第2章　ラムダ計算（λ-calculus）

2.1　ラムダ計算とは？

ラムダ記法とは、関数を簡潔に記述するための記法であり、ラムダ計算とはラムダ記法を用いて、関数の性質について論じるための体系である。ラムダ計算は、単純で、しかも形式的な扱いが可能である。なお、ラムダはギリシャ文字の λ のことであり、関数を表記するのに、この λ という文字を用いることから、この名で呼ばれている。

ここでは型無し（untyped）のラムダ計算を紹介することにする。

ラムダ計算はひじょうに単純な体系であるが、＿＿＿＿＿＿＿＿＿＿＿＿や＿＿＿＿＿＿＿＿などの計算モデルと同等の計算能力を有することを知られている。つまり、現実のコンピュータと理論的には同等の計算能力を持つのである。

問 2.1.1 チャーチの提唱（*Church's thesis*）という言葉について、その意味を調べよ

ラムダ記法・ラムダ計算は理論的な計算機科学のなかで、ひじょうに良く使われる。この章ではラムダ計算のごく基本的な部分を紹介する。

2.2　ラムダ式のかたち（文法）

変数を表す記号が $x, y, z, \ldots$ のように定められている（通常はアルファベット 1 文字）として、次のような "かたち" をしているものをラムダ式と呼ぶ。

1. 変数 — 任意の変数記号はラムダ式である。
2. 関数適用 — M, N をラムダ式とするとき、(MN) もラムダ式である。
   この式の直観的な意味は、＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿である。
3. 関数抽象 — M をラムダ式、x を変数とするとき、$(\lambda x.M)$ もラムダ式である。
   この式の直観的な意味は、＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿である。

BNF で表現すると以下のとおりである。

$$M \quad ::= \quad V \mid \text{“(”}\ M\ M\ \text{“)”} \mid \text{“(”}\ \text{“}\lambda\text{”}\ V\ \text{“.”}\ M\ \text{“)”}$$
$$V \quad ::= \quad \text{“}x\text{”} \mid \text{“}y\text{”} \mid \text{“}z\text{”} \mid \cdots$$

この他に 1, 2, 3, ... や +, – などの定数を導入する場合もあるが、しばらくは、変数・関数適用・関数抽象の 3 つのみからなる純ラムダ計算を紹介する。

ラムダ式の例

1. $(\lambda x.x)$ — これは x という引数を受け取って、x をそのまま返すので、恒等関数を表している。Scheme の記法ならば

   ```
   (lambda (x) x)
   ```

   という関数であり、C ならば、

   ```
   int id(int x) { return x; }
   ```

   という関数 `id` に相当する。(λ 式や Scheme では x が `int` 型という制限はないが、C ではこのように型の制限のない関数を定義することはできないので、便宜上 `int` 型にしている。)

2. $(\lambda x.(\lambda y.x))$ — これは、x という引数を受け取り、$(\lambda y.x)$ という関数を返す関数である。そして、$(\lambda y.x)$ という関数は、y という引数を受け取り、(これを無視して) x を返す関数である。つまり、式全体は高階関数である。

   λ 記法には、多引数の関数を表す記法はないので、このような "関数を返す関数" で、多引数関数の代用とすることが行われる。つまり、C の記法で、

   ```
   int foo(int x, int y) { return x; }
   ```

   と表される 2 引数の関数を $(\lambda x.(\lambda y.x))$ と書いてしまうのである。このように、多引数関数を "関数を返す関数" として表現することを、_____と言う。Curry は、著名な数理論理学者 Haskell B. Curry の名にちなんでいる。

3. $(\lambda f.(\lambda x.(f(fx))))$ — 関数 f とデータ x を受け取って、f を x に 2 回適用する関数である。

2.3 ラムダ計算のきまり(計算規則)

算数で $1+2\times3 \to 1+6 \to 7$ というように計算の規則があるように、ラムダ計算も計算規則(書き換え規則)が決められている。例えば $(\lambda x.x)$ は恒等関数であり、任意のラムダ式 M に対して、$((\lambda x.x)M) \to M$ と書き換えることができる。また、$(((\lambda f.(\lambda x.(f(fx))))M)N) \to ((\lambda x.(M(Mx)))N) \to (M(MN))$ である。

