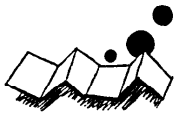
解　説

非手続き型言語におけるデータ構造†



片　山　卓　也††

1. はじめに

Pascal や Fortran, C などの通常の手続き型プログラム言語では，ステートメントの逐次実行と代入文による状態変化という計算原理により計算過程が記述されるが，この原理に基づかないいくつかのプログラム言語や，計算モデルが研究・提案され，また実用に供されている．これらの新しいプログラム言語は，非手続き型と呼ばれることが多い．実際にはこれらの言語がすべて純粋な意味で手続き的ではないというわけではないが，従来の手続き型の計算原理とは異なる原理のもので計算が進められることは事実であり，本稿でも，これらの言語を非手続き型と総称することにする．単に提案や研究をされただけのものも考えれば，数多くの非手続き型言語があるものと考えられるが，(1)関数型プログラム言語，(2)論理型プログラム言語，(3)オブジェクト指向プログラム言語などが代表的なものであろう．本稿の目的はこれらの言語におけるデータ構造，その特徴や問題点などについて，平易な解説を試みることである．これらの言語についての深い予備知識を仮定せずに話を進める．

つきつめて考えると，これらの非手続き型言語におけるデータ構造がこれまでの手続き型言語のデータ構造と基本的な点で大きく異なっているとは考えにくい．それにもかかわらず，これらの言語ではデータ構造の扱いやデータそのものに対する態度が大きく違っており，それがこれらの言語を特徴づけるひとつのポイントになっている．関数型や論理型言語については，それは，原則的には，履歴依存のない形でのデータの処理であり，またオブジェクト指向言語ではオブジェクトと呼ばれるデータ中心のプログラム記述であろう．また，もっと表層的な面では，非手続き型言語が，計算機ハードウェアが安価になった時代に登場した比較的新しい言語であるだけに，いわゆる高レベルなデータ構造の処理を行うようになっているということができる．

以下では，代表的な非手続き型言語のデータ構造について述べ，次にそれらに関するいくつかの話題について述べることにする．なお，本稿では，各言語のデータ構造についてのカタログを作るという立場はとらずに，主に基本的な考え方を中心にして述べる．

2. 関数型言語におけるデータ構造

2.1 関数型言語の原理

関数型プログラミングとは，プログラムをその入出力関係を実現する関数として考え，その関数の記述を行うことによりプログラミングを行おうとするものである．

プログラム＝関数の定義

たとえば，次のプログラムは，与えられた整数 x の階乗 $x!$ を計算するものである．

```
define factorial (x)
    =if x=0 then 1
            else x*factorial (x-1)
```

関数の表記法には種々のものがあるが，本稿では常識的な表記法を用いる．関数の記述（定義）の方法の違いやスタイルの違いにより種々の関数型言語が提案されている[1)-3)]．そのうちで最も古くから使われ，主に人工知能や記号処理の分野での主要言語となっているものに Lisp がある[4)]．純粋な意味での関数型言語では，関数はデータ値からデータ値を算出するものと考え，関数計算の実行にともなう状態の変化（副作用と呼ばれる）を基本的には排除している．このような純粋な関数計算のメカニズムは理解性や検証性の上からは大変に望ましいが，その効率的実行は必ずしも容易ではなく，Lisp では形こそ関数型であるが，実際には手続き型の基本原理である代入文や文の逐次実行という機構も併せて取り入れている．Lisp が Fortran と同程度の歴史をもつ言語であることを考えると，こ

† Data Structures of Nonprocedual Languages (Department of Computer Science, Tokyo Institute of Technology).

