函数プログラミングのエッセンスと考え方

2012年5月31日　　小笠原　啓

IT Planning, Inc.

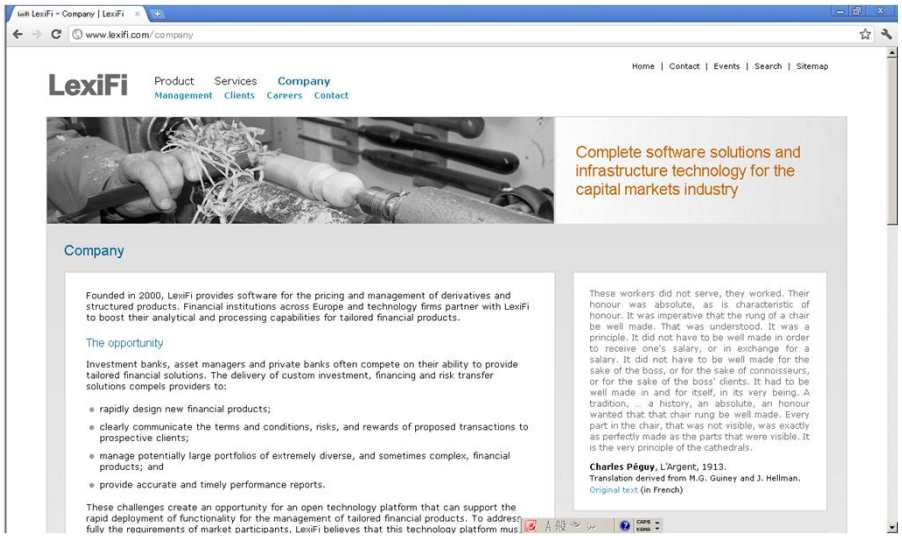
アジェンダ

- 関数プログラミングの今
- 高階関数の考え方
- 代数的データ型の紹介
- 形式手法との繋がり

第1章
関数プログラミングの今

関数プログラミング
近年の成功事例：LexiFi

- 2000年設立のLexiFi社。関数プログラミングを用いて金融派生商品の契約を記述できるDSLを発明。
- 複雑化した金融取引に信頼性と自動化をもたらし、大きな影響を与えた。



LexiFi社Webページ(フランス)

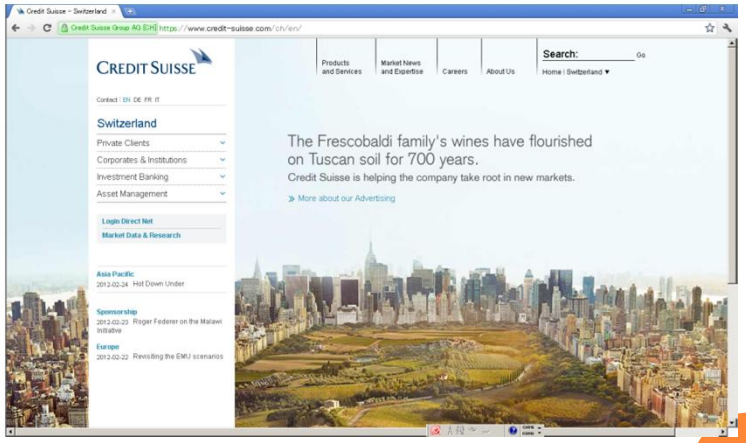
金融業界での応用

JANE STREET

- 2002年、Jane Stree Capitalがトレーディングシステム(High-Frequency)をOCamlで構築。
- 300名を抱える自家運用(private fund)に。

CREDIT SUISSE

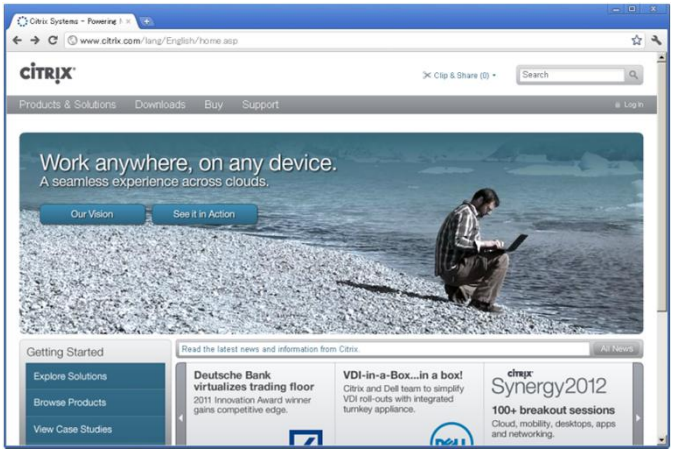
- 2006年、クレディスイスがデリバティブや計量的金融(クォンツ)にHaskell/Microsoft F#を採用。
- Standard Chartered 銀行もHaskellを採用。