ラムダ計算の変換規則を正式に紹介する前に、いくつかの必要な用語を説明しておく。

束縛変数・自由変数 $(\lambda x.M)$ という部分式があるとき、x はこの部分式で束縛され(bound)ているという。M の中で束縛された形で出現(occur)する変数を _____(bound variable)、束縛されていない形で出現する(自由に出現する)変数を _____(free variable)という。

例えば、$(\lambda x.(xy))$ の x は束縛変数だが、y は自由変数である。また、$((\lambda z.z)z)$ の z は束縛された形でも、自由にも出現している。一番右端の z は $(\lambda z.\ldots)$ という形の中に入っていないからである。

この最後の例のように、束縛変数と自由変数に同じ名前が使われていると、混乱の元である。そこで、以下の議論では束縛変数は自由変数と名前がぶつからないように、適宜、名前の付け替えをするものと仮定する。

一般にプログラミング言語で仮引数の名前は、他の変数とぶつからない限り、付け替えても良い。ラムダ記法でも同様であり、$(\lambda x.(yx))$ と $(\lambda z.(yz))$ は同じものと見なされる。（このように変数の名前を付け替えることを＿＿＿＿と呼ぶことがある。）ただし、$(\lambda x.(yx))$ と $(\lambda y.(yy))$ は別物である。このように名前の衝突する α 変換 は許されない。

置換　ラムダ式 M, N と変数 x があるとき、$[N/x]M$ という記法を M の中の自由な x の出現をすべて N で置き換えて得られるラムダ式を表すものとする。（この $[\_/\_]\_$ という記法自体はラムダ式の枠外の、“メタ”な記法である。）

例えば、$[(\lambda z.z)/x](\lambda y.(xy))$ は、$(\lambda y.((\lambda z.z)y))$ となるが、$[(\lambda z.z)/y](\lambda y.(xy))$ は、$(\lambda y.(xy))$ のままである。y は $(\lambda y.(xy))$ の中に自由に出現していないからである。

β 変換　ラムダ計算の計算規則は、基本的に β 変換と呼ばれる変換規則のみである。これは、ラムダ式の中の

$$((\lambda x.M)N)$$

という形をした部分式を

$$[N/x]M$$

に書き換える変換である。この書き換えが適用可能な部分式のことを＿＿と呼ぶ。

例:　$(((\lambda f.(\lambda x.(f(fx))))(\lambda y.y))z) \xrightarrow{\beta} ((\lambda x.((\lambda y.y)((\lambda y.y)x)))z) \xrightarrow{\beta} ((\lambda y.y)((\lambda y.y)z)) \xrightarrow{\beta} ((\lambda y.y)z) \xrightarrow{\beta} z$

もっと面白い例として＿＿＿＿＿＿＿＿＿＿＿＿というラムダ式は、β 変換の結果、自分自身に戻る。ラムダ計算には通常のプログラミング言語にあるような繰り返し文や再帰がないが、それでも止まらない計算を表現することができるということがわかる。

これ以上 β 変換を施すことができないラムダ式を＿＿＿＿という。M から β 変換を繰り返して、N という正規形に到達するとき、N を M の正規形と呼ぶ。上の例でわかるように正規形を持たないラムダ式というものも存在する。

2.4　ラムダ式の略記法

上の定義のままだと、括弧が多くなりすぎるので、次のような略記法の約束を導入して、括弧の数を節約する。

1. $\lambda x_1 x_2 \cdots x_n.M \equiv (\lambda x_1.(\lambda x_2.(\cdots(\lambda x_n.M))))$
   つまり、λ 抽象が続く場合は λ を節約して 1 つだけ書く。

2. $M_1 M_2 M_3 \cdots M_n \equiv ((\cdots((M_1 M_2)M_3)\cdots)M_n)$
   つまり、関数適用は左に結合する。

$\lambda xy.M_1M_2M_3$ は $(\lambda x.(\lambda y.((M_1M_2)M_3)))$ の略記となる。つまり、λ 抽象よりも関数適用の方が優先度が高い。また、$(\lambda x.M_1)M_2$ は括弧を省略してしまうと、$\lambda x.(M_1M_2)$ と区別がつかなくなってしまうので、括弧は省略できない。