†† 東京工業大学工学部

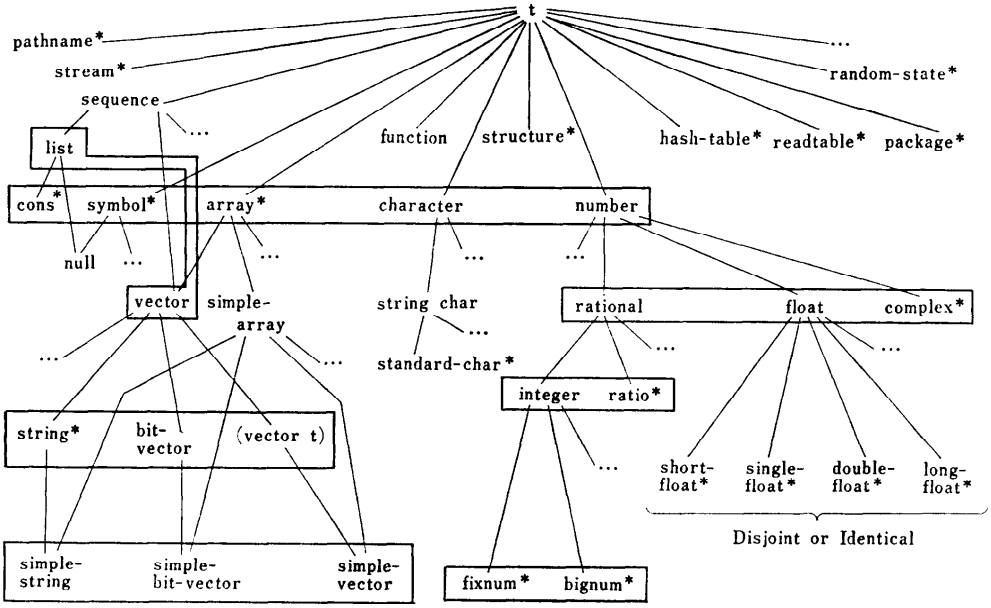


図-1 Common Lisp のデータ構造[5]

れはむしろ当然であるといってもよいかも知れない．しかしながら，最近の関数型プログラムへの関心の高まりは，計算機ハードウェアの低廉化と高機能高信頼性ソフトウェアに対する要求の高まりから，純関数型プログラミングの長所を見直そうとするものであると見ることができよう．

2.2 関数型言語のデータ構造

関数型言語にどのようなデータ構造が備わっているかは，もちろん個々の言語によって異なり，きわめて多数のデータ構造をもっているものから，単に原始データ構造とリスト構造しかないものまでさまざまである．たとえば最近の話題である Common Lisp では，それが各種方言の統一化をめざすということもあって非常に多くのデータ型を持っている[5]（図-1 参照）．その一方で，たとえば，Backus の FP には構造のあるデータとしてはリスト構造しかない．これらの違いは，その言語の目的によるものであるが，関数型言語のデータ構造の特徴としては，リスト構造と関数型データ，再帰的データ構造をあげることができるであろう．以下ではこれらについて簡単に述べてみたい．

2.2.1 リスト構造

リスト構造とは，有限個のデータの並びである．x_1, x_2, ..., x_n を要素とするリスト構造を次のように表すことにする*．

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

要素の個数が0のリストは［ ］と表される．リスト $[x_1, x_2, \ldots, x_n]$ の要素は再びリスト構造でもよい．次のものはリスト構造である．

$[a, b, c, d]$

$[[x, 10], [y, 15], [z, 20]]$

$[x, [y, z, [u, v]], w]$

したがって，究極的にリストの要素となる原始データの集合を X とすると，このデータから構成されるリスト構造は次のように帰納的に定義されることになる．

（1） ［ ］はリスト構造である．

（2） $x_1, \ldots, x_n$ が X の要素であるかまたはリスト構造であるならば，$[x_1, \ldots, x_n]$ はリスト構造である．

リスト構造はいわゆる多分木を表現しているが，通常 Lisp ではドット対と呼ばれる2分木を基本にしており，リスト構造 $[x_1, x_2, \ldots, x_n]$ は次のドット対の入れ子構造の別記法と考えられる．

$(x_1.(x_2.(\ldots(x_n.\mathrm{nil})\ldots)))$

ここで (x, y) は x と y から構成されるドット対であり，nil は空リストを表すための特別な原始記号である．

---

* この表記法が必ずしも一般的というわけではない．たとえば Lisp では
$(x_1\ x_2\ \ldots\ x_n)$
のように表され，また，FP では次のように表される[1]．
$\langle x_1\ x_2\ \ldots\ x_n \rangle$

リスト構造に対する演算としては次に定義される3つの関数 head, tail, cons が基本的である.

head $([x_1, x_2, ..., x_n])=x_1$

tail $([x_1, x_2, ..., x_n])=[x_2, ..., x_n]$

cons $(x_1, [x_2, ..., x_n])=[x_1, ..., x_n]$

これらの基本関数を用いると，たとえば，2つのリスト構造 u と v を接続するための関数 append (u, v) は次のように書くことができる.

```
define append (u, v)
  =if u=[ ] then v
    else cons (head (u), append (tail (u), v))
```

また，リスト u の要素を逆順に並べたリストを構成する関係 reverse (u) は

```
define reverse (u)
  =if u=[ ] then [ ]
    else append (reverse (tail (u)), [head (u)])
```

あるいは

```
define reverse (u)=rev (u, [ ])
define rev (u, v)
  =if u=[ ] then v
    else rev (tail (u), cons (head (u), v))
```

のように定義することができる.

