# The Railway Package Manual

Your Name

`satysfi-railway` は SATySFi のパスを楽に描くためのパッケージです。

SATySFi では標準で直線や Bezier 曲線を描く機能が付いているものの、曲線のパーツを使い回すことは容易ではなく、規則的で対称的な図形を描くためにはユーザが点の座標をいろいろと計算する必要がありました。`satysfi-railway` はパスの情報を相対座標で保持する `Rail.rail` 型を用いることで、一度作成した曲線を容易に使い回せるようにしました。単純な形をした玩具のレールもつなぎ合わせれば複雑な形の線路ができるように、複数の rail を繋げれば複雑な図形をも楽に作成することができるようになります。以下、具体例を交えて説明します。

# 1. Gallery

`satysfi-railway` は `Rail` モジュールを提供します。`Rail` には `Rail.rail` 型があり、`Rail` モジュール内にあるすべての関数はすべて `rail` 型の作成、結合、変換などに関連する機能を持ちます。

## 1.1. 作成

`rail` 型の値は `Rail.init` によって簡単に作成することができます。

```
let rail = Rail.init in rail
```

しかしこれでできるのはいわば「空のパス」であり、このままでは何の意味もありません。`Rail.push-line` および `Rail.push-curve` で線を追加することができます。

```
let rail = Rail.init
  |> Rail.push-line (50pt, 20pt)
  |> Rail.push-curve ((50pt, -20pt), (10pt, 10pt), (-10pt, -10pt))
in
```

`push-line` は `vector` 型の引数を持ちます。これは `length * length` 型のシノニムですが、「点の位置」というより「ベクトル」と考えた方が近いものです。`push-line x` は「起点からベクトル `x` の方向に直線を引く」ということを意味します。

`push-line` は `vector * vector * vector` 型の引数を持ち、

- 目的地のベクトル
- 曲線開始時の制御ハンドルの相対座標
- 曲線終了時の制御ハンドルの相対座標

を表します。たとえば上の例では

- 最終的に x 軸方向に `50pt` 上、y 軸方向に `20pt` 下の位置に向かう
- 曲線の開始時の制御ハンドルは `(10pt, 10pt)`
- 曲線の終了時の制御ハンドルは `(-10pt, -10pt)`

となっており、確かにその通りに線が引かれています。

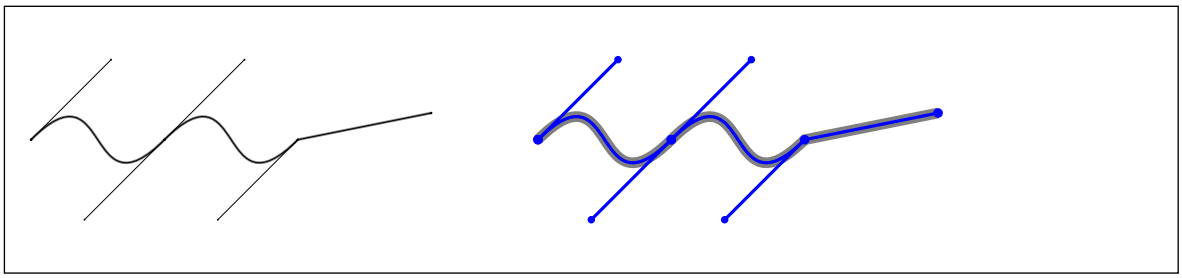
## 1.2. 描画

せっかく `rail` を作っても、パスやグラフィックスとして描画できなければ意味がありません。`rail` は `to-path`, `to-loop` 関数によって `path` 型に変換できます。ともに `point` と `rail` 型を引数に持ち、与えられた `point` を開始点の座標として持つような `rail` をパスの形式で出力します。`to-loop` 関数の場合、開始点と終了点は直線で結ばれます。ループを描きたいときは、基本的に開始点と終了点の座標を一致させるのがよいでしょう。

```
let rail1 = Rail.init
  |> Rail.push-curve ((50pt, 0pt), (30pt, 30pt), (-30pt, -30pt))
  |> Rail.push-smooth-curve (50pt, 0pt) (-30pt, -30pt)
  |> Rail.push-line (50pt, 10pt)
in
[
  rail1 |> Rail.to-path (10pt, 30pt) |> stroke 1pt Color.red;
  rail1 |> Rail.to-loop (10pt, 20pt) |> stroke 1pt Color.black;
]
```

デバッグ用途には gr-debug 関数が役に立ちます。これは to-path と同様の引数で graphics list 型を出力するものであり、rail に関する多くの情報を表示することができます。具体的には点の位置およびハンドルが表示されるようになります。gr-debug では線の太さや色を変更することもできます。

