関数プログラミングの実際

武市 正人

近年，多様化してきているプログラミングパラダイム(手続き型，関数型，論理型，対象指向型など)の中でも，関数プログラミングの考え方は比較的古くから存在しているものである．関数プログラミングの起源がChurch[9]の**ラムダ計算法**(lambda calculus)にあるという指摘もあるが，計算機プログラミング，あるいはソフトウェア工学という観点からは，1960年代のMcCarthy[15]，Landin[14]などがその発祥と見てよいであろう．

関数プログラミングに関する研究では，ラムダ計算法に遡る基礎的な計算の理論や，あらたな計算モデルの構築などについて成果が積み重ねられてきている．しかし，従来の研究がこのような方向に片寄りがちであったことから，アルゴリズムを関数で表現するという意味の関数プログラミングの効用が理解されていない面も少なからずあるように思われる．関数プログラミングを論じる際に計算モデル(計算機構)によって関数プログラムの評価機構を理解することも必要であるが，プログラミングにおいてはプログラムの構成法や検証法などの方法論を確立することのほうがより重要であるといえよう．関数プログラミングに関する特集[1]を見ても，これまでは関数プログラムの評価機構やその基礎理論を扱ったものが多かった．その一方で，最近はプログラミングそのものを論じるものも増えてきている．このことは，10年前に出版されたHendersonによる教科書[10]が評価方式に基づいてプログラミングのことを述べているのに対して，最近のBirdとWadlerによる教科書[8]ではもっぱらプログラムの構成法を扱っていることにも見られる．

本稿では，まず，プログラミングに重点を置いて，手続き型プログラミングと対比させて関数プログラミングの意義を述べ，ついで，関数プログラミングの特徴を概説する．関数プログラム言語やその実現法については最近の動向を示すにとどめる．なお，本稿は活発に進められている関数プログラミングに関する研究を紹介しようということではなく，関数プログラミングの意義を確認するためのものである．したがって，とりあげる話題についても，網羅的ではないことに留意されたい．最近の研究成果については，これまでの[1] [2] [3]の特集や[4] [5]などの国際会議録を参考にされたい．

1　関数プログラミングの意義

関数プログラミングを，そのことばの意味するとおり，関数によるプログラムの構成法という観点から見るとき，これを1960年代末から1970年代にかけて提唱された**構造的プログラミング**(structured programming)に対比させることができる．いずれも，従来より知られていたプログラミングの概念を整理して方法論を提示したものである．関数プログラミングと構造的プログラミングとの関連については[22]や[13]にも指摘されている．構造的プログラミングは増大するソフトウェア経費を抑えるためには，洗練された制御構造とデータ構造を用いてプログラムを設計・作成すべきであるという教えであり，手続き型言語を用いたプログラミングにおいては，現在はもはや当然のこととして実践されていることである．しかし，関数プログラミングの立場から見ると，構造的

Current Trends in Functional Programming.
Masato Takeichi, 東京大学工学部計数工学科.
コンピュータソフトウェア, Vol. 8, No. 1 (1991), pp. 3 11.
1990年8月2日受付.

プログラミングは，必ずしも成功しているとはいえない．

構造的プログラミングにおいては，**下降型プログラム開発**(top-down program development)，**段階的プログラム開発**(stepwise refinement)が推奨される．そこでは，手続き型言語の文の構造化機構(**if** e **then** c_1 **else** c_2 や **while** e **do** c など)とデータの構造化機構(**array**[T_1] **of** T_2 や **record**…**end** など)によって，文やデータの構成要素をより大きな構成単位に組み上げるという機能を利用している．つまり，これらは構成要素を結びつける糊(glue)の役割[13]を果たしているわけである．これに対して，関数プログラミングでは，関数 f を定義し，それに引数 a を適用して値 f a を得ることがプログラムの基本である(ふつうの数学では関数値を f(a)のように書くが，関数プログラミングでは引数を囲む括弧は用いないほうが便利である)．むしろ，その機能しか存在しないといったほうがよいであろう．関数 f と関数 g を合成してあらたな関数 f•g を作ることは数学における合成関数そのものである．しかし，これも関数を合成する関数(•)に引数 f と g を与えた結果の値である．このように，関数プログラミングにおいては，関数を組み合わせる関数がプログラムの構成要素を結合する糊になるのである．

