以下の文章はIBMの“APL2 Language Summary
(IBM資料番号 SX26-3851)の一部を翻訳したものです。

2008年9月30日　　　　三枝　協亮
(有)エー・ピー・エル・コンサルタンツ・オブ・ジャパン
URL: http://aplcons.com
Mail: kyosuke.saigusa@nifty.com

APL2の紹介

この資料はこれからAPLを習得されようとしていらっしゃる方に、APL2言語のシンタックスと特性についてある程度の理解をしていただく事を目標にしています。またAPLがはじめてでない人も、APL2で追加された最新の機能の記述が、役に立てていただけることを期待しています。

APLとはなにか？

APLは商用データ処理、システム・デザイン、数学・科学計算、教育など、とても広範囲な目的に使用することのできる汎用的なプログラミング言語です。特にデータベースを利用する適用業務では、その計算能力と通信機能があいまって、アプリケーション・プログラマーおよびエンドユーザーの生産性と作業の品質を高める効果があります。

APLシステムを卓上計算機として使用する場合は、APLはキーボードからの操作で使用します。遂行したい作業を指定する式はキーボードからタイプ入力し、コンピュータは計算結果を直ちに表示します。結果の表示はディスプレイ装置や印刷装置のような、キーボードに付随した装置に表示されます。キーボードや付随するディスプレイ装置のみで遂行される作業以外に、入力によっては印刷装置、ディスク・ファイルその他遠隔装置などの利用を伴うこともあります。

“APL”という文字は、Kenneth E Iverson博士の著作“A Programming Language”(New York: Wiley, 1962)に由来します。アイバーソン博士はAPL言語を、最初はハーバード大学で、その後IBMでAdin Falkoff氏などの協力を得て開発しました。APLと言う用語は現在、その作業から発展して生み出された言語を指します。

APL2はTrenchard More氏によって研究され、James A Brown氏の博士コースの卒業論文のテーマとして、さらに深く研究され、ブラウン博士の指導のもとにIBMにおいて開発された、配列の拡張を含む言語の製品化されたものです。

この言語のその他の特徴には、強力さ、無駄のなさ、単純さなどがあります。

強力さ、無駄のなさ、単純さ

プログラミング言語は作業を定義するのに、理論上必要なことのみを、無駄なく記述することが理想です。しかし、プログラム言語の多くは、解決すべき問題そのものの定義と同じくらいに、マシンに起因するさまざまな要求についても考慮しなければなりません。APL2ではマシン内部の要求に関することはユーザーに負担を与えずに、自動的に処理します。

プログラミング言語は強力さと単純さの両方を必要とします：大規模な、あるいは複雑な作業を処理する能力と、何を実行するべきかを、読み書きしやすい簡潔な、かつスマートな記述で表現できる単純さです。能力の高さと同時に単純さの追求は必ずしも相反する要求ではありません。プログラミング言語としてのAPLの強力さは、一部その単純さに起因します。その単純さの故に初心者にも、上級ユーザーにも同時に使いやすいのです。

APLの簡単な使用例

APLではプログラムを書いたり、名前を付けた変数を取り扱わなくても問題が解けることがしばしばあります。単に問題の表現をタイプ入力するだけで、結果が表示されます。ひとつの例をあげてみましょう：

細菌の多くは30分ごとに2倍に増えます。もし感染性の微生物が朝9時に増殖し始めたとしたら、作られる群体はどれぐらい早く成長するでしょうか？

正午、午後6時、真夜中での細菌の数を計算してみましょう。言い換えれば、3時間後、9時間後、15時間後です。2倍で増えますから、2のべき乗を使い、次のように書きます:

```
2⋆...
```

ここでの．．．はAPLの表現ではありません。まだ記述が終わってないことを意味します。毎時間2回2倍に増殖しますから、続けて次のように書きます。

```
      2⋆2×...
```

私たちが関心のある経過時間を書いてみてください。これでシステムがこの問題を解くために必要なすべてを記述したことになります：

```
      2⋆2×3 9 15
64 262144 1073741824
```

したがって、正午までに、バクテリアが64個になっています。午後6時までには100万個の4分の1以上、真夜中までに10億以上になっています！APL2のいくつかの特徴はこの例に示されています：見慣れた記号、例えば掛け算の×、が可能なかぎり使用されています。その他の記号は表記が水平面上に現れるように使用されています。（例えばべき乗をあらわすのに行を高くずらした表現でなく＊を用いることなど）。また（非常に重要なことですが！）　ひとかたまりの数値がいっしょに処理されていることです。

APLの特性

言語の基本的な取り扱い対象は配列（リスト、テーブル、テーブルのリストなど）であること。（例えば、A＋BはA，Bがどのような配列であっても意味を持ちます。）

シンタックスが単純であること。 関数間での実行優先順位付けはありません。また、内蔵の関数とユーザーが定義した関数（プログラム）とは同一に扱われます。

プログラムの文法規則が少ないこと。 内蔵の関数はデータのタイプに依存しない形で定義されています。また、隠れた副次的な結果を生じさせることもありません。

実行順序の制御が単純であること。 ひとつの式に、すべてのタイプの実行の流れの変化の指定（リターン、条件付きの実行、CASE、ジャンプなど）

を含めることができます。またすべての関数は、実行が中断されると、制御はそれが使用されていた場所に返されます。

外部との通信は、APLと他のシステムまたはサブ・システムの間で直接共有されるデータによって確立されます。これら共用変数は、構文上また意味論上他のデータと同一に扱われます。システム変数と呼ばれる下位分類は、APLプログラムとその稼動環境の間の便利な通信手段を提供します。

原始関数と呼ばれる内蔵の機能の利用度は、その機能をシステム的に修飾する作用子によって格段に強化されます。例えば、

・ 低減（/で表示される）は、対象関数を修飾して、リストに含まれる個々の値のすべてに作用させます。したがって　＋/　はリストLに含まれるすべての値の合計となります。

・ 軸指定（ [n] で表示される）は、対象関数を修飾して、テーブルの指定された軸方角へ作用させます。

さらに、APL2では、必要に応じて関数や作用子を定義することができます。また、そのようなユーザーに定義されたプログラムは、原始関数および原始作用子と構文上同じに取り扱われます。

原始関数や原始作用子は、各々が、容易に読み取れ、容易に書きだすことのできる、記号文字で表わされ、数も少ないです（APL2で使用される特殊記号は40個強です）。しかし、原始関数は単純な加算から複雑なタイプの書式化や、双曲線の余弦やマトリックス転換のような高等数学的な概念迄含んでいます。

APLを始めよう

APLをはじめるにあたって役に立つ話題を取り上げます:

APLは対話型です。
誰が何をタイプしたか？
表現。
APL文字セット

APLは対話型です

APL2システムはAPL2の式をひとつづつ機械の命令 (コンピュータの内部言語) に置き換え、実行し、次の行に進みます。これは従来のプログラム・コンパイラーがプログラム全体を、実行する前にすべて機械語に変換するのとは対照的です。これによりコンピュータとの高度なやり取りが可能になります。もし入力したものが正しくない場合、次に進む前に、すばやくその問題のフィードバックが得られます。

誰が何をタイプしたか？

APL2では通常、ユーザーとAPLシステムの間で長時間にわたる対話が行われます。その対話はログに記録されます、セッション・マネージャーと呼ばれるAPL2の構成機能よって管理されます。APLが開始されると、セッション・マネージャーが、既存のログを画面上に表示し、そこに記録された情報を閲覧したり、再度利用することができるようになります。ログが開いている時間をAPL2セッションと呼びます。

APL2セッションの間、あなたとAPL2とは交互に情報をログに書き加えます。あなたが情報をタイプ入力している時、APL2はあなたからの合図を待ち、入力された情報を利用して、その結果をログに表示します。あなたからの合図は通常Enterキーを押すことです。あなたがEnterを押すと、セッション・マネージャーはモード指標を入力モードからランニングモードに変更します。(この指標は通常ログ・ウインドウの左下隅に表示されます) 。次にあなたの番になると、入力モードに戻ります。

