改行直前には何も入れず、改行直後には文字の矢印を入れる」という設定にしてみます。以下の様にコードを書くと、

```
+code-printer ?:(
  CodePrinter.make-config
    CodeSyntax.rust CodeTheme.basic-light
  |> CodePrinter.set-line-break-mark (fun ctx default-w↵
   ↳idth -> (
    let before-mark =
      let mark-ib =
        `→`
        |> embed-string
        |> read-inline ctx
      in
      mark-ib ++ (inline-skip (default-width *' 0.5))
    in
    (inline-nil, before-mark)
  ))
) (`fn main() {
  let x = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1↵
```

```
    ↳ 3, 14, 15];
  println!("{:?}", x);
}`);
```

以下の様に出力され、

```
fn main() {
  let x = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
  → , 14, 15];
  println!("{:?}", x);
}
```

改行直前には何も入らず、改行直後が文字の矢印となっているのがわかります。また、文字の矢印の後ろに少しだけスペースが入っているのも、設定した通りとなっています。

「タブ文字をどの大きさで表すかの指定」は、`set-tab-size`関数で行います。型は`float -> code-printer-config -> code-printer-config`です。一つ目の引数で「タブ文字が一文字の横幅の何倍で出力されるか」を指定します。デフォルトでは`2.0`です。

「`inline-code`コマンドの出力結果の装飾」は`set-inline-hdeco-set`という関数で行います。これは、一つ目の引数に「背景色の`color`型のデータを受け取って装飾を表す deco-set データを返す」関数です。「`inline-code`コマンドの出力結果の周囲の余白」の設定は`set-inline-paddings`関数で行います。この関数は一つ目の引数に`length`4 つのタプルを受け取ります。どちらも`inline-frame-breakable`プリミティブに渡す値です。また、これも使い方はフォントサイズ設定用の関数と変わりません。

例えば、現在はインラインコードの背景は角が丸みを帯びていますが、これを無くして完全な長方形にし、余白を無くそうと考えたら、この設定関数を使います。

以下の様にコードを書くと、

```
+p{普通のインラインコード：\inline-code ?:(CodePrinter.make ↵
 ↳ -config CodePrinter.default-syntax CodeTheme.basic- ↵
```

```
 ↳ dark) (`code`);}
+p{背景の角を落とし、余白を無くすインラインコード：
  \inline-code ?:(
    CodePrinter.make-config
      CodePrinter.default-syntax
      CodeTheme.basic-dark
    |> CodePrinter.set-inline-hdeco-set (fun backcolo ↵
     ↳ r -> (
      let deco (x, y) w h d =
        [
          fill backcolor (
            Gr.rectangle
              (x, y +' h)
              (x +' w, y -' d)
          )
        ]
      in
      (deco, deco, deco, deco)
    ))
    |> CodePrinter.set-inline-paddings (0pt,0pt,0pt,0 ↵
     ↳ pt)
  ) (`code`);}
```

以下の様に出力されます。

普通のインラインコード：`code`

背景の角を落とし、余白を無くすインラインコード：`code`

しっかりと意図したとおりに出力されていることがわかります。

「コードブロックの装飾」は`set-block-vdeco-set`という関数で行います。これは、一つ目の引数に「背景色の`color`型のデータを受け取って装飾を表す deco-set データを返す」関数です。「コードブロックの出力結果の周囲の余白」の設定は`set-block-paddings`関数で行います。どちらも`block-frame-breakable`プリミティブに渡す関数で、それ以外は`inline-code`の装飾の設定用の関数と変わりません。

ここでも同様に「指定された色で背景色だけを塗り、余白無し」のデザインを設定してみます。以下の様にコードを書くと、

```
+code-printer ?:(
    CodePrinter.make-config
      CodeSyntax.rust
      CodeTheme.basic-dark
    |> CodePrinter.set-block-vdeco-set (fun backcolor -> (
      let deco (x, y) w h d =
        [
          fill backcolor (
            Gr.rectangle
              (x, y +' h)
              (x +' w, y -' d)
          )
        ]
      in
      (deco, deco, deco, deco)
    ))
    |> CodePrinter.set-block-paddings (0pt,0pt,0pt,0pt )
  ) (`fn main() {
  let lst = vec![1, 3, 5, 7, 9];
  println!("{:?}", lst);
}`);
```

以下のように出力されます。

```
fn main() {
  let lst = vec![1, 3, 5, 7, 9];
  println!("{:?}", lst);
}
```

