APLとOR(7)

配列処理の応用と新しいAPL

浜田節雄・宇土正浩

ＡＰＬの特徴は，演算の対象を配列と考え，配列処理に関する豊富な原始関数や作用素により，演算の手順が簡潔に表現できることであった．また一方，ＡＰＬの適用分野が拡大されるにともない，処理すべきデータの構造も複雑・多岐にわたってきている．

本号では，ＡＰＬの配列処理の立場からデータ処理の問題を改めて見直すこととする．まず，ＡＰＬと関係データ・モデルとの類似性を求め，「パーソナル・コンピューティング」としてのＡＰＬによる関係データ・ベースの試みについて述べる．次に，既存のＡＰＬが対象としていたデータ構造，すなわち配列を，より一般なものに拡張した新しいＡＰＬを紹介する．

1. ＡＰＬと関係データ・モデルとの類似性

1.1 ＡＰＬの配列処理

ＡＰＬが，その言語表現において最も大きな特徴としているものは，言うまでもなく，配列を中心としたデータの処理である．従来までのデータ・プロセスにおけるデータの処理形態では，記憶装置内におけるものは，スカラー量が単位であり，ファイルの入出力に関しては，レコードがその単位であった．

これは，そもそもコンピュータの装置そのものがそのように作られていることに起因しているが，適用業務を処理しようとする利用者の立場から見るとFORTRAN等の高級言語に見られる配列の処理が依然として，スカラーを単位として行なわれていること，またデータ・ベースに対するアクセスが，やはりレコード単位で行なわれていることに対して，“高級”の名にふさわしくないと感じるのは無理からぬところであろう．

汎用プログラミング言語において，初めてこの壁を打破したのがＡＰＬであった．

本来データ・プロセスというものは，利用者の立場から見るとかなり抽象的な世界である．データが機械の内部でどのように表現されているかとか，物理的な流れがどう行なわれるかについては本質的ではない．要は，入力を出力に変換する，変換機能にすぎない．

これを数学の世界で見ると，やはり数千年の歴史を有しているだけあって，はるかに進んでいる．数学者はそのデータ処理において，決してそのデータの最小単位までさかのぼって議論はしない．最も卑近な例をあげると，線形代数における行列やベクトルの表現方法であろう．たとえば，連立１次方程式を表現するのに，いちいちスカラー表現の１次式をたくさん書き並べるようなことはしないで，

$Ax=b$　（Aは係数行列；x, bはベクトル）

と書き，行列Aの特異性について吟味したのち，

$x=A^{-1}b$　（A^{-1}はAの逆行列）

と簡単に方程式を解く．

この種の考え方をプログラミングの世界に導入したのがＡＰＬであるといってよい．

1.2 配列と表

一方，データ・ベースのアクセスのほうはどうであろうか？　データを処理する立場から見ると，データ・ベースとは，ある構造をもったファイルの集合である．これを利用者がアクセスするためには，通常，PL/Iなどの高級言語を用いてプログラムを書く．高級といっても，処理の単位はやはりレコード単位である．

これはちょうど，FORTRANなどが配列を処理するのに，その要素（スカラー）を１つ１つ取り出すのに似ている．決してファイル全体（レコードの集合）に対する演算は行なわれない．ここに１つの問題がある．

70年代初頭，IBMのCoddは，関係データ・ベースという新しい考え方を発表したが，そのアイデアの１つに，“表”と“表”の演算という概念があり，これを自動化することを提唱している．

周知のとおり，関係データ・ベースでは，データの構造を，表の集合として定義し，各種要件による情報検索はいくつかの基本的な表の演算の組合せに帰着できると

はまだ せつお，うど まさひろ　日本アイ・ビー・エム㈱

している．

この考え方は，ＡＰＬにおける配列処理の概念と非常によく似ている．たとえば，表の演算の１つに，SELECTION というのがあり，この演算は，ある表の中から，特定の条件を満たす“行”のみを取り出し，新しい表を作り出すことを意味しているが，これをＡＰＬの配列処理でやると次のようになる．