2.2.2 関数型データ

リスト構造とならんで多くの関数型言語がもっている非原始データに関数型データがある．これは，関数の入力データや出力データ値として関数を許すというものであり，いわゆる高階の関数を定義することである．次の関数 map は高階関数の例である.

```
define map (f, x)
  =if x=[ ] then [ ]
    elss cons (f(head (x)), map (f, tail (x)))
```

map (f, x) の値は，リスト x の各要素に関数 f を作用させたリストを表している．たとえば，

```
define add 1(x)=x+1
```

とすると，

```
map ([1, 2, 3], add 1)=[2, 3, 4]
```

となる．高階関数は簡潔で美しい関数型プログラムを作成するには有用な道具である．高階関数の取扱いには λ 表記法と呼ばれるものが便利である．これは，関数本体にその仮引数を付加して関数を表現するものであり，たとえば，関数 add 1 は次のように表される.

```
define add 1=λx(x+1)
```

この式の意味は，仮引数が x であり，入力 x に対する値が $x+1$ であるような関数として add 1 が定義されることを表している．λ 記法は A. Church によって関数の基本的性質を研究するために導入された λ 算法と呼ばれる形式的体系のなかで用いられた表記法である[6]．λ 算法は関数評価メカニズム，パラメータ結合方式など関数型プログラムにとっても重要な概念を含んでおり，さらに端的にいえば，関数型プログラムの理論的基礎を与えているということができる．多くの関数型言語が λ 算法を模して設計されている．λ 算法では，データとそれに作用する関数の区別はなく，すべてが関数として扱われており，したがって，そこでは関数は本来高階である．((型のない) λ 算法のなかでは，$f(f)$ のような自己適用可能な関数 f も特別な配慮なしに扱われている)．λ 算法の本質は，λ 式と呼ばれる関数表現の変換規則であるが，その中心は次の形の β 変換と呼ばれるものである.

$$(\lambda x M)N \Rightarrow [N/x]M$$

右辺の $[N/x]M$ は M に現われる変数 x の自由な出現をすべて N によって置き換えた式を表している．$(\lambda x M)N$ はその本体が M によって与えられる関数 $\lambda x M$ に引数 N を与えて評価した値を表しているが，上の β 変換規則はこれが記号の置き換えによって実現されることを表している．この変換規則は，自己適用も含めた高階関数の評価が記号の置き換えという概念的には単純な操作によって実現されることを表している．高階といってもなんら神秘的なことがあるわけではない．もちろん，これは記号の置き換えという，それだけで考えれば単純な操作が大きな能力を秘めているというべきであるかもしれない.

2.2.3 再帰的データ構造

関数型プログラムでは，再帰的関数定義によって計算が記述されるのが普通であり，そこでは当然再帰的データ構造が扱われることになる．先に述べたリスト構造も再帰的に定義されていたことを思い出してほしい．したがって，型定義のある関数型言語では，再帰的データ構造を定義する機構があると，理解しやすいプログラムが構成できることになる．たとえば，整数値を端節点にもつような2分木 tree は次のように定義することができる.

```
type tree=leaf : nil+
          nonleaf : (val : integer, left : tree,
                     right : tree)
```

ここで，"+" は直和型を，また "(...)" は直積型を表す．leaf, nonleaf は直和成分識別のためのタグであり，val, left, right は直積成分識別のための成分名

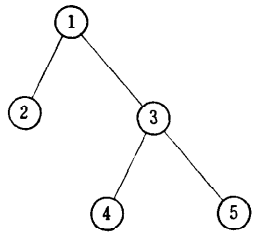


図-2 木構造の例

である．たとえば，図-2 のような木は次のように表される．

```
nonleaf : (1, nonleaf : (2, leaf : nil, leaf : nil)
              nonleaf : (3,
                         nonleaf : (4,
                                    leaf : nil,
                                    leaf : nil)
                         nonleaf : (5,
                                    leaf : nil,
                                    leaf : nil)))
```