関数プログラミングの他セクターへの広がり

CITRIX

- 仮想化サーバーXenで有名なCitrix。
- Xenの管理ツール類はOCamlで記述されている。



twitter

- Ruby on Railsで構築されたインフラをScalaに移行。フォルトトレラントな分散フレームワークも開発。

※http://cufp.org/で多くの事例を発見できます。

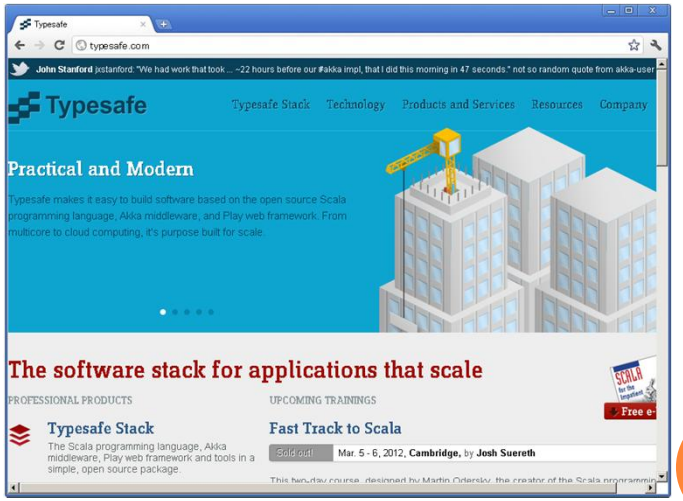
関数プログラミングを支えるビジネスも。

IntelliFactory社(ハンガリー)

- 商用のMicrosoft F#用のWebフレームワーク。
- モバイルアプリにも対応。
- 自らの製品でfpish.netを構築・運用。

TypeSafe社(アメリカ)

- 2011年Scala, Akka開発者らが起業。
- オープンソースのScala, Akka, Playのサポートからコンサルテーション・トレーニングなど。



日本企業での応用も始まっている

- 知的財産・技術情報に関するデータベース事業。Liftフレームワーク(Scala)によるWebアプリケーション。

- レガシーCOBOLプログラムの解析と仕様化をHaskellで自動化。

- yesodフレームワーク(Haskell)によるWebアプリケーション。

- Python,Ruby, Perl, PHP, JVM, .NETに対応したPDF帳票開発ツール製品をOCamlで作成。

弊社での応用事例
OCAMLによるチャートアプリケーション

- ブラウザ上で動作する金融チャートをOCamlを採用。
- 某FX大手にてサービス中。

サーバープログラム

✓ OCamlで記述されたシンプルなAPIと365D24Hの為替データリアルタイム処理デーモン。

iPhone, Android, PCクライアント

✓ OCamlプログラムをocamljsコンパイラでJavaScriptに変換。

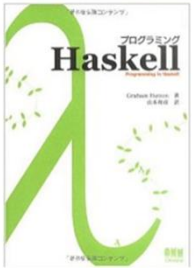
✓ 多彩なデバイスを単一のコードでサポート。

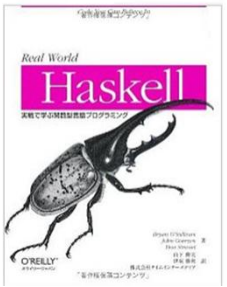
IT Planning, Inc.

ここ数年で多くの書籍も発刊

OCaml

Haskell

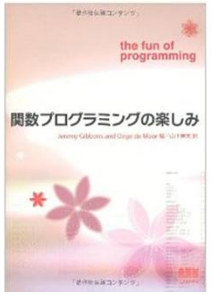




F#

Scala

関数プログラミング



なぜ関数プログラミングなのか？

複雑な処理との適合

一見複雑な手順が必要と思われる
複雑な処理・難しい手順も、

アルゴリズムを簡潔に記述できる
関数プログラミングで書けば、
シンプルな組合せになる。

- 競争力の源となる自然言語処理、大規模データ解析、プログラム解析など複雑なアルゴリズムが必要な場所に優れた力を発揮。

品質リスクの低減の一助になる

- ネットサービス、モバイルデバイスに対する市場の成熟。不具合や問題に対するリスクが大きくなってきた。

- 強力な静的型付き言語が手軽に使えるようになってきた。
  - コンパイラの性能アップ、ライブラリの増加。
  - コンパイラがある種の安全性を自動的に担保してくれる。
  - 型によるチェックを頼りに既存コードの拡張/変更が容易に。
  - テスタビリティ、ベリファイアビリティが向上。

1章　関数プログラミングの今

- 関数プログラミングは、難解な契約が飛び交う金融業界で成功を収めた。
- 現在では様々なセクターに広がりを見せ、様々な分野で応用がなされている。
- 関数プログラミングはアルゴリズムを簡潔に記述できるため、複雑な処理に優れた力を発揮する。
- 強力な静的型付け機能が素早く高い品質を実現する一助となる。

Let's 関数プログラミング！

IT Planning, Inc.

