実践で役立つ関数型

2013年1月18日

エンジニアサポートCROSS 2013

セッション：関数プログラミングの波に乗り遅れるな

小笠原　啓

IT Planning, Inc.

アジェンダ

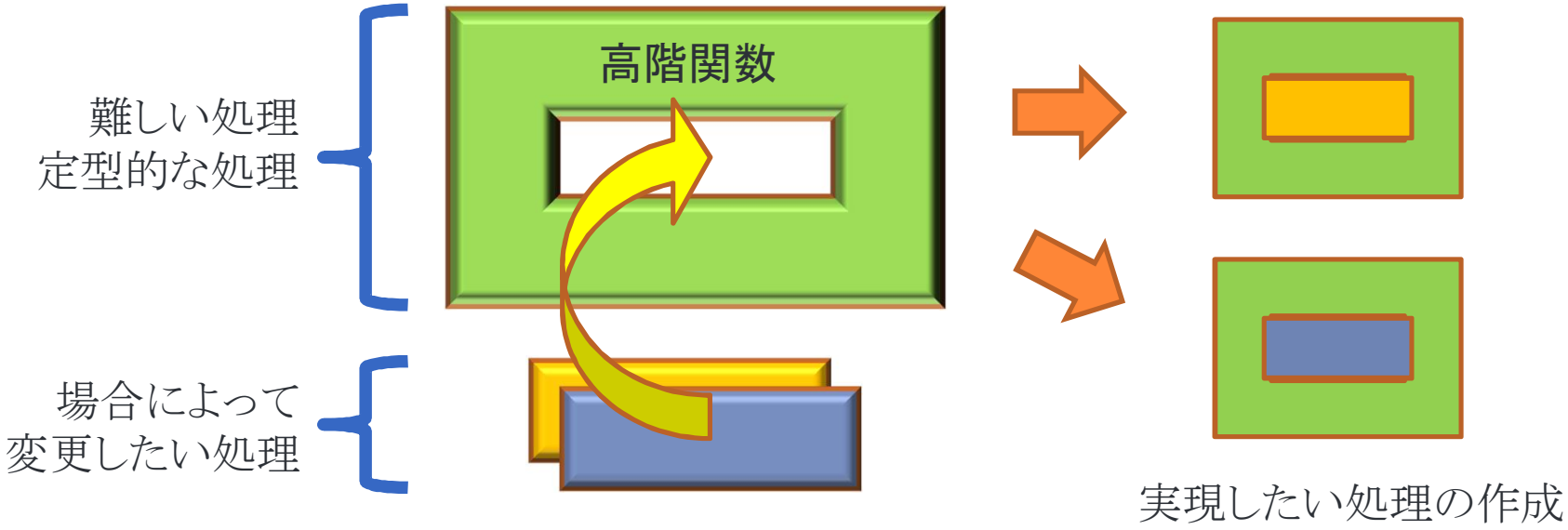
- 高階関数と処理の組み立て
- 代数的データ型の活用

第1章
高階関数と処理の組み立て

高階関数とは

- 関数を受け取る関数、もしくは関数を返す関数のこと。
- 関数プログラミングは、(高階)関数を効果的に活用するプログラミングスタイル。

高階関数の有効性
処理の柔軟な再利用



- 関数を受け取る関数は、難しい処理や定型的な処理と、場合によって変更した処理とを切り分けたもの。
- 引数に値しか渡せない場合に比べて、ロジックの再利用性を高めることができる。

例えばファイル処理

- 「ファイルを開いて、何か処理をして、最後にハンドラを閉じる」という処理は毎回同じ。

【C言語の場合】

```
void do_something(char*  fname) {
  FILE* fd = fopen(fname);
  // ハンドラfdを使う何かの処理
  // ....
  fclose(fd);  // 最後に必ず閉じる
}
```

高階関数による部品化(ローンパターン)

【関数型言語(OCaml)の場合】

```
let  with_file fname f =   (* 高階関数を予め定義しておく *)
 let  fd = open_file  fname  in
   try  f   fd  with _ -> ();
   close fd   (* ハンドラを閉じ忘れることはもうない *)

(* 利用する時はファイル処理内容を記述するだけで済む *)
with_file  fname  (fun fd ->
   (*  fdを使う何かの処理  *)
)
```

無名関数を高階関数の引数として渡している。

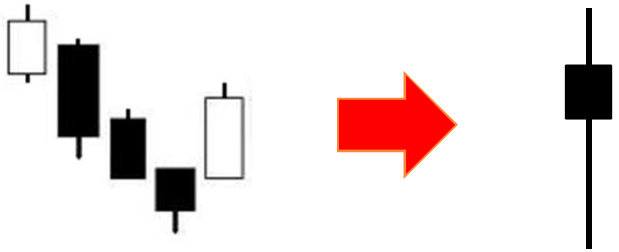
DRY(Don't Repeat Yourself)

例えばローソク足作成処理

- ローソク足とは、一定期間の株や為替の値動きを4本値(始値、高値、安値、終値)で表現したもの。

ローソク足チャートのイメージ図

- 業務用件:1分足のリストから5分足のリストを生成したい。



1分足5本

生成したい5分足

IT Planning, Inc.

