ソフトウェア工学 97−13
（1994. 3. 1）

ラムダ計算と図式表現

杉藤　芳雄
電子技術総合研究所　情報アーキテクチャ部　言語システム研究室

あらまし　ラムダ計算に登場するラムダ式の記号列表現は一般に全体構造が把握しにくく、従って式の変換操作も間違いやすい。そのため、ラムダ式の図式化が研究されてきた。本稿では、ラムダ式の図式化に関する従来の提案の概観を、筆者の提案であったラムダチャートを含めて行ない、次いで、再帰関数のようにラムダ計算として大規模になる問題への対処として、ラムダチャートの基本精神を保持しつつ記述の分量や手間を軽減する略記法（仮称ラムダ略図）を新たに提案する。そして、再帰関数の具体的な記述例として階乗関数を題材に採り上げ、本略記法の大規模問題への有効性を示す。

Lambda Calculus and its Diagrammatic Representations

Yoshio SUGITO
Computer Language Section, Computer Science Division
ELECTROTECHNICAL LABORATORY
1-1-4 Umezono, Tsukuba-shi, Ibaraki-ken, 305, JAPAN

Abstract　As it is hard to recognize correctly the structure of string representations of expressions appeared in Lambda Calculus, we are apt to make an error in its manipulations. So, to cope with the situation, the studies on diagrammatical representations of Lambda expressions have been performed. In this paper, at first, we review the methods on diagrammatical representation of Lambda expressions to be proposed until now including our Lambda Chart. But unfortunately, in case of large scale problems such as a recursive function, it is tedious to describe precisely each element appeared in a diagram, and the size of a diagram is inclined to increase redundantly. Therefore, we propose a simplified version of the Lambda Chart as a new method for such problems. And, by means of showing concretely the change of diagrams in the converting process on a factorial function as an example of a large scale problem, we demonstrate the validity of our new method.

1 はじめに

ラムダ計算 (λ-calculus)[1][2] は、そもそもは関数の本質を理論的に究明するために導入された体系であるが、計算機科学の分野でも関数型言語のようなプログラム言語との密接な関連性から有効な道具立てとしての地位を獲得している。

しかし、その簡素な記法にも拘らず、あるいはそれゆえにこそ、ラムダ計算という計算の正確な遂行がとくに初心者には容易ではないことも遺憾ながら事実であろう。

その主な原因は、LISP 言語のように括弧と文字が交錯していて全体構造が把握しにくいラムダ式という対象を相手に、煩雑な条件の付いた代入規則等の変換規則を正しく用いることの困難さにある。

このような状況を打開することを主眼として検討されてきたのが、ラムダ計算に登場する式を図式化する試みであり、それによって式の構造や計算の進行状況が視覚的にも把握しやすくなることを目指している。

本稿では、ラムダ計算の簡単な紹介のあと、ラムダ式の図式化に関する従来の提案の概観を、筆者のラムダチャート [5] を含めて行なう。次いで、再帰関数のようにラムダ計算として大規模になる問題への対処として、ラムダチャートの基本精神を保持しつつ記述の分量や手間を軽減する略記法 (仮称ラムダ略図) を新たに提案する。そして、再帰関数の具体例として階乗関数を題材に採り上げ、本略記法の大規模問題への有効性を示す。

2 ラムダ計算の素描

ラムダ計算とは、λ 記号、括弧、"." を補助記号とし、変数という記号の無限系列の存在を仮定した上で、次のように帰納的に定義される式 (expression) [ラムダ式あるいはラムダ項ともいう] に関する計算体系である。

定義 1. **式**
(i) 総ての変数は式。
(ii) X と Y が式ならば、(XY) は式。[この形の式を**適用** (application) という]
(iii) Y が式で x が変数ならば、(λx.Y) は式。[この形の式を**抽象** (abstraction) という]