BNF で表現すると以下のようになる。

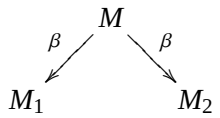
$$
\begin{array}{lcl}
M & ::= & F \mid \text{“}\lambda\text{”}\ W\ \text{“.”}\ M \\
F & ::= & A \mid F\ A \\
A & ::= & V \mid \text{“(”}\ M\ \text{“)”}
\end{array}
\qquad\qquad
\begin{array}{lcl}
W & ::= & V \mid V\ W \\
V & ::= & \text{“}x\text{”} \mid \text{“}y\text{”} \mid \text{“}z\text{”} \mid \cdots
\end{array}
$$

例: $\lambda fx.f(fx)$ は $(\lambda f.(\lambda x.(f(fx))))$ の略記であり、$(\lambda x.xx)(\lambda x.xx)$ は $((\lambda x.(xx))(\lambda x.(xx)))$ の略記である。

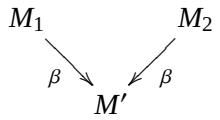
2.5 ラムダ計算の性質

よく知られているラムダ計算の性質を証明なしで紹介する。

チャーチ・ロッサー（Church-Rosser）の定理 ひとつのラムダ式に幾通りもの β 変換が可能なことがある。このとき、異なる β 変換を行なうと、別の形に枝分かれしてしまう。



しかし、うまく何回か β 変換するとこの枝分かれしたものを再び合流させることができる、



ということを述べている定理である。

これは同時に、あるラムダ式に正規形が存在するならば、それは一つしかない（α 変換による違いを除く）ということを保証している。

最左戦略 正規形が存在するラムダ式でも、下手に（上手に?）β 基を選んでいけば、いつまでも β 変換をし続けることがありうる。しかし、最も左からはじまる β 基を選んで行けば、正規形の存在するラムダ式ならば、必ず正規形に到達することが可能である。

例: $(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$ は、$(\lambda x.xx)(\lambda x.xx)$ の部分を β 変換していると、いつまでも正規形に到達しないが、最左 β 基を選ぶとすぐに正規形 $\lambda y.y$ になる。

ただし、最左戦略が正規形に到達するために最も効率の良い（つまり β 変換の少ない）方法とは限らない。（むしろそうでないことの方が多い。）

2.6 おもしろいラムダ式

いろいろなデータの表現　純粋なラムダ計算には、整数などの組み込みのデータ型がないため、一見したところ意味のある計算ができるようには見えない。しかし、実際には真偽値・整数・組などのデータは容易に純ラムダ計算の中で表現することができる。

真偽値　$\lambda tf.t, \lambda tf.f$ というラムダ式をそれぞれ *true, false* と呼ぶことにする。また、$\lambda cte.cte$ というラムダ式を *if* と呼ぶことにする。

$if\ true\ M_1\ M_2 \xrightarrow{\beta} M_1$ であり、$if\ false\ M_1\ M_2 \xrightarrow{\beta} M_2$ である。

問 2.6.1 上記の β 変換を *1* ステップずつ書いて確かめよ。

チャーチの数（Church numeral）　$\lambda fx.x, \lambda fx.fx, \lambda fx.f(fx), \ldots$ というラムダ式を 0, 1, 2, ... という整数に対応するという意味で、$c_0, c_1, c_2, \ldots$ と呼ぶ。一般に c_n は

$$\lambda fx.\underbrace{f(f(\cdots(f}_{n\text{ 個}}x)\cdots))$$

というラムダ式である。*plus* というラムダ式を次のように定義する。

$$\lambda mnfx.mf(nfx)$$

$plus\ c_m\ c_n \xrightarrow{\beta} c_{m+n}$ となる。

問 2.6.2 上記の β 変換を、*m=3, n=2* などの具体例を用いて *1* ステップずつ書いて確かめよ。

引き算・かけ算などもやや難しくなるが定義することが可能である。

問 2.6.3 次の関数をチャーチの数に対するラムダ式として定義せよ。

1. *zero* — *0* であるかどうかを判定する述語
2. *mult* — かけ算
3. *pred* — *1* を引く関数（難）
4. *sub* — 引き算（*pred* を使えば簡単）

組　*pair* を $\lambda fsd.dfs$ というラムダ式と定義する。また、*fst, snd* をそれぞれ、$\lambda p.p(\lambda fs.f), \lambda p.p(\lambda fs.s)$ とする。$fst\ (pair\ M_1\ M_2) \xrightarrow{\beta} M_1$ であり、$snd\ (pair\ M_1\ M_2) \xrightarrow{\beta} M_2$ となる。