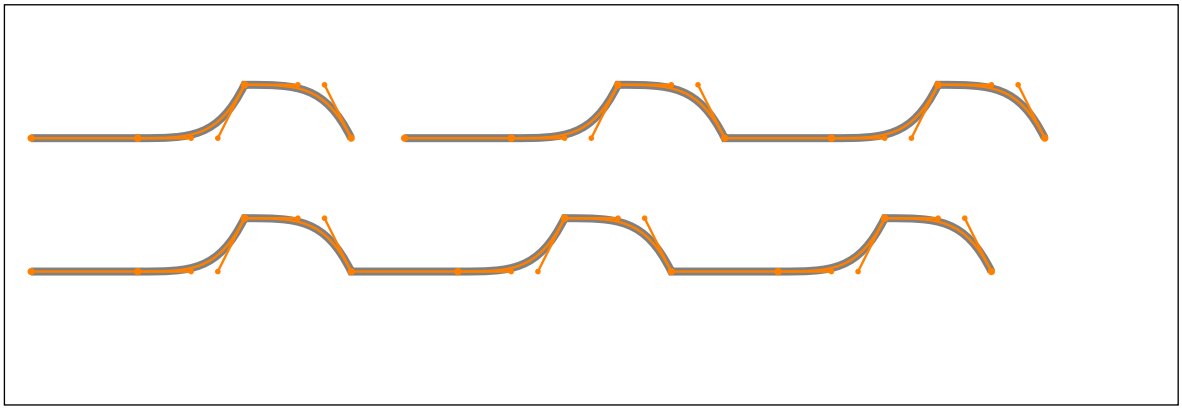
```
[
  rail1 |> Rail.gr-debug (10pt, 50pt);
  rail1 |> Rail.gr-debug ?:(Color.blue) ?:(4pt) (200pt, 50pt);
] |> List.concat
```



## 1.3. 結合

rail 同士は自由に結合することができます。

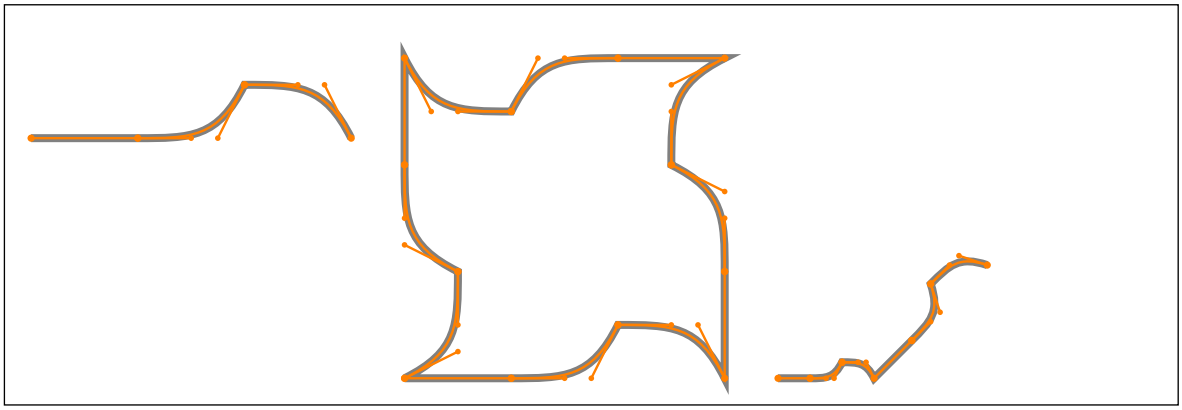
```
let rail1 = Rail.from-curves [
  ((40pt, 0pt), (0pt, 0pt), (0pt, 0pt));
  ((40pt, 20pt), (20pt, 0pt), (0pt, -20pt));
  ((40pt, -20pt), (20pt, 0pt), (0pt, 20pt));
]
in
let rail2 = rail1 |> Rail.append rail1 in
let rail3 = Rail.concat [ rail1; rail1; rail1; ] in
```