表 1

社員番号	組織コード	内線番号	勤続年数
19390	49100	490	5.0
11540	11300	355	10.0
⋮	⋮	⋮	⋮

→Aとする
（APLの数値行列）

①上記の表で，組織コードが 49100 のものだけ取り出して，新しい表（行列 B とする）を作る．

$B\leftarrow(A[\,;2]=49100)\neq A$

②組織コードが 11300 で，かつ勤続年数が10年以上．

$B\leftarrow((A[\,;2]=11300)\wedge A[\,;4]\geq 10)\neq A$

次に，同じく表の演算で PROJECTION というのがあり，表から特定の列だけ取り出して新しい表を作るというもので，たとえば，

③表１から，社員番号と内線番号だけ取り出す．

$B\leftarrow 1\ 0\ 1\ 0/A$

さらに表からある行を削除したり，追加・更新の演算も簡単にＡＰＬでできる．

④表１から社員番号が 19390 の行を削除．

$A\leftarrow(A[\,;1]\neq 19390)\neq A$

⑤表１に新しい行（V）を追加．（ＡＰＬでは V はベクトルで与える）

$A\leftarrow A,\ [1]V$

更新については，削除と追加を続けて行なえばよい．

これらの例を見てわかるとおり，データが仮想記憶装置上に配列として与えられている時の，関係データ・ベースの処理は本当に造作もないことのように思われる．

昨今の関係データ・ベース・システムの開発の困難さは，むろんこのような単純なことではない．単純でない理由は大別して次の２点であろう．

1)データの表は一般に非常に大量のデータを含んでいることを想定する必要があるので，仮想記憶装置の上だけで考えるわけにゆかない．したがって，表のための外部ファイルを構築して，しかも表の演算を能率よく行なう必要がある．

2)一般の利用者（プログラミングを知らない）のために，使いやすい検索用の言語を提供する必要がある．

たしかにこの２つの問題点は汎用データ・ベース・システムを構築する際に生ずる重大な関心事には違いない．しかし一方ＡＰＬ等を用いて，自分自身の問題解決を行なおうとする，いわゆるパーソナル・コンピューティングにおいては，少々事情が異なると思われる．

第１に，表中のデータが数メガ・バイトというふうに大量なものはまれである．また，すべての表を一時に仮想記憶装置上にもってくる必要はない．演算(検索)に必要なものだけをロードすればよい．さらに最近のＴＳＳの環境では，各利用者がそれぞれ数メガ・バイトの記憶域を使用することはほとんど常識となっている．

第２に，検索用の簡易言語は必要ない．ＡＰＬを知っていれば，十分すぎるほどの検索が可能である．

概略以上のような理由から筆者は，配列を用いた，ＡＰＬによる関係データ・ベースの利用を提案したい．

1.3 ＡＰＬによる関係データ・ベースの試み

関係データ・モデルの考え方は，利用者が手持ちの，種々雑多なデータの集合を整理して貯えておくのに非常に都合がよい．なぜならば，利用者はそれらのデータを将来どのように利用するかという利用目的を考えることなく，とにかく個々の表を作り上げることができるからである．これは“表”の正規型というものが，データのもつ意味から（利用目的からではない）理論的に決定できることに起因している．

いずれにしてもわれわれはここで，外部入出力は一切考えない．データはすべて表の型でＡＰＬの配列としてワークスペースに格納される．

配列の形で格納されていれば，先の 1.2 で簡単な例を示したとおり，PROJECTION や SELECTION などは，いとも簡単なようであるが，厳密にいうと，少々工夫が必要であろう．

(1) 表の要素には数字や文字が混っている．

数値だけの配列であれば，1.2 で示したようにプログラムもきわめて簡単であるが，文字が混在する時は配列として，文字の配列を使用する必要がある．

(2) 表の行はユニークでなければならない．

これは，関係データ・モデルの表の定義からくるもので，実際にＡＰＬで表を扱おうとする時，表の各行がユニークかどうかのチェックを行なったり，重複する行をその表から削除したりする必要がある．

このような操作はＡＰＬでも若干面倒なので，あらかじめ関数を作っておくとよい．たとえば，

```
∇Z←M  UNIQ  U ; N1 ; A
[1]  N1←1↑ρM
[2]  A←(∨/<⍀M∧.=⍉M)≠M
[3]  ⍞U, '←A'
[4]  →(N1=1↑ρA)/Z←0
```