手続き的に書くなら・・・

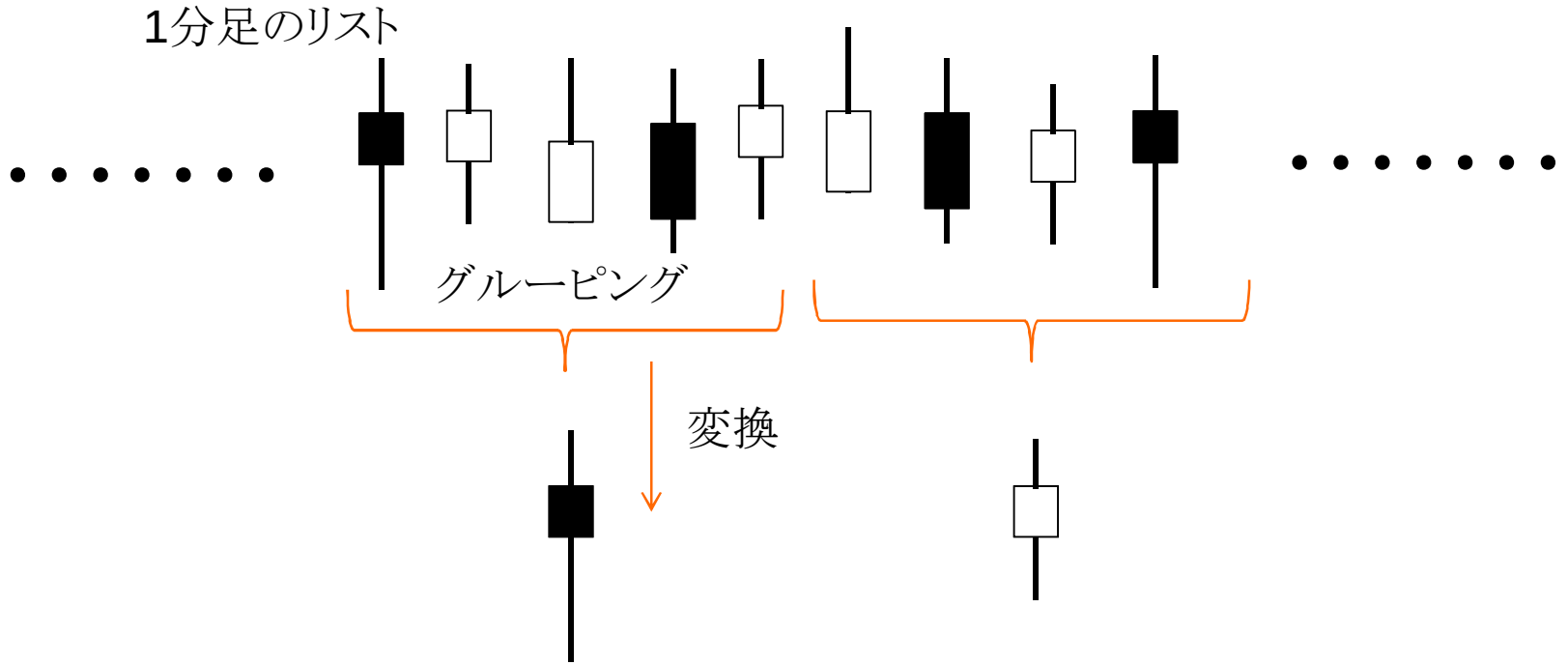
5分足のリスト　　1分足のリスト（ソート済み）

```
public List<Candle> make5m(List<Candle> c1m) {
    Decimal high, low;
    ArrayList<Candle> result = new ArrayList<Candle>();
    Candle prev = new Candle(c1m.get(0));
    for(Candle c : c1m) {
        if(same5m (prev.time, c.time)) { // 同じ5分に属するかどうか？
            prev.merge(c);  // prevの高値、安値、終値を更新
        } else {
            result.add(prev);
            prev = new Candle(c);
        }
    }
    result.add(prev);  // 最後の要素の追加
    return result;
}
```