あなたとAPL2が互いに「対話している」間、セッション・マネージャーのモード指標は非常に役に立ちます。しかし、あとでその対話を閲覧するときには助けになりません。カラー・ディスプレイを使用する場合はセッション・マネジャーに指示して入力と出力を異なる色で表示させせることができます。しかし前のセッションを閲覧する場合は、これも役立ちません。なぜならログは対話を記録するオペレーティング・システムの標準ファイル（文字の色彩属性を記録しない）に過ぎないからです。

APL2が情報を表示する場合は、新しいラインを左のマージンから始めます。出力の表示をし終えた後、左のマージンからの6文字分のスペースをとって止まり、次のキーボード入力をタイプしてよい合図とします。この位

置はあなたの「番」であることを意味しますので、あとでログを読み返すときに役に立ちます。

```
      2+2
4
      3×4
12
```

<u>表現</u>

APL2の典型的な表現は次のような形式です:

```
      AREA←3×4
```

この式は名前AREAに左向き矢印(←)の右側の 3×4 の結果の値を割り当てる効果があります。平たくは“ＡＲＥＡは３かける４です” と読むことができます。表現の左端の部分が左向き矢印を伴う名前でない場合は、式の結果は画面上に表示されます。例えば:

```
      3×4
12

      PERIMETER←2×(3+4)
      PERIMETER
14
```

式の左端の部分がここでは重要です。なぜならAPL2の式の解釈の順番は右から左に行われます。したがって、左端の部分が最後に解釈されます。しかし、式の解釈の順番についてはもう少しあとで取り扱います。

APLのシステムでは伝統的にイタリックの大文字が使用されていますが、この資料では、ディスプレー上で表示される例題の部分を大文字の標準体で表示します。もちろん実際のAPL2のセッションでは小文字も使えますし、またイタリック文字にするかそうでないかを選ぶこともできます。

Enter キー以外にセッション・マネジャーのシグナル・メニューを用いてAPL2に合図を送ることができます。これらの合図は通常入力待ちの状態のときでなく、ランニングの状態のときに用いられます。

Attention は“きりの良い所で止まってください”、Interrupt は“直

ちに止まってください”という合図です。

APL文字セット

式で用いられる文字は英数字、機能文字、その他の特殊文字、ブランクの4種類に分類されます。:

・ 英数字は名前や定数で使用されます。
・ 機能文字はデータに作用する特定の操作を表す関数文字や、関数の動きを修飾する作用子文字に使用されます。
・ その他の特殊文字は、括弧のように関数や作用子でないが、式の解釈に影響するものです。
・ ブランクは名前の区分けなどに使用されます。

ファンダメンタルズ

ここでは“APLを始めよう”の章よりもさらに詳しい言語の概略を記述しますが、これば正規な言語仕様記述ではありません。完全で詳細な仕様の記述は:

"APL2プログラミング:言語解説書” SH88−7015
　日本IBM株式会社　1995. 3

の「第二章　配列」と「第三章　構文と式」を参照してください。

註:ちなみにこの書物の原典である’APL2 Programming:Language Reference' SH21-1061 は米国IBM社のホーム・ページよりPDFの形で、無償でダウンロードすることができます。

名前
数値
関数
作用子
データ
名前に値を設定する
解釈の順序
エラー

個々の言語要素が提供する機能の要約は同じ資料の下記の項目を参照してください:

原始関数
原始作用子
システム関数
システム変数
システム コマンド

名前

名前を形成するための有効な文字は:

ABCDEFGHIJKLMNOPQRSTUVWXYZ∆
abcdefghijklmnopqrstuvwxyz⍙
0123456789¯_
その他ヨーロッパ各国語の母音文字など

(ユーザーが定義する)作業空間、関数、変数、作用子およびラベルの名前は、最初の文字を除いて、上記の文字を自由に組み合わせて用いることができます。最初の文字だけは、上記数字の行の文字を使用できません。唯一の例外は T∆ と S∆ で始まる名前は定義できないことです。

次のものはすべて有効な名前です:

A
ABc
SALES_REPORT
TAX1984
∆

次のものは有効な名前ではありません:

A B ⍝ スペースを含んでいます。
1984TAX ⍝ 数字で始まっています。
REPORT ⍝ ""で始まっています。
DATA.3 ⍝ "."は使用できません。

APL2のオペレーションは作業空間 (=APL作業単位である作業空間

と呼ばれるメモリー領域) 単位でくくられます。したがって、関数、変数、作用子、ラベルにつける名前は作業空間が異なればお互いに干渉することなく同じものを使用することができます。さらに、作業空間の名前にその作業空間内のオブジェクトと同じ名前をつけることができます。

作業空間名は、通常8文字に制限されています。また、有効なAPL名でなければなりません。(しかし、特別のシンタックスの使用によっては、オペレーティング・システムで認められるどんな名前も使用することができます)。

変数、関数、作用子およびラベルの名前は255文字以内であれば任意の長さにすることができますが、おそらくそれ以上長い名前は必要でないでしょう。

数値

数の入力および表示はすべて10進数で行われ、(小数点を伴う形も含めて)通常の表記法または指数表示が使用されます。指数表示の場合は、Multiplier と呼ばれる部分を整数または小数で表記し、続けて“E”その後ろに Scale と呼ばれる部分を整数(小数を含んではならない)で表示します。Scale は 10 の乗数で Multiplier と掛け合わされます。したがって、1.44E2 は 144 と同一です。

APL2は複素数をサポートしますが、通常実数と虚数部分を“J”で分離した形で表記します。さらに、角度をラジアンまたは度数で表現する Polar 形式も利用可能です。例えば、マイナス 1 の平方根は標準表記では、0J1 、Polar ラジアン表記では 0R1.570796327 、Polar 度数表記では 1D90 と入力します。複素数の表示は常に、J形式の標準表記で行われます。

負の数は ¯ が頭についた形で表現されます。例えば、¯1.44 と ¯144E¯2 は等しい負の数です。負の値を示す ¯ は減算関数の - と異なることに注意してください。

```
      5 3
5 3
      5 ¯3
5 ¯3
```

```
      5-3
2
      5 -3
2
```

反対に、 ¯ は関数として使用することはできません:

```
      X←5
      Y←3
      X-Y
2
      -X
¯5
      X¯Y
VALUE ERROR
      X¯Y
       ^
      ¯X
SYNTAX ERROR
      ¯X
       ^
```

X¯Y は名前としては有効ですが、まだ値が定義されていません。¯X は有効な名前ではありませんが、その文字がすべて英数字なので式として扱われ、 syntax error が発生します。

関数

関数はその引数に作用して、結果を生成させますが、その結果の値はまた、次の関数の引数として使用されることがあります。例えば:

```
      3×4
12

      2+(3×4)
14
      (-6)÷3
¯2
```

+, -, ×, ÷ のような記号によって表わされる関数は原始関数と呼ばれ、これらはどこからもコピーしないで、どの作業空間ででも自動的に利用可

能になります。

その他の関数には名前がつきます。それらはユーザー定義関数（＝APL プログラム）およびシステム関数です。システム関数は、原始関数のように、どの作業空間ででも自動的に利用可能になります。しかし、それらは、⎕ で始まる特別な名前をもっています。例えば ⎕DL は指定した時間だけプログラムの実効を遅らせる関数です。

例えば前に挙げた -6 の - のように1つの引き数をとる関数を一項関数といいます;掛け算のように2つの引き数をとる関数を二項関数といいます。APLでは一項関数の引数は常にその右側に来ます。二項関数（原始関数、システム関数、定義関数の如何にかかわらず）は左右に一個ずつ引数を持ちます。

すべての場合に、同じシンボルまたは名前は一項でもあり、二項でもあるということがあります。例えば、X-Y （二項）は X から Y を減算しますが、-Y （一項）は符号の変換です。APL2では二項として定義された関数はすべて、一項としても使用されうるものと想定します。この特性を ambi-valence と呼びます。