プログラムを複雑な対象を取り扱う問題解決の手段と考えるとき，部分問題を定式化したり部分問題の解を結合する際に，プログラムの構成要素を結合する糊の役割は重要な意味をもってくる．関数プログラミングと手続き型(構造的)プログラミングは，ともに構成要素を結合する機構を備えていて，これらの方法論は共通の原理に基づいていると見ることもできる．しかし，糊づけされる構成要素の種類が異なるのはもちろんであるが，当面はそれを考えないでおいても，これらのあいだには大きな違いがある．関数プログラミングの立場から手続き型(構造的)プログラミングを見るとき，手続き型の糊の弱さのひとつに，そこでは構成要素を結合する際の計算のパターン，たとえば **if** e **then** c_1 **else** c_2 が制御の流れしか捉えていないということがある．手続き型プログラミングは，基本的には計算によって状態(すなわち記憶の内容)を変えてゆくという考え方に立つものであり，構成要素 e, c_1, c_2 を抽象した計算のパターンと状態の変化そのものとを対応づけることは(不可能ではないにしても)簡単なことではない．これは，手続き型言語ではプログラムの意味を定めるにはその表現には現れていない状態も含める必要があるからである．たとえば，

if e **then** {c_0; c_1} **else** {c_0; c_2}

と

c_0; **if** e **then** c_1 **else** c_2

がつねに同じであるとはかぎらない．これらが等価である条件は，計算のパターンとしての糊の性質だけではなく，構成要素の e や c_0，さらには計算状態にまで言及する必要があるのである．

これに対して，関数プログラミングには計算状態という概念は存在しないし，プログラムの記述は値を**表示する**(denote)だけであり，それ以外の対象は計算に関与しないので，計算のパターンにはすべてのことが盛り込まれることになる．たとえば，リスト

$[x_1, x_2, \cdots, x_n]$

の各要素に関数 f を適用する関数 map

$$\mathrm{map}\ f\ [x_1, x_2, \cdots, x_n] = [f\ x_1, f\ x_2, \cdots, f\ x_n]$$

は，どのようなリストを対象にする場合にも，任意の(始域と終域のあっている)関数 f と g について

$$(\mathrm{map}\ f)\cdot(\mathrm{map}\ g) = \mathrm{map}(f\cdot g)$$

の法則が成立する．このように，関数プログラミングにおいては，関数を含む式で表現される一般的な法則を適用して数学的な計算操作によってプログラムを変換したり，合成したりすることができる．しかも，構成要素の f や g の条件は(たとえば型の概念に基づく)簡単な規則で述べられる．この点で，関数プログラミングにおける糊のほうが手続き型のものよりも強力なのである．

手続き型プログラムが状態を変化させるという考え方に基づいているということは，プログラミングの本質を見失わせることにもなりかねない．つまり，プログラムの性質が計算のパターンではなく，究極的な最小の構成要素である代入文などの効果に依存しているので，プログラミングがどこまでも微視的になってしまう傾向がある．ハードウェア技術の進歩により，解くことのできる問題の規模も大きくなり，それにともなってソフトウェアの規模も大きくなってきたが，それでもなお特定の言語によるコーディングがプログラミングの本質であるかのように考えられるという状況は変わっていない．もちろん，手続き型(構造的)プログラミングの分野においても進展があったことは事実であるが，知的生産活動とし

てのプログラミングを重視するという立場では，急進的な変革はもたらされていないといえよう．関数プログラミングの考え方は計算のパターンを抽象化して捉え，そこで成り立つ法則を用いてプログラムを合成したり，その性質を調べたりすることが特徴的である．問題を大局的に捉えるときの計算パターンの性質は個々の構成要素とは独立しており，それ自体としても関数としての存在意義をもっている．したがって，手続き型プログラミングの場においても，問題把握のひとつの段階として関数プログラミングを実践することが有用なことであるといえよう．ことに，「プログラムがどうして正しく動かないのか」悩み，虫取りの結果，「正しく動くようになったのはなぜか」ということが気がかりであるような良心的なプログラマは容易に関数プログラミングの意義を認めることができるであろう．

2 関数プログラミングの特徴

関数プログラミングの考え方として特徴的なことは，関数を**第1級の対象**(first class object)とするということである．手続き型言語の多くのものが関数や手続きを整数などとは異なる対象として扱っているが，関数プログラミングの立場では関数そのものを数のようなデータと同格に扱う．このことは関数を引数としたり，関数を結果の値とするような高階関数の自然な導入につながり，アルゴリズムの記述におけるモジュール性を高める重要な役割を果たすことになっている．すなわち，上にあげた関数 map のように，計算パターンを関数によって抽象することができるからである．

