2　ラムダ記法と (型のない) ラムダ計算

2.1　ラムダ記法

ラムダ記法 (lambda notation) は，関数の表記において，仮引数となる変数を明示した記法である．これにより，関数と，その関数に引数を与えた計算結果 (値) の区別がつくようになる．

例 1　数学の本での記法: $f(x) = ax^2 + bx + c$　これでは $f(x)$ が関数なのか f に x を食わせた結果の値なのかわからない．もし，関数とデータ (自然数や実数など) が，いつでも文脈から区別できるのであれば，このような曖昧さがあっても構わないが，コンピュータ科学では，関数やプログラム自身を扱う関数 (高階関数，メタプログラム) が平気で現れる．そのような場合には，文脈からだけでは，関数なのかデータなのかわからない．

ラムダ記法での関数 f: $f = \lambda x.ax^2 + bx + c$

ラムダ記法での値 $f(x)$: $f(x) = (\lambda x.ax^2 + bx + c)x = ax^2 + bx + c$

f は関数であり，$f(x)$ は値 (関数 f に x を引数として食わせた結果として得られる値) であり，両者は明確に区別される．

ラムダ記法を使うと高階の関数 (higher order function) も記述することができる．高階の関数とは，引数や返り値が関数であるような (高いレベルの) 関数である．

例 2　高階関数:

$double = \lambda f.(\lambda x.f(f(x)))$ … 関数 f とデータ x を引数としてもらい，f を x に 2 回適用した値を返す高階関数

$compose = \lambda f.(\lambda g.\lambda x.g(f(x)))$ 関数 f と関数 g を引数としてもらい，f と g を合成した関数を返す．

たとえば，$f = \lambda x.x * 2$, $g = \lambda x.x + 1$ とするとき，$double(f)$ は，$\lambda x.x * 4$ を表し，$double(g)$ は $\lambda x.x + 2$ を表し，$compose(f, g)$ は $\lambda x.x * 2 + 1$ を表す．

なお，ラムダ記法の伝統に従い，$f(x)$ という表記の括弧を省略して fx と書くことにする．もちろん，括弧が必要になればいくらでも補うこととする．

2.2　構文

(型のない) ラムダ計算の体系を定義する．本章は，型付きラムダ計算への導入であるため，厳密な形式的定義は省略して，自然言語で非形式的に述べるに留める．

まず，変数 $x, y, z, \cdots$ が無限個あるとする．また，定数 $c, d, \cdots$ が有限個もしくは無限個あるとする．ここで，何が定数になるかはプログラム言語に依存して決まるので，ここでは特に定めない．そのとき，ラムダ計算の項 (term, ラムダ項，ラムダ式ともいう) は以下で定義される．

定義 1　[型なしラムダ計算の項]

- x が変数であれば，x は項である．
- c が定数であれば，c は項である．
- M と N が項であれば，(MN) は項である．(このような項を application (適用) という．)

- x が変数であり，M が項であれば，$(\lambda x.M)$ は項である．(このような項を abstraction (抽象) という．)

たとえば，$(\lambda x.(cx))$ や $((xy)(\lambda z.c))$ は項である．

このように記述すると、括弧が大量に必要となり読みにくいので、一定のルールのもとで、括弧を省略する。

まず、一番外側の括弧はいつでも省略できる。つまり、xy は (xy) のことである。$\lambda x.M$ においては，M としてなるべく大きな項 (広い範囲) を取るようにする．つまり，$\lambda x.cx$ は $\lambda x.(cx)$ のことであって $(\lambda x.c)x$ ではない．後者の項を表現するためには，括弧は省略できない．

また，LMN のように，3 個以降の項が適用の形で並んでいるときは左から括弧をつけることにする．つまり，この項は $(LM)N$ をあらわすのであって，$L(MN)$ ではない．

例 3　括弧が省略された式:

$\lambda x.yz\lambda u.vxy$ は括弧を補うと，$\lambda x.((yz)(\lambda u.((vx)y)))$ のことである．