ユーザー定義関数は引数をとらない形で設定することができます。原始関数やシステム関数には引数をとらない形のものはありません。システムは引数をとらない関数をあたかも変数のように扱うことがあります。

作用子

関数の通常な機能は作用子を適用することにより変化します。例えば、+ や × は配列の個々の構成要素の値に作用します。

```
      A←5 6 7
      B←2 3 4
      A+B
7 9 11
      A×B
10 18 28
```

/ 作用子を使って +/ ×/ の派生的関数を組み立てると、通常の機能がひとつの引数の各要素間に作用するように変化します。同じデータを使用すると:

```
      +/A          ⍝ 5+6+7 と同じ。
18
      ×/B          ⍝ 2×3×4 と同じ。
24
```

/ 作用子は全ての二項関数に適用することができます。作用子はユーザー定義関数に対しても使用することができますし、作用子そのものを定義することができます。

用語:関数と作用子

コンピュータ言語は時として、関数(function)と作用子(operator)という用語の定義が曖昧になっていますが、APLでは、これらの用語を次のように区別します:

・ 関数は引き数としてデータ・オブジェクトをとり、その結果新しいデータを返します。
・ 作用子は関数(時としてデータ・オブジェクト)をオペランドとし、その結果派生関数と呼ぶ新しい関数を作ります。

作用子はオペランドをとり、関数は引数をとります。作用子は必ず左側にオペランドをとります。二項作用子は右側に第二オペランドをとります。派生関数としてその引数を指定する場合は、作用子とそのオペランドをカッコで括ることができます。

例えば、ρ は Reshape (形を整える)と呼ばれ、左側の引数で右側の引数の形をどのように整えるかを指定する関数です。¨は Each (個々)と呼ばれる作用子でオペランドである関数を派生関数としての引数の各要素に作用させる働きをもちます。

```
      3ρ4
4 4 4
      2 3(ρ¨)4 5
4 4 5 5 5
      2 3ρ¨4 5
4 4 5 5 5
```

最後の2つの表現はまったく同じです。このカッコは ρ がオペランドで 2 3と4 5が引数であることを理解しやすくします。APL2自身はそれを認識

するために括弧を必要としません。

データ

APL2で使用されるデータは数値または文字のどちらかのタイプです。データは次のようにして作り出されます：

・ キーボードからの入力。
・ APL2関数と作用子の実行。
・ 共用変数、システム変数、システム関数およびシステム・コマンドの使用。

次にAPL2のデータに関して、

・ 配列
・ ランクと形
・ 変数と定数
・ ブラケット指標指定
・ 指標原点
・ 配列に、より多くの構造を加えること

を説明します。

・ 配列

APL2関数は配列と呼ばれる個々のデータ要素の集合体に作用します。配列は四角形の軸に沿って整然と並べられた要素の集合体です。配列の要素は数値、文字、または他の配列、の任意の組み合わせです。例えば、ある配列は3個の要素からなっているリストで、1番目の要素は文字、2番目は数値、3番目は文字列と言うことが可能です。この3番目の要素はそれ自体が文字のリストからなる配列です。

・ ランクと形

APL2配列のランクはその次元(dimension)または軸(axis)の数のことです。他のコンピュータ言語では、dimension という用語は格納することができるデータの量と見なしているかもしれませんが、ここではそうではありません。APL2での dimension はデータの長さではなく、

軸の数を意味しています。ここでは次元と軸とは同義語です。私たちがそれらの次元(軸)に沿ったデータの量に言及する場合は、形(shape)という言葉を使います。

例えば、数の単純なリストはわずかに1個の次元(長さだけ)を持っています。したがってランクは1です:

```
      V←2 3 5 7 11 13 17 19
      V
2 3 5 7 11 13 17 19
```

APL2ではこのようなリスト形式のデータをベクトルと呼びます。
ランク2のオブジェクトの例は数値テーブルです:

```
      M←2 5⍴⍳10
      M
1 2 3 4  5
6 7 8 9 10
```

(⍴ と ⍳ については後ほど説明します。)

APL2では、このような二次元のデータをマトリックスと呼びます。

このどちらの例でも文字データ、数値または文字のデータの混合を使用することができます。

スカラーは次元を持たず、ランク0です。APL2ではランク63までの配列をサポートします。しかしランク3や4以上のものを想像することは実際には困難ですが。

配列の形は、rho(⍴)という記号であらわされる形(shape)関数を使用することで調べることができます:

```
      V
2 3 5 7 11 13 17 19
      ⍴V
8
      A←'ABCDEF'
      ⍴A
6
```

形(shape)関数は各次元(軸)に沿った要素の数を返します。上の例の

ベクトルは一次元でしかありませんでした。マトリックスの場合は二次元ですから2個の数値を返します：

```
      N
1  2  3  4
5  6  7  8
9 10 11 12

      ρN
3 4

      M
HELLO
THERE
      ρM
2 5
```

私たちは、キーボードからベクトル入力する例を取り上げましたが、マトリックスは直接入力することができません。関数によってAPL2に望む形を伝える必要があります。通常マトリックスは、それに含まれるデータ要素をリストし、次に、希望の形を作成する reshape 関数を使用することにより作られます。

reshape 関数は形(shape)関数と同じrho(ρ)シンボルを使用しますが、形を指定する左引数をとります。(この二つの関数は密接に関係しています。一項 ρ 関数の実行結果は、二項関数で作る配列の左引数を示します。)

例えば、上に示した数値のマトリックスは次のようにして、作ることができます：

```
      N←3 4ρ1 2 3 4 5 6 7 8 9 10 11 12
```

ρ シンボルの左側に使用される数値の個数は、作り出されるオブジェクトのランクを決めます。ここでは3と4の2個の数値がランク2オブジェクト、すなわちマトリックス、を作り出します。同じように、任意のオブジェクトのランクは一項 ρ 記号の使用で返される数値の数を勘定すること、言い換えれば、形の形を測定することで見つけることができます：

```
      N←3 4ρ1 2 3 4 5 6 7 8 9 10 11 12
```

```
      N
1  2  3  4
5  6  7  8
9 10 11 12

      ρN
3 4

      ρρN
2
```

reshape 関数の右引数はどのような形式でもかまいません。上で取り上げたように直接入力された要素のリストでも、すでに名前を与えて定義したデータでもかまいません：

```
      V←2 3 5 7 11 13 17 19
      M←2 4ρV
      M
2   3  5  7
11 13 17 19

      A←3 2ρ'ABCDEF'
      A
AB
CD
EF

      B←2 4ρA
      B
ABCD
EFAB
```

ρ の右引数のランクおよび形は考える必要がありません。結果の配列では右引数のデータ要素だけがその並びの順に使用されます。右引数の余分な要素は切り落とされます。足りない場合は取り込む順番は最初に戻って必要なだけ繰り返します。

任意の形とランクを持つ配列は同じようにして設定することができます。例えば：

```
      T←2 3 4ρ'ABCDEFGHIJKLMNOPQRSTUVWX'
      T
```

```
ABCD
EFGH
IJKL

MNOP
QRST
UVWX

      ρT
2 3 4
```

この三次元の配列は、各々が3行、4列の平面を2枚持っています。3次元の配列は平面をブランク行で区切って表示されます。また、より高い次元の配列は単にこの仕組みの延長線上にあるだけです。

- 変数と定数

配列に名前を付けたものを変数と呼びます。何故ならその名前はいつでも別の値で再定義することができるからです。ランクと形の説明で用いた名前はすべて変数です。

定数はAPL2の式の中でそのまま使用される数か、数のストリング、あるいは文字か、文字のストリングです。

単独で入力された単一の数値はシステムでスカラーとして認識されます。ベクトル定数は数値をスペースで区切って順に並べることで入力することができます。

文字スカラー定数は('A' のように)文字を引用符で囲んで入力することができます。同様に文字ベクトルは('THIS IS TEXT' のように)文字列を引用符で括って入力することができます。ブランクはデータの一部とみなされますが、括りの引用符はデータの一部ではありません。最後の例はブランクが2個ありますから12文字の長さになります。