とくに抽象の式において、λ が前置された変数に対する一般的な術語は見当らないようであるが、本稿では抽象変数と呼ぶことにする。以降、原則として変数は小文字、式は大文字の英字で表わすことにする。また、式に含まれる括弧に対しては、次の様な省略法 (association to the left) を用いることがある。

$$WXYZ \equiv (((WX)Y)Z)$$
$$\lambda x.XY \equiv (\lambda x.(XY))$$

厄介なことには、式 (とくに抽象) の表記には各種の流儀が存在しており、例えば次の 3 者は同じ式に対する表現である。(本稿では基本的には中央の表記を採用する。)

$\lambda x.\lambda y.\lambda z.XYZ$　　$\lambda x\lambda y\lambda z.XYZ$　　$\lambda xyz.XYZ$

ラムダ計算の計算過程で中心となる概念は後述する "代入" であるが、その際には変数が式内で自由状態か束縛状態かが鍵となる。次にそれらの定義を行なう。

定義 2. **自由** (free)
(1)x は x 内で自由として出現 (以下では単に「自由」と記す)。
(2)M または N の一方 (あるいは両方) 内で x が自由ならば、x は MN 内で自由。
(3)x と y が異なる変数で、x が M 内で自由ならば、x は λy.M 内で自由。
(4)M 内で x が自由ならば、x は (M) 内で自由。

定義 3. **束縛** (bound)
(1) 単一変数のみの式内では、いかなる変数も束縛として出現 (以下では単に「束縛」と記す) することはない。
(2)M または N の一方 (あるいは両方) 内で x が束縛ならば、x は MN 内で束縛。
(3)x と y が同一変数か、または M 内で x が束縛ならば、x は λy.M 内で束縛。
(4)M 内で x が束縛ならば、x は (M) 内で束縛。

いよいよ、ラムダ計算の中心的な変換操作であり、煩雑さで悪名高い代入規則の定義に入る。

定義 4. **代入** (substitution) **規則**
x を変数、M,N を式とする。このとき M[x:=N] (即ち、M 内に x が出現すれば N を代入) は次のように定義される式 M' である。

1. M が単一変数のとき:
   (a) M≡x ならば M'≡N
   (b) M≠x ならば M'≡M
2. M が適用 YZ のとき:
   M'≡ (Y[x:=N])(Z[x:=N])
3. M が抽象 λy.Y のとき:
   (a) y≡x ならば M'≡M
   (b) y≠x ならば
      i. x が Y 内で自由ではないか、あるいは y が N 内で自由ではない場合
         M'≡ λy.Y[x:=N]
      ii. x が Y 内で自由であり、かつ、y が N 内で自由である場合
         M'≡ λz.(Y[y:=z])[x:=N]
         ここで z は、N および Y 内で自由ではない変数の系列での初出の変数。

例　x[x:=N] ≡ N
y[x:=N] ≡ y　　(ただし y≠x)
(xy)[x:=N] ≡ (x[x:=N])(y[x:=N]) ≡ Ny
(λx.Y)[x:=N] ≡ λx.Y
(λy.ay)[x:=by] ≡ λy.ay[x:=by] ≡ λy.ay
(λy.axy)[x:=b] ≡ λy.axy[x:=b] ≡ λy.aby
(λy.axy)[x:=by] ≡ λz.(axy[y:=z])[x:=by] ≡ λz.axz[x:=by] ≡ λz.abyz

上記の代入規則は、次に述べる変換規則の一種である。変換規則は、ラムダ計算に登場する計算操作の総てである。

定義 5. 変換 (conversion) 規則
以下の 3 規則は、両方向の CNV で成立するものである。

(α)	(変数名の変更) y が M 内で自由ではないならば	λ x.M	CNV_α	λ y.M[x:=y]
(β)	(代入規則) [定義 4 で述べたもの]	(λ x.M)N	CNV_β	M[x:=N]
(η)	(Extensionality 公理) x が M 内で自由ではないならば	λ x.Mx	CNV_η	M

ここで特に (β) と (η) は、左から右への変換で抽象を含む式をより簡単な式に置き換えるので**還元** (reduction)[記号 RED] と称し、置き換えられる特定の式 (即ち左辺) をリデックス (redex) と呼ぶ。尚、定義 4 の代入規則の場合分けの最後には、抽象変数に関する y から z への (α) 変換が含まれていることに注目したい。この事実は、一見無意味に思える (α) 変換にも、ラムダ計算の中心的な変換である (β) 変換を支援するという活躍の場があることを意味しているからである。