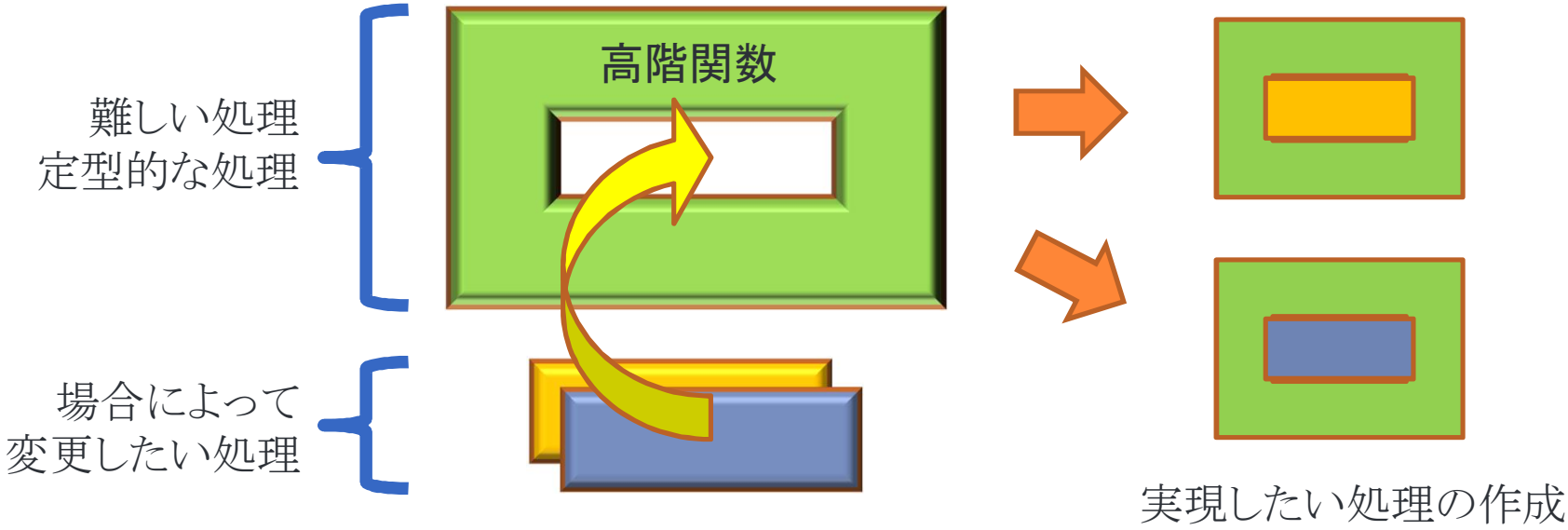
第2章
高階関数の考え方

IT Planning, Inc.

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(1)
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
}
```

高階関数による省力化(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

(* 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。

定型的なループ処理

- 「配列の中身を最初から最後まで順に取得しながら、何か処理する」という処理は毎回同じ。

【C言語の場合】

```
char  array[3] = { 1, 2, 3 };
for (int i = 0; i < 3; i++ ) {
    // 配列arrayの中身を利用する何かの処理
    // ....
}
```

高階関数による省力化

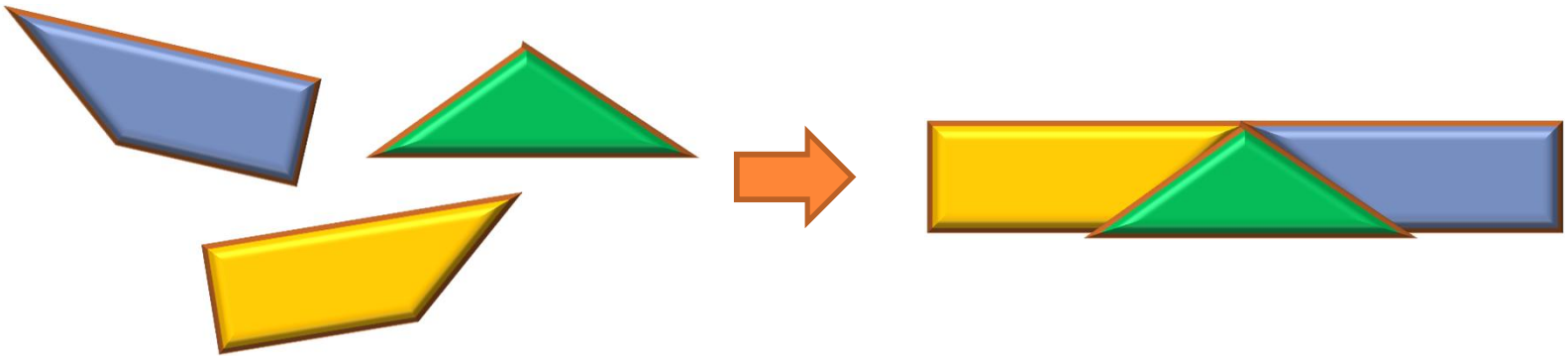
【OCamlの場合】