文字定数の中に引用符を含める場合は、引用符を二個続けます。したがって、CANNOT を縮めたものは'CAN''T' と入力します。APL2はこれを CAN'T と5文字で表示します。

二重引用符の場合は特別な考慮は必要でありません:

```
    'Do you know what "rank" means?'
```

```
Do you know what "rank" means?
     'No, I don''t'
No, I don't
```

・ ブラケット(角括弧)・指標指定

注:ここでの例は指標原点は省略値(=1)を用います。

配列の要素はブラケット・インデックスを用いて、選び出すことができます。例えば:

```
       V←2 3 5 7 11 13 17 19
       V[3 1 5]
5 2 11

       (2 3 5 7 11 13 17 19)[3 1 5]
5 2 11

       A←'ABCDEFGH'
       A[8 5 1 4]
HEAD

       'ABCDEFGH'[8 5 1 4]
HEAD
```

角括弧内の数値は選択されるデータの位置を指定します。これらの指標が範囲を超えた場合はエラー・メッセージか表示されます:

```
       'ABCDEFGH'[8 5 1 35]
INDEX ERROR
       'ABCDEFGH'[8 5 1 35]
       ^
```

どのような配列の要素もベクトルで示したと同様な方法で選び出すことができます。ただしその場合は各次元ごとに指標を指定する必要があります。

```
       M
2   3  5  7
11 13 17 19

       M[2;3]
17
```

```
      M[2 1;2 3 4]
13 17 19
3   5  7
```

注:スカラーは次元を持ちませんからブラケット・インデックスを用いることができません。しかし、同様な"インデックス"関数(⌷)は用いることができます。

最後の例は、ブラケット・インデックスを用いて配列の交差部分を選択する考え方を紹介しています。すべての行(セミコロンの左で指定)、すべての列(セミコロンの右で指定)が選択され指定された順番に並べられます。もちろん、どのような次元の配列の交差部分も選ぶことができます。結果配列の形は、選択に使用されるベクトルの形と対応します。

```
      ρM[2 1;2 3 4]
2 3

      T
ABCD
EFGH
IJKL

MNOP
QRST
UVWX
      T[2;1;4]
P
      T[2;1 2 3;1 2 3 4]
MNOP
QRST
UVWX

      ρT[2;1 2 3;1 2 3 4]
3 4
```

配列の各軸に沿って場所全部を指定する場合は、特別な取り決めで指標の指定を省略することができます。ただしセミコロンを省略することはできません。したがって上の最後の例は次のように略すことができます:

```
      T[2;;]
MNOP
QRST
```

```
UVWX
      ⍴T[2;;]
3 4
```

・ 指標原点

この資料の全体にわたって使用されている指標付けは指標原点 1です。何故なら各軸(次元)の最初の要素は指標 1で選択するからです。これはあたらしい作業空間の省略値です。しかし指標原点を 0 に設定して、原点 0 の指標付けを用いることもできます。指標原点はシステム変数 ⎕IO で制御されます。したがって:

```
      V←2 3 5 7 11 13 17 19
      ⎕IO←1
      V[1 2 3]
2 3 5

      ⍳3
1 2 3

      ⎕IO←0
      V[0 1 2]
2 3 5

      ⍳3
0 1 2
```

⍳ 関数はインターバルと呼ばれます。何故ならそれは連続整数を生成するからです。最初の整数は ⎕IO です。また整数間の間隔は 1 です。

APL2では、いつでも原点 1 か原点 0 を自由に選択することができます。いくつかのアプリケーションは原点 0 を使用したほうが便利なときがあります。特に数学的な操作を行うような場合はです。例えば進数変換を伴う計算の場合原点 0 なら明快です。ある種のインデックス操作自身多少きれいに表現できます。例えば:

```
      N
1  2  3  4
5  6  7  8
9 10 11 12
```

```
        ⎕IO←1
        '∘⎕'[1+N>6]
∘∘∘∘
∘∘⎕⎕
⎕⎕⎕⎕

        ⎕IO←0
        '∘⎕'[N>6]
∘∘∘∘
∘∘⎕⎕
⎕⎕⎕⎕
```

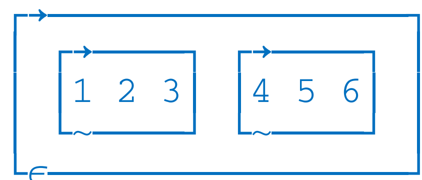
しかしながら原点 0 は時として混乱を招く恐れがあります。何故ならわれわれは数字が 0 からでなく1から始まると考えることに慣らされて育ってきているからです。[無論このどちらも正しくないです;数の始まりは負の無限値であると誰もが考えているからです。]長い間リストを 0, 1, 2 とでなく 1, 2, 3 と数えるものだと思い込まされてきています。日常

1. 家屋の番地が1から始まっている(ダウニング街0番地?)。
2. どの雑誌にもページ0がない。
3. 月の最初の日は1日である。(例外を見つけることは困難です。)

したがって、この深くしみ込んだ先入観を突き崩すことでわれわれの日常を複雑にしてしまうより、APL2は原点 1 を省略値としているのです。原点はいつでも変更できますが、あたらしい作業空間は常に省略値から始まります。

- 配列に、より多くの構造を加えること

Aという配列が二個のデータを持ち、ひとつは 1 2 3 という数値からなり、もうひとつは 4 5 6 という数値からなっていると仮定します。Aは 2 要素からなるベクトルルとして表わすことができます。その配列は次のように見えると考えることができます:



外部の箱はAを表わし、2個の要素を持っています。2個の要素はそれぞ

れ3個の要素を持つベクトルです。APL2では配列の要素は配列であってもよいのです。配列が他の配列を要素として持つ場合入れ子配列(Nested Array)と呼びます。APL2ではそのような配列を作成することは非常に容易です。カッコで複数の要素を囲えばよいのです
:

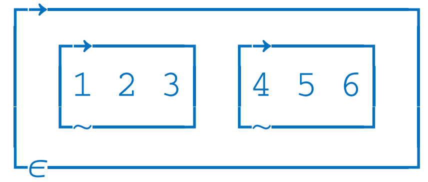
```
      MORE←(1 2 3)(4 5 6)
       ρMORE
2

      MORE
1 2 3  4 5 6
```

ρMORE は、MORE の形が2であること示します;APL2は入れ子配列を画面上に表示するときに、各入れ子要素配列の頭にブランクをはさんで表示します。APL製品で提供される公共ライブラリーに"DISPLAY" という作業空間があり、そこに "DISPLAY" という定義関数があり、それを使用するとボックス形式で入れ子配列を表示することができます;
(注:DISPLAY関数は現行APLシステムでは外部関数としても提供されていますから、

```
     3 11 ⎕NA 'DISPLAY'
```

を実行することによっても使用することができます。)

```
      DISPLAY MORE
```



配列の入れ子階層の程度を深さと呼びます。単純スカラーは深さ 0 です。単純ベクトル(高次元な単純配列も)は深さ 1 です。単純とはその要素が全てスカラー、すなわちひとつの数値、またはひとつの文字、であることです。上記MOREの深さは 2 です。深さ 2 とはその要素の少なくとも一個は深さ 1 でまたそれ以上大きな深さを持った要素がないことを意味しています。

APL2には配列の深さを示す深さ関数(≡)が提供されています。

```
      ≡1 2 3
1

      ≡MORE
2
```

下に示すマトリックス“M”は数値テーブルに文字見出しと、 0 の値を 'NONE' という文字列で置き換えている入れ子配列の例です。このマトリックスは5行3列の配列です。第1行目の各要素は文字ベクトルです。また第1列目の各要素も文字ベクトルになっています。最後の行、最後の列も文字ベクトルです。Mの深さは2です。

```
      M
FOOD CALORIES PROTEIN
milk      160       9
apple      60       1
bread      75       2
jelly      50    NONE
```

