J プログラミング言語入門

ロジャー・ストークス
Roger Stokes
jsoftware.com
2015 年 6 月 15 日改訂

J を学ぶ

日本語版･前編
第 1 章～16 章
2017/10/21

日本語翻訳版
基本翻訳 Google 翻訳
JAPLA-J 言語研究会翻訳校正
2017 年 10 月 21 日 B5 版印刷

著者：**Roger Stokes** 氏の言葉

J ソフトウェアおよびドキュメントは、J SoftWare ホームページで入手できます。
この本は、ここからさまざまな形式でも入手できます。
コメントと批判は J フォーラムに送ってください。

この本について

この本は、読者がコンピュータプログラミング言語 J.を学ぶのを助けるためのものです。
本書は初心者プログラマーから経験豊富なプログラマーまで幅広い読者層に役立つことが期待されています。プログラマーの初めは、あらゆるステップで例を見つけるでしょう。例を非常に簡単にし、一度に 1 つの新しいアイデアを紹介するように注意します。経験豊富なプログラマーは、J 表記の根本的な単純さとパワーを高く評価することができます。
本の範囲は、J Dictionary で定義されているコア J 言語です。コア言語の範囲は比較的完全であり、(最終的には）ほとんどの辞書を網羅しています。
したがって、本書では、J ユーザーガイドに記載されているグラフィック、プロット、GUI、データベースなどのトピックについては触れていませんし、J アプリケーションライブラリについても触れていません。本書の目的が何であるかを明確にすべきです。すなわち、プログラミングの基礎を教えることも、J を Vehicle として使って数学や他の科目のアルゴリズムやトピックを研究することも、決定的な参考資料を提供することもしていません。
この本は次のように編成されています。パート 1 は、さまざまなテーマに触れる基本的な紹介です。目的は、第 1 回の終わりまでに J 言語の概要と一般的な感謝を読者に提供することです。第 1 回で紹介したテーマは、本書の残りの部分でより深く詳細に展開されています。
すべての例は J701 以降で実行されています。

謝辞
私は励ましと貴重な批判と示唆のための以前のドラフトの読者に感謝しています。

参照 WEB: http://www.jsoftware.com/help/learning/contents.htm#toc

---

目次

第 1 章：Basics
基本
1.1 Interactive Use
インタラクティブな使用
ユーザーはキーボードで行を入力します。この入力行は、2 + 2 のような式であってもよい 。行が入力されると(「Enter」または「キャリッジリターン」キーを押すことによって）、式の値が計算され、次の行に表示されます。

```
   2 + 2
4
```

次に、ユーザーは別の入力行の入力を求められます。プロンプトは、カーソルが左マージンから数スペースの位置にあることによって表示されます。したがって、この本では、いくつかのスペースでインデントされた行は、ユーザーによって入力された入力を表し、インデントされていない次の行は、対応する出力を表します。

1.2 Arithmetic
算術
乗算の記号は *(アスタリスク）です。

```
   2* 3
6
```

これをもう一度試してみると、今度は 2 スペース * 3 を入力する

```
   2 * 3
6
```

結果は以前と同じで、ここのスペースはオプションであることがわかります。スペースでは、式をより読みやすくすることができます。分割の記号は%(パーセント）です。

```
   3 % 4
0.75
```

減算のために、わかりやすい-記号があります：

```
   3  -  2
1
```

次の例は、負数がどのように表されるかを示しています。負の記号は先頭の（アンダースコア）記号で、記号と数字の数字の間にスペースはありません。この記号は算術関数ではなく、小数点が記法の一部であるのと同じ方法で、数値を書き込む表記の一部です。

```
   2  -  3
_1
```

否定の記号は-であり、減算と同じ記号である：

```
   - 3
_3
```

power(べき乗)関数の記号は^(caret）です。2 立方は 8：

```
   2 ^ 3
8
```

数値の 2 乗を計算する算術関数には、* :(asteris colon）という記号があります。

```
   *: 4
16
```

1.3 Some Terminology: Function, Argument, Application, Value

いくつかの用語：関数、引数、アプリケーション、値

2 * 3 などの式を考えてみましょう。乗算関数 * がその引数に適用されると言う。左の引数は 2、右の引数は 3 です。また、 2 と 3 は引数の値と呼ばれます。

1.4 List Values

リスト値

時々、いくつかの異なる数に対して同じ計算を何回か繰り返すことがあります。数字のリストは、たとえば、各数字と次の数字の間にスペースを入れて 1 2 3 4 とすることができます。このリストの各数字の 2 乗を求めるには、次のように言うことができます：

```
   *: 1 2 3 4
1 4 9 16
```

ここでは、「Square」関数(*: ）がリストの各項目に個別に適用されることがわかります。+ などの関数に 2 つのリスト引数が指定されている場合、関数は対応するアイテムのペアに別々に適用されます。

```
   1 2 3 + 10 20 30
11 22 33
```

1 つの引数がリストで、もう 1 つの引数が単一の項目である場合、単一の項目は必要に応じて複製されます。

```
   1 + 10 20 30
11 21 31
   1 2 3 + 10
11 12 13
```

新しい関数を見ているときに、引数リストのパターンがどのように結果リストにパターンを生み出すかを確認することが役立つことがあります。
例えば、7 を 2 で割ったとき、商は 3 で あり、残りは 1 であると言うことができる。剰余を計算するビルトイン ⌹ 関数は| (垂直バー）、「残留物(Residue）」機能と呼ばれます。引数と結果のパターンは、次のように表示されます。

```
   2 | 0 1 2 3 4 5 6 7
0 1 0 1 0 1 0 1
   3 | 0 1 2 3 4 5 6 7
0 1 2 0 1 2 0 1
```

(7 mod 2）ではなく(2 | 7） を書くことを除いて、残余関数はおなじみの "mod"または "modulo"です。

1.5 Parentheses

括弧

式は括弧を含むことができ、通常の意味を持ちます。括弧の中身は実際には別個の小さな計算です。

```
   (2 + 1) *(2 + 2)
12
```

ただし、括弧は必ずしも必要ではありません。J の式を考えてみましょう: 3 * 2 + 1。それは、(3 * 2) + 1、つまり 7 か、それとも 3 *(2 + 1) 、つまり 9 かを意味しますか？

```
   3 * 2 + 1
9
```

学校の数学では、表現を書くための慣例や規則を学びます。加算する前に乗算を行う必要があります。J のルールのポイントは、書く必要がある括弧の数を減らすことです。
加算前の乗算などのルールは J にはありません。もし必要ならばいつでも括弧を書くことができます。しかし、上記の 3 * 2 + 1 の例が示すように、括弧節約規則が J にあります。ルールは、括弧がない場合、算術関数の右の引数はすべて右のものであるということです。このようの場合には 3 * 2 + 1 の右引数 * がある 2 + 1。もう一つの例があります:

```
   1 + 3 % 4
1.75
```

我々は、% が前に適用されて + は、つまり、右端の関数が最初に適用されることを、見ることができます。
この「右端の最初の」ルールは、「加算前の乗算」という共通の規約とは異なるが、同じ役割を果たします。これは単なる利便性に過ぎず、無視して括弧を書くことができます。そのメリットは、J で、数値を使った計算のために(100 のような）多くの関数があり、それ以前にどの関数を適用すべきかを覚えようとするのは難しいということです。
この本では、 "右端の最初の"規則で必要とされない括弧を持つ式を時折表示します。これを行う目的は、表現の構造を強調して、表現の読みやすさを高めることです。

1.6 Variables and Assignments
変数と代入

英語表現：
x を 100 とするのは、J で次のように書くことができます。

```
   x =: 100
```

代入と呼ばれるこの式は、値 100 を名前 x に代入します。x と呼ばれる変数が作成され、値 100 をとるとします。代入のみを含む入力行がコンピュータに入力されると、応答として何も表示されません。
計算された値が必要な場合はいつでも、割り当てられた値を持つ名前を使用することができます。

```
   x  -  1
99
```

代入内の値自体は式で計算できます。

```
   y =: x - 1
```

したがって、変数 y は、計算 x-1 の結果を記憶するために使用される。変数に割り当てられている値を確認するには、変数の名前だけを入力します。これは、特に単純な形式の、他のどのような表現でもあります。

```
   y
99
```

同じ変数に対して繰り返し代入を行うことができます。新しい値が現在の値より優先されます。

```
   z =: 6
   z =: 8
   z
8
```

変数の値は、同じ変数の新しい値を計算する際に使用できます。

```
   z =: z + 1
   z
9
```

上記のように、割り当てからなる行が入力されたときには値は表示されません。それにもかかわらず、代入は式です。より大きな式に参加できる値を持っています。

```
   1 +(u =: 99)
100
   u
99
```

変数の名前を選択する方法を示すための割り当ての例を以下に示します。

```
   x                       =: 0
   X                       =: 1
   K9                      =: 2
   finaltotal              =: 3
   FinalTotal              =: 4
   average_annual_rainfall =: 5
```

それぞれの名前は文字で始まる必要があります。文字(大文字または小文字）、数字(0-9)またはアンダースコア(_ ）のみを使用できます。大文字と小文字は区別されます。 x と X は異なる変数の名前です。

```
   x
0
   X
1
```

1.7 Terminology: Monads and Dyads

用語集：モナド(単項)とダイアド(両項)

右に単一の引数を取る関数は、モナディック関数、または略してモナドと呼ばれます。例は「Square」(* :)です。左に 1 つ、右に 1 つの 2 つの引数を取る関数は、ダイアディック関数またはダイアドと呼ばれます。例は + です。

減算と否定は、2 つの異なる機能を示す同じ記号(-）の例です。言い換えれば、我々は-モナドの場合(否定）と二項の場合(減算）の意味を持っている、ということができます。J のほとんどすべての組み込み関数は、モナドとダイアディックの両方のケースを持っています。

別の例として、除算関数が % であるか、今のように % の二項の場合を思い出してください。% のモナドの場合は相反関数す。

```
   % 4
0.25
```

1.8 More Built-In Functions
より多くの組み込み関数

このセクションの目的は、J で提供される多くの組み込み関数の小さな選択肢を調べることによって、J でのプログラミングの味の一部を伝えることです。
英語の表現を考えてみましょう：数字 2，3、および 4 以上を簡単に加算してください：
一緒に加算する 2 3 4 結果は 9 です。この式は J で次のように表現されます。

```
   + / 2 3 4
9
```

英語と J を比較すると、 "add"は + で伝えられ、"together"は / で伝えられます。同様に、式：
2 3 4 をまとめて掛け算すると、結果は 24 になるはずです。この式は、J で次のように表現されます。

```
   * / 2 3 4
24
```

我々はそれを + / 2 4 3 の意味が 2 + 3 + 4 で * / 2 3 4 の意味が 2 * 3 * 4 と見ることができます。 実際には右のリストの各項目の左にある機能を挿入するための記号 / は「挿入」と呼ばれます。一般的なスキームは、F が任意の二項関数であり、L が数字のリストである場合、a、b、c、...、y、z の 場合、

```
        F / L とは、F b F ... F y F z
```

さらなる機能に移り、これらの 3 つの命題を検討してください。

2 が 1 より大きい	(これは明らかに真実です)
2 は 1 に等しい	(これは偽である)
2 が 1 未満である	(これは偽である)

J では、「真」は数字 1 で表され、「偽」は数字 0 で表されます。3 つの命題は J で次のように表現される。

```
   > 1 2
1
   2 = 1
0
   2 <1
0
```

もし x が、たとえば、数字のリストであるとすれば：

```
   x =: 5 4 1 9
```

x のどの数字が 2 よりも大きいかを聞くことができます。

```
   x> 2
1 1 0 1
```

明らかに、第 1、第 2、および最後は、x> 2 の結果として 1 にと報告される。
x のすべての数値が 2 より大きい場合は？

```
   * / x> 2
0
```

いいえ、私たちは x> 2 が 1 1 0 1 であることを知っていたからです。ゼロ(「偽」）の存在は、乗算(ここでは 1 * 1 * 0 * 1）が 1 を生成できないことを意味します。
2 より大きい x のアイテムはいくつありますか？x> 2 に 1 を加えます：

```
   + / x> 2
3
```

x には何個の数字がありますか？1 を x = x にまとめることができます。

```
   x
5 4 1 9
   x = x
1 1 1 1
   + / x = x
4
```

しかし、リストの長さを与える組み込みの関数: #(「Tally」と呼ばれる）があります：

```
   #x
4
```

1.9 Side By Side Displays
サイドバイサイドディスプレイ

J 式をコンピュータに入力すると、式と結果が画面の下に表示されます。最後の数行をもう一度見せてください：

```
   x
5 4 1 9
   x = x
1 1 1 1
   + / x = x
4
   # x
4
```

さて、この本では、次のような数行の説明がありますが、他のページにはありません。

x	x = x	+ / x = x	# x
5 4 1 9	1 1 1 1	4	4

これの意味：
あなたが式を入力した場合、手続のこの段階では、x でレスポンス 5 4 1 9 を見ることができる。x = x と入力すると、1 1 1 1 が表示されます。サイドバイサイドディスプレイは、J システムの機能ではなく、本書の図やイラストだけを示しています。最初の行には式が表示され、2 行目には対応する値が表示されます。

あなたが代入(x =: something）をタイプすると、J システムは値を表示しません。それにもかかわらず、代入式であり、値を保持します。私たちの課題の価値を見たり、思い出させたりすることは、何度も参考になるかもしれません。私はしばしばそれらを並べて表示します。
説明：

x = 1 + 2 3 4	x = x	+ / x = x	# x
3 4 5	1 1 1	3	3

組み込み関数に戻ると、リストがあるとします。それから私たちは順番にそれらを取って、"yes、yes、no、yes、no"と言って項目を選ぶことができます。我々の選択順序は、1 1 0 1 0 として表すことができる 。そのような 0 と 1 のリストは、ビット列(または時にはビットリストまたはビットベクトル）と呼ばれます。 関数 dyadic # があり、右の引数から選択した項目を選択するために、左の引数としてビット列(選択肢の列）を取ることができます。

y =: 6 7 8 9 10	1 1 0 1 0 # y
6 7 8 9 10	6 7 9

我々は y から、以下のようないくつかの条件を満たす項目だけを選択することができます：
7 より大きい項目

y	y > 7	(y > 7) # y
6 7 8 9 10	0 0 1 1 1	8 9 10

1.10 Comments
コメント

J の行では、記号 NB。(大文字 N、大文字 B ドット）はコメントを導入する。NB に続くもの。行末までは評価されません。例えば

```
   NB.   this is a whole line of annotation
   6 + 6   NB. ought to produce 12
12
```

1.11 Naming Scheme for Built-In Functions
組み込み関数の命名規則

J の各組み込み関数には、非公式で正式な名前が付いています。たとえば、正式名 + の関数は、非公式の名前 "Plus"を持ちます。さらに、我々は正式な名前がなるよう、単項および二項の場合があるかもしれないことを見てきました - 非公式名「否定 Negate」と「マイナス Minus」に対応します。
非公式の名前は、事実上、短い記述、通常は 1 語です。それらは J ソフトウェアによって認識されません。つまり、J の式は常に正式な名前を使用します。この本では、非公式の名前が引用されるので、「マイナス」。

J のほとんどすべての組み込み関数は、1 文字または 2 文字の正式な名前を持っています。例は * と *: の関数です。2 番目の文字は常に :(コロン）または、(dot、full stop、または period）。
2 文字の名前は、基本的な 1 文字関数との関係を示すためのものです。したがって、「Square」(*:) は「Times」(*) に関連しています。
したがって、組み込みの J 関数は、最大 6 つの関連関数のファミリになる傾向があります。モナドおよびダイアディックのケースがあり、それぞれの場合に基本、コロン、およびドットの変形があります。これは > family のために説明されます。
Dyadic > 我々はすでに「より大きい」として会いました。
Monadic > 私たちは後で戻ってきます。
Monadic >. 引数を整数に丸めます。丸めは、最も近い整数に丸めるのではなく、常に上になることに注意してください。
したがって、名前： "天井 Ceiling"

```
   >. _1.7 1 1.7
_1 1 2
```

Dyadic >。その 2 つの引数のうち大きい方を選択する

```
   3>. 1 3 5
3 3 5
```

/ を使って項目の間に "Larger Of"を挿入することで、リスト内で最大の数を見つけることができます。たとえば、リスト 1 6 5 の中で最大の数は、(>。/ 1 6 5) を評価することによって求められます。次の数行は、これが 6 を与えるはずであることをあなたに確信させるためのものです。このコメントは、各行が前回と同じ結果をもたらす理由を示しています。

```
   >. / 1 6 5
6
   1 >. 6 >. 5      NB. by the meaning of /
6
   1 >. (6 >. 5)    NB. by rightmost-first rule
6
   1 >. (6)         NB. by the meaning of >.
6
   1 >. 6           NB. by the meaning of ()
6
   6                NB. by the meaning of >.
6
```

Monadic >: 非公式に「Increment」と呼ばれます。引数に 1 を加えます：

```
   >: _2 3 5 6.3
_1 4 6 7.3
```

Dyadic >: は「より大きいか等しい Large or Equal」

```
   3>: 1 3 5
1 1 0
```

★第 1 章終了

第 2 章：Lists and Tables
リストと表

計算にはデータが必要です。これまでのところ、データは単一の数字または数字のリストとしてしか見られませんでした。たとえば、テーブルなど、データを使って他のことを行うことができます。リストやテーブルのようなものは「配列」と呼ばれます。

2.1 Tables
テーブル

例えば、2 行 3 列のテーブルを$関数で構築することができます：

```
   table=: 2 3 $ 5 6 7 8 9 10
   table
5 6  7
8 9 10
```

ここでのスキームは、式(x $ y）がテーブルを作成するということです。表の次元は、行数とそれに続く列数の形式のリスト x によって与えられます。テーブルの要素はリスト y によって供給されます。

y の 項目は、最初の行、次に 2 番目の行などを埋めるように順番に取得されます。リスト y には少なくとも 1 つの項目が含まれていなければなりません。テーブル全体を埋めるために y に項目が少なすぎる場合、y は最初から再利用されます。

`2 4 $ 5 6 7 8 9`	`2 2 $ 1`
`5 6 7 8` `9 5 6 7`	`1 1` `1 1`

$の関数は、テーブルを構築するための一つの方法を提供していますが、より多くの方法があります：参照章 05。

関数は、以前のリストのように、テーブル全体に正確に適用できます。

`table`	`10 * table`	`table + table`
`5 6 7` `8 9 10`	`50 60 70` `80 90 100`	`10 12 14` `16 18 20`

1 つの引数はテーブルと 1 つのリストです。

`table`	`0 1 * table`
`5 6 7` `8 9 10`	`0 0 0` `8 9 10`

この最後の例では、明らかにリストの項目 0 1 が表の行と自動的に一致し、0 が最初の行に一致し、1 が 2 番目に一致します。 引数を互いに照合する他のパターンも可能です - 第 07 章を参照してください。

2.2 Arrays
配列

table は 2 つの次元(すなわち行と列）を持つと言われており、この意味でリストは 1 つの次元しか持たないと言える。
2 つ以上のディメンションを持つテーブルのようなデータオブジェクトを持つことができます。$関数の左の引数には、任意の数のディメンションのリストを指定できます。「配列 arrays」という単語は、いくつかの次元数を持つデータオブジェクトの一般名として使用されます。1 次元、2 次元、3 次元の配列があります：

```
3 $ 1
1 1 1
```

```
2 3 $ 5 6 7
5 6 7
5 6 7
```

```
2 2 3 $ 5 6 7 8
5 6 7
8 5 6

7 8 5
6 7 8
```

最後の例の 3 次元配列は、2 つの面、2 つの行、3 つの列を持ち、2 つの面が上下に表示されます。
モナド関数: #はリストの長さを与えることを思い出してください。

```
# 6 7
2
```

```
# 6 7 8
3
```

モナド関数$は、引数の次元のリストを与える：

```
L =: 5 6 7
$ L
5 6 7
```

```
3
```

```
T =: 2 3 $ 1
1 1 1
1 1 1
```

```
$ T
2 3
```

したがって、x が配列の場合、式(# $ x）は x の次元のリストの長さ、つまり x の次元数を 返します。リストの場合は 1、テーブルの場合は 2、となります。

```
L
5 6 7
```

```
$ L
3
```

```
# $ L
1
```

```
T
1 1 1
1 1 1
```

```
$T
2 3
```

```
# $ T
2
```

我々が取る場合は、x を、単一の数であるように、その式(# $ x）がゼロになります。

```
   # $ 17
0
```

これは、テーブルに 2 つのディメンションがあり、リストに 1 つのディメンションが含まれている一方、ディメンションカウントがゼロであるため、単一の数値には何もありません。次元数がゼロのデータオブジェクトはスカラーと呼ばれます。「配列」はいくつかの次元数を持つデータオブジェクトであるため、スカラーも配列であり、次元数はこの場合はゼロです。
私たちは(: #$ 17）が 0 であるのを見ました。スカラーには次元がないため、次元リスト(ここでは$ 17 で与えられる）は長さゼロの、つまり空のリストでなければならないと結論

づけることができます。今度は長さ 2 のリスト、例えば 2 $ 99 のような式で生成することができる ので、長さ 0 の空のリストは 0 $ 99 (実際には 0 $の任意の数)
空のリストの値は、何も表示されません。

2 $ 99 ドル	0 $ 99	$ 17
99 99		

スカラー(17 と言う）は、長さ 1 のリスト(例えば 1 $ 17）、または 1 つの行と 1 つの列(例えば 1 1 $ 17）のテーブルと同じではないことに注意してください。スカラーは次元を持たず、リストには 1 つあり、テーブルには 2 つありますが、3 つすべてが画面に表示されたときに同じに見えます：

```
S =: 17
L =: 1 $ 17
T =: 1 1 $ 17
```

S	L	T	: # $ S	: # $ L	: # $ T
17	17	17	0	1	2

表は 1 つの列のみを持つことができますが、依然として 2 次元の表です。ここでは、t は 3 行 1 列です。

t =: 3 1 $ 5 6 7	$ t	: # $ t
5 6 7	3 1	2

2.3 Terminology: Rank and Shape
用語：ランクと形状

私たちが "dimension-count"と呼ぶプロパティは、 "rank"という短い名前で呼ばれるので、一つの数字は rank-0 配列と呼ばれ、numbers のリストは rank-1 配列などです。配列の次元のリストは、その "形状"と呼ばれます。
数学用語「ベクトル」と「行列」は、「リスト」と「テーブル」(数字の）と呼ばれるものに対応しています。3 次元またはそれ以上の次元を持つ配列(または、ここではランク 3 以上の配列）は、「レポート」と呼ばれます。
配列を記述するための用語と関数の要約を次の表に示します。

```
+--------+--------+-----------+------+
|        | Example| Shape     | Rank |
+--------+--------+-----------+------+
|        | x      | $ x       | # $ x|
+--------+--------+-----------+------+
| Scalar | 6      | empty list| 0    |
+--------+--------+-----------+------+
| List   | 4 5 6  | 3         | 1    |
+--------+--------+-----------+------+
```

```
| Table  |0 1 2   | 2 3       | 2    |
|        |3 4 5   |           |      |
+--------+--------+-----------+------+
| Report |0  1  2 | 2 2 3     | 3    |
|        |3  4  5 |           |      |
|        |        |           |      |
|        |6  7  8 |           |      |
|        |9 10 11 |           |      |
+--------+--------+-----------+------+
```

上記の表は、実際には小さなＪプログラムによって作成されたもので、今話したような本物の「テーブル」です。その形状は 6 4 です。しかし、単語、数字のリストなどが含まれているので、明らかに数字の表だけではありません。ここで、数値以外の配列を調べます。

2.4 Arrays of Characters
文字の配列

文字は、アルファベット、句読点、数字などの文字です。私たちは数字の配列を持つのと同じように文字の配列を持つことができます。文字のリストは、単一引用符の間に入力されますが、引用符なしで表示されます。例えば：

```
title =: 'My Ten Years in a Quandary'
   title
My Ten Years in a Quandary
```

文字のリストは、文字列または単に文字列と呼ばれます。文字列中の一重引用符は、連続する二重引用符として入力されます。

```
   'What''s new?'
What's new?
```

空の、または長さゼロの文字列は、2 つの連続した一重引用符として入力され、何も表示されません。

''	# ''
	0

2.5 Some Functions for Arrays
配列のいくつかの関数

この時点で、配列を処理するためのいくつかの関数を調べることは役に立ちます。Ｊはこのような機能が非常に豊富です：ここではほんの数例を示します。

2.5.1 Joining
結合

組み込み関数、(カンマ)「追加」と呼ばれています。それは一緒にリストを作るために結合します。

a =: 'rear'	b =: 'ranged'	a、b

rear	ranged	rearranged

「追加」機能は、リストまたは単一項目を結合します。

x =: 1 2 3	0, x	x, 0	0, 0	x, x
1 2 3	0 1 2 3	1 2 3 0	0 0	1 2 3 1 2 3

「Append」機能は、2 つのテーブルを取り、エンドツーエンドでそれらを結合して、より長いテーブルを形成することができます。

T1 =: 2 3 $ 'catdog'	T2 =: 2 3 $ 'ratpig'	T1, T2
catdog	ratpig	catdogratpig

「追加」の詳細については、第 05 章を参照してください。

2.5.2 Items
項目

数字のリストの項目は個々の数字です。表の項目はその行です。3 次元配列の項目はその面です。一般的に、配列の項目は、最初の次元に沿って順番に表示されるものです。配列はその項目のリストです。

リストの長さを与える 組み込み関数 #("Tally"）を思い出してください。

x	# x
1 2 3	3

一般的に、# は配列の項目数を数えます。つまり、最初の次元が与えられます。

T1	$ T1	# T1
catdog	2 3	2

明らかに # T1 は次元リスト $ T1 の最初の項目です。次元のないスカラーは単一の項目とみなされます。

6
1

上記の「追加」の例を再度検討してください。

T1	T2	T1, T2
catdog	ratpig	catdogratpig

ここで、一般的に(x、y）は、x の項目に y の項目が続くリストであると言うことができます。"items"の有用性の別の例として、動詞 + / where + がリストの項目間に挿入されていることを思い出してください。

+ / 1 2 3	1 + 2 + 3

6	6

これで、一般的に配列の項目間で + / inserts +と言うことができます。次の例では、項目は行です。

T =: 3 2 $ 1 2 3 4 5 6	+ / T	1 2 + 3 4 + 5 6
1 2 3 4 5 6	9 12	9 12

2.5.3 Selecting
選択

次に、リストから項目を選択する方法を見ていきます。リスト内の位置には、0，1，2 などの番号が付けられます。最初の項目は位置 0 を占めます。その位置で項目を選択するには、関数 {(左中括弧、 "From"）を使用します。

Y =: 'abcd'	0 { Y	1 { Y	3 { Y
abcd	a	b	d

位置番号はインデックスと呼ばれます。{ 関数は、左引数として単一のインデックスまたはインデックスのリストを取ることができます。

Y	0 { Y	0 1 { Y	3 0 1 { Y
abcd	a	ab	dab

組み込みの関数 i があります。(文字 i とドット）。式(i.n）は、ゼロから n 個の連続した整数を生成する。

i. 4	i. 6	1 + i. 3
0 1 2 3	0 1 2 3 4 5	1 2 3

x がリストである場合、式(i. # x）からリスト x にすべての可能なインデックスを生成。

x =: 'park'	# x	i. # x
park	4	0 1 2 3

リスト引数で i. 配列を生成する：

i. 2 3
0 1 2
3 4 5

「Index Of」と呼ばれる二項バージョンとしての i. があります。
式(X i. y）は、それが、x における y の位置を見つける指標である。

```
   'park' i. 'k'
```

```
3
```

見つかったインデックスは、x の最初に出現する y のインデックスです。

```
   'park' i. 'a'
1
```

y が x に存在しない場合、見つかったインデックスは最後の可能な位置よりも 1 大きい。

```
   'park' i. 'j'
4
```

インデックス作成のさまざまなバリエーションについては、第 6 章を参照してください。

2.5.4 Equality and Matching
平等とマッチング

2 つの配列が同じかどうかを調べたいとします。組み込み関数 -:(minus colon を"Match"と呼ぶ）があります。2 つの引数が対応する要素に対して同じ形と同じ値を持つかどうかをテストします。