```
(* ループ処理用の高階関数 *)
let rec iter f = function
    [] -> ()
  | hd :: tl -> f hd; iter tl

(* 利用する時はループ処理内容を記述するだけで済む *)
iter (fun elem ->
    (* リストの要素elemを使う何かの処理 *)
) [ 1; 2; 3 ]
```

他にもmapやfoldなど
ループ用の様々な
高階関数がある。

高階関数の有効性(2)
スケーラビリティの創出



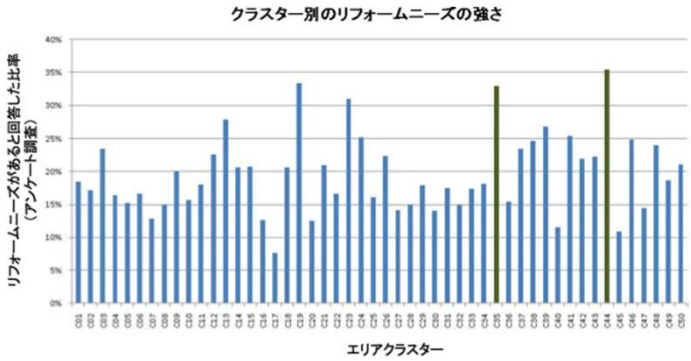
（高階）関数の部品　　実現したい処理の構築

- 高階関数を使うと、小さな処理から大きな処理を組み立てていく事ができる。
- 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。

例えば集計処理

- とある製品の年間の売上リスト(売上日、売上金額)から月別の売上合計リストを作りたい。

	A	B	C	D	E	F	G
1	日数	時刻	品種	ロット№	正味重量	風袋重量	ステータ
2	合計	401日			6,836.21	464.14	
3	1	09:34:09	25200	2512977	17.05	1.15	M
4	2	09:34:19	25200	2512977	17.05	1.15	M
5	3	09:34:59	25200	2512977	17.05	1.15	G
6	4	09:35:31	25200	2512977	17.05	1.14	G
7	5	09:36:01	25200	2512977	17.06	1.14	G
8	6	09:36:32	25200	2512977	17.05	1.15	G
9	7	09:37:04	25200	2512977	17.05	1.15	G
10	8	09:37:34	25200	2512977	17.05	1.15	G
11	9	09:38:08	25200	2512977	17.05	1.15	G
12	10	09:38:38	25200	2512977	17.05	1.15	G
13	11	09:39:10	25200	2512977	17.05	1.15	G
14	12	09:39:40	25200	2512977	17.05	1.15	G
15	13	09:40:10	25200	2512977	17.05	1.16	G
16	14	09:40:41	25200	2512977	17.05	1.15	G
17	15	09:41:13	25200	2512977	17.05	1.16	G
18	16	09:41:45	25200	2512977	17.05	1.15	G
19	17	09:42:16	25200	2512977	17.05	1.15	G
20	18	09:42:48	25200	2512977	17.05	1.16	G
21	19	09:43:19	25200	2512977	17.05	1.15	G
22	20	09:43:54	25200	2512977	17.05	1.15	G
23	21	09:44:27	25200	2512977	17.05	1.15	G
24	22	09:45:02	25200	2512977	17.05	1.16	G
25	23	09:45:35	25200	2512977	17.05	1.16	G
26	24	09:46:07	25200	2512977	17.05	1.16	G
27	25	09:46:40	25200	2512977	17.05	1.16	G
28	26	09:47:11	25200	2512977	17.05	1.16	G



手続き的に書くなら・・・

月別集計売上のリスト　　売上のリスト(ソート済み)