<u>名前に値を設定すること</u>

左向き矢印は名前に値を設定するのに使用します。この資料全体にわたって示したとうり、この矢印は通常、式の左端に近く、その左側には名前だけが置かれる形で使用されます。

```
      HYPOTENEUSE←((LEG1⋆2)+(LEG2⋆2))⋆.5
```

このように使用された時、APLの式は表現を解釈し、結果を表示する代わりに、名前に収めます。

しかし、設定矢印(左向き矢印はしばしばこのように呼ばれますが)は式の中途で使用することも可能です。この場合はその名前に与えられた値はそれに続く処理で使用することができます。例えば、

REPORT_SALES と呼ぶ関数があって、引数に地域番号をとるとします。また REGION_FOR と呼ぶ関数があり、それは都市名を取り、地域番号を返す機能があるとします。これらの関数を次のように使用することができます:

```
      REPORT_SALES REGION_FOR 'Chicago'
```

しかし地域番号をその他の式でも利用する場合は、次のようにすることもできます:

```
        REGION←REGION_FOR 'Chicago'
        REPORT_SALES REGION
```

多くのAPLユーザは式の左端で使用される名前に関して、できるだけ式を簡単にしたいと思ってます。しかし、上の式は一つにまとめることができます:

```
     REPORT_SALES REGION←REGION_FOR'Chicago'
```

このステートメントで、REGION_SALES が最後に実行される部分ですが結果を名前に収めていませんので、もし結果値を生成する関数であれば、それは画面上に表示されます。

設定矢印はひとつの式の中で何回でも使用することができます。APL2式中の中間結果は、任意の表現の左側に ⎕← を挿入することにより表示することができます。例えば:

```
       A←2+⎕←3×B←4
12

       A
14

        B
4
```

ここまでカバーされたケースは単純な設定(あるいは単純な割り当て)と呼ばれます。その理由は設定矢印でできるその他のいくつかことがらを見れば明白になります:

[インデックス付き設定]
[ベクトル設定]
[選択的設定]

[インデックス付き設定]

設定矢印の左側のが右ブラケットである場合、ブラケットの処理が最初に

実行されます。次に設定矢印の右側の値がブラケット指標で選択された要素を置き換えます。

```
      A←1 2 3
      A[2]←4
      A
1 4 3

      A[2 3]←5 6
      A
1 5 6

      A[2 3]←2
      A
1 2 2
```

ブラケット操作では単一の要素か複数の要素を選ぶことができます。複数の要素を選んだ場合、その形は右側で作られるものと合致しなければなりません。すなわち、ランクと次元とが一致しているか、もしくは一個の要素でなければなりません。一個の要素(スカラー)は最後の例で示すとうり、繰り返されて左側で指定された形になります。

[ベクトル設定]

ひとつの設定矢印で複数の名前にベクトルの要素の値を設定することができます:

```
      (A B)←14 4
      A
14
      B
4
```

ベクトルの形は割り当てられている名前の数に合致しなければなりません;したがって、通常は、その表現はベクトルでなければなりません。しかし、APLがその規則をさらに緩める1つのケースがあります:

```
      (MY THREE VARIABLES)←0
      MY
0
      THREE
0
```

(このややっこしい名前の選択について話をしましょう。)

これの説明は、設定矢印の右側の表現がベクトルでなくスカラーであるということです。スカラーは形を持ちません。これはAPLでスカラーの拡張と呼びます。形をもっていませんので、必要に応じてどのような形にもなりうるのです。

右側の表現についても単純ベクトルである必要はありません。任意の入れ子配列でも許されます:

```
  (ADR¯NUM ADR¯STREET)←555 ('Bailey' 'Avenue')
   ADR¯STREET
Bailey Avenue

      ⍴ADR¯STREET
2
```

[選択的設定]

設定矢印の左側に右括弧がある場合、ペアになった括弧内の表現はそれが実行された結果指し示す変数の場所に設定矢印の右側の実行結果の値を設定します。これを選択的設定と呼びます。使用可能な関数は:

一項:↑ ⍉ ,   ⌽ ,[X]   ⌽[X]
二項:[L]   ⊃ ⍴ ⍉ ↑ ↓ ▯ ⌽ ↑[X]   ↓[X]   ▯[X]
⌽[X]

派生関数:LO\   LO/ LO\[X] LO/[X]

選択的設定は通常配列の選択された部分の要素の置き換えに用いられます。この場合置き換えるデータと置き換えられるデータの形が一致してなければなりません。しかし置き換えるものと、置き換えられるものの対応する要素間では形、次元、深さは一致している必要はありません。通常、選択的設定は設定矢印の左側の表現を単独で実行した場合の状態を見れば理解しやすいです。 例えば:

```
      V←10 20 30 40
      2↑V
10 20
```

```
      (2↑V) ←100 200
      V
100 200 30 40
```

この場合、↑(Take)関数は V の頭の二つの要素をとるのではなく、V の頭の二つの場所を選択します。この選択結果としての場所のベクトルは入れ子構造の非常に深いところを指す場合でも単純ベクトルとなります。設定矢印の右側のデータはそのベクトルで示される場所のデータを置き換えます。

以下マトリックス“M”を使ってさまざまな選択的設定の例を示します。Mは最初に設定したものを使うものとします。

```
      M←3 4⍴'ABCDEFGHIJKL'
      M
ABCD
EFGH
IJKL

      (2 3⌷M)←'⎕'
      M
ABCD
EF⎕H
IJKL

      (1(2 3)⌷M)←'∇⋆'
      M
A∇⋆D
EF⎕H
IJKL

      (2 1↑M)←'⊖⌹'
      M
⊖∇⋆D
⌹F⎕H
IJKL

  (,M)←⍳12
      M
1  2  3   4
5  6  7   8
9 10 11  12
```

```
      M←3 4⍴'ABCDEFGHIJKL'
      (4↑,⍉M)←'○⋆÷⎕'
      M
○⎕CD
⋆FGH
÷JKL
```

上記最後のものは、選択的設定における複数の機能の応用例です。置き換えられた場所は、Mを転置(行と列を置き換える)し、行に沿っての最初の4個所です。選ばれた AEIB の場所がそれぞれ ○⋆÷⎕ で置き換えられます。

次の例ではスカラーが非スカラー配列として指定された場所に繰り返され設定されます。

```
      M←3 4⍴'ABCDEFGHIJKL'
      ((1 3)(1 4)⌷M) ←'⋆'
      M
⋆BC⋆
EFGH
⋆JK⋆
```

＊制限

Bが共用変数(他のプロセサーと共用される変数)とした場合、

```
      ((B=' ')/B)←'⋆'
```

はエラーです。何故なら共用変数が二度指定されていますから、左のBは右のBとは同一になりません。(注:共用変数は一回アクセスするたびに相手のプロセサーによって値が変更される可能性があります。)

解釈の順序

APL2では、式の解釈の順序は後ほど述べるような小数の変更指定がありますが、右から左に行われます。特に、乗算が加算に優先するというような、関数間での順位付けはありません。全ての関数は同列に扱われます。もしあなたが他の言語を使った経験があれば、これは本当に救いであるといえます。(例えばC言語では実行優先順位の階層が15段もあります。)

APL2が提供する非常に完全な原始関数群間で、実行順位をつければ

大きな混乱を招きます。それだけではありません、APL2では自分固有の関数を定義することができ、それらは原始関数と区別なく取り扱われます。あなたは、あなた自身が書くプログラムをあなた自身か別の人がかって書いたプログラムと比較して優先順位をいちいち決めなければならないとしたらどうでしょうか？

したがって、APLでは 2×3+4 が 14 （右から左、除算は優先しない）ということにショックを受けたとしても、次の例を見れば、その心配は吹き飛ぶと思います。

```
2 F00 3+4
```

では足し算は関数F00を実行する前に実行されますが、

```
2+3 F00 4
```

では足し算が最後に実行されます。

[規則の例外]

