Learning J

Roger Stokes

By Jsoftware.com

revised June 2015

J を学ぶ

日本語翻訳版

スマホ版 V.1.3

2018-05-19

Japla J 言語研究会翻訳

By Rentaro Toribe

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。

著作権©ロジャーストークス 2015
この資料は、承認が行われていれば、自由に複製することができます。

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

巻末の、J 言語アプリ
ダウンロード及びインストールの
“ご案内”をご覧ください

☆☆日本語版製作にあたって☆☆

本日本語版は、英語版では理解が困難な自分自身のために作成しました。したがって、翻訳品質に多くの問題点があることは承知のことであります。

説明が理解に苦しむ場合は、J の計算を実際に実行し、その計算の流れや出力される結果を確認して、自分のものにして欲しい。また、必要に応じて原著の「Learning J」を参照して下さい。

本日本語版は、J 言語に興味を持ちながらなかなか手が出せない、初心者のための入門書です。

目次

著者：Roger Stokes 氏の言葉 3
この本について .....
謝辞 .....
第 1 章：基本 [Basics] 19
1.1 インタラクティブな使用 [Interactive Use] 19
1.2 算術 [Arithmetic] 19
1.3 用語：関数、引数、値 [Terminology: Function…] 20
1.4 リスト値 [List Values] 20
1.5 括弧 [Parentheses] 21
1.6 変数と代入 [Variables and Assignments] 22
1.7 用語集：モナドとダイアド [Terminology: Monads and Dyads] ... 23
1.8 より多くの組み込み関数 [More Built-In Functions] 24
1.9 サイドバイサイドディスプレイ [Side By Side Displays] 26
1.10 コメント [Comments] 27
1.11 組み込み関数の命名規則 [Naming Scheme for Built-In Functions] 27
第 2 章：リストと表 [Lists and Tables] 30
2.1 テーブル [Tables] 30
2.2 配列 [Arrays] 31
2.3 用語：ランクと形状 [Terminology: Rank and Shape] 33
2.4 文字の配列 [Arrays of Characters] 33
2.5 配列のいくつかの関数 [Some Functions for Arrays] 34
2.5.1 結合 [Joining] .....
2.5.2 項目 [Items] .....
2.5.3 選択 [Selecting] .....
2.5.4 平等とマッチング [Equality and Matching] .....
2.6 箱の配列 [Arrays of Boxes] 38
2.6.1 リンク [Linking] .....
2.6.2 ボクシングとアンボクシング [Boxing and Unboxing] .....
2.7 まとめ [Summary] 41
第 3 章：関数の定義 [Defining Functions] 42
3.1 名前の変更 [Renaming] 42
3.2 挿入 [Inserting] 42
3.3 用語：動詞、演算子、副詞 [Verbs, Operators and Adverbs] 43
3.4 交換 [Commuting] 43
3.5 ボンディング [Bonding] 44
3.6 用語：共用語と名詞 [Terminology: Conjunctions and Nouns] ... 45
3.7 関数の構成 [Composition of Functions] 46
3.8 動詞列 [Trains of Verbs] 47
3.8.1 フック [Hooks] .....
3.8.2 フォーク [Forks] .....
数字のリスト L の算術平均は、L を L の項目数で割った合計で与えられます (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) ....

3.9 ものをまとめる [Putting Things Together] 50
第 4 章：スクリプトと明示的な関数 [Scripts and Explicit Functions] 54
4.1 テキスト [Text] 54
4.2 手続きのためのスクリプト [Scripts for Procedures] 54
4.3 明示的に定義された関数 [Explicitly-Defined Functions] 56
4.3.1 見出し [Heading] .....
4.3.2 意味 [Meaning] .....
4.3.3 引数変数 [Argument Variable(s)] .....
4.3.4 ローカル変数 [Local Variables] .....
4.3.5 二項動詞 [Dyadic Verbs] .....
4.3.6 ワンライナー [One-Liners] .....
4.3.7 制御構造 [Control Structures] .....
4.4 暗黙と明示的比較 [Tacit and Explicit Compared] 59
4.5 値としての機能 [Functions as Values] 60
4.6 スクリプトファイル [Script Files] 61
第 5 章：配列の作成 [Building Arrays] 63
5.1 形成リストによる配列の構築 [Building Arrays by Shaping Lists] 63
5.1.1 レビュー [Review] .....
5.1.2 空の配列 [Empty Arrays] .....
5.1.3 スカラの構築 [Building a Scalar] .....
5.1.4 より一般的な形状 [Shape More Generally] .....
5.2 追加、End-to-End への参加 [Appending, or Joining End-to-End] 65
5.2.1 同じランクにする [Bringing To Same Rank] .....
5.2.2 長さへのパディング [Bringing To Same Rank] .....
5.2.3 スカラーのレプリケーション [Replicating Scalars] .....
5.3 ステッチング、サイド・バイ・サイド結合 [Stitching,Joining Side-to-Side] 67
5.4 ラミネート、フェイス等結合動詞 [Laminating, or Joining Face-to-Face] 67
5.5 リンク [Linking] 67
5.6 配列の作成 [Unbuilding Arrays] 68
5.6.1 レイジング [Razing] .....
5.6.2 ラベリング [Ravelling] .....
5.6.3 ラベリング項目 [Ravelling Items] .....
5.6.4 アイテム化 [Itemizing] .....
5.7 大小の配列 [Arrays Large and Small] 69
第 6 章：インデックス作成 [Indexing] 72
6.1 選択 [Selecting] 72
6.1.1 選択の共通パターン [Common Patterns of Selection] .....
6.1.2 テイク、ドロップ等 [Take, Drop, Head, Behead, Tail, Curtail] .....
6.2 選択の一般的な扱い [General Treatment of Selection] 75
6.2.1 独立した選択 [Independent Selections] .....
6.2.2 索引の形状 [Shape of Index] .....

6.2.3 スカラー [Scalars] .....
6.2.4 1軸の選択 [Selections on One Axis] .....
6.2.5 物の除外 [Excluding Things].....
6.2.6 簡略化 [Simplifications] .....
6.2.7 結果の形状 [Shape of the Result] .....
6.3 配列の修正(または更新) [Amending (or Updating) Arrays] 79
6.3.1 インデックスによる修正 [Amending with an Index].....
6.3.2 動詞による修正 [Amending with a Verb] .....
6.3.3 線形指数 [Linear Indices].....
6.4 配列の項目を併合する [Merging Together the Items of an Array]
..... 83
第 7 章：ランク [Ranks]..... 84
7.1 ランク結合 [The Rank Conjunction] 84
7.1.1 モナド動詞 [Monadic Verbs] .....
7.1.2 二項動詞 [Dyadic Verbs].....
7.2 組み込みランク [Intrinsic Ranks] 85
7.3 フレーム [Frames] 87
7.3.1 契約 [Agreement] .....
7.4 結果の再構成 [Reassembly of Results] 90
7.5 ランク接続詞の詳細 [More on the Rank Conjunction] 91
7.5.1 相対セルランク [Relative Cell Rank].....
7.5.2 ユーザー定義の動詞 [User-Defined Verbs] .....
第 8 章：動詞の作成 [Composing Verbs] 95
8.1 モナドとモナドの構成 [Composition of Monad and Monad]..... 95
8.2 構成：モナドとダイアド [Composition:Monad And Dyad] 95
8.3 構成：ダイアドとモナド[Composition: Dyad And Monad]..... 96
8.4 両親媒性組成物 [Ambivalent Compositions] 96
8.5 合成の詳細 [More on Composition: Monad Tracking Monad] 97
8.6 合成 [Composition: Monad Tracking Dyad] 98
8.7 構成 [Composition: Dyad Tracking Monad] 99
8.8 アンビバレンス [Ambivalence Again] 100
8.9 要約 [Summary] 100
8.10 逆 [Inverses] 101
8.11 構成：動詞の動詞 [Composition: Verb Under Verb]..... 101
第 9 章：動詞の列 [Trains of Verbs] 103
9.1 単項フックとフォーク [Review: Monadic Hooks and Forks].... 103
9.2 両項フック [Dyadic Hooks] 103
9.3 両項フォーク [Dyadic Forks] 104
9.4 レビュー [Review] 104
9.5 逐次演算の延長 [Longer Trains] 104
9.6 アイデンティティ関数 [Identity Functions] 106
9.6.1 例：略語としてのフック [Example: Hook as Abbreviation] .....
9.6.2 例：左フック [Example: Left Hook] .....
9.6.3 例：二項 [Example:Dyad].....
9.7 キャップドフォーク [The Capped Fork] 109

9.8 定数関数 [Constant Functions] 110
9.9 ランク結合·定数関数 [Constant Functions with the Rank Conjunction] 110
9.9.1 特別なケース [A Special Case] .....
第 10 章：条件付きフォームと其の他[Conditional and Other Forms]... 113
10.1 条件付きフォーム [Conditional Forms] 113
10.1.1 3 つのケースを持つ例 [Example with 3 Cases].....
10.2 再帰 [Recursion] 116
10.2.1 アッカーマンの機能 [Ackermann's Function] .....
10.3 反復 [Iteration] 117
10.3.1 べき乗接続詞 [The Power Conjunction].....
10.3.2 変更がないまで繰り返す [Iterating Until No Change] .....
10.3.3 While の繰り返し [Iterating While].....
10.3.4 二項動詞の反復 [Iterating A Dyadic Verb] .....
10.4 暗黙の動詞を明示的に生成する [Generating Tacit Verbs from Explicit]e 120
第 11 章：暗黙の動詞の結論 [Tacit Verbs Concluded]..... 122
11.1 疑わしい場合は括弧で囲む [If In Doubt, Parenthesize] 122
11.2 名詞の名前が評価される [Names of Nouns Are Evaluated] 123
11.3 動詞の名前が評価されない [Names of Verb Are Not Evaluated] 124
11.4 未知数は動詞です [Unknowns are Verbs] 124
11.5 パラメトリック関数 [Parametric Functions] 126
第 12 章：明示的な動詞 [Explicit Verbs] 129
12.1 明示的定義結合 [The Explicit Definition Conjunction] 129
12.1.1 型 [Type] .....
12.1.2 型のニーモニック [Mnemonics for Types] .....
12.1.3 ボディスタイル [Body Styles].....
12.1.4 二価動詞 [Ambivalent Verbs].....
12.2 割り当て [Assignments]..... 131
12.2.1 ローカル変数とグローバル変数 [Local and Global Variables] ...
12.2.2 ローカル関数 [Local Functions].....
12.2.3 複数および間接割り当て [Multiple and Indirect Assignments] .....
12.2.4 引数のアンパック [Unpacking the Arguments].....
12.3 制御構造 [Control Structures]..... 136
12.3.1 レビュー [Review] .....
12.3.2 レイアウト [Layout] .....
12.3.3 式と制御構造 [Expressions versus Control Structures] .....
12.3.4 ブロック [Blocks] .....
12.3.5 if のバリアント [Variants of if.] .....
12.3.6 select. 制御構造 [The select. Control Structure] .....
12.3.7 while., whilst. 制御 [while., whilst. Control Structures] .....
12.3.8 for. 制御文 [for. Control Word] .....
12.3.9 return 制御文 [The return. Control Word].....

12.3.10 その他の制御構造 [Other Control Structures] .....
第 13 章：明示的演算子 [Explicit Operators] 145
13.1 暗黙の動詞を生成する演算子 [Operators Generating Tacit Verbs] 145
13.1.1 複数行の本体 [Multiline Bodies] .....
13.2 古いものからの新しい定義 [New Definitions from Old] 149
13.3 明示的な動詞を生成する演算子 [Operators Generating Explicit Verbs] 150
13.3.1 副詞を生成する母音 [Adverb Generating Monad] .....
13.3.2 明示的ダイアドを生成する副詞 [Adverb Generating Explicit Dyad] .....
13.3.3 明示 Monad を生成結合 [Conjunction Generating Explicit Monad] .....
13.3.4 明示的ダイアドの生成 [Generating a Explicit Dyad] .....
13.3.5 引数 - 変数の代替名 [Alternative Names for Argument-Variables] .....
13.3.6 レビュー [Review] .....
13.3.7 ボディの実行(または実行しない) [Executing the Body (Or Not)] .....
13.4 名詞を生成する演算子 [Operators Generating Nouns] 157
13.5 名詞または動詞の生成 [Generating Noun or Verb] 158
13.6 演算子を生成する演算子 [Operators Generating Operators] .. 159
第 14 章：動名詞 [Gerunds] 160
14.1 動名詞を作る: "結び"結合詞 [Making Gerunds: The Tie Conjunction] 160
14.2 動名詞(Gerund)からの動詞の回復 [Recovering the Verbs from a Gerund] 161
14.3 演算子引数･動名詞 [Gerunds As Arguments to Built-In Operators] 162
14.3.1 副詞の動名詞 [Gerund as Argument to APPEND Adverb] .....
14.3.2 議論結合詞のの動名詞 [Gerund as Argument to Agenda Conjunction] .....
14.3.3 挿入の引数としての動名詞 [Gerund as Argument to Insert] .....
14.3.4 べき乗接続の動名詞 [Gerund as argument to POWER conjunction] .....
14.3.5 訂正の引数としての動名詞 [Gerund as Argument to Amend] .....
14.4 演算子･引数の動名詞 [Gerunds as Arguments to User-Defined Operators] 168
14.4.1 Abelson と Sussman[The Abelson and Sussman Accumulator] ....
第 15 章：暗黙の演算子[Tacit Operators] 170
15.1 はじめに[Introduction] 170
15.2 結合詞からの副詞 [Adverbs from Conjunctions] 170
15.3 副詞の構成 [Compositions of Adverbs] 171
15.3.1 例：累計と積 [Example: Cumulative Sums and Products] .....
15.3.2 逐次演算列の生成 [Generating Trains] .....

15.3.3 書き換え[Rewriting] .....
第 16 章：再編成[Rearrangements]..... 175
16.1 順列[Permutations] 175
16.1.1 省略された順列 [Abbreviated Permutations] .....
16.1.2 逆順並べ替え[Inverse Permutation] .....
16.1.3 順列の原子表現[Atomic Representations of Permutations] .....
16.2 ソート[Sorting] 178
16.2.1 事前定義された照合順序[Predefined Collating Sequences] .....
16.2.2 ユーザー定義の照合順序[User-Defined Collating Sequences]....
16.3 転置[Transpositions] 183
16.4 逆転、回転、シフト [Reversing, Rotating and Shifting].... 184
16.4.1 逆転 [Reversing] .....
16.4.2 回転 [Rotating].....
16.4.3 シフト [Shifting] .....
第 17 章： アプリケーションのパターン [Patterns of Application] ... 186
17.1 スキャニング [Scanning]..... 186
17.1.1 プレフィックススキャン [Prefix Scanning].....
17.1.2 インフィックススキャン [Infix Scanning].....
17.1.3 サフィックススキャン[Suffix Scanning] .....
17.1.4 アウトライン [Outfix] .....
17.2 切断 [Cutting] 188
17.2.1 逆転 [Reversing] .....
17.2.2 ブロッキング [Blocking] .....
17.2.3 フレッチング [Fretting] .....
17.2.4 句読点 [Punctuation] .....
17.2.5 ワードフォーメーション [Word Formation].....
17.2.6 ファイル内の行 [Lines in Files].....
17.2.7 タイリング [Tiling] .....
第 18 章：セット、クラス [Sets, Classes and Relations]..... 196
18.1 セット [Sets] 196
18.1.1 メンバー [Membership] .....
ビルトインの動詞 e. があります。 （小文字の”e.”「メンバー」と呼ばれます）。 式 ″x e. y″は、x が y の任意の項目と一致するかどうか、つまり x がリスト y のメンバーであるかどうかをテストします。 例えば： .....
18.1.2 少ない [Less] .....
18.1.3 ナブ(~) [Nub]..... 197
18.1.4 ナブシーブ(~:) [Nub Sieve] .....
18.1.5 セットの関数 [Functions for Sets] .....
18.2 テーブル副詞 [The Table Adverb] 198
18.3 テーブルの装飾 [Table Decorations] 200
18.4 クラス [Classes] 201
18.4.1 自己分類 [Self-Classify].....
18.4.2 分類スキーム [Classification Schemes].....
18.4.3 キー副詞 [The Key Adverb].....
18.4.4 レターカウントの再訪.....
18.5 関係 [Relations] 206

18.5.1 関係の参加 [Join of Relations] .....
18.5.2 シンボルとは [What are Symbols?] .....
18.5.3 測定値を比較 [Measurements Compared] .....
18.5.4 シンボルの保存と復元 [Saving and Restoring the Symbol Table] .....
第 19 章：数字 [Numbers] 215
19.1 異なる 6 種類の数字 [Numbers of Six Different Kinds] 215
19.1.1 ブール値 [Booleans] .....
19.1.2 整数 [Integers] .....
19.1.3 整数のビット論理関数 [Bitwise Logical Functions on Integers] .....
19.1.4 浮動小数点数 [Floating-Point Numbers] .....
19.1.5 科学的記法 [Scientific Notation] .....
19.1.6 浮動小数点数の比較 [Comparison of Floating-Point Numbers] .....
19.1.7 複素数 [Complex Numbers] .....
19.1.8 拡張整数 [Extended Integers] .....
19.1.9 合理的な番号 [Rational Numbers] .....
19.1.10 型変換 [Type Conversion] .....
19.2 特別な番号 [Special Numbers] 225
19.2.1 "無限大" [Infinity] .....
19.2.2「不確定」の数値 ["Indeterminate" Numbers] .....
19.3 番号ベース [Number Bases] 226
19.4 数値の表記 [Notations for Numerals] 227
19.4.1 表記法の組み合わせ [Combining the Notations] .....
19.5.1 "フォーマット"の動詞 [The "Format" Verb] .....
19.6 乱数 [Random Numbers] 232
19.6.1 ロール [Roll] .....
19.6.2 一様分布 [Uniform Distribution] .....
19.6.3 その他の分布 [Other Distributions] .....
19.6.4 カードを配る [Deal] .....
第 20 章: スカラー数値関数 [Scalar Numerical Functions] 234
20.1 プラスおよびコンジュゲート [Plus and Conjugate] 234
20.2 マイナスと否定 [Minus and Negate] 234
20.3 インクリメントとデクリメント [Increment and Decrement] 235
20.4 時間と署名 [Times and Signum] 235
20.5 分割と相互 [Times and Signum] 235
20.6 ダブルとハーフ [Double and Halve] 236
20.7 床と天井 [Floor and Ceiling] 236
20.8 べき乗と指数関数 [Power and Exponential] 237
20.9 二乗 [Square] 238
20.10 平方根 [Square Root] 238
20.11 ルート [Root] 238
20.12 対数と自然対数 [Logarithm and Natural Logarithm] 238
20.13 階乗 [Factorial and outof] 239

20.14 絶対値と剰余 [Magnitude and Residue] 239
20.15 最大公約数および最小公倍数 [GCD and LCM] 240
20.16 円周倍率 [Pi Times] 240
20.17 三角関数とその他の関数 [Trigonometric and Other Functions] 241
20.18 ピタゴラスの関数 [Pythagorean Functions]..... 242
第 21 章: 因子と多項式 [Factors and Polynomials] 244
21.1 プライムとファクター [Primes and Factors] 244
21.2 多項式 [Polynomials] 245
21.2.1 係数 [Coefficients] .....
21.2.2 ルーツ:多項式の根 [Roots] .....
21.2.3 乗数とルーツ [Multiplier and Roots] .....
21.2.4 多項式 [Multinomials] .....
第 22 章: ベクトルと行列 [Vectors and Matrices] 250
22.1 Dot . 結合詞 [The Dot Product Conjunction] 250
22.2 ベクトルのスカラ積 [Scalar Product of Vectors]..... 250
22.3 行列積 [Matrix Product] 251
22.4 一般化 [Generalizations] 252
22.4.1 さまざまな動詞 [Various Verbs] .....
22.4.2 シンボリック演算 [Symbolic Arithmetic] .....
22.4.3 行列積 [Matrix Product in More than 2 Dimensions] .....
22.4.4 ドットとの比較: [Dot Compared With @:] .....
22.5 決定子 [Determinant] 255
22.5.1 特異行列 [Singular Matrices].....
22.6 行列除算 [Matrix Divide]..... 256
22.6.1 連立方程式 [Simultaneous Equations] .....
22.6.2 複素数-ベクトル [Complex, Rational and Vector Variables] ...
22.6.3 カーブフィッティング [Curve Fitting] .....
22.6.4 上位配列による除算 [Dividing by Higher-Rank Arrays] .....
22.7 アイデンティティ行列 [Identity Matrix] 261
22.8 行列の逆行列 [Matrix Inverse] 261
第 23 章: 微積分 [Calculus]..... 262
23.1 差別化 [Differentiation] 262
23.2 積分 [Integration] 263
23.3 d.のドメイン [The Domain of d.] 263
23.4 D.接合詞 [The Conjunction D.] 264
23.4.1 D. のドメイン [The Domain of D.] .....
23.4.2 D.部分的デリバティブ [Partial Derivatives with D.] .....
23.5 数値積分 [Numerical Integration] 267
第 24 章: 名前とロケール [Names and Locales] 268
24.1 背景 [Background]..... 268
24.2 ロケールとは [What are Locales?] 268
24.3 例 [Example]..... 269
24.3.1 レビュー [Review].....
24.4 現在のロケール [The Current Locale] 270

24.5 z ロケールは特別です [The z Locale Is Special] 272
24.6 式の評価 [Locative and the Evaluation of Expressions] ... 274
24.6.1 割り当て [Assignments] .....
24.6.2 名前の評価 [Evaluating Names] .....
24.6.3 動詞の適用 [Applying Verbs] .....
24.6.4 副詞の適用 [Applying Adverbs] .....
24.7 パス [Paths] 277
24.8 ロケーションとパスの結合 [Combining Locatives and Paths] . 279
24.9 間接的な場所 [Indirect Locatives] 280
24.10 名前の消去 [rasing Names from Locales] 280
第 25 章オブジェクト指向 [Object-Oriented Programming] 282
25.1 背景と用語 [Background and Terminology] 282
25.2 クラスの定義 [Defining a Class] 282
25.2.1 クラスの紹介 [Introducing the Class] .....
25.2.2 メソッドの定義 [Defining the Methods] .....
25.3 新しいオブジェクトを作る [Making New Objects] 284
25.3.1 ダイアディック・カウリング [Dyadic Conew] .....
25.4 クラスとオブジェクトのリスト [Listing the Classes and Objects]
..... 286
25.5 継承 [Inheritance] 289
25.5.1 原則の問題 [A Matter of Principle] .....
25.6 継承されたメソッドの使用 [Using Inherited Methods] 292
25.6.1 最初の質問 [First Question] .....
25.6.2 2 番目の質問 [Second Question] .....
25.6.3 3 番目の質問 [Third Question] .....
25.7 ライブラリ動詞 [Library Verbs] 293
第 26 章: スクリプトファイル [Script Files] 295
26.1 スクリプトの作成 [Creating Scripts] 295
26.2 スクリプトのロード [Loading Scripts] 296
26.3 ロード動詞 [The load Verb] 297
26.4 スクリプトのローカル定義 [Local Definitions in Scripts] .. 297
26.4.1 スクリプト内のローカル動詞 [Local Verbs in Scripts] .....
26.5 ロケールへの読み込み [Loading Into Locales] 299
26.6 繰返の回避方法 [Repeated Loading, and How to Avoid It] .. 301
26.7 ロードステータス [Load Status] 302
26.8 ライブラリスクリプト [Library Scripts] 303
26.8.1 標準ライブラリ [Standard Library] .....
26.8.2 J-APP・ライブラリ [The J Application Library] .....
26.9 ユーザー定義の起動スクリプト [User-Defined Startup Script] 305
第 27 章: 表現と変換 [Representations and Conversions] 307
27.1 クラスと型 [Classes and Types] 307
27.1.1 クラス [Classes] .....
27.1.2 タイプ [Types] .....
27.2 実行 [Execute] 309
27.3 表現 [Representations] 309

27.3.1 名詞の表示[Representations of Nouns for Display] .....
27.3.2 書式動詞のボックスの描画 [Drawing Boxes with the Format Verb] .....
27.3.3 関数の表現 [Representations of Functions] .....
27.3.4 線形表現 [Linear Representation] .....
27.3.5 括弧で囲まれた表現 [Parenthesized Representation] .....
27.3.6 ツリー表現 [Tree Representation] .....
27.3.7 原子表現 [Atomic Representation] .....
27.4 表現関数 [Representation Functions] 314
27.4.1 原子表現 [Atomic Representation] .....
27.4.2 原子表現の逆転 [Inverse of Atomic Representation] .....
27.4.3 表現関数のまとめ [Summary of Representation Functions] .....
27.4.4 再訪の実行 [Execute Revisited] .....
27.4.5 再びの連結詞 [The Tie Conjunction Revisited] .....
27.5 バイナリデータの変換 [Conversions for Binary Data] 320
27.6 ユニコード [Unicode] 322
第 28 章: データファイル [Data Files] 326
28.1 ファイルの読み書き [Reading and Writing Files] 326
28.1.1 組み込み動詞 [Built-in Verbs] .....
28.1.2 ファイルとしての画面とキーボード [Screen and Keyboard As Files] .....
28.1.3 ライブラリ動詞 [Library Verbs] .....
28.2 大きなファイル [Large Files] 329
28.3 データ形式 [Data Formats] 330
28.3.1 J-ファイルのバイナリ表現 [The Binary Representation for J-Files] .....
28.3.2 テキストファイル [Text Files] .....
28.3.3 固定長レコード [Fixed Length Records with Binary Data] .....
28.4 マップされたファイル [Mapped Files] 335
28.4.1 マップ ライブラリスクリプト [Library Script for Mapped Files] .....
28.4.2 jmf ファイルと永続変数 [jmf Files and Persistent Variables] .....
28.4.3 マップ ファイルは固定サイズです [Mapped Files are of Fixed Size] .....
28.4.4 ファイルの指定 [Given Files] .....
28.4.5 マップされた変数は特別です [Mapped Variables Are Special] ...
28.4.6 アンマップ再検索 [Unmapping Revisited] .....
第 29 章エラー処理 [Error Handling] 346
29.1 主張 [Assertions] 346
29.1.1 主張と暗黙のスタイル [Assertions and the Tacit Style] .....
29.1.2 アサーションの有効化と無効化 [Enabling and Disabling Assertions] .....
29.2 失敗後の継続 [Continuing after Failure] 349
29.2.1 ノンストップスクリプト [Nonstop Script] .....