```
public List<Integer> monthlySale(List<Sale> sales) {
    int sum = 0;
    ArrayList<Integer> result = new ArrayList<Integer>();
    Date prevMonth = sales.get(0).date.getMonth();
    for(Sale sale : sales) {
        if(prevMonth == sale.date.getMonth()) {
            sum += sale.getAmount();
        } else {
            result.add(sum);
            sum = sale.getAmount();
        }
        prevMonth = sale.date.getMonth();
    }
    if(sum != 0) { result.add(sum); } // 最後の要素(12月)の追加
    return result;
}
```

月が変わったら、ここまでの売上合計が月間売上となる。

関数プログラミング的な集計処理

【関数型言語(OCaml)の場合】

```
(* 月別にグループ化して畳み込む *)
let monthlySale =
 groupBy (fun  s1 s2 -> s1.date.month = s2.date.month)
 |> map (map (fun sale -> sale.amount))
 |> map (reduce  (+))
```

同じ月でグループ化(無名関数)

関数適用(逆順)

(+)は二引数関数。(reduce (+))も関数

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

さらに進んだ組立て ~コンビネーター~

- 特定の目的のためにデザインされた、互いに組合せ可能な関数群を指して「コンビネーター」と呼ぶ。※

例:

- パーサーコンビネーター
  - ✓ 文字列からデータ構造を構築する処理(パーシング)に特化したコンビネーター。
- リアクティブコンビネーター
  - ✓ 依存関係を第一級化して合成できるコンビネーター。
- 非同期処理コンビネーター
  - ✓ 非同期処理を記述することに特化したコンビネーター。タイムアウトとの組み合わせなどを自由に合成できる。

※正確には閉じたλ式の事。

パーサーコンビネーターの例

【OCamlの場合】

```
type 'a parser    (* パーサー型定義 *)
val return : 'a -> 'a parser (*  定数 *)
val bind : 'a parser -> ('a -> 'b parser) -> 'b parser  (* 入れ子 *)
val map : ('a -> 'b) -> 'a parser -> 'b parser  (*  変換 *)
val (>>) : 'a parser -> 'b parser -> 'b parser  (*  破棄合成  *)
val (<|>) : 'a parser -> 'a parser -> 'a parser  (* 選択 *)
val many :  'a parser -> 'a list parser  (*  繰り返し *)
val sep : 'a parser -> 'b parser -> 'b list parser (* 区切り分割 *)
val run_ch : 'a parser -> in_channel -> 'a   (*  パーサー実行 *)
```

・
・
・

パーサーコンビネーターによる実用的なCSVパーサーの記述

- 数字とダブルクォートで括られた文字列を認識。
- 文字列内にはカンマや改行が許される。
- 文字列内のダブルクォートはダブルクォートを重ねてエスケープ。

```
let quote, pquote = "", char ""
let str_content = nonOf "¥""
let quote_escape = map (fun _ -> quote) (keyword "¥"¥"")
let str_escaped = many (quote_escape <|> str_content)
let str = pquote >> (map string_of_chars str_escaped) <=< pquote
let field = (map (fun x -> `Int x) int) <|> (map (fun s -> `Str s) str)
let line = sep  (char ',') (spaces >>  field <=< spaces)
let csv_parser = sep eol line <=< (opt eol <=< spaces)
```

関数プログラミングを可能とする条件(1)
柔軟な関数の扱い

- 高階関数を上手に使いこなすためには、ある程度言語からの支援が必要。
  1. 関数を変数に入れられる/無名で定義できる機能
  2. レキシカルスコープを持つこと。
  3. 部分適用機能。
  4. 関数に関する構文が"軽い"こと。

	C言語	Java	JavaScript
第一級化	関数ポインタ	オブジェクト	○
スコープ	×	finalのみ	△(this)
部分適用	×	×	×
構文の軽さ	×	×	△
静的型付け	△	○	×

関数プログラミングを可能とする条件(2)
変数を上書きしないスタイル

- 高階関数に渡した関数はいつ呼び出されるか分からない(呼び出されないかもしれない)。

＋

- 高階関数が代入によって変わり得る変数の中身を参照している。

スパゲティコード

- 変数の内容は書き換えない。
- 配列をやめてリスト(上書き不能なデータ列)を使う。

※元々、関数プログラミングの「関数」は数学的な関数を意図しており、変数の上書きのない世界を想定したスタイル。

2章　高階関数の考え方

- 高階関数は、関数プログラミングスタイルにおいて重要な考え方。
- 高階関数とそれをサポートしてくれる言語機能があれば、関数部品の再利用性が高まりスケーラビリティも得られる。
- ただし、関数プログラミングを便利に使うには言語からの支援や変数を上書きしないスタイルが必須。

第3章
代数的データ型の紹介

トランプのデータ構造

- 数字(1から13)を持つトランプのカードを表現するデータ構造を作りたい。
- ジョーカーも含める（ここがポイント）。種類（スペード、ダイヤなど）は簡略化の為省略。
- 一枚のカードをCard型としてカードの配列（手札）を作りたい。Card型のオブジェクトからgetNumberによって数字を得たい。ただし、ジョーカーに数字は無い。

Javaによる実装例:

```
public class Card {  // 一枚のカードの表現
  private int i;
  public Card(int n) {  i = n; }
  public Card() { i = -1; }
  public int getNumber()  {  return i;  }
  public boolean isJoker() { return i == -1; }
}
```

-1をジョーカーの目印とする

- isJokerメソッドを呼び出して事前に数字を持つカードかどうかを確認し忘れると、getNumberメソッドで-1が返ってきて困る。

isJokerを忘れてgetNumberを呼ぶとアウトという

暗黙のお約束を作ってしまっている。

代替案:インターフェイスの継承

```
interface Card {} // 基底インターフェイス
public class NumberCard implements Card {
  private int number;
  public NumberCard(int i) {
    number = i;
  }      // ↓ NumberCard型にだけあるメソッド
  public int getNumber() { return number; }
}
public class Joker implements Card {
  public Joker() { }
}
```

instanceofを忘れてキャストするとアウトという

暗黙のお約束を作ってしまっている。

簡単な問題なのに、

何故こんなに難しいのか？！

そこで代数的データ型

【OCamlの場合】

```
type card = (* 一枚のカードの表現. たった3行 *)
  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。

暗黙のお約束一切なし。しかも簡単。

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。

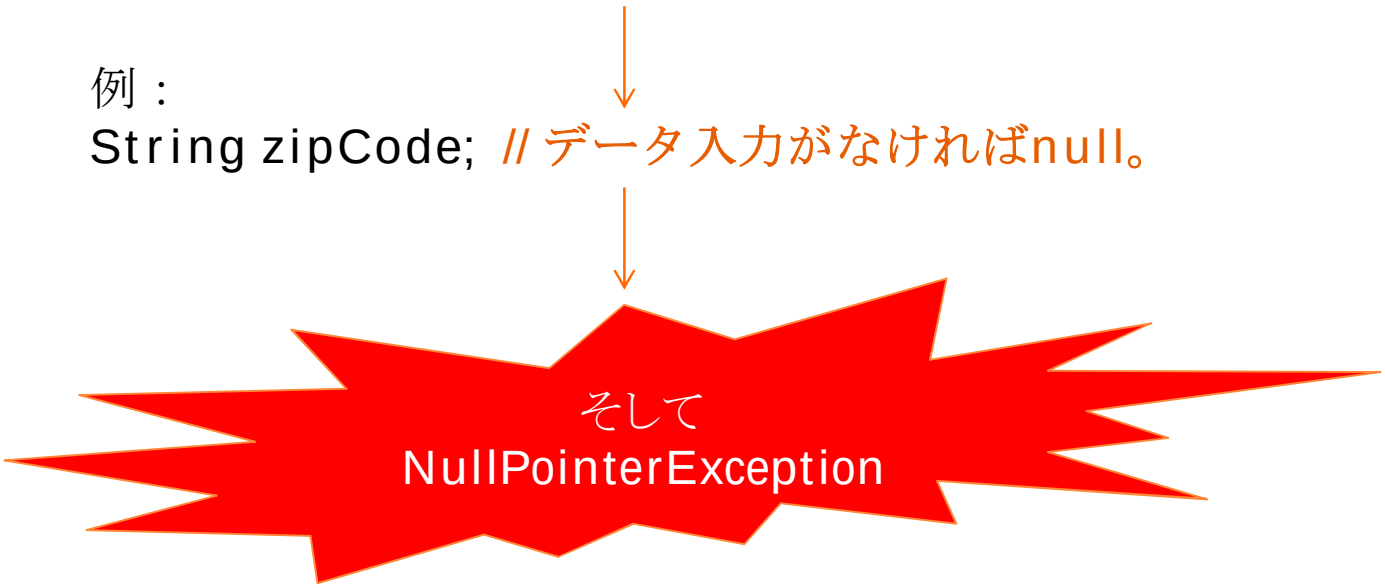
card型の値を見た時に、どのフィールドが有効なフィールドなのか判別できるようにしたい。

タグ付き共用体、判別共用体、代数的データ型

NULLかどうかは頻出の場合分け

例えば、お問い合わせフォームからの入力データ処理。

- 郵便番号データがユーザーから入力されていれば、それをzipCode変数に格納する。
- もし入力が無ければ、zipCode変数はnullとする。



OPTION型による安全性の向上

【OCamlの場合】

```
type 'a option = (*  オプション型の定義  *)
    Some of 'a
  | None
```

Some of 'a ← 値がある場合。'aは型変数。

| None ← 値がない場合。nullに相当。

- 構文のレベルでSomeとNoneのどちらかにしかアクセスできないため、nullアクセスによる例外は起こり得ない。
- 関数型言語に(基本的に)nullはない。代わりにオプション型(もしくはその拡張)を使う。

