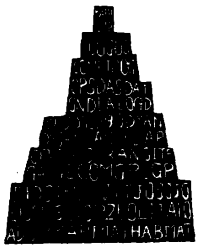

31. 関数型言語†

横井俊夫††

1. はじめに

関数型 (functional, applicative)[1),2)] というのは，プログラミング・スタイルの名前である．したがって，多くの応用分野に，このスタイルの言語を考えることができる．処理手順の表現という側面から見たプログラミング・スタイルは，次の3つに大別される．

（1） 逐次型：手順を処理要素（実行文）の系列として表現する．

（2） 関数型：手順を処理要素（式）間のデータの入出力依存性で表現する．

（3） 関係型：手順を処理要素（述語）間のデータの同値関係で表現する．

逐次型には，FORTRAN, PL/I, PASCAL 等，広く使われている言語のほとんどが含まれる．関数型には，Lisp, APL 等が入る．関係述語論理型は，PROLOG によって代表される．以下に，関数型の特徴を，特に，逐次型に比較してのものを列挙する．ただし，これらは，すべての関数型言語が等しく合せ持っているというわけではない．実際の言語は，様々な異物をかかえこんでいる．

（a） 高い透明度：計算の機構が簡明である．データの変換の過程が，素直に，計算の手順に対応する．データの流れと制御の流れが一致する．この簡明さは，プログラムの構造(文法)，さらに，基本データ構造（プログラムも同じ構造）の簡明さに反映される．

（b） 高い記述力：プログラム自体，さらに，部分的に実行されたプログラム（実行環境＋プログラム）を処理対象とする高階の機能は，高度な制御構造の記述を可能にし，逐次型の諸機能をその一部に含ませることができる．リスト構造をはじめとする動的に変化しうる基本データ構造は，高度な記号構造とその操作を，自然に表現することができる．

† Functional Languages by Toshio YOKOI (Machine Inference Section, Information Sciences Division, Electrotechnical Laboratory).

†† 電子技術総合研究所パターン情報部推論機構研究室

（c） 良いプログラミング環境：構造が簡単であるということは，効率の良いインタプリタ形式の処理系を実現でき，しかも，この処理系は動的に変化する要素が多いという性質にも適合し，それを基盤として，非常に使い勝手の良い会話型プログラミング・システムが作られている．プログラムの作成，検査，管理の容易さは，コンパイラ・ベースの逐次型言語のシステムでは，とうてい及ばぬものである．

（d） 大きいマシンの負担：プログラムの実行には，少なくとも，スタックが必要であり，動的に変化しうる基本データ構造の実現には，ヒープ（ゴミ集め機構を必要とする）の機構が必要である．したがって，その実行には，大きな記憶領域を必要とし，単純な処理で比較すれば，多くの処理時間を必要とする．

(a), (b), (c)は，大きな長所であり，(d)は，短所とみられる．特に，計算機が高価であった時代には，(d)は大きなマイナス点となり，逐次型言語に席を独占されることになる．そこで，その時代には，(a)と(b)の長所を利用して，プログラム言語の形式的な意味記述に用いられたり，高度な記号処理を必要とする人工知能研究の分野に多く用いられた．しかし，最近，計算機が低廉化し，それに比べ，プログラムの作成費が急増し，(d)を短所と見る根拠もなくなり，(a), (b), (c)の長所が，大きく注目されるようになった．計算機のパーソナル化傾向は，特に，(c)をクローズ・アップすることになる．この 10 年間の人工知能研究の再興隆は，多くの成果を生み出すとともに，人間の知的機能の解明とシステムとしての実現を，名実ともに，今後の情報処理研究の中心課題となすにいたった．それにともない，そこで果した関数型言語の役割の大きさと普遍さが広く認識されつつある．

ほかに，関数型言語をめぐる話題，特に今後のものとして，新しい計算機アーキテクチャ，新しいプログラム言語等があるが，これらに関しては，最後のまとめで触れることにする．次に，関数型言語の一般的な