29.2.2 制御構造の試行とキャッチ [Try and Catch Control Structure] .....
29.2.3 有害な結合 [Adverse Conjunction] .....
29.3 延期された実行 [Suspended Execution] 351
29.4 プログラムされたエラー処理 [Programmed Error Handling] ... 353
第 30 章: 疎配列 [Sparse Arrays] 356
30.1 はじめに [Introduction] 356
30.2 疎配列はコンパクト [Sparse Array is Compact] 357
30.3 疎配列の検査 [Inspecting A Sparse Array] 357
30.4 疎配列による計算 [Computing with Sparse Arrays] 359
30.5 疎配列の作成 [Constructing A Sparse Array] 359
30.6 疎軸と稠密(ちゅうみつ)軸 [Sparse and Dense Axes] 361
30.7 疎配列の解体 [Deconstructing a Sparse Array] 362
30.8 疎配列と関係 [Sparse Array From Relation] 363
30.9 疎配列からの関係 365
第 31 章: パフォーマンス [Performance] 367
31.1 測定された時間の測定 [Measuring the Time Taken] 367
31.2 パフォーマンスモニタ [Performance Monitor] 368
31.3 ゴールデンルール: 例 1 [The Golden Rule: Ex. 1] 369
31.4 GREx.2: Conway の「人生」[Golden Rule Ex.2: Conway's "Life"]
..... 370
31.5 GR Ex.3: 関係の参加 [Golden Rule Ex.3: Join of Relations]
..... 372
31.5.1 予選 [Preliminaries] .....
31.5.2 ファーストメソッド [First Method] .....
31.5.3 第 2 の方法: ブール行列 [Second Method: Boolean Matrix].....
31.5.4 第 3 法: 再帰的分割行列 [boolean matrix with recursive splitting] .....
31.6 GR 例 4: マンデルブロセット [Golden Rule Example 4: Mandelbrot Set] 376
31.6.1 スカラ・バージョン [Scalar Version] .....
31.6.2 平面バージョン [Planar Version] .....
31.7 辞書 B のコード [The Special Code of Appendix B of the Dictionary]
..... 380
第 32 章: ツリー [Trees] 382
32.1 はじめに [Introduction] 382
32.2 フェッチ [Fetching] 382
32.3 フェッチの領域 [The Domain of Fetch] 383
32.4「マップ」動詞 [he "Map" Verb] 384
32.5 このツリーの高さは幾つですか? [What is the Height of This Tree?]
..... 384
32.6 レベル [Levels] 386
32.7 フラット接続詞[The Spread Conjunction] 388
32.8 さまざまなパス長のツリー [Trees with Varying Path-lengths]389

32.9 ツリーの変換 [Transforming Trees] 391
32.9.1 ツリーからの関係 [Relation from Tree] .....
32.9.2 関係からのツリー [Tree from Relation] .....
32.9.3 配列からのツリーとツリーからの配列 .....
付録 1: 式の評価 [Evaluating Expressions] 396
A1.1 はじめに [Introduction] 396
A1.2 最初の例 [First Example] 397
A1.3 解析ルール [Parsing Rules] 398
A1.3.1 モナド ルール [Monad Rule] .....
A1.3.2 第 2 モナ ドルール [Second Monad Rule] .....
A1.3.3 ダイアド ルール [Dyad Rule] .....
A1.3.4 副詞 規則 [Adverb Rule] .....
A1.3.5 結合規則 [Conjunction Rule] .....
A1.3.6 トライデントルール [Trident Rule] .....
A1.3.7 Bident Rule [Bident Rule] .....
A1.3.9 括弧のルール [Parenthesis Rule] .....
A1.3.10 転送の例 [Examples of Transfer] .....
付録 2: 収集された用語 415
付録３ 419
J 言語プロセッサ PC Windows 版、Android 版のインストール 419
J 言語プロセッサ Android 版の基本的な使い方 424

第１章：基本［Basics］

1.1 インタラクティブな使用［Interactive Use］

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、次の行に結果が表示されます。

```
   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 算術［Arithmetic］

乗算の記号は * (asterisk）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 % (percent）です。

```
   3 % 4
0.75
```

減算のために、わかりやすい - (minus)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 _ (underscore）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 – (minus)であり、減算と同じ記号です：

```
    - 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 *: (asterisk colon) という記号があります。

```
    *: 4
 16
```

1.3 用語：関数、引数、値 [Terminology: Function…]

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ます。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 リスト値 [List Values]

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、次のように記すことができます：

```
    *: 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。 + (plus)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ます。剰余を計算するビルトイン J 関数は | (vertical bar)、 「剰余(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 括弧 [Parentheses]
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、かっこがない場合、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。 したがって、3 * 2 + 1 の場合、*の右の引数は 2 + 1 です。 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 程の)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 変数と代入 [Variables and Assignments]

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 用語集：モナドとダイアド [Terminology: Monads and Dyads]

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

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、次の表現： "2 3 4 を掛け合わせる"の結果は 24 でなければなりません。この式は J では以下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 は 1 より大きい	(これは明らかに真実です)
2 は 1 と同じである	(これは偽である)
2 は 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。

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

もし x が、たとえば、次のように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 より大きいかを聞く場合は？

```
   * / 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には:

#(「Tally」と呼ばれる）があります：

```
   # x
4
```

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

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、1111 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 コメント [Comments]

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。行末までは式評価されません。例えば

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

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

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：リストと表 [Lists and Tables]

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 テーブル [Tables]

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 配列 [Arrays]

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	2 3 $ 5 6 7	2 2 3 $ 5 6 7 8
1 1 1	5 6 7 5 6 7	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	# 6 7 8
2	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	T =: 2 3 $ 1	$ T
5 6 7	3	1 1 1 1 1 1	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	$ L	# $ L	T	$T	# $ T
5 6 7	3	1	1 1 1 1 1 1	2 3	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。

2 $ 99	0 $ 99	$ 17
99 99		

スカラー（17 と言う）は、長さ 1（e.g. 1 $ 17）のリスト、または 1 行と 1 列のテーブル（e.g. 1 1 $ 17）と同じもの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	3 1	2

6 7		

2.3 用語：ランクと形状 [Terminology: Rank and Shape]
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 3 4 5	2 3	2
Report	0 1 2 3 4 5 6 7 8 9 10 11	2 2 3	3

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 文字の配列 [Arrays of Characters]
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 配列のいくつかの関数 [Some Functions for Arrays]

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 結合 [Joining]

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
Cat dog	Rat pig	Cat Dog Rat pig

「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 項目 [Items]

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
Cat dog	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
Cat dog	Rat pig	Cat Dog Rat pig

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 選択 [Selecting]

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 と呼ばれる") 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 平等とマッチング [Equality and Matching]

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 箱の配列 [Arrays of Boxes]

2.6.1 リンク [Linking]

; (セミコロン、 "リンク"と呼ばれる)組み込み関数があります。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	0 {A
+--------------+--+ |The answer is|42| +--------------+--+	+--------------+ |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	$ A	s =: 1 {A	: # $ s
+--------------+--+ |The answer is|42| +--------------+--+	2	+--+ |42| +--+	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  ;  'baked beans'
   P
+----------+----+-----------+
|18 12 1998|1.99|baked beans|
+----------+----+-----------+
```

「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 ボクシングとアンボクシング [Boxing and Unboxing]

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

```
> b
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。

```
(< 1 1) , (< 2 2) , (< 3 3)
+---+---+---+
|1 1|2 2|3 3|
+---+---+---+
```

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

2.7 まとめ [Summary]

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：関数の定義 [Defining Functions]

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 名前の変更 [Renaming]

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 挿入 [Inserting]

（+ / 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 用語：動詞、演算子、副詞 [Verbs, Operators and Adverbs]

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 交換 [Commuting]

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
```

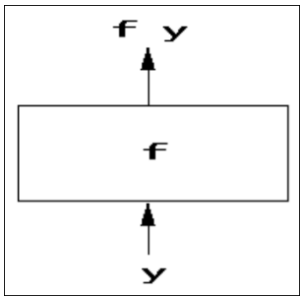
別の例として、残差動詞 | ここで 2 | 7 は従来の記法では「7 mod 2」を意味します。mod 動詞を定義することができます：

```
mod =: | ~
```

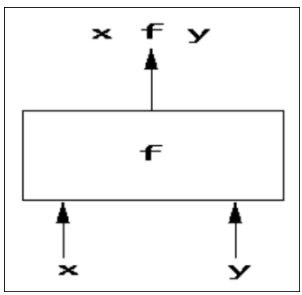
7 mod 2	2 | 7
1	1

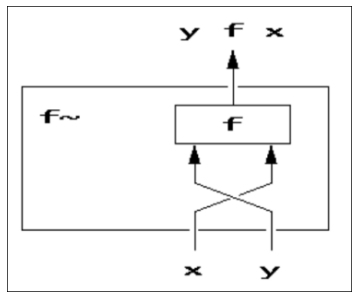
私はいくつかの図を描いてみましょう。まず、 引数 y に適用された関数 f の図で、結果（f y）を生成します。この図では、関数 f は長方形として描かれ、矢印は関数に流入する

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 ボンディング [Bonding]

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とって、2 つの引数のうちの 1 つを選択した値（この場合は 2）に固定してモナドを生成します。 & 演算子は、関数と 1 つの引数の値との間に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
```

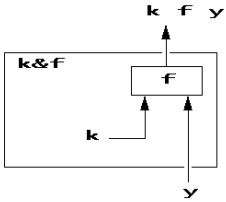
右の議論を修正するのではなく、左を修正することができるので、スキームもあります。

```
(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 用語：共用語と名詞 [Terminology: Conjunctions and Nouns]

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。
&のような 2 つの引数をとる演算子は、2 つの引数を結合しているので、J の "連結"で呼び出されます。 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 つのクラスのうちの 1 つに正確に属します。 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 関数の構成 [Composition of Functions]

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
```

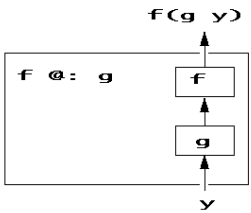
単一の二乗和関数は、和と平方の複合として書くことができます。

```
   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 動詞列 [Trains of Verbs]

「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 フック [Hooks]

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
```

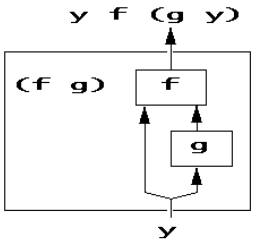
購入価格が、たとえば $50 であれば、次のように表示されます。

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 フォーク [Forks]

数字のリスト L の算術平均は、L を L の項目数で割った合計で与えられます（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

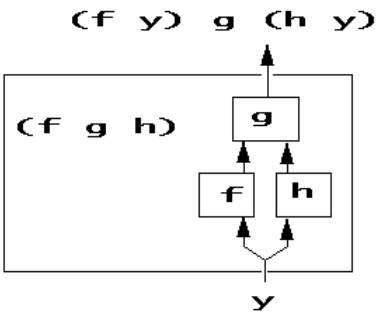
アイテムの数で割った和として平均値を計算する動詞は、三個の動詞の配列のように書くことができる：合計に続いて %、# が続きます。

```
   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 ものをまとめる [Putting Things Together]

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：スクリプトと明示的な関数 [Scripts and Explicit Functions]

「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 テキスト [Text]

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 =: 0 : 0
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 手続きのためのスクリプト [Scripts for Procedures]

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(Fahrenheit)から摂氏(Celsius)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 明示的に定義された関数 [Explicitly-Defined Functions]

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 見出し [Heading]

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 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 引数変数 [Argument Variable(s)]

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 ローカル変数 [Local Variables]

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
)
```

摂氏での t は局所変数、または摂氏ではローカルであると言います。 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実証できます

```
    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 二項動詞 [Dyadic Verbs]

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 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 制御構造 [Control Structures]

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 暗黙と明示的比較 [Tacit and Explicit Compared]

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 値としての機能 [Functions as Values]

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 スクリプトファイル [Script Files]

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：配列の作成 [Building Arrays]

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 形成リストによる配列の構築 [Building Arrays by Shaping Lists]

5.1.1 レビュー [Review]

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	$ A	$ $ A	$ $ $ A
ABC DEF	2 3	2	1

5.1.2 空の配列 [Empty Arrays]

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 スカラの構築 [Building a Scalar]

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 より一般的な形状 [Shape More Generally]

我々は、（x $ y）は、y の項目の x 字型配列を生成すると言った。 つまり、（x $ y）の形は一般に x だけでなく、x の後に y の項目の形が続きます。
y がテーブルの場合、y の項目は行、つまりリストになります。次の例では、Y の項目の形状は Y の行の長さ（4）です。

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

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 追加、End-to-End への参加 [Appending, or Joining End-to-End]

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 同じランクにする [Bringing To Same Rank]
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 DEF	uvw	ABC DEF uvw	ABC DEF uvw	uvw ABC DEF

5.2.2 長さへのパディング [Bringing To Same Rank]
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 スカラーのレプリケーション [Replicating Scalars]
"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 ステッチング、サイド・バイ・サイド結合 [Stitching,Joining Side-to-Side]

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 ラミネート、フェイス等結合動詞 [Laminating, or Joining Face-to-Face]

,: (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 リンク [Linking]

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 配列の作成 [Unbuilding Arrays]

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 レイジング [Razing]

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 ラベリング [Ravelling]

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

   , B
+-+-+-+-+
|1|2|3|4|
+-+-+-+-+

   $ , B
4
```

5.6.3 ラベリング項目 [Ravelling Items]

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
0 1  2
3 4  5

6 7  8
9 10 11

   ,. k
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
uvw

   ,. b
u
v
w
```

5.6.4 アイテム化 [Itemizing]

モナド : は "箇条書き(Itemize)"と呼ばれます。 それは 1 の先頭次元を加えることによって、どの配列からも 1 項目の配列を作成します。

```
   A
ABC
DEF

   ,: A
ABC
DEF

   $ ,: A
1 2 3
```

5.7 大小の配列 [Arrays Large and Small]

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：インデックス作成 [Indexing]

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 選択 [Selecting]

動詞{(左括弧）は "From"と呼ばれます。式(x {y）は、x から与えられた位置に従って y 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 選択の共通パターン [Common Patterns of Selection]

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 def ghi	de gh

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	ghi def

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 def ghi	beh

他の可能性があります：以下を参照してください。

6.1.2 テイク、ドロップ等 [Take, Drop, Head, Behead, Tail, Curtail]

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}）によって選択されます。

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。 最後の n を除くすべてが（-n）}. L によって選択されます。

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

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 独立した選択 [Independent Selections]

インデックス作成の一般的な式は、index {array}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 索引の形状 [Shape of Index]

結果の形状は、インデックスの形状に依存します。

L	index =: 2 2 $ 2 0 3 1	index { L
abcdef	2 0 3 1	ca db

インデックスは、-#L ～（#L）-1 の範囲内になければなりません。

L	#L	_7 { L	6 { L
abcdef	6	error	error

6.2.3 スカラー [Scalars]

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 1 軸の選択 [Selections on One Axis]

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 要素から列 0 2 を選択する。

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

6.2.5 物の除外 [Excluding Things]

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 def	ac

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 簡略化 [Simplifications]

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 J 略語、つまり（a :)（「Ace」と呼ばれる文字 a: ）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 結果の形状 [Shape of the Result]

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	**22 23 25 26**	**2 2 2**	**2 2 2**

19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36	31 32 34 35		

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)	$
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 配列の修正(または更新) [Amending (or Updating) Arrays]

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 インデックスによる修正 [Amending with an Index]

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 動詞による修正 [Amending with a Verb]

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 f が副詞 } の引数として提供される場合、f はダイアドでなければならないが、X または Y は無視されることに注意してください。

6.3.3 線形指数 [Linear Indices]

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、私達は f を再定義する：

```
   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 51 14	13 50 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 配列の項目を併合する [Merging Together the Items of an Array]

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：ランク [Ranks]

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 ランク結合 [The Rank Conjunction]

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
```

それぞれランク 0 の 6 つのセルに分割されているか、ランク 1 の 2 つのセルに分割されているか、またはランク 2 の単一セルとして分割されているとみなすことができる。ランク k のセルは k セルと呼ばれる。

7.1.1 モナド動詞 [Monadic Verbs]

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

	|d|e|f| +-+-+-+		+---+

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 二項動詞 [Dyadic Verbs]

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	Y =: 2 3	X (* " 1 0) Y
0 1 2 3	2 3	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	Y	X (* " 1 1) Y
0 1 2 3	2 3	0 3 4 9

7.2 組み込みランク [Intrinsic Ranks]

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	*: 2 3 4
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 です。

```
   #. 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 フレーム [Frames]

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証明するために、u の最初の（モナド）ランクとして、u から k を見つけることができます。

```
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 契約 [Agreement]

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	0 1	2 3 2	2	0 1 2 3

4 5 6 7 8 9 10 11				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 結果の再構成 [Reassembly of Results]

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 ∴ 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 ランク接続詞の詳細 [More on the Rank Conjunction]
7.5.1 相対セルランク [Relative Cell Rank]
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 y) は、u が y のランクより小さいランク 1 のセル、すなわち y のアイテムに適用されることを意味する。

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 ユーザー定義の動詞 [User-Defined Verbs]

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！) 我々は階乗を直接計算してみる…

```
      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 y 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：動詞の作成 [Composing Verbs]

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 モナドとモナドの構成 [Composition of Monad and Monad]

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 構成：モナドとダイアド [Composition:Monad And Dyad]

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 構成：ダイアドとモナド[Composition: Dyad And Monad]

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 両親媒性組成物 [Ambivalent Compositions]

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 合成の詳細 [More on Composition: Monad Tracking Monad]
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    means    (f @: g) " G   y
```

8.6 合成 [Composition: Monad Tracking Dyad]

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 |	+ --- + | bca |	+ --- + --- + | bca | fde |

`| fde |` `+ --- +`	`| 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 構成 [Composition: Dyad Tracking Monad]
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

対照的に、（f&:g）では、動詞 f はちょうど 1 回適用されて、1　4、9　16 を与えます。これに対して、（f&g）では、第 1 に 1，9、第 2 に 4，16 を与えた f の 2 つの別々のアプリケーションがあります。
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 x からの G セルと y からの対応する G セルの各組み合わせに別々に適用されます。 説明します：

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 アンビバレンス [Ambivalence Again]

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 要約 [Summary]

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 逆 [Inverses]

"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 構成：動詞の動詞 [Composition: Verb Under Verb]

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 &. ）の考え方は、

" 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：動詞の列 [Trains of Verbs]

この章では、第 3 章で始まった動詞列の話題を続けます。trains 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 単項フックとフォーク [Review: Monadic Hooks and Forks]

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)y は(fy)g(hy)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。

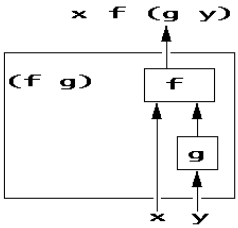
9.2 両項フック [Dyadic Hooks]

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 両項フォーク [Dyadic Forks]

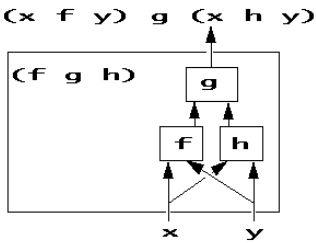
「10 プラスまたはマイナス 2」という表現がリスト 12 8 を意味するとします。フォワード(+、 -)として、x のプラスマイナス y を計算する動詞を書くことができます。

(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 レビュー [Review]

動詞の trains 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 逐次演算の延長 [Longer Trains]

ここでは、逐次演算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、任意の長さの逐次演算をフックとフォークに分析することができる。4 つの動詞の逐次演

算について、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 連逐次演算(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、y の "ノルム"は(y - 平均 y)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
```

長い逐次演算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 trains(- 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 連の逐次演算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 アイデンティティ関数 [Identity Functions]

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

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 例：左フック [Example: Left Hook]

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 trains 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 例：二項 [Example:Dyad]

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 キャップドフォーク [The Capped Fork]

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 train 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 定数関数 [Constant Functions]

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

As well as 0: there are similar functions 1: 2: 3: and so on up to 9: and also the negative values: _9: to _1:

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 trains 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 ランク結合･定数関数 [Constant Functions with the Rank Conjunction]

定数関数: _9 から 9: trains 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 " R   means   (3 : 'k') " R
```

9.9.1 特別なケース [A Special Case]

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：条件付きフォームと其の他[Conditional and Other Forms]

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 trains 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 条件付きフォーム [Conditional Forms]

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 3 つのケースを持つ例 [Example with 3 Cases]

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 @. case
```

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
error	99 100.5

10.2 再帰 [Recursion]

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:  @. empty
   Sum 1 1 2
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 アッカーマンの機能 [Ackermann's Function]

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 反復 [Iteration]

10.3.1 べき乗接続詞 [The Power Conjunction]

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-べき乗"接続詞(^:)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 y はちょうど y であり、次に f ^: 1 y は fy 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

(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 変更がないまで繰り返す [Iterating Until No Change]

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 While の繰り返し [Iterating While]

"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
```

gy 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、関数を考えてみると、

```
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 二項動詞の反復 [Iterating A Dyadic Verb]

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 暗黙の動詞を明示的に生成する [Generating Tacit Verbs from Explicit]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：暗黙の動詞の結論 [Tacit Verbs Concluded]

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 疑わしい場合は括弧で囲む [If In Doubt, Parenthesize]

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 名詞の名前が評価される [Names of Nouns Are Evaluated]

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 動詞の名前が評価されない [Names of Verb Are Not Evaluated]

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 未知数は動詞です [Unknowns are Verbs]

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 0 100
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 パラメトリック関数 [Parametric Functions]

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 : 'lambda * 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	lambda=: 3.0	P x	lambda=: 3.5	P x
0.6	3	0.72	3.5	0.84

さて、lambda 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？

私たちが暗黙の形の P を生成するならば、ラムダは現在 3.5

です。

```
   tP =: 13 : 'lambda * 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：明示的な動詞 [Explicit Verbs]

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 明示的定義結合 [The Explicit Definition Conjunction]

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 型 [Type]

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 型のニーモニック [Mnemonics for Types]

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(簡略記号)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.単項
   dyad          =: 4      NB.2 項
   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 ボディスタイル [Body Styles]

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 二価動詞 [Ambivalent Verbs]

相反する動詞は、モナドとダイアディックの両方(二価)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 : (colon) だけの行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
)
```

log 2.7182818	10 log 100
1	2

12.2 割り当て [Assignments]

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 x と y はローカルなので、y という名前の変数にグローバルな代入を行うのは通常、明示的な動詞ではエラーになります。

```
   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 y 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 ローカル関数 [Local Functions]

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。

12.2.3 複数および間接割り当て [Multiple and Indirect Assignments]

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 引数のアンパック [Unpacking the Arguments]
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 制御構造 [Control Structures]

12.3.1 レビュー [Review]

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 レイアウト [Layout]

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 式と制御構造 [Expressions versus Control Structures]

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 ブロック [Blocks]

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、T が最初の要素が 0 でない配列に評価された場合、「真」と評価されます。

```
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 if のバリアント [Variants of if.]

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 select. 制御構造 [The select. Control Structure]

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す。

```
   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 while., whilst. 制御 [while., whilst. Control Structures]

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. 制御文 [for. Control Word]

パターン

```
for_a. A do. B. end.
```

つまり、配列 A 内の各項目 a に対して、ブロック B を実行します。ここで、a は任意の名前です。 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 1 つのバリエーションは、ブロック B が A の項目がある回数だけ実行されるパターン（for do. B end. ）です。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 return 制御文 [The return. Control Word]

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 その他の制御構造 [Other Control Structures]

第 29 章では、制御構造の try. 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：明示的演算子 [Explicit Operators]

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
2 : body     is a conjunction
```

body 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 暗黙の動詞を生成する演算子 [Operators Generating Tacit Verbs]

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、以下のスキームに従って動詞を生成するために、副詞 A を定義することを目的とすると仮定します。動詞 u の場合：

```
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。整数 u の場合：

```
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: #]	F y
>&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 複数行の本体 [Multiline Bodies]

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 古いものからの新しい定義 [New Definitions from Old]

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ます。

```
   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、その議論のために以下によって小さい副詞、SMOO という |. (rotate) によって便利に提供できます：

```
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 明示的な動詞を生成する演算子 [Operators Generating Explicit Verbs]

私たちは一緒に定義することを目指しているスキームを H

と言う：

```
  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 y 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す。

```
   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 副詞を生成する母音 [Adverb Generating Monad]

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 明示的ダイアドを生成する副詞 [Adverb Generating Explicit Dyad]

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 は、

```
       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、u と言う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 明示 Monad を生成結合 [Conjunction Generating Explicit Monad]

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 明示的ダイアドの生成 [Generating a Explicit Dyad]

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  is to be   4:  '(ux)+(vy)'
```

またはこれと同等に、上記のとおりです。

```
        u H v  is to be   3: 0
                          :
                         (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 引数 - 変数の代替名 [Alternative Names for Argument-Variables]

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 レビュー [Review]

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 uvxy 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 ボディの実行(または実行しない) [Executing the Body (Or Not)]

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 名詞を生成する演算子 [Operators Generating Nouns]

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 f の固定点は、(f p) = p となるような値 p である f をとると、

```
   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 名詞または動詞の生成 [Generating Noun or Verb]

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'
```

だから私たちは以下のように言うことができます：

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

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)     where 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 演算子を生成する演算子 [Operators Generating Operators]

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 は、

```
          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：動名詞 [Gerunds]

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 動名詞を作る："結び"結合詞 [Making Gerunds: 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 x の絶対値を求めるには、 (+x) を計算するか、数値が負の場合は (-x) を計算する。

abs =: + ` - @. (< & 0)	abs _3
+`-@.(<&0)	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、動名詞 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 動名詞(Gerund)からの動詞の回復 [Recovering the Verbs from a Gerund]

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 3trains、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 trains 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	2{H	vb =: (2{H) EV	vb 'o'
+-+-+-+ |f|,|g|	+-+ |g|	g	go

+-+-+-+	+-+		

今我々は gerunds の使用に来る。

14.3 演算子引数･動名詞 [Gerunds As Arguments to Built-In Operators]

動名詞(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 副詞の動名詞 [Gerund as Argument to APPEND Adverb]

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	G2 APPEND 4 5

+-----+-----+ |count|sum | +-----+-----+ |mean |count| +-----+-----+	2 9 4.5 2

14.3.2 議論結合詞のの動名詞 [Gerund as Argument to Agenda Conjunction]

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(記憶)は次のとおりです：

abs =: + ` - @. (< & 0)	abs 6	abs _6
+`-@.(<&0)	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、trains 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(企て)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	((0 2 { G)) EV 4
+-+-+-+ |+|-|%| +-+-+-+	4.25	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 trains の作り方を見ていきます。trainsT 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 trains 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 挿入の引数としての動名詞 [Gerund as Argument to Insert]

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 べき乗接続の動名詞 [Gerund as argument to POWER conjunction]

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
```

引数 y から開始値を計算する動詞 v2 は、y の値が 0 であれば 0 1 を計算する定数関数になります。

```
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 訂正の引数としての動名詞 [Gerund as Argument to Amend]

第六章の「訂正」副詞を思い出してください。式(new index) old) は、index の項目として new を持つ old の修正版を生成します。 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 演算子･引数の動名詞 [Gerunds as Arguments to User-Defined Operators]

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 Abelson と Sussman[The Abelson and Sussman Accumulator]

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 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、およびフィルタの動名詞 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&|) です。

```
      (+ ` *: ` (2&|)) ACC 1 2 3 4
10
```

☆第 14 章終了。

第 15 章：暗黙の演算子[Tacit Operators]

15.1 はじめに[Introduction]

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 結合詞からの副詞 [Adverbs from Conjunctions]

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	< each	z =: < each 'abc'
"0	<"0	+-+-+-+ |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 副詞の構成 [Compositions of Adverbs]

A と B が副詞である場合、二項動詞(Bident)は（A B）は A と B を適用する副詞を表します。スキームは次のとおりです。

```
x (A B)     means (x   A) B
```

15.3.1 例：累計と積 [Example: Cumulative Sums and Productsl

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：

【訳者註】(バックスラッシュ \ の表示フォントは、Araial Unicode MS を使用する。)

```
   < \ '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
```

	+ cum z	* cum z
2 3 4	2 5 9	2 6 24

【訳者註】この項は、バックスラッシュ[\ ]を表示するため、font は[Arial Unicode MS]を使用します。

15.3.2 逐次演算列の生成 [Generating Trains]

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 trains 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 書き換え[Rewriting]

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：再編成[Rearrangements]

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 順列[Permutations]

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'	z =: 4 2 3 1 0 { y
abcde	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	p y	p p y	p p p p p p y
abcde	ecdba	adbce	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	C. 3 1 2 ; 4 0
+ ----- + --- + | 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	4 2 3 1 0 C. y	(3 1 2 ; 4 0) C. y
abcde	ecdba	ecdba

16.1.1 省略された順列 [Abbreviated Permutations]

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 逆順並べ替え[Inverse Permutation]

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 順列の原子表現[Atomic Representations of Permutations]

y が長さ n のベクトルで あれば、完全に！y 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	LPerms 'abcde'; 4; 4
abcde abced abdce abdec	abecd abedc acbde acbed

16.2 ソート[Sorting]
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
+---+------+----+
|WA |Mozart|1756|
+---+------+----+
|JS |Bach  |1685|
+---+------+----+
|CPE|Bach  |1714|
+---+------+----+
```

```
keys =: 2&{"1 T
+----+----+----+
|1756|1685|1714|
+----+----+----+
```

```
(/: keys) { T
+---+------+----+
|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 事前定義された照合順序[Predefined Collating Sequences]
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 ユーザー定義の照合順序[User-Defined Collating Sequences]

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(絶対値)」機能になります| 。

```
x=: 2 1 _3
2 1 _3
```

```
keys =: | x
2 1 3
```

```
x /: keys
1 2 _3
```

16.3 転置[Transpositions]

モナド動詞|：は、行列(Matrix)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'
```

```
abc
def
```

```
|: M
```

```
ad
be
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'
```

```
ab
cd

ef
gh
```

```
|: N
```

```
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
```

```
ab
cd

ef
gh
```

```
0 1 2 |: N
```

```
ab
cd

ef
gh
```

```
0 2 1 |: N
```

```
ac
bd

eg
fh
```

```
1 0 2 |: N
```

```
ab
ef

cd
gh
```

```
1 2 0 |: N
```

```
ae
bf

cg
dh
```

```
2 0 1 |: N
```

```
ac
eg

bd
fh
```

```
2 1 0 |: N
```

```
ae
cg

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 逆転、回転、シフト [Reversing, Rotating and Shifting]