例：

```
// データ入力がなければNone
val zipCode : string option = None
```

構造体(クラス)による木構造の表現
VISITORパターン

```
abstract class XML {
  abstract void visit(XMLVisitor v);
}
class Element extends XML {
  String name;
  Map<String,String> attrs;
  List<XML> children;
  void visit(XMLVisitor v) {
    v.accept(this);
  }
}
class CDATA extends XML {
  String content;
  void visit(XMLVisitor v) {
    v.accept(this);
  }
}
abstract class XMLVisitor {
  abstract void accept(Element elm);
  abstract void accept(CDATA cdata);
}
```

```
class ToStringVisitor {
  StringBuffer result;
  void accept(Element elm) {
    resut.append("<" + elm.name + printAttrs(elm.attrs) + ">");
    for(Element e : elm.children) {
      e.visit(this);
    }
    resut.append("</" + elm.name + printAttrs(elm.attrs) + ">");
  }
  void accept(CDATA cdata) {
    result.append(escapeCDATA(cdata));
  }
}
```

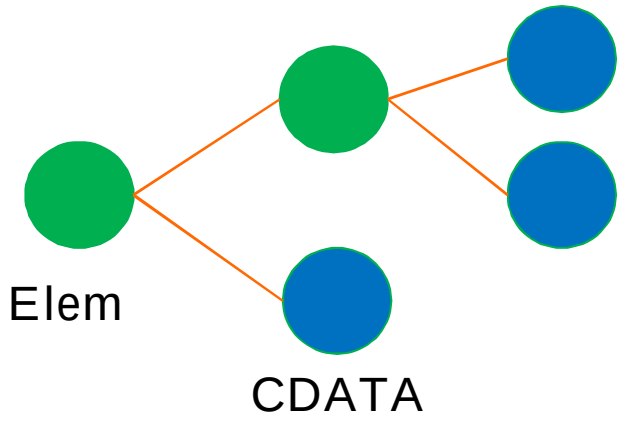
※収まらないくらい長い

代数的データ型を用いた木構造の表現

【OCamlの場合】

```
(*  XMLの定義．たった3行  *)
type xml =
   Elem of  string * string list * xml list
 | CDATA of  string

(*  visitorに対応するコード  *)
let rec xml_to_string = function
 | Element(name, attrs, children) ->
    ”<“+ name + (attr_to_string attrs) + “>”
    + List.map xml_to_string children
    ”</” + name + “>”
 | CDATA text -> escaped_string  text
```



その他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。
- 任意の二つの構造の内どちらか(Either)である構造。
- エラーデータ
  - 色々な種類のエラーを内包するデータと共に表現可能。
- 構文木
  - 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。
- ドラッグアンドドロップの状態
  - 通常の状態かドラッグ中の状態かのどちらか。ドラッグ中のデータを通常時には使わないという暗黙のお約束を表現できる。

3章　代数的データ型の紹介

- 代数的データ型は共用体の拡張版。共用体の中身の種類を判別できる。
- 代数的データ型を用いると、守らなければならないお約束を減らす事が可能となる。
  - nullは不要となりプログラムの安全性が飛躍的に高まる。
- 木構造のような、場合分けが内在する様々なデータ構造を簡潔に表現できる。

ご紹介できなかったその他話題

- 遅延評価
- モナド
- 幽霊型/GADTs
- 型クラス(Haskell)
- モジュール(OCaml)
- implicit parameter(Scala)
- self type(Scala)
- アクティブパターン(F#)
- TypeProvider(F#)
- ・・・etc

詳しくは書籍/Webで

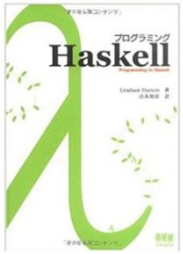
おすすめ書籍

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 

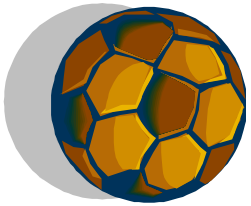
  - http://itpro.nikkeibp.co.jp/article/COLUMN/20060915/248215/?ST=develop