骨格，仕組について説明し，最近の動向，および，今後の見通しを述べることにする．

2. 関数型言語の仕組

関数型に属し，広く使われている言語は，Lisp と APL くらいで，ほかは，研究・開発に着手されたばかりのものがほとんどである．そこで，個々の言語別の説明はやめ，関数型全体を貫く言語の仕組を，ラムダ計算から，順を追って説明する．まず，本稿に登場する言語の来歴を簡単に紹介しておく．Lisp：本特集号の該当稿を参照のこと．APL：同様．FP[3]：Backus の提案になるもので，検証理論等も含めた総合的な言語システムで，基本的な言語の仕組，仕様が発表されている．SCHEME[4]：Sussman 等の提案による．Lispをよりラムダ計算本来の姿にもどそうというもので，tail transfer と自由変数，関数名の lexical scope を特徴とする．基本部分の処理系が作られ，専用のマシン[5]が (V) LSI チップとして開発されている．Lucid[6]：Ashcroft 等の提案による．単一割当規則をループに適用できるように拡張し，検証を容易に行えるようにすることを目的とした言語．処理系の試作が行われている．VAL[7]：Ackerman 等の提案による．研究，開発中のデータ・フロー計算機用の言語．言語仕様が発表されている．μ-PLANNER, CONNIVER：本特集号の“人工知能用言語”を参照．PLASMA[8]：Hewitt の提案による．Actor 理論に基づく，パターン照合，並行処理等を含む言語である．

以下，順を追って説明するが，厳密さには，あまり留意しない．

ラムダ計算：関数型言語の源は，Church の提案になるラムダ計算[9]である．すべてがラムダ式 (lambda-expr) で表される．

⟨lambda-expr⟩::=⟨identifier⟩|⟨abstraction⟩|
⟨application⟩

⟨abstraction⟩::=λ(⟨variable⟩…)⟨lambda-expr⟩

⟨application⟩::=(⟨lambda-expr⟩⟨lambda-expr⟩…)

適用形 (application) の第一要素が，抽象形 (abstraction) の場合，その (ラムダ) 変数に，順に，第二要素以下のラムダ式を対応させ，それに基づき，抽象形のラムダ式内の置き換えを行う．この操作をリダクションと呼ぶ．このリダクションを，入れ子構造となったラムダ式の随所に適用することによって計算が進められる．いささか，はしょった説明であるが，もともとの Church の目的は，計算の理論を作ることである．これを具体的なマシンのメカニズムとして実現するためには，いくつか工夫が凝らされる．それが，次の計算機構である．

計算機構：大きく3つに分類される．

(1) 書き換え：ラムダ計算の定義に忠実に，実際にラムダ式の書き換えを行う．ラムダ式は，リスト構造や文字列等でマシン内に表される．入れ子構造になったラムダ式は，通常，実行可能なリダクションが多数存在するが，それらをどの順番に実行するかということが問題となる．内部のものから順に行うのを適用順 (applicative order) といい，call-by-value の計算に類似する．外部のものから順に行うのを正順 (normal order) といい，call-by-name に対応する．実際には，この両者が，必要に応じ，交えて用いられる．多くは，各適用形に対し，どの評価順を用いるかを明示させる．この機構は，ラムダ式の複製を多量に作るため，通常の計算機には適さず，専用のマシン（リダクション・マシン[10]）として研究が行われている．

(2) SECD マシン：Landin により，通常のプログラム言語をラムダ計算上にモデル化するために考案された機構である[11]．S (Stack), E (Environment), C (Control), D (Dump) の4つの内部状態を持つマシンである．通常の計算機上にも効率良く実現されるため，Lisp をはじめ，多くの関数型言語の実現機構は，これに類似する．ラムダ変数の対応付けの情報を環境 (E) 上に置き，変数に出会うと，それに対応付けされた値を見に行くということで，リダクションを模擬する．