2.3　変数の束縛と α 同値

ラムダ計算や論理の体系を最初に習うときに，最も理解しにくいのが変数の束縛の概念である．

$\lambda x.(\lambda x.x)$ という関数 f において，一番右の x は，どちらの λ に対応するか考える．ラムダ計算の約束事では変数に対応する λ は，その変数を囲む最も内側の λ とする．したがって，f は $\lambda y.(\lambda x.x)$ と同じ関数であって，$\lambda x.(\lambda y.x)$ とは異なる関数である．この「同じ関数」、「異なる関数」という概念をきちんと考えることにする．

一般に項 M は、変数 x を 2 回以上含むことがある。それらを区別するため「項 M における x の 2 回目の出現」等という。ただし、λ の直後の変数については、「出現」と考えない。たとえば、$\lambda x.\ x\ (\lambda y.\ x)$ には、x という変数は 3 回出てくるが、最初のものは λ の直後にあるのでカウントせず、この項に x は 2 回出現するという。

「出現」という言葉を出したのは、項における位置関係により、変数の扱いが異なるからである。たとえば、たとえば、$\lambda x.\ x\ (\lambda x.\ x)\ x$ は x の出現を 3 つ含むが、1 つ目と 3 つ目の x の出現は、最初の λ と対応付き、2 番目の x の出現は、2 つ目の λ と対応付くことが (直感的には) 理解できるであろう。このような λ と変数の出現の対応関係を「束縛」と言う。「束縛」をもう少し精密に定義しよう。

項 M における変数 x の出現は、その x を囲む $\lambda x.$ のうち、最も近くにある $\lambda x.$ によって束縛される、と言う。どの λ でも束縛されない出現を，自由な出現という．例えば，$\lambda x.\ (\lambda y.\ x\ y)\ y$ は、x の出現を 1 つ、y の出現を 2 つ持つが、x の 1 番目の出現 は λx で束縛され，y の 1 番目の出現 は λy で束縛され，y の 2 番目の出現 は自由である．

項 $\lambda x.M$ に対して、M の中の x の自由な出現 (M で自由な出現は、$\lambda x.M$ では一番外の $\lambda x.$ で束縛される) をすべて一斉に別の変数 y で置き換えたものを M' とする。ただし、y は M において自由な出現を 1 つも持たない変数 (M に現れないか、現れたとしても束縛された出現しか持たない) とする。このとき、$\lambda x.M$ と $\lambda y.M'$ は実質的に同じ関数を表しており、特に α 同値であるという。

項 M に対して (M の一部に対して) 上記の置き換えを 0 回以上繰返して項 N が得られるとき、M と N も、α 同値であるといい、$M \equiv_\alpha N$ と書く．例えば，$\lambda x.\ (\lambda y.\ x\ y)\ y$ と $\lambda z.\ (\lambda y.\ z\ y)\ y$ と $\lambda z.\ (\lambda x.\ z\ x)\ y$ とは、いずれも α 同値である。ここで、最初の項で使われている x を最後の項では、別の意味で使っている

ことに注意されたい。このようなものも α 同値である。一方で、上記の 3 つの項と、$\lambda y.\ (\lambda z.\ y\ z)\ y$ とは α 同値ではない。これ以降、α 同値である 2 つの項は同一視する[*2]．なお，このテキストではときどき $M \equiv N$ と書くことがあるが，これは M と N が (括弧を省略せずに書いたときに) 文字列として完全に一致するときのことである．当然ながら，$M \equiv N$ であれば $M \equiv_\alpha N$ であるが，逆は必ずしも言えない．

なお，α 同値性は，変数の名前換えだけで形式的に一致する項の間の関係であり，$\lambda x.x * 2$ と $\lambda x.x + x$ のように意味的に一致するだけの場合は，α 同値とは呼ばない．また，$(\lambda x.x)1$ と 1 は次節で見るように，計算によって一致するが、その場合も α 同値ではない．後者については $=$ という記号を使うことにする．すなわち，$(\lambda x.x)1 \equiv_\alpha 1$ でないが，$(\lambda x.x)1 = 1$ である．

2.4 計算規則

次に計算規則を与える．ラムダ計算は関数の概念だけを持つので，計算といっても，「関数に引数を与えると，その結果の値になる」という形のものだけである．

定義 2 [β-簡約]