- 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 

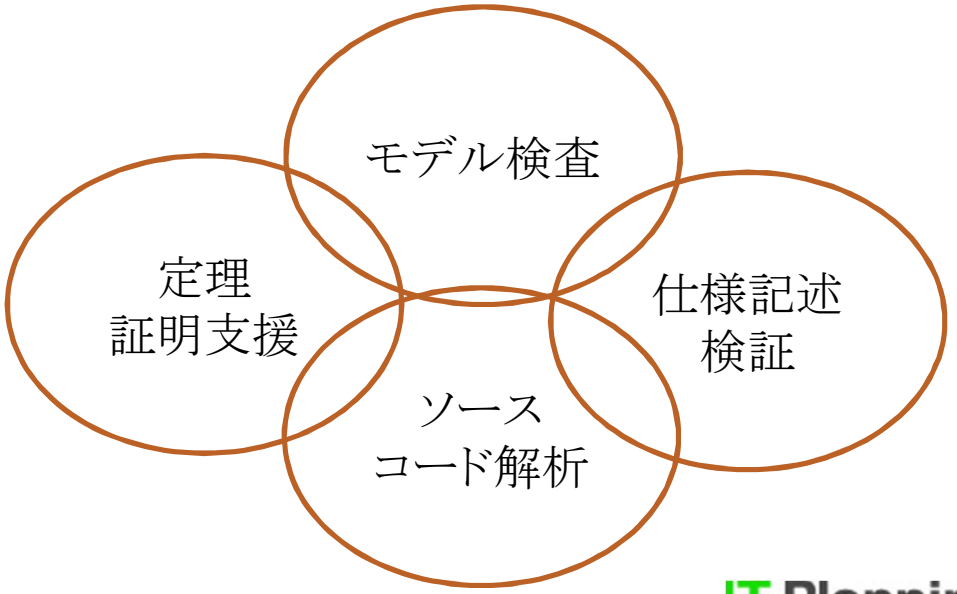
  - http://itpro.nikkeibp.co.jp/article/COLUMN/20080613/308019/



第4章
形式手法との繋がり

形式手法とは、
数学的に厳密な手段を用いて開発を支援する手法

- モデル検査とは、(一般的には)状態機械を記述し、仕様やプログラムや振る舞いを網羅的にチェックする手法。
- 仕様記述・検証とは、仕様を抽象的に記述・解析する事で、仕様段階の不具合を取り除くための手法。
- ソースコード解析とは、プログラムコードを解析することで、潜在的なエラーを見つけ出したりテストケースを生成するなどの利益を得る手法。
- 定理証明支援とは、プログラムの動作性質に対して数学的な証明を与え、テストでは網羅できない無限のケースに対して保証を与える手法。



難しそう！

IT Planning, Inc.

関数プログラミングは形式手法と相性がよい

- 関数プログラミングはなるべく副作用を排除し、変数を書き換えないスタイル。
  - => プログラムの数学的な扱いがとても簡単になる。
  - => 環境に依存しなくなるので、テストが簡単になる。
- 形式的な様々な手法の適用可能性が広がる。
  - モデル検査の状態数を抑えることができる。
  - QuickCheckによるランダムテスト。
  - Coqによるプログラムの証明とコード生成。

プログラムを証明する 定理証明支援器Coq

- フランス国立研究所(INRIA)が開発している対話的定理証明器。
- 記述したプログラムの性質に対し「証明」を与える事ができる。テストでは網羅できない無限のパターンについての検証が可能。
- Coqで関数を記述するとOCaml、Haskellなどのソースコードを生成できる。
  - ⇒（仕様に対して）バグのない関数プログラムを記述できる。

弊社での応用事例
D-BUSメッセージとJSONとの変換

- JSONで記述された要求をD-BusAPIの呼び出しに変換。
- D-Busイベントメッセージ JSONに変換。

- Coqを用いて、JSONで記述された要求で全てのD-Bus操作が可能な事を証明。
- 証明されたコードをエクストラくとしてプロダクトコードとして利用。

インタフェース層：
Adobe Flash

iSDミドルウェア：
ネイティブ (OCaml製)

Linuxネイティブ

iZE Smart Desktop

IT Planning, Inc.

4章　形式手法との繋がり

- プログラムに副作用があると、仕様からの接続やプログラムの解析/性質の記述が急に難しくなる。形式手法の適用が難しい理由の一端は副作用にある。
- 関数プログラミングは副作用を排除するプログラミングスタイル。形式手法との相性がよい。
- 「プログラムの性質を証明する」というテストとは違う新しいスタイルの品質確保手段が有効となる未来が予想される。

講演のまとめ

- 関数プログラミングは世界中で応用されている実績あるプログラミングスタイル。
- 高階関数や代数的データ型を用いることで、複雑なアルゴリズムを簡潔に記述する能力や強力な静的型付けによる安全性を得られる。
- 関数プログラミングはテスタビリティとベリファイアビリティに優れている。形式手法との相性がよく、プログラムの性質を証明する時代がやってくると思われる。

Let's 関数プログラミング！