[5] $Z\leftarrow 1\ \nabla$

Mの各行がユニークならば関数値として0を，そうでない時は1を返す．また，Uには，重複の行を削除した新しい表を格納する配列名を文字列としていれる．

(3) PROJECTION の結果は重複行を許さない．

1.2の例③では，社員番号がすでにユニークであるので結果の重複行のチェックは行なっていない．しかし厳密には常に重複行の削除を必要とする．これも少々面倒なので関数を作っておいたほうがよい．

```
∇Z←M PROJ V;NN
[1] NN←(1↓ρM)÷N
[2] Z←M[;,(NN×¯1+,V)∘.+ιN]
[3] Z←(∨/<⍀Z∧.=⍉Z)⌿Z ∇
```

表の中の要素のフィールド桁数はすべてN桁と固定し，これは大域変数としてプログラムの外で与える．

Vは選ばれた列の番号をベクトルの形で入れる．

(4) 2つの表の JOIN

これは先の例では示さなかったが，関係データ・ベースでは最も重要で，しかも演算の能率という面で一番議論の多いものであるが，ここでは大規模なものは考えないことにして，能率の面は目をつぶり，一例を書いてみよう．

```
∇Z←S JOIN T;V;W;NN;N1;N2;Q
[1] 'COL. NO. OF 1ST TABLE?'
[2] N1←⎕
[3] 'COL. NO. OF 2ND TABLE?'
[4] N2←⎕
[5] V←S[;(N×N1-1)+ιN]
[6] W←T[;(N×N2-1)+ιN]
[7] Q←,V∧.=⍉W
[8] V←,(ι1↑ρS)∘.+(1↑ρT)ρ0
[9] W←((1↑ρS)×1↑ρT)ρι1↑ρT
[10] Q←Q⌿V,[1.5]W
[11] NN←(1↓ρT)÷N
[12] T←T[;(ιN×N2-1),
                (N×N2)+ιN×NN-N2]
[13] Z←S[Q[;1];],T[Q[;2];] ∇
```

JOIN を行なう，第1の表 (S) と第2の表 (T) の列番号は，それぞれ端末からの入力となっている．また JOIN を行なった T の列は自動的に削除される．

1.4 検索の例

さてこれまで用意してきた関数を利用した検索例を以下に示す．まず次の新しい表を定義する(表2)．

これと1.2で定義した表とを使って検索を行なうが，いずれも文字行列とし (1.2では表1を数値行列としたが，ここでは文字行列とする)，各列のフィールドの長さはすべて5桁としておこう．

表 2

組織コード	所在地
49100	TOKYO
11300	OSAKA
12345	KYOTO
11330	OSAKA
⋮	⋮

→Kとする
(APL の文字行列)

①社員番号と所在地のみの表を作りたい．

```
N←5
AK←A JOIN K
AK PROJ 1 5
```

②東京に勤務している人の平均勤続年数を知りたい．

```
X←(AK[;20+ι5]∧.='TOKYO')
⌿AK[;15+ι5](+/⌽,'',X)÷1↑ρX
```

③全国に所在地の数はいくつあるか．

```
ρK PROJ 2
```

このような検索は，多少関係データ・モデルの知識を必要とするが，ＡＰＬと連動することにより，きわめて簡単にしかも柔軟に対処できる．

データ・プロセスにおいては，従来の習慣から，すぐにファイルの入出力を考えがちであるが，ＡＰＬのワークスペースを活用する方法を再認識していただきたい．

2. 新しいＡＰＬ

2.1 ＡＰＬ２概要

ＡＰＬ言語を拡張する試みは1970年代初頭から行なわれてきたが，この２，３年で急速に実働化の方向に進んできた．そして，昨年の６月にＩＢＭは拡張された新しいＡＰＬ言語「ＡＰＬ２」を発表した．