16.4.1 逆転 [Reversing]

単項| 。その議論の項目の順序を逆にする。

y	|. y	M	|. M
abcde	edcba	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 dc fe hg	cd ab gh ef

16.4.2 回転 [Rotating]

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
abcde	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	fde cab	ab cd ef gh	ef gh ab cd

16.4.3 シフト [Shifting]

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 ash y	z =: 2 3 4	_1 nsh z
abcde	**abc	2 3 4	0 2 3

☆第 16 章終了。

第 17 章： アプリケーションのパターン [Patterns of Application]

この章では、配列の選択された要素で構成されたさまざまなパターンで配列に関数を適用する方法について説明します。

17.1 スキャニング [Scanning]

17.1.1 プレフィックススキャン「Prefix Scanning」

式(f \ y)では 、結果は、動詞 f を y の連続した長い先頭セクション(接頭辞)に適用することによって生成されます。ボックス動詞(<)として f を 選択すると、簡単に目に見える結果が得られます。

y =: 'abcde'	< ¥ y
abcde	+-+--+---+----+-----+ |a|ab|abc|abcd|abcde| +-+--+---+----+-----+

数値ベクトルの累積合計を生成することができます。

```
   + / \ 0 1 2 3
0 1 3 6
```

ビットベクトルを走査することにより、様々な効果を生み出すことができる。次の例は、最初の 1 ビットの後にすべてのビットをオンにする「累積 OR」を示しています。

```
   +。/ \ 0 1 0 1 0
0 1 1 1 1 1
```

17.1.2 インフィックススキャン「Infix Scanning」

式に(XF 用の\ Y)動詞 f はの連続切片(「挿入辞」)に適用され、Y、長さの各 X。

z =: 1 4 9 16	2 < ¥ z
1 4 9 16	+---+---+----+ |1 4|4 9|9 16| +---+---+----+

x が負の 場合、セクションはオーバーラップしません。その場合、最後のセクションは全長ではありません。例えば:

z	_3 < ¥ z
1 4 9 16	+-----+--+ |1 4 9|16| +-----+--+

セクション長に 2 を選択し、各セクションに"second-minus-first"という動詞、
つまり({: - {.})を適用することで、連続した項目間の相違を計算できます。

smf =: {: - {.	smf 1 4
{: - {.	3

```
   diff =: 2&(smf \)
```

,. z	,. diff z	,. diff diff z
1 4 9 16	3 5 7	2 2

17.1.3 サフィックススキャン[Suffix Scanning]

式(f \ .y)では、y の 連続して短い後続セクション(「接尾辞」)に f を適用することによって結果が生成されます。

y	< ¥. y
abcde	+-----+----+---+--+-+ |abcde|bcde|cde|de|e| +-----+----+---+--+-+

17.1.4 アウトライン[Outfix]

式(xf \ .y)では 、動詞 f は、連続するセクションが削除された y の全体に適用され、削除されたセクションのそれぞれは長さ x です。x が負の場合、削除されたセクションはオーバーラップしません。この場合、最後に削除されたセクションは完全長ではない可能性があります。

y	2 < ¥. y	_2 < ¥. y
abcde	+---+---+---+---+ |cde|ade|abe|abc| +---+---+---+---+	+---+---+----+ |cde|abe|abcd| +---+---+----+

17.2 切断 [Cutting]

結合詞; (セミコロンドット)を「切り取り」といいます。u が動詞で n が小さい整数ならば、(u; .n)は n で指定されたさまざまなパターンで u を適用する動詞です。n の可能な値は_3 _2 _1 0 1 2 3 です。これらのケースのすべてではなく、一部を見ていきます。

17.2.1 逆転 [Reversing]

式(u; .0 y)では、すべての軸に沿って反転した y に動詞 u が適用されます。次の例では、u を identity-verb([)に設定します。

M =: 3 3 $ 'abcdefghi'	[;. 0 M
abc def ghi	ihg fed cba

17.2.2 ブロッキング [Blocking]

配列を与えると、その中のより小さな部分配列を取り出し、部分配列だけに動詞を適用することができます。
サブアレイは 2 行のテーブルで指定します。最初の行には、サブアレイの最初になるセルのインデックスがあります。2 番目の行にはサブアレイの形があります。
たとえば、元の配列の行 1 の列 1 から始まり、図形 2 2 から始まる部分配列を指定するには、次のように記述します。

```
spec =: 1 1 ,: 2 2
```

次に、指定された部分配列に identity-verb ([)を次のように適用できます。

M	spec	spec [;. 0 M
abc	1 1	ef

def ghi	2 2	hi

一般的な方式は、動詞 u の場合、式(xu; .0 y)は x で指定された y の 部分配列に動詞 u を適用するというものです。指定子 x では、形状の負の値(2 番目の行)は、対応する軸に沿って M の要素を反転させます。例えば：

```
spec =: 1 1 ,: _2 2
```

M	spec	spec [;. 0 M
abc def ghi	1 1 _2 2	hi ef

17.2.3 フレッチング [Fretting]

1 行のテキストを別々の単語に分割することに興味があるとします。次に、テキスト行の例を示します。

```
y =: 'what can be said'
```

現時点では、単語がスペースで終わると見なすと仮定します。(他の可能性もあります)すぐにわかるように、上記の y では、最後の単語'said'にスペースがないので、最初に行うのは最後にスペースを追加することです：

```
y =: y , ' '
```

ここで、u が動詞で、y が空白で終わっていれば、式(u; .2 y)は、y 内の各空白終端単語に動詞 u を別々に適用します。例えば、<、ボックス関数を適用することによって y の単語を識別することができます：

y	< ;. _2 y
what can be said	+----+---+--+----+ |what|can|be|said| +----+---+--+----+

: #動詞を 適用することで、各単語の文字を数えることができます：

y	# ;. _2 y
what can be said	4 3 2 4

; の右の議論のため の_2 の意味。y(空間) 内の最後の文字の出現によって単語が終了されること、さらにその単語に空白が含まれないことです。
より一般的には、リストは、「フレット」の発生によってマークされた「間隔」に分割されていると言います。右引数(N)の、。間隔とフレットを定義する方法を以下のように指定します。4 つのケースがあります。

n = 1	各区間はフレットで始まります。y の最初の項目は、最初の項目と等しい y の 他の項目と同様に 、フレットと見なされます。 間隔にはフレットが含まれます。
n = _1	間隔がフレットを除いて n = 1 の場合と同様である。
n = 2	各区間はフレットで終わります。最後の項目 Y は、フレットであるとみなされる 任意の他の項目であるように Y 最後に等しいです。 間隔にはフレットが含まれます。
n = _2	n = 2 の間隔はフレットを除外することを除いて、。

たとえば、次の 4 つのケースが表示されます。

```
z =: 'abdacd'
```

z	< ;. 1 z	< ;. _1 z	< ;. 2 z	< ;. _2 z
abdacd	+---+---+ |abd|acd| +---+---+	+--+--+ |bd|cd| +--+--+	+---+---+ |abd|acd| +---+---+	+--+--+ |ab|ac| +--+--+

別の例として、数字の表を入力する方法があります。0: 0 に

続いて行ごとにテーブルを入力します

```
      T =: 0: 0
1 2 3
4 5 6
19 20 21
)
```

T は 3 つの埋め込み改行文字を含む文字列で、各行の最後に 1 つあります。

$ T	+/ T = LF
30	3

今の考え方は、T を線に引くことです。各行は、J 式を表す文字列です(たとえば、文字'1 2 3')。このような文字列は、動詞"。 (二重引用符" Do "または" Execute ")を適用することで評価できます。結果は、各行に対して 3 つの数字のリストです。

TABLE =: (". ;. _2) T	$ TABLE
1 2 3 4 5 6 19 20 21	3 3

動詞(". ;. _2)は第 2 章でユーティリティ関数 ArrayMaker として導入されました。

17.2.4 句読点 [Punctuation]

テキストを処理するには、単語をスペースで区切ったり、さまざまな区切り記号で終えると見なすと便利です。私たちのフレットを 4 つの文字のいずれかとして選択したとします。

```
   frets =: ' ?!.'
```

いくつかのテキストが与えられていると、フレットの位置で真であるビットベクトルを計算することができます：

t =: 'How are you?'	b =: t e. frets
How are you?	0 0 0 1 0 0 0 1 0 0 0 1

ここでは、私達は内蔵の動詞 e.(“Member”)を利用する。もし x がリスト y のメンバーであるならば、表現（x e. y）は真に評価する。
今や、bitvector b は、テキスト t を言葉に切り分けるためのフレットを指定するために使用できる：

t	b	b < ;. _2 t
How are you?	0 0 0 1 0 0 0 1 0 0 0 1	+---+---+---+ |How|are|you| +---+---+---+

別の例として、数字のベクトルを、それぞれが昇順になるように間隔で切り取ることを検討してください。つまり、前のものよりも小さい項目は新しい間隔を開始する必要があります。私たちのデータは以下の通りです：

```
data =: 3 1 4 1 5 9
```

次に、ビットベクトルは長さ 2 の挿入部を走査し、各対の項目に> /を適用することによって計算することができる。1 を取得すると、ペアの 2 番目の項目が新しい間隔の開始点になります。すべての最初の項目が 1 であることを確認します。

```
bv =: 1 , 2 >/ \ data
```

data	data ,: bv	bv < ;. 1 data
3 1 4 1 5 9	3 1 4 1 5 9 1 1 0 1 0 0	+-+---+-----+ |3|1 4|1 5 9| +-+---+-----+

17.2.5 ワードフォーメーション [Word Formation]

組み込み関数:(セミコロンコロン、 "ワードフォーメーション"と呼ばれます)があります。J 言語の規則に従って、文字列を J 式として解析し、J 式の別々の構成要素である文字列のボックスリストを生成します。
例えば：

y =: 'z =: (p+q) - 1'	;: y

```
   z =: (p+q) - 1
```

```
+-+--+-+-+-+-+-+-+-+
|z|=:|(|p|+|q|)|-|1|
+-+--+-+-+-+-+-+-+-+
```

17.2.6 ファイル内の行 [Lines in Files]

以下の例では、ファイルを作成して始めましょう。(ファイル処理関数の詳細については、第 26 章を参照してください)。

```
   text =: 0 : 0
What can be said
at all
can be said
clearly.
)

   text (1 !: 2) < 'foo.txt'
```

文字のファイルを行にカットすることに興味があれば、ファイルを文字列変数に読み込んで文字列を切り取ることができます。各行が行終端文字で終わっていると仮定すると、ファイルの最後の文字はフレットになります。ここに例があります。

```
   string =: (1 !: 1) < 'foo.txt'  NB. read the file
   lines =: (< ;. _2) string         NB. cut into
lines
   lines
+----------------+------+-----------+--------+
|What can be said|at all|can be said|clearly.|
+----------------+------+-----------+--------+
```

テキストのファイルを行に分割する際には、注意すべき点が 2 つあります。

第 1 に、あるシステムでは、ファイル内の行が 1 つの改行文字(LF)で終了します。他のシステムでは、各行は、CR(キャリッジリターン)と LF(ラインフィード)のペアで終了する場合があります。

J は、J が実行されているシステムに関係なく、単一の LF の規約に従います。しかし、入力データに CR 文字が存在するように準備する必要があり ます。取り除くための CR から文字列、我々はビットベクトルとそれを減らすことができます (CR NOTEQUAL 文字列)、NOTEQUAL が内蔵されている動詞

なので、

```
string =: (string ~: CR) # string
```

第 2 に、テキストファイルの作成方法によっては、最終行が実際に終了することを保証できない場合があります。したがって、必要に応じて LF を文字列に追加することによって、フレット文字(LF)を自分自身に供給するように準備する必要があります。
フレットを供給して CR 文字を削除することで、文字列を整理する小さな関数は次のように書くことができます：

```
   tidy =: 3 : 0
y =. y , (LF ~: {: y) # LF   NB. supply LF
(y ~: CR) # y                NB. remove CR
)

   (< ;. _2) tidy string
+----------------+------+-----------+--------+
|What can be said|at all|can be said|clearly.|
+----------------+------+-----------+--------+
```

17.2.7 タイリング [Tiling]

式(xu; .3y)では、動詞 u は、y から抽出されたサブアレイの集合のそれぞれに別々に適用される。これらのサブアレイはタイルと呼ばれることがあります。タイルのサイズと配置は、x の値によって定義されます。ここに例があります。仮定し、y があります、

```
   y =: 4 4 $ 'abcdefghijklmnop'
```

私たちのタイルは 2 2 の形をしていなければなりません。それぞれの隣り合っている各軸に沿って 2 ずつずれています。すなわち、オフセットは 2 2 とする。タイリングをテーブルで指定します：最初の行はオフセット、2 番目の図形は、

```
   spec =: > 2 2 ; 2 2  NB.  Offset, shape
```

それで私たちは見る

y	spec	spec < ;. 3 y
abcd efgh ijkl	2 2 2 2	+--+--+ |ab|cd| |ef|gh|

mnop		+--+--+ |ij|kl| |mn|op| +--+--+

指定されたタイリングによって、エッジに不完全な部分(「シャード」)が残ることがあります。シャードは、3 または_3 の「切り取り」に適切な引数を与えることによって、包含または除外することができます。

```
    sp =: > 3 3; 3 3
```

y	sp	sp < ;. 3 y	sp < ;. _3 y
abcd efgh ijkl mnop	3 3 3 3	+---+-+ |abc|d| |efg|h| |ijk|l| +---+-+ |mno|p| +---+-+	+---+ |abc| |efg| |ijk| +---+

第 17 章終了。

第 18 章：セット、クラス [Sets, Classes and Relations]

この章では、J の組み込み関数について詳しく説明します。接続テーマは、やや緩やかに、セット、クラス、リレーションを使って作業しています。
リストの中には、手元の目的のために、アイテムの順序が無関係であり、重複するアイテムの存在が無関係であるとします。次に、リストを有限集合と見なす(表現する)ことができます。要約では、集合 3 1 2 1 は 1 2 3 と同じ集合とみなされる 。
たとえば、リストの各整数が奇数クラスまたは偶数クラスのいずれかに属しているという意味で使用される単語「クラス」。「関係」とは、1 つの列の値とそれに対応する値との関係を表す 2 つ以上の列の表を意味します。たとえば、2 つの列との関係は、ペアのセットです。

18.1 セット [Sets]

18.1.1 メンバー [Membership]

ビルトインの動詞 e. があります。 （小文字の”e.”「メンバー」と呼ばれます）。 式 ″x e. y″は、x が y の任意の項目と一致するかどうか、つまり x がリスト y のメンバーであるかどうかをテストします。 例えば：

y=: 'abcde'	'a' e. y	'w' e. y	'ef' e. y
abcde	1	0	1 0

明らかに、y 内の項目の順序は無関係であり、y 内に重複も存在する。

z=: 'edcbad'	'a' e. z	'w' e. z	'ef' e. z
edcbad	1	0	1 0

テーブルに特定の行が含まれているかどうかをテストできます。

t =: 4 2 $ 'abcdef'	'cd' e。t
ab	1

cd ef ab	

18.1.2 少ない [Less]

ビルトインの動詞 e. (「Less」と呼ばれる、小文字の e.) があります。式 x -. Y は y のメンバーであるものを除いて x の項目のリストを生成する 。

x =: 'consonant'	y =: 'aeiou'	x -. y
consonant	aeiou	cnsnnt

明らかに、y 内の項目の順序は無関係であり、y 内に重複の存在も存在する。

18.1.3 ナブ(~) [Nub]

組み込みの動詞～.があります。(「Nub」と呼ばれるティルダドット)～. の表現。 y は重複のない y の項目のリストを生成する。

nub =: ~.	y =: 'hook'	nub y
~.	hook	hok

テーブルの行に nub を 適用することができます：

t	nub t
ab cd ef ab	ab cd ef

18.1.4 ナブシーブ(~:) [Nub Sieve]

動詞 "nub sieve" (～:)は、真である真理値である真理値ベクトルを与える。

y	b =: ~: y	b # y	nub y
hook	1 1 0 1	hok	hok

18.1.5 セットの関数 [Functions for Sets]

set-union、set-intersection、set-equality などの慣習的な関数は、使用可能な組み込み関数を使用して簡単に定義できます。たとえば、一方のメンバーが他方のメンバーであれば、2 つのセットは等しく、逆の場合も同じです。

```
seteq =: */ @: (e. , e.~)
```

1 2 3 seteq 3 1 2 1	1 2 3 seteq 1 2
1	0

18.2 テーブル副詞 [The Table Adverb]

副詞/動詞を生成することを思い出してください。たとえば+/はリストを合計する動詞です。より正確には、リストを合計するのは+ /のモナドケースです。+ /のダイアディックの場合はテーブルを生成します。例えば：

X =: 0 1 2	Y =: 3 4 5 6	Z =: X +/ Y
0 1 2	3 4 5 6	3 4 5 6 4 5 6 7 5 6 7 8

この例から、左の引数 X の項目と右の引数 Y の項目のすべての可能な組み合わせに動詞+が適用されていることがわかります。

もう一つの例があります：

```
   'abc' = / 'face'
0 1 0
0 0 0 0
0 0 1 0
```

結果は、第 1 行に'a' = 'face'の値を示し、第 2 行に'b' = 'face'の値を示します 。上記の例のように、左からのアイテムと右からのアイテムのすべての可能な組み合わせに動詞を適用することを目指しているなら、満足すべき条件があります。

条件はこれです。まず、第 7 章から、動詞には動詞 b で示すように、モナド、左、右の 3 つのランクがあることを 思い出してください。0。ここでの動詞の左のランクは、左の引数の

ランクと同じでなければならず、動詞の右のランクは、右の引数のランクと同じでなければなりません。この状態は、小さな関数、副詞で確認できます。

```
   check =: 1 : 0
:
L =. 1 { u b. 0        NB. left  rank of verb u
R =. 2 { u b. 0        NB. right rank of verb u
assert. L = # $ 0 { x  [ 'check rank of item of x'
assert. R = # $ 0 { y  [ 'check rank of item of y'
1  NB. OK
)
```

すべての関連ランクがゼロであるため、上記の例では条件が満たされています。

```
   X  + check Y
1
```

ここでは、条件が満たされていない例を示します。X と Y の項目がスカラーであると仮定し、我々はボックスド・ペアを作成することによってそれらを結合しようとします。

fbp =: <@: 、	3 fbp 4
<@ :,	+ --- + | 3 4 | + --- +

最初の試み：

X	Y	X fbp / Y
0 1 2	3 4 5 6	+-------------+ |0 1 2 3 4 5 6| +-------------+

これは､我々が望む結果ではありません.X と Y を指定すると、fbp の左と右のランクが条件を満たさない、つまりゼロではないため、

```
   fbp b. 0
___            NB.アンダーバー

   X fbp check Y
```

```
|assertion failure
|   L=#$0{x['check rank of item of x'
```

対処法は、fbp の引数に必要なランクを指定することです。

```
   X fbp " 0 0 / Y
+---+---+---+---+
|0 3|0 4|0 5|0 6|
+---+---+---+---+
|1 3|1 4|1 5|1 6|
+---+---+---+---+
|2 3|2 4|2 5|2 6|
+---+---+---+---+
```

要約すると、一般的なスキームは、もし我々が

```
                        z =: xf / y
```

そして f は(ランク条件を満たすチェック上記)、次いで、 z は、その結果のテーブルであります

```
                        (<i; j){z =(i {x)f(j {y)
```

すなわち、z は、 x の項目と y の項目の可能なすべてのペアリングを含む。

18.3 テーブルの装飾 [Table Decorations]

テーブルを検討しているので、テーブルの行と列にラベルを付ける 1 つまたは 2 つの料理書の例を提供すると便利です。まず、データが数値表であるとします。

```
   ] data =: 3 3 $ i. 9
0 1 2
3 4 5
6 7 8
```

この関数を使用してデータを書式化することを目指しています。列の幅は 8 桁です。

```
   format =: 8j4& ":
```

必要なラベルは次のとおりであるとします。

```
   rowlabels =: 'first'; 'second' ; 'third'
   collabels =: 'small'; 'medium' ; 'large'
```

これら 3 つの値を計算すると

```
   D =: < format data
```

```
   R =: < > rowlabels
   C =: < ; (_8 & {. ) (&. >) collabels
```

(上記の C の式では、_8 の値は列幅 8 のラベルを右揃えにするための値です)
これで、行ラベルを使用してデータを表示できます。

```
   R,D
+------+------------------------+
|first | 0.0000  1.0000  2.0000|
|second| 3.0000  4.0000  5.0000|
|third | 6.0000  7.0000  8.0000|
+------+------------------------+
```

または列ラベルを使用します。

```
   C ,: D
+------------------------+
|   small  medium   large|
+------------------------+
| 0.0000  1.0000  2.0000|
| 3.0000  4.0000  5.0000|
| 6.0000  7.0000  8.0000|
+------------------------+
```

または両方

```
    2 2 $ a: , C , R ,D
+------+------------------------+
|      |   small  medium   large|
+------+------------------------+
|first | 0.0000  1.0000  2.0000|
|second| 3.0000  4.0000  5.0000|
|third | 6.0000  7.0000  8.0000|
+------+------------------------+
```

18.4 クラス [Classes]

18.4.1 自己分類 [Self-Classify]

文字列の中に出現する文字の数(文字の頻度分布)を見つける問題を考えてみましょう。1 つのアプローチがあります。
それぞれの手紙がナブと等しいかどうかをテストするテーブルを作ります。

`y =: 'hook'`	`nub y`	`(nub y) =/ y`

hook	hok	1 0 0 0 0 1 1 0 0 0 0 1

式((nub y)= / y)は(= y)と省略することができます。組込み動詞=のモナドの場合は、「自己分類」と呼ばれます)。

y	nub y	(nub y) =/ y	= y
hook	hok	1 0 0 0 0 1 1 0 0 0 0 1	1 0 0 0 0 1 1 0 0 0 0 1

= y の 各行を合計すると、ナブの文字の順番でカウントが取得されます。

y	= y	+/ " 1 =y
hook	1 0 0 0 0 1 1 0 0 0 0 1	1 2 1

カウントはナブの文字とペアにすることができます：

y	nub y	(nub y) ;" 0 (+/ " 1 =y)
hook	hok	+-+-+ |h|1| +-+-+ |o|2| +-+-+ |k|1| +-+-+

18.4.2 分類スキーム [Classification Schemes]

ガーデナーは、pH 値に応じて、酸、中性またはアルカリ性の土壌タイプを分類します。6 未満の pH は酸として分類され、6～7 は中性であり、7 以上はアルカリ性であると仮定する。ここでは、pH 値を分類するための動詞があります。酸の場合は A、中性の場合は N、アルカリ(またはライム)の場合は L

を返します。

```
classify =: ({ & 'ANL')  @: ((>: & 6) + (> & 7))
```

classify 6	classify 4.8 5.1 6 7 7.1 8
N	AANNLL

分類の機能は、我々は、分類方式を定義するものとして考えることができます。実際にクラスの名前である文字 ANL はスキームのキーと呼ばれます。

18.4.3 キー副詞 [The Key Adverb]

いくつかのデータ(リストなど)が与えられた場合、各アイテムを分類して対応するキーのリストを生成することができます。

data =: 7 5 6 4 8	k =: classify data
7 5 6 4 8	NANAL

クラス A のすべてのデータ(キーA のすべてのデータ)を選択してグループ化することができます。

data	k	k = 'A'	(k = 'A') # data
7 5 6 4 8	NANAL	0 1 0 1 0	5 4

今度は、各クラスのアイテムを数えたいとします。つまり、同じキーのすべての項目のグループにモナド動詞: #を個別に適用することを目指します。これを行うには、組み込みの副詞/を使用できます。(スラッシュドット、「キー」と呼ばれます)。

data	k =: classify data	k # /. data
7 5 6 4 8	NANAL	2 2 1

別の例として、ボックス動詞 < を適用することによってメンバーを出現させることができるメンバーを数える代わりに、

data	k =: classify data	k < /. data

7 5 6 4 8	NANAL	+---+---+-+ |7 6|5 4|8| +---+---+-+

我々が適用する動詞は、最初のメンバーを分類することによって、それぞれの別々の議論のクラスをそれ自身発見することができます：ここで動詞 u はボックスリストを生成します：キーとカウント：

```
    u =: (classify @: {.) ; #
```

data	k =: classify data	k u /. data
7 5 6 4 8	NANAL	+-+-+ |N|2| +-+-+ |A|2| +-+-+ |L|1| +-+-+

"Key"副詞の一般的な仕組みは次のとおりです。式 xu /で。y は、我々が取る y はリストであること、及び X は、対応する項目のキーのリストであり、Y いくつかの分類スキームに従って、および u は各クラスに別々に適用される動詞です。スキームは次のとおりです。

```
            x u /. y    means   (= x) (u @ #) y
```

説明する：

```
    y =: 4 5 6 7 8
    x =: classify y
    u =: <
```

y	x	= x	(= x) (u @ #) y	x u /. y	y
4 5 6 7 8	AANNL	1 1 0 0 0 0 0 1 1 0 0 0 0 0 1	+---+---+-+ |4 5|6 7|8| +---+---+-+	+---+---+-+ |4 5|6 7|8| +---+---+-+	4 5 6 7 8

我々は、各列がわかり= x がからアイテムを選択し、Y、及び U は、この選択に適用されます。

18.4.4 レターカウントの再訪

文字列中の文字の数を見つける例を思い出してください。

y =: 'LETTUCE'	= y	(nub y) ; " 0 +/ "1 (= y)
LETTUCE	1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0	+-+-+ |L|1| +-+-+ |E|2| +-+-+ |T|2| +-+-+ |U|1| +-+-+ |C|1| +-+-+

ここにバリエーションがあります。私たちは実際に、異なる文字のように多くの異なるクラスを持つ分類スキームを持っていることに注意してください。各文字は、それぞれのクラスのキーです。したがって、y u /. y という形式の式を書くことができます。

適用された動詞 u には、毎回、文字のリストがすべて表示されます。それは: #でカウントし、{、で最初のものを取ります。クラスのラベルにする

```
u =: {. ; #
```

y	= y	y u /. y
LETTUCE	1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0	+-+-+ |L|1| +-+-+ |E|2| +-+-+ |T|2| +-+-+ |U|1| +-+-+ |C|1|

		+-+-+

18.5 関係 [Relations]

以下のような出版物がいくつかあるとします。

- “Pigs" by Smith, on the subject of pigs
- "Pets" by Brown, on cats and dogs
- "Dogs" by Smith and James, on dogs

そのような出版物をカタログ化することを目指しています。そのようなカタログのための適切なデータ構造は、作者をタイトルに関連付けるテーブルと、タイトルをサブジェクトに関連付ける別のテーブルとであり得る。例えば：

author	title
Smith	"Pigs"
Brown	"Pets"
Smith	"Dogs"
James	"Dogs"

title	subject
"Pigs"	pigs
"Pets"	dogs
"Pets"	cats
"Dogs"	dogs

そのようなテーブルを「関係」と呼んでもよい。行の順序は重要ではありません。ここでは、わかりやすくするために、2つの列との関係に固執します。

今我々は関係を表現することを選択する。最初のアプローチでは、ボックス化された文字列の表を選択します。authors-titles の関係は次のとおりです。

```
   ]  AT  =: (". ;. _2) 0 : 0
'Smith'  ; 'Pigs'
'Brown'  ; 'Pets'
'Smith'  ; 'Dogs'
'James'  ; 'Dogs'
)
+-----+----+
|Smith|Pigs|
+-----+----+
```

```
|Brown|Pets|
+-----+----+
|Smith|Dogs|
+-----+----+
|James|Dogs|
+-----+----+
```

titles-subjects 関係は次のとおりです。

```
   ] TS =: (". ;. _2) 0 : 0
'Pigs' ; 'pigs'
'Pets' ; 'cats'
'Pets' ; 'dogs'
'Dogs' ; 'dogs'
)
+----+----+
|Pigs|pigs|
+----+----+
|Pets|cats|
+----+----+
|Pets|dogs|
+----+----+
|Dogs|dogs|
+----+----+
```

18.5.1 関係の参加 [Join of Relations]

authors-titles relation AT と titles-subjects relation TS から、どの著者がどの subject に対してタイトルを書いたかを示す authors-subjects 関係を計算することができます。AT と TS はタイトルに関して結合されていると言います。結合は次のようになります：

```
+-----+----+
|Smith|pigs|
+-----+----+
|Brown|cats|
+-----+----+
|Brown|dogs|
+-----+----+
|Smith|dogs|
```

```
+-----+----+
|James|dogs|
+-----+----+
```

このセクションの計画では、ジョインを計算するための関数を見てから、改良されたバージョンで、そしてリレーションをボックス化された文字列ではなくシンボルのテーブルとして表現するという利点があります。最後に、いくつかのパフォーマンスの比較を見ていきます。
方法は以下の通りです。私たちは、行からなるすべての可能なペア考慮 にテーブルから AT と行 TS テーブルから TS を。at、ts の各ペアは次の形式です。

```
author; title; title; subject
```

title が title と一致する場合、つまり項目 1 が項目 2 と一致する場合、著者とサブジェクト、つまり項目 0 と 3 が抽出されます。テストと at、ts のペア から抽出する動詞は、

```
   test =: 1&{ =  2&{
   extr =: 0 3 & {
```

これらの動詞は、ペアリングを行うために適切な連合に差し込むことができます。この結合を書く際には、この集合が大きくなる可能性があるため、可能な対の集合が同時に存在することを避けることを目的とする。我々はまた、結果に重複を避けることを目指しています。ここでは最初の試みです。

```
   PAIR =: 2 : 0
:
z =.  0 0 $ ''
for_at. x do.
   for_ts.  y do.
     if. u at,ts do. z =. z, v at,ts  end.
   end.
end.
~. z
)
```

結合動詞は次のように書くことができます。

```
   join =: test PAIR extr
```

我々は見る：

AT	TS	AT join TS
+-----+----+ |Smith|Pigs| +-----+----+ |Brown|Pets| +-----+----+ |Smith|Dogs| +-----+----+ |James|Dogs| +-----+----+	+----+----+ |Pigs|pigs| +----+----+ |Pets|cats| +----+----+ |Pets|dogs| +----+----+ |Dogs|dogs| +----+----+	+-----+----+ |Smith|pigs| +-----+----+ |Brown|cats| +-----+----+ |Brown|dogs| +-----+----+ |Smith|dogs| +-----+----+ |James|dogs| +-----+----+

参加上記で定義した動詞のため、遅いテストと EXTR 動詞が単一に適用される X、Y 時対-それらは、スカラー演算です。これらの動詞にできるだけ多くのデータを与えて一度に動かせるようにすれば、パフォーマンスは向上します。(これは J の普遍的なルールです)。ベクトルまたは配列の引数が優れています。ここでは修正された PAIR と結合のベクトル指向のバージョンがありますが、それでもペア全体を構築することはありません。

```
   VPAIR =: 2 : 0
:
z =.  0 0 $ ''
for_at. x do.
     z =. z , |: v (#~"1  u) |: at , "1 y
end.
~. z
)

   vjoin  =: test VPAIR extr
```

これまでと同じ結果が得られます。

AT join TS	AT vjoin TS

+-----+----+ |Smith|pigs| +-----+----+ |Brown|cats| +-----+----+ |Brown|dogs| +-----+----+ |Smith|dogs| +-----+----+ |James|dogs| +-----+----+	+-----+----+ |Smith|pigs| +-----+----+ |Brown|cats| +-----+----+ |Brown|dogs| +-----+----+ |Smith|dogs| +-----+----+ |James|dogs| +-----+----+

上記のように、ボックス化された文字列のテーブルとしてリレーションを表すことは、効率的ではありません。繰り返し値の場合、文字列全体が繰り返されます。値は、文字列全体を比較することによって比較されます。
今度は別の可能性を見てみましょう。ボックス化された文字列ではなく、シンボルのテーブルでリレーションを表すことができます。

18.5.2 シンボルとは [What are Symbols?]