もちろんこのようなデータ型定義がなくても，たとえば次のような関数 istree を定義すれば同様のことが可能であるが，静的な型検査による誤りの発見の観点からは再帰的な型定義の方が一般には望ましいと考えられる．

```
define istree (x)
  =if x=[ ] then true
    else if isinteger (head (x)) and
            istree (head (tail (x))) and
            istree (head (tail (tail (x)))) and
            tail (tail (tail (x)))=[ ]
         then true
         else false
```

3. 論理型言語におけるデータ構造

3.1 論理型言語の原理

広い意味では，論理型プログラミングとは形式論理を用いて計算を記述することであると考えられ，用いられる形式論理も一階述語論理，高階述語論理，時間論理あるいは様相論理など種々のものがありうる．しかしながら，通常論理型プログラミングというときには，一階述語論理あるいはその部分系である Horn 論理，さらにそれに制御構造を加えたプログラム言語 Prolog によるプログラミングをさすことが多い[7),8)]．このような立場からは，プログラムとはその入力と出力との間の関係，あるいはその関係を表現する述語と考えられる．

プログラム＝述語の定義

そしてプログラミングとはそのような述語の定義記述を行うことであるということになる．これはその論理体系のなかで，その述語を定義するための公理を与えることである．たとえば，自然数 x の階乗 $x!$ を計算するためのプログラムは x と $y=x!$ との間の述語 factorial (x, y) に対する公理

(1) factorial $(0, 1)$

(2) factorial (u, v)←sub $(u, 1, x)$,
factorial (x, z),
times (u, z, v)

であると考える．ここで，(2)において“←”は含意記号であり，A←B は“B ならば A である”を表す，また，“,”は連言 (and) を表し，現れるすべての変数は全称的に限定されているとする．sub (x, y, z), times (x, y, z) はそれぞれ $x-y=z$, $x*y=z$ を表す述語であり，他のところでその定義が与えられているとする．

このような形で factorial の定義が与えられたとき，たとえば，$y=3!$ の計算は，factorial $(3, y)$ を満たす y が存在することを表す命題

$\exists y$ [factorial $(3, y)$]

に対する証明として考えることができ，(もし 3! の値が定義されれば) 証明の過程で定まる y の値が 3! の値ということになる．証明のひとつのやり方は，$\exists y$ [factorial $(3, y)$]の否定

$\forall y$[～factorial $(3, y)$]

あるいは，全称記号を取り，～A を ←A と表現した

(3) ←factorial $(3, y)$

を作り，(1), (2), (3)から矛盾を導く方法がある．一般に考えている命題がすべて(1), (2), (3)のように

A←$B_1, B_2, \ldots, B_n$ $n \geqq 0$
←B

の形をしており，各 A, B, B_i が否定のつかない素命題 (リテラルという) のとき，それらは Horn 節と呼ばれるが，Horn 節によって与えられる述語の定義にもとづく計算は，上のように証明と考えること以外に，手続きの実行あるいは項の書き換えという見方をすることができる．Horn 節にもとづく論理型言語 Prolog は，節の順序づけおよび節のなかでのリテラルの順序づけ，カット命題の導入などにより Horn 論

理に実行制御を付加した非決定性のプログラム言語である.

3.2 論理型言語のデータ構造

3.2.1 関係の記述

論理型言語の著しい特徴は,それが関係を定義しようとしていることであり,データ構造という立場から見た場合,データ構造とその処理の記述が同一の形式のなかで行われることである.たとえば,先の2分木 tree に対応する述語 tree は

```
tree (nil)
tree (nonleaf (n, t1, t2))
    ←integer (n), tree (t1), tree (t2)
```

によって記述することができ,また,このような2分木の中の integer の値の和を求めるプログラムは次の述語 sum によって記述される.

```
sum (nil, 0)
sum (nonleaf (n, t1, t2), v)
    ←sum (t1, v1), sum (t2, v2),
      add 3 (n, v1, v2, v)
```

ここで add 3 (x, y, z, w) は $w=x+y+z$ を表す述語である.このようにデータ構造の記述とその処理の記述が同一の枠組のなかで一様に記述できるのが論理型言語の特徴である.このような記法はもちろん論関数型言語の枠組でも可能であったが,論理型言語における方がより直接的な記述が可能である.もちろん,関数型言語の場合にそうであったように,データ型定義とその処理の記述が異なった方が見やすいという立場もあり,一様的であることが良いかどうかは一概には決められない.