例　$(\lambda x\lambda y.y)ab \quad RED_\beta \quad (\lambda y.y)b \quad RED_\beta \quad b$
$\lambda x\lambda y.axy \quad RED_\eta \quad \lambda x.ax \quad RED_\eta \quad a$

3　ラムダ式の図表現

前章で述べたラムダ計算の概要から、ラムダ式を図表現する理由や目的を考えると、次のようなことが挙げられる。
(1) 初心者に補助的あるいは視覚的な表現形態を提供すること。
(2) 記号列での式表現における各種 “方言” の存在への “共通語” を提供すること。
(3) 記号列のままでは変数や括弧の氾濫の中で全体構造の正確な把握が困難なこと。
(4) 記号列のままでは代入規則のような煩雑な変換規則の正しい運用が困難なこと。
(5) 変換規則をグラフ還元 (graph reduction) 的見地から把握すること。

前章から明らかなように、ラムダ式を図表現する際には、適用や抽象の扱い方が中心課題である。従来のラムダ式の図式化の流れを丁寧に説明している文献 [4] によれば、結局はそれらの配置を縦 (垂直) 方向か横 (水平) 方向のいずれに定めるべきかということになる。例えば、ラムダ図 LAD[4] では適用を縦、抽象を横に配置し、T 式 [3] や筆者のラムダチャート [5] では丁度その逆になっている。

[4] では適用を横に配置するとラムダ式が横長になりがちであることを指摘している。とくに再帰関数の場合には Y コンビネータの登場で横長が顕著になるという。

この再帰関数の問題は、次章で導入するラムダ略図により実際に階乗の再帰関数を記述する際に検討する。

変換規則の運用の容易さを重視する観点から、適用を横方向に、抽象を縦方向に、それぞれ配置する方式を [5] で採用した。適用を横方向に配置することは、記号列表現で慣れ親しんできた横方向の適用との対応付けが容易かつ自然であり、左項に右項を適用して代入するという同様に親しんできた操作も横方向に適用を配置すればこそである。

抽象を縦方向に配置すること、即ち、抽象変数を縦方向に上から下に配置することの利点は、代入操作の進行により構造が全体的にせり上がることを視覚的に把握できることや、代入規則における (α) 変換を施すべき箇所の指摘し易さにある。即ち、この (α) 変換は、重複する抽象変数名のうちで下位にある方のそれを頂点とする部分木内に対してのみ実施すればよい。

全体構造の把握しやすさという観点からラムダ式の図表現を検討する場合、抽象変数とそれ以外の変数との形状面からの区別は予想以上に看過できない。ところが、[3] や [4] では、いずれも抽象変数と他の変数とを形状面からは区別せずに四角形のままであり、変数名に λ 記号を添えることの有無による区別で済ませている。

そこで、抽象変数には他の変数と異なる形状を付与すれば、わざわざ λ 記号を添える必要もなくなると同時に、変換規則の運用の際にも抽象変数の形状が良い目印となり、運用箇所 (即ち、リデックス) の探索に好都合である。実際、[5] では抽象変数は λ 記号なしで円形内に、その他の変数は四角形内に、それぞれ配置している。

また、[5] では便法を積極的に採用している。例えば、括弧の省略記法が可能な箇所では、それに基づいた構造を利用すること、適用構造を同一水準上に並べる場合には各要素の四角形を密着させてもよいこと、注釈の記述を許して部分式の頂点に構造名を添えたり、(α) 変換の結果を記したりすること、等である。

4 略記法の導入

ここまでラムダチャートの記述例のような図表現を一切明示しなかったのは、無論意図的にである。一つには必要ならば文献 [5] 等を参照していただくことで重複を避けたかったためであるが、より重要なこととしては、再帰関数の問題のように式の展開が大規模なものになる問題を記述するためにはラムダチャートの精神を継承しつつ記述の分量や手間を軽減するラムダ略図の方がふさわしく、従って本稿ではラムダチャートへの予備知識や先入観に囚われず直接ラムダ略図を導入すべきと考えたからである。