関数プログラムの性質で最も重要なものは，プログラムのなかに書かれている式(表現)の意味が，その文脈のみによって定まり，式の評価過程には依存しないという**参照透明性**(referential transparency)である．参照透明性は関数プログラムの変換や合成などプログラム開発に対する方法論の基礎になっているものであり，このことが関数プログラミングをほかの(手続き型などの)プログラミングに対してきわだたせている点である．

代表的な計算のパターンを表わす記法を(非形式的にではあるが)示しておこう．まず，Turner[21]によって考案された**リストの内包表記**(list comprehension)である．これは，

```
[f x|x<-A; p x]
```

の形のものであり，集合の表記法と似ている．この式は「リスト A の要素 x のうちで，述語 p を満たすものに対する f x の値から成るリスト」を表わす．一般には，f x のところは任意の式であってもよい．たとえば，0 から 10 までの整数から成るリスト[0..10]をもとにして，偶数であることを判定する述語 even によって

```
[x|x<-[0..10]; even x] = [0, 2, 4, 6, 8, 10]
```

となる．また，関数 map を

```
map f xs = [f x|x<-xs]
```

で定義することもできる．これとは逆に，簡単な規則によって内包表記を map とほかのいくつかの関数を用いた式に変換することもできる．

左右のそれぞれの「たたみ込み(fold)」を行う計算パターンも一般的である．

$$\mathrm{foldr\ f\ a\ [x_1, x_2, \cdots, x_n]} = \mathrm{f\ x_1\ (f\ x_2\ (\cdots\ (f\ x_n\ a)\cdots))}$$

$$\mathrm{foldl\ f\ a\ [x_1, x_2, \cdots, x_n]} = \mathrm{f\ (\cdots\ (f\ (f\ a\ x_1)\ x_2)\cdots)\ x_n}$$

二項演算子 + を関数と見るとき，これを(+)で表わすものとする．すなわち

```
(+) a b = a+b
```

であるとするとき

$$\mathrm{foldr\ (+)\ 0\ [x_1, x_2, \cdots, x_n]} = x_1 + (x_2 + (\cdots (x_n + 0)))$$

$$\mathrm{foldl\ (+)\ 0\ [x_1, x_2, \cdots, x_n]} = (((0 + x_n) + \cdots) + x_2) + x_1$$

であり，いずれもリストの要素の総和を表わしている．そこで，これを用いて

```
sum = foldr (+) 0 = foldl (+) 0
```

のようにして，リストの総和を求める関数 sum を定義することができる．すなわち，関数 f と値 a を適切に選ぶことによって，一般的なこれらの計算パターンから具体的な計算を表わす関数を得るのである．さらに，関数(☆)と値 a が

```
x☆(y☆z) = (x☆y)☆z
a☆x = x☆a = x
```

を満たすときには，任意のリスト xs に対して

```
foldr (☆) a xs = foldl (☆) a xs
```

が成立するといった法則が導き出される．計算パターン

のこのような法則に基づいてプログラムを変換したり，プログラムの仕様と実現とを結びつけたりすることが可能になる．ここにあげたような計算パターンを表わす関数の形式的な定義は再帰的に行うが，これらを用いて表現されるプログラム自体は必ずしも再帰的な定義によるものではなく，むしろ既知の構成要素を組み合わせた単純な等式によって定義されるものである．実際，Bird と Wadler による教科書[8]では，第4章までは再帰的な定義はまったく現われないが，たたみ込み関数などを用いてさまざまな分野のプログラムを形式的な手順で構築する方法を提示している．

このようにして構成したプログラムの長さは同じ機能を果たす手続き型のプログラムに比べてはるかに短くなる．しかも，プログラムの構築過程は等式の置き換え原理に基づいているので論理の飛躍がなく，その過程を容易に検証することができる．これらのことは，関数プログラムの参照透明性がプログラミングに大きな貢献をしていることを例証しているといえよう．