ＡＰＬ言語処理システムは，1)言語の対象物である配列，関数，作用素，2)言語のシステム環境を支援するシステム変数，システム関数，システム・コマンド，と3)言語のシステム環境と環境外とを仲介する補助プロセッサーから構成されている．ＡＰＬ２の新しい機能の特徴は，1)に対して大幅な拡張を行なった点にある．言語の面で，これだけの変革があったのはＡＰＬ誕生以来はじめてである．

現在のＡＰＬが取扱える配列はスカラー要素の順序集合で，そこに含まれる要素はすべて数値スカラーか，あるいはすべて文字スカラーかのどちらかである．ＡＰＬ２では，この制約を取りはずし次のように拡張した：

- 配列の要素として，数値スカラー，文字スカラー以外に「配列スカラー」も可能．(いわゆる入れ子構造配列，nested array)
- 配列の要素として，上記３種類のスカラーが混在す

ることも可能．（いわゆる混合配列，mixed array）

• 数値スカラーとして，実数以外に「複素数」も可能．

この拡張にともなって，新しい原始関数と作用素が導入された．付録にＡＰＬ２の原始関数と作用素の一覧表を掲げる．

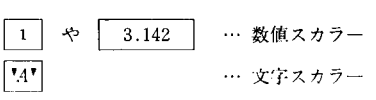
本稿では，これらのなかで特に拡張された配列に関するものを取りあげ，ＡＰＬ２における配列処理の特徴について述べ，応用の可能性をみることとする．

2.2 ＡＰＬ２の配列

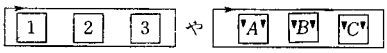
ＡＰＬ２では，配列の属性として従来の形状（shape）と階数（rank）の他に，入れ子構造（nesting）の状況を知るための深さ（depth）という概念が必要となってきた．深さが０の配列は 1）数値スカラーか 2）文字スカラーかで，この２つを単純（simple）スカラーと呼ぶ．深さが１の配列には 1）従来の配列すべてと 2）数値スカラーと文字スカラーだけから構成される混合配列が含まれ，これらを単純配列と呼ぶ．深さ２以上のものが非単純配列で，いわゆる入れ子構造をもった配列である．

配列の構造を説明する便宜上，配列の「箱型」表示をもちいる．

1）スカラーは箱で囲まれる．

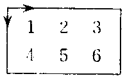


2）ベクトルは横矢印付の箱でスカラー要素の列を囲む．

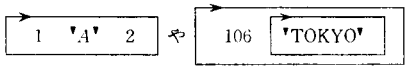


ただし，ベクトル中のスカラー要素を囲む箱は省略することがある．また，連続する文字スカラーの間の「'」や「ブランク」も省略する．すなわち，

3）マトリックスは縦横矢印付の箱で方形に並べられたスカラー要素を囲む．（ベクトルと同様な省略をもちいる）



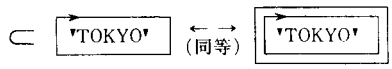
以上の箱型表示をもちいて，ＡＰＬ２の新しい配列を表示してみると，



などがその例である．前者は要素数３，深さ１の混合・単純配列（ベクトル）．後者は要素数２，深さ２の混合・非単純配列（ベクトル）で，その第２要素は要素数５の文字ベクトルを「スカラー化」したもので，いわゆる入れ子構造となっている．

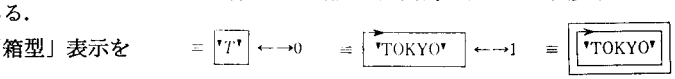
2.3 新しい原始関数の例

任意の配列をスカラー化し，配列スカラーとするための新しい原始関数が「Enclose（封入）」で記号は「⊂」をもちい，単項関数である．この関数の機能を箱型表示で示すと，



となる．つまり，任意の配列は１つの箱で囲い込まれ（封入され）てしまう．この逆関数が「Disclose（開封）」で記号は「⊃」を単項形式でもちいる．

封入された配列スカラーは，それ自身が，任意の配列の要素となりうる．封入は幾重にも繰り返すことが可能であり，配列の要素の封入の度合いを知るための原始関数が「Depth（深さ）」で記号は「≡」の単項形式をもちいる．単純スカラーの深さが０で，単純配列の深さが１である．単純配列が封入されると深さ２となり，封入ごとに深さは１増加し，開封されると１減少する．