X =: 'abc'	X -: X	Y =: 1 2 3 4	X -: Y
abc	1	1 2 3 4	0

引数が何であれ、Match の結果は常に 1 つの 0 または 1 です。
例えば、文字の空のリストは、空の数字のリストと一致するとみなされます。

```
   ''  -: 0 $ 0
1
```

それらは同じ形をしており、(対応する要素がないので）対応するすべての要素が同じ値を持つことは事実です。
引数が等しいかどうかをテストする もう一つの関数、 =("Equal"と呼ばれる）があります。 =は引数の要素を要素ごとに比較し、引数と同じ形のブール値の配列を生成します。

Y	Y = Y	Y = 2
1 2 3 4	1 1 1 1	0 1 0 0

したがって、=の 2 つの引数は、同じ形(または少なくとも Y = 2 の互換性のある形の例のように）を持たなければなりません。それ以外の場合は、エラーが発生します。

Y	Y = 1 5 6 4	Y = 1 5 6
1 2 3 4	1 0 0 1	length error

2.6 Arrays of Boxes
箱の配列
2.6.1 Linking
リンク

組み込み関数があります。(セミコロン、 "リンク"と呼ばれる）。2 つの引数をリンクしてリストを形成します。2 つの引数は異なる種類のものであってもよい。たとえば、文字列と数字をリンクすることができます。

```
   A =: 'The answer is'  ;  42
   A
```

```
+-------------+--+
|The answer is|42|
+-------------+--+
```

結果 A は長さ 2 のリストであり、ボックスのリストと呼ばれます。A の最初のボックスの中に文字列 'The answer is'があります。2 番目のボックスの中には 42 番の数字があります。ボックスは、ボックスに含まれる値の周りに描かれた四角形で画面に表示されます。

```
A
```

```
+-------------+--+
|The answer is|42|
+-------------+--+
```

```
0 {A
```

```
+-------------+
|The answer is|
+-------------+
```

ボックスは、その内部にどのような種類の値が入っていてもスカラーです。したがって、ボックスは数字のように通常の配列にパックすることができます。したがって、A はスカラのリストです。

```
A
```

```
+-------------+--+
|The answer is|42|
+-------------+--+
```

```
$ A
```

```
2
```

```
s =: 1 {A
```

```
+--+
|42|
+--+
```

```
: # $ s
```

```
0
```

ボックスの配列の主な目的は、おそらく異なる種類の複数の値を 1 つの変数にアセンブルすることです。たとえば、購入の詳細(日付、金額、説明）を記録する変数は、ボックスのリストとして作成できます。

```
   P =: 18 12 1998; 1.99; '焼きたての豆'
   P
+ ---------- + ---- + ---------- +
| 18 12 1998 | 1.99 |焼きたての豆|
+ ---------- + ---- + ---------- +
```

「リンク」と「追加」の違いに注意してください。"Link"は異なる種類の値を結合しますが、"Append"は常に同じ種類の値を結合します。つまり、「Append」の 2 つの引数は、両方とも数字の配列、文字の両方の配列、または両方のボックスの配列でなければなりません。それ以外の場合は、エラーが通知されます。

```
'answer is'; 42
```

```
+---------+--+
|answer is|42|
+---------+--+
```

```
'answer is' , 42
```

```
error
```

場合によっては、文字列を数値と組み合わせて、たとえば計算の結果を何らかの説明とともに表示することもできます。上に見たように、説明と番号を「リンク」することができました。しかし、数字を文字列に変換し、文字列としてこの文字列と説明を追加することで、よりスムーズなプレゼンテーションを作成することができます。
数値を文字列に変換するには、組み込みの "Format"関数":(二重引用符コロン）を使用します。次の例で は、n は単一の数値ですが、s は n の書式化された値です。長さ 2 の文字。

```
n =: 42
```

```
s =: ": n
```

```
# s
```

```
'answer is ' , s
```

```
42
```

```
42
```

```
2
```

```
answer is 42
```

"Format"の詳細については、第 19 章を参照してください。今度はボックスのテーマに戻ります。ボックスは四角で描かれているので、画面上に結果を簡単な表形式で表示することができます。

```
   p =: 4 1 $ 1 2 3 4
   q =: 4 1 $ 3 0 1 1

   2 3 $ 'p';'q';'p+q';p;q;p+q

+---+---+-----+
| p | q | p+q |
+---+---+-----+
|1  |3  |4    |
|2  |0  |2    |
|3  |1  |4    |
|4  |1  |5    |
+---+---+-----+
```

2.6.2 Boxing and Unboxing
ボクシングとアンボクシング

組み込み関数 <(左角括弧、 "ボックス"と呼ばれます）があります。<を値に適用すると、単一のボックス化された値を作成できます。

```
   < 'baked beans'
+-----------+
|baked beans|
+-----------+
```

ボックスには数字が入っていますが、それ自体は数字ではありません。ボックス内の値に対して計算を実行するには、そのボックスが「開かれ！ 取り出された値を表すようにする必要があります。関数 >(右角括弧）は「開く」と呼ばれます。

```
b =: < 1 2 3
```

```
> b
```

```
+-----+
|1 2 3|
+-----+
```

```
1 2 3
```

< (ボックス化)を漏斗として 描くと効果的です。幅の広い端に流れ込むデータがあり、狭い端から流れ出すと、スカラー、すなわち無次元または点状のボックスがあります。逆の >

(ボックス開く)、ボックスはスカラであるため、カンマ関数でボックスのリストにまとめられますが、セミコロン関数は、文字列の組み合わせとボクシングを組み合わせるため、しばしばより便利です。

(<11) 、(<2 2) 、(<3 3)	1 1; 2 2; 3 3
+ --- + --- + --- + | 1 1 | 2 2 | 3 3 | + --- + --- + --- +	+ --- + --- + --- + | 1 1 | 2 2 | 3 3 | + --- + --- + --- +

2.7 Summary

まとめ

結論として、J 内のすべてのデータオブジェクトは、0 次元、1 次元以上の次元を持つ配列です。配列は、数字の配列、文字の配列、またはボックスの配列(さらには可能性があります)です。

★第 2 章終了

第 3 章：Defining Functions
関数の定義

J には、組み込み関数のコレクションが付属しています。我々は*や+のようないくつかを見てきました。このセクションでは、独自の関数を定義するために、組み込み関数をさまざまな方法で組み立てる方法を最初に見ていきます。

3.1 Renaming
名前の変更

関数を定義する最も簡単な方法は、独自の選択肢の名前をビルトイン関数に与えることです。定義は代入です。たとえば、組み込みの*: 関数と同じ意味を持つ square を定義するには、次のようにします。

```
   square =: *:
   square 1 2 3 4
1 4 9 16
```

もっと記憶に残るように私たちが自分の名前を好むなら、これをすることもできます。同じ組み込み関数に 2 つの異なる名前を付けることができます。1 つはモナドの場合に使用し、もう 1 つはダイアディックの場合に使用します。

```
   Ceiling =: >.
   Max =: >.
```

Ceiling 1.7	3 Max 4
2	4

3.2 Inserting
挿入

(+ / 2 3 4) は 2 + 3 + 4 を意味し、同様に (* / 2 3 4) は 2 * 3 * 4 を意味することを 想起されたい。関数を定義し、代入を使って関数 sum に名前を付けることができます。

```
   sum =: + /
   sum 2 3 4
9
```

ここで、sum =: + / は、+ / がそれ自身で関数を表す式であることを示しています。
この表現 + / は、「挿入」として理解することができる / 関数に + を適用してリスト加算関数を生成します。
つまり、/ はそれ自体が一種の機能です。その左には 1 つの引数があります。引数とその結果の両方が関数です。

3.3 Terminology: Verbs, Operators and Adverbs
用語：動詞、演算子、副詞

私たちは 2 種類の機能を見てきました。まず、数字から数値を計算する + や * などの「普通」の関数があります。J では、これらを「動詞」と呼びます。
次に、関数の関数を計算する/などの関数があります。この種の関数は動詞と区別するために"演算子"と呼ばれます。

1 つの引数を取る演算子は「副詞」と呼ばれます。副詞は常に左の引数をとる。したがって、式(+ /) では、副詞/が動詞+に適用されてリスト要約動詞が生成されます。
この用語は英文の文法に由来します。動詞は物に作用し、副詞は動詞を修正します。

3.4 Commuting
交換

1 つの副詞 / を見て、別の副詞を見てみましょう。副詞 ~ は左と右の引数を交換する効果があります。

'a' , 'b'	'a' ,~ 'b'
ab	ba

このスキームは、引数 x と y を持つダイアド f

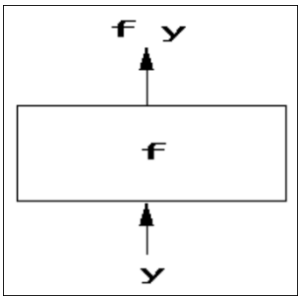
```
x f~ y      means    y f x
```

別の例として、残差動詞 | ここで 2 | 7 は従来の記法では「7 mod 2」を意味します。mod 動詞を定義することができます：

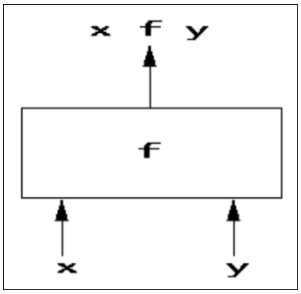
```
mod =: | ~
```

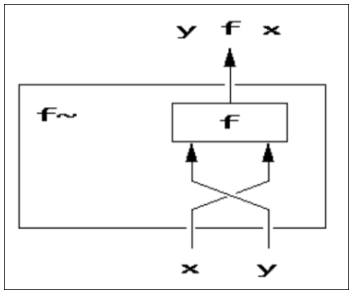
7 mod 2	2 | 7
1	1

私はいくつかの写真を描いてみましょう。まず、 引数 y に適用された関数 f の図で、結果 (f y) を生成します。この図では、関数 f は長方形として描かれ、矢印は関数に流入するか、または関数から流出する引数です。各矢印には式が付けられています。



ここでは、(x f y) を生成するために引数 x と y に適用される Dyadic f についての同様の図があります(下・右図参照)。





ここでは、関数(f~）の図を示します。関数 f の内部には、関数 f が含まれていて、矢印の交差した配列を含むように描くことができます。(上左図・参照)

3.5 Bonding
ボンディング

私たちは動詞二重定義して、二重の x の意味の x * 2 とするような、つまり、二重の「2 を掛ける」を意味することです。これを次のように定義します。

```
   double =: * & 2
   double 3
6
```

ここでは、ダイアドを取る*、及び(この場合、2）選択された値の 2 つの引数のいずれかを固定することにより、そこからモナドを生成します。&演算子は関数と一つの引数の値との間の結合を形成すると言われています。スキームは：f がダイアディック関数であり、k が f の右引数の値である場合、

```
          (f & k) y    means    y f k
```

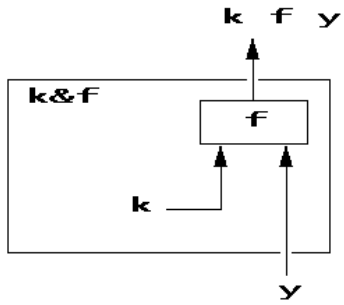
右の議論を修正するのではなく、左を修正することができるので、スキームもあります。

```
          (k & f)  y   means    k f y
```

たとえば、販売税率を 10%とし、税を計算する関数を購入価格から計算すると、次のようになります。

```
   tax =: 0.10 & *
   tax 50
5
```

これは関数 k & f を示す図です。



3.6 Terminology: Conjunctions and Nouns
用語：共用語と名詞

式(* & 2) は、& 演算子が 2 つの引数(動詞 * と 2 ）に適用される関数であり、その結果が「2 倍」動詞であると記述することができます。
以下のような 2 つの引数演算子 & は、それはその 2 つの引数を conjoins ので、「接続詞」J で呼ばれています。対照的に、副詞は引数が 1 つしかない演算子であることを思い出してください。
J のすべての関数は、組み込み関数であれユーザ定義であれ、モナド動詞、ダイアディック動詞、副詞または結合詞の 4 つのクラスのいずれかに正確に属します。ここでは、- のよう

な二価の記号を- モナド否定または二項減算という 2 つの異なる動詞を示すものとみなします。

J のすべての式は、ある型の値を持ちます。関数ではないすべての値はデータです(前のセクションで見たように実際は配列です）。

J では、データ値、つまり配列は、英文法の類推に基づいて「名詞」と呼ばれます。私たちは動詞ではなく、いくつかの次元を持つことを強調する配列を強調するために名詞と呼ぶことができます。

3.7 Composition of Functions
関数の構成

英語の表現を考えてみましょう：数 1 2 3 の平方和、すなわち(1 + 4 + 9）、または 14 です。私たちは、上記の動詞を合計と平方について定義したので、J 式は次のようになります。

```
   sum square 1 2 3
14
```

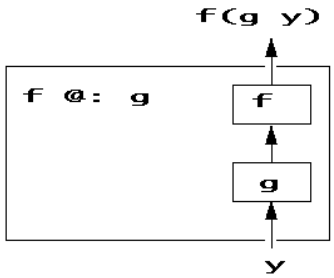
単一の二乗和関数は、和と平方の複合として書くことができます。

```
   sumsq =: sum @: square
   sumsq 1 2 3
14
```

記号 @: (at colon）は、 "合成"演算子と呼ばれます。スキームは、f と g が動詞であるならば、任意の引数 y は、

```
          (f @: g) y    means  f (g y)
```

スキームの図は次のとおりです。



この時点で、読者はなぜ構成を示すために私たちが(f @: g) を書くのか、単に(f g) だけではないのだろうと思っているかもしれません。簡単な答えは、(f g) は何か他のことを意味し、私たちはそのようにします。
組成の別の例では、華氏で表した温度を摂氏 32 に変換するには、関数 s を合計して 32 を減算し、合計を 5%9 とする。

```
   s       =: - & 32
   m       =: * & (5%9)
   convert =: m @: s
```

s 212	m s 212	convert 212

180	100	100

明確にするために、これらの例は、名前付き機能の構成を示した。もちろん、関数を表す式を作成することもできます。

```
   conv =: (* & (5%9)) @: (- & 32)
   conv 212
100
```

関数を表す式を名前を付けずに適用することができます：

```
   (* & (5%9)) @: ( - & 32) 212
100
```

上記の例では、モナドとモナドを構成していました。次の例は、ダイアドでモナドを構成することができることを示しています。一般的なスキームは次のとおりです。

```
            x (f @: g) y   means    f (x g y)
```

たとえば、数個の品目の注文の合計原価は、数量に対応する単価を掛けて合計したものです。説明する：

```
   P =:  2 3        NB. prices
   Q =:  1 100      NB. quantities
   total =: sum @: *
```

P	Q	P*Q	sum P * Q	P total Q
2 3	1 100	2 300	302	302

構成の詳細については、第 08 章を参照してください。

3.8 Trains of Verbs
動詞列

「痛みも、利益もない」という表現を考えてみましょう。これは文法的ではないにしてもかなり分かりやすい圧縮された慣用形式です。主動詞を持たない文ではありません。J には同様の概念があります：短い慣用リストに意味を与えるスキームに基づく、圧縮された慣用形式。これを次に見ます。

3.8.1 Hooks
フック

上記で定義した動詞税を想起して、購入に対する税額を 10%の割合で計算します。定義はここで繰り返されます：

```
   tax =: 0.10 & *
```

購入時に支払われる金額は、購入価格に計算された税金を加えた金額です。支払う金額を計算するための動詞を書くことができます：

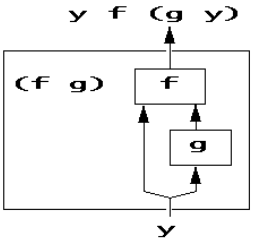
```
   payable =: + tax
```

購入価格が、たとえば $50 であれば、次のように表示されます。

tax 50	50 + tax 50	payable 50
5	55	55

定義(payable =: + tax) には、2 つの動詞 tax がきます。このシーケンスは、割り当ての右側にあることによって分離されています。このような動詞の孤立したシーケンスは「train」と呼ばれ、2 つの動詞の列は「hook」と呼ばれます。
括弧内の 2 つの動詞を分離するだけでフックを形成することもできます：

```
   (+ tax) 50
55
```



フックの一般的なスキームは、f がダイアドであり、g がモナドである場合、任意の引数 y に対して：

(f g) y means y f (g y)

スキームの図は次のとおりです。

別の例として、「床」動詞< 引数の整数部分を計算します。次に、数値が整数であるかどうかをテストするために、それがその床と等しいかどうかを尋ねることができます。「等しい」という意味の動詞はフック(= <.) です。

```
   wholenumber =: = <.
```

y = 3: 2.7	<. y	y = <. y	wholenumber
3 2.7	3 2	1 0	1 0

3.8.2 Forks
フォーク

番号のリストの算術平均 L は、の和によって与えられる L 内の項目数で割った L。(項目数はモナド動詞 # によって与えられることを思い出してください)

L =: 3 5 7 9	sum L	# L	(sum L) % (# L)
3 5 7 9	24	4	6

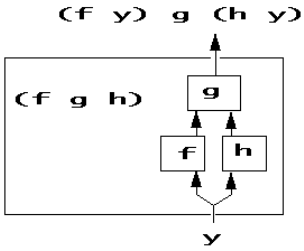
アイテムの数で割った和として平均値を計算する動詞は、三個の動詞の配列のように書くことができる：合計に続いて %、# が続きます。

```
   mean =: sum % #
   mean L
6
```

3 つの動詞の孤立したシーケンスは、フォークと呼ばれます。一般的なスキームは、f がモナドであり、g がダイアダであり、h がモナドであり、任意の引数 y についてモナドであり、

```
(f g h) y     means   (f y) g (h y)
```

このスキームの図は次のとおりです。



フォークの別の例では、リスト内の数字の範囲と呼ばれるものは 、中位の動詞がカンマ (,) である場合にフォーク最小のフォークによって与えられます。
第 01 章 から、リスト内の最大の数字は動詞 >./ で与えられるので、最小のものは <./ で与えられることを思い出してください。

```
   range =: <./  ,  >./
```

L	range L
3 5 7 9	3 9

フックとフォークは、動詞の列であり、動詞の「train」とも呼ばれます。tran の詳細については、第 09 章を参照してください。

3.9 Putting Things Together
ものをまとめる

私たちが上に見たいくつかのアイデアをまとめた長い例を試してみましょう。
その考え方は、与えられた数字のリストを単純に表示する動詞を定義し、それが何であるかを各数字に対して合計のパーセンテージとして表示することです。
この例のための完全なプログラムを紹介することから始めましょう。そうすれば、私たちがどこに行くのかを明確に見ることができます。私はあなたにこれを詳細に勉強することを期待していません、なぜなら以下の説明があるからです。display とそのサポート関数と呼ばれる動詞を定義して、7 行のプログラムを見ています。

```
   frac     =: % +/
   percent  =: (100 & *) @: frac
   round    =: <. @: (+ & 0.5)
   comp     =: round @: percent
   br       =: ,.  ;  (,. @: comp)
   tr       =: ('Data';'Percentages') & ,:
   display  =: tr @: br
```

非常にシンプルなデータから始めれば、

```
   data =: 3 1 4
```

その後、我々は、参照ディスプレイ全体の動詞は、(丸数字で）与えられるとパーセンテージとして各番号を示す：4 であり、50%8。

```
   display data
+---+----------+
```

```
|Data|Percentages|
+----+-----------+
|3   |38         |
|1   |13         |
|4   |50         |
+----+-----------+
```

まず、それぞれの数値を合計で割って、それぞれの割合を分数として表示することを目指します。フック(% +/) は適切です。これは、分割除算として読み取ることができます。我々がそれを frac と呼ぶならば

```
   frac =: % +/
```

それから私達は見る

data	+/data	data % (+/data)	frac data
3 1 4	8	0.375 0.125 0.5	0.375 0.125 0.5

パーセンテージは、端数に 100 を掛けたものです。

```
   percent  =: (100 & *) @: frac
```

data	frac data	percent data
3 1 4	0.375 0.125 0.5	37.5 12.5 50

% それぞれに 0.5 を加え、動詞(< 整数部）を取ることによって、パーセンテージを最も近い整数に丸めましょう。動詞 round は次のとおりです。

```
   round    =: <. @: (+&0.5)
```

次に、データから表示された値を計算する動詞は次のとおりです。

```
comp     =: round @: percent
```

percent data	round percent data	comp data
37.5 12.5 50	38 13 50	38 13 50

ここでは、データと計算値を列に表示します。リストから 1 列の表を作成するには、組込み動詞を使用できます。(カンマドット、「ラヴェルアイテム」と呼ばれます）。

data	,. data	,. comp data
3 1 4	3 1 4	38 13 50

ディスプレイの一番下の行を作るために、動詞 br を、データと計算された値をともに列としてリンクするフォークとして定義します。

```
   br  =:
.  ;  (
. @: comp)
```

data	br data
3 1 4	+-+--+ |3|38| |1|13| |4|50| +-+--+

ディスプレイの一番上の行(列の見出し）を追加するには、便利なビルトインの動詞があります ,:(comma Colon、「Laminate」、第 05 章で説明します）

```
   tr  =: ('Data';'Percentages') &
:
```

data	br data	tr br data
3 1 4	+-+--+ |3|38| |1|13| |4|50| +-+--+	+----+-----------+ |Data|Percentages| +----+-----------+ |3 |38 | |1 |13 | |4 |50 | +----+-----------+

私たちはすべてをまとめます：

```
   display =: tr @: br
   display data
+----+-----------+
|Data|Percentages|
+----+-----------+
|3   |38         |
|1   |13         |
|4   |50         |
+----+-----------+
```

この表示動詞には、値を計算する関数 comp(丸められた割合）と、結果を表示するのに関係する剰余という 2 つの側面があります。comp の定義を変更することによって、他の関数の値の表を表示することができます。comp を組み込みの平方根動詞(%:)と定義するとします。

```
   comp =: %:
```

また、tr 動詞で指定された一番上の行の列見出しを変更することもできます。

```
   tr   =: ('Numbers';'Square Roots') & ,:
   display 1 4 9 16

+-------+------------+
|Numbers|Square Roots|
+-------+------------+
| 1     |1           |
| 4     |2           |
| 9     |3           |
|16     |4           |
+-------+------------+
```

レビューでは、J というボンディング、コンポジション、フック、フォークの特徴を持つ小さな J プログラムを見てきました。すべての J プログラムと同様に、これは多くの可能な方法の 1 つに過ぎません。
この章では、まず関数を定義する方法を見てきました。関数には動詞と演算子の 2 種類があります。これまでは定義動詞だけを見てきました。次の章では、動詞を定義する別の方法を見ていきます。第 13 章 以降では、演算子を定義します。

☆第 3 章終了。

第 4 章：Scripts and Explicit Functions
スクリプトと明示的な関数

「スクリプト」と呼ばれるものは、J の一連の行で、計算を実行するためにシーケンス全体がオンデマンドで再生されます。この章のテーマは、スクリプトで定義された関数、およびファイル内のスクリプトです。

4.1 Text
テキスト

変数 txt への代入を次に示します。

```
   txt =: 0 : 0
What is called a "script" is
a sequence of lines of J.
)
```

式 0: 0 は、 "つぎのように"を意味します。すなわち、0: 0 は、その引数として取る動詞であり、結果として、それに続く行にソロの右括弧で始まる行まで。
txt の値は、これらの 2 行で、1 文字の文字列です。文字列に改行文字(LF）が含まれているため、txt が複数の行に表示されます。 txt には一定の長さがあり、それはランク 1、つまり単なるリストであり、2 つの改行文字が含まれています。

```
   txt
What is called a "script" is
a sequence of lines of J.
```

$ txt	# $ txt	+/ txt = LF
55	1	2

私たちは txt は "text"変数、つまり 0 個以上の改行文字を含む文字列であるとします。

4.2 Scripts for Procedures
手続きのためのスクリプト

ここでは、ステップバイステップの手順として説明した計算を見ていきます。非常に簡単な例として、華氏から摂氏への変換を 2 つのステップで記述することができます。いくつかの温度を考えると T は華氏で言います：

```
   T =: 212
```

その後、最初のステップは、32 コール結果を減算される t を、と言います

```
   t =: T - 32
```

2 番目のステップでは、t に 5%9 を掛けて温度を摂氏で与えます。

```
   t * 5%9
100
```

いろいろな T の値でこの計算を何回か実行しようとしているとします。この 2 行の手順は、必要に応じて再生できるスクリプトとして記録できます。このスクリプトは、J の行がテキスト変数に格納されているため、次のようになります。

```
   script =: 0 : 0
t =: T - 32
t * 5%9
```

```
)
```

このようなスクリプトは、例えば 0 と呼ぶことができる式 0 ! : 1 で与えられるビルトイン J 動詞で実行できます。

```
   do =: 0 !: 1
```

ここで、表現 0 ! : 1 は、左の引数 0 と右の引数 1 を接頭辞 ! :(感嘆符 コロン("Foreign Conjunction"と呼ばれる））で生成する動詞として理解できます。 !: 動詞のグループに編成された一連のユーティリティ関数またはシステムサービスを提供します。詳細については、この辞書を参照 here してください。
この例では、0 の左の引数はスクリプト実行グループを指定し、1 の右の引数はそのグループの特定のメンバーを取り出します。つまり、エラーに関係なく最後までスクリプトを実行し、実行を画面に表示する。
ここで do script を入力すると、キーボードから入力されたかのように画面上に行が表示されるはずです。

```
   do script
   t =: T - 32
   t * 5 % 9
100
```

T とは異なる値でスクリプトを再実行することができます

```
   T =: 32
   do script
   t =: T - 32
   t * 5 % 9
0
```

4.3 Explicitly-Defined Functions
明示的に定義された関数

関数はスクリプトで定義できます。ここでは、動詞としての華氏から摂氏への変換の例を示します。

```
   Celsius =: 3 : 0
t =: y - 32
t * 5 % 9
)
```

Celsius 32 212	1 + Celsius 32 212
0 100	1 101

この定義の主な特徴は次のとおりです。

4.3.1 Heading
見出し

この関数は 3 : 0 の式で導入されます。これは "次のような動詞"を意味します。(対照的に、0 : 0 は「次の文字列」を意味する）。
3 : 0 のコロンは連結詞です。その左の引数(3）は "動詞"を意味します。その右の引数(0)は、 "行の後に"を意味します。詳細は第 12 章を参照してください。このように導入された関数は、「明示的に定義された」または「明示的に」と呼ばれます。

4.3.2 Meaning

意味(Meaning)

式 Celsius 32 212 は、以下のように記述またはモデル化することができる計算を実行することによって、動詞 Celsius を引数 32 212 に適用する。

```
   y =: 32 212
   t =: y - 32
   t * 5 % 9
0 100
```

最初の行の後は、スクリプトに従って計算が進行することに注意してください。

4.3.3 Argument Variable(s)

引数変数

引数(32 212）の値は、変数 y としてスクリプトに供給されます。この "引数変数"は、モナド関数では y という名前です。(ダイアディック関数では、以下に示すように、左の引数は x、右は y です）

4.3.4 Local Variables

ローカル変数

Celsius 定義を繰り返しました：

```
   Celsius =: 3 : 0
t =: y  -  32
t * 5 % 9
)
```

変数 t への代入が含まれていることがわかります。この変数は、Celsius の実行中にのみ使用されます。残念ながら、t へのこの代入は、摂氏以外で定義された t とも呼ばれる他の変数の値と干渉します。デモを行うには：

```
   t =: 'hello'
   Celsius 212
100

   t
180
```