高階関数や参照透明性のもたらすソフトウェア工学上の位置づけは前節ですでに述べたことであるが，これとは別の観点からも関数プログラミングの特徴をあげることができる．すなわち，関数プログラミングには基本概念がきわめて少ないということである．手続き型プログラミングを学ぶ際の障壁の一つが**代入**(assignment)の概念であることは随所で指摘されている．これは，計算過程と計算状態という考え方が初心者にとっては新奇なものでなじめないということにも理由があろう．また，同時に式，文，データなどの多様な計算対象を理解しなくてはならないということもある．これらの計算対象相互の関係を理解して，問題解決のプログラムを書くことができるようになるには相当の修練を要することである．これらの問題点は，手続き型プログラムが膨大な状態数をもつ複雑なシステムの表現であり，期待する効果が状態を変更する操作(代入)によって得られるというところにある．その一方で，関数プログラミングには基本的には**関数抽象**(abstraction)と**関数適用**(application)しか存在しない．すなわち，抽象化と具体化の表現しか必要としないのである．実用的な観点からは，数のような基本的なデータや(非破壊的な)リストなどのデータ構造を導入したり，計算対象の命名機能(宣言)を用意することが多いが，基本的な概念が少ないことはプログラミングをきわめて理解しやすいものにしている．

関数プログラミングに基本概念が少ないことは問題の仕様記述に関数プログラムを利用する面でも役に立つ．問題の仕様記述を行う際には，問題領域において常識と考えられることを前提として特定の問題を表現する必要があるので，記述に用いる言語には複雑な概念が含まれていないのが望ましいからである．関数プログラミングのこの特徴は仕様記述という側面でも注目されており，実践的なソフトウェア工学にも貢献しうるものといえよう．付録には，問題の直接的な表現から始めて関数プログラミングによってプログラムを得る例を示してある．手続き型プログラムと対比させて考えると，関数プログラムの構築にあたって考慮すべきことの少ないことがわかるであろう．

関数プログラミングそのものの特徴というわけではないが，**非正格**(non-strict)な意味をもつ言語によるプログラミングは，ほかのプログラミング方法論には見られない独特の方法論を提供する．次節で見るように，関数プログラム言語にはこのような意味をもつ言語が少なくない．これは，言語の意味論としての興味だけではなく，これを関数プログラムの上に実現する評価方式(**遅延評価**，lazy evaluation)によって，無限リストのような対象をも扱うことができ，あらたなプログラミングの視点を与えるものであるからである．付録の例でもこの機能を使っている．

3 関数プログラム言語とその実現法

関数プログラミングの基本的な概念は関数プログラム言語として反映されている．どの分野の言語についても同様であるが，概念の整理が未成熟な時期には文字通り数多くの言語が提案されるものである．関数プログラム言語については，現在はこれまでの研究成果のなかで広く合意の得られた提案を整理して，基準となるような言語が現われている時期であるといえよう．もちろん，関数プログラミングにあらたな概念が現われないというわけではなく，むしろ，これは今後の言語設計の出発点となるものと考えるべきである．

関数プログラミングの立場では，LISP の存在が健全な関数プログラミングの発展を阻害したという声さえ聞

かれる[23]．純LISP(pure LISP)といわれる部分については純粋な関数プログラム言語であるが，1960年頃のハードウェアの能力の制約から，実用を重視するあまりに不純な副作用を導入して手続き型言語の道を歩んだというわけである．このことは，高速で高容量のハードウェアが得られるようになって，純粋の関数プログラム言語が現実の問題を解くために利用できるようになったこととも符合している．

関数プログラム言語については解説記事[11]に詳しいので，ここで重ねて述べることはしないが，最近の言語に備わっている機能を簡単にまとめておこう．

すでに述べたように，**高階関数**(higher order function)を扱う機能(つまり，関数の引数として関数をとることができ，関数の結果として関数を返すことができること)はプログラミングの方法論の点からも重要なものであり，すべての関数プログラム言語に備わっている．しかし，関数の引数や結果としての関数を扱う機能を備えていても，つぎのような場合に現われる高階関数を認めない言語もある．関数はたんに既存の関数を値として返すだけではなく，関数適用の結果として生まれる関数を返すこともありうるのである．すなわち，引数を2個以上もつ関数は

$$f\ x_1\ x_2 \cdots x_m = \cdots$$

のように定義されるが，引数 $a_1, a_2, \cdots, a_k$ を与えて得られる値

$$f\ a_1\ a_2 \cdots a_k$$

は，$k<m$ のときには $(m-k)$ 個の引数をもつ関数となるからである．高階関数ではこのような**引数の部分適用**(partial application, partial parametrization)が可能であり，これを利用したプログラミングも関数プログラミングのひとつの特徴となっている[19]．最近の言語では，このような関数も許すものが多い．