なお，単純スカラーは封入，開封に影響されず，封入されても深さは０のままである．

配列スカラーを要素とする新しい配列を作成するには従来からの「ベクトル記法」を拡大解釈して利用する．つまり，ベクトル記法によれば，封入と連結の原始関数をもちいて作成される次の式と同等なベクトルを簡便に作ることができる：

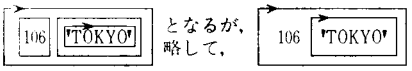
$$A\ \ B\ \ C \longleftrightarrow (\subset A),\ (\subset B),\ \subset C$$

たとえば，

$X\leftarrow 106\ \ \text{'TOKYO'} \longleftrightarrow X\leftarrow(\subset 106),\ \subset\text{'TOKYO'}$

$\longleftrightarrow X\leftarrow 106,\ \subset\text{'TOKYO'}$

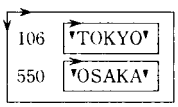
このベクトル X を箱型表示すれば，



とする．ところで，X の深さ（$\equiv X$）は２である．すなわち任意の配列の深さは要素中の最大深度とする．

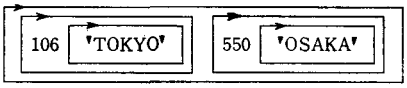
ベクトル記法をもちいると，非単純マトリックスも従来と同様な方法で作成できる．

$Y\leftarrow 2\ \ 2\rho\ \ 106\ \ \text{'TOKYO'}\ \ 550\ \ \text{'OSAKA'}$



カッコを使えば，より複雑な入れ子構造をもつ配列を定義できる．カッコでくくられた部分は，それだけで１つの配列スカラーを形成する．

$Z\leftarrow(106\ \ \text{'TOKYO'})\ \ (550\ \ \text{'OSAKA'})$



さて次に，配列の要素を選択するための原始関数について若干述べる．従来からの指標付け原始関数もＡＰＬ２の配列に対して作用させることができる．たとえば，

$X[2]$ や $Y[1;2]$ ←→ 'TOKYO'

によって，配列の要素を選択できるが，その要素が配列スカラーの場合（上記の例）には，それを開封してさらにその中の要素を選択する必要も生じてくる．これを実行するには，開封と指標付け原始関数を組合せて適用すればよい．たとえば，'TOKYO' の頭１文字は，

$(\supset X[2])[1]$ や $(\supset Y[1;2])[1]$ ←→ 'T'

によって選択できる．しかし，配列Zに対して同様な結果をうるためには，

$(\supset(\supset Z[1])[2])[1]$ ←→ 'T'

となる．深い入れ子構造をもつ配列に対して，任意の深さにおける要素を選択する手間が繁雑になってくる．

このため，ＡＰＬ2では新しい原始関数「Pick(摘出)」を導入した．記号は開封と同じ「⊃」を２項形式でもちいる．これを利用すれば，上の例はそれぞれ，

```
2   1 ⊃ X      ←→  'T'
(1  2)1 ⊃ Y    ←→   〃
1   2  1⊃ Z    ←→   〃
```

と簡潔に記述できる．左辺の整数ベクトルを「path index (経路指標)」と呼ぶこともあり，実際，配列の入れ子構造を辿りながら，求める要素に到達する道順を示している．

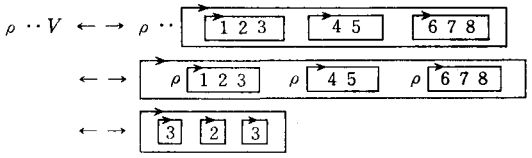
2.4 新しい原始作用素の例

ＡＰＬ２では，作用素も(従来の関数と同様な方法で)ユーザーが定義できるようになったので，「原始作用素」と「ユーザー定義作用素」を区別する必要がでてきた．

ここでは，新しく導入された原始作用素の例として「Each(個別)」をとりあげ，この作用素がもつ配列処理機能について述べる．個別作用素は記号「¨」をもちい，引数となる関数（単項，２項いずれも可）を「スカラー関数化」する．つまり，導出された関数はその引数配列の要素ごとに適用される．たとえば，ベクトルV

