技術者／プログラマのためのラムダ計算、論理、圏

檜山 正幸 (HIYAMA Masayuki)

2009年1月24日 (土曜) 15:00開始

またの名は

ラムダ・エクササイズ
ヒヤマ'ズブートキャンプ

今日の予定（おおよそ）

1. まえおき／まえせつ -- 10分くらい
2. 心の準備とかオーバービューとか -- 20分くらい
3. 関数に慣れて、サッサッと核心に至る
   -- 40分くらい
4. 大きいラムダと小さいラムダ -- 40分くらい
5. ベータ変換まわり -- 30分くらい

早めに進めばイータ変換の話も。
状況により、予定は変更されるかも知れません。

全体（今回とその続き）の目標

ラムダ計算、自然演繹による推論、デカルト閉圏の三位一体を知る

今回は特に、スノーグローブ現象の例としてのラムダ計算

中島玲二氏の言葉を引用 (1)

『数理情報学入門』(朝倉書店 1982)「はじめに」より:

このような理論的方法を学ぶことは、プログラミングに対する認識を、特定のプログラム言語、慣用の計算機システム、ルーティン化したプログラム作成の手順などによって規定される閉じた世界から解き放ち、次元の異なった抽象的な視点からプログラム言語やプログラムを眺める機会を与えるだろう。これはソフトウェア製品の質的向上のために好ましい結果をもたらすと考える。

賛成！

中島玲二氏の言葉を引用 (2)

以下λ式なる数学的対象を扱うが、λ式がなぜプログラムの一種と考えられるかは、後で議論することにして、まずλ式とはいかなる生き物であるかを知り、自由に操れるように習熟する。λ式に慣れ親しんでいると意外な味が出てきて、自然にその存在理由が明らかになったというのは、多くの人々の経験するところである。

反対。つうか、無理。

ブートキャンプで出来ること

十分な時間を取り、計算のトレーニングをするのが望ましいのは当然ですが、短時間でなにがしかの感覚を得るために、形式的（フォーマル）な計算より、式や計算が指示（denote）する対象をリアルに生々しく捉えるコツをつかみましょう。