ラムダ略図では、ラムダチャートにおける縦の抽象、横の適用という配置原理は踏襲するものの、抽象変数の円形やその他の変数の四角形は省略して、それぞれ単に抽象変数名 (ただし角括弧で挟む) や変数名だけで済ます。

まず、ラムダ式の BNF 表記を次のように考える。ここで ⟨変数⟩ は、最終的には任意長の英数字記号列とする。

```
<式> ::=   <変数>
         | <式> <式>          適用 (application)
         | λ <変数>.<式>      抽象 (abstraction)
         | (<式>)             ( )
```

これに対応するラムダ略図の BNF 風の記述は次のようになる。

```
<式> ::=   <変数>

         |  |----|
         |  <式> <式>          適用 (application)

         |  [<変数>]
         |     |               抽象 (abstraction)
         |   <式>

         |     |
         |   <式>              ( )
```

変換規則 (α)、(β)、(η) はラムダ略図では次のように運用する。ただし、ここで x,p は任意の変数、M,N は任意の式である。

```
          [x]                [p]
( α )      |       ==>        |          (p は M 内で自由ではない)
           M                M[x:=p]

        |----|                |
( β )  [x]   N     ==>     M[x:=N]
         |
         M
```

```
        [x]
(η)     |          ==>        M          (x は M 内で自由ではない)
        |----|
        M    x
```

また、抽象や適用における要素の連係のために縦線や横線を用いる場合、次のような便法を用いることもある。

```
|               |              |             |
|-|-|-|   ==>   a-b-c-d       [a]    ==>   [a]
a b c d                       |            [b]
                             [b]            |
                              |
```

5 再帰関数の記述

本章では、階乗関数を前章で導入したラムダ略図により表現し、その計算過程を追跡することに取り組む。

まずは、整数の取扱いから始めることにするが、ラムダ計算では非負整数を次のようなラムダ式で定義する習慣がある。この場合、整数 n は式の本体における変数 f の個数 n に対応している。以下に登場する関数はすべてこのように定義された整数を対象とするものである。

```
0 = λ f λ x.x
1 = λ f λ x.fx
2 = λ f λ x.f(fx)
3 = λ f λ x.f(f(fx))
.....
```

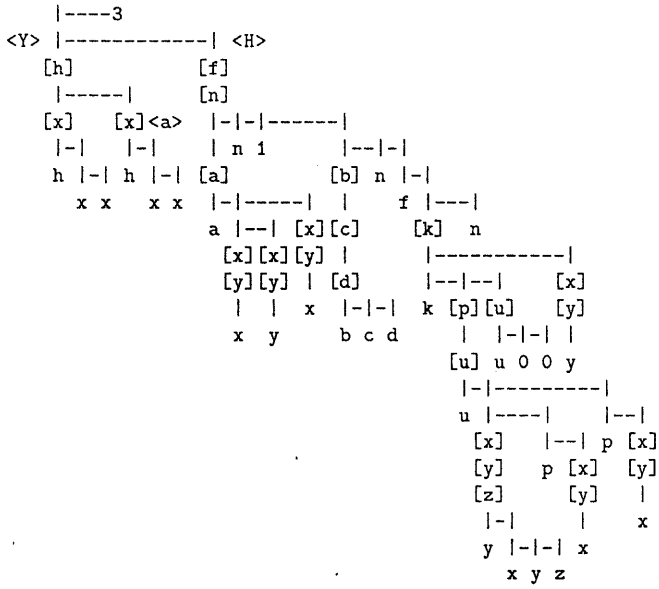
階乗関数 Fact は以下に示すように各種関数の積み重ねで定義されるものである。

```
Fact = Y H
  Y = λ h.(λ x.h(xx))(λ x.h(xx))      ここで   Yg  CNV  g(Yg)  [Y コンビネータ]
  H = λ f λ n.IsZero n 1 (Times n (f (Pred n)))
    IsZero = λ k.k(True(False))(True)
      ここで     IsZero n  CNV  True    n = 0 のとき
                           CNV  False   それ以外のとき
    Times m n = λ m λ n λ f.m n f       ここで   Times m n  CNV  mxn
    Pred = λ k.(k(λ p λ u.u(Suc(p True))(p True))(λ u.u 0 0))False
      ここで     Pred n  CNV  n-1
    True = λ x λ y.x                    ここで   True  a b  CNV  a
    False = λ x λ y.y = 0               ここで   False a b  CNV  b
    Suc = λ x λ y λ z.y(xyz)            ここで    Suc n   CNV  n+1
```