3.2.2 項による構造体の記述

論理型言語のデータ構造的側面としては,それが関係というデータ構造の定義を通してアルゴリズムの記述を行うということの他に,もちろん,その論理型言語に組み込まれているデータ構造がある.このようなデータ構造は各論理型言語ごとに変わるであろうが,標準的には,

(1) 数,記号などの原始データ構造
(2) 項によって表現される木構造
(3) リスト構造

がある.(3)のリスト構造は本質的には(2)の一部であるが,表面的には別のものと考える方が適当であろう.

一般に節

$A \leftarrow A_1, A_2, \ldots, A_n$

の中のリテラル A, A_i は

述語記号 (項, 項, ⋯, 項)

の形をしている.項は定数記号,変数記号,関数記号から次のように帰納的に定義されるものである.

(1) 定数記号や変数記号は項である.
(2) $t_1, \ldots, t_n$ が項, f が関数記号なら

$$f(t_1, \ldots, t_n)$$

は項である.

Prolog では,定数記号は数記号およびアトムと呼ばれる文字列であり,関数記号もアトムである.たとえば,次のものは項である.

```
man (age(20), name (john))
plus (x, plus (y, z))
line (point (x1, y1), point (x2, y2))
```

これは Pascal などのプログラム言語におけるレコード構造やいわゆる構造体に相当するものである. man や age, name などの関数記号は伝統的にこのように呼ばれているが,単なるアトムであり,実際には関数を表しているわけではない.項は関数記号という識別名のついた構造体を表している.

リスト構造も原理的には項として表現されるが,通常は次のような表記法が用いられる.

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

これは,ドット対構成用の関数記号 "." と空リスト用アトム [ ] を用いた項

$.(x_1, .(x_2, .(\ldots .(x_n, [\ ])\ldots)))$

を表している.

一般に構造をもったデータの処理には,成分から全体を構成するための構成子と,全体から成分を取り出す選択子が必要になる.たとえばリスト構造の例では,[ ], cons が構成子であり, head, tail が選択子である.通常の言語ではこれらの構成子と選択子を用いてデータ構造の処理が記述されるが, Prolog などの論理型言語では変数を含んだ項やリスト構造の記述を許すことにより,構成子のみを用いたデータ構造の処理の記述が行われている.2つのリストを結合する関数 append を思い出してほしい.これに対応する述語 append は次のように書かれる.

```
append ([ ], y, y)
append ([a|x], y, [a|z])←append (x, y, z)
```

append (x, y, z) は x と y の連結が z であることを表し, a, x, y, z は変数であり, $[a|x]$ は cons (a, x) を表す.変数を含んだ項の処理は統一化(ユニフィケーション)と呼ばれるパターン・マッチング操作に

より実現される．このような記述はいくつかの関数型言語や等式言語でも行われている．

4. オブジェクト指向言語におけるデータ構造

4.1 オブジェクト指向言語の原理

4.1.1 オブジェクト

オブジェクト指向言語では，記述すべき問題中に存在する“もの”を抽象化したオブジェクトと呼ばれるものを中心にしてプログラミングが行われる[9)~11)]．オブジェクトとは，データとそれに作用する手続き群をひとまとめにし，その内部構造がオブジェクトの外側からは見えないようにしたものである（カプセル化という）．オブジェクトの外側から見た場合，オブジェクトはそれに含まれる変数の値によって決まる状態をもち，その状態を操作するための手続き（メソッドという）群の名前のみによって外側世界とのインタフェースがとられている抽象的な存在である．

オブジェクト＝内部状態＋メソッド

各々のオブジェクトは独立物であり，それらがメッセージの交換を行いながらプログラムの実行が進行するのがオブジェクト指向言語の基本原理である．たとえば，(1+3)*2 の計算はオブジェクト指向言語では次のような過程を通して行われる．

（1） 1というオブジェクトにメッセージ +3が送られる．

（2） このメッセージに対する結果として1は4というオブジェクトを返す．

（3） この4というオブジェクトにメッセージ *2を送り，その結果としてオブジェクト8が返される．

メッセージ+3はメソッド名“+”と“+”の引数を表すオブジェクト3から構成されている．このメッセージを受け取るオブジェクト1の記述には，一般に，+nというメッセージを受けたときに，n+1 に対応するオブジェクトを返すという動作が含まれている．一般にメッセージの形は

メソッド名 パラメータ1 パラメータ2...