ここで、右から左への実行順番を変化させる変更指定について説明します。実行の優先順位は：

1. 括弧
2. ブラケット表記
3. 設定の対象
4. ベクトル表記
5. 作用子オペランド結束

基本的な規則すなわち<u>正規の関数実行順位</u>がこれに続きます。

1. 括弧

括弧は式の解釈の順序を制御するのによく使われる方法です。括弧の外の関数を作用させる前に、括弧の中が先に解釈されます。もし解釈の結果が配列、関数または作用子であり、対になっていさえすれば、括弧はいつでも使用することができます。括弧は何重に使用してもその制限はありません。

関数の左側の引き数がひとつのものとして解釈されるために、括弧がしばしば使用されます。例えば (7−3)×2 は 8 ですが、7−3×2 は 1 です。

括弧は関数の右側の引数に対しては、APLの正常な解釈の順番に従っているという意味で、括弧をつける必要がありません。このためにおもなデータが右にくるように気をつけてAPLの関数を設計すれば、APLはほとんどの他の言語より括弧の数を減らすことができます。

2. ブラケット表記

ブラケットは非常に有用な記号ですが同時にAPLの他の構文とは完全には適合しません。ブラケットは括弧と同様にその中で指定された表現の解釈が完全に優先されます。しかしブラケットは括弧と異なり左ブラケットの左にくるものによって関数または作用子として扱われます。また左側にくる関数またはデータと強い結合力を持ちます。

関数としてのブラケットについては、ブラケット指標を参照してください。作用子としてのブラケットは、特定の原始および派生関数に限定していますから、"軸指定"の変化形式として、それらの関数またはそれらを派生させる作用子のところで説明します。

ブラケット表記の左側のものとの強い結びつきは括弧を除いた他のいかなる構成よりも優先度が高いです。

```
      A←1 2 3
      A[2] ←4
      A
1 4 3
      A A[2]
1 4 3 4

      1 1 2 3 5 8[4]
RANK ERROR
      1 1 2 3 5 8[4]
     ^

       (1 1 2 3 5 8)[4]
3
```

3. 設定の対象

通常は、設定矢印の左側は変数の名前、あるいはこれから変数の名前になる未使用の名前です。左矢印は設定(Specification または Assignment)と呼ばれます。左側の名前と右側で指定した表現の解釈を関連付けます。設定は正式には関数でも作用子でもありませんが、右側の表現で作られる値を、あたかも関数の結果のように、あとでAPLの式の中で使用できるようにします。(しかし、それは値が要求されたときのみ行われます。本当の関数と異なり、設定が式の一番左にきた場合、その値は画面上に表示されません:シンタックス・エラーになります。)

二つの構成がこの通常の働きを変化させます。

- 設定矢印の左が右ブラケットなら、指標付き設定が実行されます。
- 設定矢印の左が右括弧なら、選択的設定が実行されます。

いずれの場合もその値が同じ式でさらに使用される場合は、左側で指定された変数ではなく、右側の値が続けて使われることに注意してください。

```
      A←1 2 3
      2+A
3 4 5

      2+A[2]←10
12

      B←'HELLO'
      2+(A B)←1 2
3 4

      A
1

      B
2
```

4. ベクトル表記

スペースだけによって分けられた一連の値の表現は、ベクトルとして扱われます。いくつかの例をあげます:

```
      2 3 4
      'A' 'B' 'C'
      1 (2 3) 4
```

上の最後の例での (2 3) はひとつの値の表現ですが、残りの表現と合わさって、三要素の入れ子配列を作ります。その考え方をさらに拡張した例を紹介します:

```
      2 'ABC'
      1 (2+3) 4
      AA←2 3
      1 AA 4
```

この最後の例は前のグループの最後の例とまったく同じ値を持っていることに注意してください。

他のAPL表現のように、ベクトル表記内の要素は、右から左に解釈されます:

```
      A←1
      A (A←2) A
2 2 1
```

しかしベクトル表記によるベクトルの形成はその右側の関数または作用子の実行よりも優先します。

```
      1 2 3 + 4 5 6
5 7 9
      A←1
      3 A + 1 1
4 2
```

ベクトル記法は設定には優先しません。

```
      A←1
      A B←2 3
1  2 3
      A
1
      B
2 3
```

```
      (A B) ←2 3
      A
2
      B
3
```

また、ブラケット指標にも優先しません。

```
      A←1 2 3
      A A[2]
1 2 3 2
```

5. 作用子オペランド結束

［一項作用子］

ほとんどの作用子は一項形式です；すなはち左側に一個オペランドをとります。ここではほとんど混乱は生じません。APL2は右から左に実行され、作用子にぶつかるとその左のオペランドを見つけます。

・ もし配列が見つかったら、まず最初にベクトル表記のルールを当てはめます。そして結果しての配列をオペランドとして使用します。
・ もし関数が見つかったら、それがオペランドになります。
・ もし他の作用子が見つかったら、APL2はその作用子で派生関数を作り、その派生関数をオペランドとして使用します。

例をあげて最後のケースを説明します。Each (¨)作用子はオペランド関数をデータ引数の一番上の階層のそれぞれの要素に適用します。低減(/)作用子はオペランド関数をデータ引数の一番上の階層の隣同士の要素の間に適用します。したがって：

```
      ×/2 3 ¯1
¯6

      ×¨2 3 ¯1
1 1 ¯1
```

第1のケースでは、×は各要素の間に挿入され、掛け算使用されています：

```
      2×3×¯1
```

```
¯6
```

第2のケースでは、× は各要素に適用されます。したがって、それは値の符号返す、一項関数になります:

```
      (×2)(×3)(×¯1)
1 1 ¯1
```

(この場合は ¨ なしでも同じ結果になります。スカラー関数を参照。)

これまでは背景を説明しましたが、これから本題の例に入ります:

```
      ×/¨(2 3 ¯1)(2 3 4)
¯6 24
```

右から左へと見てきて、APL2は2つのベクトルを作り、それを合わせてひとつの入れ子配列とします。次に ¨ 作用子が見つかりますから、その左を見ます。そこで / が見つかり、それがまた作用子ですのでオペランドでないと理解します。さらに左を見て派生関数 ×/ 、すなわち乗算低減が形成されることを見つけだします。そこで初めてその派生関数を ¨ のオペランドとして使えるようになります:

```
      (×/2 3 ¯1)(×/2 3 4)
¯6 24
```

[二項作用子]

APL2には二項原始作用子が一つあります。配列積作用子ですが、ユーザーは二項作用子を定義して追加することができます。配列積作用子は、左右の関数を指定することで、無数の内積派生関数の変化形を作ることができます。元来の数学におけるこの用語に対応した特定の内積派生関数は +.× で、これは左と右の値の組み合わせの積を取り、それらの和を求めます。

```
      A←2 3⍴⍳6
      B←3 4⍴⍳12
      A
1 2 3
4 5 6

      B
```

```
1  2  3  4
5  6  7  8
9 10 11 12
```

Aの第1行にBの最初の2列をそれぞれ掛け合わせると:

```
      +/1 2 3 × 1 5 9
38
      +/1 2 3 × 2 6 10
44
```

+.× はAの各行とBの各列と全てに関して同じことを実行します:

```
      A+.×B
38 44  50  56
83 98 113 128
```

ここで注目しなければならないポイントは、何故このような奇抜な機能(実際にはすごく役に立つものですが)を必要とするのかということではなく、何故APL2は式のシンタックスをこのように分析するのかということです。右端の表現 ×B に注目してください。それはAPL2で次のように解釈できます:

```
      ×B
1 1 1 1
1 1 1 1
1 1 1 1
```

特別面白い結果ではありませんが、Bの各要素が正であることを告げます。

しかし、APL2はこの表現をこのようには解釈しません。これはオペランドが右から左に関数を処理することよりも優先することを示す最初の実際の例です。APL2は、右から左に式を眺め × に遭遇した時に、とりあえずその先の左引数を探します。すなわち、× が一項符号判定関数か二項乗算関数かを判定しようとします。そこで二項作用子の . を見つけ、× がオペランドとして使用されていることを理解します。