記号は、文字列データを効率的に計算するためのものです。シンボルは、文字、ボックス、および数字が異なるデータ型であるのと同じ方法で、別個のデータ型です。記号は、文字列を識別または参照するスカラーです。
ボックス内の文字列に ビルトインの動詞 s: (小文字のコロン)を適用すると、シンボルを作成できます。

```
   a  =: s: <'hello'
```

ここで、変数 a は symbol 型の値を持ちます。通常の方法でこの値を検査します。

```
   a
`hello
```

値が左引用符で始まる元の文字列として表示されることを確認してください。にもかかわらず、表示された場合、文字列のように見える、それはスカラーです。

a	$ a	# $ a

`hello		0

元の文字列は、シンボルテーブルと呼ばれる J システムによって自動的に管理されるデータ構造に格納されます。文字列はシンボルテーブル内で複製されません。したがって、別のシンボル b が a と同じ文字列から作成された場合、b は a に等しくなります。

a	b =: s: <'hello'	b = a
`hello	`hello	1

比較は、元の文字列を比較する必要がなく、単純なスカラー平等であることに注意してください。
上記の関係は、シンボルの配列に変換して、以前と同じように結合することができます。

sAT =: s: AT	sTS =: s: TS	sAT vjoin sTS
`Smith `Pigs `Brown `Pets `Smith `Dogs `James `Dogs	`Pigs `pigs `Pets `cats `Pets `dogs `Dogs `dogs	`Smith `pigs `Brown `cats `Brown `dogs `Smith `dogs `James `dogs

シンボルは、元の文字列の順序を反映するために辞書順に並べられます。したがって、シンボルのテーブルはソートすることができます：

sAT	/:~ sAT
`Smith `Pigs `Brown `Pets `Smith `Dogs `James `Dogs	`Brown `Pets `James `Dogs `Smith `Dogs `Smith `Pigs

18.5.3 測定値を比較 [Measurements Compared]

ここでは、式を評価するための時間を秒単位で与えるユーティリティ動詞があります(平均 4 回の実行)。

```
time =: (8j5 & ":) @: (4 & (6!:2))
```

上記の関係の例は意味のあるパフォーマンス測定には小さすぎるので、それぞれを 100 回複製することでより大きな関係を作ります。

```
   AT  =: 100 $ AT
   TS  =: 100 $ TS
   sAT =: 100 $ sAT
   sTS =: 100 $ sTS
```

比較する 4 つのケースがあります：

```
   t1 =: time 'AT  join  TS'   NB. scalar method, boxed strings
   t2 =: time 'sAT join  sTS'  NB. scalar method, symbols
   t3 =: time 'AT  vjoin TS'   NB. vector method, boxed strings
   t4 =: time 'sAT vjoin sTS'  NB. vector method, symbols
```

我々は見る：

```
   3  3  $  '  ';  'strings';  'symbols';'scalar';t1;t2;
'vector';t3;t4
+------+--------+--------+
|      |strings |symbols |
+------+--------+--------+
|scalar| 1.06800| 0.02214|
+------+--------+--------+
|vector| 0.01449| 0.00124|
+------+--------+--------+
```

では第 31 章私たちは、関係の参加コンピューティングのパフォーマンスの話題に戻ります。

18.5.4 シンボルの保存と復元 [Saving and Restoring the Symbol Table]

仮定データは、シンボルの配列です。

```
   ] data =: s: 2 2 $ 'hello'; 'blah';'blah';'goodbye'
`hello `blah
`blah  `goodbye
```

データ 内のシンボルの場合、元の文字列(たとえば'hello')はシンボルテーブルにのみ格納され、データ自体には格納されません。シンボルの値を表示するには、元の文字列が必要です。
新しいセッションでデータを読み込むことを目的として、ファイルにデータ を書き込むとします。新しいセッションの開始時に、シンボルテーブルは空です。したがって、シンボルテーブルを以前のセッションから保存し、新しいセッションの開始時にシンボルテーブルを元に戻す必要があります。
まず、値をファイルに保存してそれを取得する 2 つのユーティリティ関数を次に示します。(ファイル内のデータの詳細については、第 27 章と 第 28 章を参照してください)。

```
   save =: 4 : '(3!:1 x ) 1!:2 < y '
   retr =: 3 : '3!:2 (1!:1 < y )'
```

データを、たとえば data.xyz という名前のファイルに保存します。

```
   data save 'data.xyz'
```

シンボルテーブル自体は変数ではありませんが、式 0 s: 10 は値を与えます。この値を symtab.xyz という名前のファイルに保存します

```
   (0 s: 10) save 'symtab.xyz'
```

新しい J セッションを開始します。シンボルテーブルは最初は空のため、最初のセッションで保存したファイルからシンボルテーブルを元に戻します。

```
   10 s: (retr 'symtab.xyz')
 1
```

正しいシンボルテーブルがあれば、ファイルからシンボルデータの配列を取り出すことができます：

```
   DATA =: retr 'data.xyz'
```

シンボルが正しく解釈されていることがわかります。

```
   DATA
`hello `blah
```

```
`blah  `goodbye
```

第 18 章終了

第 19 章：数字 [Numbers]

この章で扱うトピックは次のとおりです。
- J で利用できるさまざまな種類の番号
- 特別な数字(無限と不確定)
- 数字を書くための表記法
- 数字の表示方法とフォーマット方法
- 塩基数
- 乱数

19.1 異なる 6 種類の数字 [Numbers of Six Different Kinds]

J は、次のような数の計算をサポートしています。
- ブーリアン **booleans** (または真理値)
- 整数 **integers**
- 実数 **real**(または浮動小数点数)
- 複素数 **complex-number**
- 拡張された整数 **exttended-integer**(つまり、正確に表現された任意の大きな整数)
- 有理数 **rationals**(すなわち、拡張された整数の対)

各種類の番号は、メモリ内に独自の内部表現を持っています。たとえば、真理値 0 と 1 のみを含む配列 は、浮動小数点形式ではなく、コンパクトな内部形式(ブール値)で格納されます。同様に、(比較的小さな)整数だけを含む配列は、「整数」と呼ばれるコンパクトな形式で格納されます。
適切な表現の選択は、J システムによって完全に自動的に管理され、通常はプログラマが認識しなければならないものではありません。しかし、数値の表現をテストする手段があります。その目的のためのユーティリティ関数があります。

```
   types =: 'bool';'int';'float';'complex';'ext int';'rational'
   type  =: > @: ({ & types) @: (1 4 8 16 64 128 & i.) @: (3 !:0)
```

type 0=0	type 37	type 2.5	type 12345678901
bool	int	float	float

19.1.1 ブール値 [Booleans]

ブール値の論理演算用の組み込み関数があります。これらの

関数に従来の名前を付ける：

```
and    =: *.
or     =: +.
not    =: -.
notand =: *:
notor  =: +:
```

真理値表を表示することができます：

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

p	q	p and q	p or q	not p	p notand q
0	0	0	0	1	1
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

さらに論理関数を通常の方法で定義することができます。例えば、スキームとの論理的な関係

```
p implies q    means    not (p and not q)
```

定義されていないフックで構成していません

```
implies =: not @ (and not)
```

p	q	p implies q
0	0	1
0	1	1
1	0	0
1	1	1

上記の真理値表では、p、q = 2 進数で 00 01 10 11、0 1 2 3 という順番で行が与えられていることに注意してください。真理値表の行を標準の順序で使用すると、result-column はこの例では 4 ビットの数値、1 1 0 1 として読み取ることができます。これは、2 つの引数の論理関数がすべて 16 個しかなく、4 ビットの結果を与えることでそれらのいずれかを指定できることを意味します。

組み込みの副詞があります。（小文字の b 点、「ブール」と

呼ばれます)。これは 0-15 の範囲の整数をとり、4 ビットの結果を表現し、対応する論理関数を生成します。
例えば、論理的な意味では、その 4 ビットの指定は 11 1 1 または 13 であり、含意を 13 b と定義する別の方法を示しています。 私たちは見る：

p	q	p implies q	p (13 b.) q
0	0	1	1
0	1	1	1
1	0	0	0
1	1	1	1

ブール値は、算術値を持つものとして解釈できるため、ブール値を数値と見なします。説明するために、含意には以下のような真理値表があります。

p implies q	p <: q
1	1
1	1
0	0
1	1

数値としてのブール値の別の例として、リスト内の正の数の合計は次のようになります。

z =: 3 _1 4	b =: z > 0	b * z	+/ b * z
3 _1 4	1 0 1	3 0 4	7

19.1.2 整数 [Integers]

32 ビットマシン上で整数の間の範囲 _2147483648 及び 2147483647。
整数による算術演算の結果は、最大整数よりも大きい場合は浮動小数点に変換されます。

maxint=:2147483647	type maxint	z =: 1+maxint	type z
2147483647	int	2.14748e9	float

19.1.3 整数のビット論理関数 [Bitwise Logical Functions on Integers]

J はすべての期待される関数を整数で提供するので、ここではあまり言及する必要はありません。しかし、これは整数のビット単位の論理関数が組み込みの副詞 b を介して利用可能であることに言及すると良いかもしれません。我々は上記で会った。まず、整数の最初と最後の数ビット(32 ビットと仮定)を表示するユーティリティ関数を示します。

```
   bits =: (3 : ' (8{.y), ''...'', (24 }.y) ') @: ({&'01')@:((32#2) & #:)
```

bits 15	bits _15
00000000...00001111	11111111...11110001

論理関数 k は k b によって与えられることを想起されたい。ここで、k は 0～15 の範囲にある。関数(k + 16)b。論理的には同じですが、ブール値ではなく、ビット単位で、つまり機械語で整数に適用されます。
例えば(1b。)は論理的でブール値であり、(17b。)は論理的で整数である。

```
   bits 15 (17 b.) _15
00000000...00000001
```

動詞(32 B)のビット回転 Y によって左方向に X の負の x の場所、または右向き。同様に(33b。)シフトおよび (34b。)は、符号ビットを右方向に伝搬する「符号付きシフト」を実行する。例えば:

```
   ] bits a =: 1
00000000...00000001

   ] bits b =: _1 (32 b.) a   NB. rotating rightwards
10000000...00000000

   ] bits c =: _1 (34 b.) b   NB. shifting right, propagating sign-bit
11000000...00000000

   ] bits d =: 2  (33 b.) c   NB. shifting left, removing sign-bits
00000000...00000000
```

もう 1 つの例として、Collatz 関数を第 10 章: 偶数の場合

は半分に、それ以外の場合は 3 つに分けて追加してください。
ここにビットごとのバージョンがあります。

```
   odd    =: (17 b.) & 1
   halve  =: _1 & (33 b.)          NB. OK for an even number !
   triple =: + (1 & (33 b.))

   collatz =:   halve ` (1 + triple) @. odd

   collatz ^: (i. 10) 5
 5 16 8 4 2 1 4 2 1 4
```

19.1.4 浮動小数点数 [Floating-Point Numbers]

浮動小数点数は、(1)小数部および整数部があるように、コンピュータで表される数値です。(2)番号の値にかかわらず、一定量のコンピュータ記憶装置がその番号によって占有される。したがって、(3)数値が表現される精度は、(PC 上の)最大で約 17 桁の 10 進数に制限される。
浮動小数点数の例は 0.25 2.5 12345678901 です。
「実数」という言葉は、「浮動小数点」とほぼ同じ意味で使用されます。

19.1.5 科学的記法 [Scientific Notation]

時には「科学記法」と呼ばれるものは、非常に大きいまたは非常に小さい数字を書く便利な方法です。例えば、1500000 は 1.5e6、つまり 1.5*10^6 と書くことができます。一般的なスキームは、XeY という形式で書かれた数字で、Y は(正または負の)整数で(X * 10^Y)を意味します。

3e2	1.5e6	1.5e_4
300	1500000	0.00015

3e2 では、文字 e はどんな種類の関数でもない ことに注意してください。小数点が記法の一部であるのと同じように、数字を書くための記法の一部です。
文字 3 の後に e が続き、2 が続き、数字 300 を表す数字であると言う。3 の文字列の後に 0 が 続き、0 が同じ数字を示す別の数字です。異なる数字の数字は、異なる数字を表す便利な方法を提供します。数字で表される数字は、変数とは対照的に「定数」とも呼ばれます。

私たちは数字の話題に戻ってきます。今度はさまざまな種類の話題に戻ります。

19.1.6 浮動小数点数の比較 [Comparison of Floating-Point Numbers]

2 つの数字は、その差が比較的小さい場合、等しいとみなされます。たとえば、a と b の差はゼロではないことがわかりますが、a = b という式では「true」が生成されます。

a =: 1.001	b =: a - 2^_45	a - b	a = b
1.001	1.001	2.84217e_14	1

2 つの数の「相対的な差」は差の大きさを大きさの大きい方で割ったものです:

```
    RD =: (| @: -) % (>. &: |)
```

a = b が真であるためには、相対差（a RD b）は、デフォルトで 2 ^ _44 である「比較許容誤差」と呼ばれる小さな値を超えてはいけません。

a RD b	2^_44	a = b
2.83933e_14	5.68434e_14	1

したがって、2 つの数を比較するには、相対差と公差とを比較する必要があります。後者の比較自体は厳密であり、つまり許容誤差を伴わない。
相対差は 1 でなければならず、したがって公差よりも大きくなければならないので、0 はゼロでない数値には寛容には等しくありません 。

tiny =: 1e_300	tiny = 0	tiny RD 0
1e_300	0	1

しかし、1+tiny は許容値 1 に等しい。

tiny	tiny = 0	1 = tiny + 1
1e_300	0	1

現在有効な比較トレランスの値は、ヌル引数に適用されるビ

ルトインの動詞 9！:18 によって与えられ ます。それは現在 2^_44 です。

9!:18 ''	2^_44
5.68434e_14	5.68434e_14

組み込みの動詞 9！:19 を引数 y に適用すると、 続いて許容値が y に設定されます。次の例は、公差が 2^_44 の場合は a = b ですが、公差がゼロに設定されている場合は、もはや a = b ではないことを示しています。

(9!:19) 2^_44	a = b	(9!:19) 0	a = b
	1		0

9!:18 によって照会され、9!:19 によって設定された許容値は、グローバルパラメータであり =. での計算の結果に影響します。グローバルパラメータに関係なく、指定された公差 t を適用する動詞は、=!.t と書くことができます。たとえば、厳密な(zero-tolerance ゼロ容認の)等価は、次のように定義できます。

```
    streq =: = !.  0
```

グローバル許容値をデフォルト値にリセットすると、次のようになります。

(9!:19) 2^_44	a - b	a = b	a streq b
	2.84217e_14	1	0

= との比較は許容され、<, <:, >, >:, ~: と -: との比較も同様です。 例えば、a-b の差は正であるが、a>b であることを真実にするには小さすぎる。

a - b	a > b
2.84217e_14	0

許容誤差の範囲は 0 と 2 ^ _35 です。 つまり、許容値を 2 ^ _35 より大きく設定しようとするとエラーになります。

(9!:19) 2^_35	(9!:19) 2^_34

	error

大きな公差を許さないことの効果は、浮動小数点に変換されたときに 2 つの異なる整数が等しいと比較することはないということです。

19.1.7 複素数 [Complex Numbers]
-1 の平方根は、慣例的に「i」と呼ばれる虚数である。従来、例えば 3 + i4 として書かれた複素数は、3j4 として書かれた J 内にある。
J において、虚数は実数部ゼロを有する複素数として表される。したがって、-1 の平方根である "i"は 0j1 と書くことができます。

i =: %: _1	i * i	0j1 * 0j1
0j1	_1	_1

複素数は、通常の方法で 2 つの別々の実数から、あるいは組み込み関数 j でより便利に構築できます。(小文字の j 点、「複合体」と呼ばれます)。

3 + (%: _1) * 4	3 j. 4
3j4	3j4

複素数を使った算術演算の例

2j3 * 5j7	10j21 % 5j7	2j3 % 2
_11j29	2.66216j0.472973	1j1.5

3j4 のような複素数はスカラーである単一の数です。実数部と虚数部を別々に抽出するには、組込み動詞 + を使用できます。(「リアル/想像」と呼ばれるプラスのドット)。大きさと角度(ラジアン単位)を別々に抽出するには、組み込み動詞 *. を使用できます。(アスタリスクドット、「長さ/角度」と呼ばれます)。

+. 3j4	*. 3j4

3 4	5 0.927295

大きさと角度が与えられれば、正弦と余弦を取るか、組み込み関数 r. を使ってより簡単に複素数を作ることができます。(小文字の r.ドットは、"Polar"と呼ばれます)。

```
sin =: 1 & o.
cos =: 2 & o.
mag =: 5
ang =: 0.92729522  NB. Radians
```

mag * (cos ang) + 0j1 * sin ang	mag r. ang
3j4	3j4

大きさ X と角度(ラジアン単位) Y の複素定数 は、XarY という形で書くことができます。 これは X r. Y を意味します。同様に、角度が度で与えられるならば、XadY と書くことができます。

5ar0.9272952	5ad53.1301
3j4	3j4

19.1.8 拡張整数 [Extended Integers]

浮動小数点数は、コンピュータのメモリに限られた記憶領域を持ち、正確に約 17 桁までの整数を表すことができます。より長い数値による正確な計算のために、「拡張整数」が利用可能である。「拡張整数」は、何桁の数字が必要であっても、正確に整数を表す数字です。拡張された整数は、文字 'x'が後に続く数字で書き込まれます。以下を比較してください:

a =: *: 10000000001	b =: *: 10000000001x
1e20	100000000020000000001

ここで a は近似値であり、b は正確な結果である。

type a	type b
float	ext int

私たちは、追加することを見ることができる 1 に 1 を加えながら、違いはありません B 違いを生む。

(a + 1) - a	(b + 1) - b
0	1

19.1.9 合理的な番号 [Rational Numbers]

「有理数」は、2 つの整数の比を正確に表す単一の数値です。例えば、3 分の 2 は 2 対 3 の比です．2/3 は、2r3 の表記で有理数として書くことができます。
合理化のポイントは、それらが拡張整数を使った正確な表現であるということです。有理数を用いた算術演算は正確な結果を与えます。

2r3 + 1r7	2r3 * 4r7	2r3 % 5r7
17r21	8r21	14r15

有理数は、拡張整数を分割することで構成できます。以下を比較してください:

2 % 3	2x % 3x
0.666667	2r3

与えられた浮動小数点数から x: 動詞を、

x: 0.3	x: 1 % 3
3r10	1r3

有理数は、x: の逆数、すなわち動詞 x: ^: _1 を使用して浮動小数点近似に変換できます。

float =: x: ^: _1	float 2r3
+--+--+--+ |x:|^:|_1| +--+--+--+	0.666667

有理数が与えられれば、その分子と分母は動詞 2&x: で復元でき、長さ 2 のリストが得られます。

nd =: 2 & x:	nd 2r3
+-+-+--+ |2|&|x:| +-+-+--+	2 3

19.1.10 型変換 [Type Conversion]

ブール型、整数型、拡張整数型、有理式、浮動小数点型、複合型の 6 つのタイプがあります。
算術演算は、複数の型を組み合わせて行うことができます。たとえば、整数に拡張を加えたものには、拡張された、および合理的な時間が与えられ、float は浮動小数点を与えます。

1 + 10^19x	1r3 * 0.75
10000000000000000001	0.25

一般的なスキームは、6 つのタイプがプログレッションを形成することです。ブール値から整数、拡張から有理数、浮動小数点から複雑なものまでです。ブール値は、最も単純な「最も小さい」タイプであり、最も一般的なタイプまたは「最も高い」タイプのものです
2 つのタイプの異なるタイプがある場合、低いタイプのタイプはより高いタイプのタイプに一致するように変換されます。結果は「より高い」ものです。

type 1r3	type 1%3	z =: 1r3, 1%3	type 1r3
rational	float	0.333333 0.333333	rational

19.2 特別な番号 [Special Numbers]

19.2.1 "無限大" [Infinity]

浮動小数点数は、(PC 上で)コンピュータのメモリに格納される方法のため、約 1e308 を超えないようにすることができます。より大きな結果を出そうとする算術演算は、実際には「無限大」と呼ばれる特別な数を生成し、_(アンダースコア)と書かれます。例えば:

1e308 * 0 1 2	1e400	1 % 0
0 1e308 _	_	_

_ _(underscore underscore) として書かれた "負の無限大"もあります。Infinity は浮動小数点数です。

```
   type _
float
```

19.2.2「不確定」の数値 ["Indeterminate" Numbers]

浮動小数点数は 64 ビットの値ですが、すべての 64 ビット値が浮動小数点数として有効なわけではありません。有効でないものは、「Not a Number」または「NaN」と呼ばれます。このような値は、信頼できない外部ソースから J プログラムにインポートされたデータなどで発生する可能性があります。数字の値を表示するとき、J システムは実際には数字ではない浮動小数点数をシンボル _. として報告します 。(_. アンダーバードットは"不確定"と呼ばれる)。

_. の浮動小数点演算の意味のある結果を生むために議論に頼ることはできません。従って _. は単にエラーの兆候とみなされます。

_. のための唯一の信頼できるテストは、動詞 128 !: 5 です。次の例では、信頼できないテスト X = _. かつ、信頼性の高いテスト 128 !: 5 X の結果の違いに注意してください。

X =: 1.5 _. 2.5	X = _.	128!:5 X
1.5 _. 2.5	0 0 0	0 1 0

19.3 番号ベース [Number Bases]

番号 5 は、バイナリで 1 0 1 として表されます。バイナリ数字を計算するための組み込み関数 monadic : #: (ハッシュコロン、 "Antibase Two")があります。結果はリストであることに注意してください。

```
   #: 5
1 0 1
```

2 進数字は 2 進表現であると言います。より一般的には、ベース n 表現を生成することができる。dyadic :#: の左側の引数("Antibase"と呼ばれる)は、基数と桁数の両方を指定します。4 桁の 2 進数を得るには、次のように書くことができます。

```
   2 2 2 2 #: 5
```

```
0 1 0 1
```

63 を 2 つの 8 進数(8 進数)

```
   8 8: 63
7 7
```

混合基底表現が可能です。7265 秒で何時間、何分、何秒がありますか？

```
   24 60 60: #: 7265
2 1 5
```

逆関数は、指定された基底の数字のリストから数値を生成します。Monadic ：#。ベース 2 と呼ばれます。バイナリ 1 0 1 は 5 です

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

ダイアディック：#。「ベース」と呼ばれます。その左の引き数は、右の引き数の数字の基数を指定します。

```
   2 #. 1 0 1
5
```

同等に：

```
   2 2 2 #. 1 0 1
5
```

右側の各桁の左に指定された底辺がなければなりません。そうでない場合は、エラーが通知されます

```
    2 2 #. 1 0 1
|length error
|   2 2    #.1 0 1
|[-535] c:¥users¥homer¥14¥js¥19.ijs
```

再び、混合塩基が可能である： 2 時間 1 分 5 秒は 7265 秒

```
   24 60 60  #. 2 1 5
7265
```

19.4 数値の表記 [Notations for Numerals]

我々は、手紙が形成された数字上で見てきた E、R および J 例えば： 1E3、2R3、および 3j4 を。ここでは、数字を形成するためのより多くの文字を見ていきます。

XpY の形の 文字 p で書かれた数字は、X * pi ^ Y を 意味し 、ここで pi はおなじみの値 3.14159265 ....

pi =: 1p1	twopi =: 2p1	2p_1
3.14159	6.28319	0.63662

同様に、XxY の形式の文字 x で書かれた数字は、X * e ^ Y を 意味し 、ここで、e は親密な値 2.718281828 ....

e =: 1x1	2x_1	2 * e ^ _1
2.71828	0.735759	0.735759

これらの p と x の形式は、多くの桁を書き出すことなく、定数を正確に書く便利な方法を提供します。最後に、10 以外の基数で数値を書くことができます。たとえば、2 進数 101 の 2 進数または 2 進数の値は 5 で 、2b101 と書くことができます。

```
   2b101
5
```

一般的なスキームは、NbDDD.DDD は、数字 DDD.DDD の数字ベース N の数字です。10 より大きい桁では 9 より大きい桁が必要になるので、文字'a'を値 10、'b'を値 11、'z'を値 35 とする。
例えば、文字'f'は数字値 15 を持つので、16 進数(16 進数)では 16bff と書かれた数字は 255 の値を持つ。数ベース N は 10 進数で与えられます。

16bff	(16×15)+15
255	255

もう一つの例。10b0.9 は明らかにベース 10 の数字で " 9/10 "を意味するので、ベース 20 では 20b0.f は "15 分の 20"を意味し 、

```
   10b0.9 20b0.f
0.9 0.75
```