(3) 駆動型：データ・フロー計算機構[12]に代表される並列計算を基盤とした機構である．適用形の各ラムダ変数への値がすべて揃うと，その適用形のリダクションが起動される．リダクションを，対応付けられた値を参照している所，すべてに送り出すというメカニズムで代行する．これにより，適用順の計算が並列に行われる．正順の計算に対しては，次のような機構を考えることができる．送り出すものを計算結果ではなく，結果を入れる容器にする．中身に関しては，並行して計算を進める．容器を受け取った側は，中身が必要になった時，はじめて取り出しに行き，まだ未到着の場合は，そこで到着を待つ．これは，いわゆる lazy—評価の機構[13]の，一般化，並列処理化と見ることができる*．

* そこで，筆者は，lazy (のろま) に対し，hasty (せっかち)—評価機構と呼んでいる．

構造の導入：ラムダ計算では，すべてがラムダ式で表される．自然数やリストのラムダ式による表現は，教科書でよく見掛ける巧妙な例である．できるだけ少ない構成要素で，すべてを表現するということは，理論的には大いに価値のあることであるが，実際のプログラム言語としては，よりまとまった記述要素の導入が必要となる．データ構造とそれを操作する基本関数，条件式や再帰呼び出しなどの，典型的な制御構造の導入が図られる．データ構造として，数値，文字列，リスト (Lisp)，対 (FP の列)，行列 (APL) 等が導入され，それらを表す記法と基本演算関数が設けられる．制御構造としては，まず，抽象形を名前で参照する仕組が導入され，

〈difinition〉::=〈name〉：〈abstraction〉

〈application〉::=(〈name〉〈expr〉……)

再帰呼び出しが自然に表現される．次に，条件分岐を一般的に表す条件式，

〈conditional-expr〉::=(Cond 〈pair〉……)

〈pair〉::=(〈condition〉〈action〉)

が導入される (Lisp, FP 等)．そして，データ構造のすべての要素，あるいはすべての部分構造にある関数を施し，ある値を求めるという繰返し制御が導入される (Lisp の MAP 関数，APL の operator 等)．

適用の類型化(記号化)：リダクションは，ラムダ変数を媒介して行われる．この変数の介在は，計算の意味をより明確に定めたいとすると，あまり好ましいものではない．そこで，ラムダ変数の参照のされ方，すなわち，抽象形のタイプを類型化し，タイプを表す記号の組合せですべての抽象形を表し，変数という概念を消し去ろうという試みが行われる．1つは，理論面からの試みで，コンビナトリ論理[14]である．まず，ラムダ式を一変数のラムダ式に変換する (Currying)．

λ(〈var〉……)〈expr〉

→　λ〈var〉.λ〈var〉.……〈expr〉

次のような抽象形のタイプ，すなわち，コンビネータを設ける．

〈combinator〉::= **I** | **S** | **K**……

$\mathbf{I}=\lambda x.\ x$

$\mathbf{S}=\lambda f.\ \lambda g.\ \lambda x.\ f\,x.\,(g\,x)$

$\mathbf{K}=\lambda x.\ \lambda y.\ x$

すべての抽象形を，このコンビネータの組合せで表す．

もう1つは，プログラム言語の立場から，より高い機能のコンビネータを豊富に用意し，プログラムを理解しやすく，形式的に取り扱いやすくしようという試みである．FP に代表される．基本データ構造の操作や要素に演算を施す（基本）関数と呼ばれる演算コンビネータが用意される．

〈function〉::=〈selector〉|〈reverse〉|〈add〉……

次に，関数やデータ構造（オブジェクト）から，新しい関数を作り出す高階のコンビネータ，関数形が用意される．

〈functional-form〉::=〈composition〉|〈construction〉|〈condition〉|〈apply-to-all〉……

〈composition〉::=〈function〉∘〈function〉

プログラム（抽象形）は，関数形の系列として表され，非常に簡明で，形式的な姿をとることになる．

授受機構の高度化：ラムダ変数を消し去るというのとは逆に，変数の機能，引数結合の機能を拡張，高度化し記述力を強化する試みがある．いずれにしろ，引数，値の授受機構が，要となるものであるという認識は共通である．