最近の関数プログラム言語は計算対象の型(type)をもち，しかも，静的な型検査のできる**強い型ぎめ**(strong typing)を採用するものが多い．型の概念は理論的な考察の対象でもあるが，実用的な観点からも重視されていて，効率のよい処理系の実現にも利用されている．さらに，**多様型**(polymorphic type)，**抽象データ型**(abstract data type)，**モジュール構造**などの採用などの点でも言語設計に大きな進展が見られる．これらは，いずれもML[16]で有効性の確かめられたもので，最近の関数プログラム言語に対するMLの貢献には大きなものがあるといえよう．

すでに触れたことであるが，プログラミング方法論との関係では，言語の意味が**正格である**(strict)か**非正格である**かということが言語を二分することになる．正格な言語というのは，定義されない値 $\perp$ を引数に与えると $f\perp=\perp$ となる(すなわち，結果も定義されない)という意味をもつもののことである．一方，非正格な意味をもつものは，$f\perp$ の値が必ずしも $\perp$ ではないというものである．こうした非正格な意味を標準とする言語においては，正格な意味に基づく言語には見られない特徴がある．実用的な点では，非正格な意味を用いると，入力や出力を副作用と考えることなく実現できることが重要である．非正格な意味をもつ関数プログラム言語については，TurnerによるSASL, KRC, Mirandaの一連の言語設計の考え方が大きく影響を与えている．すでに述べたリストの内包表記はそのひとつである．最近の代表的な関数プログラム言語では，ML[16]が正格，Miranda[24], Haskell[12]が非正格である．

関数プログラムによる問題解決は関数プログラムの**評価**(evaluation)によって達成される．数学における関数は集合間の対応を与えているが，プログラムにおいてはそのような対応を，われわれの目に見える形にして示す必要がある．プログラムは目的とする値を表現するものであると考えられる．値を表現している式のどの部分も，(参照透明性によって)それと等価な表現に置き換えても意味(値)は変わらないので，関数プログラムをつぎつぎとより簡潔な形式に変換する(**簡約する**，reduce)ことによって，われわれに都合のよい表現にすればよいわけである．このことが関数プログラムの評価に相当する．

評価を行う機構を用意するとなると，時間の概念，状態の概念が表面に出てくる．あらたな計算機構を考えるにしても，評価を行うには簡約の段階を重ねることになるであろうから，状態遷移が伴うことになる．すなわち，プログラムの評価には，関数プログラミングには存在しない状態の概念が顔を出すことになる．そればかりでなく，関数プログラムの実現(implementation)においては，プログラム上には表現されない状態を操作する必要があるので，手続き型言語と機械語の関係以上の間隙を

埋めなくてはならなくなる．プログラムには副作用が含まれないが，その分だけ評価系は副作用の塊とも言えるものになるのである．

言語処理系の実現法については，正格な言語に対しては普通の手続き型言語の処理系の作成技法が応用できる部分が多い．しかし，非正格な言語については，Turner[20]によって**組合せ論理**(combinatorial logic)の**結合子**(combinator)を用いる考え方が提案されて以来，効率のよい実現法の研究が活発に行われてきた．この方面の研究成果は[18]にまとめられている．また，**抽象解釈**(abstract interpretation)を用いた種々の最適化の手法は，ともすれば発見的・経験的・場当り的になりがちな最適化処理を，数学的基盤が確立された信頼性のあるものにしている点で重要なものといえよう．抽象解釈の最適化への応用については[6]を参照されたい．

関数プログラムの評価法に関してはこれまでにも数多くの提案がなされているが，関数プログラムの実用性を議論するとなると，効率のよい処理系は欠くことのできないものである．この点で特筆すべきことのひとつは，Augustsson と Johnsson による Lazy ML のコンパイラ[7]であろう．ほとんどすべての部分をその関数プログラム言語で書き，実用的な効率のコードを生成するというものである．処理系はプログラムの性質を調べる上でも重要な役割を担うものでもあるので，関数プログラミングを進めるためには一層の研究と実験を進めることが必要である．

関数プログラムには参照透明性という望ましい性質があるので，プログラムのいくつかの部分を並行して評価することができるという特徴もある．手続き型などのほかの種類のプログラムでも並行処理が重要な研究課題になっているが，関数プログラムについても並行評価の方式について研究が進められている．手続き型言語や並列 LISP の研究における成果を利用をすることもできるであろうが，新たな方法論を提供している遅延評価と並行評価の関係を明確にして，効率のよい並行評価の処理系を構築する研究も進められている．また，関数プログラミングを支援するためのアーキテクチャについても古くから研究が行われているが，本稿ではこれについては述べる余裕がない．たとえば，[5]を参考にされたい．