19.4.1 表記法の組み合わせ [Combining the Notations]
表記-文字の E、R、J AR の 広告 P の X 及び B は組み合わせて使用してもよいです。例えば、1r2p1 を "pi over two"

と書くことができます。可能な組み合わせのいくつかのさらなる例がここにあります。
XrY の形式の数字は、X%Y の数を示す 。XeYrZ の形式の数字は、e が r の前に考慮されているため、(XeY)%Z の数を示します。

1.2e2	(1.2e2) % 4	1.2e2r4
120	30	30

フォーム内の数字 XjY は複素数で表す (X J Y) (すなわち、_1 は* Y)): (X +(% 。フォームで符号 XrYjZ は数表す 。(XRY)J が Z ため、R はあるが j の前に考慮する。

3r4	(3r4) j. 5	3r4j5
3r4	0.75j5	0.75j5

XpY の形式の数字は、X * pi ^ Y の数を示す 。XjYpZ の形の数字は、j が p より前に考慮されるため、(XjY)* pi ^ Z を示し ます。

3j4p5	(3j4) * pi ^ 5
918.059j1224.08	918.059j1224.08

XbY の形式の数字は、ベースの Y の数を示す 。XpYbZ の形式の数字は、p が b の前に考慮されるため 、Z-in-base-(XpY) の番号を示します。

(3*pi)+5	1p1b35
14.4248	14.4248

19.5 数字の表示方法 [How Numbers are Displayed]
番号は J で表示され、デフォルトでは有効数字は 6～7 桁までです。これは、通常、小さな整数は正確に表示されますが、大きな数字や有効数字の多い数字がほぼ表示されることを意味します。

10 ^ 3	2.7182818285	2.718281828 * 10 ^ 7
1000	2.71828	2.71828e7

表示に使用される有効桁数は、「印刷精度」と呼ばれるグローバル変数によって決まります。2 つの関数を定義すると:

```
ppq =: 9 !: 10    NB. print-precision query
pps =: 9 !: 11    NB. print-precision set
```

式 ppq "は現在有効な印刷精度の値を与えますが、 pps n は印刷精度を n に設定します。

ppq ''	e =: 2.718281828	pps 8	e
6	2.71828		2.7182818

19.5.1 "フォーマット"の動詞 [The "Format" Verb]

内蔵の動詞があります「: (コロンと呼ばれ、ダブルクォート『フォーマット』)単項のフォーマットは、現在有効な印刷精度の数を表す文字列に数値を変換する以下の例では、ということに注意してください。。スカラでは、一方、a の書式化された表現は文字のリストです。

a =: 1 % 3	": a	$ ": a
0.33333333	0.33333333	10

引数は数値のリストで、結果は単一の文字列です。

b =: 1 % 3 4	": b	$ b	$ ": b
0.33333333 0.25	0.33333333 0.25	2	15

ダイアディック・フォーマットでは、表現をより詳細に制御できます。左の引数は複雑です: 例えば、8j4 という値は 8 文字の幅と小数点以下 4 桁の数値を書式化します 。

c =: % 1 + i. 2 2	w =: 8j4 ": c	$ w
1 0.5 0.33333333 0.25	1.0000 0.5000 0.3333 0.2500	2 16

幅が 0 と指定されている場合(例えば 0j3 の場合)、十分な幅が許されます。小数点以下の桁数が負の場合(10j_3 の場合)、数値は「科学的表記法」で表示されます。

c	0j3 ": c	10j_3 ": c

1 0.5 0.33333333 0.25	1.000 0.500 0.333 0.250	1.000e0 5.000e_1 3.333e_1 2.500e_1

モナド形式は通常の方法で複素数を表示しますが、二項形式は複素数の実部のみを表示します。

z =: 3.14j2.72	": z	6j2 ": z
3.14j2.72	3.14j2.72	3.14

組み込み動詞 8！: n によって、より多くの書式設定オプションが提供されます。ここでは、J ディクショナリで説明されている多くのオプションのいくつかを示す小さな例を示します。
フォーマットされる数字のテーブルが N

```
    ] N =: 3 2 $  3.8  _2000 0  123.45 _3.14 15000
  3.8  _2000
    0 123.45
_3.14  15000
```

私たちは、N の各列を別々にフォーマットすることができます。最初の列の数字は、ゼロ、小数点以下 0 桁の 6 文字の幅のときに空白として表示されるものとします。このような「書式設定のフレーズ」を書いています。

```
   fp1 =: 'b6.0'
```

ここで、「b」は、ゼロのときに空白を意味します。
2 番目の列に対して、各 3 桁の間にカンマが必要であるとします。負の数を次の「CR」で表示する必要があります。したがって、負でない数字の後に 2 つの空白文字を続けて、小数点が垂直に並ぶようにします。小数点以下 2 桁の 12 文字の幅が必要です。
適切な書式指定フレーズは次のようになります。

```
   fp2 =:  'cn {CR} q {} 12.2'
```

ここで、'c'はカンマを意味し、n {CR}は負数の後の CR を意味し、q {}は非負の後の 2 つのスペースを意味する。
書式設定動詞 8 を適用する！: 2 を見る

N	(fp1;fp2) (8!:2) N

3.8 _2000	4 2,000.00CR
0 123.45	123.45
_3.14 15000	-3 15,000.00

フォーマットされた結果は、N が 3 行を持ち、第 1 列と第 2 列に 6 と 12 の幅を指定したため、次元 3×18 の文字表です。

```
   $ (fp1;fp2) (8!:2) N
3 18
```

19.6 乱数 [Random Numbers]

19.6.1 ロール [Roll]

乱数を生成するための組み込み関数があります。モナディック？「ロール」と呼ばれてい ます。n は、0 から n-1 とマークされた n 個の面を有するダイを丸めた結果を与える。

```
   ?6
4
```

つまり、?n は i の項目から選択される。n は ランダムに等確率である。
乱数のリストは、n 値を繰り返すことによって生成されます。例えば、6 面ダイの 4 つのロールは、

```
   ? 6 6 6 6
3 4 5 2
```

またはより便利には、

```
   ? 4 $ 6
2 3 5 3
```

19.6.2 一様分布 [Uniform Distribution]

引数が 0 の場合、モナドは？0 より大きく 1 より小さい一様分布のランダムな実数を生成する。

```
   ? 0 0 0 0
0.14112615 0.083891464 0.41388488 0.055053198
```

19.6.3 その他の分布 [Other Distributions]

組み込み動詞？等価な数値を生成します。J アプリケーションライブラリによって提供される他のさまざまなディストリビューション stats / distribs Addon

19.6.4 カードを配る [Deal]

ダイアディック ？「取引」と呼ばれます。 x ？y は、i からランダムに選択された x の整数のリストです。y を置き換えることなく、つまり x の整数はすべて異なります。
デッキ内のカードは番号が付けられていると仮定 0 から 51 を、次いで、13？52 は、ランダムに選択された 13 枚のカードの片手を扱います。

```
   13 ? 52
44 2 36 30 0 6 43 26 28 1 34 48 41
```

デッキ全体のシャッフルは 52 で与えられますか？52。シャッフルして 4 本の手を扱うには:

```
   4 13 $ 52 ? 52
15 18  3  8 11 25 27 51 20 31 50 48 35
45 39 21 29 10 33 32 41 36  0 34 16 22
19 14 37  2 24 42  6  7 30 46 47 28 26
38 23 40 13  4  9  5 12 49  1 44 43 17
```

第 19 章終了。

第 20 章: スカラー数値関数 [Scalar Numerical Functions]

この章では、数値から数値を計算するための組み込みのスカラー関数について説明します。この章は関数のまっすぐなカタログであり、セクションへのリンクは次のとおりです。

Ceiling	Conjugate	cos	cos^{-1}	cosh	cosh^{-1}
Decrement	divide	Double	Exponential	Factorial	Floor
GCD	Halve	Increment	LCM	Logarithm	Log, Natural
Magnitude	Minus	multiply	Negate	OutOf	PiTimes
Plus	power	Pythagorean	Reciprocal	Residue	Root
Signum	sin	sin^{-1}	sinh	sinh^{-1}	Square
SquareRoot	tan	tan^{-1}	tanh	tanh^{-1}	

20.1 プラスおよびコンジュゲート [Plus and Conjugate]

ダイアディック+は算術加算です。

2 + 2	1.5 + 0.25	3j4 + 5j4	2r3 + 1r6
4	1.75	8j8	5r6

Monadic + は「共役」です。実数 y については、共役は y である。複素数 xjy (x + 0jy)の場合、共役は x - 0jy です。

+ 2	+ 3j4
2	3j_4

20.2 マイナスと否定 [Minus and Negate]

二項演算子- 算術減算です。

2 - 2	1.5 - 0.25	3 - 0j4	2r3-1r6
0	1.25	3j_4	1r2

Monadic - は「否定」です。

- 2	- 3j4

_2	_3j_4

20.3 インクリメントとデクリメント [Increment and Decrement]

Monadic >: は「インクリメント」と呼ばれます。引数に 1 を加えます。

>: 2	>: 2.5	>: 2r3	>: 2j3
3	3.5	5r3	3j3

Monadic <: は "Decrement"と呼ばれます。これは、引数から 1 を減算する。

<: 3	<: 2.5	<: 2r3	<: 2j3
2	1.5	_1r3	1j3

20.4 時間と署名 [Times and Signum]

ダイアディック * は乗算です。

2 * 3	1.5 * 0.25	3j1 * 2j2	2r3 * 7r11
6	0.375	4j8	14r33

Monadic * は「Signum」と呼ばれます。実数のために Y の値 (* y) がある _1 又は 0 または 1 として、y は、ゼロまたは正、負です。

* _2	* 0	* 2
_1	0	1

より一般的には、y は実数または複素数であり、signum は y % | y と等価です。複素数の符号は大きさ 1 で引数と同じ角度を持ちます。

y =: 3j4	| y	y % | y	* y	| * y
3j4	5	0.6j0.8	0.6j0.8	1

20.5 分割と相互 [Times and Signum]

二項演算子は除算です。

2 % 3	1.4 % 0.25	3j4 % 2j1	12x % 5x
0.666667	5.6	2j1	12r5

1 % 0 は「無限」ですが 0 % 0 は 0 です。

1 % 0	0 % 0
_	0

Monadic %は、 "相反する"関数です。

% 2	% 0j1
0.5	0j_1

20.6 ダブルとハーフ [Double and Halve]

Monadic + : は "二重"動詞です。

+: 2.5	+: 3j4	+: 3x
5	6j8	6

Monadic - : は「半分」動詞です:

-: 6	-: 6.5	-: 3j4	-: 3x
3	3.25	1.5j2	3r2

20.7 床と天井 [Floor and Ceiling]

Monadic <.(左角括弧内の点)は "Floor"と呼ばれます。実数 y の場合、y の底は、整数 y、つまり y を超えない最大の整数に丸められます。

<. 2	<. 3.2	<. _3.2	<. 4r3
2	3	_4	1

複素数 y の場合、床は単位円の中心 y 内にあります。つまり、(y - <. y)の大きさは 1 未満です。

y =: 3.4j3.4	z =: <. y	y - z	| y-z
3.4j3.4	3j3	0.4j0.4	0.565685

この条件(大きさ1未満)はの床が言うことを意味3.8j3.8はないが3j3が、4j3 ので3j3条件を満たしていません。

y =: 3.8j3.8	z =: <. y	| y-z	| y - 3j3
3.8j3.8	4j3	0.824621	1.13137

Monadic >. は「天井」と呼ばれます。 実数 y の場合、y の上限は、整数 y に丸められた y 、つまり、y 以上の最小の整数です。 例えば：

>. 3.0	>. 3.1	>. _2.5
3	4	_2

天井は複合 y に適用されます。

>. 3.4j3.4	>. 3.8j3.8
3j4	4j4

20.8 べき乗と指数関数 [Power and Exponential]

Dyadic ^ は「べき乗 Power」動詞です：（x ^ y）は x の y を示します。

10 ^ 2	10 ^ _2	100 ^ 1%2
100	0.01	10

Monadic ^は累乗(または反対数)です。^ y は(e ^ y)を意味し、e はオイラー定数、2.71828 ... 。

^ 1	^ 1.5	^ 3r2	^ 0j1
2.71828	4.48169	4.48169	0.540302j0.841471

彼の墓石に刻印されたオイラー方程式は次のようになります。

$e^{i\pi} + 1 = 0$

```
   (^ 0j1p1)+ 1
0j1.22465e_16
```

上 の^ 3r2 の例は、有理数は^の領域にあるが、結果は実数

であり、合理的ではないことを示している。合理的な議論は実際には最初に実数に変換される。

20.9 二乗 [Square]

Monadic *: は「二乗」です。

*: 4	*: 2j1
16	3j4

20.10 平方根 [Square Root]

Monadic %: は「平方根」です。

%: 9	%: 3j4	2j1 * 2j1
3	2j1	3j4

20.11 ルート [Root]

x が整数の場合、x %: y は、y の x 番目の根(x の y 乗根)です。

3 %: 8	_3 %: 8
2	0.5

より一般的には、(x %: y)は(y ^% x)の略語です。

x =: 3 3.1	x %: 8	8 ^ % x
3 3.1	2 1.95578	2 1.95578

20.12 対数と自然対数 [Logarithm and Natural Logarithm]

Dyadic ^. は、基底 x の対数関数です。 つまり、（x ^. y）は、x を基底とする対数です。

10 ^. 1000	2 ^. 8	10 ^. 2j3	10 ^. 2r3
3	3	0.556972j0.426822	_0.176091

Monadic ^ 「自然対数」関数です。

e =: ^ 1	^. e	^. 2j3	^. 2r3

2.71828	1	1.28247j0.982794	_0.405465

^. の例 上記の 2r3 は、有理数が ^. の領域にあることを示している。 結果は現実的で合理的ではありません。合理的な議論は実際には最初に実数に変換される。

20.13 階乗 [Factorial and outof]

階乗関数 !. は Monadic です。

! 0 1 2 3 4	! 5x 6x 7x 8x
1 1 2 6 24	120 720 5040 40320

y 個のオブジェクトの中から選択された x 個のオブジェクト の組み合わせの数は、x ! y 。

1 ! 4	2 ! 4	3 ! 4
4	6	4

20.14 絶対値と剰余 [Magnitude and Residue]

Monadic(単項) | は「絶対値」と呼ばれます。実数 y の場合、y の Magnitude は Absolute(絶対値)です。

| 2	| _2
2	2

より一般的には、y は実数または複素数で、大きさは(%: y * + y)と等価です。

y =: 3j4	y * + y	%: y * + y	| y
3j4	25	5	5

二項動詞| 「剰余」と呼ばれます。余り y はによって分割され、X は で与えられる(| Y X) 。

10 | 12	3 | _2 _1 0 1 2 3 4 5	1.5 | 3.7
2	1 2 0 1 2 0 1 2	0.7

もし| X y が 0 の場合、x は y の約数です。

4 | 12	12 % 4
4 | 12	12 % 4

「剰余」関数は複素数に適用されます。

a =: 1j2	b=: 2j3	a | b	a | (a*b)	(b-1j1) % a
1j2	2j3	0j_1	0	1

20.15 最大公約数および最小公倍数 [GCD and LCM]

x と y の 最大公約数(GCD)は(x + .y)で 与えられます。実在と合理は+の領域にあります。

6 +. 15	_6 +. _15	2.5 +. 3.5	6r7 +. 15r7
3	3	0.5	3r7

複素数は+の領域にもあります。。

a=: 1j2	b=:2j3	c=:3j5	(a*b) +. (b*c)
1j2	2j3	3j5	2j3

最小公倍数 X 及び Y は (X * Y)で与えられる。

(2 * 3) *. (3 * 5)	2*3*5
30	30

20.16 円周倍率 [Pi Times]

動詞が組み込まれています o.（小文字のオー .）。Monadic o.「Pi Times」と呼ばれる。その引数に 3.14159 を掛けます...

o. 1	o. 2	o. 1r6	o. 2j3
3.14159	6.28319	0.523599	6.28319j9.42478

o. の例。上の 1r6 は、合理的なものが sin の領域にあるが、その結果は現実的で合理的ではないことを示している。合理的な議論は実際には最初に実数に変換される。

任意の精度の結果は、動詞 < に拡張整数の引数を与えることによって生成できます。
<. @ o. Ｊインタプリタに特別なコードがあります。

```
   ] z =: (<. & o.) 10^40x
3141592653589793238462643383279502884197 1
   datatype z
extended
```

20.17 三角関数とその他の関数 [Trigonometric and Other Functions]

場合 y はラジアン単位の角度であり、その後の正弦波 Y は式で与えられる 1 o. y. (6 を超える π) の sin は 0.5

y =: o. 1r6	1 o. y
0.523599	0.5

ダイアディックの一般的なスキーム o. それは(k o. y)は、k によって選択された関数を y に適用することを意味します。ここで、y はラジアン単位の角度です。
利用可能な機能に従来の名前を付けると、次のようになります。

```
   sin   =:  1 & o.  NB.  sine
   cos   =:  2 & o.  NB.  cosine
   tan   =:  3 & o.  NB.  tangent

   sinh  =:  5 & o.  NB.  hyperbolic sine
   cosh  =:  6 & o.  NB.  hyperbolic cosine
   tanh  =:  7 & o.  NB.  hyperbolic tangent

   asin  =: _1 & o.  NB.  inverse sine
   acos  =: _2 & o.  NB.  inverse cosine
   atan  =: _3 & o.  NB.  inverse tangent

   asinh =: _5 & o.  NB.  inverse hyperbolic sine
   acosh =: _6 & o.  NB.  inverse hyperbolic cosine
   atanh =: _7 & o.  NB.  inverse hyperbolic tangent
```

y	sin y	asin sin y	sin 1r4
0.523599	0.5	0.523599	0.247404

上 の sin 1r4 の例は、合理的なものが sin の領域にある が、その結果は現実的で合理的ではないことを示している。合理的な議論は実際には最初に実数に変換される。

20.18 ピタゴラスの関数 [Pythagorean Functions]
「ピタゴラス」機能もあります:

```
 0 o. y  means   %:   1 - y^2
 4 o. y  means   %:   1 + y^2
 8 o. y  means   %: - 1 + y^2
_4 o. y  means   %:  _1 + y^2
_8 o. y  means - %: - 1 + y^2
```

y =: 0.6	0 o. y	%: 1 - y^2
0.6	0.8	0.8

複素数上の関数の別のグループ:

```
 9 o. xjy means  x  (real part)
10 o. xjy means  %: (x^2) + (y^2) (magnitude)
11 o. xjy means  y  (imag part)
12 o. xjy means  atan (y % x) (angle)
```

9 o. 3j4	10 o. 3j4	11 o. 3j4	12 o. 3j4
3	5	4	0.927295

そして最後に

```
        _9 o. xjy means xjy  (identity)
_10 o. xjy   means x j -y (conjugate)
_11 o. xjy   means 0j1 * xjy (j. xjy)
_12 o. a means (cos a)+(j. sin a) (inverse angle)
```

例えば:

a =: 12 o. 3j4	_12 o. a
0.927295	0.6j0.8

第 20 章終了。

第 21 章: 因子と多項式 [Factors and Polynomials]

この章では、組み込み関数を見ていきます:

- Monadic q: 数の素因数を計算する
- Dyadic q: 素数のべき乗としての数を表す
- Monadic p: 素数を生成する
- Dyadic p. 多項式を評価する
- Monadic p. 多項式の根を見つける

21.1 プライムとファクター [Primes and Factors]

組み込み関数 monadic q: 与えられた数の素因数を計算します。

q: 6	q: 8	q: 17 * 31	q: 1 + 2^30
2 3	2 2 2	17 31	5 5 13 41 61 1321

番号 0 は q のドメインにありません。番号 1 は q: のドメインにありますが、要因がないとみなされます。つまり、要因のリストは空です。

q: 0	q: 1	# q: 1
error		0

多数(約 2 ^ 53 以上)の場合、その有効な数字がすべて q に供給されるように、その値を拡張整数として指定する必要があります。

```
   q: 1 + 2 ^ 53x
3 107 28059810762433
```

素数は、その要素のリストの唯一のメンバーです。したがって、素数の検定は、フックとして簡単に書くことができます:

その要素のメンバー

pr =: e. q:	pr 8	pr 17	pr 1
e. q:	0	1	0

正の整数は、連続する素数のべき乗の積として書くことができます。いくつかの力はゼロになるでしょう。たとえば、

```
      9 =(2 ^ 0)*(3 ^ 2)*(5 ^ 0)*(7 ^ 0)
1
```

力のリスト、ここでは 0 2 0 0 ... は二項 q で生成することができます: 。左の引数 x は、生成するパワーの数を指定します。

4 q: 9	3 q: 9	2 q: 9	1 q: 9	6 q: 9
0 2 0 0	0 2 0	0 2	0	0 2 0 0 0 0

"無限大" (_)の左の引数を与えることは、生成された力の数が十分であることを意味し、その場合、最後のものは非ゼロになります。

_ q: 9	_ q: 17 * 31
0 2	0 0 0 0 0 0 1 0 0 0 1

素数を生成 する組み込み関数 monadic p: (小文字の p コロン)があります。たとえば(p: 17)は 18 番目のプライムです。

p: 0 1 2 3 4 5 6	p: 17
2 3 5 7 11 13 17	61

私のコンピュータでは、このように生成できる最大の素数は、p: 2 ^ 26 と p: 2 ^ 27 の間にあります。

p: 2^26	p: 2^27	p: 2^27x
1339484207	error	error

21.2 多項式 [Polynomials]

21.2.1 係数 [Coefficients]

x が変数の 場合、従来の表記法、たとえば

$$a + bx + cx^2 + dx^3 + \ldots$$

x の多項式と言います。 係数 a、b、c、d ...のリストに C を書くと、

```
    C =: _1 0 1
```

x に 値を代入します(たとえば、

```
    x = 2
```

多項式はJの形で+ / C * x ^ i. # C のように書くことができます。

C	#C	i.#C	x	x^i.#C	C*x^i.#C	+/C*x^i.# C
_1 0 1	3	0 1 2	2	1 2 4	_1 0 4	3

二項動詞 p. は、この表現を C p. x に短縮することを可能にし、

+/C*x^i.# C	C p. x
3	3

スキームは、係数 C のリストに対して:

```
C p. x   means   +/ C * x ^ i. # C
```

多項式関数は C&P の形式で記述するのが便利です。

p =: _1 0 1 & p.	p x
_1 0 1&p.	3

この形式にはいくつかの利点があります: コンパクトで書くこと、効率的に評価すること、そして(われわれが分かるように)区別しやすいことです。

21.2.2 ルーツ:多項式の根 [Roots]

係数 C のリストが与えられれば、多項式関数 C&p.の根を C にモナド p.を適用することによって計算することができる。

C	p =: C & p.	Z =: p. C
_1 0 1	_1 0 1&p.	+-+----+ |1|1 _1| +-+----+

結果 Z は、M; R、すなわち M の後ろに R のリストが続くボックス型構造であることがわかります。R の各ルートに適用される p がゼロになることが予想されます。

'M R' =: Z	R	p R

+-+----+ |1|1 _1| +-+----+	1 _1	0 0

乗数 M の意義は 次のとおりです。根 R のリストに対して r、s、t ...を書くと、

```
    'rs' =: R
```

その後、M は、その結果、多項式である C の P。x は次のように等価に書くことができます。

```
   M * (x-r)*(x-s)
 3
```

またはよりコンパクトに、

```
   M * */x-R
 3
```

私たちはそのモナドを見ました。与えられた係数 C は、 乗数 - 根 M、R を計算する。さらに、与えられた M; R、その後、モナド p。係数 C を計算する。

C	MR =: p. C	p. MR
_1 0 1	+-+----+ |1|1 _1| +-+----+	_1 0 1

21.2.3 乗数とルーツ [Multiplier and Roots]

上記の p の左の議論を見ました。スキームを用いて係数のリストにすることができます。

```
          C p. x   is   +/ C * x ^ i. # C
```

p の左引数。multiplier; list-of-roots の形式にすることもできます。このようにして、指定された根を持つ多項式関数を生成することができます。根が 2 3 であると仮定する。

p =: (1; 2 3) & p.	p 2 3
(1;2 3)&p.	0 0

このスキームは、

```
          (M;R) p. x   means   M * */ x - R
```

M; R が p. C のとき、 (M; R) p. x は C p. x と同じであると期待する。

C	MR=: p.C	MR p. x	C p. x
_1 0 1	+-+----+ |1|1 _1| +-+----+	3	3

21.2.4 多項式 [Multinomials]

多項式にゼロ係数が多い場合、関数を「多項式」形式、つまり係数がゼロの項を省略し、代わりに係数 - 指数ペアのリストを指定する方が便利です。ここに例があります。多項式_1 0 1&p。0 でない係数は 1 番目と 3 番目、_1 1 であり、対応する指数は 0 2 です。こうしてペアを形成する:

coeffs =: _1 1	exps=: 0 2	pairs =: coeffs ,. exps
_1 1	0 2	_1 0 1 2

今度は、ペアを boxed left 引数として p に渡すことができます。 結果は元の多項式の場合と同じであると期待します。

x	pairs	(< pairs) p. x	_1 0 1 p. x
2	_1 0 1 2	3	3

多項式の場合、指数は非負の整数に限定されません。たとえば、指数と係数は次の式で与えられます。

```
E =: 0.5 _1 2j3
C =: 1 1 1
```

関数の多項式は次のようになります。

```
f =: (< C                    .E) & p.
```

比較のために、同等の関数:

```
g =: 3 : '+/ C * y ^ E'
```

私たちは見る

x=: 2	f x	g x
2	_0.0337641j3.49362	_0.0337641j3.49362

☆第 21 章終了

第 22 章: ベクトルと行列 [Vectors and Matrices]

この章では、ベクトルと行列を使った計算をサポートする組み込み関数について説明します。

22.1 Dot . 結合詞 [The Dot Product Conjunction]

第 08 章 の動詞の構成を思い出してください。積和動詞から構成することができ、合計および製品 で: @連動。

P =: 2 3 4	Q =: 1 0 2	P * Q	+/ P * Q	P (+/ @: *) Q
2 3 4	1 0 2	2 0 8	10	10

接続詞があります。(ドット、「ドットプロダクト」と呼ばれます)。2 つのリストの積和を計算するために、@: の代わりに使用することができます。

P	Q	P (+/ @: *) Q	P (+/ . *) Q
2 3 4	1 0 2	10	10

明らかに、. 結合詞は、構成の形式であり、@: または @ のバリエーションです。以下では、ベクトルや行列を使って作業する方が便利であることがわかります。

22.2 ベクトルのスカラ積 [Scalar Product of Vectors]

P は 3 つの数のリストであることを思い出してください。これらの数を 3 次元空間上の点の座標として解釈すると、P は 0 0 0 の点から P 点までの長さと方向を持つ線分であるベクトルを定義するものとみなすことができます。ベクトル P を参照することができます。
P と Q をベクトルとして解釈すると、P (+ / *) Q は P と Q の「スカラー積」と呼ばれるものを与えます。同じものの他の名前は「外積」または「内積」または「行列積」である。 この章では、関数ドットを定義するニュートラルな用語「ドットプロダクト」に固執しましょう。

dot =: +/ . *	P	Q	P dot Q
+/ .*	2 3 4	1 0 2	10

ベクトル P と Q のスカラー積の教科書の定義は、次の形式で現れます。

```
(magnitude P) * (magnitude Q) * (cos alpha)
```

ここでベクトルの大きさ(または長さ)は成分の二乗和の平方根であり、α は P と Q との間の最小の非負角である。この形式の P ドット Q との等価性を示すために、vector -of-angle-between-vectors のためのベクトルの大きさと ca の効用動詞 ma を定義することができます。

《訳者註》**内積はスカラー積、外積はベクトル積と呼ばれる。**

```
ma  =: %: @: (+/ @: *:)
ca  =: 4 : '(-/ *: b,(ma x-y), c) % (2*(b=.ma x)*(c=. ma y))'
```

ベクトル 3 4 の大きさは 5 であり、P とそれ自身の間の角度がゼロであると期待し、余弦は 1 になると予測します。

ma 3 4	P ca P
5	1

ドット動詞は上記のテキストブック形式と同等 であることがわかります。

P	Q	P dot Q	(ma P)*(ma Q)*(P ca Q)
2 3 4	1 0 2	10	10

22.3 行列積 [Matrix Product]

ドットと 呼ばれる動詞はベクトルと行列の「行列積」です。

M =: 3 4 ,: 2 3	V =: 3 5	V dot M	M dot V	M dot M
3 4 2 3	3 5	19 27	29 21	17 24 12 17