元の値('hello'）を持つ 変数 t が摂氏実行時に変更されていることがわかります。この望ましくない影響を避けるために、摂氏の内側の t は厳密に私的なものであり、t と呼ばれる他の変数とは区別されます。
この目的のために、特別な形式の割り当てがあり、記号=が付いています 。(等しいドット）。私たちの改訂版の定義は

```
   Celsius =: 3 : 0
t =. y - 32
t * 5 % 9
)
```

そして、私たちはと言うトンで Celsius はローカル変数である、またはその t はに対してローカルで Celsius。対照的に、関数の外で定義された変数はグローバルであると言います。

ここで、摂氏ではローカル変数 t への代入 はグローバル変数 t に影響しないことを証明することができます

```
    t =: 'hello'
   Celsius 212
100

   t
hello
```

引き数変数 y もローカル変数です。したがって、(Celsius 32 212）の評価は 、計算によってより正確にモデル化されます。

```
   y =。32,212
   t =。y  - 32
t * 5 % 9
0 100
```

4.3.5 Dyadic Verbs
二項動詞

Celsius 3 : 0 で導入され、単一の引数 y の観点から定義されるモナド動詞です。対照的に、二項動詞は 4 : 0 で導入されています。左と右の引数は常にそれぞれ x と y と名付けられます。次に例を示します。2 つの数字の「正の差」は、大きい方から小さい方へと変化します。

```
   posdiff =: 4 : 0
larger  =. x >. y
smaller =. x <. y
larger - smaller
)
```

3 posdiff 4	4 posdiff 3
1	1

4.3.6 One-Liners
ワンライナー(One-Liners)

1 行のスクリプトを文字列として書くことができ、コロン結合の右の引数として与えられます。

```
   PosDiff =: 4 : '(x >. y) - (x <. y)'
   4 PosDiff 3
1
```

4.3.7 Control Structures
制御構造

スクリプトで定義された関数の例を見てきた例では、実行は最初の行の式で始まり、次の行に進みます。
このストレート・スルー・パスだけが可能なパスではありません。次に実行する式を選択することができます。
例として、与えられた長さ、幅、高さからボリュームを計算する関数を次に示します。引数が 3 つの項目(長さ、幅、高さ）のリストとして正しく指定されているかどうかを調べる関

数であるとします。そうであれば、ボリュームが計算される。そうでない場合、結果は文字列 'ERROR'になります。

```
   volume =: 3 : 0
if.   3 = # y
do.   * / y
else. 'ERROR'
end.
)
```

私たちは見る：

volume 2 3 4	volume 2 3
24	ERROR

if から end までの最後までの「do.」、「else.」そして「end.」の制御構造と呼ばれるものが、その中に形成されていれば、実行します。
制御構造の詳細については、第 12 章を参照してください。

4.4 Tacit and Explicit Compared
暗黙と明示的比較

ここでは、2 つの異なるスタイルの関数定義を見てきました。この章で紹介した明示的なスタイルは、引数を表す変数を明示的に記述しているので、そう呼ばれています。したがって、 上記のボリュームでは、変数 y は引数の明示的な言葉です。
対照的に、前の章で見たスタイルは「暗黙」と呼ばれています。なぜなら、議論の対象となる変数は言及されていないからです。たとえば、陽性差関数の明示的および暗黙の定義を比較します。

```
   epd =: 4 : '(x >. y) - (x <. y)'
   tpd =: >. - <.
```

暗黙のスタイルで定義された多くの関数も明示的に定義することができ、その逆も可能です。どのスタイルが望ましいかは、最も自然に見えるものに依存しますが、定義する関数を考えてみましょう。選択肢は、一方では、スクリプト化された一連のステップ、または他方では、より小さな機能の集合に、問題を分解することです。
暗黙のスタイルはコンパクトな定義を可能にします。このため、暗黙の機能は体系的な分析と変換に役立ちます。実際、J システムは、広い種類の暗黙の機能のために、そのような変換を逆および派生として自動的に計算することができる。

4.5 Functions as Values
値としての機能

関数は値であり、式を入力することで値を表示することができます。式は名前のように単純なものにすることができます。暗黙的な関数と明示的な関数のいくつかの値は次のとおりです。

```
   - & 32
+-+-+--+
|-|&|32|
+-+-+--+
```

```
   epd
+-+-+------------------+
|4|:|(x >. y) - (x <. y)|
+-+-+------------------+

   Celsius
+-+-+-----------+
|3|:|t =. y - 32|
| | |t * 5 % 9  |
+-+-+-----------+
```

各関数の値は、ここではボックス構造で表されます。これはデフォルトですが、いくつかの可能性があります：第 27 章を参照してください。ここでは、関数を生成するために再びタイプインすることができる文字のシーケンスとしての関数を示す「線形表現」についてのみ述べる。次のように入力して、セッションを切り替えることで、関数を線形表現で表示することができます。

```
   (9!: 3) 5
```

たとえば、次のように表示されます。

```
epd
4 : '(x >. y) - (x <. y)'
```

以下の章では、関数の値がこの線形表現で表示されることがよくあります。

4.6 Script Files
スクリプトファイル

私たちは、単一変数の定義に使用されるスクリプト(J の行）を見ました：テキスト変数または関数。対照的に、J の行をテキストとして保持するファイルは、多くの定義を格納できます。このようなファイルはスクリプトファイルと呼ばれ、ファイルを読むことですべての定義を一緒に実行できるという利点があります。
ここに例があります。任意のテキストエディタを使用して、コンピュータ上に次のような 2 行のテキストを含むファイルを作成します。

```
squareroot =: %:
z = 1、(2 + 2）、(4 + 5)
```

J スクリプトファイルは、慣習的に.ijs で終わるファイル名を持っています。したがって、たとえば、フルパス名が c: ¥ temp ¥ myscript.ijs のファイルが(Windows では）作成されたとします 。
次に、J セッションでは、このファイル名を文字列として保持する変数 F を定義してファイルを識別すると便利です。

```
   F =:  'c：¥ temp ¥ myscript.ijs'
```

この 2 行のスクリプトファイルを作成したら、キーボードで入力することで実行できます：

```
      0 ! : 1 <F
```

キーボードから入力されたかのように画面上の線が表示されるはずです。

```
   squareroot =: %:
```

```
   z =: 1、(2 + 2) 、(4 + 5)
```

ファイルからロードしたばかりの定義で計算できるようになりました：

```
   z
1 4 9
   squarerootz
1 2 3
```

J セッションのアクティビティは、通常、スクリプトファイルの編集、スクリプトファイルからの定義の読み込みまたは再読み込み、およびキーボードでの計算の開始が混在しています。あるセッションから別のセッションに引き継がれるものは、スクリプトファイルだけです。J システム自体の状態またはメモリは、セッション中に入力されたすべての定義とともに、セッションの終了時に消滅します。したがって、J セッションを終了する前に、すべてのスクリプトファイルが最新であること、つまり保存したいすべての定義が含まれていることを確認することをお勧めします。
セッションの開始時に、J システムは「プロファイル」と呼ばれる指定されたスクリプトファイルを自動的にロードします。(詳細は第 26 章を参照してください）。プロファイルは編集することができ、一般的に役立つ独自の定義を記録するのに適しています。私たちは第 4 章と第 1 部の最後に来ました。以下の章では、第 1 部で触れたテーマをより深く詳細に扱います。

☆第 4 章終了

第 5 章：Building Arrays
配列の作成

この章では、配列の構築について説明します。最初にリストから配列を構築し、次に大きな配列を作るためにさまざまな方法で配列を結合します。

5.1 Building Arrays by Shaping Lists
シェーピングリストによる配列の構築

5.1.1 Review
レビュー

第 2 章 から、「アイテム」という言葉の意味を思い出してください。数字のリストの項目は数字です。テーブルの項目はその行です。3 次元配列の項目はその面です。
リコールはまた、`x $ y` は、リストの項目の配列生成 `Y` を形状で、`x` リストによって与えられた寸法、即ち、`x` 。例えば：

```
2 2 $ 0 1 2 3
```

```
0 1
2 3
```

```
2 3 $ 'ABCDEF'
```

```
ABC
DEF
```

リスト `y` が必要な項目の数よりも少ない数を含んでいる場合、必要な項目の数 を補うために `y` を循環的に再利用する。つまり、すべての要素が同じであるなど、単純なパターニングを表示するために配列を構築できます。

```
2 3 $ 'ABCD'
```

```
ABC
DAB
```

```
2 2 $ 1
```

```
1 1
1 1
```

```
3 3 $ 1 0 0 0
```

```
1 0 0
0 1 0
0 0 1
```

"Shape"動詞 dyadic `$` には、その引数の次元のリスト、すなわち形状のリストを生成する "ShapeOf"(モナド`$`）という付随動詞があります。説明する：

```
A =: 2 3 $ 'ABCDEF'
```

```
ABC
DEF
```

```
$ A
```

```
2 3
```

```
a =: 'pqr'
```

```
pqr
```

```
$ a
```

```
3
```

任意の配列 `A` に対して、その次元リスト `$ A` は 1 次元のリスト(形状）です。したがって `$ $ A` は 1 項目のリスト(ランク）です。したがって、`$ $ A` は常に `1` という数字だけを含むリストです。

```
A
```

```
ABC
```

```
$ A
```

```
2 3
```

```
$ $ A
```

```
2
```

```
$ $ $ A
```

```
1
```

DEF			

5.1.2 Empty Arrays
空の配列

配列は、その次元の長さがゼロであることができます。ゼロ長さまたは空のリストは、次元のリストとして 0、項目の値に任意の値(何に関係なく）を記述することで構築できます。

E =: 0 $ 99	$ E
	0

E が空の 場合は項目がないため、項目を追加した後に結果が 1 つの項目になります。

E	$ E	w =: E, 98	$ w
	0	98	1

同様に、ET が行を持たない空のテーブルであり、3 つの列がある場合、行を追加した後に結果は 1 つの行になります。

ET =: 0 3 $ 'x'	$ ET	$ ET, 'pqr'
	0 3	1 3

5.1.3 Building a Scalar
スカラの構築

スカラーを構築する必要があるとします。スカラにはディメンションがありません。つまり、ディメンション・リストは空です。スカラーを作る$の左の引数として空のリストを与えることができます：

S =: (0 $ 0) $ 17	$ S	$ $ S
17		0

5.1.4 Shape More Generally
より一般的な形状

我々は、（x $ v）は、v の項目の x 字型配列を生成すると言った。すなわち、一般的にの形状である(X は$ y は）だけでなくなり、X、むしろ x の項目の形状続い Y。
場合 y はテーブルであり、その後の項目 y が行である、すなわち、リストです。次の例では、項目の形状 Y は、 の行の長さ Y である、4。

X =: 2	Y =: 3 4 $ 'A'	Z =: X $ Y	$ Z
2	AAAA AAAA	AAAA AAAA	2 4

	AAAA		

次のセクションでは、既存の配列を結合して新しい配列を構築する方法について説明します。

5.2 Appending, or Joining End-to-End
追加、またはエンドツーエンドへの参加

どの配列も項目のリストと見なすことができるので、たとえば表の項目はその行です。動詞 (カンマ) は「追加」と呼ばれます。式(x, y) は、x の項目の後に y の項目が続くリストです。

```
   B =: 2 3 $ 'UVWXYZ'
   b =: 3 $ ' uvw '
```

a	b	a, b	A	B	A, B
pqr	uvw	pqruvw	ABC DEF	UVW XYZ	ABC DEF UVW XYZ

上記(A、B) の例では、A の項目は長さ 3 のリストであり、B の項目も長さのリストである。したがって、A のアイテムは、B のアイテムと互換性があり、すなわち、ランクと長さが同じである。もし彼らがしなければ？この場合、「Append」動詞は、同じランクに持ち帰り、長さにパディングし、必要に応じてスカラーを複製することによって、他のものに合うように 1 つの引数を引き伸ばそうとします。これは以下の例を示しています。

5.2.1 Bringing To Same Rank
同じランクにする

テーブルに行を追加したいとします。たとえば、3 文字のリスト b(上記）を 2 行 3 列の表 A(上）に追加して新しい行を作成することを検討してください。

A	b	A, B
ABC DEF	uvw	ABC DEF uvw

A の 2 つの項目の後に b の 1 つの項目を追加したいが、b は 1 項目の項目ではないことに注意してください。b を 1×3 テーブルに再構成することによって、つまり b のランクを上げることによって、それを行うことができます。しかしながら、必要であれば 1 の先行次元を供給することによって、「Append」動詞が低ランク引数を 1 項目配列に自動的に引き伸ばすので、これは必ずしも必要ではありません。

A	b	A, (1 3 $ b)	A, B	b, A
ABC	uvw	ABC	ABC	uvw

DEF		DEF uvw	DEF uvw	ABC DEF

5.2.2 Bringing To Same Rank

長さへのパディング

1 つの引数の項目が他の項目の項目より短い場合、それらは長さにパディングされます。文字配列は空白文字で埋められ、数値配列はゼロで埋められます。

A	A, 'XY'	(2 3 $ 1), 9 9
ABC DEF	ABC DEF XY	1 1 1 1 1 1 9 9 0

5.2.3 Replicating Scalars

スカラーのレプリケーション

"Append"のスカラー引数は、他の引数と一致するために必要に応じて複製されます。次の例では、スカラー'*'がどのように複製されているかを確認しますが、ベクトル(1 $ '*') はパディングされています。

A	A, '*'	A,1 $ '*'
ABC DEF	ABC DEF ***	ABC DEF *

5.3 Stitching, or Joining Side-to-Side

ステッチング、またはサイド・バイ・サイド結合

二項動詞、。(コンマドット) は「ステッチ」と呼ばれます。式(x、。y) では、x の各項目には、対応する y の項目が追加され、結果の項目が生成されます。

a	b	a ,. b	A	B	A ,. B
pqr	uvw	pu qv rw	ABC DEF	UVW XYZ	ABCUVW DEFXYZ

5.4 Laminating, or Joining Face-to-Face

ラミネート、またはフェイス・ツー・フェイスへの結合動詞、 :(コンマコロン) は「ラミネート」と呼ばれます。(x、: y) の結果は、 常に 2 つの項目を持つ配列です。最初の項目は x で、2 番目の項目は y です

a	b	a ,: b

pqr	uvw	pqr uvw

x と y がテーブルの 場合、その結果を 1 つのテーブルとして上に置いて、その最初の次元に沿って長さ 2 の 3 次元配列を形成することができます。

A	B	A ,: B	$ A ,: B
ABC DEF	UVW XYZ	ABC DEF UVW XYZ	2 2 3

5.5 Linking
リンク

動詞 ; (セミコロン) は「リンク」と呼ばれます。箱のリストを作成するのに便利です。

'good' ; 'morning'	5 ; 12 ;1995
+----+-------+ |good|morning| +----+-------+	+-+--+----+ |5|12|1995| +-+--+----+

例えば、5; 12; 1995 の例では、(x; y) は常に(<x) 、(<y) ではないことに注意してください。"Link"はボックスのリストを作成するためのもので、正しい引数がすでにボックスのリストであることを認識します。生成する動詞(<x) 、(<y) を定義すると、

```
   foo =: 4 : '(< x) , (< y)'
```

我々はこれら 2 つを比較することができます：

1 ; 2 ; 3	1 foo 2 foo 3
+-+-+-+ |1|2|3| +-+-+-+	+-+-----+ |1|+-+-+| | ||2|3|| | |+-+-+| +-+-----+

5.6 Unbuilding Arrays
配列の作成

我々は、4 つの二項動詞を見てきました： "追加" (,) ,” スティッチ"、(,.) 、"ラミネート"(:)と "リンク" (;) 。これらはそれぞれモナドケースを持っています。

5.6.1 Razing

レイジング

モナド`;` 「レイズ」と呼ばれています。引数の要素をアンボックスし、それらをリストにアセンブルします。

`B =: 2 2 $ 1; 2; 3; 4`	`; B`	`$; B`

`+-+-+`
`|1|2|`
`+-+-+`
`|3|4|`
`+-+-+`
	`1 2 3 4`	`4`

5.6.2 Ravelling

ラベリング

モナドは、「ラヴェル」と呼ばれています。引数の要素をリストにアセンブルします。

`B`	`, B`	`$ , B`

`+-+-+`
`|1|2|`
`+-+-+`
`|3|4|`
`+-+-+`
	`+-+-+-+-+`
	`|1|2|3|4|`
	`+-+-+-+-+`
		`4`

5.6.3 Ravelling Items

ラベリング項目

モナド、。「ラヴェルアイテム」と呼ばれます。これは、引数の各項目を個別にラブして表を形成します。

`k =: 2 2 3 $ i. 12`	`,. k`

`0 1 2`
`3 4 5`
 	
`6 7 8`
`9 10 11`
	`0 1 2 3 4 5`
	`6 7 8 9 10 11`

「ラブ項目」は、1 列の表をリストから外すのに便利です。

`b`	`,. b`

`uvw`	`u`
	`v`
	`w`

5.6.4 Itemizing
アイテム化

モナドは、 : 「箇条書き」と呼ばれています。これは、任意の配列のうち、先頭の寸法添加することにより、1 項目のアレイを作る 1。

A	,: A	$,: A
ABC DEF	ABC DEF	1 2 3

5.7 Arrays Large and Small
大小の配列

これまで見てきたように、配列は$動詞で構築できます。

```
   3 2 $ 1 2 3 4 5 6
1 2
3 4
5 6
```

小さな配列の場合、内容を 1 行に記述することができるため、$を使用する代わりに、次元を明示的に指定する必要がありません。

> 1 2; 3 4; 5 6	1 2 , 3 4 ,: 5 6
1 2 3 4 5 6	1 2 3 4 5 6

大きなテーブルを構築するには、便利な方法は次のとおりです。最初に、ここには「ユーティリティ」動詞(現在の目的には有用な動詞ですが、今はその定義を勉強する必要はありません）。

```
   ArrayMaker =:  "。;;。_2
```

ArrayMaker の目的は、スクリプトの行から行ごとに数値表を作成することです。

```
   table =: ArrayMaker 0: 0
1 2 3
4 5 6
7 8 9
)
```

table	$ table
1 2 3 4 5 6 7 8 9	3 3

(ArrayMaker の仕組みについては、第 17 章を参照してください）。ボックスの配列は同じ方法でスクリプトから入力することもできます：

```
  X =: ArrayMaker 0: 0
'hello'; 1 2 3; 8
'Waldo'; 4 5 6; 9
)
```

X	$ X
+ ----- + ----- + - + | hello | 1 2 3 | 8 | + ----- + ----- + - + | Waldo | 4 5 6 | 9 | + ----- + ----- + - +	2 3

☆第 5 章終了。

第 6 章：Indexing
インデックス作成

インデックス付けは、配列の要素を位置で選択するための名前です。このトピックでは、要素の選択、選択した要素の並べ替えによる新しい配列の整理、配列の選択要素の修正または更新などを行います。

6.1 Selecting
選択

動詞{(左括弧）は "From"と呼ばれます。式(x {v）は、x から与えられた位置に従って v から要素を選択します。例えば、リコールからチャプター02 と、その L がリストされ、その後の項目の位置 L が、ように 0~1 の番号が付けとされています。式(0 {L）は、の最初の項目の値を与える L と 1 {L を 2 番目の項目を与えます。

L =: 'abcdef'	0 {L	1 {L
abcdef	a	b

{ の左の引数は"インデックス"と呼ばれます。

6.1.1 Common Patterns of Selection
選択の共通パターン

いくつかのアイテムを一緒に選択することができます：

L	0 2 4 {L
abcdef	ace

L から選択されたアイテムは、複製され、並べ替えられることができる：

L	5 4 4 3 {L
abcdef	feed

インデックス値は負の値になります。_1 の値は最後の項目を選択し、_2 は最後の項目の次の項目を選択します。正負の指標が混在することがあります。

L	_1 {L	_2 1 {L
abcdef	f	eb

たとえば、行 1 の列 2 の表の単一の要素が索引(<1; 2）で選択されます。

T =: 3 3 $ 'abcdefghi'	(<1; 2) {T
abc def ghi	f

指定された行と列のすべての要素をテーブルから選択して、より小さなテーブル(サブアレイと呼ばれる）を生成することができます。行 1 と 2、列 0 と 1 からなるサブアレイを選択する には、インデックス(<1 2; 0 1）を使用し、

```
T       (<1 2; 0 1) {T
abc     de
def     gh
ghi
```

テーブルから完全な行を選択することができます。表は項目のリストであり、各項目は行であることを思い出してください。したがって、テーブルから行を選択することは、リストから項目を選択することと同じです。

```
T       1 {T    2 1 {T
abc     def     ghi
def             def
ghi
```

完全な列を選択するには、すべての行を選択するのが簡単です。

```
T       (< 0 1 2 ; 1 ){ T
abc     beh
def
ghi
```

他の可能性があります：以下を参照してください。

6.1.2 Take, Drop, Head, Behead, Tail, Curtail
テイク、ドロップ、ヘッド、ビーヘッド、テール、カール

次に、簡単な索引付けの簡単な形式を提供する動詞のグループを調べます。組み込みの動詞 {があります。(左のブレース・ドット、「Take」と呼ばれます）。リスト L の最初の n 個の項目は、 (n {L}

```
L       2 {. L
abcdef  ab
```

L から n 個の項目を (n {L}) で 取り、n が L の長さより大きい場合、結果は長さ n にパディングされ、必要に応じてゼロ、スペースまたは空のボックスが埋め込まれます。
たとえば、与えられた文字列から正確に 8 文字の文字列を作成する必要があるとします。何らかの説明があります。これは 8 より長くても短くても構いません。短い場合はスペースで埋めます。

```
s =: 'pasta'    # s     z =: 8 {. s     # z
pasta           5       pasta           8
```

組み込みの動詞}があります。 (右括弧の点、「ドロップ」と呼ばれます）。L の最初の n 個の項目を除くすべてが(n) L によって選択される 。

L	2}. L
abcdef	Cdef

L の最後の n 個の項目は、(-n) {で選択され ます。L。最後の n を除くすべてが(-n) }によって選択され ます。L

L	_2 {. L	_2 }. L
abcdef	ef	abcd

n = 1 の 特別な場合には、Take と Drop の略語があります。リストの最初の項目はモナド{}で選択されます。 (左括弧のドット、 "頭"と呼ばれる）。最初のものを除くすべてが}によって選択されます。(右括弧内の "Behead"）。

L	{. L	}. L
abcdef	a	bcdef

リストの最後の項目は、モナド{:(左括弧のコロン、「テール」と呼ばれます）によって選択されます。最後のものを除いて、すべてが}:(右括弧のコロン "Curtail"で選択されます。

L	{: L	}: L
abcdef	f	abcde

6.2 General Treatment of Selection
選択の一般的な扱い

それはいくつかの用語を持つのに役立ちます。一般的には、n 次元の配列を持ちますが、3 次元の配列を考えます。プレーン番号、行番号、列番号を指定することで、1 つの要素が選択されます。私たちは、平面が第 1 軸に沿って、第 2 軸に沿った行と、第 3 軸に沿って、並んでいると言う。

インデックス作成に特別な表記はありません。むしろ{の左の議論は、選択および再編成を表現または符号化するデータ構造である。このデータ構造は、どのような方法でも簡単に構築できます。それはどのようにそれを構築するための説明です。

6.2.1 Independent Selections
独立した選択

インデックス作成の一般的な式は、index {arrav}の形式です。ここで index はスカラの配列です。インデックス内の各スカラーは独立した独立した選択をもたらし、結果は一緒にアセンブルされます。

L	0 1 {L
abcdef	ab

6.2.2 Shape of Index
索引の形状

結果の形状は、インデックスの形状に依存します。

L	インデックス=: 2 2 $ 2 0 3 1	インデックス{L
abcdef	2 0 3 1	ca db

インデックスは、範囲内になければならない- : #1 に(: #1) -1。

L	#L	_7 {L	6 {L
abcdef	6	エラー	エラー

6.2.3 Scalars
スカラー

インデックス 内の各スカラーは、単一の数字かボックスのいずれかです(もちろん、1 つがボックスの場合はすべてです）。スカラーが単一の数字であれば、配列から項目を選択します。

A =: 2 3 $ 'abcdef'	1 {A
abc def	def

ただし 、インデックスのスカラーがボックスの場合、連続する軸に適用されるセレクタのリストが含まれます。この目的のためにボックスがどこに使用されているかを示すために、ボックス関数に SuAx という名前を使用することができます。

```
SuAx =: <
```

次の例では、行 1 の列 0 の要素を A から選択します。

A	(SuAx1 0) {A
abc def	d

6.2.4 Selections on One Axis
1 軸の選択

連続形の軸のためのセレクタのリストに(SuAx の P、R、C）は、と言うの各 P、R 及び C はスカラーです。このスカラーは数字かボックスのいずれかです(そしてボックス化されている場合はすべてです）。数値は軸の 1 つを選択します。最後の例のように、必要に応じて 1 つの平面、行または列が選択されます。

ただし、セレクタがボックスの場合は、すべて同じ軸に適用可能な選択リストが含まれます。この目的のためにボックスがどこに使われているかを示すために、ボックス関数には Sel という名前を使うことができます。

```
   Sel = : <
```

たとえば、行 1 の A 要素から選択するには、列 0 2:

```
A     (SuAx (Sel 1), (Sel 0 2)) { A
abc   df
def
```

6.2.5 Excluding Things
物の除外

特定の軸上のものを選択する代わりに、さらに別のレベルのボクシングで囲まれた物件番号のリストを提供することで、物事を除外することができます。この目的のためにボックスがどこに使われているかを示すために、ボックス関数に Excl という名前を使用することができます。

```
   Excl =: <
```

たとえば、行 0 の A 要素から選択するには、列 1 を除くすべての列を選択します。

```
A     (SuAx (Sel 0), (Sel (Excl 1))) { A
abc   ac
def
```

何も除外しない、つまり空のリスト(0 $ 0) を除外することで特定の軸上のすべてのものを選択することができます。たとえば、行 1 の A 要素から選択するには、すべての列を選択します。

```
A     (SuAx (Sel 1),(Sel (Excl 0$0))) { A
abc   def
def
```

6.2.6 Simplifications
簡略化

式(Excl 0 $ 0) は、ボックス化された空のリストを示します。これには組み込みの] 略語、つまり(a :)(「Ace」と呼ばれる文字 - コロン）があります。このコンテキストでは、「すべて」を意味すると考えることができます。

```
A     (SuAx(Sel 1) 、(Sel a :)) {A
abc   def
def
```

形式(SuAx p、q、...、z) の任意のインデックスで、最後のセレクタ z が「すべて」形式 (Sel(Excl 0 $ 0)) または(Sel a :)の場合、省略された。

```
A     (SuAx (Sel 1),(Sel a:)) {A     (SuAx (Sel 1)) {A
```

abc def	def	def

フォームのインデックス（SuAx(Sel p）、(Sel q）、...）で「すべて」フォームが完全に存在しない場合、インデックスは(SuAx p; q; ...）。たとえば、行 1、列 0 および 2 の要素を選択するには、次のようにします。

A	(SuAx(Sel 1）、(Sel 0 2））{A	(SuAx 1; 0 2）{A
abc def	df	df

最後に、すでに見てきたように、各軸で 1 つだけを選択すると、単純なボックス化されていないリストで十分です。たとえば、行 1、列 2 の要素を選択するには、次のようにします。

A	(SuAx 1;2) { A	(SuAx 1 2) { A
abc def	f	f

6.2.7 Shape of the Result
結果の形状

仮定 B は、 3 次元配列です。

```
B = 10 + i である。3 3 3
```

我々は定義 p はの第 1 の軸に沿って平面を選択する B、及び R は第 2 の軸に沿って行を選択し、c は 第 3 の軸に沿って列を選択します。

```
p =: 1 2
r =: 1 2
c =: 0 1
```

p; r; c で選択すると、結果 R の形は p、r、c の形の連結であることがわかります

B	R =: (< p;r;c) { B	$ R	($p),($r),($c)
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36	22 23 25 26 31 32 34 35	2 2 2	2 2 2