ご注意

1. 新興宗教のセミナーではありません
2. あやしいセールスのセミナーではありません
3. スピリチャル関係のセミナーではありません

そうではなくて、技術と科学に関するセミナーです。

このセミナーでは

- オフライン直接対面でなくては絶対に伝わらないこと
- 文章として書くのが絶望的に困難なこと
- 檜山が四苦八苦してやっとマスターした（のか？）こと

を、お伝えします。

そのつもりで資料を作ったら、次からのスライドとか、極度にアヤシイ文面になりました。

今日 伝えたいこと

1. スノーグローブ現象
2. 記号の解釈と使用法の多様性
3. 上記1, 2を理解するための気持ちと体の使い方

スノーグローブ

スノーグローブ：モデル化の不思議

- ホントのいまの自分
- 世界の外に出た自分
- モデルの小世界（マイクロコスモス）に入った自分

スノーグローブ：
仮想と現実が入り混じった世界

- お勉強でCPUエミュレータを作る
- JavaでJVMを実装する
- もし、Rhino上でCPUエミュレータを作ったら

スノーグローブ現象と無縁ではいられない私達

記号：例えば「+」

- 宿題のなかの 2 + 3
- 電卓を叩く2 + 3
- two plus three
- "two" + "three"
- 電池のデッパリ

記号：例えば「=」

- 2 + 3 =
- 2 + 3 = 5
- 5 = 2 + 3
- x = x + 1
- =

理解：能力とテクニック

1. リアルな幻覚を見る能力
2. 感情移入、没入する能力
3. 幽体離脱する能力

練習：

1. 左手に持ったカードを見る
2. 電子レンジ内のグラタンのマカロニになる
3. 月まで行ってみる

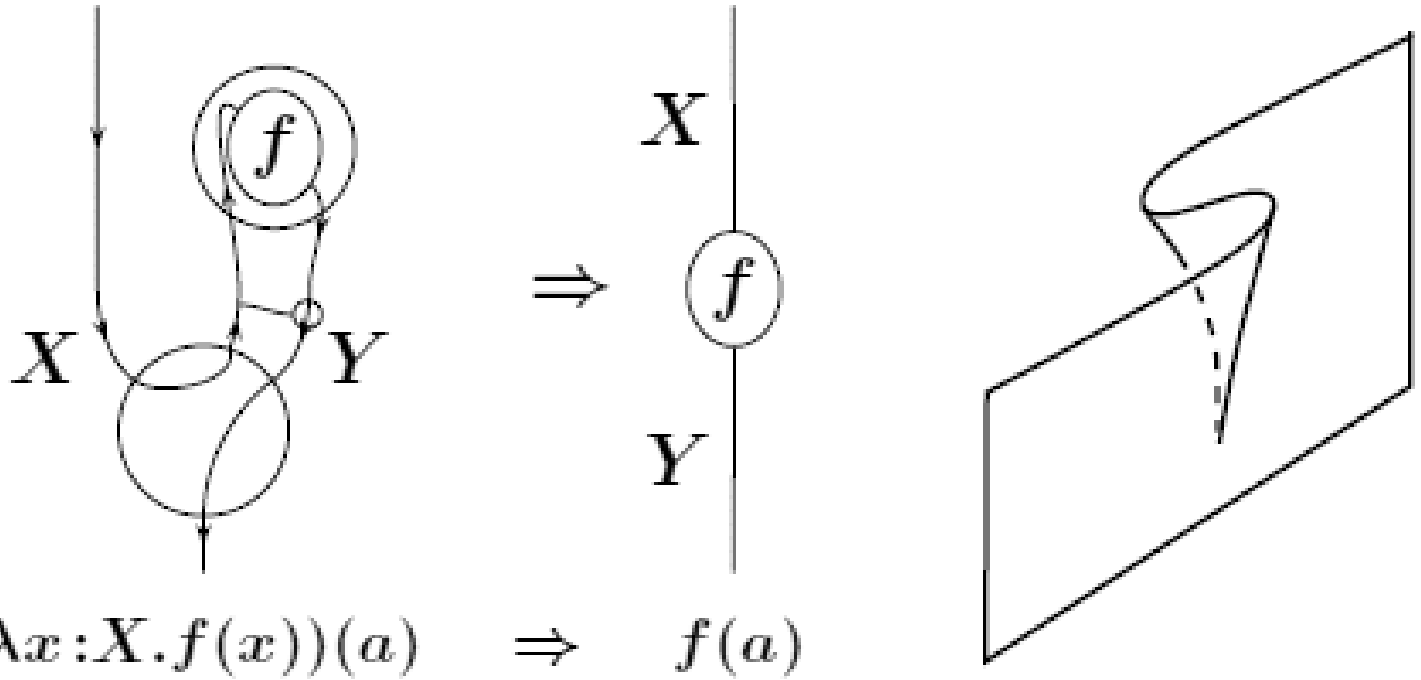
今日の目的／目標のまとめ

1. スノーグローブ現象の実例を感得しよう
2. 記号の多様性を納得しよう
3. そのために、幻覚視、感情移入、幽体離脱を使いこなそう

どこがラムダやねん!?

-- 大丈夫、大丈夫！

バエズの絵で一服



ラムダ計算とは

関数の計算法です。
となると:

- 関数とは何？ →今日述べる
- 計算とは何？ →今日は述べない

計算とはとりあえず、データの変形・加工・合成など、数の加減乗除、文字列処理などを思い起こせばよい。

関数に対する(関数を含んだ)計算と、関数が行う計算の両方が興味の対象となる。

関数とは

- 関数は昔「函数」と書いた
- 関数を、箱に入った計算マシンと考えよう
- 基本的にブラックボックス
- が、分解したり中に潜り込んだりすることもある

関数を絵に描こう

さあ、イマジネーションと幻覚視能力を使おう。

ここでいきなり核心に入る

- 5種の箱（計算マシン） a, b, c, d, e
- c, d, e には補助(?)マシンc', d', e'
- aは足し算をする箱
- データは紙カード

ここでいきなり核心に入る

計算マシン	出力時擬音効果	出力するもの
a	プイッ	値そのもの（データ）
b	ガシャン	関数そのもの（マシン）
c	シュポ	謎のコード（データ）
d	シュポポ	謎のコード（データ）
e	ペロン	人間ほぼ可読な式（データ）

5つの箱と3つの立場

次は等しい。

1. a(2, 3)
2. (b(2))(3)
3. c'(c(2), 3)
4. d'(d(2), 3)
5. e'(e(2), 3)

5つの箱と3つの立場

- b（ガシャン） -- 数学の立場：本物の高階関数
- c, d（シュポ、シュポポ）
  -- 計算科学／工学の立場：ノイマン／ゲーデルのコード化
- e（ペロン）
  -- 論理（記号計算）の立場：人間が操作可能な記号的言語（だが、人間は操作しないことが多い）

実際は、無意識に複数の立場を行ったり来たりして、便利に使い分けたり、あるいは混乱・困惑・挫折している。

計算科学／工学と
論理の立場は似ている

人間可読でも謎の模様でも、記号／符号（コード）であることに変わりはない。

- a(x, y) = c'(c(x), y) （シュポ）
- a(x, y) = d'(d(x), y) （シュポポ）
- a(x, y) = e'(e(x), y) （ペロン）