これを一歩進めてみます。/ は前にも使いましたが、一項作用子です。+.×/ をAPL2はどのように解釈するでしょうか? ×/ は派生関数ですから +. の右オペランドであるかもしれませんが、しかし +.× も派生関

数で、/ の左オペランドであるかもしれません。答えは二項作用子の右オペランドの結合度は左オペランドより非常に強いということです。すなわち、

```
+.×/        は
(+.×)/      と同じで、
+.(×/)      と同じでありません。
```

エラー

実行することができない式を入力すると、エラー報告が発せられます。APL2の初心者はエラーを引き起こす入力をタイプすることを必要以上に心配してしまうことがあります。これらのエラー報告は言語を習得するにあたって最も役に立つ手助けとなります。APL2のエラー報告は明快で、簡潔で、正確であるように設計されています。

エラーを発生させないようにできる限りの事をするのではなく、わざと沢山のエラーの条件を試してみることは役に立つと思います。これはさまざまな関数がどのように定義されているかを学習するよい方法です。使いながら学ぶことは常に望ましいことです。してはいけないことを何か入力してしまうことを心配しないように。あなたが入力したもので機械が壊れることはありません。あなたは完全に自由に実験することができます。

APL2のエラー報告はエラーの性格を示し、またどこでエラーが起き、どこで実行が止まっているかを示す山印を表示します。たとえば:

```
      B←1 2 3 + A←4 5
LENGTH ERROR
      B←1 2 3+A←4 5
       ^    ^
```

これらのエラー報告から豊かな情報を得ることができます。このメッセージがいったい何をわれわれに告げようとしているか見てみましょう:

```
      B←1 2 3 + A←4 5       これがはあなたがタイプした行です。
LENGTH ERROR                 2つの長さが一致していません。
      B←1 2 3+A←4 5          あなたの入力行はこれです。*
       ^    ^      解釈が進行した場所とエラーの発見された場所
```

（＊入力された行の不必要なブランクは取り除かれています。）

コードを示す行の下に、山印が2個表示されるのが典型的な形です。左
側の山印はAPL2が右から左の行の走査をどこまで進めたかを示します。
（ここで、システムは　１　２　３　が数値ベクトルで左引数として取り扱われ、
設定矢印のところでとまりました。設定はまだ実行されていません。）右の
山印は実際のエラーが発見された場所を示します。通常、それはエラー
が発見されたときにAPL2がどの関数を解釈していたかを示します。この
例では ＋ 関数の引数が互いに一致していません、したがって要求され
た加算が実行できません。

山形が重なるときがありますが、そのときはひとつしか表示されていないよ
うに見えます。

APL2配列の補足説明：

APL2はすべてのデータを配列として取り扱います。APL2の配列は必ず次の5個の属性を持ちます：

1. 値（VALUE）
2. 領域（DOMAIN）. . . . . 文字領域か数値領域
3. 次元（DIMENSION）. . . . . . . 軸の本数
4. 軸ごとの長さ（LENGTH）
5. 深さ（DEPTH）

このうち"深さ"（DEPTH）属性はAPL2になってはじめて導入された概念で、APL2を強く特徴付けるものであり、それまでの単純配列（SIMPLE ARRAY）のみの世界から、複合配列（COMPLEX ARRAY）を含む汎用配列（GENERAL ARRAY）の概念を確立しました。APL2における汎用配列の定義によれば、深さ0または1の配列は単純配列で、深さ2以上の配列は複合配列と呼ばれます。

汎用配列の概念は一般の古くからのAPLユーザーにとても時として多少複雑で理解しにくい面があります。したがってAPL2では個々の配列の属性を視覚的にわかりやすく表示するためのツールを提供していて、ユーザーはいつでも使うことができ、理解を助ける手立てとなります。

以下DISPLAYを用いて、APL2の配列の考え方を、図で表示しながら解説します：

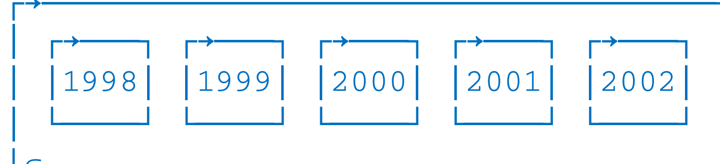
単純配列（Simple Array）：

```
      DISPLAY 1
1                        ⍝ 値:1、領域:数値、次元:0、長さ:なし、深さ:0

      DISPLAY '1'
1                        ⍝ 値:'1'、領域:文字、次元:0、長さ:なし、深さ:0

      DISPLAY 100
100                      ⍝ 値:100、領域:数値、次元:0、長さ:なし、深さ:0

      DISPLAY '100'
┌→──┐                   ⍝ 値:'100'、領域:文字、次元:1、長さ:3、深さ:1
│100│                    ⍝ 長さを持つ配列には枠が付く。
└───┘                    ⍝ → は横軸が長さを持つことを示す。

      DISPLAY 1 0 0
┌→────┐                 ⍝ 値:1 0 0、領域:数値、次元:1、長さ:3、深さ:1
│1 0 0│                  ⍝ ~ は数値配列を示す。
└~────┘                  ⍝   要素の値がスペースで区切られる。
```

```
      DISPLAY '100',1 0 0
┌→──────────┐                  ⍝ 値:'100',1 0 0、領域:混合
│100 1 0 0 │                  ⍝ 次元:1、長さ:6、深さ:1
└+──────────┘                  ⍝ + は混合配列を示す。

      DISPLAY 'AB C',1 0 0
┌→───────────┐                 ⍝ 値:'AB C',1 0 0、領域:混合
│AB C 1 0 0 │                 ⍝ 次元:1、長さ:7、深さ:1
└+───────────┘

      DISPLAY 2 3ρι6
┌→────┐
↓1 2 3│                        ⍝ 値:ι6、領域:数値
│4 5 6│                         ⍝ 次元:2、長さ:2 3、深さ:1
└~────┘                        ⍝ ↓ は縦軸が長さを持つことを示す。

      DISPLAY 2 3ρ'123456'
┌→──┐
↓123│                          ⍝ 値:'123456'、領域:文字
│456│                          ⍝ 次元:2、長さ:2 3、深さ:1
└───┘                          ⍝ 要素の値はスペースで区切られない。

      DISPLAY DT←?6 5ρ10000
┌→───────────────────────┐
↓1316 7557 4587 5328 2190│
│ 471 6789 6793 9347 3836│    ⍝ 値:DT、領域:数値
│5195 8310  346  535 5298│    ⍝ 次元:2 長さ:6 5、深さ:1
│6712   77 3835  669 4175│
│6868 5890 9305 8462 5270│
│ 920 6540 4160 7012 9104│
└~───────────────────────┘

      DISPLAY 2 3 4ρι24
┌┌→──────────┐
↓↓ 1  2  3  4│ ⍝ 値:ι24、領域:数値、次元:3 長さ:2 3 4、深さ:1
││ 5  6  7  8│
││ 9 10 11 12│ ⍝ ↓ の本数は縦軸以上の長さ(0 を除く)がある
││           │ ⍝   軸の数に対応する。
││13 14 15 16│
││17 18 19 20│
││21 22 23 24│
└└~──────────┘
```

複合配列 (Complex Array):

```
DISPLAY CH←'1998' '1999' '2000' '2001' '2002'
```



⍝ 値:CH、領域:文字
⍝ 次元:1、長さ:5、深さ:2
⍝ ∊ は複合配列を特徴付
⍝ ける入れ子構造を示す。

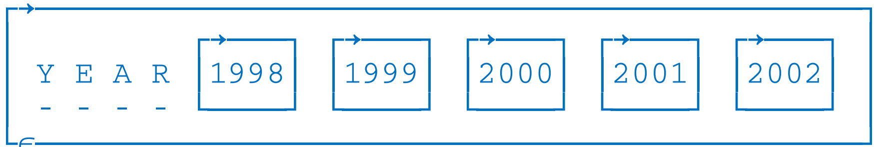
```
LH←'TOKYO' 'OSAKA' 'NAOYA' 'YOKOHAMA' 'FUKUOKA'
LH←LH, ⊂'SAPPORO'
DISPLAY LH
```