B は 3 次元であり、R も同様である。私たちが期待するように、セレクタ(r、say）が長さ 1 のリストであるとき、この連結連結は保持されます：

r =: 1 $ 1	S =: (< p;r;c){B	$ S	($p),($r),($c)
1	22 23 31 32	2 1 2	2 1 2

セレクタ r がスカラのときは連結の形が成り立ちます。

r =: 1	T =: (< p;r;c){B	$ T	($p),($r),($c)	$ r
1	22 23 31 32	2 2	2 2	

この最後の例では、r はスカラーなので、r の形は 空のリストなので、r に対応する軸 は消えてしまい、結果 T は 2 次元です。

6.3 Amending (or Updating) Arrays
配列の修正(または更新)

時には、比較的少数の位置で新しい値を除いて、既存の配列と同じ配列を計算する必要があります。我々は、選択された位置でアレイを「更新する」または「修正する」と言うかもしれない。配列を修正するための J 関数は}
(右中括弧、 "Amend"と呼ばれます)。

6.3.1 Amending with an Index
インデックスによる修正

配列を修正するには、次の 3 つが必要です。
元の配列
原稿が改訂されるべき位置の指定。これは、{上記のインデックスとまったく同じインデックスである可能性があります。
指定された位置にある既存の要素を置き換える新しい値。
したがって、改正を行うための J 式は、一般的な形式をとることができます:

```
        newvalues index} original
```

たとえば、リスト L を修正して最初のアイテム(インデックス 0) を '*' で
置き換えるには:

L	new=:'*'	index=:0	new index } L
abcdef	*	0	*bcdef

}は副詞であり、二項修正動詞(index) を生成するために index を引数にとります。

```
   ReplaceFirst =: 0}
   '*' ReplaceFirst L
* bcdef
```

(index) }は他のダイアディックのような動詞で、

通常の方法で値を返します。したがって、必要性を修正して配列を変更するには、結果全体を古い名前に再割り当てする必要があります。したがって、修正はしばしば次のパターンで行われる。

```
                A  =:  new index } A
```

J システムは、これがデータの不必要な移動を伴わない効率的な計算であることを保証する。行 1 の列 2 の表を修正するには、次のようにします。

A	'*' (<1 2)} A
abc def	abc de *

複数の要素を修正するには、新しい値のリストを指定し、インデックスによって選択された値のリストを置き換える

L	'*: #' 1 2} L
abcdef	a * #def

6.3.2 Amending with a Verb
動詞による修正

仮定し、Y は数字のリストであり、我々は与えられた値を超えたすべての数字ようにそれを修正したい X が置き換えられている X。(この例では、この関数の組み込み J 動詞(<。) は無視してい ます) 。
Y を修正する 指標は、X と Y から計算する必要があります。インデックスを計算する関数 f を次に示します。

```
   f = 4:  '(y> x) : #(i。: #y) '
```

X = 100	Y =: 98 102 101 99	Y> X	X f Y
100	98 102 101 99	0 1 1 0	1 2

補正は、上記のように、(X f Y) のインデックスを供給することによって行われる。

Y	X(X f Y) } Y
98 102 101 99	98 100 100 99

副詞を"改正" }表現でき (X(X F の Y) } Y は) と略記される(X は F} Y) 。

X(X f Y) } Y	X f} Y
98 100 100 99	98 100 100 99

以来}副詞であり、それは引数の指標のいずれかとして受け入れることができる(Y F X) 又は動詞 F。

```
   cap =: f}
```

```
   10 キャップ 8 9 10 11
   8 9 10 10
```

動詞の場合に注意 f は副詞の引数として供給される}、その後、f は、それは無視することができるが、ダイアドでなければならない X または Y を。

6.3.3 Linear Indices
線形指数

私たちはちょうど動詞で改訂リストを見てきました。動詞の目的は、修正する場所を見つけることです。つまり、リスト内の値から修正する指標を計算することです。リストではなくテーブルでは、インデックスは 2 次元でなければならず、インデックスを構築する際の動詞のタスクはそれに応じてより困難になります。テーブルを線形リストに平坦化し、それをリストとして修正し、リストを再びテーブルに再構築する方が簡単です。
たとえば、テーブルがあるとします。

```
   M =: 2 2 $ 13 52 51 14
```

次に、索引検索動詞 f を使用して、平坦化、修正、および再構築を次のように示します。

M	LL =: M	Z = 50f} LL	($ M) $ Z
13 52 51 14	13 52 51 14	13 50 50 14	13 50 50 14

しかし、より良い方法があります。最初に、我々の索引発見動詞 f は M では なく(LL =: 、M) でなく引数として取ることに注意してください。したがって、M の元の形状に関する情報 は、インデックスファインダ f では利用できない。この例では、これは問題ではありませんが、一般的に、M の形状と値の両方に依存するようにすることができます。f が引数の全体として M を取った方が良いでしょう。この場合、f は独自の平坦化を行わなければなりません。したがって、f:

```
   f = 4 : 0
y =., y
(y> x) # (i. #y)
)
```

M	50 f M
13 52 51 14	1 2

インデックスファインダ f は、引数として配列をとり、フラット化された配列、いわゆる「線形インデックス」にインデックスを渡します。この新しい f を使用した修正プロセスは、次のように表示されます。

M	($M) $ 50 (50 f M) } (, M)
13 52	13 50

51 14	50 14

最後に、与えられた f が線形インデックスを出力すると、(}) は最後の式を次のように省略することができます。

M	50 f} M
13 52 51 14	13 50 50 14

6.4 Merging Together the Items of an Array
配列の項目を併合する

私たちはそれを見ました}は副詞です。これは、引数 m に適用して動詞(m) を生成することができます。この動詞のダイアディックの場合を上回って、配列の修正に使用されています。今、(m) のモナドの場合を見てみましょう 。この動詞は「修正項目」として知られています。仮定 T はテーブルです。

```
   T =: 3 4 $ 'ABCDEFGHIJKL'
```

してみましょう m はインデックスのリストです。m の各項目は、T の対応する列へのインデックスであることを意図しています。

```
  m =: 1 2 0 2
```

動詞 m を T に適用すると、各列を別々に索引付けすることによって結果が生成されることがわかります。

T	m	(m }) T
ABCD EFGH IJKL	1 2 0 2	EJCL

結果を記述する別の方法は、T の項目がベクトル m に従って結合されるということである。副詞の引数 }は、2 つの代替配列の 1 番目または 2 番目を選択するブール値配列です。例えば：

```
   A =: 'pot'  NB. first alternative
   B =: 'dig'  NB. second alternative
   1 0 1 }  A ,: B
dog
```

☆第 6 章終了。

第 7 章：Ranks
ランク

配列のランクは次元数です。スカラーはランク 0、数字のリストはランク 1、ランク 2 のテーブルなどがあります。
この章の主題は、動詞が適用されるときにどのように議論のランクが考慮されるかです。

7.1 The Rank Conjunction
ランク結合

まず、いくつかの用語。アレイは、いくつかの異なる方法で「セル」に分割されているとみなすことができます。このように、

```
   M =: 2 3 $ 'abcdef'
   M
abc
def
```

2 個の細胞、ランク 1 のそれぞれに 6 個の細胞ランク 0 のそれぞれに分割、または分割されているとみなすことができる、またはランク 2 の単一セルであるとしてランクのセル k が呼び出される k 個の β 細胞を。

7.1.1 Monadic Verbs
モナド動詞

ボックス動詞(モナド < ）は、引数のランクにかかわらず、引数全体に 1 回だけ適用され、単一のボックスを生成します。

L =: 2 3 4	<L	M	<M
2 3 4	+-----+ | 2 3 4 | +-----+	abc def	+---+ | abc | | def | +---+

ただし、各セルを個別にボックス化することもできます。"(二重引用符、"ランク "と呼ばれる）結合詞があり、各スカラー、すなわち各 0 セルにボックスを書く(< " 0 ）。

M	< " 0 M	< " 1 M	< " 2 M
abc def	+-+-+-+ | a | b | c | +-+-+-+ | d | e | f | +-+-+-+	+---+---+ | abc | def | +---+---+	+---+ | abc | | def | +---+

一般的なスキームは、表現(u " k y）において、モナド動詞 u が y の各 k-cell に別々に適用されることである。
私たちは配列の k-cell を表示する動詞を定義することができます。

```
   cells  =: 4 : '< " x y'
```

M	0 セル M	1 セル M

```
abc
def
```

```
+ - + - + - +
| a | b | c |
+ - + - + - +
| d | e | f |
+ - + - + - +
```

```
+ --- + --- +
| abc | def |
+ --- + --- +
```

7.1.2 Dyadic Verbs
二項動詞

テーブルを考えると、各行に個別の数値を掛けるにはどうすればよいですか？我々は、"0 セルによる 1 セルの乗算"と理解できる動詞(* " 1 0") を乗算します。例えば、

```
X =: 2 2 $ 0 1 2 3
```

```
0 1
2 3
```

```
Y =: 2 3
```

```
2 3
```

```
X (* " 1 0) Y
```

```
0 2
6 9
```

一般的なスキームは、式

```
X (u " (L R)) Y
```

意味：ダイアド適用 u とからの L-細胞からなる各対に別々X とから、対応する R-セル Y。各列に別々の数を掛けるために、x の各 1 セルと y の単独セルとを結合する

```
X
```

```
0 1
2 3
```

```
Y
```

```
2 3
```

```
X (* " 1 1) Y
```

```
0 3
4 9
```

7.2 Intrinsic Ranks
組み込みランク

J では、すべての動詞は、自然言語の、または本質的な、その議論のランクを持っています。例をいくつか示します。最初の例では、次の点を考慮してください。

```
*: 2
```

```
4
```

```
*: 2 3 4
```

```
4 9 16
```

ここでは、算術関数「正方形」は当然単一の数字(0 セル）に適用されます。ランク 1 の配列(リスト）が引数として与えられると、その関数は引数の各 0 セルに別々に適用されます。言い換えれば、(モナド）*: の自然ランクは 0 です。 別の例として、組み込みの動詞 #. があります。(「ベース 2」と呼ばれるハッシュドット）。引数はバイナリ表記の数値を表すビット列(リスト）で、その数値の値を計算します。例えば、バイナリでは 1 0 1 は 5

```
   #. 1 0 1
5
```

動詞 #. ビットリスト、つまり 1 セルに自然に適用されます。ランク 2 の配列(テーブル）が引数として指定されると、動詞は各セルに別々に、つまりテーブルの各行に適用されます。

```
t =: 3 3 $ 1 0 1 0 0 1 0 1 1
```

```
#. t
```

1 0 1 0 0 1 0 1 1	5 1 3

したがって、モナドの自然なランク #. は 1。
3 つ目の例では、既に見てきたように、<のモナドの場合は、その議論のランクが何であれ、その議論の全体に一度だけ適用されます。従って、<の自然なランクは無限大の数であり、無限大であり、_で表される。これらの例はモナド動詞を示した。同じように、すべての二項動詞は 2 つの自然なランクを持ちます。例えば、ダイアディックの天然のランク+である 0 ので+は、左右に数(ランク 0）をとります。一般的に、動詞はモナドとダイアディックの両方のケースを持ち、それゆえに「内在的ランク」と呼ばれる 3 つのランクを持っています。
動詞の内在的ランクは、組み込みの副詞 b の助けを借りて示される。(小文字の b ドット、「基本特性」と呼ばれます）。任意の動詞 u に対して、式 u b。0 は、モナド、左、右の順にランクを与えます。

*: b. 0	#. b. 0	< b. 0
0 0 0	1 1 1	_ 0 0

利便性のために、ランク結合は、「単一のランク(モナドの場合）または 2 ランク(ダイアダの場合）または 3 ランク(相反する動詞の場合）からなる正しい引数を受け入れることができます。
1 ランクまたは 2 ランクは、次のように自動的に 3 つに拡大されます。

(<"1) b. 0	(<"1 2) b. 0	(<"1 2 3) b. 0
1 1 1	2 1 2	1 2 3

7.3 Frames
フレーム

仮定 u は列を合計し、次に列の和を合計することによって、テーブル内のすべての数値を合計動詞であることです。私たちは、あることを指定 u はモナドランク 2 を持つことです。

```
   u =: (+/) @: (+/) " 2
```

w =: 4 5 $ 1	u w	u b. 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	20	2 2 2

4 次元配列 A の形状が 2 3 4 5 であると仮定する。

```
   A =: 2 3 4 5 $  1
```

私たちは、A を 2 行 3 列の配列として考えることができます。各セルは 4 行 5 列です。今、計算(u A）を考えてみましょう。動詞 u はランク 2 であり、2 セルごとに別々に適用され、2 行 3 列の結果が得られます。

(ので、各結果はスカラーであり、u はスカラーを生成する）、ひいては全体的な結果は、3 つのスカラーにより 2 であろう。

u A	$ u A
20 20 20 20 20 20	2 3

形状 2 3 は、その 2 つのセルに関して A の「フレーム」と呼ばれ、その 2 つのフレームは短いものである。A の k-フレームは、最後の k 次元を A の形状から、または等価的に A の k-セルの配列の形状から落とすことによって与えられる。

```
frame =: 4 : '$ x cells y'
```

$ A	$ A
2 frame A	2 frame A

一般に、動詞 u がランク k を持ち、各 k-セルから形 s のセルを計算すると仮定する。(同じのは、我々は、各セルのために、想定されています）。次に、全体的な結果の形状(U A）がある：の k 枠形状続く S。
これが事実であることを実証するために、我々は見つけることができる k 個から U の最初の(モナド）ランクとして、U:

```
   k =: 0 { u b. 0
```

形状 s は、A の典型的な k-セルに u を適用することで見つけることができます。

```
   s =: $ u  0 { > (                k cells A)
```

この例では、u はスカラーを生成するため、形状 s は空のリスト です。

k	s	kfr =: k frame A	kfr, s	$ u A
2		2 3	2 3	2 3

ここでは、動詞 u がその引数の各セルに同じ形の結果を与えると仮定しました。これは必ずしもそうではありません - 下記の「結果の再構成」の項を参照してください。

7.3.1 Agreement
契約

ダイアドには 2 つの固有のランクがあり、1 つは左の引数、もう 1 つは右の引数です。これらのランクが動詞 u に対して L と R であると仮定する。
場合 u は引数に適用され、X および Y、 U はからの L-細胞からなる各対に個別に適用される X とから、対応する R-セル Y。たとえば、dyad u にランク(0 1) があるとします。これは、X からの 0 セルと Y からの 1 セルを組み合わせたものです。

```
        u =: < @ , " 0 1
        X =: 2 $ 'ab'
        Y =: 2 3 $ 'ABCDEF'
```

X	Y	X u Y

ab	ABC DEF	+----+----+ |aABC|bDEF| +----+----+

ここで、X の 0 フレームは Y の 1 フレームと同じであることに注意してください。これらの 2 つのフレームは合っていると言われている。

X	Y	$X	$Y	0 frame X	1 frame Y
ab	ABC DEF	2	2 3	2	2

これらの 2 つのフレームが同じでない場合はどうなりますか？一方が他方の接頭辞であれば、彼らはまだ同意することができます。1 つのフレームがベクトルである場合すなわち、F、及び他のフレームのように書くことができる(F、G) のいくつかのベクトルのために、ここに例があります。

```
   X =: 2 3 2 $ i. 12
   Y =: 2     $ 0 1
```

ランク(0 0) を持つ + などのダイアドでは、X の 0 フレームと Y の 0 フレームに興味があります。

X	Y	0 frame X	0 frame Y	X+Y
0 1 2 3 4 5 6 7 8 9 10 11	0 1	2 3 2	2	0 1 2 3 4 5 7 8 9 10 11 12

我々は 2 つのフレームであることがわかり 2 および 2 3 2 及びそれらの差 g がことである 3 2。

ここで、Y はより短いフレームを有する。次に、Y の各セルは、X の単セルだけでなく、むしろセルの 2 3 型アレイに対応します。

このような場合、Y の細胞は自動的に複製されて、同一の細胞の 3 2 -形状のアレイを形成する。事実上、より短いフレームは長さに合わせられ、より長いフレームに同意する。ここに例があります。

X	Y	YYY =: (3 2&$)"0 Y	X + YYY	X + Y
0 1 2 3 4 5 6 7 8 9 10 11	0 1	0 0 0 0 0 0 1 1 1 1 1 1	0 1 2 3 4 5 7 8 9 10 11 12	0 1 2 3 4 5 7 8 9 10 11 12

私たちが見てきたのは、下位の引数が自動的に複製され、上位の引数に一致する方法です。これは、1 つのフレームがもう一方の接頭辞であれば可能です。さもなければ、長さエラーがあります。問題のフレームは、動詞の本質的な二項のランクによって決定されます。
引数 x と y に対して、u がランク L と R を持つダイアドであり、x の L フレームが f である場合、g と y の R フレームは f となる(y フレームを短くする)
次いで、(xuy) として計算される(徐(x u (g& $)"R y) 。

7.4 Reassembly of Results
結果の再構成

ここでは、別々のセルの計算結果を全体的な結果にどのように再構成するかを簡単に見ていきます。
たとえば、その引数に動詞を適用するフレームが f であるとします。次に、結果の f 型枠組みの中で、個々の結果をその場所に詰め込んだものとして視覚化することができます。各個々の結果、セルが同じ形状を有している場合、S は言い、次に全体的な結果の形状になります(S、F) 。ただし、個々の結果がすべて同じ形状であるとは限りません。例えば、次の動詞検討 R スカラーかかり、y とし、rank-生成 Y 結果。

```
   R =: (3 : '(y $ y) $ y') " 0
```

R 1	R 2
1	2 2 2 2

R が配列に適用される とき、全体的な結果は、それぞれの別個の結果がボックスの f 字型の配列内の適切なボックスに充填されることを想定することによって説明することができる。そして、すべてがすべて一緒にアンボックスされます。すべてのセルを同じ形にする必要がある場合は、余白と余分な寸法を供給する unboxing であることに注意してください。

(R 1); (R 2)	> (R 1) ; (R 2)	R 1 2
+-+---+ |1|2 2| | |2 2| +-+---+	1 0 0 0 2 2 2 2	1 0 0 0 2 2 2 2

したがって、結果全体の形状は、(f、m) によって与えられる。ここで、m は個々の結果の中で最大の形状である。

7.5 More on the Rank Conjunction
ランク接続詞の詳細

7.5.1 Relative Cell Rank
相対セルランク

ランク結合は、ランクに対して負の数を受け入れます。したがって式(U| 1 v は) u はのランクよりも低いランク 1 のセルに適用されることを意味し、Y の項目に、すなわち、Y。

X	$ X	< " _1 X	< " _2 X

```
0  1
2  3
4  5

6  7
8  9
10 11
```

```
2 3 2
```

```
+---+-----+
|0 1| 6  7|
|2 3| 8  9|
|4 5|10 11|
+---+-----+
```

```
+---+---+-----+
|0 1|2 3|4 5  |
+---+---+-----+
|6 7|8 9|10 11|
+---+---+-----+
```

7.5.2 User-Defined Verbs
ユーザー定義の動詞

ランク接続詞は、「ユーザー定義の動詞の特別な意味を持っている意義はそれだけでその考慮動詞を定義するために私達にできることです。 『自然』ランクを：我々はそれが上位の引数に適用することができるという可能性を無視し、他のでは。動詞が自然なランクの引数にのみ適用されると仮定して定義を書くことができます。その後、ランク付けを使用して定義に仕上げることができます。
数 n の階乗は、1 から n までの数の積である。したがって、(現時点で J の組み込み動詞を無視する！）我々は階乗を直接

```
   f =: */ @: >: @: i.
```

なぜなら i. n が数字を与えます... 0 1(N-1) 、および >: i. n は 1 2 ... n を与える。見てみよう：

f 2	f 3	f 4	f 5
2	6	24	120

ベクトルは引数として期待通りに働くでしょうか？

```
   f 2 3
4 10 18
```

明らかにそうではありません。理由は、(f 2 3) が計算によって(i 2 3) 、(i 2 3) が(i 2) に続いて(i 3) を意味しないことです。対処法は、引数がスカラー(ランク 0 のセル）ごとに f を別々に適用するように指定することです。

```
   f  =: (*/ @: (>: @: i.)) " 0
   f 2 3 4 5
2 6 24 120
```

階級結合の重要性の第 2 の例として、明示的に定義された動詞を見る。ここでのポイントは、引数が特定のランク、つまりスカラーであるという前提で定義を書くことができ、後でランクの引数に拡張することだけを扱うことが有用であることを繰り返すことです。
明示的な動詞については、その内在的なランクは常に無限であると仮定されることに注意してください。これは、J システムは動詞が実行されるまで定義を見ていないからです。ランクは無限であるため、明示的な動詞の全体の引数は常に単一のセル(またはダイアドのセルのペア）として扱われ、複数のセルを扱うための自動拡張はありません。
たとえば、数値の絶対値は動詞によって計算できます。

```
   abs =: 3 : 'if. y < 0 do.
```

```
 y else. y end.'
```

abs 3	abs_3
3	3

abs は明示的に定義されている ので、モナド(第 1) ランクは無限大です：

```
   abs b. 0
_ _ _
```

つまり、abs が配列 v に適用された場合、任意のランクの値が 1 回だけ適用され、その結果が y または-y になることが定義からわかります。他に可能性はありません。
それがあれば確かにそうである Y は、次いでベクターである (Y <0) のベクトル結果をもたらすが、式(Y は<0 の場合) 1 つの決定を行います。(この決定は実際には y <0 の全体では なく、最初の要素のみに基づいて行われます。詳細については第 12 章を参照してください）。したがって、引数に陽性と陰性の両方が含まれる場合、この決定は引数の一部で間違っていなければなりません。

```
   abs 3 _3
3 _3
```

したがって、上記のように abs を定義すると、それは引数の各スカラーに別々に適用されると言うことが重要です。したがって、abs のより良い定義は次のようになります。

```
   abs =:(3 : 'if. y < 0 do. -y else. y end.') " 0
   abs 3 _3
3 3
```

☆第 7 章終了。

第 8 章：Composing Verbs
動詞の作成

この章では、2 つの動詞を組み合わせて新しい複合動詞を生成する演算子に関係しています。

8.1 Composition of Monad and Monad
モナドとモナドの構成

リコール第 03 章構成の組み合わせは、@:（At Colon と呼ぶ）。与えられた動詞の和と平方和は、複合動詞、二乗和を定義することができます。

```
   sum =: + /
   square =: *:
```

sumsq =: sum @: square	sumsq 3 4
sum@:square	25

一般的なスキームは、f と g がモナド

```
(f @: g) y    means   f (g y)
```

特に、f が全体の結果(g y) に適用されることに注意してください。たとえば、g がテーブルの各行に個別に適用されると仮定すると、次のようになります。

```
   g =: sum "1
   f =: <
```

y =: 2 2 $ 1 2 3 4	g y	f g y	(f @: g) y
1 2 3 4	3 7	+ --- + | 3 7 | + --- +	+ --- + | 3 7 | + --- +

私たちは今、最も基本的な構成を見ました。ここでいくつかのバリエーションを見てみましょう。

8.2 Composition:Monad And Dyad
構成：モナド(単項)とダイアド(両項)

場合 f はモナドであり、G ダイアドであり、次いで、(f @: G) のようにダイアディック動詞であります。

```
x (f @: g) y    means    f (x g y)
```

例えば、2 つのベクトル x と y の積の和は、「スカラー積」と呼ばれます。

```
   sp =: + / @: *
```

x =: 1 2	y =: 2 3	x * y	+ /(x * y)	x sp y
1 2	2 3	2 6	8	8

最後の例は、式（x（f：g）y）において、動詞 f が（x g y）の全体に一度適用されることを示した。

8.3 Composition: Dyad And Monad
構成：ダイアド(両項)とモナド(単項)

接続詞 &:(ampersand colon,"Appose"と呼ばれる）は、ダイアド F とモナドを構成する。スキームは次のとおりです。

```
x (f &: g) y   means   (g x) f (g y)
```

たとえば、2 つのリストの長さが等しいかどうかは、動詞(= &: #）でテストできます。

```
   eqlen =: =&: #
```

x	y	#x	#y	(#x）=(#y）	x eqlen y
1 2	2 3	2	2	1	1

ここで、f は(g x)と(g y)の全体に一度適用されます。

8.4 Ambivalent Compositions
両親媒性組成物

レビューするために、我々は構成のための 3 つの異なるスキームを見た。これらは：

```
(f @: g)y = f(g y)
x(f @: g)y = f(x g y)
x(f &: g)y =(g x)f(g y)
```

第 4 のスキームがあり、

```
(f &: g)y = f(g y)
```

これは明らかに最初のものと同じです。この明らかな重複は、私たちが二元的な定義を書くことに興味がある場合、つまり、モナドとダイアディックの両方の場合に役立つ場合があります。

動詞 g が相反である場合、第 1 および第 2 のスキームから、コンポジション f @: g もまた相反的であることに留意されたい。

g は、y の逆数である（| .y）と x 個の場所による y の回転である（x | .y）を持つ、相反する組み込み動詞です。

y =: 'abcdef'	(< @: |.) y	1 (< @: |.) y
abcdef	+------+ |fedcba| +------+	+------+ |bcdefa| +------+

上の第 3 および第 4 のスキームから、動詞 f が相反である場合、 (f &: g)は相反であることになる。たとえば、f が動詞 %(reciprocal または divide)で、g が *: (平方)の時。

% *: 2	(% &: *:) 2	(*: 3) % (*: 2)	3 (% &: *:) 2
0.25	0.25	2.25	2.25

8.5 More on Composition: Monad Tracking Monad

合成の詳細

接続詞のある@(「頂上」と呼ばれるのは、)。これは@: の組み合わせです。以下は、(f @: g)と(f @ g)の違いを示す例です。

```
   y =: 2 2 $ 0 1 2 3
```

y	f	g	(f @: g) y	(f @ g) y
0 1 2 3	<	sum"1	+---+ |1 5| +---+	+-+-+ |1|5| +-+-+

私たちは、(f @: g)動詞 f が一度適用されるのを見ます。しかし、と(F する@ G) 、各個別のアプリケーションのために、G の対応するアプリケーションが存在する F は。我々は、f のアプリケーションが g のアプリケーションを追跡すると言うことができます。

第 7 章 から、動詞は一般的にモナド、左、右の 3 つのランクを持ち、動詞 f の場合、これらのランクは式 f b によって得られることを思い出してください。0。例えば

g	g b. 0
sum"1	1 1 1

モナドランクと仮定 G がである G。

```
   G =: 0 { (g b. 0)
```

そして、(f @ g)は、各 G-セルに別々に適用される (f: g) "G を意味する (f: g)。

(f @ g) y	(f @: g)"G y
+ - + - + | 1 | 5 | + - + - +	+ - + - + | 1 | 5 | + - + - +

したがって、一般的なスキームは次のとおりです。

```
            (f @ g)y は(f: g)を意味する "G y
```

8.6 Composition: Monad Tracking Dyad

合成

次に、ダイアディック g のコンポジション(f @ g)を調べます。仮定 F 及び G はによって定義されます。

```
   f =: <
   g =: |. " 0 1  NB. dyadic
```

ここで、xgy は、x の対応するスカラーによって y 内のベクトルを回転させることを意味します。例えば：

x =: 1 2	y =: 2 3 $ 'abcdef'	x g y
1 2	abc def	bca fde

次に、f @: g と f @ g の違いを示す例を示します

f(xgy)	x(f: g)y	x(fg)y
+ --- + | bca | | fde | + --- +	+ --- + | bca | | fde | + --- +	+ --- + --- + | bca | fde | + --- + --- +

私たちは、(f @: g)動詞 f が一度適用されるのを見ます。(F する@ G) 、各個別のアプリケーションのために、G の対応するアプリケーションが存在する F は。
dyad g の左右の順位が L と R であると仮定する。次に、(fg)は、x からの L セルと y からの対応する R セルの 各対に別々に加えられる(f: g)を意味する 。すなわち、(f @ g)は (f: g) "G を意味し、G = L、R である。

G =: 1 2 { (g b. 0)	G =: 1 2 { (g b. 0)	G =: 1 2 { (g b. 0)
0 1	+ --- + --- + | bca | fde | + --- + --- +	+ --- + --- + | bca | fde | + --- + --- +