（1）　セグメント：$(+a_1a_2\cdots a_n)$ のような記述を可能にするため，不定個の引数を渡す仕組である．

$(\lambda((\mathrm{Seg}\ x))$〈expr〉 $e_1\ e_2\cdots e_n)$

xには，$e_1, e_2, \cdots, e_n$ の値のリストが渡される (Lisp 等)．

（2）　デフォールト：対応する実引数がある場合は，その値が，無い場合は，デフォールト値が渡される．たとえば，

$(\lambda(x_1x_2\,(\mathrm{Default}\ x_3\ e_4))\mathrm{expr}\ e_1\ e_2)$

x_3 には，e_4 の値が渡される．

（3）　続き関数：意味記述の立場から，ラムダ計算に逐次計算のメカニズムを付与するために考案された[15]．これは，計算値に名前を付け，参照するという逐次型の良さを取り込む仕組に利用できる．

$\lambda(x_1x_2\cdots x_n)$ 〈expr〉

に，もう1つ引数を追加して，

$\lambda(x_1x_2\cdots x_n\ c)\ c$(〈expr〉)

c に渡される関数 (関数引数) が，続き (continuation) と呼ばれるもので，もとの 〈expr〉 の計算結果を受け取る役目を果す．たとえば，

F_1: $\lambda(x_1x_2\ c)\ c$(〈expr〉)

$(F_1\ P_1\ P_2\ \lambda(y_1y_2)F_2)$

とすると，この適用形は，

$(y_1y_2):=F_1(P_1P_2)$;

F_2;

という逐次実行に対応する．ただし，この代入文は，

逐次型のものとは異なり，ラムダ変数への結合として，副作用無しに，名前付けのみを行う．これは，VAL 等の言語の考え方である単一代入規則の一般化である．続き関数の考え方は，SCHEME, PLASMA の中心機能となっている．

(4) パターン照合：ラムダ変数の系列が，パターンとなり，変数の結合は，パターン照合に一般化される．リダクションは，複数の適応形に対し，まとめて行われ，照合したもののどれかが選択される．

〈abstraction〉::=λ〈pattern〉〈expr〉

〈application〉::=((Lambdas 〈abstraction〉…)〈expr〉)

選択は，後もどり等で処理される．この仕組は，μ-PLANNER, CONNIVER 等の人工知能向言語に取り入れられ，色々なシステムの作成に利用された．しかし，言語自身の未熟さから，大きく成長するにはいたらなかった．その成果は，最近，関係（述語論理）型という新しいスタイルの言語に受け継がれ，大きな展開を見せている．

パッケージ化：抽象形をより具体的に見ると，2種類の抽象化を見いだすことができる．アクション（サブジェクト）指向とオブジェクト指向である．

〈abstraction〉::=〈action-oriented〉|〈object-oriented〉

〈action-oriented〉::=λ(〈ob〉〈param〉……)
(〈ac〉〈ob〉〈param〉……)

〈object-oriented〉::=λ(〈ac〉〈param〉……)
(〈ac〉〈ob〉〈param〉……)

アクション指向は，通常の関数の定義で，操作対象となるオブジェクトから見て，アクションを定める．オブジェクト指向は，施されるアクションから見て，オブジェクトを定める．このオブジェクト指向の抽象形に，アクション名のスコープを局所化するパッケージ化の機能をつけると，いわゆる抽象データ型となる．

並列処理：先のパターン照合の項で説明した適用形を，照合できたすべての抽象形を起動すると拡張すると，陽に並列処理を記述する仕組のもととなる．

〈application〉::=((Par-Lambdas 〈abstraction〉……)
〈expr〉)

