Class

2011/07/06

森田 昭夫

Class[]とは

- Object Oriented Programmingのための構文糖
  - クラスベースのOOPの一実装
  - Object Orientedは、思想です
    - OOPは、その実践の一形態
    - 他にも OO Analysis, OO Modeling, OO Designなどがある
  - Class[](言語上の機能)が無くても、OOPは出来る
    - Xlibは、Classical Cで OOPしている
  - 詳しくは、ソフトウェア工学の本を参照のこと
- 特殊な名前空間を作る構文要素
  - 専用のMember組み込み演算子(@)でアクセスできる名前空間を作れる
  - 厳密には、Symbolの連想配列になっている模様…

Class[]事始め(1)

■クラスの定義

```
c_Symbol = Class[{super-classes...},
  {class-symbols...}, {instance-symbols...},
  member-declarations...];
```

- シンボル cを基底に名前空間を生成する(クラス c)
- シンボル cにクラスcの生成子を上方値束縛する
- Class[]自身も、上方値束縛
  - 無名クラスは生成出来ない(関数型言語なのに！)
- class/instanceスコープは他のスコープと独立している
  - クラス memberからは、instanceスコープ(インスタンス毎), classスコープ(クラス固有)が優先的に見える
  - class/instanceスコープから明示的に外部シンボルを参照するにはLiteral[_Symbol]を使う
    - Literalを再定義すると何が起こる？
  - 予約メソッド: Constructor(生成子),Destructor(破壊子)
  - 予約シンボル: This(インスタンス自身), Options(インスタンスに渡された Rule, RuleDelayedのリスト)

Class[]事始め(2)

インスタンスの生成

a_Symbol = c[parameters...];

- クラス cのインスタンスを生成し、シンボル aにインスタンスへのアクセス子を束縛する
  - シンボル aは、インスタンスを参照している！
- parametersのうち、Rule, RuleDelayedは class/instance-symbolsに該当するものが有れば、class/instance-symbolsに束縛し、それ以外のRule, RuleDelayedは Options(暗黙のinstance-symbol)に束縛する。それ以外のparametersは、c@Constructor[]へ渡される
  - Constructor[]の値は、(a = c[...])に割り当てられる
- シンボル aへの Unset操作は、インスタンスの破壊操作に上方値束縛されている
  - インスタンスの破壊時に c@Destructor[]が評価される
  - Destructor[]の値は、受け取れない

インスタンスの取扱い注意

- 破壊のタイミング
  - アクセス子が束縛されたシンボルがUnsetされる時

```
c = Class[{},{},{},Constructor[]:=Print["Construct: ", This];Destructor[]:=Print["Destruct: ", This]];
a = c[];
b = a;
b =.;                ← ここで破壊される
```

    - Moduleスコープでインスタンスを束縛していると、スコープ終了時に破壊される！(ファクトリ関数を作る際に注意)

```
factory[] := Module[{tmp},
 tmp = c[];
 With[{return = tmp}, Clear[tmp]; return]];
```

- アクセス制御
  - メンバへの可視域を制御できない
    - 予期しないメンバの再束縛に注意

```
c = Class[{},{A},{B},];
a = c[A->1, B->2];
b = c[A->3, B->4];          ← ここで、a@Aも書き換わっている
```

- Copy Constructorが存在しない

Classの応用(1)

- 普通に Object Oriented Programmingする
  - SADScriptはクラスベースの00
    - 多重継承、多態性、動的束縛をサポート
    - カプセル化は、サポートしない！
- 普通に Duck Typingする
  - 動的束縛だから、普通に使っているよね？
- 末尾処理の予約
  - Destructor[]に末尾処理を記述したインスタンスを局所スコープに束縛しておく
    - スコープ離脱時に必ず実行される
    - Return[], Break[]によるスコープ脱出に便利
- クロージャの代わり？
  - それっぽく書けるけど、以下の問題が残る
    - カプセル化されないので、内部状態が見えてしまう
    - インスタンス自体が無名化出来ない

Classの応用(2)

- 名前空間コンテナ
  - ライブラリの実装等で名前空間を分離するための入れ物
    - 補助関数や定数を大域スコープから隔離する
    - 純関数とWithスコープでも実装できるよ！
  - ライブラリ自体を名前空間に隔離する(Mix-inクラス)
- Mixinクラス
  - 導出クラスに機能を提供する実体化されないクラス
    - 機能の実装を分離する手法
    - 親子関係のないクラスに同一の機能を挿入する手法
- 名前空間によるオーバーロード
  - クラススコープの解決が他のスコープに優先することを用いて、組み込みシンボルを乗っ取る手法
  - 当然、組み込み演算子もオーバーロード出来る
    - 実装ミスがあると無限再帰に陥るので注意
  - ユーザー定義オペレータを実現できる

末尾処理の予約

- StandardForm[]もどき

```
$StandardWildCard = Class[{}, {}, {WildCard},
  Constructor[] := (WildCard = $WildCard; $WildCard = "SAD");
  Destructor[] := ($WildCard = WildCard)];
StandardWildCard[argv_] := Module[{$environment},
  $environment = $StandardWildCard[];
  argv];
SetAttributes[StandardWildCard, {HoldAll}];

$WildCard = "RegExp";
StandardWildCard[
  foo; bar];          ← $WildCard == "SAD"で評価される
zoo;                  ← $WildCard == "RegExp"で評価される
```