スキームは次のとおりです。

```
x (f@g) y =  x (f@:g) " G y
```

8.7 Composition: Dyad Tracking Monad
構成

第 3 章では、結合演算子として&を 見いだしたことを思い出してください。1 つの引数を名詞とし、もう 1 つの引数を 2 項動詞とすると、結果はモナドです。例えば、+&6 は引数に 6 を加えるモナドです。
&の 両方の引数が動詞の場合、&は別の解釈をします。この場合、それは "Compose"と呼ばれる合成演算子です。ここで、ダイアディック f のコンポジション f&g を見てみましょう。
仮定グラムは「スクエア」関数であり、f は二つのリストを結合し、「コンマ」機能です。

```
f =: ,
g =: *:
```

x =: 1 2	y =: 3 4	gx	gy
1 2	3 4	1 4	9 16

ここでは、(f &: g)と(f & g)の違いを示す例を示します。

(g x) f (g y)	x (f &: g) y	x (f & g) y
1 4 9 16	1 4 9 16	1 9 4 16

我々は、(f&: g)において動詞 f が 1 回だけ適用され、1 4、9 16 を与えることを見る。対照的に(F&G)の二つの別々のアプリケーションがある f がまず与え、1,9 及び第二 4,16。

このスキームは、

```
x (f & g) y  means  (g x) (f " G,G) (g y)
```

ここで G は g のモナドランクです。ここで f は、各組合せに別々に適用される G から-cell X と対応する G がから-cell Y。説明する：

G = 0 {(g b.0)	(g x)(f "(G, G))(g y)	x(f & g)y
0	1 9 4 16	1 9 4 16

8.8 Ambivalence Again
アンビバレンス

組成物 F&G が曖昧であることができます。ダイアディックの場合、x f&g y、上で見ました。モナドの場合、f&g y は f @ gy と同じ意味です。

```
f =: <
g =: *:
```

f&g 1 2 3	f @ g 1 2 3
+ - + - + - + | 1 | 4 | 9 | + - + - + - +	+ - + - + - + | 1 | 4 | 9 | + - + - + - +

8.9 Summary
要約

ここまでに見た 8 つのケースの概要を示します。

```
        @:  (f @: g) y = f(g y)
        @: x(f @: g) y = f(x g y)

        &:  (f &: g) y = f(g y)
        &: x(f &: g) y =  (g x) f (g y)

        @    (f @ g)  y =  (f @: g) "G y
        @   x(f @ g)  y = x(f @: g) "LR y

        &    (f&g)    y =(f @: g) "G y
        &   x(f&g)    y =(g x) (f " (G、G)) (g y)
```

ここで、G は g のモナドランクであり、LR は g の左右ランクのベクトルです。

8.10 Inverses
逆

"Square"動詞(* :)は、 "Square-root"動詞(%:)の逆数であると言われています。逆動詞はそれ自身の逆です。

*: 2	%: 4	%4	% 0.25
4	2	0.25	4

J の多くの動詞には逆があります。式 f ^: _1 が動詞 f の逆であるように組み込みの^ :(キャレットコロン、 "Power"と呼ばれる)があります。(これは、従来の表記法で f $^{-1}$ を書くのと同じです。)
たとえば、正方形の逆数は平方根です。

sqrt =: *: ^: _1	sqrt 16
*: ^: _ 1	4

^: 組み込み動詞だけでなく、作曲などのユーザー定義動詞の逆も自動的に見つけることができます。例えば、「2 乗平方根」の逆数は「1/2 の平方根」であり、

```
   foo =: (2&*)@: %:
   fooINV =: foo ^: _1
```

foo 16	fooINV 8	foo fooINV 36
8	16	36

8.11 Composition: Verb Under Verb
構成：動詞の動詞

私たちは現在、結合詞&で構成を見ています 。(アンダースコアと呼ばれるアンパサンド・ドット)。考え方は、 " f Under g "というコンポジションは: g を適用し、次に f を、次に g の逆数を適用するということです。
一例として、対数をとって半分にし、対数をとって対数をとることで、数の平方根を求めることができます。半分は- : とし、対数は^であることを思い出してください。

SQRT =: - : &. ^.	SQRT 16
- : &.^.	4

一般的なスキームは、

```
            (f &. g) y   means  (g ^: _1) f g y
```

☆第 8 章終了。

第 9 章 : Trains of Verbs
動詞の列

この章では、第 3 章で始まった動詞列の話題を続けます。列車は孤立した一連の関数であり、(+ * -)のように順番に書かれていることを思い出してください。

9.1 Review: Monadic Hooks and Forks
Monadic フックとフォーク

第 03 章 からモナドフックを想起しましょう。

```
        (f g) y   means    y f (g y)
```

ここでは、簡単な注意として、例を挙げます。整数は床に等しい：

y =: 2.1 3	y =: 2.1 3	y =: 2.1 3	y =: 2.1 3
2.1 3	2 3	0 1	0 1

スキームと一緒に、モナディックフォークを思い出してください：

(fgh)v は(fv)g(hv)を意味し、

たとえば、数値のリストの平均は、合計を項目数で割ったものです。

```
   sum =: + /
   mean =: sum%: #
```

y =: 1 2 3 4	y =: 1 2 3 4	y =: 1 2 3 4	y =: 1 2 3 4	y =: 1 2 3 4
1 2 3 4	10	4	2.5	2.5

今度はさらにいくつかのバリエーションを見ていきます。

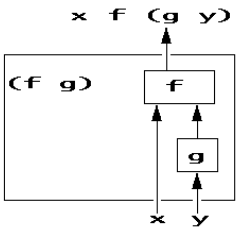
9.2 Dyadic Hooks
ダイアディックフック

3 時間 15 分は 3.25 時間です。(3 時 15 分)が 3.25 であるような動詞 hr はフックとして書くことができます。我々はしたいの x 時間 y があることを、X +(Y の 60%)ので、フックは次のとおりです。

```
   hr =: +(%&60)
   3 hr 15
15.35
```

ダイアディックフックのスキームは次のとおりです。

```
        x (f g) y   means   x f (g y)
```



9.3 Dyadic Forks

ダイアディック・フォーク

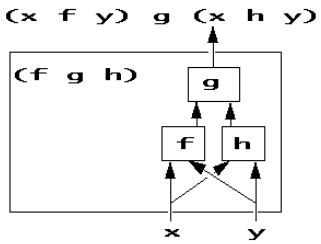
「10 プラスまたはマイナス 2」という表現がリスト 12 8 を意味するとします。フォワード (+、 -)として、x のプラスマイナス y を計算する動詞を書くことができます。

(10+2) , (10-2)	10 (+,-) 2
12 8	12 8

ダイアディック・フォークのスキームは次のとおりです。

```
x (f g h) y   means    (x f y) g (x h y)
```

このスキームの図は次のとおりです。



9.4 Review

レビュー

動詞の列車には 4 つの基本的なスキームがあります。

```
      (fgh)y    = (fy)  g(hy)   monadic fork
     x(fgh)y    = (xfy) g(xhy)  dyadic  fork
      (fg)y     =  y    f(gy)   monadic hook
     x(fg)y     =  x    f(gy)   dyadic  hook
```

9.5 Longer Trains

Train の延長

ここでは、列車として定義できる機能のクラスを広げる方法を見ていきます。一般に、任意の長さの列車をフックとフォークに分析することができる。4 つの動詞の電車について、EFGH、スキームは、ということです

```
         e f g h    means   e (f g h)
```

つまり、4 列車(efgh)はフックで、最初の動詞は e、2 番目の動詞は fork (fgh)です。たとえば、y が数値のリストであるとします。

```
   y =: 2 3 4
```

次に、v の "ノルム"は(v - 平均 v)として定義され、ここで平均は(sum%: #)として上で定義され ます。y のノルムに対する次の式はすべて等しいことがわかります。

```
   y - mean y
_1 0 1

   (- mean) y       NB. as a hook
_1 0 1

   (- (sum % #)) y  NB. by definition of mean
_1 0 1
```

```
   (- sum % #) y    NB. as 4-train
_1 0 1
```

長い電車である程度の芸術的判断が求められています。この 4 つの列車(- 合計%: #)としての最後の定式化は、重要な考え方が平均を引いているかもしれないほど明確に引き出しません。処方(- 平均)はより明確である。
5 つの動詞の列の場合、スキームは次のようになります。

```
         d e f g h   means  d e (f g h)
```

つまり、5 列(defgh)は第 1 動詞 d、第 2 動詞 e、第 3 動詞 fork (fgh)を持つフォークです 。たとえば、month day year：

```
   date =: 28 2 1999
```

月と年を別々に抽出する動詞を定義する：

```
   Da =: 0 & {
   Mo =: 1 & {
   Yr =: 2 & {
```

日付は 5 つの列車によって異なる方法で提示することができます：

(Da , Mo , Yr) date	(Da , Mo , Yr) date
28 2 1999	+ - + - + ---- + | 2 | 28| 1999 | + - + - + ---- +

動詞列(abc ...)の一般的な体系は、動詞の数が偶数か奇数かによって異なります。

```
          even:  (a b c ...)    means   hook (a (b c ...))
          odd :  (a b c ...)    means   fork (a b (c ...))
```

9.6 Identity Functions
アイデンティティ関数( )

組み込みの動詞、モナディック[(左括弧、 "同"と呼ばれる)があります。引数と同じ結果を返します。

[99	['a b c'
99	a b c

二者の場合、また、同様の動詞があります]。全体として、私たちはこれらの計画を持っています：

```
          [ y   means y
        x [ y   means x
          ] y   means y
        x ] y   means y
```

[3	2 [3	] 3	2] 3
3	2	3	3

モナド[とモナディック]は両方とも「同じ」と呼ばれています。二項は、[「左」と呼ばれています。ダイアディック]は「右」です。
式(+ %)はフォークです。引数 x と y に対して、以下を 計算する：

```
(x+y) % (x ] y)
```

あれは、

```
(x+y) % y
```

2] 3	(2 + 3) % (2] 3)	2 (+ %]) 3
3	1.66667	1.66667

アイデンティティ機能の別の用途は、[割り当ての結果が表示させることです。式 FOO = 42 式ながら割り当てである[FOO = 42 ではない：それは、単に割り当てを含んでいます。

```
      foo =: 42       NB.  nothing displayed
      [ foo =: 42
42
```

[動詞] のもう 1 つの用途は、複数の割り当てを 1 行にまとめることです。

a =: 3 [b =: 4 [c =: 5	a =: 3 [b =: 4 [c =: 5
3	3 4 5

[は動詞な ので、その引数は名詞でなければなりません(つまり、関数ではありません)。したがって、[[すべての名詞に評価されなければならない]]と結合された割り当ては、

9.6.1 Example: Hook as Abbreviation
例：略語としてのフック

モナドフック(gh)は、モナディックフォーク([gh])の略語です。実証するために、私たちが持っているもの：

```
   g =
   : 、h =: *:
   y =: 3
```

そして、次の各式は等価です。

```
   ([ g h) y       NB. a fork
3 9
   ([ y) g (h y)   NB. by defn of fork
3 9
   y g (h y)       NB. by defn of [
3 9
   (g h) y         NB. by defn of hook
3 9
```

9.6.2 Example: Left Hook
例：左フック()

モナドフックには一般的なスキームがあることを思い出してください

```
           (f g) y    =    y f (g y)
```

どのようにして列車としてスキームの関数を書くことができますか？

```
(  ?  ) y  =  (f y) g y
```

2 つの可能性があります。1 つはフォーク(f g])です：

```
   f =: *:
   g =: ,

   (f g ]) y        NB. a fork
9 3
   (f y) g (] y)    NB. by meaning of fork
9 3
   (f y) g y        NB. by meaning of ]
9 3
```

別の可能性について、リコール～その仕組みと副詞を：

```
(x f~ y) means   y f x
```

私たちの Train はフック（g～f）として書くことができます。

```
   (g~ f) y      NB. a hook
9 3
   y (g~) (f y)  NB. by meaning of hook
9 3
   (f y) g y     NB. by meaning of ~
9 3
```

9.6.3 Example:Dyad

例：ダイアド

これでは意味があります[と] 左右の引数に立っとみなすことができるが。

```
   f =:  'f'&、
   g =:  'g'&、
```

foo =: (f @: [) , (g @:])	'a' foo 'b'
f@:[, g@:]	fagb

9.7 The Capped Fork

キャップドフォーク()

途切れのない列として記述できる関数のクラスは、 "Cap"動詞[: (leftbracket colon)]の助けを借りて拡大することができます。
体系は次のとおりです。動詞 f と g の場合、fork:

```
[: f g     means   f @: g
```

たとえば、上記の f と g を使用すると、

y=:'y'	f g y	(f @: g) y	([: f g) y
y	fgy	fgy	fgy

3 つの動詞([: fg)のシーケンスが フォークのように見えることに注目してください。この「キャップ付きフォーク」では、適用される中間動詞 f の MONADIC のケースです。

[: 動詞は ONLY フォークの左側の動詞として有効です。動詞としては空のドメインを持ちます。つまり、どの引数にも適用できません。その有用性は、長い電車の建設にある。たとえば、次のようにします。

```
   h =: 'h'&
```

式(f , [: g h)は動詞を表す 5-Train である：

```
   (f , [: g h) y        NB. a 5-train
fyghy

   (f y) , (([: g h) y)  NB. by meaning of 5-train
fyghy

   (f y) , (g @: h y)    NB. by meaning of [:
fyghy

   (f y) , (g h y)       NB. by meaning of @:
fyghy

   'fy'  , 'ghy'         NB. by meaning of f g h
fyghy
```

9.8 Constant Functions
定数関数

ここでは、動詞の列として書くことができる関数のクラスを広げる方法を見ていきます。引数にかかわらずゼロの値を返すビルトインの動詞 0: (ゼロコロン)があります。モナドとダイアディックのケースがあります：

0: 99	0: 2 3 4	0: 'hello'	88 0: 99
0	0	0	0

ならびに 0: 同様の機能がある: 1 : 2 : 3 等までの 9: また、負の値: _9: へ_1 は:

1: 2 3 4	_3: 'hello'
1	_3

0: 結果が一定であるため、定数関数と呼ばれます。定数関数は、定数を必要とするが動詞を記述しなければならない場所(なぜなら、動詞は自然に動詞のみを含むため)の列車で発生する可能性があるので便利です。
たとえば、その引数が負(0 未満)かどうかをテストする動詞は、 (<&0)として書くことができますが、代わりにフックとして書くこともできます。

```
   negative =: < 0:
```

x =: _1 0 2	0: x	x < (0: x)	negative x
_1 0 2	0	1 0 0	1 0 0

9.9 Constant Functions with the Rank Conjunction

ランク結合による定数関数

定数関数： 9 へ 9: 列車を定義する方法をより多くの選択肢を提供します。これらは 1 桁のスカラー定数に制限されています。定数関数を書くより一般的な方法を見ていきます。k を問題の定数 とすると、

```
   k =: 'hello'
```

(3: 'k')と 書かれた明示的な動詞は、k の一定の結果を与えるでしょう：

k	(3 : 'k') 1	(3 : 'k') 1 2
hello	hello	hello

動詞(3: 'k')は明示的なので、そのランクは無限です。(私たちが見たように、その後のスカラーに別々にそれを適用するには章 07)我々は、ランク指定する必要が R の 0 をランク組み合わせの助けを借りて、":

k	R =: 0	((3 : 'k') " R) 1 2
hello	0	hello hello

式((3: 'k') ' R)は、「左の引数として、上のような動詞または名詞をとることができるので、(k "R")と略記することができる。

k	R	((3 : 'k') " R) 1 2	('hello' " R) 1 2
hello	0	hello hello	hello hello

場合ことに注意 k は名詞であり、次いで、動詞(K「R)を意味する：定数値 K 。引数の各ランク R のセルのために生成する一方、場合 、V は動詞であり、次いで、動詞(V」R)引数の各ランク-R セルに適用される動詞 v を意味します。

定数関数の一般的なスキームは" is:

```
                k "は R(3: 'k')" R
```

9.9.1 A Special Case

特別なケース

華氏で表される温度が与えられると、摂氏での等価物は 32 を引いて 5 から 9 を掛けて計算されます。

```
   Celsius =: ((5%9) & *) @: (- &32)
   Celsius 32 212
0 100
```

摂氏 を定義する別の方法は、3 つの動詞の列であるフォークのようなものです。

```
   Celsius =: (5%9 "_ ) * (-&32)
   Celsius 32 212
0 100
```

上記の Celsius のフォークは、左の動詞が定数関数として存在することに注意してください。ここでは、フォークの特殊なケースがあります(これは、名詞動詞動詞の形で省略することができます)。

```
   Celsius =: (5%9) * (-&32)
   Celsius 32 212
0 100
```

フォークの略語の一般的なスキーム(J6 で新)は、n が名詞、u と v が動詞、

```
          n u v  means the fork  (n"_) u v
```

☆第 9 章終了。

第 10 章 : Conditional and Other Forms
条件付きフォームと他のフォーム

Tacit 動詞、つまり引数変数を使用せずに定義された動詞は、第 03 章で紹介されました。暗黙の定義のこのテーマを続けて、第 08 章では作曲者の使い方と第 10 章の動詞の列車を見ました。
この章の計画は、動詞を暗黙のうちに定義するさらなる方法を検討することです。
条件付きフォーム
再帰形式
反復形式
明示的な定義から暗黙の定義を生成する

10.1 Conditional Forms
条件付きフォーム

数字(ある正の整数)を考えてください。それが奇数の場合は、3 を掛けて 1 を加算します。それ以外の場合は、考えた数を半分にします。このプロシージャは、1 から新しい番号 4 を、そして 4 から新しい番号 2 を計算します。
この手順を反復することによって生成された一連の数字は、Collatz シーケンス、または Hailstone シーケンスと呼ばれることがあります。例えば：

```
        17 52 26 13 40
```

この手順のための関数を書くために、私たちは 3 つの動詞で始まり、たとえば半減半減し、MULT は乗算および-1 を追加し、そして奇数は奇数をテストします：

```
   halve =: -:
   mult  =: 1: + (* 3:)
   odd   =: 2 & |
```

halve 6	mult 6	odd 6
3	19	0

新しい数字の手続きは明示的な動詞として書くことができます：

```
       COLLATZ =: 3 : 'if. odd y do. mult y else.  halve y end. '
```

それと同等に暗黙の動詞として：

```
  collatz =: halve ` mult @. odd
```

COLLATZ 17	collatz 17
52	52

collatz の定義では、シンボル`(backquote)は" Tie "結合と呼ばれます。それは 2 つの動詞のリストを作るために半分と多分を結びつける。(そのようなリストは "gerund"と呼ばれ、第 14 章で gerunds のより多くの使い方を見ます)。 接続@。「アジェンダ」と呼ばれています。その右の引数は、左の引数である動詞のリストから別の動詞を選択する動詞です。したがって、collatz y を評価する際には 、奇数 y の値を用いてリストを索引付けする(半

分にする)。次に、選択された動詞が y に適用されます。すなわち、奇数 y が 0 または 1 であるので、y の半分または それに応じて多 y が計算される。
この例では、引数が奇数であるかどうかを考慮する 2 つのケースがあります。一般に、いくつかの場合があります。一般的なスキームは、u0、u1、... un が動詞で、t が 0 ... n の整数を計算する動詞 である場合、動詞：

```
            foo =: u0 `u1` ... `un @。t
```

明示的な動詞でモデル化することができます。

```
   FOO =: 3 : 0
if.     (t y) = 0  do. u0 y
elseif. (t y) = 1  do. u1 y
      ...
elseif. (t y) = n  do. un y
end.
```

)すなわち、動詞 t は引数 y をテストし、次に(ty)が 0 か 1 かどうかに応じて u0 または u1 または...が y に適用されます。

10.1.1 Example with 3 Cases
3 つのケースを持つ例

毎月、銀行が顧客の口座の残高に応じて金利を支払う、または請求するとする。3 つのケースがあります。
残高が 100 ドル以上の場合、当行は 0.5%の利息を支払う
残高がマイナスの場合、銀行は利息を 2%で請求します。
それ以外の場合は、残高は変更されません。
3 つの動詞は、3 つのそれぞれの場合に 1 つです：

```
pi =: * & 1.005 NB.  pay interest
   ci =: * & 1.02       NB.  charge interest
   uc =: * & 1          NB.  unchanged
```

π1000	ci _100	uc 50
1005	_102	50

今度は、与えられたバランスから 0 または 1 または 2 を動詞にしたいと考えています。私たちはケースの番号付け方法を自由に選ぶことができます。次の動詞は、バランスが 0 ドル以上の場合は 1、100 ドル以上の場合は 1 となります。

```
   case =: (>: &0)+(>: &100)
   case _50 0 1 100 200
0 1 1 2 2
```

今度は、バランスの処理を動詞 PB で表すことができます .3 つの動詞を正しい大文字と小文字の順序で書き込むように注意してください。

```
   PB =: ci'uc'pi @。場合
```

PB_50	PB 0	PB 1	PB 100	PB 200
_51	0	1	100.5	201

残高(PB の議論)は、3 つの可能なケースのうちのちょうど 1 つに該当すると予想される。引数が天びんのリストであるとします。大文字小文字は、大文字小文字だけでなく大文字小文字のリストを提供します。これはエラーです。救済策は、引数の各項目に PB 関数を個別に適用することです。

PB 99 100	(PB "0)99 100
エラー	99 100.5

10.2 Recursion
再帰

数字のリストの合計を計算するために、動詞+/を見てきましたが、合計動詞を定義する別の方法を見てみましょう。
空の数値リストの合計はゼロです。それ以外の場合は、最初の項目+残りの項目の合計です。3 つの動詞を定義すると、空のリストをテストし、最初の項目を取り出し残りの項目を取ります：

```
   empty =: # = 0:
   first =: {.
   rest  =: }.
```

考慮すべき 2 つのケースは次のとおりです。
空のリストです。この場合、0 を返す関数を適用します。ゼロを返す
空でないリストを返します。この場合、最初のものと残りのものの合計が必要です。

```
   Sum =: (first + Sum @ rest) `0: @。空の
   合計 1 1 2 4
4
```

ここでは、動詞 "Sum"はそれ自身の定義で繰り返されるので、定義は再帰的であると言われています。 このような再帰的な定義では、再帰する名前は$: (ドルコロン、 "自己参照"と呼ばれます)と書くことができます。これは "this function"を意味します。これにより、名前を代入することなく、式として再帰関数を書くことができます。以下は、式としての "Sum"関数です：

```
   ((first + $: @ rest) `0: @。empty)1 2 3
6
```

10.2.1 Ackermann's Function
アッカーマンの機能

Ackermann の機能は非常に再帰的であることで賞賛されています。教科書は、ダイアドの明示的な定義のような形でそれを示しています：

```
   Ack =: 4 : 0
if.       x = 0  do.  y + 1
elseif.   y = 0  do.  (x - 1) Ack 1
elseif.   1      do.  (x - 1) Ack (x Ack y -1)
```

```
end.
)

   2 Ack 3
9
```

暗黙のバージョンは、Roger Hui(Vector、Vol 9 No 2、October 1992、page 142)によるものです。

```
   ack =: c1 ` c1 ` c2 ` c3 @. (#. @(,&*))
   c1 =: >:@]                    NB. 1 + y
   c2 =: <:@[ ack 1:             NB. (x-1) ack 1
   c3 =: <:@[ ack [ack <:@]      NB. (x -1) ack x ack y -1
   2 ack 3
9
```

c2 を 定義する行では、$: ではなく ack と呼ばれることに注意してください。ここで $: は c2 を意味するからです。
ここにもう一つのバージョンがあります。暗黙のバージョンは、x と y を動詞[と]として最初に定義することで、もう少し従来のものに見えるようにすることができます。また、1 つの行に対して 1 つのケースのみをテストします。

```
   x =: [
   y =: ]
   ACK =: A1 `  (y + 1:)                    @. (x = 0:)
   A1  =: A2 ` ((x - 1:) ACK 1:)             @. (y = 0:)
   A2  =:      (x - 1:) ACK (x ACK y - 1:)
   2 ACK 3
9
```

10.3 Iteration
反復

10.3.1 The Power Conjunction
パワーコンジャンクション

数を考え、それを倍にして、結果を倍にして、もう一度二倍にします。3 回の倍加の結果は元の 8 倍になります。組込み動詞+: は "double"で、動詞 "three doublings"は+: ^: 3 として "Power"結合詞(^:)を使用して記述できます。

+: +: +: 1	(+: ^: 3)1
8	8

一般的なスキームは、動詞 f と整数 n

```
       (f ^: n) y  means  f f f ... f f f f  y
                          <--- nf's --->
```

f ^: 0 v はちょうど v であり、次に f ^: 1 v は fv であることに注意してください。たとえば、collatz 動詞 "奇数の場合は、3 の 1/2 の乗算または 3 倍の加算"を思い出してください。

(collatz ^: 0)6	(collatz ^: 1)6	collatz 6
6	3	3

Power を使用すると、collatz を 0 回、1 回、2 回など、6 から始めるなど 、シリーズを生成できます

```
   (collatz ^: 0 1 2 3 4 5 6)6
6 3 10 5 16 8 4
```

10.3.2 Iterating Until No Change

変更がないまで繰り返す

Power 結合に無限大( )の右引数が与えられた 式 f ^: は、結果が前の結果と同じになるまで f が適用される動詞です。スキームは次のとおりです。

```
       f ^: _ y    means
                   r  such that r = f f ... f f y
                          and r = f r
```

ここに例があります。関数 P が次のように定義されているとします。

```
      P =: 3 : '2.8  * y * (1          y)'
```

次に、0.5 の近傍の引数に関数を繰り返し適用すると、20 回程度の反復の後に結果は約 0.643 の値に決まります。

```
   (P ^: 0 1 2 3 19 20 _)0.5
0.5 0.7 0.588 0.6783 0.6439 0.642 0.6429
```

r = P r であるので 、この値 r は、P の固定点と呼ばれる

r =: (P ^: _) 0.5	P r
0.6429	0.6429

10.3.3 Iterating While

While の繰り返し

"Power"連結詞の右の引数は、実行される反復回数を計算する動詞です。スキームは次のとおりです。

```
          (f ^: g) y  means  f ^: (g y) y
```

gv が 0 または 1 を計算する 場合、f は 0 回または 1 回適用され ます。たとえば、ここでは偶数を半分にして奇数だけを残す動詞があります。

```
   halve =: -:
   even  =: 0: = 2 & |
```

foo =: halve ^: even	(foo " 0) 1 2 3 4
halve^:even	1 1 3 2

ここで、関数

```
   w =: (halve ^: even) ^: _
```

これは、「半分にしても結果が変わらない限り、これを続けてください」という意味です。

```
   w(3 * 16)
3
```

スキームは、g が 0 または 1 を返すと、書かれた関数(f ^: g ^: _)は明示的な定義によってモデル化できるということです。

```
   model =: 3 : 0
while. (g y)
   do. y =.  f y
end.
y
)

   f =: halve
   g =: even
```

(f ^: g ^: _) 3 * 16	model 3*16
3	3

10.3.4 Iterating A Dyadic Verb
二項動詞の反復

0 に 3 を 2 回 加算すると 6 が得られます

```
   ((3&+) ^: 2) 0
6
```

この表現は、以下のように省略することができます。

```
   3 (+ ^: 2) 0
6
```

与えられた左の引数(3)は最初に固定されているので、反復動詞はモナド 3&+です。一般的なスキームは次のとおりです。

```
        x (u ^: w) y  means   ((x&u) ^: w) y
```

w は名詞や動詞です。

10.4 Generating Tacit Verbs from Explicit
暗黙の動詞を明示的に生成する

e は次のように明示的に定義された動詞です。

```
   e =: 3 : '(+/ y) % # y'
```

コロン接続詞の右引数は、"body"と呼ぶことができます。次に、同じ body で 3：の代わりに 13：を書くことによって、暗黙の動詞、つまり e と同等の動詞を生成することができます。

```
   t =: 13 : '(+/ y) % # y'
```

e	t	e 1 2 3	t 1 2 3

`3 : '(+/ y) % # y'`	`+/ % #`	`2`	`2`

ここで明示的なダイアグラムの例を示します。

```
   ed =:  4 : 'y % x'
```

同等の暗黙のダイアドは、同じボディで `4:` ではなく `13:` を書くことで生成できます。

```
   td =: 13 : 'y % x'
```

`ed`	`td`	`2 ed 6`	`2 td 6`
`4 : 'y % x'`	`%~`	`3`	`3`

私たちが `13: body` と書くと、`body` は `y`(ただし `x` は含まない)を含んでいると結論づけることができます。結果は、モナドの場合が `3: body` に等しい暗黙の動詞になります。一方、`body` に `x` と `y` の両方が含まれている場合、その結果は、`dyadic` の場合が `4: body` に等しい暗黙の動詞になります。