意図した通り、「背景色塗るだけで余白無し」になっていますね。

4. Syntax と Theme の作り方

CodePrinter.make-config で使用する Syntax と Theme は、ユーザーが作成することもできます。ここではその作り方を説明します。

4.1. Syntax の作り方

Syntax は CodePrinter.syntax 型によって表現されます。この型の実装は公開されていません。この型を生成するためには CodePrinter.make-syntax 関数を使う必要があります。この関数は

```
(|
  line-comment : syntax-rule;
  block-comment : syntax-rule;
  string : syntax-rule;
  keywords : string list;
  identifier : syntax-rule;
  others : (string * syntax-rule) list;
|)
```

というレコードを引数に取ることで CodePrinter.syntax 型のデータを生成します。レコードのそれぞれのラベルの意味はそれぞれ、

- line-comment：一行で終了するコメント
- block-comment：開始文字列と終了文字列で挟まれたコメント
- string：文字列
- keywords：特別扱いとなる単語
- identifier：その他関数名や変数名、数字など
- others： markdown などの、上記の区分けに分類できないものを表すためのものです。

keywords に与えるものは、そのキーワードそのものを表す文字列のリストを与えます。リストの先頭から評価されるため、先にある文字列が部分文字列となっている文字列がある

と、その文字列は上手く評価されません。例えば、`[`do`; `double`]`というリストを与えた時、`do`が先に判定に使われるため、`double`が上手く評価されなくなります。この場合は順番を入れ替えて`[`double`; `do`]`とすると上手く評価されます。

その他のラベルに与える`syntax-rule`というデータは、syntax を表すデータです。定義は公開されていません。このデータは

- `CodePrinter.syntax-rule-line`
- `CodePrinter.syntax-rule-block`
- `CodePrinter.syntax-rule-fun`

の 3 つの関数でそれぞれ作成することができます。

`syntax-rule-line`関数の型は`string -> syntax-rule`です。その構文のルールを表す正規表現文字列を与えることで、`syntax-rule`を生成します。この正規表現文字列は SATYSFI の`regexp-of-string`プリミティブによって SATYSFI 内部の正規表現型に変換され、SATYSFI の`string-scan`プリミティブに渡されます。

`regexp-of-string`は内部で OCaml の Str ライブラリで提供されている`regexp`という関数を使用しています。そのため、この`syntax-rule-line`に渡すことができる正規表現文字列は以下の構文を持ちます。

- `.`：改行を除くすべての文字にマッチします。
- `*`：後置で使います。先行する正規表現の 0 回以上の繰り返しにマッチします。
- `+`：後置で使います。先行する正規表現の 1 回以上の繰り返しにマッチします。
- `?`：後置で使います。先行する正規表現の 0 回もしくは 1 回の出現にマッチします。
- `^`：マッチさせる文字の先頭、もしくは改行文字の直後にマッチします。
- `$`：マッチさせる文字の末尾、もしくは改行文字の直前にマッチします。
- `\|`：中置で使います。二つの正規表現のどちらかにマッチさせたいときに使用します。
- `[ .. ]`：文字集合を表します。`[a-zA-Z]`のようにして使い、`-`で範囲を表します。`-`そのものを表したいときは、`-`を集合の先頭か末尾に書きます。`[^0-9]`のように、集合の先頭に`^`を書くと、補集合を表します。

- `\( .. \)`：正規表現をグループ化します。
- `\`：特殊文字をエスケープします。次の文字が特殊文字です。`^`・`$`・`.`・`*`・`+`・`?`・`\`・`[`・`]`

`string-scan`というプリミティブは内部で OCaml の Str ライブラリで提供されている`string-match`という関数を使用しています。`string-scan`とうプリミティブは、与えられた文字列と正規表現について、その文字列の先頭から始まる部分文字列でその正規表現にマッチするものがあった場合はその部分文字列を返し、マッチする部分文字列が無かった場合は`None`を返す。という挙動をします。

例えば、「整数を`identifier`に分類したい」という場合は、

```
identifier = syntax-rule-line `0\|-?[1-9][0-9]*`;
```

という風に書きます。

`syntax-rule-block`は`string -> string -> syntax-rule`という型を持っていて、一つ目の引数に「開始部分の文字列にマッチする正規表現を表す文字列」を与え、二つ目の引数には終了部分のそれを与えます。正規表現を表す文字列のルールについては`syntax-rule-line`と同じです。終了部分の文字列については、もっとも近い場所にあるものが採用されます。終了部分の文字列がマッチしたときにはじめて、開始部分の文字列の先頭から終了部分の文字列の末尾までの全ての文字が登録されます。