次の5分に属する1分足がきたら、5分足が完成したとして結果に追加。

やりたい事はなんだったのか？

- 5本毎の1分足にまとめる。(グルーピング)
- 5本の1分足を1本に変換する(変換)



関数プログラミング的な書き方

【関数型言語(OCaml)の場合】

```
(* グループ化して変換する *)
let c5m candles =
  groupBy (fun  c1 c2 -> same5m c1.time c2.time)  candles
  |>  map (reduce merge)
```

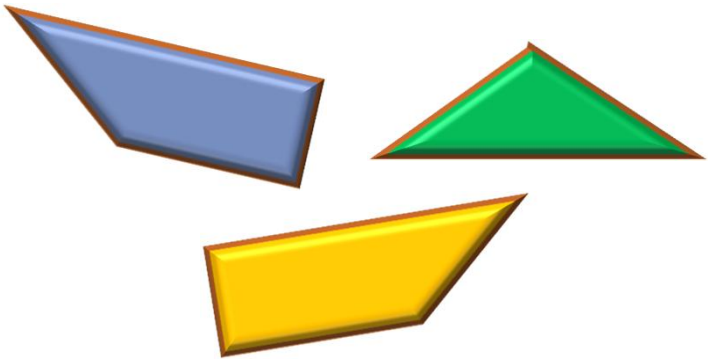
同じ5分でグループ化(無名関数)

5本のローソク足を1本のローソク足に変換

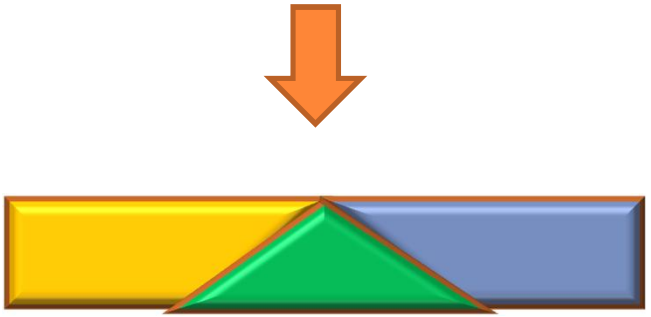
- groupBy, map, reduceは高階関数。
- 汎用的な関数を組み合わせて必要な処理を構築する。

高階関数を組み合わせて処理を構築

- コレクション処理(リスト)
  - map:変換する
  - fold:畳み込む(reduceの一般形)
  - groupBy:グルーピング
  - filter:抽出する
  - find:見つける
  - split:分割する
- 処理の合成
  - $:関数合成
  - retry:リトライする
  - memoize:記憶する



(高階)関数の部品



実現したい処理の構築

高階関数を使うと、汎用的な部品から欲しい部品を組み立てていく事ができる。

1章　高階関数と処理の再利用

- 高階関数とは、関数を受け取る関数、もしくは関数を返す関数のこと。
- 汎用的な高階関数を再利用/組み合わせることで、ロジックを簡潔に表現できるようになる。

第2章
代数的データ型の活用

代数的データ型は共用体の拡張版

【C言語の場合】

```
typedef enum { Joker } joker;
typedef union {
  int number
  joker joker;
} card;
```

共用体は、ある瞬間にどちらのフィールドが有効なのかは教えてくれない。

card型の値を見た時に、どのフィールドが有効なフィールドなのか判別できるようにしたい。

タグ付き共用体、判別共用体、代数的データ型

代数的データ型は共用体の拡張版(2)

【OCamlの場合】

```
type card = (* 一枚のカードの表現. *)
  Number of int
| Joker

(* 使い方 *)
let foo = function (* カード型の引数を受け取る関数 *)
  Number i ->  ...  (*  安全な場合分け  *)
| Joker -> ...
```

NumberかJokerのどちらかという意味。

様々なデータ構造を表現できる

- リスト構造
  - 終端かデータかのどちらか。基礎的なデータ構造。
- オプション構造
  - データが存在するかしないか。危険なnullの安全な代替。
- 構文木
  - (非)終端記号を代数的データ型に対応させる。
  - 構文木を簡潔に扱えるので、関数型言語はプログラミング言語を扱うのが得意と言われている。