暗黙の関数を生成する目的で、本体は単一の文字列または 1 行に制限されています。`3: body` では、定義が入力されたときに本文が評価されないことを思い出してください。しかし、`13: body` では、実際には body が評価されます。例えば：

`k =: 99`	`p =: 3 : 'y+k'`	`q =: 13 : 'y+k'`	`p 6`	`q 6`
`99`	`3 : 'y+k'`	`99 + ]`	`105`	`105`

`p` は `k` で定義され、`q` は定義されないことが わかります。しながら、`P` 及び `Q` は存在等価である、の値の任意の後続の変更 `k` は もはや同等それらをレンダリングしないであろう。

`k = 0`	`p 6`	`q 6`
`0`	`6`	`105`

値が割り当てられていない名前は動詞を表すものとみなされます。次の例では、`f` は未割り当て、 `C` は事前定義された連結詞、`g` は事前定義された動詞です。

```
   C =: @:
   g =: %:
```

`foo =: 13 : '(f C f y) , g y'`	`f =: *:`	`foo 4`
`f@:f , g`	`*:`	`256 2`

☆第 10 章終了。

第 11 章：Tacit Verbs Concluded
暗黙の動詞の結論

この章では、暗黙の動詞の表現を書く際のいくつかの一般的な点について考察する。
暗黙の動詞の例を以下に示します。引数に 3 を掛けます：

f =: *&3	f 4
*&3	12

第 3 章から、結合演算子がダイアダの引数の 1 つを修正することによってダイアドからモナドを生成することを思い出してください。 このスキームは、N が名詞、V が二項動詞の場合：

(N & V) y	means	N V y
(V & N) y	means	y V N

我々は、結合演算子を典型的な演算子の例として取り上げます。ここでは、引数は名詞または動詞です。一般に、N は名詞を表す式であり、V は動詞を表す式です。これらの式がどのように評価されるかを見ていきます。 一般規則は付録 1 に正式に定められているが、ここでは主な点のいくつかを非公式に取り上げる。

11.1 If In Doubt, Parenthesize
疑わしい場合は括弧で囲む

ここに別の暗黙の動詞があります。一般的な形式は V&N です。その引数を 5%4 倍(5/4 倍)、すなわち 1.25 倍します。

scale =: * & (5 % 4)	scale 8
*&1.25	10

かっこは約 5%4 ここで必要ですか？それらを省略すると、次のようになります。

SCALE =: * & 5 % 4
SCALE
1.25

彼らは明らかに違いを生み出します。SCALE は動詞ではなく数字です。1.25 の結果は、引数%4（4 の逆数）に動詞*＆5 を適用することによって生成されます。

% 4	(* & 5) (% 4)	* & 5 % 4
0.25	1.25	1.25

私たちは一般的な規則を持っています：非公式には、接辞が隣接する動詞の前に適用されると言うことができます。したがって、式*&5%4 において、最初のステップは、その引数*と 5 に&演算子を適用することです。
なぜ 5%でなく 5%4 という正しい議論があるのですか？ 別の一般的な規則のために：結合詞の右の引数はできるだけ短いです。 連合体には「短い権利範囲」があると言います。 対照的に、私たちは、動詞は、以前は「最右端」の動詞の規則を表現するための「長い正しい範囲」を持っていると言います。

演算子の左引数はどうですか？副詞または連合体は、「長い左スコープ」、すなわちできるだけ多くの語を持つと言われています。たとえば、引数の 2 乗に 3 を加える動詞 z がありま す。 3 プラス 2 の 2 乗は 7 です。

z =: 3 & + @: *:	z 2
3&+@:*:	7

@: の左の引数は 3&+の全体であることがわかります。
特定のケースで疑問がある場合は、常に意思を明確にすることができます。式の一部、すなわち関数 - 動詞または演算子の周りに、目的の引数とともにかっこを書くことができます。例えば、動詞 z は次のようにかっこで書くことができます。

z =: (3&+)@: *:	z 2
3&+ @: *:	7

時にはカッコが必要で、時々そうではありませんが、疑問があればカッコを強調してみましょう。

11.2 Names of Nouns Are Evaluated
名詞の名前が評価される

一般的な形式の式で N&V または V&N に生じる任意の名詞の名前 N はすぐに評価されます。ここでは、5 分の 1 を乗算する関数 f の例を示します。数値は次のように与えられる。%B の A 及び B は名詞です。

a =: 5	b = 4	f =: *&(a%b)	f 8
5	4	*&1.25	10

関数 f には計算された数値 1.25 が含まれており、%b が評価されていることが わかります。

11.3 Names of Verb Are Not Evaluated
動詞の名前が評価されない

N&V で動詞式の V は必ずしも完全には評価されません。式 V が動詞の名前であれば、その名前で十分です。

w =: *	g =: w&(a%b)	g 8
*	w&1.25	10

11.4 Unknowns are Verbs
未知数は動詞です

新しい名前に遭遇した場合、可能な場合にはまだ定義されていない動詞とみなされます。

h =: ytbd&(a%b)	ytbd =: *	h 8

ytbd&1.25	*	10

今まで知られていなかった名前のシーケンスは、動詞の列とみなされます。

```
   Ralph Waldo Emerson
Ralph Waldo Emerson
```

結果として、動詞は「トップダウン」方式で、すなわち後で詳細を提示して定義することができる。たとえば、摂氏から華氏へのコンバータがトップダウンで表示されます。

```
ctof =: shift @ scale
          shift =: +&32
          scale =: *&(9%5)
```

ctof	ctof 0 100
shift@scale	32 212

ctof は(名前の)スケールとシフトだけで定義され ていることがわかります。したがって、 スケールやシフト を変更すると、ctof の定義を効果的に変更します。

```
   ctof 100
212
   scale =: *&2
   ctof 100
232
   scale =: *&(9%5)
   ctof 100
212
```

従属関数の 1 つを変更するだけで関数の定義を変更する可能性は、望ましいと見なされる場合もありません。これは、小さな部分を変更するだけで定義を修正できる限り、便利です。しかし、それは誤りの源になることがあります。我々は新しい動詞を導入することが、規模はそれを忘れ、言うスケールは既に従属として定義されて CTOF。
ctof をその下位機能の偶発的な再定義 から保護する方法があります。まず、明示的に定義されたラッパーを配置し、スケールとシフトをローカルに作成することができます。

```
   CTOF =: 3 : 0
shift =. + & 32
scale =. * & (9 % 5)
shift @ scale y
)
   CTOF 100
212
```

第 2 の方法は、言い換えると、「修正」副詞を用いて、ctof の定義を「フリーズ」または「固定」することである。(文字-f ドット)。ctof と(ctof f.)の動詞の値の違いを観察してください。

ctof	ctof f.
shift@scale	+&32 @(*&1.8)

私たちはその副詞を見ます。暗黙の動詞に適用すると、定義によって名前が置き換えられ、組み込み関数に関してのみ定義される同等の動詞が与えられます。ctof のもう一つの定義があります。

```
scale =: *&(9%5)
   shift =: +&32
   ctof =: (shift @ scale)f。
```

ctof	ctof 0 100
+&32 @(*&1.8)	32 212

この定義の後、scale と shift の名前はまだ定義されていますが、もはや ctof の定義では重要ではありません。

11.5 Parametric Functions
パラメトリック関数

次の例は、名詞の評価結果と、暗黙の動詞の式にない動詞を示しています。
曲線は、例えば、次のような式によって特定することができる。

```
                y  =  lambda * x * (1   x)
```

この方程式は、類似の放物曲線のファミリーを記述し、数の異なる値を選択することによってファミリーの異なるメンバーを選び出します。
この式に対応する関数は、動詞 P として明示的に書くことができます。

```
   P = 3: λ* y *(1                    y) '
```

ここで、lambda は関数 P の引数ではなく、結果に違いをもたらす変数である数値です。私たちは、と言うラムダはパラメータであり、またはその関数 P はパラメトリックです。

x = 0.6	ラムダ=: 3.0	P x	ラムダ= 3.5	P x
0.6	3	0.72	3.5	0.84

さて、ラムダをパラメータとして暗黙のバージョンの P を書くことはできますか？
ラムダは現在 3.5 です。 私たちが暗黙の形の P を生成するならば、

```
   tP = 13:  'ラムダ* y *(1-y)'
   tP
3.5 *] * 1  - ]
```

ラムダはパラメータではなく定数として扱われることがわかります。これは私たちが望むものではありません。今回は、ラムダがあらかじめ指定されていないことを確認して、再度消去します。

```
   erase <'lambda'
1
   tP =: 13 : 'lambda * y * (1-y)'
   tP
[: lambda [: * ] * 1 - ]
```

ここで、tP は動詞の列であり、ラムダ(未知)は動詞とみなされることがわかります。この仮定は、数としてのラムダの意図された意味と矛盾する。したがって、ラムダを数値とすると、エラーが発生します。

lambda=: 3.5	tP x
3.5	error

事前にラムダが指定されている かどうかにかかわらず、完全に暗黙の P と等価なものは不可能であるように見える。しかし、我々は近づくことができます。
1 つの可能性は、「完全に暗黙のうちに」妥協することです。ここで、tP は動詞列であり、最初の引数はその引数に関係なく lambda の値を出力するように明示的に定義されています。

tP =: (3 : 'lambda') *] * (1: -])	tP x
3 : 'lambda' *] * 1: -]	0.84

別の可能性は、「正確に同等」で妥協することです。ここではパラメータ lambda を数値ではなく、定数を提供する関数(第 09 章を参照)をとります。
たとえば、パラメータの値は次のように書くことができます。

```
   lambda =: 3.5 " 0
```

そして、TP はように定義することができます。

tP =: lambda *] * (1: -])	tP x
lambda *] * 1: -]	0.84

これで、関数を再定義することなくパラメータを変更できます：

lambda =: 3.75 " 0	tP x
3.75"0	0.9

☆第 11 章終了。

第 12 章 : Explicit Verbs
明示的な動詞

この章は、動詞の明示的定義のテーマである第 04 章から続く。

12.1 The Explicit Definition Conjunction
明示的定義結合

第 4 章から明示的な二項動詞の例を思い出してください.2 つの数字の「正の差」は、より大きいマイナスを小さなものとして定義します。

```
   PosDiff =: 4 : '(x >. y) - (x <. y)'
   3 PosDiff 4
1
```

関数を明示的に定義する一般的なスキームは、明示的定義結合(:,colon)に 2 つの引数をフォームされる。

```
          type : body
```

本文では、変数 x と y が引数です。

12.1.1 Type
タイプ

タイプは数字です：タイプ 3 の関数はモナド動詞または相反する動詞です。タイプ 4 の関数は厳密に二項動詞です(つまり、モナドの場合はありません)。他のタイプもあります：タイプ 1 とタイプ 2 は演算子です(第 13 章を参照)。タイプ 13 については、第 10 章で説明します。

12.1.2 Mnemonics for Types
型のニーモニック

標準の J プロファイルはいくつかの変数をあらかじめ定義して、型などのニーモニック名を提供します。

noun	=: 0	NB.名詞
adverb	=: 1	NB.副詞
conjunction	=: 2	NB.結合
verb	=: 3	NB.動詞
monad	=: 3	NB.Monad
dyad	=: 4	NB.Dyad
def	=: :	NB.Def
define	=: : 0	NB.定義

したがって、上記の PosDiff の例は次のように書くこともできます：

```
   PosDiff =: dyad def '(x>.y ) - (x <. y)'
   3 PosDiff 4
1
```

12.1.3 Body Styles
ボディスタイル

明示的な定義の本文は、1 行以上のテキストから構成されます。本文を提供するにはいくつかの方法があります。上の例の PosDiff は、文字列として書かれた 1 行を表示します。

: colon 演算子には 、0 の右引数を指定して複数行の本体を導入することができます。

```
   PosDiff =: 4 : 0
larger  =. x >. y
smaller =. x <. y
larger - smaller
)

   3 PosDiff 4
1
```

別の変形では、改行体を埋め込むことによって複数行体をコンパクトに書くことができる。LF はラインフィード文字としてあらかじめ定義されています。全身は括弧でくくらなければならないことに注意してください。

```
   PosDiff =: 4 : ('la =. x >. y', LF, 'sm =. x <. y', LF, 'la - sm')
```

PosDiff	3 PosDiff 4
+-+-+-----------+ |4|:|la =. x >. y| | | |sm =. x <. y| | | |la - sm | +-+-+-----------+	1

もう 1 つのバリエーションでは、ボクシングされた行のリストを使用します(これも括弧で囲んで)。

```
   PosDiff = 4: ( 'la = .x> .y'; 'sm = .x <.y'; 'la-sm')
```

PosDiff	3 PosDiff 4
+-+-+-----------+ |4|:|la =. x >. y| | | |sm =. x <. y| | | |la - sm | +-+-+-----------+	1

これらは構文のバリエーションではなく、 : 演算子の右引数として受け入れるデータ構造を構築するための代わりの式であることに注意してください。

12.1.4 Ambivalent Verbs
二価動詞

相反する動詞は、モナドとダイアディックの両方の場合があります。この定義では、最初にモナドの場合が表示され、続いてソロコロンと二項の場合の行が表示されます。例えば：

```
   log =: 3 : 0
^. y    NB. monad - natural logarithm
:
x ^. y   NB. dyad  - base-x  logarithm
)
```

ログ 2.7182818	10 log 100
1	2

12.2 Assignments
割り当て

このセクションでは、明示的な関数を定義する際に重要な代入を考慮します。

12.2.1 ローカル変数とグローバル変数(Local and Global Variables)

例を考えてみましょう

```
   foo =: 3: 0
L =。y
G =: y
L
)
```

ここでは、フォームの割り当て

```
              L =. expression
```

expression の値を L というローカル変数に代入します。L がローカルであると言うと、関数 foo が実行されている間だけ L が存在し、さらにこの L は L という名前の他の変数とは区別されます。対照的に、フォームの割り当て

```
              G =: expression
```

式 の値を G という名前のグローバル変数に代入します。言っ G がユニークな変数という世界的な手段であり、G は独自の権利で、独立して存在しています。
例示するために、我々は、と呼ばれる 2 つのグローバル変数定義 L 及び G は、実行 FOO を ことを示すために L で述べた foo が グローバルと同じではありません L をしながら、G は FOO で述べたグローバル G と同じです。

```
   L =: 'old L'
   G =: 'old G'
```

foo	foo 'new'	L	G
+-+-+-------+ |3|:|L =. y| | | |G =: y| | | |L | +-+-+-------+	new	old L	new

J6 以降の J のバージョンでは、既存のローカル変数と同じ名前の変数にグローバル割り当て(=: 付き)を行うことはエラーとみなされます。
たとえば、引数の変数の x と y は局所的であるので、通常、という名前の変数にグローバル割り当てを行うための明示的な動詞でエラーとなり、Y。

```
   foo =: 3 : 0
z =. y + 1
y =: 'hello'
z
)
```

```
   foo 6
|domain error: foo
|   y    =:'hello'
```

明示的な定義の中から v という名前のグローバルに代入したいのであれば、ローカル y を最初に消去する必要があります。

```
   foo =: 3: 0
z =。y + 1
erase <'y'
y =:  'hello'
z
)
   foo 6
7
   y
hello
```

12.2.2 Local Functions
ローカル関数

私たちは名詞であるローカル変数を見てきました。我々はまた、地域の機能を持っているかもしれない。ローカル関数は、次の例のように、暗黙でも明示的でもよい

```
   foo =: 3 : 0
Square  =. *:
Cube    =. 3 : 'y * y * y'
(Square y) + (Cube y)
)
   foo 2
12
```

しかし、私たちが持つことができないことは、内部の複数行の本文で定義された明示的な局所関数です。複数行の本文は、ソロの右かっこで終わるスクリプトなので、そのような本体を別の内部に入れることはできません。代わりに、次の例のスケールのような内部関数の本体の代わりの形式を使用できます。

```
   FTOC =: 3 : 0
   line1   =. 'k =. 5 % 9'
   line2   =. 'k * y'
scale =. 3 : (line1 ; line2)
scale y - 32
)

   FTOC 212
100
```

内部関数のトピックに関する最後の 1 つのポイント。変数または関数の名前は、グローバルまたはローカルのいずれかです。ローカルの場合は、それが定義されている関数で認識されます。しかし、それは内部関数では認識されません。例えば：

```
   K =: 'hello '

   zip =: 3 : 0
```

```
K =. 'goodbye '
zap =. 3 : 'K , y'
zap y
)

   zip 'George'
hello George
```

グローバルな K とローカルの K があることがわかります。内部関数 zap は、グローバルな K を使用します。これは、zip に対してローカルである K が zap に対してローカルではないためです。

```
K =: 'hello '

   zip =: 3 : 0
K =. 'goodbye '
zap =. 3 : 'K , y'
zap y
)

   zip 'George'
hello George
```

グローバルな K とローカル K があることが分かります。内部関数 zap はグローバルな K を使用します。これは、zip に対してローカルな K が zap に対してローカルではないからです。

12.2.3 Multiple and Indirect Assignments
複数および間接割り当て

J は、異なる
項目に異なる名前を割り当ててリストを展開する便利な手段を提供します。

'day mo yr' =: 16 10 95	day	mo	yr
16 10 95	16	10	95

代入の左に単純な名前の代わりに、スペースで区切られた名前の文字列があります。
バリエーションはボックス化された一連の名前を使用します。

('day';'mo';'yr') =: 17 11 96	day	mo	yr
17 11 96	17	11	96

割り当ての左手の括弧は、「間接的割り当て」と呼ばれるものを与えるために、一連の名前として評価されます。説明する：

```
   N =:  'DAY'; 'MO'; 'YR'
```

(N)=: 18 12 97	DAY	MO	YR
18 12 97	18	12	97

便宜上、複数の割り当ては、自動的にボクシングの 1 つのレイヤーを右側から削除します。

(N) =: 19;'Jan';98	DAY	MO	YR
+--+---+--+ |19|Jan|98| +--+---+--+	19	Jan	98

12.2.4 Unpacking the Arguments
引数のアンパック

すべての J 関数は、正確に 1 つまたは正確に 2 つの引数をとります - ゼロではなく、2 つ以下の引数。これは制限のように見えるかもしれませんが、実際はそうではありません。値のコレクションは、J 関数への複数の引数を実際に形成するために、リストまたはボックスリストにパッケージ化することができます。ただし、J 関数は値を再度解凍する必要があります。これを行う便利な方法は、複数の割り当てを行うことです。例えば、次の根を見つけるためになじみの式 (a*x^2) +(b*x)+c は、係数のベクトル所与の、B、C があるかもしれません。

```
rq =: 3 : 0
'a b c' =. y
((-b) (+,-) %: (b^2)-4*a*c) % (2*a)
)
```

rq 1 1_6	rq 1; 1; _6
2_3	2_3

12.3 Control Structures
制御構造

12.3.1 Review
レビュー

第 4 章 から、次のように定義される正の差関数を思い出してください。

```
   POSDIFF =: 4 : 0
if.   x > y
do.   x - y
else. y - x
end.
)

   3 POSDIFF 4
1
```

if. to end.は「制御構造」と呼ばれる。それで、if. do. else. and end.は「制御語」と呼ばれる。
このセクションの計画では、この例を使用して制御構造の一般的な説明を行い、次にいくつかの特定の制御構造を見ていきます。

12.3.2 Layout
レイアウト

制御構造を構成する式や制御語のレイアウトを自由に選択できます。任意の制御語の直前または直後に、任意の行末は任意であるため、削除するか挿入するかを選択できます。例えば、からできるだけ多くを除去することにより、POSDIFF 我々が得ます。

```
   PD =: 4 : 'if.  x > y  do.  x - y  else.  y - x  end. '
   3 PD 4
1
```

12.3.3 Expressions versus Control Structures
式と制御構造

私たちは表現を評価することを話します。代入は値を生成するので表現と見なしますが、この場合は代入を「実行する」と言っても自然なことです。「実行する」と「評価する」という言葉は、多かれ少なかれ同じ意味で使用されます。
制御構造を実行(または評価)すると、その中の式の 1 つの値である値が生成されます。それにもかかわらず、制御構造は式ではなく、式の一部を形成することはできません。次は構文エラーです。

```
   foo =: 3 : '1 + if. y > 0 do. y else. 0 end.'
   foo 6
|syntax error: foo
|       1+
```

式と制御構造の区別を見ると、明示的定義の本体は項目のシーケンスであり、項目は式または制御構造のいずれかです。ここでは、本体が式であり、その後に制御構造が続き、式が続く例を示します。

```
   PD1 =: 4 : 0
w =. x - y
if. x > y do. z =. w  else. z =. - w end.
z
)

   3 PD1 4
1
```

制御構造体によって生成された値は、制御構造体がシーケンス内の最後の項目でない場合は破棄されます。ただし、この値はアイテムが最後である場合にキャプチャすることができ、その値は関数によって提供される結果になります。
したがって、前の例は次のように単純化することができます。

```
   PD2 =: 4 : 0
w =. x - y
if. x > y do. w else. - w end.
)

   3 PD 4
1
```

12.3.4 Blocks
ブロック

上記の例はパターンを示しています：

```
               if. T do. B1 else. B2 end.
```

つまり、式 T が "true"と評価された場合は、式 B1 を実行し、そうでない場合は式 B2 を実行します。
式 T は場合は「真」に評価するとみなされる T は、最初の要素はされない任意の配列に評価 0。

```
   foo =: 3 : 'if.  y do. ''yes''  else. ''no''  end.'
```

foo 1 1 1	foo 'abc'	foo 0	foo 0 1
yes	yes	no	no

より一般的には、 T、B1 および B2 は「ブロック」と呼ばれるものであってもよい。ブロックはアイテムのシーケンスであり、アイテムは式または制御構造のいずれかです。ブロックによって提供される結果は、ブロックの最後の項目の値です。
ここでは、T ブロックと B2 ブロックのそれぞれがシーケンスで構成されるリストの合計を形成する例を示します。

```
   sum =: 3 : 0
if.
    length  =. # y     NB. T block
    length  = 0        NB. T block
do.
    0                  NB. B1 block
else.
    first =. {. y      NB. B2 block
    rest  =. }. y      NB. B2 block
    first + sum rest   NB. B2 block
end.
)

   sum 1 2 3
6
```

ここでは、T ブロック(真または偽)の値がシーケンス内の最後の式の値(length = 0)であることがわかります。
ブロックの項目は(内部)制御構造であってもよい。たとえば、お粥の温度を分類する関数は次のようになります。

```
   ClaTePo =: 3 : 0
if. y > 80  do.      'too hot'
else.
     if. y < 60 do. 'too cold'
     else.          'just right'
     end.
end.
)

   ClaTePo 70
just right
```

12.3.5 Variants of if.
if のバリアント

最後の例のもう少しのバージョンは次のとおりです。

```
   CLATEPO =: 3 : 0
```

```
if.     y > 80 do. 'too hot'
elseif. y < 60 do. 'too cold'
elseif. 1      do. 'just right'
end.
)

   CLATEPO 70
just right
```

パターンを示す：

```
        if. T1 do. B1 elseif. T2 do. B2  ... elseif. Tn do. Bn end.
```

このスキームによれば、すべてのテスト T1 ... Tn が失敗すると、ブロック B1 .. Bn のどれも実行されないことに留意されたい。したがって、上の CLATEPO の例のように、Tn を一定値 1 でキャッチオールテストにすることができます。
すべてのテストが失敗し、ブロック B0 ... Bn のどれも実行されない場合、結果は i になります。0 0 は null 値の J 規約です。

```
   foo =: 3 : 'if. y = 1 do. 99 elseif. y = 2 do. 77 end. '
   (i. 0 0) -: foo 0
1
```

パターンもあります:

```
         if. T do. B end.
```

ここで B が実行されるか、実行されません。たとえば、プラスの違いはもう一度：

```
   PD =: 4 : 0
z =. x - y
if. y > x do. z =. y - x end.
z
)
   3 PD 4
1
```

12.3.6 The select. Control Structure

選択。制御構造

if を使用して、名前を分類する動詞のこの例を考えてみましょう。 制御構造。

```
   class =: 3 : 0
t =. 4 !: 0 < y
if.     t = 0  do. 'noun'
elseif. t = 1  do. 'adverb'
elseif. t = 2  do. 'conjunction'
elseif. t = 3  do. 'verb'
elseif. 1      do. 'bad name'
end.
)

   class 'class'
verb
   class 'oops'
bad name
```

制御構造である、Select によって、より洗練された処方が可能である。

```
   CLASS =: 3 : 0
select.  4 !: 0 < y
case. 0 do. 'noun'
case. 1 do. 'adverb'
case. 2 do. 'conjunction'
case. 3 do. 'verb'
case.   do. 'bad name'
end.
)

   CLASS 'CLASS'
verb
   CLASS 'oops'
bad name
```

私たちが 3 方向分類、名詞、動詞、演算子(副詞や結合詞を意味する)にのみ興味があるとします。もちろん、次のように書くことができます：

```
   Class =: 3 : 0
select.  4 !: 0 < y
case. 0 do. 'noun'
case. 1 do. 'operator'
case. 2 do. 'operator'
case. 3 do. 'verb'
case.   do. 'bad name'
end.
)
```

これは、以下のように省略することができます。

```
   Clss =: 3 : 0
select.  4 !: 0 < y
case. 0    do. 'noun'
case. 1;2  do. 'operator'
case. 3    do. 'verb'
case.      do. 'bad name'
end.
)
```

Clss 'Clss'	o =: @:	Clss 'o'	Clss 'oops'
verb	+--+ |@:| +--+	operator	bad name

12.3.7 その間。また、制御構造（The while. and whilst. …）

一般的なパターン

```
            while. T do. B end.
```

ブロック B は、ブロック T が真と評価される限り、繰り返し実行される。階乗関数の例を次に示します。

```
   fact =: 3 : 0
r =. 1
while. y > 1
do.    r  =. r * y
       y =. y - 1
end.
r
)

   fact 5
120
```

The variation whilst. T do. B end. means

```
            B
            while. T do. B end.
```

すなわち、ブロック B は 1 回実行され、ブロック T が真である限り繰り返し実行される。

12.3.8 for.