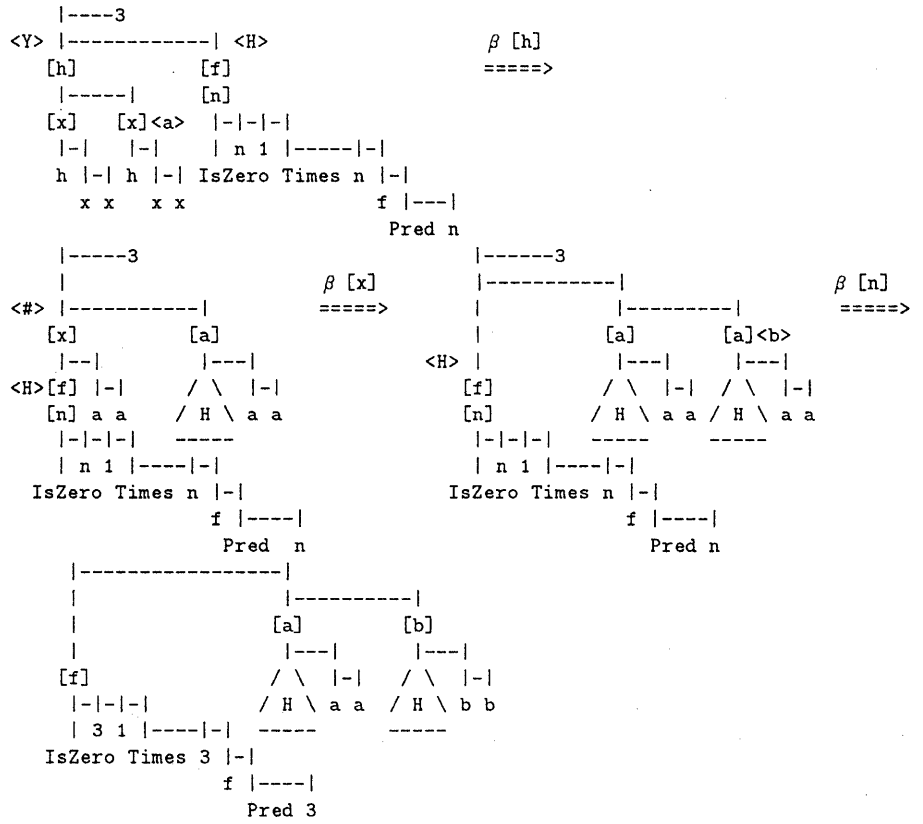
それでは、3 の階乗をラムダ略図の支援によるラムダ計算として求めることにしよう。まず、階乗関数 (Fact 3) を関数 Y,H の定義式で記述すると次のようになる。

```
<Fact>|----3
  <Y> |-------------| <H>
     [h]            [f]
      |-----|       [n]
     [x]   [x]<a>    |-|-|-|
      |-|   |-|      | n 1 |----|-|
      h |-| h |-| IsZero Times n |-|
        x x   x x                f |---|
                                   Pred n
```

これを、数字の定義式以外をすべて展開すると次のようになる。このほぼ完全展開形をを見れば明らかなように、全体構造はとくに横長の傾向があるという訳ではなくて縦にも長くなっており、全体としてほぼ正方形の右下がりの対角線方向に式が展開されていることが分かる。即ち、第 3 章で紹介した、適用を横に配置すると特に再帰関数の場合には Y コンビネータの登場でラムダ式が横長になりがちという [4] の指摘は、少なくとも本問題に関する限り成立していない。一般に、再帰関数のような大規模問題になれば、式は縦および横の両方向に長くなり、適用を縦方向に配置するか横方向に配置するかによる紙面消費上の差は殆どないのが実状である。