4 おわりに

一般に，抽象的に述べてある数学的な法則を心得ると具体的な工学的な問題をその言葉で定式化することができるようになる．関数プログラミングは，まさにこれと同様に，数学と工学を結ぶものであるといえる．このときに必要なことは，初等的な数学の知識(等式の置き換えによる推論，数学的帰納法など)である．数学的な取扱いは(程度の差はあるであろうが)，問題の表面には現われていない不明確な部分を掘り起こし，信頼のできる定式化としての仕様記述を得る基礎となりうる．問題を記述している仕様からプログラムを導出する，あるいは仕様を満たすプログラムを構築する際に，法則として確立された事実を適用しつつ実用的なプログラムを得るという“**プログラムの計算術**(calculus)”が関数プログラミングの真髄であろう．

数年前から，[17]のように，実用的なプログラムに対して手続き型プログラミングとの比較もなされ，関数プログラミングの有効性の確認とその問題点の解明が行われてきている．こうしたなかで，最近では，関数プログラミングが実用的な問題解決にも利用されている．また，IFIP(情報処理国際連合)の作業グループ WG2.8 では関数プログラミングを主題として活動しており，言語 Haskell もそのひとつの成果である．おなじく，IFIP WG2.1 では**構成的アルゴリズム論**(constructive algorithmics)として，アルゴリズムの表現・変換などに関数プログラムを用いている．このように，関数プログラミングの分野の実用的研究・基礎的研究は非常に活発になってきている．プログラムの計算術を確立して，抽象された規則・法則をソフトウェア工学に活かし，実用的な関数プログラミングをめざすためには，関数プログラムの性質を利用したプログラミングの方法論を確立するとともに，関数プログラムの評価を支援する処理系の研究も重要であるといえよう．

参考文献

[1] 特集：関数型プログラミング，コンピュータソフトウェア，Vol. 4, No. 4(1987), pp. 3-55.
[2] 特集：関数型プログラミングとその応用，情報処理，Vol. 29, No. 8(1988), pp. 808-916.
[3] Special Issue : Lazy Functional Programming,

Comput. J., Vol. 32, No. 2(1989), pp. 97-186.
[4] *Proc. of ACM Symposium on LISP and Functional Programming*, ACM, 1982 年より隔年.
[5] *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, 1981(ACM), 1985 (Springer LNCS 201), 1987(Springer LNCS 274), 1989 (ACM).
[6] Abramsky, S. and Hankin, C. : *Abstract Interpretation of Declarative Languages*, Ellis Horwood, Chichester, 1987.
[7] Augustsson, L. and Johnsson, T. : The Chalmers Lazy-ML Compiler, *Comput. J.*, Vol. 32(1989), pp. 127-141.
[8] Bird, R. and Wadler, P. : *Introduction to Functional Programming*, Prentice-Hall, Englewood Cliffs, 1988.
(邦訳：関数プログラミング，近代科学社，1991)
[9] Church, A. : *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, 1941.
[10] Henderson, P. : *Functional Programming-Application and Implementation*, Prentice-Hall, Englewood Cliffs, 1980.
[11] Hudak, P. : Conception, Evolution, and Application of Functional Programming Languages, *ACM Computing Surveys*, Vol. 21(1989), pp. 359-411.
(邦訳：関数プログラム言語の概念・発展・応用，コンピュータ・サイエンス'89，共立出版，1991)
[12] Hudak, P. and Wadler, P. eds. : Report on the Programming Language Haskell, to appear in *ACM SIGPLAN Notices*.
[13] Hughes, J. : Why Functional Programming Matters, *Tech. Rep.* 16, Programming Methodology Group, Chalmers University of Technology, 1984. [3]にも掲載.
[14] Landin, P. J. : The Mechanical Evaluation of Expressions, *Comput. J.*, Vol. 6(1964), pp. 308-320.
[15] McCarthy, J. : Recursive Functions of Symbolic Expressions and Their Computation by Machine, *Commu. ACM*, Vol. 3(1960), pp. 184-195.
[16] Milner, R. : A Proposal for Standard ML, *Proc. 1984 ACM Conf. on LISP and Functional Programming*, ACM, 1984, pp. 184-197.
[17] Peyton Jones, S. L. : Yacc in Sasl : an Exercise in Functional Programming, *Softw. Pract. Exper.*, Vol. 15 (1985), pp. 807-820.
[18] Peyton Jones, S. L. : *The Implementation of Functional Programming Languages*, Prentice-Hall, Englewood Cliffs, 1987.
[19] Takeichi, M. : Partial Parametrization Eliminates Multiple Traversals of Data Structures, *Acta Inf.*, Vol. 24 (1987), pp. 57-77.
[20] Turner, D. A. : A New Implementation Technique for Applicative Languages, *Softw. Pract. Exper.*, Vol. 9(1979), pp. 31-49.
[21] Turner, D. A. : The Semantic Elegance of Applicative Languages, *Proc. of the 1981 Conf. on Functional Programming Languages and Computer Architecture*, ACM, 1981, pp. 85-92.
[22] Turner, D. A. : Recursion Equations as a Programming Language, *Functional Programming and Its Applications : An Advanced Course*, Cambridge University Press, 1982, pp. 1-28.
[23] Turner, D. A. : Functional Programs as Executable Specifications, *Mathematical Logic and Programming Languages*, C. A. R. Hoare et al. eds., Prentice-Hall, Englewood Cliffs, 1984, pp. 29-54.
[24] Turner, D. A. : Miranda-A Non-Strict Functional Language with Polymorphic Types, *Functional Programming Languages and Computer Architecture*, Springer-Verlag, LNCS 201, 1985, pp. 1-16.
[25] Wirth, N. : *Systematic Programming* : An Introduction, Prentice-Hall, Englewood Cliffs, 1973.
(邦訳：系統的プログラミング入門，近代科学社，1975)