$V\leftarrow(1\ 2\ 3)\ (4\ 5)\ (6\ 7\ 8)$

に対して，各要素ごとにそれぞれの形状を求めることは



により可能である．この場合，形状原始関数（単項）が個別原始作用素の引数となり，導出された関数（単項）が引数配列Vに適用されている．導出された関数はスカラー関数化されているので，Vの個別要素ごとに引数関数（ρ）が実行され，その答えが結果の配列の要素となる．

引数関数として，任意の関数（原始，ユーザー定義，導出関数）がとれる．たとえば，Vの各要素ごとにそれぞれの先頭要素を摘出することは，

$1\supset\ddot{}V$ ←→ 1 4 6

により可能であり，各要素ごとの部分和を求めることは，

$+/\ddot{}V$ ←→ 6 9 21

により可能である．

個別原始作用素の導入により，配列処理の際，もしそれがなければ避けられないプログラムのループを，大幅に削減できる．

3. 配列処理の応用

拡張された配列をもちいれば，どのようなデータ構造が自然に表現でき，また処理できるであろうか．ＡＰＬ２では，「スカラー化された配列は配列の要素となる」というように配列を再帰的に定義しているので，拡張された配列が表現できるデータ構造は格段にひろがったと言える．そのなかで，特に第１章との関連で有用と思われる「表型データ」と「属性リスト型データ」について述べる．

3.1 表型データ

例として表１，表２(第１章)をとりあげる．表１は数値スカラーだけを含むので，これは従来の配列（単純配列）で表現できる．しかし，表２は「所在地名」を含むので，従来だと表を，たとえばm行10列の，文字行列で表現せざるをえなかった．ＡＰＬ２では，たとえばm行２列の非単純配列として,表２をより自然に表現できる．

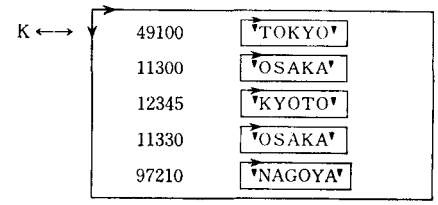


図1に，これらの配列をもちいて表型データの処理例を示す．処理①は配列Aの組織コード（$A[;2]$）と対応する所在地（$K[;2]$）を配列Kから，指標調べ原始関数（ι）をもちいて取り出し，配列Aに連結（,）させたものである．この場合，組織コードは数値スカラーであるから原始関数ιの使い方は従来と変わりない．しかし，処理②はこの関数が配列スカラー（所在地，文字ベクトルがスカラー化されたもの）に対しても，まったく同様に適用できることを示している．ここで，配列Bは表3をm行3列の非単純配列で表現した．

表 3

所在地	郵便番号	電話番号
TOKYO	106	03-586-1111 →Bとする
NAGOYA	450	052-583-4111
KYOTO	600	075-221-0166
⋮	⋮	⋮

処理③，④は東京に勤務している人の平均勤続年数を計算している．処理⑤では表2（K）の第2列にある所在地の名前のなかでユニークなものだけを，新しい原始関数「Unique」∩を使って選択し，その数を求めた．

3.2 属性リスト型データ

データ処理では，処理の対象となる事象（entity）がもつ種々の属性を表現する必要がでてくる．たとえば，

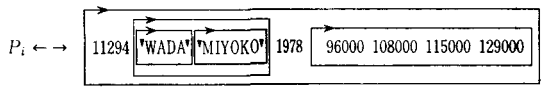
表 4

社員：	属性	属性値
	社員番号	11294
	名前 姓	WADA
	名前 名	MIYOKO
	入社年	1978
	給料履歴	96000 108000 115000 129000

表4は「社員」のもつ属性と属性値の例である．属性値は多様なデータ・タイプ（数値スカラー，ベクトル，文字列他）をとりうる．このデータがすべての社員について必要となる．これを属性リスト型データと呼ぶ．