  - Moduleスコープ脱出時に Destructor[]で環境を復元

名前空間コンテナ(1)

- Helper functionの隔離

```
$foo = Class[{}, {}, {},
  (* helper functions *)
  bar[argv___] := (...);
  zoo[argv___] := (...);

  (* Public interface *)
  interface[argv___] := zoo[bar[argv]];
  ];

(* Public interface on Global scope *)
foo[argv___] := $foo@interface[argv];
```

  - 実装 $foo@bar[], $foo@zoo[]は、Classスコープ$fooに隔離
  - Class-symbolによる各種定数の保持も出来る

名前空間コンテナ(2)

■ 名前空間の分離

```
Statistics = Class[{}, {}, {},
  Average[l_] := Plus@@l / Max[1, Length[l]];
  Sigma[l_]   := Sqrt[Plus@@((l - Average[l])^2) / Max[1, Length[l] - 1]];
  ];

Print[Statistics@Sigma[Range[10]]];
```

- 共通性を持つ関数群を一つの名前空間にまとめている

■ 別の名前空間への注入

```
$foo = Class[{Statistics}, {}, {},
  bar[l_] := Average[l^2];              ← Statisticsクラスへのメソッド呼び出し
  ];

Print[$foo@bar[Range[10]]];          ← Range[10]の二乗平均
```

- Statisticsクラスメソッドをクラス$foo注入している
  - 実装を Statisticsクラスに移譲している(Mixinクラス的)
  - Average[]メソッドが存在すれば、Statisticsクラスは別の実装でも良い(速度や精度に最適化した実装を選べる)

名前空間によるオーバーロード(1)

- 組み込み演算子・関数のオーバーロード
  - 大域スコープでは、上方値束縛しか出来ない
    - 大域スコープでなければ、可能！

```
$foo = Class[{}, {}, {},
  Times[argv__String] := StringJoin[argv];     ← 文字列同士の乗法演算
  Times[argv___] := Literal[Times][argv];       ← 組み込みTimesへの移譲
  ];

$bar = Class[{$foo}, {}, {}, ← 本当は無名クラスで書きたいところ
  Test[] := (
    Print["Test1: ", "foo" * "bar" * "zoo"];         ← *は $foo@Times
    Print["Test2: ", 20 * 30];
  )];
$bar@Test[];
```

  - スコープ内なら、組み込み演算子のオーバーロードが可能
    - Literal[Times]はオーバーロード前の大域スコープな演算子
    - Literalを下手に再定義するとはまるはず
    - Set系など普段意識無く書く演算子の再定義には注意

名前空間によるオーバーロード(2)

- ユーザー定義演算子(1)
  - 例えば、関数でデータを次々変換する場合…

```
dest = f5[f4[f3[f2[f1[source]]]]];
```

  - 括弧が山盛り！
    - 途中の処理の変更や追加・削除が面倒
    - 例外処理を入れるには、各f*に実装する必要がある
    - データの流れが直感に反する
  - 例えば、次のように書きたい＆実装したい

```
dest = (source | f1 | f2 | f3 | f4 | f5);
```

    - 順位の低い左結合演算子|で、unixのパイプ風
    - 例外が発生したら、後の処理はスルーして例外値を返す
  - 具体的な実装を考えると…
    - ClassスコープでAlternatives演算子を再定義
    - 例外値の扱いは、Haskellの Maybeモナドが使えそう

名前空間によるオーバーロード(3)

- ユーザー定義演算子(2)
  - Maybeモナドを実装してみた

```
$Maybe = Class[{}, {}, {},
  (* Helper functions *)
  Compute[func_, Nothing[msgs___]] := Nothing[msgs___];
  Compute[func_, Just[args___]] := With[{result = func[args]}, Switch[Head[result],
    Just|Nothing,     result,
    _,                Nothing[]]];

  (* Operator overload *)
  Alternatives[Nothing[x___], funcs__] := Nothing[x];
  Alternatives[Just[x___], funcs__] := Module[{v = Just[x]},
    Scan[If[Head[v = Compute[#, v]] === Nothing, Return[v]]&, {funcs}]; v];
  Alternatives[x___] := Literal[Alternatives][x];            (* Delegate *)
  ];
```

  - f*の戻り値は、Just[result](成功)かNothing[message](失敗)
  - Just[source] | f1 | f2 | f3 | f4...は、fnの戻り値が Just[result] なら次のfn+1をfn+1[result]で呼び出す。途中で失敗したら戻り値Nothing[message]を返し評価を終了する

名前空間によるオーバーロード(4)

- ユーザー定義演算子(3)
  - Maybeモナド環境を実装

```
Maybe[body_] := Unevaluated[body]/.{Alternatives->$Maybe@Alternatives};
SetAttribute[Maybe, HoldAll];
```

  - 使ってみる…

```
f1 = Just[Print["f1: ", #]; # + 1]&;
f2 = Just[Print["f2: ", #]; ToString[#]]&;
f3 = Switch[#, _Real, f1[#], _, Nothing[#]]&;
Print[Maybe[Just[0] | f1 | f1 | f3 | f1]];
Print[Maybe[Just[0] | f1 | f2 | f3 | f1]];
```