⍝ 値:LH、領域:文字、次元:1、長さ:6、深さ:2

```
DISPLAY 'YEAR'
```

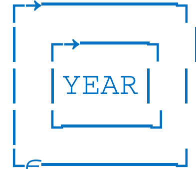
⍝ 値:'YEAR'、領域:文字、次元:1、長さ:4、深さ:1
⍝ これは複合配列にならない点に注意

```
DISPLAY 'YEAR',CH
```



⍝ 値: 'YEAR',CH、 領域:文字、 次元:1、 長さ:9、 深さ:2

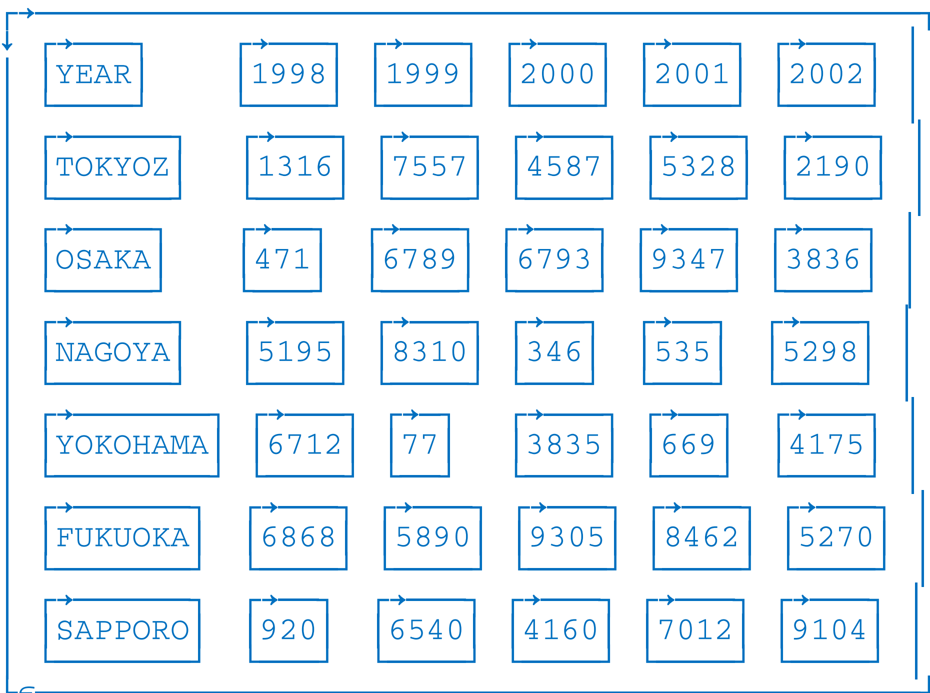
```
DISPLAY YR←⊂'YEAR'
```



⍝ 値: ⊂'YEAR'、領域:文字、次元:0、長さ:なし、深さ:2
⍝ 中身長さ4の文字ベクトルを含む次元 0
⍝ したがって長さなしのスカラー。

```
      CT←(YR,CH),[1]LH,DT

      CT
YEAR     1998 1999 2000 2001 2002
TOKYO    1316 7557 4587 5328 2190
OSAKA     471 6789 6793 9347 3836
NAOYA    5195 8310  346  535 5298
YOKOHAMA 6712   77 3835  669 4175
FUKUOKA  6868 5890 9305 8462 5270
SAPPORO   920 6540 4160 7012 9104

      DISPLAY CT
┌→──────────────────────────────────────────────────────────────┐
↓ ┌→───┐        ┌→───┐   ┌→───┐   ┌→───┐   ┌→───┐   ┌→───┐    │
│ │YEAR│        │1998│   │1999│   │2000│   │2001│   │2002│    │
│ └────┘        └────┘   └────┘   └────┘   └────┘   └────┘    │
│ ┌→─────┐                                                    │
│ │TOKYOZ│       1316     7557     4587     5328     2190     │
│ └──────┘                                                    │
│ ┌→────┐                                                     │
│ │OSAKA│        471      6789     6793     9347     3836     │
│ └─────┘                                                     │
│ ┌→─────┐                                                    │
│ │NAGOYA│       5195     8310     346      535      5298     │
│ └──────┘                                                    │
│ ┌→───────┐                                                  │
│ │YOKOHAMA│     6712     77       3835     669      4175     │
│ └────────┘                                                  │
│ ┌→──────┐                                                   │
│ │FUKUOKA│      6868     5890     9305     8462     5270     │
│ └───────┘                                                   │
│ ┌→──────┐                                                   │
│ │SAPPORO│      920      6540     4160     7012     9104     │
│ └───────┘                                                   │
└∊──────────────────────────────────────────────────────────────┘
⍝ 値:CT、 領域:混合、 次元:2、 長さ:7 6、 深さ:2
⍝ この配列の数値の部分(CT[1+⍳6;1+⍳5]) は
⍝  計算など直接数値処理の対象となる。
```

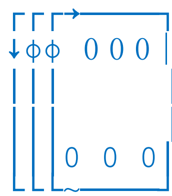
```
      CT←(YR,CH),[1]LH,⍕¨DT
      CT
YEAR     1998 1999 2000 2001 2002
TOKYO    1316 7557 4587 5328 2190
OSAKA    471  6789 6793 9347 3836
NAOYA    5195 8310 346  535  5298
YOKOHAMA 6712 77   3835 669  4175
FUKUOKA  6868 5890 9305 8462 5270
SAPPORO  920  6540 4160 7012 9104

      DISPLAY CT
```




```
⍝ 値:CT、 領域:文字、 次元:2、 長さ:7 6、 深さ:2、
⍝ APL以外のシステムとのデータの受け渡し、印刷出力、外部記憶装置出力
⍝ する場合など文字化する必要がある。
```

Null （長さ 0 の軸を含む配列）:

```
      DISPLAY 0⍴0
┌⊖┐
│0│  ⍝ 値:無、領域:数値、次元:1、長さ:0、深さ:1
└~┘  ⍝ ⊖ は横軸の長さが0であることを示す。

      DISPLAY ⍳0
┌⊖┐
│0│  ⍝ 値:無、領域:数値、次元:1、長さ:0、深さ:1
└~┘  ⍝ ⊖ は横軸の長さが0であることを示す。

      DISPLAY 0⍴' '
┌⊖┐
│ │ ⍝ 値:無、領域:文字、次元:1、長さ:0、深さ:1
└─┘

      DISPLAY 0 0 0⍴0
┌┌┌⊖┐
⌽⌽0│  ⍝ 値:無、領域:数値、次元:3、長さ:0 0 0、深さ:1
└└└~┘  ⍝ ⌽ は縦軸以上の軸の長さが0であることを示す。

      DISPLAY 2 0 0 3⍴1 2 3
```




```
⍝ 値:無、領域:数値、次元:4、長さ:2 0 0 3、深さ:1
⍝ ⌽ は縦軸以上の軸の長さが0であることを示す。

      DISPLAY 2 0 0 3⍴'ABC'
┌┌┌→──┐
↓⌽⌽   │  ⍝ 値:無、領域:文字、次元:4、長さ:2 0 0 3、深さ:1
│││   │  ⍝ これら値の無い配列をナル(null)と呼ぶが、このような高次元のnull
│││   │  ⍝ を必要とする場面はほとんど無く、処理の過程で不注意で、発生さ
│││   │  ⍝ せてしまうこともある。たとえば高次元のデータ配列におとし( ↓ )処
└└└───┘   ⍝ 理をするような場合である。
```

註: nullそのものは条件分岐式を組み立てたり、計算にnullを作用させて結果をnull化するなどさまざまな場面で活用されるもので、APLでは非常に重要な概念であるが、ここではこれ以上は取り上げない。ちなみにスカラーは長さがnullの配列を指す。