> 項 L の中に $(\lambda x.M)N$ の形があるとき，それを $M\{x := N\}$ で置き換える操作を β-簡約と呼ぶ．β-簡約のことを $\rightarrow_\beta$ もしくは $\rightarrow$ と表記する．

ここで，$M\{x := N\}$ は、すぐ後で定義する。

項の中の $(\lambda x.M)N$ の形の部分項を，計算可能な部分項 (reducible expression)，もしくは，レデックス (基, redex) という．

β-簡約の定義に出てくる代入 (substitution) は以下のように定義される．

定義 3 [代入] 項 M の中の自由な x の出現に 項 N を代入して得られる項 $M\{x := N\}$ は以下のように定義される項である．

- $x\{x := N\} \equiv N$
- $y\{x := N\} \equiv y$, ただし x と y が異なる変数のとき
- $(LM)\{x := N\} \equiv (L\{x := N\})(M\{x := N\})$
- $(\lambda y.M)\{x := N\} \equiv \lambda z.((M\{y := z\})\{x := N\})$, ただし,$z$ は M, N に自由な出現を持たない変数

代入の定義の最後のケースが非常にややこしいが，これの詳細と改善策については型付きラムダ計算の章で述べることにして，ここでは，例を与えるのみとする．

$$
\begin{aligned}
(\lambda x.\lambda x.x)1 &\rightarrow (\lambda x.x)\{x := 1\} \\
&\equiv \lambda z.(x\{x := z\})\{x := 1\} \\
&\equiv \lambda z.z\{x := 1\} \\
&\equiv \lambda z.z \\
&\equiv_\alpha \lambda x.x
\end{aligned}
$$

[*2] 同一視するとはいっても，見た目に違う項であるのでそれらを同じと見るためには多少の訓練が必要である

$$
\begin{aligned}
(\lambda x.\lambda y.xy)y &\to (\lambda y.xy)\{x := y\} \\
&\equiv \lambda z.(xy\{y := z\})\{x := y\} \\
&\equiv \lambda z.(xz)\{x := y\} \\
&\equiv \lambda z.yz
\end{aligned}
$$

2.5 合流性と計算戦略

ラムダ計算の計算規則は，非決定的 (non-deterministic) である．すなわち，項 M を計算するやり方が 2 通り以上あることがある．たとえば，$(\lambda x.x+1)((\lambda y.y+2)3)$ という項は，以下の 2 つのレデックスを持つ．

$$
\begin{aligned}
\underline{(\lambda x.x+1)((\lambda y.y+2)3)} &\to ((\lambda y.y+2)1)+1 \\
(\lambda x.x+1)(\underline{(\lambda y.y+2)3}) &\to (\lambda x.x+1)(3+2)
\end{aligned}
$$

現実のプログラミング言語は，通常，決定的な計算規則をもつ[*3]ので，ラムダ計算を現実のプログラミング言語のモデルと考える場合には，適用可能な計算規則を制限する必要がある．適用できる計算規則を定めることを「計算戦略 (strategy) を定める」という．

ここでは，最も重要な 2 つの計算戦略のみをあげる．

定義 4 [値呼び戦略, call-by-value] 関数呼びだし $(\lambda x.M)N$ の計算において，まず，実引数 N を計算して，次に，その計算結果である v を M に代入して，項 $M\{x := v\}$ を作り，最後にこの項の計算を行い，結果を得る．

定義 5 [名前呼び戦略, call-by-name] 関数呼びだし $(\lambda x.M)N$ の計算において，実引数 N を計算せずに，項 $M\{x := N\}$ を作り，この項の計算を行い，結果を得る．

計算結果のことを「値 (value)」と呼ぶので，第一の戦略は値呼びといわれる．第二の戦略は，$M\{x := N\}$ の計算において，N という実引数を (x という) 名前で参照するので，名前呼びといわれる．