例えば、「`%`からはじまり、改行文字で終わる文字列をコメントに分類したい」という場合は

```
line-comment = syntax-rule-block `%` `$`;
```

という風に書きます。

`syntax-rule-fun`は正規表現では表すことのできない複雑なルールに対応する際に使用します。型は`(string -> (string * string) option)-> syntax-rule`です。「文字列を受け取り、それを読み進めていき、もしルールにマッチする 1 文字目から始まる部分文字列があった場合は、その部分文字列と残りの文字列のペアを返し、マッチしなかった場合は`None`」を返す、という挙動をする関数を与えてください（`string`

-scan の挙動を手動で行うイメージです)。

「何もマッチさせたくない」というときはこの syntax-rule-fun を用い、

```
block-comment = syntax-rule-fun (fun _ -> None);
```

という風に書きます。

others ラベルに与えるものは、syntax-rule とそのルールに付ける名前のペアのリストです。line-comment や string といったラベルを独自に作ることができるイメージです。リストの先頭から評価されます。もし、このルールにマッチした文字列があり、さらにそのルールに付けたラベルに対応する装飾が theme の方にも定義されていた場合、その Theme に従って装飾されます。そのラベルに対応する装飾が Theme の方に定義されていなかった場合はその他の文字と同じ扱いになります。

4.2. Theme の作り方

Theme は CodePrinter.theme 型によって表現されます。この型の実装は公開されていません。この型を生成するためには CodePrinter.make-theme 関数を使う必要があります。この関数は

```
1 (|
2   backcolor : color;
3   basic : context -> context;
4   comment : context -> context;
5   string : context -> context;
6   keyword : context -> context;
7   identifier : context -> context;
8   others : (string * (context -> context)) list;
9 |)
```

というレコードを引数に採ることで theme データを生成します。

backcolor ラベルには、そのコードの背景色を color 型で与えます。

`basic`ラベルには`context -> context`の関数を与えます。この関数に外部から与えられた文脈型が適用されたものが文字を組む際に使用されます。ここでフォントの変更や文字色の変更を行います。[*1]これによって生成された文脈型を便宜的に「basic 文脈」と呼びます。この basic 文脈は Syntax で`line-comment`・`block-comment`・`string`・`keywords`・`identifier`・`others`のいずれにも分類されなかった文字に適用されます。また、`others`に分類されても対応する装飾が無かった文字列もこの basic 文脈が適用されます。

- `comment`
- `string`
- `keyword`
- `identifier`

に渡す関数は`basic`に渡す関数と同じく、フォントや文字色などを変える関数です。ただし、その関数に与えられる文脈型は basic 文脈です。

`others`に渡すものは、`make-syntax`関数に渡すレコードの`others`に渡されたルールに対応するラベルと、それに対応する文脈型を変更する関数のリストです。

---

1 厳密には、コマンド外部の文脈型に対して

- `set-font`
- `set-leading`
- `set-paragraph-margin`
- `set-min-gap-of-lines`

で調節した後に`set-basic-font-size`が適用された文脈型がこの関数に与えられ、それによって生成された新しい文脈型に対して

- `set-hyphen-penalty`
- `set-space-ratio`
- `set-space-ratio-between-scripts`

でさらに調節を加えた後に文字を組む際に使用されます。

5. 必要なバージョンや依存など

このライブラリは`read-file`プリミティブを使用しています。 そのため、SATySF$_{\mathrm{I}}$ のバージョン 0.0.6 より後、特に`56895a1`コミット以降のものを使用してください。

また、このライブラリは SATySF$_{\mathrm{I}}$ の標準ライブラリと、satysfi-base と satysfi-fonts-dejavu の 2 つの外部ライブラリに依存しています。 それぞれのインストールは satyrographos を使用することを想定しています。

6. バグ報告・修正や機能追加の提案

このパッケージはバグが存在するかもしれません。 バグを発見した場合は以下の URL に報告してください。

```
https://github.com/puripuri2100/satysfi-code-printer/issues
```

このパッケージに対してコードの修正や機能追加の提案をしたい場合は、GitHub の機能を用いて以下の URL にプルリクエストを送ってください。

```
https://github.com/puripuri2100/satysfi-code-printer/pulls
```

バグ報告・修正提案・機能追加提案をお待ちしております。

7. ライセンスとコピーライト

このパッケージとドキュメントは MIT ライセンスのもとで配布されます。