数学の立場は模倣できる

今回は伝統的数学の立場（ガシャン）は無視するが：

- (c2(x))(y) = c'(c(x), y)
- (d2(x))(y) = d'(d(x), y)
- (e2(x))(y) = e'(e(x), y)

とできる。これは、二番目の箱内にコードを固定的に貼り付けたものをガシャンと出力すること。二番目の箱のコピーがたくさんあればよい。

数学の立場は模倣できる

計算マシン	出力時擬音効果	出力するもの
c2	ガチャッ	関数（マシン）
d2	ガチャッチャッ	関数（マシン）
e2	ゴロン	関数（マシンとみなした…）

「2引数←→1引数」を整理しよう

- a(x, y) = c'(c(x), y)（シュポ）

絵に描いてみる。

- 実は、c'がcに付属しているのではない
- cがc'に対応している、コンパチブルである
- c'は、c以外の関数コードジェネレータともペアを組むことができる
- だから、c'という書き方じゃないほうがいいね

関数コードの実行エンジン

- Eと書こう
- ExecとかEngineとか
- 関数コード（紙カード）は関数（箱、マシン）ではない！
- 関数コードは計算手順を記述したデータ、それ自身は計算を行わない
- 仮想であれ現実であれ、Eはマシン
- Eは、コード構文＝機械語を持つ
- エンジンは無数にある
- エンジンは、自分自身のコードしか理解できない
- エンジンの能力はいろいろだが、万能であるものを選ぶと便利

f^ という書き方を憶えてね

Eが、とある関数コードの実行エンジンとして、
fを、Eのコード体系＝Eの機械語によってコード化した関数コードをf^E と書く。
ただし、fの引数の1つはf^Eのパラメータとして残るので、実引数を具体化した結果 f^E(x) が関数コード。

- f(x, y) = E(f^E(x), y)

これ重要！

通常は、Eを固定して単に f^ と書く。

オマケ:Eの、いろいろな定式化

fが2引数関数で、f^ は、fの関数コード、ただしパラメータは残ってないとする。

1. E(f^, a, b) = f(a, b) -- Eは3引数
2. E(f^, [a, b]) = f(a, b) -- Eは2引数、第2引数はタプル
3. E([f^, a, b]) = f(a, b) -- Eはタプル1引数、タプルは3項
4. E([f^, [a, b]]) = f(a, b) -- Eはタプル1引数、タプルは2項だが入れ子

オマケ：Eの、いろいろな定式化

- 1番目のEは、可変引数にするか、E0, E1, E2, E3, … など、シリーズを準備する。
- 2番目のEは、applyと書かれることが多い。
- 3, 4番目のEは、evalと書かれることが多い。タプルの代わりに、同じ意味の関数コード（式のコード）を使うとよりevalらしい。

いよいよラムダだ

- λ -- 小文字のラムダ
- Λ -- 大文字のラムダ
- ^ -- ラムダに似ている
- < > -- 寝ころんだラムダ2つ＝ラムダ括弧

ちなみに綴りは lambda

大きなラムダ式とその計算

インフォーマルなラムダ計算

- これは、我々が日常的に使う道具
- 紙やホワイトボードに書くもの
- 読み書きと計算は人間がする
- 経験と直感（直観？）が基盤
- 生きている私（あなた）が、世界を記述し、推論を行うために使う
- 大きなラムダ式は、関数を直接表現する、関数そのものの代理
- しばしば、大きなラムダ式と関数そのものが同一視される（区別しなくなる）

大きなラムダ式 (1)

例で示す。

1. <x, y| x + y>
2. <x| 3×x + 2>
3. <a| 3×a + 2>
4. <a, x, y| a×x + 2×y + 1>
5. <| 5>

大きなラムダ式 (1)

- <引数変数並び | 変数を含むかも知れない式> の形
- 本体（ボディ）の式に、引数並びに出てこない変数があるのはNG
- 原則的に、入れ子は許さない（伝統的数学の立場なら許すが）
- Roland Backhouse（http://www.cs.nott.ac.uk/~rcb/）さんあたりが使ってます

質問：<| 5> は 5 と同じか？

大きなラムダ式 (2)

- fが既に存在する（fと名付けられた）関数のとき、f = <x| f(x)> と書いてよい。
  例： sqrt = <x| sqrt(x)>
- fが2引数なら f = <x, y| f(x, y)>、3引数以上も同様。
- f = <x, y| x × x + y × y> のようにして、式で定義される関数に名前を与えてもよい。