の形をしている．各メソッドの記述では，そのメソッド名をもつメッセージを受け取ったときに，どのような動作（他のオブジェクトにメッセージを送ったり，あるいは内部状態を変更するなどを含めて）をするかが指定されている．

手続き型言語の世界では，手続きやサブルーチンなどの概念が複雑なプログラムを構成する上で有用な概念として用いられてきた．これは制御構造に関するカプセル化あるいは抽象化ということができる．データ構造に関する抽象化の概念としては，抽象データ型がある．これはそのデータ型に属する値の生成，操作などを行う手続きや関数群によってそのデータ型を表現し，その実現的詳細を外部から隠蔽したものである．その意味でこれはオブジェクトの概念と非常に似た概念である．抽象データ型とオブジェクトの概念のちがいは，抽象データ型があくまでデータ型であり，通常は手続き型や関数型言語のデータ的側面の構成要素であり，抽象化されていないデータ構造の場合と同様に手続きや関数によって操作される受動的対象であるのに反し，オブジェクトは自分自身の中に状態をとり込み，それ自身で能動的存在であることである．

4.1.2 クラスと階層関係

オブジェクト指向型プログラムの実行過程はオブジェクトの自律的動作によって表現されるから，オブジェクト指向言語におけるプログラムとはオブジェクトの動作記述であり，これはその状態とメソッドの記述ということになる．しかしながら，一般に，オブジェクト指向原理にもとづく問題の記述では同じ性質をもったオブジェクトが複数個あらわれることが多く，このときには個々のオブジェクトに対して同一の記述を行うことは面白くない．したがって，オブジェクトに対してそのひな形を用意し，そのひな形で各々のオブジェクトの動作記述を行い，各々のオブジェクトはそのひな形から動的に生成されるという立場をとる．このようなひな形のことをクラスという．したがって，オブジェクト指向言語におけるプログラムとは，クラスの定義によって与えられることになる．

プログラム＝クラスの定義

クラスの定義は，基本的には状態を表す変数の定義とメソッドの定義からなる．

クラスは共通の性質をもったオブジェクトのひな形として導入されたが，これは一般的にはある種の概念を表している．多くの場合，概念にはその上位の概念や下位の概念を考えることができ，これらに対応して上位のクラスや下位のクラスといったクラス間の階層関係を導入すると理解しやすいプログラムを構成することができる．一般に下位のクラスは上位のクラスに条件や性質を付加したものであり，具体的には，上位クラスに変数やメソッドを追加したものとして下位クラスが定義される．たとえば，ある会社に製品開発部と販売部があったとき，それぞれの部のメンバからなるクラスを定義することを考えよう．各部のメンバは

その会社の社員であるから，たとえば社員番号や氏名，給与といった属性が対応するが，このほかにも，製品開発部のメンバには現在担当中のプロジェクトや保有する特許といったものが，また販売部のメンバには売上げ目標や客先リストなどの属性が付随するであろう．このようなときには，まず，社員番号，氏名，給与などの変数およびそれらに関するメソッドをもつクラスとして社員を定義し，その下位クラスとして開発部メンバと販売部メンバを定義する．開発部メンバのクラス定義では，社員クラスが上位クラスであることを明示し，さらにそれのみに特有な変数とメソッドを定義する．オブジェクトの生成などに関する記述を除くと，クラス「社員」「開発部メンバ」の記述は，たとえば，次のようになる．

```
クラス名　　社員
上位クラス名　システム
変数　　社員番号，氏名，給与，…
メソッド　社員番号？　↑社員番号
　　氏名？　　↑氏名
　　昇給n　給与←給与＋n
　　⋮
クラス名　　開発部メンバ
上位クラス　社員
変数
　担当プロジェクト，保有特許…，
メソッド
　プロジェクト？
　　↑担当プロジェクト
　プロジェクト設定p
　　担当プロジェクト←p
　　⋮
```