## 1.4. 変換

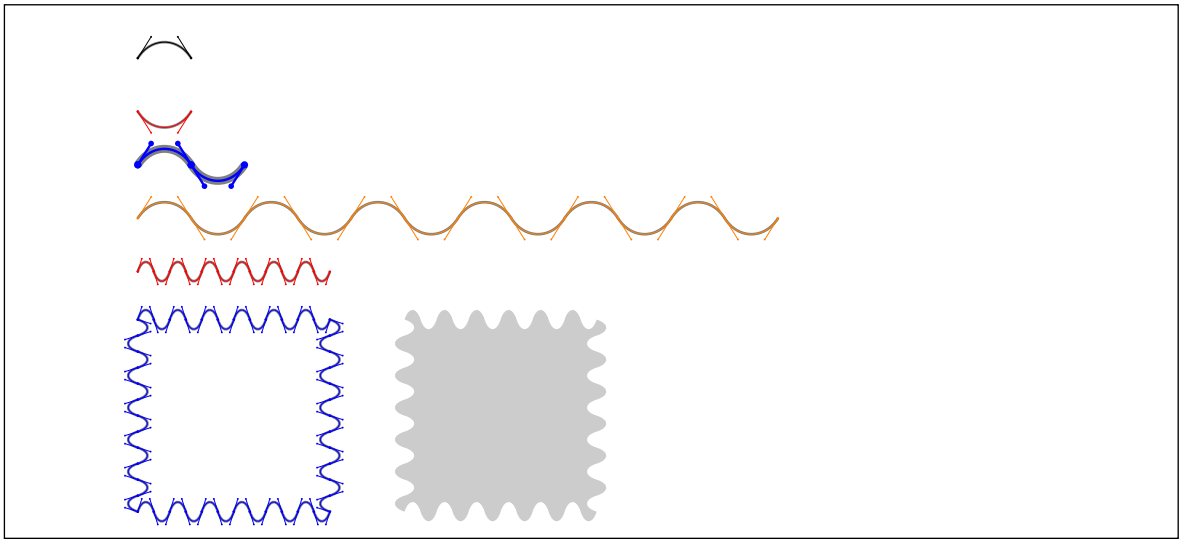
`rail` は結合だけでなく、回転やスケール変換などを施すことができます。

```
let rail1 = Rail.from-curves [
  ((40pt, 0pt), (0pt, 0pt), (0pt, 0pt));
  ((40pt, 20pt), (20pt, 0pt), (0pt, -20pt));
  ((40pt, -20pt), (20pt, 0pt), (0pt, 20pt));
]
in
let rail2 = Rail.(
  concat [ rail1; rail1 ^ 90.; rail1 ^ 180.; rail1 ^ 270.; ]
)
in
let rail3 =
  Rail.(rail1 * 0.3 --- ((rail1 * 0.5) ^ 45.) )
in
```



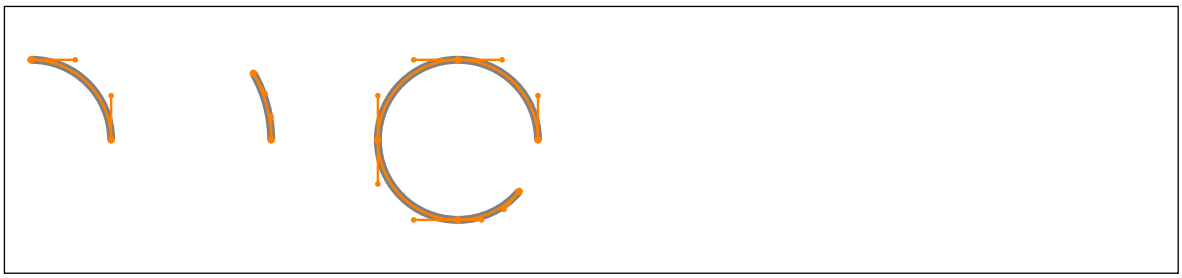
そのほかにも様々な関数が提供されています。ちょっと遊んでみましょう。

```
let rail1 = Rail.from-curves [((20pt, 0pt), (5pt, 8pt), (-5pt, 8pt))]
in
let rail2 = rail1 |> Rail.reflect-y in
let rail3 = Rail.concat [rail1; rail2] in
let rail4 = rail3 |> Rail.repeat 6 in
let rail5 = rail4 |> Rail.scale 0.3 0.3 in
let rail6 =
  Rail.map-repeat (fun i -> Rail.(rail5 ^ ((float i) *. 90.0))) 4
in
let path = rail6 |> Rail.to-loop (150pt, 10pt) in
[
  rail1 |> Rail.gr-debug                    (50pt, 180pt);
  rail2 |> Rail.gr-debug ?:Color.red        (50pt, 160pt);
  rail3 |> Rail.gr-debug ?:Color.blue ?:3pt (50pt, 140pt);
  rail4 |> Rail.gr-debug ?:Color.orange     (50pt, 120pt);
  rail5 |> Rail.gr-debug ?:Color.red        (50pt, 100pt);
  rail6 |> Rail.gr-debug ?:Color.blue       (50pt, 10pt);
  [fill (Color.gray 0.8) path];
] |> List.concat
```