Z =を計算するには、A の最後の次元は B の最初の次元と等しくなければなりません。

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

$ A	$ B	Z =: A dot B	$ Z
2 5	5 4	10 10 10 10 10 10 10 10	2 4

この例では、最後と最後のディメンションが結果から消えることを示しています。これら 2 つの次元が等しくなければ、エラーが通知される。

$ B	$ A	B dot A
5 4	2 5	error

22.4 一般化 [Generalizations]

22.4.1 さまざまな動詞 [Various Verbs]

ドットプロダクト動詞はドットプロダクト動詞を(+ /. *)で表します。パターン上に他の動詞(u . v)を形成することができます。

たとえば、人物 i が人物 j の子であり、行 i の列 j で正方形のブール値行列で表される人物の関係を考えてみましょう。動詞（ +. logical-or）と（ *. logical-and）を使って動詞（+. /. *.）との孫関係を計算することができます。

```
   g   =: +. / . *.
```

"子"関係を行列 C にする:

```
   C =: 4 4 $ 0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0
```

そして、孫関係は、いわば、子関係が二乗されたものです。

C	G =: C g C
0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0

人 3 が人 1 の子であり、人 1 が人 0 の子である ことを C から見ることができる。したがって、G 人 3 に見られるように、人 0 の孫である。

22.4.2 シンボリック演算 [Symbolic Arithmetic]

'シンボリック演算'とは、文字列'a'と'b'を象徴的に追加して文字列'a + b'を得ることです。以下は、(ボックス化された)文字列に対していくつかの限られたシンボリック演算を行うユーティリティ関数の小さなコレクションです。

```
   pa     =: ('('&,) @: (,&')')
   cp     =: [ ` pa @. (+./ @: ('+-*' & e.))
   symbol =: (1 : (':';'< (cp > x), u, (cp > y)'))("
0 0)

   splus  =: '+' symbol
   sminus =: '-' symbol
   sprod  =: '*' symbol

   a =: <'a'
   b =: <'b'
   c =: <'c'
```

a	b	c	a splus b	a sprod b splus c
+-+ |a| +-+	+-+ |b| +-+	+-+ |c| +-+	+---+ |a+b| +---+	+-------+ |a*(b+c)| +-------+

象徴的な生成物の変形として、従来の表記法のような効果を与えるために乗算記号を削除することができました:

```
   sprodc =: '' symbol
```

a sprod b	a sprodc b
+---+ |a*b| +---+	+--+ |ab| +--+

"ドットプロダクト"の引数として、シンボリック演算を実行するための動詞を提供することができます。想起されるドット動詞(+ /. *)については、記号バージョンは次のとおりです。

```
   sdot =: splus / . sprodc
```

説明する:

```
   S =: 3 2 $ < "0 'abcdef'
   T =: 2 3 $ < "0 'pqrstu'
```

S	T	S sdot T
+-+-+ |a|b| +-+-+ |c|d| +-+-+ |e|f| +-+-+	+-+-+-+ |p|q|r| +-+-+-+ |s|t|u| +-+-+-+	+-----+-----+-----+ |ap+bs|aq+bt|ar+bu| +-----+-----+-----+ |cp+ds|cq+dt|cr+du| +-----+-----+-----+ |ep+fs|eq+ft|er+fu| +-----+-----+-----+

22.4.3 行列積 [Matrix Product in More than 2 Dimensions]

3 次元の例は十分に一般的です。記号的に:

```
   A =: 1 2 3 $ <"0 'abcdef'
   B =: 3 2 2 $ <"0 'mnopqrstuvwx'
```

A	B	Z =: A sdot B
+-+-+-+ |a|b|c| +-+-+-+ |d|e|f| +-+-+-+	+-+-+ |m|n| +-+-+ |o|p| +-+-+ +-+-+ |q|r| +-+-+ |s|t| +-+-+ +-+-+ |u|v| +-+-+ |w|x| +-+-+	+----------+----------+ |am+(bq+cu)|an+(br+cv)| +----------+----------+ |ao+(bs+cw)|ap+(bt+cx)| +----------+----------+ +----------+----------+ |dm+(eq+fu)|dn+(er+fv)| +----------+----------+ |do+(es+fw)|dp+(et+fx)| +----------+----------+

A の最後の次元は、B の最初の次元と等しくなければなりません。結果 Z の形状は、A の先行寸法の後に B の後続寸法が

続きます。

$A	$B	$Z
1 2 3	3 2 2	1 2 2 2

Z の 無次元スカラー は A の「行」と B の「列」を組み合わせているため 、A と B の最後と最初の次元は消えてしまった。結果 Z では、A の各行が B の全体と別々に結合されていることがわかります。

22.4.4 ドットとの比較: [Dot Compared With @:]

第 7 章 から、ダイアディック動詞 v には左右のランクがあることを 思い出してください。与えられた動詞からランクを抽出するためのいくつかのユーティリティ関数があります。

```
   RANKS   =: 1 : 'u b. 0'
   LRANK   =: 1 : '1 { (u RANKS)'   NB. left rank only
```

* RANKS	* LRANK
0 0 0	0

フォームのダイアディック動詞を定義する一般的なスキーム (U . V)です。

```
        u . v   means  u @ (v " (1+L, _))   where L
= (v LRANK)
```

または同等に、

```
            u . v   means (u @: v) " (1+L, _)
      and hence
           +/.*   means (+/ @: *)" 1 _
```

したがって、 . と @: の違いがわかります。単純ベクトル引数の場合は同じですが、引数の次元は同じでなければなりませんが、一般的な行列乗算に必要な条件ではありません。(上記の例では) A の各行は B 全体です。

22.5 決定子 [Determinant]

モナド動詞(- / . *)は行列の行列式を計算します。

```
   det =: - / . *
```

M	det M	(3*3)-(2*4)
3 4 2 3	1	1

記号的に:

```
   sdet =: sminus / . sprodc
```

S	sdet S
+-+-+ |a|b| +-+-+ |c|d| +-+-+ |e|f| +-+-+	+---------------------------+ |(a(d-f))-((c(b-f))-(e(b-d)))| +---------------------------+

22.5.1 特異行列 [Singular Matrices]

行列は、行(または列)が線形独立でない場合、つまり、ある行(または列)が定数を掛けて別の行列から取得できるかどうかが特異であると言われます。特異行列は行列式がゼロである。

以下の例では、A は(記号的な)特異行列であり、m は定数の乗数です。

A =: 2 2 $ 'a';'b';'ma';'mb'	sdet A
+--+--+ |a |b | +--+--+ |ma|mb| +--+--+	+-------+ |amb-mab| +-------+

結果の項(amb-mab)はすべての a、b、および m についてゼロでなければならないことがわかります。

22.6 行列除算 [Matrix Divide]

22.6.1 連立方程式 [Simultaneous Equations]

組み込みの動詞 %. (パーセントドット)を「マトリックス分割」と呼びます。これは、連立一次方程式系の解を見つけるために使用できます。例えば、従来のように書かれた方程式を考える:

```
3x + 4y = 11
2x + 3y =  8
```

私たちは非公式に、行列方程式として書き直すと、

```
M dot U = R
```

ここで、M は、係数の行列であり 、U は、未知数のベクトルであり、Y、X および R は、右辺値のベクトルです。

M =: 3 4 ,: 2 3	R =: 11 8
3 4 2 3	11 8

未知数 U(すなわち、x、y)のベクトルは、R を行列 M で割ることによって求めることができる。

M	R	U =: R %. M	M dot U
3 4 2 3	11 8	1 2	11 8

M ドット U が R に等しいこと、すなわち U が方程式を解くことが 分かります。

22.6.2 複素数-ベクトル [Complex, Rational and Vector Variables]

解くべき方程式は複雑な変数であってもよい。例えば:

M	R =: 15j22 11j16	U =: R%。M	M dot U
3 4 2 3	15j22 11j16	1j2 3j4	15j22 11j16

または合理的に。この場合 、合理的な結果を出すには、M と R の両方が合理的でなければならない。

```
M =: 2 2 $ 3x 4x 2x 3x
R =: 15r22 11r16
```

M	R	U =: R%。M	M dot U
3 4 2 3	15r22 11r16	_31r44 123r176	15r22 11r16

前の例では、U の未知数はスカラでした。今、未知数がベクトルであり、我々の解決方程式が次のようになっているとします。

```
3x + 4y = 15 22
2x + 3y = 11 16
```

だから我々は書く：

```
M =: 2 2 $ 3 4 2 3
R =: 2 2 $ 15 22 11 16
```

M	R	U =: R%。M	M dot U
3 4 2 3	15 22 11 16	1 2 3 4	15 22 11 16

未知数 x と y は、U の行、つまりベクトルです。

22.6.3 カーブフィッティング [Curve Fitting]

一連のデータ点に最も適した直線をプロットすることを目指すとします。データポイントが x、y ペアの場合は、次のようになります。

```
x =: 10 20 30
y =: 31 49 70
```

次式の a と b を見つけることを目的とします。

```
y = a + bx
```

3 つのデータポイントは、2 つの未知数 a および b に 3 つの方程式を与える。

```
1 . a + 10 . b = 31
1 . a + 20 . b = 49
1 . a + 30 . b = 70
```

したがって、係数 M の行列を得る。

```
M =: 3 2 $ 1 10 1 20 1 30
```

行列 M を y を除して、未知数 U（すなわち、a、b）のベクトルを得る。

M	y	U =: y %. M	M dot U
1 10 1 20 1 30	31 49 70	11 1.95	30.5 50 69.5

ここでは未知数よりも多くの方程式があります(M の列よりも行が多い)ので、解 U はすべての方程式に最適です。M and U は、y に近いが、正確には y ではないことが分かる。
「最良適合」は、誤差の二乗の和が最小になることを意味し、誤差は y - M dot U によって与えられる。平方和が最小になると、U をわずかに摂動させることにより、平方和が増加することが予想される。

+/ *: y - M dot U	+/ *: y - M dot (U + 0.01)
1.5	1.6523

この方法は、多項式をデータ点の集合に適合させることに直接的に拡張される。 我々が合うことを目指すと仮定しよう

$$y = a + bx + cx^2$$

データポイント:

```
   x =: 0   1  2  3
   y =: 1   6 17 34.1
```

解決すべき 4 つの方程式は次のとおりです。

```
      1.a + bx0 + cx02 = y0
      1 a + bx1 + cx12 = y1
      1.a + bx2 + cx22 = y2
      1.a + bx3 + cx32 = y3
```

行列 M の列は（1, x, x ^ 2）であり、便宜的には（x ^ / 0 1 2）で与えられる。

```
   M =: x ^ / 0 1 2
```

未知数 a、b、c は次のようにベクトル U によって与えられる。

M	y	U =: y %. M	M dot U
1 0 0 1 1 1	1 6 17 34.1	1.005 1.955 3.025	1.005 5.985 17.02 34.09

1 2 4 1 3 9			

この例が示すように、未知数よりも多くの方程式が存在する可能性がありますが、明らかに少ないことはありません。すなわち、R %. M である。行列 M は行より多くの列を持たなければならない。

22.6.4 上位配列による除算 [Dividing by Higher-Rank Arrays]
ここでは、U =: R%の例を示します。M であり、除数 M はランク 3 である。

```
   M =: 3 2 2 $ 3 4 2 3 0 3 1 2 3 1 2 3
   R =: 21 42
```

M	R	U =: R %. M	M dot U	M dot"2 1 U
3 4 2 3 0 3 1 2 3 1 2 3	21 42	_105 84 28 7 3 12	error	21 42 21 42 21 42

dyadic ランク %. は_2、

```
   %. b. 0
2 _ 2
```

したがって、この例では、R の全体が M の 3 つの行列のそれぞれと別々に結合されます。つまり、3 つの別々の方程式セットがあり、それぞれが同じ右辺 R を持ちます。したがって、3 つの別々のソリューション(U の行)があります。
条件 R = M ドット U は明らかに保持されない(M の最後の次元は U の最初の次元と等しくないため)が、U の対応する行を有する M の各行列に対して別々に保持する。

22.7 アイデンティティ行列 [Identity Matrix]
それ自体で分割された（非特異的な）正方行列 M は、（M dot I）= M となるような「恒等行列(単位行列)」を生成する。

```
   M =: 3 3 $ 3 4 7 0 0 4 6 0 3
```

M	I =: M %. M	M dot I
3 4 7 0 0 4 6 0 3	1 0 0 0 1 0 0 0 1	3 4 7 0 0 4 6 0 3

22.8 行列の逆行列 [Matrix Inverse]
モナド動詞 **%.** は行列の逆行列を計算します。つまり、**%.** M は適切な単位行列 **I** に対して **I %. M** と等価です。

M	I =: M %. M	I %. M	%. M
3 4 7 0 0 4 6 0 3	1 0 0 0 1 0 0 0 1	0 _0.125 0.1667 0.25 _0.3438 _0.125 0 0.25 0	0 _0.125 0.1667 0.25 _0.3438 _0.125 0 0.25 0

ベクトル V の場合、逆 W は逆数の大きさと同じ方向を持ちます。従って、大きさの積は 1 であり、その間の角度の余弦は 1 です。

V	W =: %. V	(ma V) * (ma W)	V ca W
3 5	0.08824 0.1471	1	1

第 22 章終了

第 23 章: 微積分 [Calculus]

この章では、差別化と統合のための J 演算子について説明します。
それは以下をカバーします、
•分析的に差異化して統合する結合 d. は、関数を表す式を関数を表す式に変換します。
•D. は数値的に区別されるため、微分可能な関数の範囲を広げます。 また、偏微分も扱います。
•数値積分機能を備えたライブラリスクリプト。

23.1 差別化 [Differentiation]

組み込みの d.（小文字の d dot）があります。 その左の引数は、差別化される関数です。 その右の引数は、1 次微分が必要な場合は 1、2 次微分の場合は 2 などです。 "キューブ" 関数 ^&3 の 1 次導関数は "3 倍の正方形"です。

```
   ^&3  d. 1
3&*@(^&2)
```

一般的なスキームは、e が関数を表す式である場合、e d. n は、e の n 番目の導関数を表す式です。キューブ関数を多項式 0 0 0 1 & p. として表現するもう一つの例があります。

```
   0 0 0 1 &p. d. 1
0 0 3&p.
```

動詞キューブを定義するとします。

cube =: 0 0 0 1 & p.	cube 2
0 0 0 1&p.	8

d との区別。そのデリバティブが期待通り 3 倍の正方形であることがわかりますが、デリバティブの表現はそれほど有益ではありません。

(cube d. 1) 2	cube d. 1
12	cube d.1

その理由は、キューブは動詞を示す名前であり、そのような名前は動詞が適用されるまで一般的に評価されないからです。

(付録 1 を参照してください)。キューブの派生物を検査したい場合は、副詞 f を適用して名前キューブの評価を強制することができます。

cube d. 1	(cube f.) d. 1
cube d.1	0 0 3&p.

あるいは、f を適用して、微係数の式の評価を強制することもできます。

cube d. 1	(cube d. 1) f.
cube d.1	0 0 3&p.

23.2 積分 [Integration]

右の引数_1 の場合、結合 d. は左の引数を統合します。 「3 倍の正方形」の積分は「立方体」です。

```
   0 0 3 & p. d. _1
0 0 0 1&p.
```

23.3 d.のドメイン [The Domain of d.]

d. と微分可能または積分可能な関数は、まずスカラーでなければなりません。 つまり、スカラー引数を取ってスカラー結果を提供しなければならず、すべての中間数量はスカラーでなければなりません。 ここに例があります。 関数 "(x-1)*(x-2)"はいくつかの異なる方法で書くことができます。 ここに 2 つあります：

f =: -&1 * -&2	g =: (*/) @: (- & 1 2)	f 3	g 3
-&1 * -&2	*/@:(-&1 2)	2	2

f はスカラであり、d. の領域にある。しかし、g はベクトルである中間量 x - 1 2 を形成するのでスカラーではない。したがって、g は d. の領域にはない。これを実証するために、我々はデリバティブの評価を強制する。

(f d. 1) f.	(g d. 1) f.
_3 2x&p.	error

第二に、 d. は定数関数、多項式、指数関数、x、積分関数、x ^ n を区別することができ ます。

3: d. 1	0 2 &p. d. 1	^ d. 1	^&4 d. 1
0"0	2"0	^	4&*@(^&3)

f と g が d と微分 可能であれば フォーク(f + g)、(fg)、(f * g)、f%g も同様です。

f =: ^&3	g =: 0 2 & p.	((f + g) d. 1) f.
^&3	0 2&p.	2 0 3x&p.

三角関数は d. で微分可能です。 フォーク(sin + cos)の導関数は(cos - sin)です。

sin =: 1&o.	cos =: 2&o.	(sin + cos) f. d. 1
1&o.	2&o.	2&o. + -@(1&o.)

f と g が d. と微分 可能であれば 、そして組成は fg および fg: g です。

f	g	(f @ g d. 1) f.
^&3	0 2&p.	0 0 24x&p.

23.4 D.接合詞 [The Conjunction D.]

結合D.(大文字のD dot)は導関数を計算します。それはd. と2 つの点で異なります。

- ●数値的方法を使用することによって、任意の関数を区別することができます。つまり、d. のドメインに限定されません。
- ●スカラー関数に限定されず、ベクトル引数を持つ関数を微分して偏微分を生成することができます。

23.4.1 D. のドメイン [The Domain of D.]

D.は数値的方法を使用することができる ので、その引数は任意の関数とすることができる。例えば、上の関数 g を思い出して、"(x-1)*(x-2)"を計算すると、d のドメイン外である

ことが証明されました。しかし、それは D. の領域内にあります。その導関数は「2x-3」です。

g =: (*/) @: (- & 1 2)	(g d. 1) 3	(g D. 1) 3
*/@:(-&1 2)	error	3

23.4.2 D.部分的デリバティブ [Partial Derivatives with D.]

次に、ベクトル引数からスカラーを計算する関数を見てみましょう。例えば、点(x、y)の高さが

```
(sin x) * (cos y)
```

ベクトル引数(x、y)を持つ高さ関数は次のように書かれます:

```
h =: (sin @ {.) * (cos @ {:)
```

式(h D.1)(x、y)は、点(x、y)における関数 h の 2 つの勾配の x 方向および y 方向の数値を計算します。

x=: 0.4	y =: 0.5	p =: h D. 1 x,y
0.4	0.5	0.8083071 _0.1866971

結果 p は、x および y に関して偏微分の値を与える。
この結果を確認することができます。線 y = 0.5 に沿った高さについて、関数 q say を定義すると仮定します。この結果を確認することができます。 線 y = 0.5 に沿った高さについて、関数 q say を定義すると仮定します。 q（x）を h（x、0.5）にして、

```
q =: h @: (, & 0.5)
```

ここで考えられるのは、引数 x に適用される q の導関数は、x での h の最初の偏微分と同じでなければならないということです。

h D.1 x,y	q D. 1 x
0.8083071 _0.1866971	0.8083071

今度はベクトルからベクトルを計算する関数の偏微分を見て

みましょう。ここでは、3 空間の点(x、y、z)を 2 空間の点(2x、3y)に変換する関数の例を示します。

foo =: (2 3 & *) @: (1 1 0 & #)	foo 1 1 1
2 3&*@:(1 1 0&#)	2 3

一般に、このような関数は長さ m の引数ベクトルをとり、長さ n の結果ベクトルを生成します。したがって、引数の各要素に関して結果の各要素に 1 つずつ、m * n の偏微分が存在します。点 xyz = 1 1 1 の foo の 6 つの偏微分は次のように表されます。

pd =: foo D 1	pd 1 1 1
foo D.1	2 0 0 3 0 0

スカラーからスカラーを生成するキューブの ような関数を考えてみましょう。ベクトル引数を指定すると、同じ長さのベクトル結果が生成され、結果の要素は引数の対応する要素のみに依存します。

cube	cube a =: 1 2 3
0 0 0 1&p.	1 8 27

したがって、スカラー関数の場合、引数の対応する要素に関する結果の要素に対するものを除き、すべての偏微分はゼロです。

pd =: cube D. 1	pd 2 3 4
cube D.1	12 0 0 0 27 0 0 0 48

スカラ関数が完全に評価された形で与えられ、d の領域にある場合 、D.の組み合わせは、偏微分関数の解析的表現を生成します。

PD =: (0 0 0 1 & p.) D.1	PD 2 3 4
(* =/~@(i.@$))@:(0 0 3&p.)	12 0 0 0 27 0 0 0 48

23.5 数値積分 [Numerical Integration]
ライブラリスクリプト・ファイルがあり integrat.ijs が。統合のためにいくつかの異なる演算子が含まれています。ドキュメンテーションはスクリプトファイルに記載されています。
JSoftware の Web サイトからダウンロードできます。ここには integat.ijs への リンクがあります
たとえば C:\temp のようなディレクトリにダウンロードしたと仮定して、それを J セッションにロードします:

```
load 'c:¥temp¥integrat.ijs'
```

提供される統合演算子の 1 つは、結合 適応(適応積分による「数値積分」)です。式 f adapt(L、U)は、限界 L と U の間の f の定積分の数値を計算します。例えば、0 と 1 の間の 3 &*の積分は 1.5 であると予想します。

f =: 3&*	f adapt 0 1
3&*	1.5

第 23 章終了

第 24 章: 名前とロケール [Names and Locales]

この章では、ロケールを見ていきます。ロケールの関心は 2 つあります。1 つは大きなプログラムを編成する方法で、もう 1 つは次の章で説明するように、J のオブジェクト指向プログラミングの基礎となるものです。

24.1 背景 [Background]

できるだけ多くのプログラムが、互いに独立して独立したいくつかの部分で最適に開発されることは、一般的には合意されています。たとえば、大きなプログラムの独立した部分は、独自のスクリプトファイルを持つ統計関数の集合である可能性があります。
独立したスクリプトで定義されているものについては、他のスクリプトでどのような名前が定義されているかに関わらず、自由に名前を選択することが期待されます。明らかに、独立したスクリプトを組み合わせる際に問題が生じる可能性があります。同じ名前が異なるスクリプトで異なる定義を誤って受け取った場合はどうでしょうか？「ロケール」の J 施設は、この問題に対処する方法を提供します。

24.2 ロケールとは [What are Locales?]

フォーム(name =: something)の割り当てを入力すると、名前 の定義があります。すべての定義は、 "ロケール"と呼ばれる J システムのメモリのある領域に格納されます。大まかに言えば、ロケールはディレクトリがファイルになるための定義です。ロケールの重要な機能は次のとおりです。

- 同時にいくつかの異なるロケールが存在する可能性があります。
- 各ロケールには一連の定義が格納されます。
- 同じ名前は、異なる定義を持つ異なるロケールで同時に発生する可能性があります。

したがって、「ロケール L で定義された名前 N」という形式の名前は、矛盾なく一意的に定義されます。このような名前は、N_L_(N アンダーバーL アンダーバー)として記述することができ、「場所名」と呼ばれます。最後に
いつでも、現在のロケールは 1 つだけです。現在のロケールは、その定義がすぐに使用できるものです。

したがって、普通の名前 N は、通常、現在のロケールで定義された" N "を意味します。
ロケールは名詞、動詞、副詞、接続詞のいずれでもありません。つまり、ロケールは変数に代入することも、関数の引数として渡すこともできません。彼らは名前を持っていますが、名前でロケールを参照する必要があるときはいつでも、特別な構文形式(N_L_などのロケータ名)か、文字列を形成するために名前を引用することによって行います。

24.3 例 [Example]

ウィスキーの価格と一般的な従業員給与との相関関係に興味があるとします。我々は独立した起源の、経済的なデータと統計的な機能を備えた 2 つのスクリプトが利用可能であると仮定しよう。これらのスクリプトファイルは、次のように作成されている可能性があります。

```
   (0 : 0) (1 !: 2) < 'economic.ijs'
y  =: 1932  1934  1957  1969  1972   NB. years
s  =: 1000  1000  3000  9000 11000   NB. salaries
p  =: 1.59  1.68  2.00  4.50  5.59   NB. prices
)

   (0 : 0) (1 !: 2) < 'statfns.ijs'
m =: +/ % #          NB.  Mean
n =: - m             NB.  Norm
v =: m @: *: @: n    NB.  Variance
s =: %: @: v         NB.  Standard Deviation
c =: m @: (*&n)      NB.  Covariance
r =: c % (*&s)       NB.  Correlation Coefficient
)
```

この 2 つのスクリプトを読み込んで、価格 p と給与 s との相関係数を式(prs)の値として計算したいと考えています。
残念ながら、名前 s は異なるスクリプトの中で異なることを意味することがわかります。両方のスクリプトを同じロケールにロードする場合、s の定義はもう一方のスクリプトを上書きします。対処法は、2 つのスクリプトを異なるロケールにロードすることです。
常に base という名前のロケールがあり、デフォルトではこれは通常現在のものです。組み込み動詞(0!:0)を使用して、現在のロケール(base)に economic.ijs をロードします。

```
   (0 !: 0) < 'economic.ijs'
```

次に、statfns.ijs を呼び出したい別のロケール、たとえば stat にロードします。これを最小限の新しい装置で行うために、組み込み動詞(18！： 4)を使用することができます。

```
(18 !: 4) < 'stat'
(0 !: 0)  < 'statfns.ijs'
(18 !: 4) < 'base'
```

最初の行は stat ロケールを作成し、それを最新のものにします。2 行目は、現在のロケール stat に statfns.ijs をロードし ます。3 行目は、現在の状態を復元するために、基本ロケールを再度電流にします。
現時点では、データ変数 s と p は現在のベースにあるため利用可能です。相関係数関数 r は、現行ではない stat にあるため、まだ使用できません 。次のステップは、r_stat_という名前のロケータ形式を使用して 、相関係数関数を r -as-in-locale- stat に定義することです

```
corr =: r_stat_
```

corr は base で定義されている ため(ベースが現在のため)、使用可能です。私たちに必要なものはすべて利用可能です。給与と価格との相関を計算することができます。

s corr p	p corr s	p corr p
0.993816	0.993816	1

24.3.1 レビュー [Review]

私たちが見てきたことは、独立したスクリプト間の名前の衝突を解消するために、場所の名前を使用することです。それが取ったのは、比較的わずかなアドホックな定義でした。
この小さな例では、競合は容易に特定され、スクリプトの 1 つを編集することによって容易に修正することができました。しかし、重要な点は、独立したスクリプトを改ざんして一緒に働かせることを避けることです。

24.4 現在のロケール [The Current Locale]

いくつかのロケールが共存するかもしれませんが、いつでも 1 つだけが最新です。現在のロケールの名前を示すビルトイ

ンの動詞(18！： 5)があります。

```
   (18 !: 5) ''  NB. show name of current locale
+----+
|base|
+----+
```

現在のロケールの重要性は、単純な名前が評価されるのは現在のロケールにあるということです。

```
   s
1000 1000 3000 9000 11000
```

ロケール stat にロードされた statfns.ijs の s の値ではなく、base にロードされた script economic.ijs に定義されている s の値を取得することに注目してください。
新しい定義が格納されている現在のロケールです。現在のロケールでどの名前が定義されているかを見るには、引数が 0 1 2 3 の組み込み動詞(4！： 1)を使用できます。結果として得られる長い名前のリストは、library-verb リストで便利に表示できます。

```
   list (4 !: 1) 0 1 2 3  NB. show all names in current
locale
ASSERTING  CH          IP          RUN         RUNR
TD        THIS
and       cd        corr      dir       drop      e
first
fst        hello     implies   indexfile indexing
is_bool   is_box
is_char   is_cmplx    is_extint  is_float    is_int
is_list   is_number
is_rat      is_real     is_scalar  is_string  last
most      not
p         print     pwd       rest      run       s
snd
take      thd       y
```

新しい動詞を定義し、その名前がリストに表示されるのを見ることができます:

```
   foo  =: +/
   list (4 !: 1) 0 1 2 3
ASSERTING  CH          IP          RUN         RUNR
TD        THIS
and       cd        corr      dir       drop      e
first
```

```
foo       fst       hello     implies   indexfile
indexing  is_bool
is_box    is_char    is_cmplx  is_extint  is_float
is_int    is_list
is_number is_rat     is_real   is_scalar is_string
last      most
not         p           para        print       pwd
rest      run
s         snd       take      thd       y
```

上記のように、組み込み動詞(18！： 4)で現在のロケールを変更することができます 。私たちは stat ロケールを現在のものにすることができます:

```
   (18 !: 4) < 'stat'
```