例えばこんな例題。

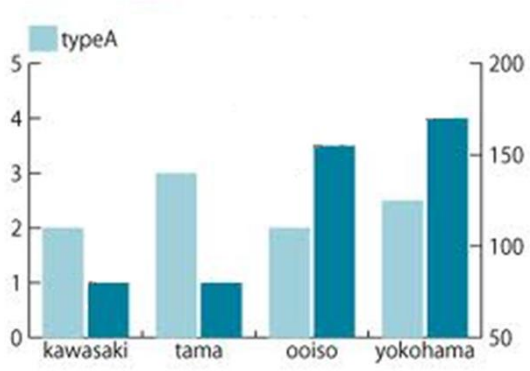
- 棒グラフボタン、折れ線グラフボタン、散布図ボタンがあり、ボタンを押すと排他的に対応するグラフが表示される。
- 重ね描画モードがONの時は、三つのボタンは独立したトグルボタンになる。

重ね描画モード

棒グラフ

折れ線

散布図

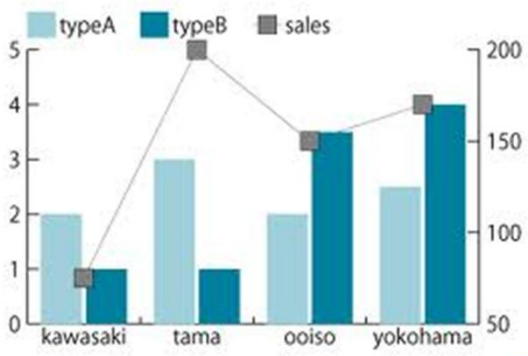


重ね描画モード

棒グラフ

折れ線

散布図



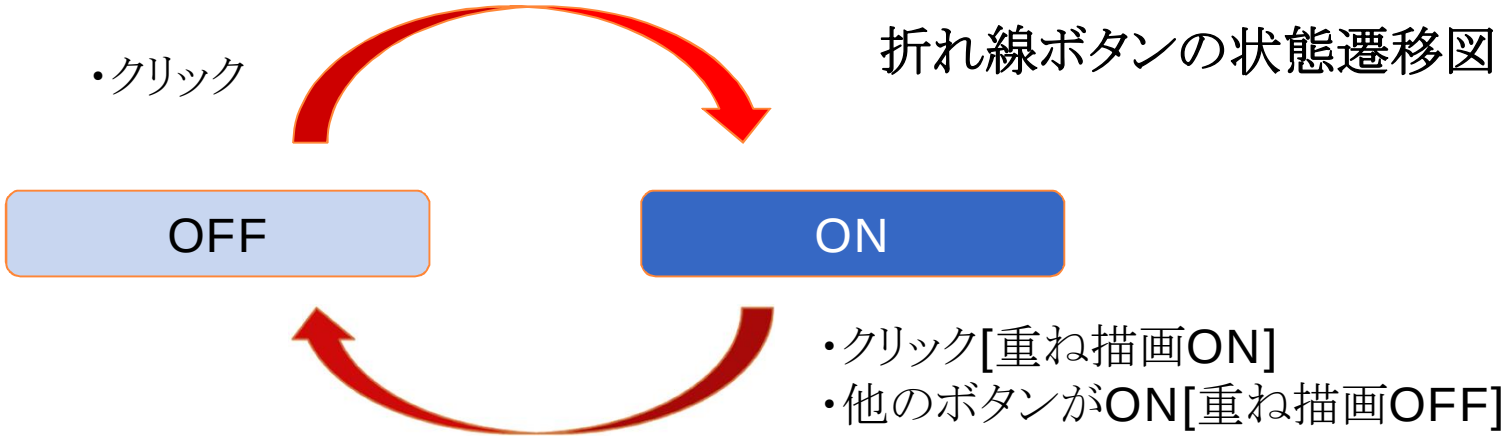
状態を定義してみる。

【関数型言語(OCaml)】

```
(* 全部で4つのON/OFF状態を定義。初期値も付与. *)
let overwrite_mode = ref false
let bar_graph = ref true
let line_graph = ref false
let scatter_diagram = ref false
```

状態遷移図を記述。(折れ線ボタン)

- 重ね描画モードがONの時、
  - 既に折れ線ボタンがONなら、これをOFFに。
  - 折れ線ボタンがOFFなら、これをONに。
- 重ね描画モードがOFFの時、
  - 既に折れ線ボタンがONなら、何もしない。
  - 折れ線ボタンがOFFなら、これをONに。他のONのボタンをOFFに。



折れ線ボタンの状態遷移図

コーディングすると、こんな感じ。

```
let on_bargraph_click () =  (*  折れ線クリック *)
  if  !overwrite_mode then
      bar_graph := not !bar_graph (*  反転 *)
  else
      if  !bar_graph then ()
      else begin
          bar_graph := true;
          line_graph := false;          (*  排他的動作 *)
          scatter_diagram := false;  (*  排他的動作 *)
      end
```