パターン

```
            for_a. A do. B. end.
```

は、配列 A の各項目 a に対してブロック B を実行することを意味します。ここでは、任意の名前を指定できます。変数 a は A の各項目の値を順番にとります。たとえば、リストを合計するには：

```
   Sum =: 3 : 0
r =. 0
for_term. y do.  r =. r+term end.
r
)

   Sum 1 2 3
6
```

アイテムの値の 変数 a に加えて、変数 a index を使用してアイテムのインデックスを指定できます。たとえば、この関数は項目に番号を付けます。

```
   f3 =: 3 : 0
r =. 0 2 $ 0
for_item. y do.  r =. r , (item_index; item) end.
r
)

   f3 'ab';'cdef';'gh'
+-+----+
|0|ab  |
+-+----+
|1|cdef|
+-+----+
|2|gh  |
+-+----+
```

もう一つのバリエーションは、のパターンです。そうです。B 端。 ブロック B は A の項目がある回数だけ実行される。たとえば、リストの項目を数える動詞があります。

```
   f4 =: 3 : 0
count =. 0
for. y do. count =. count+1 end.
)

   f4 'hello'
5
```

12.3.9 The return. Control Word
リターン。コントロールワード

小さな整数スカラをテストする動詞を定義する必要があるとします。「小さい」とは、100 未満の大きさを意味します。これは、

```
   test =: 3 : 0
    if. 4 = 3 !:0 y  do.        NB. integer ?
        if. 0 = # $ y  do.      NB. scalar ?
            100 > | y           NB. small ?
        else. 0
        end.
    else. 0
    end.
)
```

test 17	test 'hello'	test 1 2 3	test 101
1	0	0	0

明らかに、整数を調べてから整数を小数にする必要があります。したがって、ネストされた ifs。
ここでは、代替案があります：

```
   test =: 3 : 0
    if. 4 ~: 3!:0 y do. 0 return. end.    NB. not integer
    if. 0 ~: # $ y  do. 0 return. end.    NB. not scalar
    100 > | y                             NB. small ?
)
```

test 17	test 'hello'	test 1 2 3	test 101
1	0	0	0

リターン の影響。制御語は動詞のそれ以上の実行を短絡し、この例では各復帰時に 0 になる最も最近計算された値を送ることである。。

12.3.10 Other Control Structures
その他の制御構造

第 29 章では、制御構造の trv. catch. end.について説明します。
他の制御語と構造は、J 辞書でカバーされています。

☆第 12 章終了。

第 13 章：Explicit Operators
明示的演算子

この章では、演算子の明示的な定義、つまりコロン(:)結合で定義された副詞と結合詞について説明します。
明示的定義のスキームは次のとおりです。

```
        1 : body     is an adverb(副詞です)
        2 : body     is a conjunction(接続詞です)
```

bodv は 1 行以上のテキスト行です。そのように定義された演算子によって生成される結果の可能性は、暗黙の動詞、明示的な動詞、名詞または他の演算子である。私たちはそれぞれのケースを順番に見ていきます。

13.1 Operators Generating Tacit Verbs
暗黙の動詞を生成する演算子

第 07 章から組み込み階級結合を思い出してください。任意の動詞 u について、式 "0"は、その引数の 0 セル（スカラー）に u を適用する動詞である。
ここで、副詞 A を定義して、スキームに従って動詞を生成することを目指すと仮定します。
任意の動詞 u

```
      u A   is to be     u " 0
```

副詞 A は、次のように明示的に定義されています。

```
   A =: 1 : 'u " 0'      f =: <A      f 1 2
   1 : 'u " 0'           <"0          +-+-+
                                      |1|2|
                                      +-+-+
```

定義(A =: 1: 'u "0') では、コロンの左引数は 1 で、副詞を意味します。
右の引数は文字列'u "0'です。この文字列は暗黙の動詞の形式を持ち、u は任意の動詞を副詞 A に引数として与えることを表します。副詞の明示的な定義では、常に u です。
英語の文法では、副詞は動詞を修正するため、副詞がそう呼ばれています。対照的に、J では、副詞と結合詞は一般に名詞や動詞を引数として取ることができます。以下の例では、副詞 W は、スキームに従って動詞を生成します。for integer u

```
      u W    is to be   < " u
```

つまり、U W boxesrank- の u 引数の細胞。W の定義は次のように表示されます。

```
   W =: 1:  '<"u'     0 W     z =:  'abc'     0 W z       1 W z
   1:  '<"u'          <"0     abc             +-+-+-+     +---+
                                              |a|b|c|     |abc|
                                              +-+-+-+     +---+
```

副詞の別の例については、dyad : #を思い出してください。ここで、x: #y は、ビット列 x に従って y から項目を選択します。

`y =: 1 0 2 3`	`1 0 1 1: #y`
`1 0 2 3`	`1 2 3`

0 より大きい項目を選択するには、テスト動詞(>&0)を適用して、

`y`	`>&0 y`	`(> 0 y): #y`
`1 0 2 3`	`1 0 1 1`	`1 2 3`

0 より大きい項目を選択するための暗黙の動詞 は、フォーク f として書くことができます：

`f =: >&0: #]`	`fy`
`>&0: #]`	`1 2 3`

このフォーク は、テスト>&0 の代わりに動詞が供給されることに応じて、項目を選択する動詞を生成するために副詞 B に一般化することができます。

```
   B =: 1:  'u: #]'
```

テスト動詞として&1 を指定した 場合：

`g =: (>&1) B`	`y`	`g y`
`>&1: #]`	`1 0 2 3`	`2 3`

私たちは、身体のことを見る B はで、生成するフォークです u が供給される引数動詞放置します。2 つの引数を取る連合は、(2: '...')で定義されます。左の引数は u、右は v です。たとえば、1 つの動詞を適用して別の動詞を結果、つまり構成に適用するには、THEN を連想させるとします。私たちが望む計画は次のとおりです。

```
          u THEN v     is to be v @: u
```

THEN の定義は次のとおりです。

`THEN =: 2:  'v @: u'`	`h =: *: THEN <`	`h 1 2 3`

`2:  'v @: u'`	`<@: *:`	`+ ----- +`
		`| 1 4 9 |`
		`+ ----- +`

別の例として、0 より大きいリストの項目を(: #で)カウントすることを検討してください。これを行う動詞は次のようになります。

`foo =: # @: (>&0 # ])`	`y`	`foo y`
`#@:(>&0 # ])`	`1 0 2 3`	`3`

我々は一般化することができ FOO を与えられた動詞適用するには、U を 別の特定の動詞で選択した項目に、V。我々はスキームと一緒に C を定義する

```
            u C v   is to be   u @: (v # ])
```

C の定義は簡単です。

C =: 2 : 'u @: (v #])'	f =: # C (>&0)	y	f y
2 : 'u @: (v #])'	#@:(>&0 #])	1 0 2 3	3

13.1.1 Multiline Bodies
複数行の本体

コロンの右の引数は、演算子の定義の本体と呼ばれることがあります。これまでの例では、本文は文字列であり、例えば、「v: @: u」のような模擬暗黙の動詞でした。これは私たちのオペレーターによって提供される動詞です。より一般的には、本体は複数の行にすることができます。考え方は、演算子を引数に適用すると、全身が実行されるということです。つまり、各行は順に評価され、結果は評価された最後の行の値になります。これは明示的な動詞とまったく同じですが、結果は型 "array"ではなく "function"の型の値です。
ここでは、複数行の本体の例を示します。前の例は 2 つのステップで実行されています。v によって選択されたアイテムに u を適用するために、D の結合スキームを書くことができる。

```
        u D v  is to be  (u @: select) where select is v # ]
```

および D によって定義されます。

```
   D =: 2 : 0
select =: v # ]
u @: select
)
```

再び 0 より大きな項目を数えると、

f =: # D (>&0)	y	f y
#@:select	1 0 2 3	3

D の最初の行は、右の引数から内部関数 select を計算します。2 番目の行は、左の引数で select を構成し、これは D によって出力される結果動詞です。
今、この定義は望ましくない特徴を持っています。select はグローバル(with =:)として定義されています。選択したものがローカルであれば良いでしょう。
しかし、我々は動詞の値を見ることで、見ることができる F 以上、それを選択し、我々は適用されたときに利用可能でなければなりません f は。select が D に対してローカルの 場合、必要なときには使用できません。
実際には、「修正」副詞(f。)(英字-f 点)を使用してローカル 選択を行うことができます。動詞に「修正」を適用すると、対応する動詞によって名前が置き換えられた同等の動詞が生成されます定義。つまり、「修正」は、暗黙の動詞をそのプリミティブに分解します。例えば：

p =: +	q =: *	r =: p,q	r f.
+	*	p , q	+ , *

選択 をローカル にするために修正を使用する方法は次のとおりです。次の例では、最後の行に result-expression を修正していることに注目してください。

```
   E =: 2 : 0
select =. v # ]
(u @: select) f.
)
```

0 より大きい数を数える動詞を書くことができます：

g =: # E (>&0)	y	g y
#@:(>&0 #])	1 0 2 3	3

f とは異なり 、g にはローカル名がないことがわかります。

13.2 New Definitions from Old
古いものからの新しい定義

数字のリストを与えられた関数を発展させることを目指して、リスト内の 2 つの隣り合うもの、前と次のもののそれぞれの平均で各数字を置き換えることを目指すと仮定しよう。最初または最後に、隣人がゼロであると仮定します。
適切な「データ平滑化」機能を書くことができる。

```
   sh    =: |. !. 0    NB. shift, entering zero
   prev  =: _1 & sh    NB. right shift
   next  =: 1 & sh     NB. left shift
   halve =: -:

   smoo  =: halve @: (prev + next)
```

私たちが見るかもしれない 数 N のリストのために：

N =: 6 2 8 2 4	prev N	next N	smoo N
6 2 8 2 4	0 6 2 8 2	2 8 2 4 0	1 7 2 6 1

ここで、ゼロにシフトするのではなく、データを回転させる別のスムージング関数も必要であるとします。(データは、例えば、繰り返し波形のサンプルであってもよい)。
smoo から必要な変更は、シフト動詞 sh が回転動詞、つまり、(|。)にならなければならないということだけです。
smoo の定義が大規模で複雑な場合は、再度入力することを避けることをお勧めします。代わりに、名前 sh が "rotate"を意味する環境で、すでに定義した定義を再評価することができます。この環境は、便利な少し副詞によって提供することができる、SMOO はと言います|。
その引数のために(回転する)

```
   SMOO =: 1 : ('sh =. u' ; 'smoo f.')
```

smoo の回転型は次のように与えられます。

```
   rv =: |. SMOO
   rv
-:@:(_1&|. + 1&|.)
```

N	N	N
smoo N	smoo N	smoo N

この例では、式(smoo)を関数に一般化するために副詞を使用しています。smoo は sh に関して定義されているので、 引数として sh をとる関数に一般化しています。

13.3 Operators Generating Explicit Verbs

明示的な動詞を生成する演算子

私たちは一緒に定義することを目指していると H は、スキームと、言うに：

```
  u H v    is to be     3 : 0
                       z =. v y
                       y u z
                       )
```

これを行うには、乱雑なやり方があります。私はあなたに厄介なやり方を最初に見せるので、きちんとした方法のメリットを評価することができます。
わかりやすい方法：以前の例と同じスタイルで H を書くことができます。つまり、定義の本体は、演算子が引数に適用されたときに渡される値を計算します。この場合、値は 3: string の形式になります。ここで string は引数から構築する必要があります。例えば：

```
   H =:  2 : 0
    U =. 5!:5 < 'u'
    V =. 5!:5 < 'v'
    string =. 'z =. ', V , 'y',  LF
    string =. string , 'y ', U , ' z', LF
    3 : string
)
```

我々は見る

```
   foo =: + H *:
   foo 5
30
```

結合 H はかなり醜いですが、生成された関数 foo の値は明白です。

```
   foo
3: 0
z =。*: y
y + z
)
```

今我々はこの結合を定義するためのきちんとした方法に来る。これまでは、結果を提供するためにボディが実行される演算子を見てきました。彼らが第一種の運営者であるとしよう。ここで、演算子の本体は実行されず、生成される動詞のテンプレートとして機能する第 2 の種類の演算子を調べます。例えば：

```
   K =: 2 : 0
z =. v y
y u z
)
```

明らかに、K の定義は H の定義よりも小さいが、同等である。K の本体には、演算子 の引数変数 u と v と、生成された動詞の引数変数 v の両方が含まれていることに注意してください。これは、オペレータが第 2 種のものであることを決定する引数変数の組み合わせである。

```
   bar =: + K *:
   bar 5
30
```

生成された動詞バーは foo と等価ですが、異なって表示されます。

```
   bar
+ (2 : 0) *:
z =. v y
y u z
)
```

今度は、第 2 種の演算子の例をさらに詳しく見ていきます。

13.3.1 Adverb Generating Monad

副詞を生成する母音

次の明示的なモナド動詞、e を考えてみましょう。test-verb >&0 を適用して、0 より大きい項目を選択します。

`e =: 3 : '(>&0 y) # y'`	`y`	`e y`
`3 : '(>&0 y) # y'`	`1 0 2 3`	`1 2 3`

e を 一般化して、与えられたテスト動詞に従って項目を選択する副詞 F say を形成することができる。私たちが望む体系は：任意の動詞 u:

```
       u F   is to be    3 : '(u y) # y'
```

副詞 F は次のように定義されます。

```
   F  =: 1 : '(u y) # y'
```

動詞>&1 F は、1 より大きい項目を選択します。

`y`	`>&1 F y`
`1 0 2 3`	`2 3`

F の本文では 、変数 u は、副詞 F に引数として与えられる動詞を表します。この引き数が >&1 の場合、 y は生成された明示的動詞 3 の引き数('>&1 y): #y'を表します。
つまり、生成された動詞を定義する私たちの方法は、明示的な定義の本文を、 動詞を代入する場所に u を書き出すことです。

13.3.2 Adverb Generating Explicit Dyad

明示的ダイアドを生成する副詞

我々は副詞たいと W をスキームと、言う、 ：任意の動詞のための u

```
          u W     is to be      4 : '(u x) + (u y)'
```

第 12 章 から、明示的ダイアドを書く別の方法があることを思い出してください。4 で始めるのではなく、3: で始めることができます 。そして、ソリッドコロンがモナドとダイアデ

ィックの場合を分ける複数行の本体を記述します。ここではモナドケースはないので、上記のスキームは次のように書くことができます：

```
          u W    is to be     3 : 0
                             :
                             (u x) + (u y)
                             )
```

副詞の明示的な定義 W は素直に、次のとおりです。

```
   W =: 1 : 0
:
(u x) + (u y)
)
```

私たちは見る：

(*: 2) + (*: 16)	(*: 2) + (*: 16)
2 (*: W) 16	2 (*: W) 16

別の例として、我々は副詞をしたいと仮定し、T が与えられた動詞適用するために、と言う の u をベクトル引数でスカラーのすべての組み合わせに x ベクトル引数でスカラーと y の。そこ内蔵の副詞である/このため、「表」と呼ばれるが、しかし、ここではホームメイドのバージョンです。スキームは次のとおりです。

```
          u T   is to be 4 : ' x (u " 0 0) " 0 1  y'
```

あれは、

```
          u T   is to be 3 : 0
                           :
                           x  (u " 0 0) " 0 1  y
                           )
```

したがって T は

```
   T =: 1 : 0
:
x  ((u " 0 0) " 0 1)  y
)
```

その結果は：

```
    1 2 3 + T 4 5 6 7
5 6 7  8
6 7 8  9
7 8 9 10
```

13.3.3 Conjunction Generating Explicit Monad
明示的な Monad を生成する結合

接続詞は、u と v という 2 つの引数をとります。
前と同じように、明示的な動詞の本文を書き出すことによって生成された動詞を指定します。ここで、y は生成された動詞の引数を表し、u と v は結合詞に引数 - 動詞が供給されることを表します。この例では、本文は複数行です。前述のように、 u は v によって選択されたアイテムに適用されます

```
   G  =: 2 : 0
```

```
selected =. (v y) # y
u selected
)
```

0 より大きい数を数える動詞は、 : #G(>&0)と書くことができます：

y	# G (>&0) y
1 0 2 3	3

13.3.4 Generating a Explicit Dyad
明示的ダイアドの生成

動詞 u と v について、概略的に 結合詞 H を求めたいとする

```
        u H v は 4:  '(ux)+(vy)'
```

またはこれと同等に、上記のとおりです。

```
        u H v は 3: 0 でなければならない
                          :
                         (ux)+(vy)
                         )
```

明示的な定義 H は素直に、次のとおりです。

```
   H = 2: 0
:
(ux)+(vy)
)
```

見てみましょう：

(*: 2) + (%: 16)	2 (*: H %:) 16
8	8

13.3.5 Alternative Names for Argument-Variables
引数 - 変数の代替名

完全性を期すために、引数を名詞に制限するために、つまり動詞がエラーとして通知されるようにするために、演算子への引数は u および v ではなく m および n と命名されることに注意する必要があります。
さらに、歴史的な理由から、引数変数が x または y、またはその両方である場合、最初の種類の演算子が得られます。つまり、u または v または m または n がない場合、x および y は u および v と等価 です。
これらの代替名は、これ以上考慮されません。

13.3.6 Review
レビュー

ここまでは、1: body または 2 : body で導入された演算子には、2 種類の定義があることがわかりました。

- 最初の種類の演算子では、結果の値を計算するために本体が実行されます(評価されます)。結果はどのようなタイプであってもかまいません。体内で発生する引数変数は、u または v またはその両方です。
- 第 2 種の演算子では、結果は常に明示的な関数になります。演算子の本体は実行されず、生成された関数の本体になります。ここで x と y は通常の方法で生成された関数への引数であり、この本体の u または v は演算子への引数の値を受け取るプレースホルダーです。
J システムは、どの種類の引数変数 uvxv が本体内に存在するかを判断することによって、どのような種類のものであるかを認識する。もし我々が(u または v)と(x または y)の両方を持っていれば、演算子は第 2 種のものです。さもなければそれは最初の種類です。

13.3.7 Executing the Body (Or Not)
ボディの実行(または実行しない)

上記のように、最初の種類の演算子では、引数が与えられたときに本体が実行(または評価)されます。これを実証することができます。
まず、引数を画面に表示するユーティリティ動詞を次に示します。

```
  display =: (1!:2)&2
```

ここで、最初の種類の演算子に'hello'という表示を挿入します。

```
    R =: 2 : 0
display 'hello'
select =. v # ]
(u @: select) f.
)
```

R がその議論に適用されるとき、本体が明らかに実行されます。

```
   f =: # R (>&0)
hello
  f 1 0 2 0 3
3
```

対照的に、第 2 種の演算子の場合、引数が与えられると、本体は実行されず、結果関数の本体になります(引数を代入した後)。これを実証するには、演算子の本体に'hello'という表示を挿入し、それが結果関数の一部になることを観察します。

```
   S =: 2 : 0
display 'hello'
selected =. (v y) # y
u selected
)
```

我々はのボディていることがわかり S のときに実行されていない S は、その引数に適用されますが、生成された動詞のときには、実行された G が適用されます。

```
   g =: # S (>&0)
   g 1 0 2 0 3
hello
3
```

13.4 Operators Generating Nouns
名詞を生成する演算子

演算子は動詞だけでなく名詞も生成できます。ここに例があります。
関数の固定点 F が値であり、P は そのようなこと(FP)= P。私たちは取る場合は 、F をします

```
   f =: 3 : '2.8 * y * (1                    y)'
```

0.642857 が f の固定小数点である ことがわかります

```
   f 0.642857
0.642857
```

すべての関数に固定小数点があるわけではありませんが、ある場合はそれを見つけることができます。適切な開始値を選択して、変更がなくなるまで関数を反復することができます(^: - 第 10 章を参照)。粗固定小数点ファインダは、与えられた関数を引数としてとり、開始値として 0.5 をとる副詞 FPF として記述することができます。

FPF =: 1 : '(u ^: _) 0.5'	p =: f FPF	f p
1 : '(u ^: _) 0.5'	0.642857	0.642857

13.5 Generating Noun or Verb
名詞または動詞の生成

J の 2 行を考えてみましょう。

```
               sum =: + /
               mean =: sum  % #
```

時にはよりスムーズなプレゼンテーションがあるかもしれません：

```
               mean =: sum % #  where sum =: +/
```

我々はのために利用可能適し定義持っていた提供。これはどう？

```
   where =: 2:  'u'
```

だから私たちは言うことができます：

```
   平均=: 合計%: #ここで、合計=: + /
```

結果は期待通りです：

mean	mean
mean 1 2 3 4	mean 1 2 3 4

どこの動詞や名詞になることができる の正しい引数：

```
    (z + 1)*(z-1)ここで z = 7
48
```

where は右の引数を無視する結合詞ですが、右の引数を評価すると、代入によって左に利用できるようになります。上記の sum と z への代入は、通常のグローバル代入であるため、sum または z をどこで ローカライズしないのかについて注意してください。

13.6 Operators Generating Operators

演算子を生成する演算子

副詞を生成する副詞の例 Operators Generating Operators を次に示します。
まず、(第 15 章で述べたように)結合詞に一つの議論を与えれば、副詞が得られることに注意してください。表現(@: * :)は、「四角で作った」という副詞です。説明する:

CS =: @: *:	- CS	- CS 2 3	- *: 2 3
@:*:	-@:*:	_4 _9	_4 _9

今度は、このセクションの主要な例に戻ります。このスキームに従って副詞を生成する明示的な副詞 Ksay を定義することを目指す：動詞 u

```
        u K    is to be     @: u
```

副詞 K は以下のように定義することができる。副詞 K が副詞 L を与えることが分かります。

K = 1: '@: u'	L =: *: K	- L	- L 2 3
1: '@: u'	@: *:	- @: *:	_4 _9

☆第 13 章終了。

第 14 章 : Gerunds
動名詞

gerund(動名詞)とは何ですか、それは何のために良いですか？簡単に言うと、gerund は動名詞を表します。主に動名詞を単一の引数としてオペレータに提供するのに便利です。
この章の計画は次のとおりです。

1．gerunds を紹介する
2．gerunds を引数として取り込むことができる組み込みの演算子を見る
3．巨大な引数を取るユーザ定義の演算子を見る

14.1 Making Gerunds: The Tie Conjunction
ジェランドを作る: The Tie Conjunction

第 10 章 から、いくつかのケースで動詞をどのように定義したかを思い出してください。リマインダとしての小さな例がここにあります。数値の絶対値を見つけるために、X 、我々は計算を(+ X) 、または(-x)の数は、このように、負の場合：

`abs =: + ` - @. (< & 0)`	`abs =: + ` - @. (< & 0)`
`abs _3`	`abs _3`

式(+ ` - )は動詞のリストのように見えます。ここでは、2 つの動詞+ と-は、 "結び"結合詞(`、backquote、different from ')と結びついて、gerund を生成します。

```
    + ` -
+-+-+
|+|-|
+-+-+
```

gerund (+ ` -) は 2 つのボックスのリストであり、それぞれに動詞の表現が含まれていることがわかります。gerund は名詞です - 箱のリスト。3 つの動詞を表す別の gerund があります：

```
G =: + ` - ` abs
   G
+-+-+---+
|+|-|abs|
+-+-+---+
```

各ボックスの内部には、動詞を表す、または符号化するデータ構造がある。ここでは、この表現の詳細については関心がありません。これについては第 27 章で説明します。

14.2 Recovering the Verbs from a Gerund
動名詞(Gerund)からの動詞の回復

gerund に詰め込まれた動詞は、式(`: 6)で示される組み込み副詞" Evoke Gerund "で再度解凍することができます。私たちはこれを EV と呼ぶことにしよう。

```
   EV =: `: 6
```

副詞 EV を gerund に適用すると、gerund のすべての動詞列が生成されます。次の例では、結果 foo は 3 列車、つまりフォークです。

```
   f =: 'f' & ,
   g =: 'g' & ,
```

H =: f `、` g	foo =: H EV	foo 'o'
+-+-+-+ |f|,|g| +-+-+-+	f、g	fogo

個々の動詞は、囲みリスト H を索引付けして EV を適用することによってアンパックすることができます。

H	2 {H	vb =: (2 {H)EV	vb 'o'
+ - + - + - + | f |、| g | + - + - + - +	+ - + | g | + - +	g	go

より短い列車は、再び索引付けすることによって、元気いっぱいから解凍することができます。

H	1 2 {H	tr =: (1 2 {H)EV	tr 'a'
+-+-+-+	+-+-+-+	+-+-+-+	+-+-+-+

今我々は gerunds の使用に来る。

14.3 Gerunds As Arguments to Built-In Operators
組み込み演算子への引数としての Gerunds

gerunds の主な用途は、複数の動詞を含む単一の引数として演算子に供給できることです。我々は最初に、冗長な議論をしている組み込み演算子を見てから、自家製演算子の例を見ていきます。

14.3.1 Gerund as Argument to APPEND Adverb
APPEND 副詞の引数としての Gerund

式(`: 0)で示される組み込み副詞「APPEND」があります。これは、単一の引数に動詞のリストを適用して、結果のリストを提供します。例えば：

```
   APPEND =: `: 0
   sum    =: +/
   count  =: #
   mean   =: sum % count
   G1     =: count ` sum ` mean
```

G1	foo =: G1 APPEND	foo 1 2 3
+-----+---+----+ |count|sum|mean| +-----+---+----+	count`sum`mean`:0	3 6 2

副詞は APPEND と呼ばれます。なぜなら、gerund 内の個々の動詞の結果が追加され、それがリストに形成されるからです。一般的な体系は、動詞 u、v、w、...に対して

```
(u`v`w...) APPEND y  means  (u y), (v y), (w y) , ...
```

もう一つの例は、一次元のリストではなく、動詞の配列であることを示しています。"Tie" によって形成された動詞 G1 のリストは、表のような配列に再構成することができ、結果の形は同じです。

```
G2 =: 2 2 $ G1
```

```
+-----+-----+
|count|sum  |
+-----+-----+
|mean |count|
+-----+-----+
```

```
G2 APPEND 4 5
```

```
  2 9
4.5 2
```

14.3.2 Gerund as Argument to Agenda Conjunction

Agenda Conjunction への議論としての Gerund

上で定義し た abs 動詞を思い出してください。リマインダーは次のとおりです：

```
abs =: + ` - @. (< & 0)
```

```
+`-@.(<&0)
```

```
abs 6
```

```
6
```

```
abs _6
```

```
6
```

ここでは、「議題」の接尾辞(@。)は、右の動詞を取ります。バリエーションとして、(@。) は右の名詞を取ることもできます。この名詞は単一の数字または数字のリストになります。単一の数字は、動詞から動詞を選択する指標とみなされます。例えば。

```
G =: + ` - ` %
```

```
+-+-+-+
|+|-|%|
+-+-+-+
```

```
f =: G @. 0
```

```
+
```

```
1 f 1
```

```
2
```

数字のリストは、列車を形成するために動詞から動詞を選択する指標のリストとして取られる。次の例では、選択された 2 つの動詞がフックを形成します。

```
G
```

```
+-+-+-+
|+|-|%|
+-+-+-+
```

```
h =: G @. 0 2
```

```
+ %
```

```
h 4
```

```
4.25
```

スキームは、gerund G とインデックス I のためです：

```
G @. I   means   (I { G) EV
```

例えば：

```
G
```

```
+-+-+-+
```

```
(G @. 0 2) 4
```

```
4.25
```

```
((0 2 { G)) EV 4
```

```
4.25
```

```
|+|-|%|
+-+-+-+
```

このスキームでは、上記の索引付けによるアンパックの省略形を示します。次に、もっと構造の整った列車の作り方を見ていきます。列車 T を考えてみましょう。

T =: *(- 1 :)	T 3	T 4
*(- 1 :)	6	12

(T x)= x *(x -1) を計算する。括弧は、T がフックであり、第 2 のアイテムもフックであることを意味する。このように括弧で構造化された列車は、括弧を示すためにボックス化されたインデックスを使用して、雑誌から項目を索引付けすることによって、議題を組み立てることができます。

```
   foo =: (* `                    ` 1:) @. (0 ; 1 2)
```

T	foo	foo 3
*(- 1 :)	*(- 1 :)	6

14.3.3 Gerund as Argument to Insert
挿入の引数としての Gerund

我々は以前、単一の動詞に適用された挿入副詞に遭遇しました：動詞は、リストの連続する項目の間に挿入されます。より一般的には、挿入が gerund に適用されるとき、それはリストから連続した項目の間の gerund からの連続する動詞を挿入します。すなわち、もし G が gerund (f`g`h` ...)であり、X がリスト(x0、x1、x2、x3、...)ならば

```
            G/X    means   x0 f x1 g x2 h x3 ...
```

ger =: + `%	ger / 1 2 3	1 + 2%3
+-+-+ |+|%| +-+-+	1.66667	1.66667

gerund が短すぎると、必要な数の動詞を構成するために周期的に再利用されます。これは、挿入されたときに 1 動詞の gerund が挿入された動詞と同じように動作することを意味します。