この機能の議論も PLASMA から発する．並列処理を陽に表現する機能を，どのようにして関数型の枠組の中にとらえるかは，これからの話題である．

3. ま と め

最後に，関数型言語をめぐる今後の大きな展開を述べ，まとめとする．

新プログラム言語へ：プログラムの正当性検証の研究は，既存言語を対象に，理論化，システム化が進められたが，多くの成果は得られたものの，もう1つの展開に欠け，停滞期にあるといえる．その中で，仕様，検証，合成の立場から，プログラムの作成過程，プログラム言語を，そのおおもとから見直し，再検討しようという動きが起りつつある．そして，関数型の持つ高い透明度と記述力が高く評価され，最近の関係述語論理型言語と一体化した新しいプログラム言語が共通項として浮び上りつつある．

新計算機アーキテクチャへ：Lisp マシン，APL マシンが，パーソナル・コンピュータとして，すでに商用化の段階に入り，この傾向は，ますます大きくなる．逐次型言語を対象にした IBM 型の計算機を，徐々に侵食し，さらに，大きな新しい利用分野を開いていくものと思われる．さらに，データ・フロー・マシンに代表されるような，新しい並列計算を基盤とするマシンに結びつく．ただし，この研究分野の現状は，すべてが望ましいものというわけではない．あまりに性急なハードウェア化が目立つ．その中で，言語，計算機構を，おおもとから見直し，アーキテクチャを再検討しようという動きが起りつつある．今後の VLSI 技術の進展に重ね合わせると，大きな実現への可能性が熟成されつつある．

参 考 文 献

1) Burge, W. H.: Recursive Programming Techniques, Addison-Wesley (1975).
2) Henderson, P.: Functional Programming: Application and Implementation, Prentice-Hall (1980).
3) Backus, J.: Can Programming be liberated from the von Neumann style? A functional style and its algebra of programs, Commun. ACM, Vol. 21, No. 8, pp. 613-641 (1978).
4) Sussman, G. J. and Steele, G. L. Jr.: SCHEME: An Interpreter for Extended Lambda Calculus, AI Memo 349, MIT (1975).
5) Steele, G. L. Jr. and Sussman, G. J.: Design of a LISP-Based Microprocessor, Commun. ACM, Vol. 23, No. 11, pp. 628-645 (1980).
6) Aschcroft, E. A.: Lucid, a Nonprocedural Language with Iteration, Commun. ACM, Vol. 20, No. 7, pp. 519-526 (1977).
7) Ackerman, W. B. and Dennis, J. B.: VAL-A Value-Oriented Algorithmic Language: Preliminary Reference Manual, LCS/TR-218, MIT, (1979).

8) 米沢明憲：ACTOR 理論について，情報処理，Vol. 20, No. 7, pp. 580-589 (1979).
9) Church, A.: The Calculi of Lambda-conversion, Princeton Univ. Press (1941).
10) Berkling, K. J.: Reduction Language for Reduction Machine, Proc. of the 2nd Symposium on Computer Architecture, pp. 133-140 (1975).
11) Landin, P. J.: The Mechanical Evaluation of Expressions, Computer Journal, Vol. 6, No. 4, pp. 308-320 (1963).
12) 横井俊夫：計算機アーキテクチャの新たな展開ーデータ・フロー・マシンを目差して，電子工業月報，Vol. 22, No. 4, pp. 5-10 (1980).
13) Henderson, P. and Morris, J. Jr.: A Lazy Evaluator, Proc. 3rd ACM Symp. POPL, pp. 95-103 (1976).
14) Curry, H. B. et al.: Combinatory Logic, Vol. 1, Vol. 2, North-Holland (1958, 1972).
15) Reynolds, J. C.: Definitional Interpreters for Higher-Order Programming Languages, Proc. ACM Annual Conference, pp. 717-740 (1972).

(昭和56年4月15日受付)