ＯＫ、何か問題が？

問題点

- ボタンは3つ。折れ線ボタン以外のもう2つのボタンもほぼ同じロジックなのに、コピペする？同様に、4つ目のボタンが増えた時、全てのボタンのロジックを変更する？
- overwrite_modeがOFFの時は、ボタンのどれか一つがONである事を覚えておけばいいのに、全てのボタンのON/OFFを管理している。冗長な上にミスの元じゃない？

代数的データ型で状態を見直してみる

```
type button = Bar | Line | Scatter
type state =
    OverWrite of button list   (* 重ね描画ON *)
  | Normal of button           (* 重ね描画OFF *)
```

- 重ね描画モードONの時には、ONになっているボタンのリストを保持する。
- 重ね描画モードOFFの時には、3つのボタンの内どれか一つが必ずON。

管理すべき状態が4つから2つに半減！
（しかも同時には1つの状態のみ）

ボタンを一般化した状態遷移。

```
let on_button_click b = function
  | OverWrite bs when List.exists ((=) b) bs ->
        OverWrite (List.filter ((<>) b) bs)
  | OverWrite bs ->  OverWrite (b :: bs)
  | Normal _ -> Normal b  (* 排他的ON *)
```

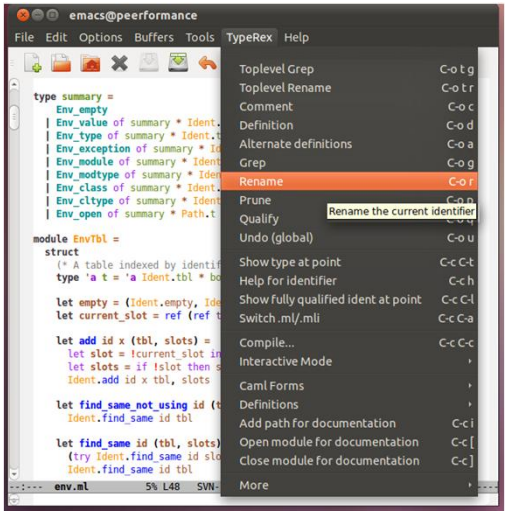
- 一般化したのに、たったの4行。
- パターンマッチによる見通しのいい場合分け。
- 網羅性チェックも働く。

2章　代数的データ型の活用

- 代数的データ型は共用体の拡張版。様々な場合分け構造を表現できる。
- イベントドリブンプログラム（状態遷移系）の表現にも、代数的データ型は活躍できる。

おまけ：OCamlエコシステムの進化

- 定番のライブラリ/フレームワークが揃ってきた
  - Webシステムフレームワーク：Oxygen/Lwt
  - 拡張ライブラリ：Core
- 便利なパッケージマネージャー
  - OPAMで依存関係気にせず一発インストール
- 開発環境
  - Emacs + typrexでリッチな操作
  - 動作確認に便利なREPL



Haskell, Scala, F#も同様に
環境は整っている。

まとめ

- 実務で実際にあった例を二つ紹介。どこにでも転がっていそうな普通の例。高階関数や代数的データ型を使ってスマートに解決できた。
- OCamlのエコシステムは進化してきており、フレームワーク・開発環境など現場に適用できるレベルになってきた。

関数プログラミングは
現場でも役に立つシステム

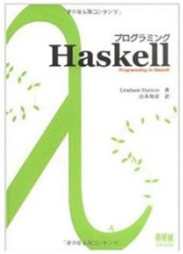
おすすめ書籍

関数プログラミングを学びたいなら
プログラミングの基礎

OCamlを学びたいなら
プログラミングOCaml

Scalaを学びたいなら
Scalaスケーラブルプログラミング第二版

F#を学びたいなら
実践F#



Haskellを学びたいなら
プログラミングHaskell

極意に触れたい人は
関数プログラミングの楽しみ

ネットで読める連載読み物

- 本物のプログラマはHaskellを使う 

  - http://itpro.nikkeibp.co.jp/article/COLUMN/20060915/248215/?ST=develop

- 数理科学的バグ撲滅方法論のすすめ 

  - http://itpro.nikkeibp.co.jp/article/COLUMN/20060915/248230/?ST=develop

- 刺激を求める技術者に捧げるScala講座 

  - http://itpro.nikkeibp.co.jp/article/COLUMN/20080613/308019/