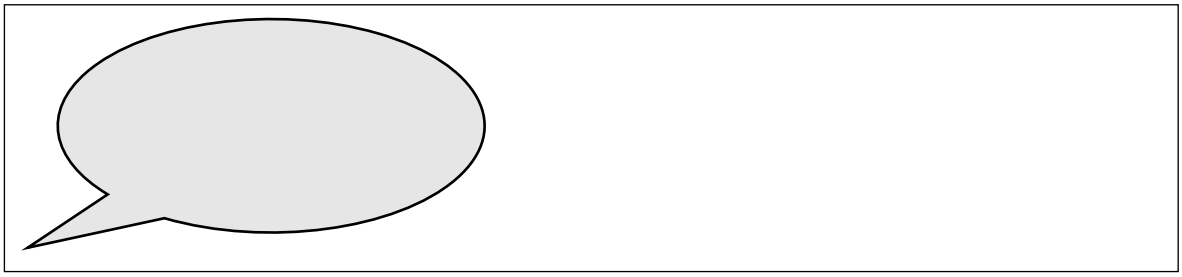
曲線の中で最もよく用いられるのは円弧です。`Rail.circular-sector` を用いれば、任意の角度の円弧を描画することができます。デフォルトでは四分円を描きますが、オプション引数によって円弧の角度を指定することができます。

```
let rail1 = Rail.circular-sector 30pt in
let rail2 = Rail.circular-sector ?:30.0 50pt in
let rail3 = Rail.circular-sector ?:(320.) 30pt in
```



これを使えば、楕円形の吹き出しだって簡単に作れますね。

```
let speech-bubble =
  Rail.(
    (((circular-sector ?:(340.) 40pt) ^ 240.) |> Rail.scale 1.8 1.0)
      --- Rail.from-curves [((-20pt, -20pt), (0pt, 0pt), (0pt, 0pt))]
  )
in
```

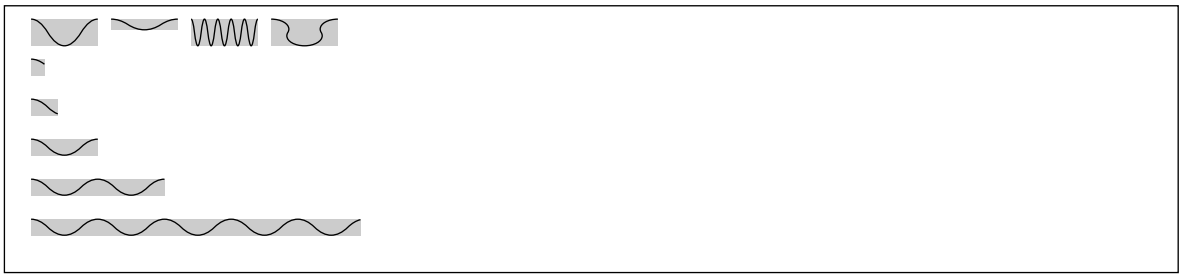


Rail の応用として、WavyLine というモジュールが用意されています。これを用いると、任意の長さ、波長を持つなめらかな波線を引くことができます。

```
let wave-example conf (x, y) len =
  [
    fill (Color.gray 0.8)
      (Gr.rectangle (x, y) (x +' len, y -' conf#amplitude *' 2.));
    WavyLine.new len conf
      |> Rail.to-path (x, y) |> stroke 0.5pt Color.black;
  ]
in
[
  wave-example (| amplitude = 5pt; lambda = 25pt; slope-ratio = 100.;
|) ( 10pt, 95pt) 25pt;
  wave-example (| amplitude = 2pt; lambda = 25pt; slope-ratio = 100.;
|) ( 40pt, 95pt) 25pt;  % 振幅変更
  wave-example (| amplitude = 5pt; lambda =  5pt; slope-ratio = 100.;
|) ( 70pt, 95pt) 25pt;  % 波長変更
  wave-example (| amplitude = 5pt; lambda = 25pt; slope-ratio = 0.5 ;
|) (100pt, 95pt) 25pt;  % 波形変更

  % 任意の長さで切り取ることが可能（波の途中で切っても OK）
  wave-example (| amplitude = 3pt; lambda = 25pt; slope-ratio = 100.;
|) ( 10pt, 80pt) 5pt;
  wave-example (| amplitude = 3pt; lambda = 25pt; slope-ratio = 100.;
|) ( 10pt, 65pt) 10pt;
  wave-example (| amplitude = 3pt; lambda = 25pt; slope-ratio = 100.;
|) ( 10pt, 50pt) 25pt;
```

```
  wave-example (| amplitude = 3pt; lambda = 25pt; slope-ratio = 100.;
|) ( 10pt, 35pt) 50pt;
  wave-example (| amplitude = 3pt; lambda = 25pt; slope-ratio = 100.;
|) ( 10pt, 20pt) 123.45pt;
] |> List.concat
```



`WavyLine` によって波線を引く `\uwave` コマンドも用意されています。 The quick fox jumps over the 怠惰な犬.