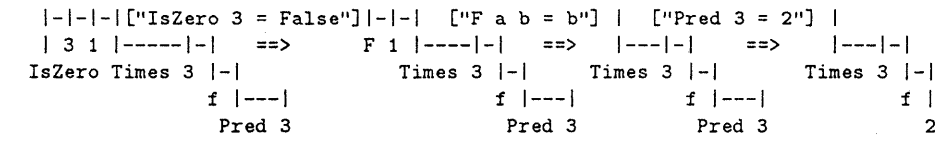


では、実際に 3 の階乗を計算してみよう。以下の計算過程で、内部に H と書かれている三角形は、関数 H に相当する部分木を意味している。

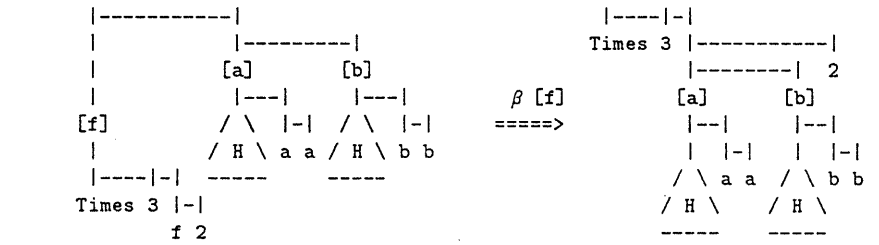


この段階で、部分木 [a]-[b] での変換のみに拘泥すると、この変換過程は永久に続くので停止しなくなる。即ち、引数

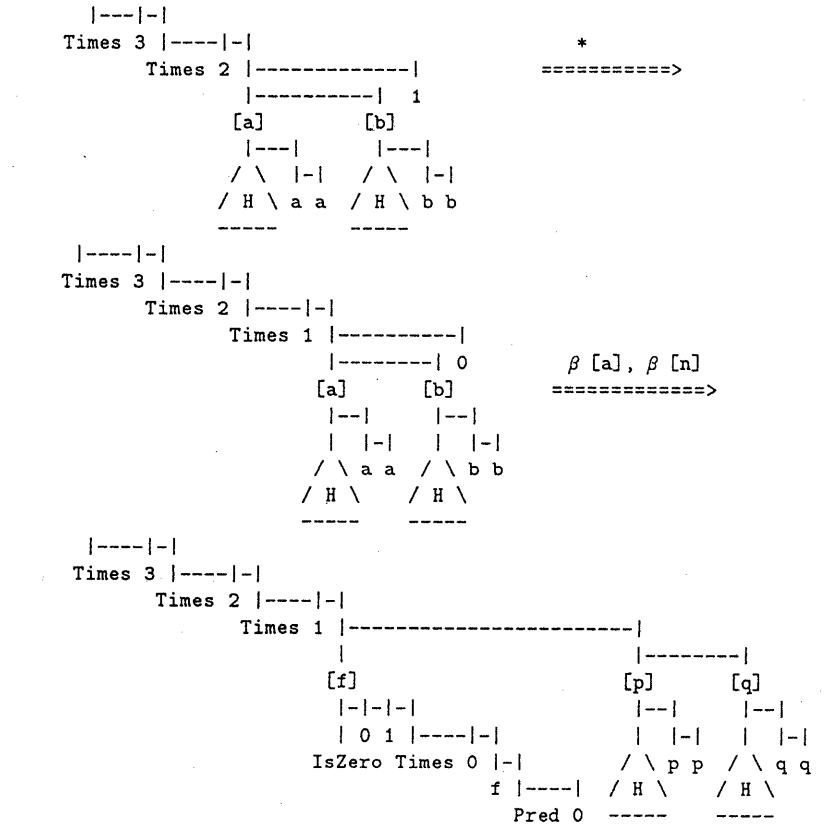
評価を先行させる適用順評価(または値呼出し)は“危険”である。もし、常に最左端リデックスを変換する正規順評価(または名前呼出し)を採用するならば、停止性が存在する場合には計算が必ず停止することが保証されているので“安全”である。それゆえ左部分木[f]に関して変換を行なうことにすると次のようになる。