なお，「システム」は最上位のクラスを表す（Smalltalk 80 では Object と書かれる）．メソッドの記述においてその左側は受理可能なメッセージ式の形を表し，その右側はそのメッセージに対するこのオブジェクトの動作を表す．たとえば，↑氏名は変数「氏名」に束縛されているオブジェクトをこのメッセージに対する答えとして返すことを表す．下位のクラスに属するオブジェクトは，そのすべての上位クラスの変数やメソッドを継承する．すなわち，この例では，開発メンバに属するオブジェクトは，クラス「開発部メンバ」で陽に指定された変数やメソッドの他にクラス「社員」で定義されているものをすべて保有することになる．これにより記述の経済性のみならず，クラス階層を上手に定義すると実世界に存在する概念階層を自然に表現

```
Object
  Magnitude
    Character
    Date
    Time
    Number
      Float
      Fraction
      Integer
        LargeNegativeInteger
        LargePositiveInteger
        SmallInteger
    LookupKey
      Association
  Link
    Process
  Collection
    SequenceableCollection
      LinkedList
        Semaphore
      ArrayedCollection
        Array
        Bitmap
          DisplayBitmap
        RunArray
        String
          Symbol
        Text
        ByteArray
      Interval
      OrderedCollection
        SortedCollection
    Bag
    MappedCollection
    Set
      Dictionary
        IdentityDictionary
  Stream
    PositionableStream
      ReadStream
      WriteStream
        ReadWriteStream
          ExternalStream
            FileStrem
    Random
  File
  FileDirectory
  FilePage
  UndefinedObject
  Boolean
    False
    True
  ProcessorScheduler
  Delay
  SharedQueue
  Behavior
    ClassDescription
      Class
      MetaClass
  Point
  Rectangle
  BitBlt
    CharacterScanner
    Pen
  DisplayObject
    DisplayMedium
      Form
        Cursor
        DisplayScreen
    InfiniteForm
    OpaqueForm
    Path
      Arc
        Circle
      Curve
      Line
      LinearFit
      Spline
```

図-3　Smalltalk 80 のクラス階層

でき，理解しやすいプログラムが得られる．

4.2 オブジェクト指向言語のデータ構造

これまで述べてきたように，オブジェクト指向言語ではデータとそれに対する処理手続きをひとまとめにしたオブジェクトと呼ばれる抽象的データ構造を中心にしてデータ処理の記述が行われる．一般に，プログラム言語によるアルゴリズムの記述は，データと手続き（あるいは関数・述語）の記述から構成されるが，通常の手続き型言語では，手続きがどのようにデータを処理するかという形で手続き中心の記述が行われる．これに反しオブジェクト指向言語では，データがどのような処理を受けるかという観点からデータ中心の記述が行われることになる．この点からすれば，オブジェクト指向言語では，広い意味でのデータ構造の記述によってプログラムの記述を行っていることになる．

一般にプログラム言語におけるデータ構造は，(1)それにどのような原始データ構造が含まれ，(2)どのようなデータ構造化機能（単純な構造から複雑な構造を構成する機能）があるか，という観点から理解するのが適当である．(1)については，オブジェクト指向特有の色づけはあるにしても，すなわち，このような原始データといえどもオブジェクトであり，メッセージの授受機能をもつ能動的存在ではあるが，基本的には通常の手続き型言語におけるそれと同等なものであると考えることができる．

オブジェクト指向言語における最も基本的な構造化の手法は，クラス定義における変数の宣言である．これは，そのクラスに属するオブジェクトのデータに関する内部構造を定義するものであるが，逆にいえば，そのオブジェクトがそれらの変数に束縛されるオブジェクトから構成されることを表している．これは通常の手続き型言語におけるレコード構造あるいは構造体に相当するものである．一般に変数には任意のオブジェクトを束縛できることを考えると，任意のポインタ構造をこれを用いて実現できることになりきわめて複雑な構造の構築が可能になる．これ以外の構造化の手法については個々のオブジェクト指向言語により異なるが，たとえば Smalltalk 80 では図-3 で示されるような組み込みクラスが用意されている[9]．それぞれのクラスがどのようなものであるかを説明する余裕はないが，その名前から想像していただきたい．この中には手続き型言語では基本的データ構造とは考えにくいものも多く含まれているが，これは，オブジェクト指向言語におけるクラスが単なる静的データ構造ではなく，機能単位にまとめられた抽象データ型であるためである．また，クラスはクラス階層とともに定義されるが，これはデータ構造の定義がそのような階層のなかで行われることを意味する．これにより，概念の上下関係にもとづくデータ型の定義が可能になるが，これは通常の手続き型言語にはない機能である．

5. おわりに

代表的な非手続き型言語である関数型言語，論理型言語およびオブジェクト指向言語をそのデータ構造の立場からながめてきた．これらの言語は従来の手続き型言語にない優れた特徴をもち，将来が楽しみな言語であるが，そのような特徴はデータ構造の立場からもうかがい知ることができる．関数型や論理型言語では，基本的には値を中心とした履歴性のない明快なプログラムを可能とすると同時に，高階関数の使用や関係（述語）定義による新しいプログラミングスタイルを提供している．またオブジェクト指向言語ではオブジェクトというデータ中心のプログラムの記述が行われる．