付　録

ここでは，ひとつの関数プログラミングの例を示す．構造的プログラミングを進める際の段階的プログラム開発法を述べている教科書[25]の 15.3 節の例題を考えることとする．以下のプログラムの導出は[23]の流儀による．

例題：2 つの自然数の 3 乗の和で 2 通りに表わされる数(Ramanujan 数)のうちで最小のものを求めよ．すなわち，

$$x = a^3+b^3 = c^3+d^3$$

となる自然数で $a \neq c$, $a \neq d$ となる最小のものを求める問題である．

この問題の手続き型プログラミングによるひとつの解は，図 1 のようなものである．その構成は段階的プログラム開発法にしたがって，最終的には Pascal のプログラムとして示されている．問題は決して自明なものではなく，手続き型プログラミングにおいては配列を用い

```
var i, il, ih, min, a, b, k : integer;
  j, p, S : array [1..12] of integer;
begin i := 1; il := 1; ih := 2;
  j[1] := 1; p[1] := 1; S[1] := 2; j[2] := 1; p[2] := 8; S[2] := 9;
  repeat min := S[i]; a := i; b := j[i];
    if j[i] = i then il := il+1 else
      begin
        if j[i] = 1 then
          begin ih := ih+1; p[ih] := ih*ih*ih;
            j[ih] := 1; S[ih] := p[ih]+1
          end;
        j[i] := j[i]+1; S[i] := p[i]+p[j[i]]
      end
    i := il; k := i;
    while k < ih do
      begin k := k+1;
        if S[k] < S[i] then i := k
      end
  until S[i] = min;
  writeln(min, a, b, i, j[i])
end.
```

図 1　手続き型プログラム

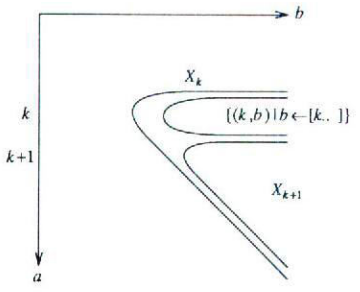

図2 列の分割

て複雑な処理をしている．手続き型プログラムの開発過程については[25]を参照されたい．ここでは，この問題を解く関数プログラムを求める過程を示すこととする．

まず，上の等式を満たす自然数の組の集合Rは，

R = {((a, b), (c, d))|
a^3+b^3 = c^3+d^3; a+c かつ a=d}

である．この問題では最小のRamanujan数を要求しているが，それには集合Rの元((a, b), (c, d))のうちで

sumcubes(a, b) = a^3+b^3

が最小のものを求めればよいので，部分問題として集合Rを求めることを考えよう．

集合Rの元を系統的に求めるには，自然数の対(a, b)がsumcubes(a, b)の値の小さい順に並んでいる列(リスト)sをもとにして得ることができる．すなわち，列sの隣接する2つの対(a, b)と(c, d)の対((a, b), (c, d))から成る列のうちでsumcubesの値が同じものだけをとればよい．それには，綴じあわせ関数

zip2 xs ys = [(x, y)|x<-xs; y<-ys]

によってzip2 s(tl s)として求めた列を用いる．ここで，関数tlは列の後部(tail)を表わすものである．こうすると，集合Rを表現する列ramanujanは

ramanujan = [(x, y)|(x, y)<-zip2 s(tl s);
sumcubes x = sumcubes y]

のように表わすことができる．

つぎの問題は，列sを求めることである．自然数の列k, k+1, …を

[k..]