等号の意味

- <x| x × (x + 1)> = <x| x × x + x> は成立する
- そもそも関数fとgが等しいとは、許されるどんな具体的な値aに対しても f(a) = g(a)
- 箱（マシン）中身が見えていても、それは考慮しない
- fとgが等しければ、fのグラフとgのグラフは等しいし、逆も真

大きなラムダ式の計算規則

- アルファ規則 <x| f(x)> = <y| f(y)>（ボディが式であってもよい）
- ベータ規則 <x| f(x)>(a) = f(a)
- イータ規則 <x| f(x)> = f

いずれもインフォーマルラムダ計算の規則。経験と直感で納得。自明と言える。

小さなラムダ式とその計算

フォーマルなラムダ計算

- 関数のコード化に使う言語
- 関数コード実行エンジンのマシン語（プログラミング言語）
- 人間への指令ではなくて、マシンへの指令
- データとして扱う
- 数学的に厳密な定義に基づく
- 計算するのは（仮想的でも抽象的でも）実行エンジン
- 人間が計算するときもあるが、それは感情移入

小さなラムダ式の構文

- 定数リテラル：1, 2, trueとか適当に
- 関数記号（関数ではない、インストラクション）：A, Mとか適当に
- 変数（ほんとは型付きだがあまり型に注意してない）
- 適用記号・
- ラムダ記号λ
- ピリオド、カンマ、括弧（用法が2つ）

例：λ(x, y).(A・(M・(2, x), y))、あとホワイトボードに

略式の小さなラムダ式

- <x, y|2 × x + y + 3> 大きいラムダ式
- λ(x, y).(A‧(A‧(M‧(2, x), y), 3)) 小さいラムダ式
- λ(x, y).(2 × x + y + 3) 略式の小さいラムダ式

略式では、小さなラムダ式の内部が人間可読（つうか読みやすく）に書いてある

ラムダ抽象＝ラムダオペレータ

- 例：Λy<x, y| 2 × x + y> = <x| λy.(2 × x + y)>
- 大きなラムダのボディ部の変化
  2 × x + y → λy.(2 × x + y)
  をラムダ抽象と呼ぶことが多い
- だが、ラムダ抽象は大きなラムダ式に働く操作である！ 関数から、関数コードジェネレータを生み出す
- 伝統的数学の立場では、関数から、
  関数ジェネレータ＝高階関数 <x| <y| 2 × x + y>>
  を生み出す

ラムダ抽象の絵

描こう。

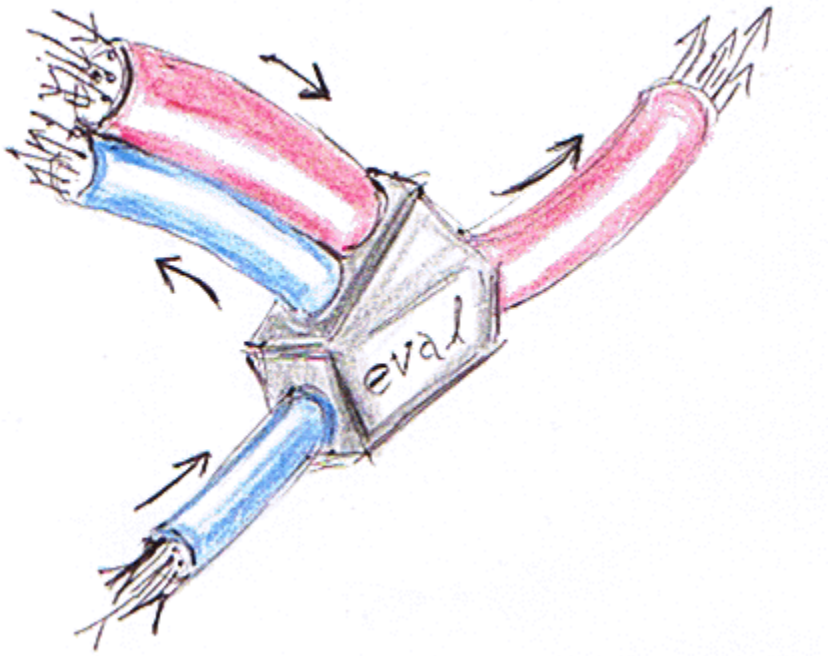
基本等式

- Λ(f) = f^
- Exec(f^(a), b) = f(a, b)

いろいろなバリエーションがある

ラムダ抽象の絵もっと

こりゃ向きが違うが、



スノーグローブとベータ変換

- 我々が大きなラムダ式の計算に行うベータ規則による計算
- 基本等式を使ったベータ変換（これも人間の視点）
- 関数コードの実行エンジンが行うベータ変換
- エンジンの変種、Exec, Apply, Evalでもベータ変換が微妙に違う