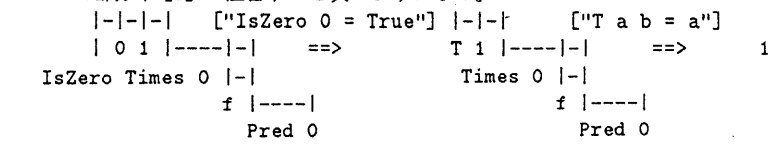
この結果をもとの構造に戻すと次のようになる。



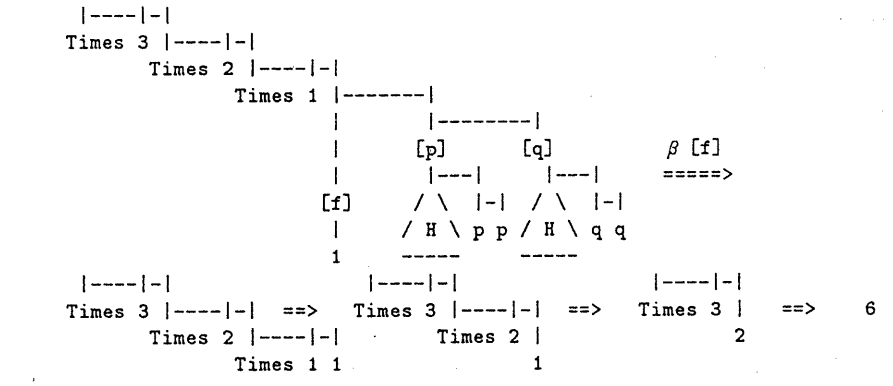
ここで、部分木[a]-[b]は最初の方に既出の部分木⟨#⟩に等しい。それゆえ、同様の計算過程が再び登場することになる。



ここで部分木 [f] に注目すると次のようになる。



この結果をもとの構造に戻すと次のようになる。



即ち、3 の階乗は "Fact 3 = 3! = 6" となる。

6 おわりに

再帰関数のような記述や計算の量が大規模になる場合への現実的な対処法として、筆者が以前に提案したラムダチャート [5] の簡易化をラムダ略図という仮称により提案し、実際に階乗という再帰関数を記述して、その有効性の度合いを調べた。

[5] で述べたように、紙面の消費される方向が横長か縦長かを重視するよりは、変換規則の運用の容易さの方を重視して適用や抽象の配置方法を決定したい、というのが筆者の方針であったが、少なくとも大規模問題に対しては紙面の消費方向に縦か横かの顕著な傾向が見当らないということで、前述の方針の見通しの良さが裏付けられたと言えよう。

ラムダチャートの長所である変換規則の運用の容易さをなるべく温存しつつ、記述の分量や手間を軽減するというラムダ略図の試みは、とくに大規模問題の場合には重畳的に有効であると考える。

今後の課題としては、型付きラムダ計算 (Typed Lambda Calculus) における図式化の検討が依然として挙げられる。

References

[1] A.Church:"The Calculi of Lambda-Conversion", Princeton University Press, 1941.

[2] H.P.Barendregt:"The Lambda Calculus: Its Syntax and Semantics", North-Holland, 1984.

[3] 飯島正, 岡田謙一, 横山光男, 北川節:" T式による関数型プログラム開発 ", 情報処理学会第 33 回全国大会講演論文集, pp.789-790, 1986.

[4] 二村良彦, 野木兼六, 高野明彦:" ラムダ式の図形表現 ", bit, Vol.12, No.12, 1990.

[5] 杉藤芳雄:" ラムダ式の図表現について ", 電子情報通信学会ソフトウェアサイエンス研究会, ss93-19(1993-07).