このような優れた特徴をもっている反面，いくつかの問題点も指摘されている．たとえば，これらの言語の多くは型なしである．すなわち，そこに現われる変数や関数のデータ型がプログラム中に明示されておらず，実行時になってはじめて定まるものが多い．これは柔軟なプログラムを作るうえからは有利であるが，データ型を利用した静的な誤りの検査のうえからは問題がある．この問題を解決するひとつの方法は，プログラムテキストからデータ型を自動的に作り出すことである．これは型推論と呼ばれ，多くの研究が行われており，また実際に型推論機構を組み込んだ言語もある[3),12)~14)]．

関数型言語や論理型言語では状態という概念がなく，値だけの世界で明確なプログラムが作成可能であることが大きな特徴であるが，これは効率的な実行という面からは問題になる．たとえば，サイズの大きなデータが少しずつ処理を受けながら関数や述語間を渡されていくような場合には，そのようなデータ値の授受のための時間的，領域的オーバヘッドが大きな負担になる．この問題を解決するひとつの方法は部分構造の共有化を上手に行い，処理の前後で値の変化しない部分構造値を共有化しようとする方法である．これはリスト構造のような柔軟なデータ構造には適用可能で

あるが，配列のような構造に対しては適用しにくい．もうひとつの方法は，プログラムを解析してデータの流れを調べておき，本質的にデータ値のコピーが必要でなければ処理前のデータに部分的更新を行ってしまう方法である．この方法は関数型言語のひとつである属性文法に適用され大きな効果を上げている[15]．

本稿では紙面の関係や原理的側面を解説するという目的からいくつかの重要な話題について触れることができなかった．無限長データとしてのストリームやその処理機構，特に関数型言語における遅延評価[16]や並行論理型プログラム言語[17]との関係，等式プログラミング[18]と抽象データ型との関連など非手続き型プログラム言語のデータ構造的側面には本稿で述べられなかったが興味あるテーマが数多くあることを付記しておきたい．

最後に本稿をまとめるにあたりいろいろご検討いただいた宮地利雄氏に深く感謝いたします．

参考文献

1) Backus, J.: Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra, Comm. ACM. 21, 8, pp. 613-641 (Aug. 1978).
2) Turner, D. A.: The Semantics of Applicative Languages, Proc. Conf. on Functional Programming Languages and Computer Architecture (Oct. 1981).
3) Henderson, P.: Functional Programming Application and Implementation, Prentice-Hall International (1980).
4) McCarthy, J. 他: Lispl. 5 Programmer's Manual, The MIT Press (1962).
5) Steele, G. L.: Common Lisp, Digital Press, 1984 (井田昌之訳 bit 別冊，共立出版，1985).
6) 中島：数理情報学入門，朝倉書店 (1982).
7) Kowalski, R.: Predicate Logic as Programming Languages, IFIP 74, pp. 569-574, North-Holland (1974).
8) Clocksin, W. F. and Mellish, C. S.: Programming in Prolog, Springer-Verlag 1981 (中村克彦訳 Prolog プログラミング，マイクロソフトウェア 1983).
9) Goldberg, A. and Robson, D.: Smalltalk-80 The Language and its Implementation, Addison Wesley (1983).
10) Bobrow, D. G. and Stefik, M. J.: The Loops Manual, Xerox (1983).
11) 宮地，篠田：オブジェクト・オリエンテッド型言語，Computer Today 11, 10, pp. 51-56 (1985).
12) Milner, R.: A Theory of Polymorphism in Programming, JCSS 17, pp. 348-375 (1978).
13) Katayama, T.: Type Inference and Type Checking for Functional Programming Languages, Proc. Lisp and Functional Programming Conf. (1984).
14) Borning, A. H. and Ingalls, D. H.: A Type Declaration and Inference System for Smalltalk, Conf. Record of 8th POPL (1981).
15) Farrow, R.: Experience with an Attribute Grammar-Based Compiler, Conf. Record of 9th POPL (1982).
16) Henderson, P. and Morris, J. H.: A Lazy Evaluator, Conf. Record of 3rd POPL (1976).
17) Shapiro, E. Y.: A Subset of Concurrent Prolog and its Interpreter, ICOT Technical Report TR-033 (1983).
18) Burstall, R. M. and Goguen, J. A.: Informal Introduction to Specification Using Clear, The Correctness Problem in Computer Science, J. S. Moore, ed., Academic Press (1981).

(昭和 60 年 12 月 23 日受付)