計算の効率の面からいうと，値呼び戦略も名前呼び戦略も一長一短であり，どちらが常に優れている，ということはない．そこで，両者の長所をとった必要呼び戦略 (call-by-need) というものがある．これは，名前呼び戦略に似ているが，$M\{x := N\}$ という代入を行わずに (代入をしてしまうと，N が たくさんコピーされる可能性がある)，実際には N へのポインタをばらまくものである．N の計算が何回も呼ばれる可能性があるが，最初に呼ばれたときに N へのポインタの先を N の計算結果に置きかえてしまえば，2 回目からは結果を拾うだけでよく，計算が高速になる．

[*3] これは必ずしも正確ではない．プログラミング言語の仕様書の中には，「どちらであるか決めない」ということにより，非決定的な計算規則を許すものがある．たとえば，C 言語の仕様書では，関数の実引数を計算する順番は決まっていない．したがって，`foo(a,b)` という関数呼び出しにおいて，a と b ともに副作用をもつ計算の場合，どちらを先に計算するかによって結果は異なることがある．また，並列プログラミング言語は必然的に非決定的である．

2.6 再帰定理

ここまでのところ，型のないラムダ計算の体系は大変シンプルであるため，つまらない体系のように思える．実際，現代的プログラム言語はどれも，ラムダ計算よりはるかに豊富で複雑な構文，計算規則をもっている．

次の定理は，その予想に反して，「計算」という観点では，型のないラムダ計算はとんでもない力を秘めていることを示している．

まず、部分関数 $f;\ A \to B$ とは、A のすべての要素に対して、B の要素 1 つを対応付けるかあるいは、1 つも対応付けない (未定義) のいずれかであるような対応付けである。関数 $f : A \to B$ は、A のすべての要素に B の要素 1 つを対応付けるので、関数と部分関数の違いは、後者は、未定義を許すことである。たとえば、$f(x) = 1/x$ は、実数から実数への部分関数であるが、関数ではない。($f(0)$ は未定義である。) この f は、正の実数から実数への関数であり、部分関数でもある。部分関数を、関数と区別して、$f;\ A \to B$ と書くことがある (コロンのかわりにゼミコロンを使う。)

定理 1（計算可能関数） $\mathcal{N}$ を自然数の集合とし，$\mathcal{N}^n$ を $\mathcal{N}$ の n 個の直積とする．

部分関数 $f;\mathcal{N}^n \to \mathcal{N}$ に対して，以下の条件は全て同値である．

- f はチューリング機械 (Turing Machine) で表現可能である．
- f は partial recursive function (帰納的部分関数) である．
- f は型のないラムダ計算で定義可能である．

チューリング機械は，非常に原始的なコンピュータ (というより，プログラム言語) であるが，現代的な高性能コンピュータと同等の計算能力を持っている．したがって，上記の定理は，ラムダ計算も現存するあらゆる高性能なプログラム言語と同等の計算能力を持っていることを示している．

今日のコンピュータ科学の最重要の基礎の 1 つである，Church の提唱 (Church's Thesis, Church-Turing の提唱ともいわれる) は，計算可能 (部分) 関数とは，上記のいずれかで定義される (部分) 関数のことと定義しよう，というものである．

上の定理は驚くべきものであるが，その証明の詳細をここで示すことはできない．そのかわりに，証明の鍵となる定理を述べておく．

定理 2（再帰定理 (Recursion Theorem)） 型なしラムダ計算において，再帰呼び出しは常に解を持つ．すなわち，

どんな項 F と変数 x に対しても，$fx = Ffx$ が成立する項 f を作ることができる．

ここで $=$ は β-簡約によって一致させることができる，という意味である．

再帰定理の証明のためには

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

というラムダ項 (Curry の不動点演算子, Y-combinator) を考えればよい．すると，

$$\begin{aligned} YFx &\equiv (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))Fx \\ &\to (\lambda x.F(xx))(\lambda x.F(xx))x \\ &\to F((\lambda x.F(xx))(\lambda x.F(xx)))x \\ F(YF)x &\equiv F((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))F)x \\ &\to F((\lambda x.F(xx))(\lambda x.F(xx)))x \end{aligned}$$