14.3.4 Gerund as argument to POWER conjunction
POWER 接続の引数としての Gerund

第 10 章 から、POWER 結合(^ :)は、右引数として、左引数として与えられた動詞の反復回数を指定することができることを思い出してください。簡単な注意として、1 の 3 倍は 8 です：

```
   double =: +:
   (double ^: 3) 1
```

```
8
```

変形として、反復回数は、動詞の右引数によって計算することができる。このスキームは、動詞 u と v に対して：

```
(u ^: v) y   means   u ^: (v y) y
```

例えば：

```
decr =: <:
```

`double ^: (decr 3)3`	`(double ^: decr)3`
`12`	`12`

より一般的には、正しい議論は gerund として与えることができ、その中の動詞は反復プロセスの始めにいくつかの計算を行います。スキームは次のとおりです。

```
u ^: (v1 ` v2) y   means   u ^: (v1 y) (v2 y)
```

例として、フィボナッチシーケンスを計算する動詞を定義します。ここで、各項は、前の 2 つの項の和です。動詞は、用語の数を指定するために引数を取るので、例えば、`FIB 6` に `0` を与えるようにします。`1 1 2 3 5`
反復される動詞は、u は言う、最後の二つの合計を追加することによって、前のシーケンスから次のシーケンスを生成します。我々が定義するもの：

```
u       =: , sumlast2
sumlast2 =: +/ @ last2
last2    =: _2 & {.
```

シーケンス `0 1` で始まる反復スキームは、

`u 0 1`	`u 0 1`	`u 0 1`
`u u 0 1`	`u u 0 1`	`u u 0 1`

ここでは、gerund の 2 つの動詞を定義します。n 項のシーケンスを生成するには、動詞 u を(n-2)回適用する必要があるので、引数 y から反復回数を計算する動詞 v1 は次のようになります。

```
v1 =:   &2
```

動詞 v2 の引数から開始値を計算し、y は、我々は計算定数関数になりたい `0 1` のどのような値 y と。

```
v2 = 3:  '0 1'
```

これですべてをまとめることができます：

`FIB =: u ^: (v1 `v2)`	`FIB 6`
`(, sumlast2)^:(v1`v2)`	`0 1 1 2 3 5`

この例では、gerund(v1 と v2)の 2 つの動詞をモナド動詞(u)に示し、繰り返しの最初にいくつかの計算を実行しました。二項動詞はどうですか？
まず、反復二項動詞では、最初の引数は実際に反復されるモナドを与えるためにバインドされているので、スキームは次のようになります。

```
        x  u ^: k  y    means    (x&u) ^: k y
```

定数 k ではなく、3 つの動詞 U V と W を gerund として提示して事前計算を行うことができます。スキームは次のとおりです。

```
       x u ^: (U`V`W) y  means  (((x U y)&u) ^: (x V y))  (x W y)
```

または同等にフォークとして：

```
        u ^: (U`V`W)   means   U (u ^: V) W
```

たとえば、次のように定義するとします。

```
   U =: [
   V =: 2:
   W =: ]
```

次に、以下の p と q が等しいことがわかります。3 が 2 に 4 を 2 加算すると 10 になります。

p =: + ^: (U'V'W)	3 p 4	q =: U(+ ^: V)W	3 q 4
+ :(U、V、W)	10	U + ^: V W	10

14.3.5 Gerund as Argument to Amend

訂正の議論としてのジェーンド

第 六章の「改正」副詞を思い出してください。式(新しいインデックス}古い) の修正バージョン生成古いを有する、新規でアイテムとして指数。例えば：

```
      'o' 1} 'baron'
boron
```

より一般的には、 "Amend"副詞は、3 つの動詞、例えば U`V`W の派生語である議論をとることができます。スキームは次のとおりです。

```
       x (U`V`W) } y  means (x U y) (x V y) } (x W y)
```

つまり、新しい項目、インデックス、および「古い」配列はすべて、指定された x と y から計算されます。
以下はその例です(ディクショナリーから適応されています)。i 番目の行に定数 k を掛けて行列を修正する動詞 R を定義しよう。R の左引数はリスト ik で あり、右引数は元の行列である。 R は、3 つの動詞の gerund に適用される "Amend"副詞として定義されます。

```
   i =: {. @ [    NB. x i y  is  first of x
   k =: {: @ [    NB. x k y  is  last of x
   r =: i { ]     NB. x r y  is  (x i y)'th  row of y

   R =: ((k * r) ` i ` ]) }
```

例えば：

```
   M =: 3 2 $ 2 3 4 5 6 7
   z =: 1 10 NB. 1 times 10
```

z	M	z I m	z k M	z r M	z RM
1 10	2 3 4 5 6 7	1	10	4 5	2 3 40 50 6 7

14.4 Gerunds as Arguments to User-Defined Operators
ユーザー定義演算子への引数としての Gerunds

前のセクションでは、組み込み演算子(副詞または結合詞)に gerund を供給することを示しました。ここでは、gerund を引数として扱う独自の演算子を定義します。
演算子の主な考慮点は、gerund 引数から個々の動詞を回復する方法です。ここで便利なのは、@のアジェンダです。私たちは上で見ました。1 つまたは複数の動詞を gerund から選択できることを思い出してください。

G	G @. 0	G @.0 2
+ - + - + - + | + | - |% | + - + - + - +	+	+ %

オペレーターのために今。たとえば、フォークのような動詞を生成 するために、副詞 A を定義しましょう。

```
          x (f `g ` h  A)  y   is to mean   (f x) g (h y)

  A =: 1 : 0
f =. u @. 0
g =. u @. 1
h =. u @. 2
((f @ [) g (h @ ]))  f.
)
```

A を示すために、ここでは、x の最初の項目を最後の y に結合する動詞があります。最初と最後の項目は組み込み動詞{}によって生成され ます。(左括弧の頭、「頭」)と{:(左括弧のコロン、「尾」と呼ばれます)。

H =: {. ` , ` {:	zip =: H A	'abc' zip 'xyz'
+--+-+--+ |{.|,|{:| +--+-+--+	{.@[, {:@]	az

14.4.1 The Abelson and Sussman Accumulator

エーベルソンとサスマンアキュムレータ

gerund 引数を持つユーザー定義の明示的演算子の別の例を次に示します。Abelson と Sussman(「コンピュータプログラムの構造と解釈」、MIT Press 1985)は、さまざまな計算がすべて「アキュムレータ」と呼ばれる次の一般計画にどのように適合しているかを説明しています。

引数からの項目(リスト)は、「フィルタリング」機能で選択されます。選択された各アイテムに対して、値が「マッピング」機能を使用して計算されます。別々のマッピングの結果は、「結合」機能を有する全体的な結果に結合される。この計画は、容易に、副詞として J で実現することができる ACC は、次のように言って、。

```
ACC =: 1 : 0
com =. u @. 0
map =. u @. 1
fil =. u @. 2
((com /)  @:  map  @:  (#~ fil))  f.
)
```

ACC は、引数として 3 つの動詞、順にコンバイナ、マップ、およびフィルタの gerund をとります。例として、与えられたリストの奇数の二乗和を計算します。ここで、奇数をテストするフィルタは(2&|)。

```
      (+ ` *: ` (2&|)) ACC 1 2 3 4
10
```

☆第 14 章終了。

第 15 章：Tacit Operators
暗黙の演算子

15.1 Introduction
はじめに

J には、多くのビルトイン演算子(副詞と結合詞)が用意されています。では第 13 章私たちは、明示的に私たち自身の演算子を定義を見ました。この章では、副詞を暗黙のうちに定義しています。

15.2 Adverbs from Conjunctions
副詞からの副詞

リコール章 07 ランク組み合わせは、(「) 。例えば、動詞は(<」 0) ボックスを適用(<) 引数の各ランク 0(スカラー)の項目に。

```
   <"0  'abc'
+-+-+-+
|a|b|c|
+-+-+-+
```

接続詞には 2 つの引数があります。私たちが 1 つだけを供給すれば、結果は副詞になります。例えば、与えられた動詞を各スカラーに適用するための副詞は、("0"

```
each =: " 0
"0
```

```
< each
<"0
```

```
z =: < each 'abc'
+-+-+-+
|a|b|c|
+-+-+-+
```

スキームは、接続詞 C と名詞 N の場合、式(CN)は次のような副詞を表します。

```
x (C N)  means  x C N
```

結合詞に供給される引数は、名詞または動詞、または左または右になります。全体として、4 つの同様の方式があります。

```
x (C N)  means  x C N
x (C V)  means  x C V
x (N C)  means  N C x
x (V C)  means  V C x
```

シーケンス CN CV NC および CV は「ビッド」と呼ばれます。それらは、2 つの引数の関数を取り、引数の 1 つの値を修正して 1 つの引数の関数を得るボンディングの形式です。しかし、ダイアディック動詞(例えば+と 2 のように)と結合詞を結合することには違いがあります。接続すると、&のような結合演算子は必要ありません。私達はちょうど書く(0」)なし介在オペレータとの理由がある場合にはということです。+&2、省い&与える+ 2 を意味している：のモナド場合は適用さ+を与え、2 に 2 を。しかし、接続詞はモナドの場合を持たないので、bident ("0")は結合として認識されます。
「アンダー」の連想語を思い出してください。章 08 これにより F&.G は適用さ動詞であるグラムを、次に、その引数に f のの、逆グラム。私たちが f と g をとるとすれば：

```
   f =: 'f' & ,
   g =: >
```

f が各ボックスの内側に適用されている ことがわかります。

z	(f &. g) z
+-+-+-+ |a|b|c| +-+-+-+	+--+--+--+ |fa|fb|fc| +--+--+--+

さて、フォームの使用して CV を、我々は副詞を定義することができ、それぞれが 「各ボックス内」を意味します：

EACH =: &. >	f EACH	z	f EACH z
&.>	f&.>	+-+-+-+ |a|b|c| +-+-+-+	+--+--+--+ |fa|fb|fc| +--+--+--+

15.3 Compositions of Adverbs
副詞の構成

場合 A と B は副詞であり、次いで、二座(AB)が適用さ副詞示し A、次いで及び B を。スキームは次のとおりです。

```
        x (A B)   means (x  A) B
```

15.3.1 Example: Cumulative Sums and Products
例：累計と積

副詞¥(バックスラッシュ、Prefix と呼ばれる)が組み込まれています。式 f ¥ y では、動詞 f は、y の連続するより長い先頭セグメントに適用されます。例えば：

```
   < \ 'abc'
+-+--+---+
|a|ab|abc|
+-+--+---+
```

式+/ \ y は、y の累積合計を生成します。

```
   +/ \ 1 2 3
1 3 6
```

累積的な合計、製品などを生成するための副詞は、2 つの副詞の象徴として書くことができます。

```
   cum =: / \    NB. adverb adverb
```

z =: 2 3 4	+ cum z	* cum z
2 3 4	2 5 9	2 6 24

15.3.2 Generating Trains
列車の生成

ここで、副詞を定義して、動詞の列、つまりフックまたはフォークを生成する方法を見ていきます。
最初からリコール第 14 章 タイの組み合わせ(`動名詞を作る)、および EVOKE 動名詞の副詞(`: 6)動名詞からの列車を作ります。
ここで、A と B が副詞であるとします。

```
   A =: *`     NB. verb conjunction
   B =: `: 6   NB. conjunction noun
```

次に、複合副詞

```
   H =: AB
```

フックメーカーです。したがって、<: H は、"x times x-1" というフック* <: を生成します。

<: A	<: AB	h =: <: H	h 5
+-+--+ |*|<:| +-+--+	* <:	* <:	20

15.3.3 Rewriting
書き換え

動詞の定義を、その用語を並べ替えることによって、同等の形式に書き換えることができます。階乗関数 f の定義から始めてみましょう。因数 5 は 120 です。

```
   f =: (*($: @: <:)) `1: @. (= 0 :)
   f 5
120
```

ここでの考え方は、一連のステップで f を$: adverb の形式 に書き換えることです。各ステップは新しい副詞を導入する。最初の新しい副詞は、形式 conj 動詞を持つ A1 です。

```
   A1 =: @. (= 0 :)
   g =: :( *($: @: <:)) `1: A1
   g 5
120
```

副詞 A2 には、conj 動詞の形があります

```
   A2 =:  `1:
   h =: :( *($: @: <:))A2 A1
   h5
120
```

副詞 A3 の形式は adv adv

```
   A3 =: (* `)(`: 6)
```

```
   i =: :( $: @: <:) A3 A2 A1
   i 5
120
```

副詞 A4 の形は conj verb

```
   A4 =: @: <:
   j =: $: A4 A3 A2 A1
   j 5
120
```

A1 と A4 を組み合わせる：

```
   A =: A4 A3 A2 A1
   k =: $: A
   k 5
120
```

エキスパンド A:

```
   m =: $: (@: <:) (* `) (`: 6) (` 1:) (@. (= 0:))
   m 5
120
```

m と f は同じ動詞であることが わかります。

f	m
(* $:@:<:)`1:@.(= 0:)	(* $:@:<:)`1:@.(= 0:)

☆第 15 章終了。

第 16 章：Rearrangements
再編成

この章では、並べ替え、ソート、トランスポーズ、リバース、回転、シフトの各項目の並べ替えについて説明します。

16.1 Permutations
順列

ベクトルの置換は、必ずしも同じ順序ではないが最初の全ての項目を有する別のベクトルである。例えば、z は y の置換であり、ここで、

```
   y =:  'abcde'
abcde
   z =: 4 2 3 1 0 {y
ecdba
```

インデックスベクトル 4 2 3 1 0 は、それ自体、インデックス 0 1 2 3 4 の順列、すなわち、i である。5 であり、したがって 5 次の順列ベクトルと言われる。
この並べ替えの効果に注目してください。最初と最後の項目は入れ替えられ、中間の 3 つの項目はその間で回転します。したがって、この順列は、2 つのアイテムをサイクリングし、3 つのアイテムをサイクリングする組み合わせとして記述することができる。この置換の 6(= 2 * 3)回適用後、元のベクトルに戻ります。

```
   p =: 4 2 3 1 0 & {
```

```
   y
abcde
   p y
ecdba
   p p y
adbce
   p p p p p p y
abcde
```

置換 4 2 3 1 0 は、2 のサイクルおよび 3 のサイクルとして表すことができる。この循環表現を計算するための動詞は、モナディディック C である。

```
   C. 4 2 3 1 0
+-----+---+
|3 1 2|4 0|
+-----+---+
```

したがって、(4 2 3 1 0)は直接表現と呼ばれ、(3 1 2; 4 0) は巡回表現と呼ばれます。
MonadicC.はどちらかの形式を受け入れ、もう一方の形式を生成します：

```
   C. 4 2 3 1 0
+ ----- + --- +
| 3 1 2 | 4 0 |
+ ----- + --- +
   C. 3 1 2 ; 4 0
4 2 3 1 0
```

二項動詞 C は、その左の引数として形式を受け入れ、その右の引数を置換することができます。

```
   y
abcde
   4 2 3 1 0 C. y
ecdba
   (3 1 2 ; 4 0) C. y
ecdba
```

abcde	ecdba	ecdba

16.1.1 Abbreviated Permutations
省略された順列

ダイアディック C. は、(直接的な)置換ベクトルの略語である左の引数を受け入れることができる。その効果は、指定された項目を、指定された項目を一度に 1 つずつ順にテールに移動することです。

y	2 C. y	2 3 C. y
abcde	abdec	abecd

省略形では、連続した項目が元のベクトルから取得されます。次の 2 つの例がどのように異なる結果をもたらすかに注目してください。

y	2 3 C. y	3 C. (2 C. y)
abcde	abecd	abdce

左の引き数がボックス化されている場合、各ボックスは順番にサイクルとして適用されます。

y	(<3 1 2)C. y	(3 1 2 ; 4 0) C. y
abcde	acdbe	ecdba

a が(n 次の)省略順列ベクトルである 場合、a の全長相当は、(U n) によって与えられ、U は効用関数である。

```
   U =: 4 : 0
z =: y | x
((i. y) -. z), z
)
```

例えば、省略順列 a が(1 3)であるとすると、 次のようになります。

y	a =: 1 3	a C.	f =: a U(#y)	f C. y
abcde	1 3	acebd	0 2 4 1 3	acebd

16.1.2 Inverse Permutation
逆順並べ替え

場合 f は全長置換ベクトルであり、その後、逆置換は次式で与えられる (: F /) 。(次のセクションでは、動詞/を見ていきます。)

y	f	z =: f C. y	/: f	(/: f) C. z
abcde	0 2 4 1 3	acebd	0 3 1 4 2	abcde

16.1.3 Atomic Representations of Permutations
順列の原子表現

v が長さ n のベクトルで あれば、完全に ! v の n 個の異なる置換。次数 n のすべての順列の表は、式(tap n)によって生成することができ、ここで、tap は、によって定義されるユーティリティ動詞である。

```
   tap =: i. @ ! A. i.
   tap 3
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

これらの順列は明確に定義された順番であり、従って次数 n の任意の順列は表の索引(タップ n)によって単に識別する ことができることが分かる。このインデックスは、順列のアトミック表現と呼ばれます。モナド動詞 A は原子表現を計算します。例えば、注文 3 置換、例えば所与 2 1 0、次に A. 2 1 0 は、テーブルのインデックスを生成(tap 3) 。

```
A. 2 1 0    5 { tap 3
5           2 1 0
```

二項動詞 A は、置換のアトミック表現を適用する。

```
2 1 0 { 'PQR'    5 A. 'PQR'
RQP              RQP
```

A. の使用例を以下に示します。何かのすべての順列を実行するプロセス(単語のアナグラムを検索するなど)は非常に時間がかかることがあります。それゆえ、一度に 100 個ずつ実行することが望ましいかもしれません。
ここには、限られた数の置換を見つける動詞があります。引数は boxed リストで、置換されるベクトル、開始順列番号(つまり原子インデックス)、それに続く検出される permuation のカウントが続きます。

```
   LPerms =: 3 : 0
'arg start count' =. y
(start + i. count) A. " 0 1 arg
)
```

```
LPerms 'abcde'; 0; 4
abcde
abced
abdce
abdec

LPerms 'abcde'; 4; 4
abecd
abedc
acbde
acbed
```

16.2 Sorting
ソート

内蔵のモナド、`/:` (スラッシュコロン、「グレードアップ」と呼ばれます)があります。リストのための `L`、式は`(/: L)`へのインデックスの集合を与える `L`、及びこれらの指標は、置換ベクターです。

`L =: 'barn'`	`/: L`
barn	`1 0 3 2`

これらのインデックスは、`L` の項目を 昇順に選択します。すなわち、式`((/: L){L)` は `L` の項目を順番に出力する。

`L`	`/: L`	`(/: L) { L`
barn	`1 0 3 2`	abnr

降順にソートするには、モナド`¥ :(`バックスラッシュコロン、「グレードダウン」)を使用できます。

`L`	`(\: L) { L`
barn	rnba

以来 `L` は、文字のリストで、その項目がアルファベット順にソートされています。数値リストまたはボックスリストは適切にソートされます。

`N =: 3 1 4 5`	`(/: N){N`
`3 1 4 5`	`1 3 4 5`

`B =: 'pooh';'bah';10;5`	`(/: B) { B`

`+----+---+--+-+`
`|pooh|bah|10|5|`
`+----+---+--+-+`
	`+-+--+---+----+`
	`|5|10|bah|pooh|`
	`+-+--+---+----+`

テーブルの行をソートすることを検討してください。次に、3 行のテーブルの例を示します。

```
T =: (". ;. _2) 0 : 0
'WA' ;'Mozart';  1756
'JS' ;'Bach'  ;  1685
'CPE';'Bach'  ;  1714
)
```

表の行を第 2 列(第 3 列)に示された生年月日の順にソートすることを目指すものとします。
列 2 には、表がソートされるキーが含まれているとします。
私たちは動詞 `2&(1)`でキーを抽出し、 `/:` をキーに適用して順列ベクトルを生成し、次にテーブルを置換します。

`T`	`keys =: 2&{"1 T`	`(/: keys) { T`

```
+---+------+----+
|WA |Mozart|1756|
+---+------+----+
|JS |Bach  |1685|
+---+------+----+
|CPE|Bach  |1714|
+---+------+----+
```

```
+----+----+----+
|1756|1685|1714|
+----+----+----+
```

```
+---+------+----+
|JS |Bach  |1685|
+---+------+----+
|CPE|Bach  |1714|
+---+------+----+
|WA |Mozart|1756|
+---+------+----+
```

式(/: keys {T)は、/: 、(Sort と呼ばれる)の二項の場合を使用して、(T /: keys)

```
(/: keys) { T
```

```
+---+------+----+
|JS |Bach  |1685|
+---+------+----+
|CPE|Bach  |1714|
+---+------+----+
|WA |Mozart|1756|
+---+------+----+
```

```
T /: keys
```

```
+---+------+----+
|JS |Bach  |1685|
+---+------+----+
|CPE|Bach  |1714|
+---+------+----+
|WA |Mozart|1756|
+---+------+----+
```

¥: の二項の場合も同様で、ソートとも呼ばれます。
ここで、姓で 2 つの列をソートし、次にイニシャルでソートする必要があるとします。キーは列 1、列 0 です。

```
keys =: 1 0 & { " 1 T
```

```
+------+---+
|Mozart|WA |
+------+---+
|Bach  |JS |
+------+---+
|Bach  |CPE|
+------+---+
```

```
T /: keys
```

```
+---+------+----+
|CPE|Bach  |1714|
+---+------+----+
|JS |Bach  |1685|
+---+------+----+
|WA |Mozart|1756|
+---+------+----+
```

これらの例は、キーが表であり、/: 動詞が表の行を順番に並べる順列ベクトルを生成することを示しています。このような場合、表の最初の列が最も重要であり、次に 2 番目の列が続きます。

16.2.1 Predefined Collating Sequences
事前定義された照合順序

文字は「アルファベット順」、数字は「数字順」、ボックスは明確な順番にソートされます。特定の型のすべての可能なキーをソートする順序は、照合順序(その型のキー用)と呼ばれます。3 つの定義済み照合順序があります。文字の照合シーケンスは ASCII 文字セットです。ビルトインの J 名詞 a。「アルファベット順」のすべての 256 文字の値を返します。大文字は小文字の前に来ることに注意してください。

```
   65 66 67 97 98 99 {a。
ABCabc
```

数値的な議論では、複素数は実数部と虚数部で順序付けられます。

```
n =: 0 1 _1 2j1 1j2 1j1
```

```
n /: n
```

```
0 1 _1 2j1 1j2 1j1
```

```
_1 0 1 1j1 1j2 2j1
```

ボックス配列の場合、順序は各ボックスの内容によって決まります。この優先順位は、まずタイプ別に、ボックス配列より前の文字配列に先行する空配列の前にある数値配列を使用します。

```
k =: (<'abc'); 'pqr'; 4; ''; 3
```

```
+ ----- + --- + - ++ - +
| + --- + | pqr | 4 || 3 |
|| abc || | || |
| + --- + | | || |
+ ----- + --- + - ++ - +
```

```
k /: k
```

```
+ - + - ++ --- + ----- +
| 3 | 4 || pqr | + --- + |
| | || || abc ||
| | || | + --- + |
+ - + - ++ --- + ----- +
```

同じタイプの配列内では、低ランクは高ランクに先行します。

```
m = 2 4; 3; (1 1 $ 1)
```

```
+ --- + - + - +
| 2 4 | 3 | 1 |
+ --- + - + - +
```

```
m /: m
```

```
+ - + --- + - +
| | 3 | 2 4 | 1 |
+ - + --- + - +
```

同じタイプとランクの配列内で、実際には配列はラベリングされ、要素ごとに比較されます。この場合、`1 2` は、より優先される `1〜3`(ため、`2 <3`)、および `3 3` よりも優先される `3 3 3`(ので、`3 3` がより短い `3 3 3`)。2 つの配列が同じ場合は、前の配列が優先されます(つまり、元の順序は乱されません)。

```
a =: 2 3 $ 1 2 3 4 5 6
b =: 3 2 $ 1 2 5 6 3 4
c =: 1 3 $ 1 2 3
d =: 1 3 $ 1 1 3
```

```
u =: a; b; c
```

```
+ ----- + --- + ----- +
| 1 2 3 | 1 2 | 1 2 3 |
| 4 5 6 | 5 6 | |
| | 3 4 | |
+ ----- + --- + ----- +
```

```
u /: u
```

```
+ --- + ----- + ----- +
| 1 2 | 1 2 3 | 1 2 3 |
| 5 6 | | 4 5 6 |
| 3 4 | | |
+ --- + ----- + ----- +
```

```
w =: a; b; c; d
```

```
+ ----- + --- + ----- + ----- +
| 1 2 3 | 1 2 | 1 2 3 | 1 1 3 |
| 4 5 6 | 5 6 | | |
| | 3 4 | | |
+ ----- + --- + ----- + ----- +
```

```
w /: w
```

```
+ ----- + --- + ----- + ----- +
| 1 1 3 | 1 2 | 1 2 3 | 1 2 3 |
| | 5 6 | | 4 5 6 |
| | 3 4 | | |
+ ----- + --- + ----- + ----- +
```

16.2.2 User-Defined Collating Sequences
ユーザー定義の照合順序

キーはデータから計算されます。キーの計算方法を選択することによって、照合シーケンスを選択することができます。
たとえば、数値のリストを絶対値の昇順にソートするとします。適したキー・コンピューティング機能は、その後、「マグニチュード」機能になります| 。

x =: 2 1 _3	キー=: | バツ	x /: keys
2 1 _3	2 1 3	1 2 _3

16.3 Transpositions
移調

モナド動詞|：は、行列を転置します。つまり、第 1 軸と第 2 軸を交換します。

M =: 2 3 $ 'abcdef'	|: M
abc def	広告 は cf

より一般的には、|は n 次元配列の軸の順序を逆にします。

N =: 2 2 2 $ 'abcdefgh'	|: N
ab cd ef gh	ae cg bf dh

二項転置は、左引数として与えられた(フルまたは省略された)置換ベクトルに従って軸を置換する。3 次元配列の場合、6 つの可能な順列があり、第 1 のものは同一性順列

N	0 1 2 |: N	0 2 1 |: N	1 0 2 |: N
ab cd ef gh	ab cd ef gh	ac bd eg fh	ab ef cd gh

1 2 0 |: N	2 0 1 |: N	2 1 0 |: N
ae bf	AC 例えば	ae cg

cg dh	、BD FH	bf dh

箱入りの短縮引数を与えることができます。2 つ以上のボックス化された軸番号が一緒に実行されて 1 つの軸を形成します。二次元では、これは対角線をとることに相当します。

K =: i。3 3	(<0 1)|: K
0 1 2 3 4 5 6 7 8	0 4 8

16.4 Reversing, Rotating and Shifting
逆転、回転、シフト

16.4.1 Reversing
逆転

単項|。その議論の項目の順序を逆にする。

y	|。y	M	|。M
アブデス	エドバ	abc def	def abc

「項目を反転する」とは、最初の軸に沿って反転することを意味します。他の軸に沿った逆転は、階数連結(")で行うことができます。

N	|。N	|。 "1 N	|。"2 N
ab cd ef gh	ef gh ab cd	BA の DC FE HG	cd ab gh ef

16.4.2 Rotating
回転

ダイアディック| 引数 x によって与えられた量だけ y の項目を回転させます。x の正の値は左に回転します。

y	1 |。y	_1 |。y
アブデス	bcdea	eabcd

x の 連続する数値は、連続する軸に沿って y を回転させます。

M	1 2 |。M	N	1 2 |。N
abc def	フェード キャブ	ab cd ef gh	ef gh ab cd

16.4.3 Shifting
シフト

巡回回転によって持ち込まれるアイテムは、代わりに詰め物アイテムと置き換えることができる。移動動詞が書かれています(|。 ! 。f)。ここで f は塗りつぶし項目です。

```
   ash   =: |. !. '*'    NB. alphabetic shift
   nsh   =: |. !. 0      NB. numeric shift
```

y	_2 灰	z =: 2 3 4	_1 nsh z
アブデス	** abc	2 3 4	0 2 3

☆第 16 章終了。