これで、このロケールでどの名前が定義されているかを確認できます。

```
    (4 !: 1) 0 1 2 3
+-+-+-+-+-+-+
|c|m|n|r|s|v|
+-+-+-+-+-+-+
```

再度 ベースに 戻る

```
   (18 !: 4) < 'base'
```

24.5 z ロケールは特別です [The z Locale Is Special]

z という名前のロケールは、次の意味で特殊です。名前が評価され、その名前の定義が現在のロケールに存在しない場合、z ロケールはその名前を自動的に検索されます。ここに例があります。例えば、変数 q の定義をロケール z に入れます。

```
   (18 !: 4) < 'z'
   q =: 99
   (18 !: 4) < 'base'
```

ここで、q という名前が現在のロケール(ベース)に存在しないことがわかります。

```
   list (4 !: 1) 0 1 2 3
ASSERTING CH          IP          RUN         RUNR
TD        THIS
and       cd        corr      dir       drop      e
first
foo       fst       hello     implies   indexfile
indexing  is_bool
```

```
is_box    is_char    is_cmplx   is_extint  is_float
is_int    is_list
is_number is_rat     is_real    is_scalar  is_string
last      most
not         p            para         print       pwd
rest      run
s         snd       take       thd        y
```

それにもかかわらず、簡単な名前 q を評価してロケール z で定義された値を得ることができます。

```
   q
99
```

基底にない z のものを 見つけることができるので、ロケール z は一般的な有用性の機能の自然な家です。J セッションを開始すると、ロケール z には事前定義された便利なライブラリ関数の集まりが自動的に設定されます。
z ロケール のこれらの関数の名前はすべてすぐに使用できますが、100 以上の名前は基本ロケールを混乱させません。(18！： 4) や(4！： 1)などの組み込み動詞の使用を上回っています。しかし、ロケール z のライブラリ動詞はしばしばより便利です。ここには小さな選択肢があります:

coname ''	show name of current locale
conl 0	show names of all locales
names ''	show all names in current locale
nl ''	show all names in current locale (as a boxed list)
cocurrent 'foo'	locale foo becomes current
clear 'foo'	remove all defns from locale foo
coerase 'A';'B';'C'	erase locales A B and C

現在のロケールで名前が見つからない場合、検索は z ロケールに自動的に進みます。言い換えれば、各ロケールから z ロ

ケールへの「パス」と呼ばれるものがあります。私たちは、以下のパスのテーマに戻ります。

24.6 式の評価 [Locative and the Evaluation of Expressions]

24.6.1 割り当て [Assignments]

フォーム n_L_ =: の割り当ては、何かの値をロケール L の名前 n に代入します。ロケール L がまだ存在しない場合は作成されます。例えば:

```
   n_L_ =: 7
```

ロケール L に 名前 n を値 7 で作成します。この時点で、ロケール内のすべての定義を表示するユーティリティ関数を導入すると便利です。このビュー関数をロケール z に入れます :

```
   VIEW_z_ =: 3 : '(> ,. ('' =: ''&,)@:(5!:5)"0) nl
''''''
   view_z_ =: 3 : 'VIEW__lo '''' [ lo =. < y'
```

いくつかの定義を追加した場合:

```
   k_L_ =: 0
   n_M_ =: 2
```

私たちはロケール L と M で何を持っているかを調べることができます:

view 'L'	view 'M'
k =: 0 n =: 7	n =: 2

代入のテーマに戻ると、現在のロケールが L ならば (foo_M_ =: something) は以下を意味します:

```
evaluate something in locale L to get value V say.
cocurrent 'M'
foo =: V
cocurrent 'L'
```

例えば:

```
   cocurrent 'L'
```

view 'L'	view 'M'	k_M_ =: n-1	view 'M'
k =: 0 n =: 7	n =: 2	6	k =: 6 n =: 2

24.6.2 名前の評価 [Evaluating Names]

ここでは、式で発生する場所の名前を調べます。スキーム(このスキーム 2 を呼ぶ)は、現在のロケールが L ならば(n_M_)は

1. cocurrent 'M'
2. evaluate the name n to get a value V
3. cocurrent 'L'
4. V

For example:

```
cocurrent 'L'
```

view 'L'	view 'M'	n_M_
k =: 0 n =: 7	k =: 6 n =: 2	2

24.6.3 動詞の適用 [Applying Verbs]

式(f_M_n)を考える。これは、次のことを意味します。(現在のロケールで定義されているように)引数 n に適用される関数 f(ロケール M で定義されたもの)この場合、f から n のアプリケーションは ロケール M で行われます。次に例を示します。

```
u_M_ =: 3 : 'y+k'
cocurrent 'L'
```

view 'L'	view 'M'	u_M_ n
k =: 0 n =: 7	k =: 6 n =: 2 u =: 3 : 'y+k'	13

引数 n は現在のロケール L から来るが、定数 k は動詞 u を取ったロケール(M)から来る ことがわかる。スキーム(スキーム 3 と呼ぶ)は次のとおりです。現在のロケールが L の場合、(f_M_ something) は以下を意味します。

```
1.evaluate something in L to get a value V say
2.cocurrent 'M'
3.in locale M, evaluate the expression f V to get
4.a value R say
5.cocurrent 'L'
6.R
```

別の例があります。動詞 g は、f が見つかった同じロケールから取得されます。

```
   g_L_ =: +&1
   g_M_ =: +&2
   f_M_ =: g

   cocurrent 'L'
```

view 'L'	view 'M'	f_M_ k
g =: +&1 k =: 0 n =: 7	f =: g g =: +&2 k =: 6 n =: 2 u =: 3 : 'y+k'	2

24.6.4 副詞の適用 [Applying Adverbs]

仮定 A_X_はロケールでの副詞である X。引数への A_X_の適用は、現在のロケールではなくロケール X で行われます。これを実証するために、新しいロケール X と Y に定義を入力することから始め ます。

```
    A_X_ =: 1 : 'u & k'    NB. an adverb
    k_X_ =: 17
    k_Y_ =: 6
```

現在 Y を現在のロケールにする:

```
    cocurrent 'Y'
```

引数+に 副詞 A_X_を適用します。

view 'X'	view 'Y'	+ A_X_
A =: 1 : 'u & k' k =: 17	k =: 6	+&17

明らかに、結果は、Y である現在のロケールからではなく、 ロケール X から k を取ることによって生成される。
スキームは、現在のロケールが Y で、A が副詞である場合、式 f A_X_は以下を意味します。

```
1.evaluate f in locale Y to get a value F say.
2.cocurrent X
3.evaluate F A in locale X to get a result G say.
4.cocurrent Y
5.G
```

24.7 パス [Paths]

z ロケールには便利な "ライブラリ"関数が含まれており、ロケールから z までのパスがあることを 思い出してください。ライブラリ動詞 copath でロケールからのパスを調べることができます。ロケールベースからのパスがちょうど z であることが期待されます。

```
   copath 'base'   NB. show path
+-+
|z|
+-+
```

パスは、ボックス化された文字列(のリスト)として表されます。ロケール間で独自の検索パス構造を構築することができます。私たちは、二項コパスを使って、基底に stat と z のパスを 与えます。

```
   ('stat';'z') copath 'base'
```

結果が期待どおりであることを確認します。

```
   copath 'base'
+----+-+
|stat|z|
+----+-+
```

このパスを適切な位置に置くことで、基底 が最新である間にbase、stat 、z の名前を見つけることができます。

```
    cocurrent 'base'

    s     NB. in base
1000 1000 3000 9000 11000

    r     NB. in stat
c % *&s

    q     NB. in z
99
```

L から M へのパスを設定するとします。私たちはすべてのパスをロケール z で終わらせたいと思うことに注意してください。そうしないと、z の便利なものにアクセスできなくなるので、パスを L から M から z にします。

```
    ('M';'z') copath 'L'
```

パスに沿って名前にアクセスすると、現在のロケールは変更されません。 位置名を使って動詞 u を参照し、パスに沿って u を検索することの効果を比較する。

```
    cocurrent 'L'
```

view 'L'	view 'M'	u_M_ 0	u 0
g =: +&1 k =: 0 n =: 7	f =: g g =: +&2 k =: 6 n =: 2 u =: 3 : 'y+k'	6	0

評価するとき(u_M_0)には、M へのロケールの変更があるので、変数 k は M の値を 6 として取り上げられること がわかります。評価する(u 0)では、u がパスに沿って見つかると、変数 k は現在のロケールから取り出され、L の値は 0 になります。名前がパスに沿って見つかると、定義が一時的に現在のロケールにコピーされたようになります。別の例があります。

view 'L'	view 'M'	f_M_ 0	f 0

g =: +&1 k =: 0 n =: 7	f =: g g =: +&2 k =: 6 n =: 2 u =: 3 : 'y+k'	2	1

24.8 ロケーションとパスの結合 [Combining Locatives and Paths]

スクリプトファイルの定義をロケールに取り込みたいことがあります。これを行うには、上記の 3 つのステップがあります。
(1)cocurrent(または 18！: 4)を使用して、指定されたロケールに移動します。
(2)スクリプトファイルを 0 !: 0 で実行する
(3)cocurrent(または 18！: 4)を使用して元のロケールに戻ります。
もっとうまくいく方法は次の通りです。最初に定義する:

```
POP_z_ =: 0 !: 0
```

そして、ロケール Q を 、economic.ijs ファイルから読み込むと、次のように書くことができます。

```
POP_Q_ < 'economic.ijs'
```

これは次のように動作します:

POP の動詞は、ロケールで定義されている z 。
POP_Q_ <と出会うと、システムはロケール Q を一時的に現在のものにし、存在しなければ Q を作成します。
システムは POP の定義を探します。POP はもちろんロケール z で定義されている ため、Q ではそれが見つかりません。
次に、システムは Q から z への経路を調べ、POP を見つける。
現在のロケールはまだ(一時的に)Q であることに注意してください。
POP の動詞は一時的に、現在のロケールでは、その引数に適用される Q 。
現在のロケールは、それが以前にあったものに自動的に復元されます。

ベースに戻る。(ロケール間を挟んでいる場所を把握することをお勧めします)。

```
   cocurrent <'base'
```

24.9 間接的な場所 [Indirect Locatives]

変数は、ロケールの名前を囲み文字列として保持できます。例えば、M がロケールであるとすると、

```
   loc =: < 'M'
```

次に、k_M_のようなロケータ名は、等価的に k__loc（u はアンダースコアのアンダースコア）という形式で書くことができます。

```
   k_M_
6

   k__loc
6
```

この間接的な形式のポイントは、関数への引数としてロケール名を提供するのが便利であるということです。

```
   NAMES =: 3 : 0
locname =. < y
names__locname ''
)

   NAMES 'L'
g k n
```

24.10 名前の消去 [rasing Names from Locales]

var という名前の変数を作成するとします。たとえば、

```
   var =: 'hello'
```

それが存在することを示します。つまり、var という名前は基本ロケールのネームリストの名前の 1 つです。

```
   (<'var') e. nl_base_ ''
1
```

これで消去ライブラリの動詞で消去できます:

```
   erase <'var'
1
```

それがもはや存在しないことを実証する

```
   (<'var') e. nl_base_ ''
0
```

ここでは、基本ロケールで変数 foo を作成し、z ロケールで foo とも呼ばれる変数 foo を作成するとします。ベースから z へのパスが常にあることを思い出してください

```
   foo    =: 'hello'
   foo_z_ =: 'goodbye'
```

我々はそれらが両方とも存在することを実証する:

```
   (<'foo') e. nl_base_ ''
1
   (<'foo') e. nl_z_ ''
1
```

ベースから foo を 消去して、それがなくなったことを証明しますが、z の foo はまだそこにあります:

erase <'foo'	(<'foo') e. nl_base_ ''	(<'foo') e. nl_z_ ''
1	0	1

ここで再び消去すると、foo はパスに沿って見つけられ、z から消去されます。

erase <'foo'	(<'foo') e. nl_base_ ''	(<'foo') e. nl_z_ ''
1	0	0

第 24 章終了

第 25 章オブジェクト指向 [Object-Oriented Programming]

25.1 背景と用語 [Background and Terminology]

この章では、「OOP」は「オブジェクト指向プログラミング」の略です。ここには OOP のサムネイルスケッチがあります。
場合によっては、プログラムは関連するデータの集まりを構築し、維持し、使用する必要があります。コレクションは、ある意味では全体的なものであると考えるのが自然です。例えば、「スタック」は一連のデータ項目であり、最も最近追加された項目が最初に削除される項目である。スタックを大量に使うつもりなら、スタックの構築と使用に特化した関数を書く価値があるかもしれません。
いくつかのデータといくつかの専用機能の組み合わせをオブジェクトと呼びます。すべてのオブジェクトは、類似のオブジェクトの特定のクラスに属します。スタックは Stack クラスのオブジェクトであると言います。
与えられたクラスのオブジェクトの専用関数は、そのクラスの「メソッド」と呼ばれます。例えば、Stack クラスのオブジェクトの場合、新しいアイテムを追加するメソッドと、最後に追加されたアイテムを取得するメソッドが必要になります。
オブジェクトは、そのデータを表すために 1 つ以上の変数を必要とします。このような変数はフィールドと呼ばれます。したがって、スタックの場合、アイテムのリストである単一のフィールドを持つことを選択できます。
要約すると、OOP は、オブジェクトの有用なクラスを識別し、メソッドとフィールドを定義してクラスを定義し、そのメソッドを使用することで構成されます。
異なるクラスの定義にプログラムを編成することで、OOP は複雑さを管理する方法と見ることができます。以下の簡単な例は、OOP アプローチの機構を説明することを意図していますが、OOP の動機づけではあまり提供しません。
私たちは、多くのライブラリ関数を使用します。ほとんどの場合、名前は "co"で始まり、 "クラスとオブジェクト"を意味します。この章の最後に簡単な概要を示します。

25.2 クラスの定義 [Defining a Class]

25.2.1 クラスの紹介 [Introducing the Class]

簡単な例として、「スタック」と呼ばれるクラスを定義します。ライブラリ関数 coclass とともに新しいクラスが導入されました。

```
coclass.
   coclass 'Stack'
```

その効果のためにコクラスが使用され、その結果は使用されない。コクラスの効果は、スタック と呼ばれる新しいロケールを確立し、現在のものにすることです。これを確認するために、現在のロケールの名前を調べることができます:

```
   coname ''
+-----+
|Stack|
+-----+
```

25.2.2 メソッドの定義 [Defining the Methods]

新しい物体が 2 つのステップで生まれます。最初のステップでは、ライブラリの動詞集 を使用して、フィールドのない基本的なオブジェクトを作成します。単なるプレースホルダです。2 番目のステップでは、フィールド変数に値を作成して割り当てることによって、新しいオブジェクトの構造と初期内容を提供します。

私たちは以下の最初のステップを扱います。私たちが今見ている第二のステップ。これは、従来作成と呼ばれる方法 (「オブジェクトの作成」ではなく「フィールドの作成」を意味する)によって行われます。これは私たちが定義しなければならない最初の方法です。

たとえば、Stack オブジェクトは items という単一のフィールドを持ち、最初は空のリストであると判断します。

```
   create =: 3 : 'items =: 0 $ 0'
```

このメソッドと Stack クラスの 間の接続は、現在のロケールで作成が定義されたことだけです。これは Stack です。

この作成メソッドは動詞です。この例では、その引数を無視し、その結果は重要ではありません。その効果のために純粋に実行されます。その効果は、(暗黙的に指定された)オブジ

ェクトが 空リストとして items という単一のフィールドを持つように設定されることになります。
私たちの 2 番目の方法は 、スタック内の項目の前に新しい値をプッシュすることです。

```
    push =: 3 : '# items =: (< y) , items'
```

プッシュ方式は動詞です。引数 y はプッシュされる新しい値です。私たちはここで設計決定
を下しました.Y はボクシングされ、その後プッシュされます。結果には関心はありませんが、結果があるはずですので、単なる商品ではなく(: #個の商品)を返却することにしました 。
次に、スタック上の「最上位」(最近追加された)アイテムを検査する方法。その項目の値を返します。スタックは変更されません。

```
    top =: 3 : '> {. items'
```

次に、スタックの一番上の項目を削除するメソッド。

```
    pop =: 3 : '#  items =: }. items'
```

最後に、Stack オブジェクトを "破壊"するメソッドです。つまり、終了したらスタックオブジェクトを削除します。この目的のために、ライブラリ関数 codestroy があります。

```
    destroy =: codestroy
```

これで、Stack クラスの定義が完了しました。上記のコクラスの 'Stack'ステートメントの範囲内にあるので、現在のロケールは Stack です。このクラス定義を使用するには、通常の作業環境である基本ロケールに戻ります。

```
    cocurrent 'base'
```

25.3 新しいオブジェクトを作る [Making New Objects]

これで Stack オブジェクトを作成して使用できるようになりました。新しいスタックは 2 つのステップで作成されます。最初のステップでは、ライブラリ動詞集を使用します。

```
    S =: conew 'Stack'
```

S に割り当てられたキュー の結果は、新しく作成されたオブジェクト自体ではありません。むしろ、S の値は実際には新

しく作成された Stack オブジェクトを識別する一意の参照番号です。簡潔にするため、「スタック S」とは、S によって参照されるオブジェクトを意味します。
スタック S は現在存在しますが、その状態は未定義です。したがって、オブジェクトを作成する第 2 のステップは、create メソッドを使用して S の状態を空のスタックに変更することです。create は引数を無視するので、引数として 0 を指定します

```
   create__S 0
```

これで、スタック S に値をプッシュし、先入れ先出しの順序で値を取得できます。以下では、式(push__S 'hello'は、メソッド hello をオブジェクト S に適用したメソッドプッシュを意味します。

```
   push__S 'hello'
1
   push__S 'how are you?'
2
   push__S 'goodbye'
3
   pop__S 0
2
   top__S 0
how are you?
```

25.3.1 ダイアディック・カウリング [Dyadic Conew]

新しいオブジェクトの作成に関連する 2 つのステップ、つまり作成に 続いて作成された 2 つのステップは、ダイアディック・キューを使用して 1 つに縮小できます。スキームは次のとおりです。

```
o =: conew 'Class'
create__o arg
```

以下のように省略することができます。

```
o =: arg conew 'Class'
```

つまり、自動的に呼び出される、conew の左引数が create

に渡されます。この単純な Stack クラスでは、create はその引数を無視しますが、1 つのステップが 2 より小さい場合でもです。例えば:

```
   T =: 0 conew 'Stack'
   push__T 77
1
```

```
   push__T 88
2
   top__T 0
88
```

25.4 クラスとオブジェクトのリスト [Listing the Classes and Objects]

このセクションでは、作成したオブジェクトとクラスの人口を調べます。式(18！: 1)0 1 は、すべての既存のロケールのリストを生成します。

```
   (18!:1) 0 1
+-+-+-----+----+----+-+-------+--------+------+-----+-+
|0|1|Stack|base|ctag|j|jadetag|jcompare|jregex|jtask|z|
+-+-+-----+----+----+-+-------+--------+------+-----+-+
```

3 種類のロケールの名前がここにあります。第一に、第 24 章で説明した base、z などの通常のロケールがあります。これらは J システムによって自動的に作成されます。使用している J のバージョンによっては、ここに示したものとは異なるリストが表示されることがあります。
第二に、Stack などのロケールがあります。スタックロケールが定義するスタッククラスを。このロケールを表示すると(第 24 章の view ユーティリティ関数を使用して)

```
   view 'Stack'
IP      =: 1
create  =: 3 : 'items =: 0 $ 0'
destroy =: codestroy
pop     =: 3 : '#  items =: }. items'
push    =: 3 : '# items =: (< y) , items'
top     =: 3 : '> {. items'
```

私たちは可変 IP(自動的に作成された)と私たちがスタックに対して定義したメソッドを見ています。
第 3 に、0 などのロケールがあります。ここでは、名前は数字の文字列('0')です。そのようなロケールはオブジェクトです。変数 S の値は<'0'であるため、ここではオブジェクト S はロケール'0'になります。

```
S     view >S

+-+   COCREATOR =: <'base'
|0|   items     =: <;._1 '|how are you?|hello'
+-+
```

このロケールをオブジェクトとして識別する 変数 COCREATOR、およびオブジェクトのフィールドが表示されます。
オブジェクトからのパスは、動詞 18！： 2 によって与えられます。

```
   18!:2 S
+-----+-+
|Stack|z|
+-----+-+
```

S は Stack オブジェクトな ので、パス上の最初のロケールは Stack です。リコール第 24 章なお、以降 S = <「0」次いで発現 push__S 99 件の手段。
現在のロケールを'0'に変更します。これで、オブジェクト S のフィールド(つまり、ロケール'0'の items 変数)が利用可能になりました。
引数 99 にプッシュ動詞を適用します。以来プッシュロケールではありません「0」、検索がロケールからのパスに沿って作られて「0」ロケールに私たちを取るスタック そこからのプッシュは、それが適用される前に取得されます。
現在のロケールを現状に戻します。

既存のオブジェクトとそのクラスをすべて一覧表示するユーティリティ関数です。

```
   obcl =: 3 : '(, ({. @: (18!:2)))"0  (18!:1) 1'
```

現在 、Stackオブジェクトを参照する変数SとTがあります。

```
S     T     obcl ''
+-+   +-+   +-+-----+
|0|   |1|   |0|Stack|
+-+   +-+   +-+-----+
                |1|Stack|
                +-+-----+
```

(また、使用しているJのバージョンによっては、Jシステムによって自動的に生成されたオブジェクトやクラスが追加で使用されることがあります)。
スタックは、 Sは、言う使用して除去することができる破壊する方法 スタッククラスを。

```
   destroy__S  ''
1
```

私たちはそれがなくなったのを見ます。

```
   obcl ''
+-+-----+
|1|Stack|
+-+-----+
```

結論として、オブジェクトはロケールであることに特に注意してください。つまり、オブジェクトは変数でも値でもありません。オブジェクトは、関数の引数となるか、関数によって返されるものではありません。オブジェクトを別のオブジェクトの一部にすることはできません。
これらのことはすべて、オブジェクトの代わりにオブジェクトの名前を使って行うことができます。すべてのオブジェクトには、文字列で表される一意のシステム生成名があります。この文字列は単なる普通の値であり、すべての通常のことを

それで行うことができます。

25.5 継承 [Inheritance]

ここでは、新しいクラスが既存のクラスでどのように構築できるかを見ていきます。主な考え方は、クラスによっては、古いクラスの特殊バージョンとして新しいクラスを開発できるということです。

たとえば 、オブジェクトが値のコレクションである Collection というクラスがあるとします。例えば、オブジェクトが重複のないコレクションであり、このクラスを Set と呼ぶことができる新しいクラスを定義することができます。そして、設定オブジェクトは、特別な種類のあるコレクションオブジェクト。

このような場合、Set クラスは親クラス Collection の子である と言います。子オブジェクトの特殊なプロパティを実現するために、子は親のメソッドを継承し、おそらくいくつかを修正し、おそらく新しいメソッドを追加します。

簡単な例として、Collection という親クラスから始めます。

```
   coclass 'Collection'
   create  =: 3 : 'items =: 0 $ 0'
   add     =: 3 : '# items =: (< y) , items'
   remove  =: 3 : '# items =: items -. < y'
   inspect =: 3 : 'items'
   destroy =: codestroy
```

ここで inspect メソッドは、コレクションのすべてのメンバーのボックス化されたリストを生成します。

簡単なデモンストレーション:

```
   cocurrent 'base'
   C1 =: 0 conew 'Collection'
   add__C1 'foo'
1
   add__C1 37
2
   remove__C1 'foo'
1
   inspect__C1 0
```

```
+--+
|37|
+--+
```

私達がセットクラスを定義するので、そのセットが、ライブラリ動詞 coinsert を持つコレクションの子供である必要があることを指定します。

```
   coclass 'Set'
   coinsert 'Collection'
```

Set に重複がない というプロパティを表現するには、add メソッドだけを変更する必要があります。ここでうまくいくものがあります:

```
   add =: 3 : '# items =: ~. (< y) , items'
```

Set に 必要なその他のメソッドはすべて、親クラスの Collection から継承されています。セットの定義は終了し、 使用準備が整いました。

```
   cocurrent 'base'
   s1 =: 0 conew 'Set'  NB. make new Set object.
   add__s1 'a'
1
   add__s1 'b'
2
   add__s1 'a'
2
   remove__s1 'b'
1
   inspect__s1 0        NB. should have just one 'a'
+-+
|a|
+-+
```

25.5.1 原則の問題 [A Matter of Principle]

Set クラス の add メソッドの定義を思い出してください。

```
   add_Set_
3 : '# items =: ~. (< y) , items'
```

それには不快な特徴があります: それを書くには、Collection オブジェクトの内部についての知識、つまりboxed リストである items というフィールドがあります。コレクション のメソッドは、Collection オブジェクトのすべての処理に適しているはずです。原則として、私たちがメソッドに固執し、内部でうなることを避けるならば、Collection の内部への将来の可能性のある変更から、ある程度は盾にしたいと考えています 。このような変更は、たとえば、パフォーマンスを改善するためのものです。
add をもう一度やり直して みましょう。今回は親のメソッドに可能な限り固執します。親検査メソッドがメンバーシップのボックスリストを生成するという知識を使用します 。引数 y がメンバシップの中にない場合は、それを親 add メソッドで追加します。

```
   add_Set_ =: 3 : 0
if. (< y) e. inspect 0
do.  0
else. add_Collection_ f. y   NB. see below !
end.
)
```

それほど素晴らしいことではありませんが、それは私たちが原則をもって支払う代金です。セット s1 でそれを試して:

```
   inspect__s1 0
+-+
|a|
+-+
   add__s1     'a'
0
   add__s1     'z'
2
   inspect__s1 0
+-+-+
|z|a|
+-+-+
```

25.6 継承されたメソッドの使用 [Using Inherited Methods]

Set クラス の add メソッドの定義を見てみましょう。

```
   add_Set_
3 : 0
if. (< y) e. inspect 0
do.  0
else. add_Collection_ f. y   NB. see below !
end.
)
```

答えが必要な質問がいくつかあります。

25.6.1 最初の質問 [First Question]

メソッドはどのように継承されますか？つまり、親 Collection クラスの inspect メソッドが Set メソッドとして使用できるのはなぜですか？要するに、この方法は、経路に沿って、すなわち、

- セットなどのオブジェクト s1 がロケールです。これには、オブジェクトのフィールド変数が含まれています。
- クラスのメソッドが実行されると、現在のロケールは(一時的に)そのクラスのオブジェクトのロケールになります。これは、メソッドを呼び出す方法から、method__オブジェクト引数の形式の式に従います。
- オブジェクトロケールからのパスはクラスロケールに移動し、そこから親ロケールに移動します。したがって、この方法は経路に沿って見出される。

s1 などの Set オブジェクトには Set へのパスがあり、Collection へのパスがあります。

```
   copath > s1
+---+----------+-+
|Set|Collection|z|
+---+----------+-+
```

25.6.2 2 番目の質問 [Second Question]

In the definition of add_Set_

```
   add_Set_
3 : 0
if. (< y) e. inspect 0
do.  0
else. add_Collection_ f. y   NB. see below !
end.
)
```

親メソッドがあることを考えると検査として単に呼ばれている 検査、なぜ親方法があるが add が呼ばれ add_Collection_？add と呼ばれるメソッドを定義しているため、add への参照は致命的な循環性になります。

25.6.3 3 番目の質問 [Third Question]

親 add メソッドが add_Collection_ f として指定されている理由 ？
そのため add_Collection_が locative 名前であり、そして locative 名で式を評価すると、ロケールの変更を伴うだろう。第 24 章から、add_Collection_0 はロケール Collection で評価されることを 思い出してください。これは不正確です。メソッドを適用するときにオブジェクトロケールにする必要があります。
以来、F。(add_Collection_f。)評価が終了する頃には 、ロケールを変更せずに適用できる関数の完全評価値で正しいロケールに戻ります。

```
   add_Collection_ f.
3 : '# items =: (< y) , items'
```

25.7 ライブラリ動詞 [Library Verbs]

ここでは、選択したライブラリ動詞の簡単な概要を示します。

coclass 'foo'	introduce new class foo
coinsert 'foo'	this class to be a child of foo
conew 'foo'	introduce a new object of class foo

conl 0	list locale names
conl 1	list ids of object locales
names_foo_ ''	list the methods of class foo
copath <'foo'	show path of class foo
coname ''	show name of current locale