ＡＰＬ２では次のように表現できる．

$P \leftarrow\rightarrow (\subset P_1), (\subset P_2), \cdots, \subset P_n$




```
      A                                          A: 社員番号 | 組織コード | 内線番号 | 勤続年数
19390 49100 490 5                                   ┌──────┐
11545 11300 355 6                                   │ 表 1 │
11294 49100 492 4                                   └──────┘
67664 97210 585 2
92515 11300 982 5

      K                                          K: 組織コード | 所在地
49100 TOKYO                                         ┌──────┐
11300 OSAKA                                         │ 表 2 │
12345 KYOTO                                         └──────┘
11330 OSAKA
97210 NAGOYA

      □←AK←A,K[K[;1]⍳A[;2];2]  ──────────①
19390 49100 490 5 TOKYO
11545 11300 355 6 OSAKA
11294 49100 492 4 TOKYO
67664 97210 585 2 NAGOYA
92515 11300 982 5 OSAKA

      B                                          B: 所在地 | 郵便番号 | 電話番号
TOKYO    106 03-586-1111                            ┌──────┐
NAGOYA   450 052-583-4111                           │ 表 3 │
KYOTO    600 075-221-0166                           └──────┘
OSAKA    550 06-441-6111
FUKUOKA  812 092-411-3006
NAHA     900 0988-62-3245

      □←AKB←AK,B[B[;1]⍳AK[;5];2 3]  ──────②
19390 49100 490 5 TOKYO  106 03-586-1111
11545 11300 355 6 OSAKA  550 06-441-6111
11294 49100 492 4 TOKYO  106 03-586-1111
67664 97210 585 2 NAGOYA 450 052-583-4111
92515 11300 982 5 OSAKA  550 06-441-6111

      (AKB[;5]∈⊂'TOKYO')≠AKB  ──────────③
19390 49100 490 5 TOKYO 106 03-586-1111
11294 49100 492 4 TOKYO 106 03-586-1111

      (+/X)÷⍴X←(AKB[;5]∈⊂'TOKYO')≠AKB[;4]  ──④
4.5
      □←NOB←(∩K[;2])/K[;2]  ──────────⑤
TOKYO OSAKA KYOTO NAGOYA
      ⍴NOB
4
```

図 1　表型データの処理例

つまり，個々の社員の属性値を非単純ベクトル（箱型表示参照）としてもち，それをスカラー化した要素を連結したものが全社員の属性値を表現している．

図2はこの配列の処理例である．処理①の部分は，配列Pが5つの要素からなるベクトルであり，第1番目の要素を摘出原始関数⊃で解剖しているところである．処理②は個別原始作用素¨を利用して，全員の社員番号を摘出し，③と④では同様に姓と名をそれぞれ摘出している．処理⑤では，全員の最新の給料を取り出している．最後に処理⑥と⑦で，社員の属性から③，④，⑤で得られたデータを，前節の表型データと連結表示した．

参考文献

［1］ APL 2 Language Manual, SB 21-3015, IBM

［2］ An Introduction to APL 2, SB 21-3039, IBM

（なお，第2章・第3章で使用したＡＰＬ２の新しい関数名などの訳語は，まだ標準として定まっていない．）

```
P: 社員番号   名 前 (姓 名)   入社年   給 料 履 歴

      ⍴P
5
      1⊃P
 11294   WADA MIYOKO   1978  96000 108000 115000 129000
      1 1⊃P                                              ①
11294
      1 2 1⊃P
WADA
      1 2 2⊃P
MIYOKO

      1⊃¨P ──────────────────────────────────────────────②
11294 11545 19390 67664 92515

      (⊂2 1)⊃¨P ─────────────────────────────────────────③
 WADA IDE TANAKA SATO AOKI
      (⊂2 2)⊃¨P ─────────────────────────────────────────④
 MIYOKO AKIO ICHIRO MAKOTO FUMIKO

      1 4⊃P
96000 108000 115000 129000
      ⍒1↑1 4⊃P
129000

      ⍒1↑¨4⊃¨P ──────────────────────────────────────────⑤
 129000  145000  160000  131000  133000

      □←LL←A[;1]⍋1⊃¨P ───────────────────────────────────⑥
3 2 1 4 5                                                ⑦
      □←AKB←((⊂2 1)⊃¨P)[LL],((⊂2 2)⊃¨P)[LL],AKB,(⍒1↑¨4⊃¨P)[LL]
 TANAKA ICHIRO 19390 49100 490 5 TOKYO  106 03-586-1111   160000
 IDE    AKIO   11545 11300 355 6 OSAKA  550 06-441-6111   145000
 WADA   MIYOKO 11294 49100 492 4 TOKYO  106 03-586-1111   129000
 SATO   MAKOTO 67664 97210 585 2 NAGOYA 450 052-583-4111  131000
 AOKI   FUMIKO 92515 11300 982 5 OSAKA  550 06-441-6111   133000
```