のように表わすものとすると，

X = [(a, b)|a<-[1..]; b<-[a..]]

は対象とすべき自然数の対をすべて含んでいる．関数sumcubes(a, b)がaとbに関して対称的であるので，bの範囲をb≧aに制限しているのである．そこで，列sはこのXをsumcubesの値に関して大きさの順に並べたものであるということになる．ここで，列Xをaの値によって

X_k = [(a, b)|a<-[k..]; b<-[a..]]

のように表わすものとすると，

X = X_1

であることになる．関数fsort

fsort r X_k

が列X_kを関数rの値に関して大きさの順に並べた列を返すものとすると，

s = fsort sumcubes X_1

ということになる．

ここで，列X_kを図2に示すように2つの部分に分割すると，

X_k = [(k, b)|b<-[k..]] ++ X_{k+1}

である．ここで，++は列を連接する演算を表わすものである．列X_kをrの値に関して大きさの順に並べるには，右辺の2つの列のそれぞれを大きさの順に並べたものを併合すればよいから，関数fmerge

fmerge r (x:xs) (y:ys)
= x:fmerge r xs (y:ys), if r x≦r y
= y:fmerge r (x:xs) ys, otherwise

を用いて，

fsort r X_k =
fmerge r (fsort r [(k, b)|b<-[k..]])
(fsort r (X_{k+1}))

となる．関数fmergeに現われているa:asは列asの前にaを付加することを表わし，それが左辺の引数の位置に現われるときは，引数がその形をしていることを求めている．また，右辺のコンマ以降は条件を表わす通常の数学の表記法と同じである．

さて，関数fsortを定義する上の式のfmergeの第1引数は，関数rが単調であるときには

(k, k):(k, k+1):(k, k+2):…

であり，さらに，(k, k)はrに関してX_{k+1}のどの要素よりも小さいので

fsort r X_k =
(k, k):
fmerge r[(k, b)|b<-[k+1..]](fsort r X_{k+1})

となる．実際，sumcubesは単調であるのでこの定義が

```
|| Finding Ramanujan numbers

ramanujan :: [((num,num),(num,num))]
ramanujan = [(x,y)|(x,y)<-zip2 s (tl s); sumcubes x = sumcubes y]
            where s = fsort sumcubes 1

sumcubes :: (num,num)->num
sumcubes (a,b) = a^3+b^3

fsort :: ((num,num) -> num) -> num -> [(num,num)]
fsort r k = (k,k): fmerge r [(k,b)|b<-[k+1..]] (fsort r (k+1))

fmerge :: (* -> num) -> [*] -> [*] ->[*]
fmerge r (x:xs) (y:ys) = x: fmerge r xs (y:ys), if r x <= r y
                       = y: fmerge r (x:xs) ys, otherwise

-----------------------------------------------------------------
                The  Miranda  System

              version 2.009 last revised 14 November 1989

               Copyright Research Software Ltd, 1989


compiling ram.m
checking types in ram.m
for help type /help
Miranda hd ramanujan
((1,12),(9,10))
Miranda take 3 ramanujan
[((1,12),(9,10)),((2,16),(9,15)),((2,24),(18,20))]
Miranda /q
miranda logout
```

図3　関数プログラム

有効である．

ここであらためて fsort r X_k を fsort r k と書くこととすると，関数プログラムは完成である．プログラムを Miranda で書き，実行したものを図3に示してある．列の先頭要素(head)をとる関数

hd (x:xs) = x

を用いると，最小の Ramanujan 数を得ることができる．また，列の先頭から n 個の要素を求める関数 take n を用いると，原理的にはいくつでも必要なだけの Ramanujan 数を得ることができる．

このプログラミングでは，非正格な意味をもつ関数プログラム言語を想定したが，このような枠組では，値が必要になったときに必要なだけの評価が行われるものと考えればよい．したがって，プログラムのなかで無限リストを使ったとしても，それは表現のためであって，実際にはそのうちの有限個の要素しか必要とされないか，あるいは無限の要素を求めようとして永遠に処理が停止しないかのいずれかであるということになる．