となり，YFx と $F(YF)x$ が β-簡約によって一致することがわかった．よって、どんな F に対しても YF という項を考えれば、$fx = Ffx$ の解 f となる。

例 4　自然数の階乗を求める関数 f は，Scheme 言語では，以下のように「定義」される．

```
(define (f n)
  (if (= n 0) 1
    (* n (f (- n 1)))))
```

これは、数学的には「定義」ではない。なぜなら、f の定義の中に f が現れるからである。しかし、プログラミング言語ではこのようなものも「定義」と見なしている。そこで、なぜ、これが「f の定義」と見なしてよいかを再帰定理を使って説明しよう。

上記の「定義」は、ラムダ計算では，以下のような方程式としてあらわせる。

$$\mathtt{f}n = (\lambda f.\lambda n.\mathtt{if}(n = 0,\ 1, n * (f(n-1))))\mathtt{f}n$$

ここで、if-then-else を if(a,b,c) の形で書いた。

$F = \lambda f.\lambda n.\mathtt{if}(n = 0,\ 1, n * (f(n-1)))$ とおくと、上記の方程式は、$\mathtt{f}x = F\mathtt{f}x$ の形をしている。再帰定理により，この方程式は YF という解を持つ．すなわち，上記の Scheme の「定義」が、YF という項を定義していると思えばよいことになる。(厳密にいうと、再帰定理は、YF が解の 1 つになっていることを主張しているだけなので、上記の Scheme の定義が YF を定義しているつもりなのか、他の解を定義しているつもりなのかは定められないが、少なくとも、定義されるものが (1 つは) 存在する、ということは保証される。)

このようにして、再帰呼び出しによる関数定義 (伝統的な数学の範囲では、定義になっているかわからないもの) は、ラムダ計算の中で、再帰定理の形で、きちんと捉えることができる。

[2015/10/5 追加] 上記のテキストにおいて、Scheme 言語との関係は、やや不備があった。というのは、上記の Y という不動点演算子を、実際のプログラム言語 (Lisp や Scheme) でそのままプログラムとして記述しても、停止しないプログラムになってしまうからである。実際の Scheme では、上記の f という関数は、非負の整数に対して、その階乗を返してくれるので、停止しないプログラムを定義しているとは思えない。

この原因は、実際のプログラム言語は、「値呼び」という特定のやりかたで計算するためである。上記の再帰定理の証明では、値呼びとは別の (ある特定の) 順番に計算を進めたが、これは Scheme とは異なる順番であった。

そこで、Scheme などの値呼び計算の場合に利用できる不動点演算子というものが考えられる。これを Y_{cbv} と書くことにすると、次のように与えられる。

$$Y_{cbv} = \lambda f.(\lambda x.\lambda y.f(xx)y)(\lambda x.\lambda y.f(xx)y)$$

先ほどの Y とは少しだけ異なっており、これを Scheme や Lisp のプログラムとして記述すれば、きちんと (計算が止まるべきときには、ちゃんと止まる) 再帰関数を与える。

Scheme の一種である MzScheme (現在の Racket) 処理系で、Y_{cbv} と 上記の階乗関数を走らせた例を以下に載せる。

```
% mzscheme
Welcome to Racket v5.2.1.

(let ((y (lambda (f)
           ((lambda (x) (f (x x)))
            (lambda (x) (f (x x)))))
      )
      (factorial_gen (lambda (f) (lambda (x)
         (if (= x 0) 1 (* x (f (- x 1)))))))
      )
  ((y factorial_gen) 5))

^C (not terminating)
>
(let ((y_cbv (lambda (f)
               ((lambda (x) (lambda (y) ((f (x x)) y)))
                (lambda (x) (lambda (y) ((f (x x)) y)))))
      )
      (factorial_gen (lambda (f) (lambda (x)
          (if (= x 0) 1 (* x (f (- x 1)))))))
      )
  ((y_cbv factorial_gen) 5))

120
```

1 つ目の実行で^C とあるのは、実行が停止しないため、ctrl-C で停止した、という意味である。一方で、2 つ目の実行では Y_{cbv} を使っているので、きちんと計算が停止し、5 の階乗である 120 が答えとして返ってきている。