図 2　属性リスト型データの処理例

付録A. ＡＰＬ２の原始関数と作用素の一覧表

表A-1　原始関数（その１）（文献［１］29ページより）

Symbol	Monadic	Pg	Dyadic	Pg
+	Conjugate	36	Add	78
-	Negative	40	Subtract	93
×	Direction	36	Multiply	87
÷	Reciprocal	41	Divide	81
|	Magnitude	39	Residue	99
⌊	Floor	38	Minimum	86
⌈	Ceiling	35	Maximum	85
*	Exponential	37	Power	91
⍟	Natural Log.	40	Logarithm	84
○	Pi Times	41	Circular	80
!	Factorial	37	Binomial	79
~	Not	40	{note[1]}	
?	Roll	42	{note[1]}	
∈	Type	42	{note[1]}	
∧			And	79
∨			Or	91
⍲			Nand	87
⍱			Nor	88
<			Less	83
≤			Not Greater	89
=			Equal	82
≥			Not Less	90
>			Greater	83
≠			Not Equal	88

Note:
All dyadic forms may take an axis.
[1] The dyadic form is not pervasive.

表A-2 原始関数（その2）（文献［1］32ページより）

Class	Sym	Monadic	Pg	Dyadic	Pg
Structural	⍴	{note¹}		Reshape	99
	,	Ravel []	49	Catenate []	95
	⌽	Reverse []	52	Rotate []	100
	⊖	Reverse []	52	Rotate []	100
	⍉	Transpose	53	Transpose	102
	⊂	Enclose []	46		
	⊃	Disclose []	43	{note¹}	
	∪	Unite	54		
Selection	⊃	First	56	Pick	114
	↑			Drop []	105
	↓			Take []	118
	/			Replicate []	115
	⌿			Replicate []	115
	\			Expand []	107
	⍀			Expand []	107
	⌷	{note¹}		Index []	109
	~	{note¹}		Without	120
Selector	⍳	Interval	61	Index of	131
	⌷	Index set	60	{note¹}	
	∩	Unique []	61		
	∈	{note¹}		Member	131
	⍋	Grade Up	59	Grade Up	129
	⍒	Grade Down	57	Grade Down	128
	?	{note¹}		Deal	122
	⍷			Find []	122
	⍸			Find Ind. []	125
Mixed	⊤			Encode	132
	⊥			Decode	132
	⌹	Mat. Inv.	65	Mat. Divide	133
	⍁	Eigen	64		
	⍎	Poly. Zeros	67		
Transform.	⍴	Shape	74	{note¹}	
	≡	Depth	68	Match	137
	⍎	Execute	69		
	⍕	Format	70	Format	138
Misc.	→	Branch	148		
	[;]			Indexing	146

Notes:
[] indicates that an axis specification is optional.
¹ This function is in another class.

表A-3 原始作用素（文献［1］155ページより）

Class	Name	Producing Monadic	Pg	Producing Dyadic	Pg
Monadic	Each	*F¨ R*	156	*L F¨ R*	157
	Reduce	*F/ R*	160	*L F/ R*	158
	Reduce	*F⌿ R*	160	*L F⌿ R*	158
	Scan	*F\ R*	165		
	Scan	*F⍀ R*	165		
	Reduce w Axis	*F/[A] R*	165	*L F/[A] R*	159
	Scan w Axis	*F\[A] R*	167		
	Bracket Axis	*F[;] R*	168	*L F[;;] R*	172
Dyadic	Inner Product			*L F.G R*	178
	Outer Product			*L ∘.G R*	176

Notes:
F and *G* are function operands of an operator.
A is a simple scalar or vector axis specification.
L and *R* are array arguments of a derived function.