問 2.6.4 上記の β 変換を *1* ステップずつ書いて確かめよ。

問 2.6.5 リストを表現するために、*cons, nil, isNull, car, cdr* に対応するラムダ式を定義せよ。

Y コンビネータ　$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ というラムダ式を Y と呼ぶ。$YF \xrightarrow{\beta} (\lambda x.F(xx))(\lambda x.F(xx))$ であるが、この右辺を U と置くと、$U \xrightarrow{\beta} FU \xrightarrow{\beta} F(FU) \xrightarrow{\beta} \cdots \xrightarrow{\beta} F(F(\cdots(FU)\cdots))$ となることがわかる。U は F の不動点と考えられるため、Y のことを不動点演算子（fixed point operator）とも呼ぶ。このような Y は再帰関数を定義するのに用いることができる。

例えば、*fact* というラムダ式を次のように定義する。

$$Y\ (\lambda f\, x.\mathit{if}\ (\mathit{zero}\ x)\ c_1\ (\mathit{mult}\ x\ (f\ (\mathit{pred}\ x))))$$

これは、おなじみの階乗の関数を定義している。

問 2.6.6 *fact* が階乗の関数を表現していることを、c_3 などの具体的な数を用いて、確かめよ。*zero, mult, pred* などのラムダ式はすでに定義されているものと仮定して良い。（つまり、$\mathit{pred}\ c_3 \xrightarrow{\beta} c_2$ などは途中のステップを書かなくて良い。）

$$\begin{aligned}
\mathit{fact}\ c_3 &\equiv Y\ (\lambda f\, x.\mathit{if}\ (\mathit{zero}\ x)\ c_1\ (\mathit{mult}\ x\ (f\ (\mathit{pred}\ x))))\ c_3 \\
&\xrightarrow{\beta} Uc_3 \quad \text{ここで } F \overset{def}{\equiv} \lambda f\, x.\mathit{if}\ (\mathit{zero}\ x)\ c_1\ (\mathit{mult}\ x\ (f\ (\mathit{pred}\ x))) \\
&\qquad\qquad U \overset{def}{\equiv} (\lambda x.F(xx))(\lambda x.F(xx)) \\
&\xrightarrow{\beta} FUc_3 \\
&\xrightarrow{\beta} \mathit{if}\ (\mathit{zero}\ c_3)\ c_1\ (\mathit{mult}\ c_3\ (U\ (\mathit{pred}\ c_3))) \\
&\xrightarrow{\beta} \ldots
\end{aligned}$$

2.7　この章のまとめ

以上でラムダ計算が、ひじょうに単純な体系ながら、強力なプログラミング言語とみなすことができるということがわかる。少なくとも再帰と条件分岐などの制御構造、整数などのデータ型をラムダ計算の中で表現することが可能である。また、2 つのラムダ式が等価であるという議論も比較的容易にできる[1]。

原理的には、より複雑なプログラミング言語の意味をラムダ式として表現することも可能である。しかしながら、実際にすべての計算を純粋なラムダ計算で記述すると、量が多くなりすぎてたいへんである。そこで、次章では Haskell というプログラミング言語を紹介する。Haskell は、基本的にはラムダ計算に、いろいろな便利な構文（構文上の糖衣）と高度な型システムを導入した（だけの）プログラミング言語である。

この章の参考文献

[1] 中島玲二・長谷川真人・田辺誠「コンピュータサイエンス入門 — 論理とプログラム意味論」岩波書店, 1999 年, ISBN4-00-006190-9

[1] 本当は、2 つのラムダ式が等価であるという議論が簡単にできるのは、正規形が存在する場合だけである。正規形が存在しないラムダ式の場合には、互いに β 変換できないのに、“同じ” としか考えられないラムダ式が存在する。これが、D_∞ や $P\omega$ などの領域（domain）に関する理論が必要な理由である。

補足資料が http://www.kurims.kyoto-u.ac.jp/~cs/csnyumon/から手に入る。この補足の A 章にラムダ計算の解説がある。

[2] 高橋正子「計算論 — 計算可能性とラムダ計算」近代科学社, 1991 年, ISBN4-7649-0184-6
ラムダ計算について丁寧に解説している。

[3] ラビ・セシィ（神林 靖 訳)「プログラミング言語の概念と構造」アジソン・ウェスレイ, 1995 年, ISBN4-7952-9663-4
12 章にラムダ計算の解説がある。