第 25 章終了

第 26 章: スクリプトファイル [Script Files]

J の行の形式のテキストを含むファイルは、スクリプトファイルまたはスクリプトと呼ばれます。慣例により、スクリプトは.ijs で終わるファイル名を持ちます。スクリプトファイルで J の行を実行するプロセスは、スクリプトの「ロード」と呼ばれます。
私たちは、特定のプログラミングプロジェクトのために独自のスクリプトを作成します。さらに、J システムには、汎用ユーティリティの事前定義スクリプトのライブラリが付属しています。
この章の計画は、
スクリプトをロードするためのビルトイン動詞
ライブラリー・スクリプトの便利なロードを含む、ロード動詞とその利点
「プロファイル」スクリプトは、J セッションの開始時に自動的にロードされます

26.1 スクリプトの作成 [Creating Scripts]

独自のスクリプトを保存するディレクトリを特定すると便利です。
その目的のために便利な ディレクトリ j701-user があります。J インストールの一部として自動的にインストールされます。そのフルパス名は

```
   jpath '~user'
c:/users/homer/j701-user
```

変数、scriptdir と言うと、選択したスクリプトディレクトリの名前を末尾の '/'

```
   ] scriptdir =: (jpath '~user') , '/'
c:/users/homer/j701-user/
```

スクリプトは通常、テキストエディタを使用して作成されますが、J を使用して必要なときにスクリプトの小さな例を作成することができます。組み込みの動詞 1！：2 を使用して、たとえば example.ijs というファイル名で、小さなスクリプトを作成する例を次に示します。

```
   (0 : 0) (1!:2) < scriptdir,'example.ijs'
plus =: +
k    =: 2 plus 3
k plus 1
)
```

26.2 スクリプトのロード [Loading Scripts]

スクリプトをロード するビルトインの動詞は 0！： 1 です。引数は、ボックス化された文字列としてのファイル名です。

```
   0!:1 < scriptdir,'example.ijs'
   plus =: +
   k    =: 2 plus 3
   k plus 1
6
```

私たちは、実行されたときのスクリプトの行の表示と、計算の結果値を画面に表示します。plus と k の定義が利用可能になりました:

plus	k
+	5

私たちが見たように、 動詞 0！： 1 は、スクリプトを表示して読み込みます。スクリプトにエラーがある場合、0！：1 が停止します。表示するかどうか、エラー後に読み込みを停止するか、読み込みを続行するかを選択できます。同様の動詞が 4 つあります。

```
0!:0  no display   stopping on error
0!:1  with display stopping on error
0!:10 no display   continuing on error
0!:11 with display continuing on error
```

例えば:

```
   0!:0 < scriptdir,'example.ijs'
```

私たちは画面上に何も見ない。k プラス 1 のスクリプトで計算された値 は破棄されます。

26.3 ロード動詞 [The load Verb]

動詞の読み込みは事前に定義されています。つまり、標準 J 設定で自動的に使用できます。それは 0!: 0 のようにスクリプトを読み込むのに使うことができます

```
   load < scriptdir,'example.ijs'
```

スクリプトは表示せずにロードされ、エラー時に停止します。表示とともにロードされ、エラー時に停止するコンパニオン動詞 loadd があります。

```
   loadd < scriptdir, 'example.ijs'
   plus =: +
   k    =: 2 plus 3
   k plus 1
6
```

load と loadd には 0 !: n と比較していくつかの利点があります。最初のものは、ファイル名を囲む必要はないということです。

```
   loadd scriptdir, 'example.ijs'
   plus =: +
   k    =: 2 plus 3
   k plus 1
6
```

26.4 スクリプトのローカル定義 [Local Definitions in Scripts]

ここで、スクリプトのローカル変数の扱いについて見ていきます。以下はスクリプトの例です。

```
   (0 : 0) (1!:2) < scriptdir, 'ex1.ijs'
w   =: 1 + 1
foo =: + & w
)
```

変数 w が動詞 foo を定義するのを助ける唯一の目的を持っているとし、そうでなければ w は興味がないとします。w をローカル変数にする方が良いでしょう。

第一に、我々はに割り当てる必要があるワットで=。 明示的な関数でローカル変数に代入するのと同じ方法で行います。私たちの改訂版は次のようになります:

```
   (0 : 0) (1!:2) < scriptdir, 'ex2.ijs'
w   =. 1 + 1
foo =: + & w
)
```

第 2 に、明示的な関数の外側（すなわち、「トップレベル」）「=.」は「=:」と同じであるため、w はローカルである、すなわち明示的な関数である必要があります。 "0！：n"の周りに明示的に定義されたラッパーが必要です、以下のように:

```
   LL =: 3 : '0!:0 y'
```

スクリプトを読み込むと

```
   LL < scriptdir, 'ex2.ijs'
```

結果を見てください:

foo	w
+&2	error

私たちは foo が期待どおりであり、意図したように w の価値がないことを見ています。したがって、w はスクリプトの実行、または厳密に言えば、LL の実行にローカルであった。負荷動詞 の利点は、w をローカルにするために必要な明示的な関数を提供することです。

```
   erase 'foo';'w'
1 1

   load scriptdir, 'ex2.ijs'
```

foo	w
+&2	error

26.4.1 スクリプト内のローカル動詞 [Local Verbs in Scripts]

前の例では、ローカル変数 w は名詞でした。ローカル動詞では、さらに検討が必要です。次に、ローカル動詞(sum)を使用してグローバル動詞(mean)の定義を支援するスクリプトの例を示します。

```
   (0 : 0) (1!:2) < scriptdir, 'ex3.ijs'
sum  =. +/
mean =: sum % #
)

   load  < scriptdir, 'ex3.ijs'
```

これは機能しません。なぜなら、必要な和と 和はローカルであり、もう利用できなくなるからです。

mean	sum
sum % #	error

救済策は、副詞 f を用いて、mean の定義を「修正」することである。(私たちが第 12 章で行ったように)。私たちの改訂版は

```
   (0 : 0) (1!:2) < scriptdir, 'ex4.ijs'
sum =. +/
mean =: (sum % #)  f.
)
```

今、平均は和とは無関係です

```
  load  < scriptdir, 'ex4.ijs'
```

mean	sum
+/ % #	error

26.5 ロケールへの読み込み [Loading Into Locales]

私たちは 24 章のロケールを見ました。0!: n でスクリプトをロードするか、ロード すると、現在のロケールになり、スクリプトの定義が読み込まれます。

デフォルトでは、現在のロケールはベースです。一般的に、指定されたロケール、たとえばロケール 1 にスクリプトをロードしたい場合があります。
これは一つの方法です:

```
   load_one_  scriptdir, 'example.ijs'
   plus_one_
```

別の方法は、スクリプト自体にロケールを指定させることです。例えば、

```
   (0 : 0) (1!:2) < scriptdir, 'ex5.ijs'
18!:4 < 'two'
w   =. 1 + 1
foo =: + & w
)
```

スクリプトはそれ自身をロケール 2 に変換します

```
   load scriptdir, 'ex5.ijs'

   foo_two_
+&2
```

0: ！n と比較して負荷の 利点があります。現在のロケールがベースであることに注意してください。

```
   18!:5 ''  NB. current locale before loading
+----+
|base|
+----+
```

ex5.ijs をロードする と、スクリプトがロケール 2 にアクセスしたかどうかにかかわらず、現在のロケールは引き続きベースになります。

```
   load scriptdir,'ex5.ijs'
   18!:5 ''  NB. current locale after loading
+----+
|base|
+----+
```

ただし、同じスクリプトを 0!: n でロードすると、以前の現在のロケールは復元されません。

```
   18!:5 '' NB. current locale before loading
+----+
|base|
+----+
   0!:0 < scriptdir,'ex5.ijs'
   18!:5 '' NB.  current locale after loading
+---+
|two|
+---+
```

私たちは、自己操縦スクリプトは 0!: n ではなく load で読み込まれるべきだと結論づけます。
私たちは基本に戻ります。

```
   18 !: 4 < 'base'
```

26.6 繰返の回避方法 [Repeated Loading, and How to Avoid It]

負荷の もう 1 つの利点はこれです。1 つのスクリプトが 2 番目のスクリプトに依存しているとします。最初の行に load 'second'のような行が含まれている場合、最初のロード時に 2 番目の行が自動的にロードされます。
最初のスクリプトを再度ロードすると(たとえば、エラーを修正した後)、2 番目のスクリプトが再びロードされます。これは不必要または望ましくない場合があります。あらかじめ定義された動詞 require は load と似ています が、すでにロードされている場合はスクリプトをロードしません。
ここにデモがあります。最初のスクリプトに次の 2 行があるとします。

```
   (0 : 0) (1!:2) < scriptdir,'first.ijs'
    require scriptdir, 'second.ijs'
    a =: a + 1
)
```

ここで変数 a はカウンタです: first.ijs が読み込まれるたびに、 a がインクリメントされます。同様に 2 番目のスクリ

プトの場合:

```
   (0 : 0) (1!:2) < scriptdir, 'second.ijs'
    b  =: b + 1
)
```

カウンタ a と b をゼロに設定し、最初のスクリプトを読み込んでカウンタを調べます。

(a =: 0),(b =: 0)	load scriptdir, 'first.ijs'	a,b
0 0		1 1

明らかに、各スクリプトは 1 回実行されています。最初にもう一度ロードすると、再び実行されたことがわかりますが、2 番目には実行されていません。

load scriptdir,'first.ijs'	a,b
	2 1

26.7 ロードステータス [Load Status]

J システムは、load または 0!: 0 のいずれであっても、セッションにロードされたすべてのスクリプトを追跡します。引数が NULL のビルトイン動詞 4!: 3 は、ファイル名のボックスリストとしてレポートを提供します。現在のセッションのこのレポートの最後のエントリがいくつかあります。

```
   ,. _4 {. 4!:3 ''
+-----------------------------------+
|c:¥users¥homer¥j701-user¥ex4.ijs   |
+-----------------------------------+
|c:¥users¥homer¥j701-user¥ex5.ijs   |
+-----------------------------------+
|c:¥users¥homer¥j701-user¥first.ijs |
+-----------------------------------+
|c:¥users¥homer¥j701-user¥second.ijs|
+-----------------------------------+
```

上記の スクリプト example.ijs で plus を定義したことを思い出してください。組込み動詞 4!: 4 は、どのスクリプトからどの名前がロードされたかを追跡します。引数は名前

(例えばプラス)で、結果は 4!: 3 によって生成されたスクリプトのリストへのインデックスです。私たちは、実際にはスクリプト example.ijs を読み込むことによってプラスが定義されていることがわかります

```
   i =: 4!:4 < 'plus'
14
   i { 4!:3 ''
+-----------------------------------+
|c:¥users¥homer¥j701-user¥example.ijs|
+-----------------------------------+
```

26.8 ライブラリスクリプト [Library Scripts]

26.8.1 標準ライブラリ [Standard Library]

J システムには、事前定義された機能の有用なコレクションを含むスクリプトファイルが付属しています。
セッションの開始時に自動的に読み込まれるスクリプトのリストを見ることができます。このために、組み込みの動詞 4!: 3 を使用して、ファイル名のボックスリストを生成します。ここに最初の 9 つのスクリプトがあります:

```
   ,. 9 {. 4 !: 3 ''
+-----------------------------------------------+
|C:¥users¥homer¥j701¥bin¥profile.ijs             |
+-----------------------------------------------+
|C:¥users¥homer¥j701¥system¥util¥boot.ijs        |
+-----------------------------------------------+
|C:¥users¥homer¥j701¥system¥main¥stdlib.ijs      |
+-----------------------------------------------+
|C:¥users¥homer¥j701¥system¥util¥scripts.ijs     |
+-----------------------------------------------+
|C:¥users¥homer¥j701¥system¥main¥regex.ijs       |
+-----------------------------------------------+
|C:¥users¥homer¥j701¥system¥main¥task.ijs        |
+-----------------------------------------------+
|C:¥users¥homer¥j701¥system¥util¥configure.ijs   |
+-----------------------------------------------+
|c:¥users¥homer¥j701-user¥config¥recent.dat      |
+-----------------------------------------------+
|c:¥users¥homer¥j701¥system¥main¥ctag.ijs        |
```

```
+-----------------------------------------+
```

これらの中には、スクリプトファイル stdlib.ijs があります
で定義された関数 stdlib.ijs はで終わる Z、彼らはプログラマに便利用意されていたロケール。z ロケールには 300 以上のものが定義されています。

```
   # nl_z_ ''
369
```

たとえば、J ユーザーマニュアルに 記載されているファイル処理ユーティリティ関数 は、文字「f」で始まる z ロケールにあります。

```
   6 6 $ 'f' nl_z_ ''
+--------+----------+--------+----------+------
---+------------+
|f2utf8                                  |fappend
|fappends|fapplylines|fboxname |fc          |
+--------+----------+--------+----------+------
---+------------+
|fcompare|fcompares |fcopynew|fdir        |ferase
|fetch       |
+--------+----------+--------+----------+------
---+------------+
|fexist  |fexists   |fgets   |fi          |flatten
|fliprgb      |
+--------+----------+--------+----------+------
---+------------+
|fmakex                                 |foldpara
|foldtext|fpathcreate|fpathname|fputs        |
+--------+----------+--------+----------+------
---+------------+
|fread   |freadblock|freadr  |freads      |frename
|freplace     |
+--------+----------+--------+----------+------
---+------------+
|fsize   |fss       |fssrplc |fstamp      |fstring
```

```
|fstringreplace|
+--------+----------+--------+----------+------
---+------------+
```

26.8.2 J-APP・ライブラリ [The J Application Library]

J Application Library(JAL)を構成するスクリプトファイルの広範なコレクションがあります。ここに JAL が記載されています

26.9 ユーザー定義の起動スクリプト [User-Defined Startup Script]

すべての J セッションの開始時に自動的にロードされるように、独自の定義のコレクションがあるとします。これを実現するために、私たちの定義を startup.ijs という名前のスクリプトファイルに入れることができます。このファイルの絶対パス名は、次の式で与えられます。

```
   jpath '~config/startup.ijs'
c:/users/homer/j701-user/config/startup.ijs
```

ここに例があります。いくつかの定義を含むスクリプトファイルを作成します。この例では、型チェックに便利な動詞をいくつか定義できます。

```
   (0 : 0)  (1 !: 2) < jpath '~config/startup.ijs'
is_int    =: 4 = 3 !: 0
is_char   =: 2 = 3 !: 0
is_number =: 1 4 8 16 64 128  e.~  3!:0
is_scalar =: 0 = # @: $
is_list   =: 1 = # @: $
is_string =: is_char *. is_list
)
```

このスクリプトファイルを配置すると、次のセッションで自動的にロードされます。新しいセッションの開始時に読み込まれたスクリプトのリストを調べることでこれを検証します。

```
   ,. 11 {. 4 !: 3 ''
+-----------------------------------------+
|C:¥users¥homer¥j701¥bin¥profile.ijs       |
```

```
+-----------------------------------------------+
|C:¥users¥homer¥j701¥system¥util¥boot.ijs       |
+-----------------------------------------------+
|C:¥users¥homer¥j701¥system¥main¥stdlib.ijs     |
+-----------------------------------------------+
|C:¥users¥homer¥j701¥system¥util¥scripts.ijs    |
+-----------------------------------------------+
|C:¥users¥homer¥j701¥system¥main¥regex.ijs      |
+-----------------------------------------------+
|C:¥users¥homer¥j701¥system¥main¥task.ijs       |
+-----------------------------------------------+
|C:¥users¥homer¥j701¥system¥util¥configure.ijs  |
+-----------------------------------------------+
|c:¥users¥homer¥j701-user¥config¥recent.dat     |
+-----------------------------------------------+
|c:¥users¥homer¥j701¥system¥main¥ctag.ijs       |
+-----------------------------------------------+
|c:¥users¥homer¥j701¥system¥util¥jadetag.ijs    |
+-----------------------------------------------+
|c:¥users¥homer¥j701-user¥config¥startup.ijs    |
+-----------------------------------------------+
```

startup.ijs がロードされ、その定義が利用可能である ことがわかります。

```
   is_string 'hello'
1
```

第 26 章終了

第 27 章: 表現と変換 [Representations and Conversions]

この章では、関数とデータのさまざまな変換を見ていきます。

27.1 クラスと型 [Classes and Types]

私たちが物事を他のものに変えているのであれば、私たちがどのようなことをしているのかを示す関数から始めておくと便利です。

27.1.1 クラス [Classes]

与えられた割り当て、名前=: 何か、何かが名詞、動詞、副詞または結合詞を表す式です。つまり、何かが属する 4 つのクラスがあります。

ここにはクラスと呼ば れるビルトインの動詞 4!: 0 があります。

```
   class =: 4!:0
```

クラス'<name>にクラスを適用すること で、何かのクラスを発見することができます。例えば、

n =: 6	class < 'n'
6	0

n のクラス のための 0 の結果は、n が名詞であることを意味する。ケースは次のとおりです。

```
          0  noun
          1  adverb
          2  conjunction
          3  verb
```

文字列'n'は有効な名前ではないか、n は名前として有効ですが n に値は割り当てられません。

```
         _2  invalid
         _1  unassigned
```

例えば:

C =: &	class <'C'	class <'yup'	class <'1+2'
&	2	_1	_2

class の引数は 、'C'のような文字列を作るためにその名前を引用することによって目的のオブジェクトを識別します。なぜ引数は単にオブジェクトではないのですか？なぜなら、クラス関数の目的によって、 オブジェクトは動詞、名詞、副詞または結合詞であり、副詞または結合詞は他の関数の引数として提供できないからです。
何故なの？関心対象が連結 C であると仮定する。どのようにクラスが定義されていても、動詞か副詞かにかかわらず、フォーム(クラス C)または(クラス C)の表現はいずれもバイデントまたは構文エラーです。関数のクラスが引数 C に適用されることはありません。したがって、その名前を引用して C を識別する必要があります。

27.1.2 タイプ [Types]

名詞は、整数の配列、浮動小数点数や文字の配列などである可能性があります。任意の配列の型は、組み込み動詞 3!:0 を適用することによって検出され ます。例えば

(3!:0) 0.1	(3!:0) 'abc'
8	2

8 の結果は浮動小数点を意味し、結果 2 は文字を意味します。結果の可能性のあるケースのいくつかは次のとおりです。

```
1       boolean
2       character (that is, 8-bit
        characters), also called "literal"
        in the Dictionary
4       integer
8       floating point
16      complex
32      boxed
64      extended integer
128     rational
65536   symbol
```

```
 131072   wide character (16-bit)
```

標準ライブラリには 有用な動詞データ型もありますが、これは数値ではなく引数の型の名前を生成します。

datatype 0.1	datatype 'abc'
floating	literal

27.2 実行 [Execute]

内蔵の動詞がある」。 (と呼ばれるドット、ダブルクォート『』を実行する)。その引数が有効な J の式を表す文字列で、結果はその式の値です。

```
    ". '1+2'
3
```

文字列は代入を表すことができ、代入が実行されます。

". 'w =: 1 + 2'	w
3	3

文字列が動詞または副詞または結合詞を表す場合、結果は null になります。これは、Execute 自体が動詞であり、その結果が名詞でなければならないためです。ただし、関数を取得するための割り当てを正常に実行できます。

". '+'	". 'f =: +'	f
		+

27.3 表現 [Representations]

式をキーボードで入力すると、値が計算され、画面に表示されます。より正確には、それは一連の文字として表示されるその値の表現です。

たとえば、関数 foo を定義すると、次のようになります。

```
    foo =: +/ % #
```

foo の定義を見てください：

```
    foo
 +-----+-+-+
 |+-+-+|%|#|
```

```
||+|/|| | |
|+-+-+| | |
+-----+-+-+
```

私たちは画面上で foo の表現を見る。私たちが見るものは、いくつかの可能性の中からどのオプションが選択されるかによって異なります。関数の表現のためのデフォルトのオプションは、上の foo の例のように、ボックス化された構造です。他にも次のような可能性があります。

27.3.1 名詞の表示[Representations of Nouns for Display]

文字型の名詞の場合、J インタープリターがその文字を画面に書き込むだけで、その値をユーザーに表すことができます(適切な場所に改行を入れます)。
対照的に、数字やボックス、記号などの他のデータ型の名詞の場合、表示可能な表現ではデータ値を文字に変換する必要があります。
この目的のために、組込み関数": (二重引用符のコロン、Format と呼ばれる)があります。すでに第 19 章で format を使用しています。
Format 動詞は、引数と同じに見える文字表現である名詞を生成します。

```
   ] n1 =: 'toujours' ; 'l''audace'
+--------+--------+
|toujours|l'audace|
+--------+--------+

   ] r1 =:  ": n1   NB. a representation of n1
+--------+--------+
|toujours|l'audace|
+--------+--------+
```

しかし N1 とその表現 r1 は異なるデータ型、及び異なる寸法のものです。

datatype n1	datatype r1	$ n1	$ r1
boxed	literal	2	3 19

27.3.2 書式動詞のボックスの描画 [Drawing Boxes with the Format Verb]

フォーマット": 動詞は 3 つのことを行います:

●引数の文字表現を生成します。
●数字の表示方法を制御できます。第 19 章を参照してください。
●ボックスの表示方法を制御できます。これを次に見ます。

描画ボックスに使用する文字を指定できます。この本では、「+ | - '文字を使用することができます。これらは、すべての Web ブラウザで正しく表示されることが前提です。ただし、J セッションでは、他の文字で囲まれたボックスが表示され、より気に入った外観になります。
Format 動詞で使用されるボックス描画文字は、グローバルパラメータで指定されます。このパラメータの値は、9 ! : 6 '':

```
   9!:6 ''
+++++++++|-
```

現在有効な 11 文字が表示されます。パラメータは 9 ! : 7 で設定できます 。以下は左引数で与えられたボックス描画文字で右引数をフォーマットする動詞です。グローバルパラメータを保存して復元することに注目してください。

```
   fmt =: 4 : 0
    assert. 11 = # x
    t =. 9!:6 ''   NB. save current box-draw chars
    9 !:7 x        NB. set box-draw chars to new value
    z =. ": y
    9 !: 7 t       NB. restore original box-draw
chars
    z
)
```

11 個の文字のどこに行くのかを示すために、0123456789A の文字でボックスを描くことができます

```
   ] w =: 2 2 $ < '    '
+----+----+
|    |    |
```

```
+----+----+
|    |    |
+----+----+
      '0123456789A' fmt w
0AAAA1AAAA2
9    9    9
3AAAA4AAAA5
9    9    9
6AAAA7AAAA8
```

Dyadic Format は、ボックス内のデータの配置場所を制御します。詳細は、辞書を参照してください。

27.3.3 関数の表現 [Representations of Functions]

関数の表現を生成するためのいくつかのオプション、つまり動詞、副詞または結合詞の表現があります。
デフォルトでは、現在のオプションは "ボックス化された表現"であるので、ボックスの構造としてグラフィカルに表現された動詞 foo(上で定義した)が見えます。

```
   foo
+-----+-+-+
|+-+-+|%|#|
||+|/|| | |
|+-+-+| | |
+-----+-+-+
```

以下に説明する他のオプションも利用できます。画面上で関数を表示するオプションを選択して現在のものにするには、次の式のいずれかを入力します。

```
(9!:3) 2  NB. boxed (default)
(9!:3) 5  NB. linear
(9!:3) 6  NB. parenthesized
(9!:3) 4  NB. tree
(9!:3) 1  NB. atomic
```

現在のオプションは、別のオプションを選択するまで有効です。

27.3.4 線形表現 [Linear Representation]

線形表現を選択して foo を見直すと、次のようになります。

```
   (9!:3) 5  NB. linear

   foo
+/ % #
```

foo は、キーボードで入力できる形式、つまり式として認識されます。
線形形式は元の定義と同等ですが、必ずしも文字通り同一ではないことに注意してください。括弧を最小限に抑える傾向があります。

```
   bar =: (+/) % #

   bar
+/ % #
```

関数、すなわち動詞、副詞および結合詞が現在の表現で示されています。対照的に、現在のオプションにかかわらず、動詞によって生成される表現には常に名詞が表示されます。線形は現在のものですが、

```
   noun =: 'abc';'pqr'

   noun
+---+---+
|abc|pqr|
+---+---+
```

27.3.5 括弧で囲まれた表現 [Parenthesized Representation]

カッコで囲まれた表現は、関数を式として示す際に線形に似ています。線形とは異なり、括弧で囲まれた形式は、式の論理構造をより明瞭にするために有用に括弧を追加します。

```
   (9!:3) 6  NB. parenthesized
   zot =: f @: g @: h
   zot
(f@:g)@:h
```

27.3.6 ツリー表現 [Tree Representation]

ツリー表現は、構造をグラフィカルに表示する別の方法です。

```
   (9!:3) 4  NB. tree
   zot
            +- f
      +- @: -+- g
-- @: -+- h
```

27.3.7 原子表現 [Atomic Representation]

完全性のために、ここでは原子表現を述べる。私たちはそれに戻ってきます。
続行する前に、現在の表現オプションをリニアに戻します。

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

27.4 表現関数 [Representation Functions]

画面上に表現を表示するための現在の選択肢に関係なく、適切な組み込み動詞を適用することによって、任意の表現を名詞として生成することができる。

y が割り当てられた値を持つ名前である場合、y の表現は、以下の動詞の 1 つを引数 <'y'に適用することによって生成される名詞です。

```
   br =:  5!:2    NB. boxed
   lr =:  5!:5    NB. linear
   pr =:  5!:6    NB. parenthesized
   tr =:  5!:4    NB. tree
   ar =:  5!:1    NB. atomic
```

例えば、囲みと括弧で囲まれた形式の zot は、次のように表示されます。

br < 'zot'	pr < 'zot'
+--------+--+-+ |+-+--+-+|@:|h| ||f|@:|g|| | | |+-+--+-+| | | +--------+--+-+	(f@:g)@:h

私たちは名詞の様々な表現を得ることができます。例えば、箱入りと線形のように:

br <'noun'	lr <'noun'
+---+---+ |abc|pqr| +---+---+	<;._1 ' abc pqr'

5！: n によって生成される表現は、それ自体が名詞です。
動詞 foo の線形形式は、長さ 6 の文字列です。

foo	s =: lr <'foo'	$ s
+/ % #	+/ % #	6

s の 6 文字は、動詞を表す式を表します。文字列 s で表される動詞をキャプチャするには、文字列の先頭に文字を割り当てて割り当てを行い、割り当てを実行します。

s	$ s	a =: 'f =: ' , s	". a	f 1 2
+/ % #	6	f =: +/ % #		1.5

27.4.1 原子表現 [Atomic Representation]

私たちは、で見た第 10 章と第 14 章 機能のシーケンスを形成することができることは有用です。これは、動詞の列ではなく、gerunds を意味します。例えば、動詞のシーケンスとみなされる gerund は、引数の特定の場合に適用可能な動詞を見つけるために索引付けすることができます。
インデックスを作成できるようにするには、配列を名詞の配列にする必要があります。したがって、動詞をその動詞を表す名詞に変換することに興味があり、その逆もあります。gerund は原子表現を含むそのような名詞のリストです。アトミックな表現は逆を持つため、この目的に適しています。他の表現関数のどれも真逆行列を持たない。
何かの原子表現は、内部構造を持つ単一のボックスです。たとえば、h がフックとして定義された動詞であるとします。(フックは、単純な構造を持たない動詞の最も簡単な例です)。

```
h =: + %
```

h の ボックス表現とアトミック表現を比較する

br <'h'	ar < 'h'

+-+-+ |+|%| +-+-+	+----------+ |+-+------+| ||2|+-+-+|| || ||+|%||| || |+-+-+|| |+-+------+| +----------+

内部構造は、元の定義を再解析することなく動詞を名詞から効率的に復元することを可能にする符号化である。定義が格納されている内部形式を反映します。それは構造のもう一つのグラフィック表示ではありません。
エンコーディングはディクショナリに記述されています。ここでは詳しく説明しません。非常に簡単に言えば、この例では、最初の動詞が+ で、2 番目が%である場合、h はフックです(2 は "フック"のエンコーディングです)。
次の例は、名詞、動詞、副詞または結合詞の原子表現を生成できることを示しています。

```
N =: 6
V =: h
A =: /
C =: &
```

ar <'N'	ar <'V'	ar <'A'	ar <'C'
+------+ |+-+-+| ||0|6|| |+-+-+| +------+	+-+ |h| +-+	+-+ |/| +-+	+-+ |&| +-+

27.4.2 原子表現の逆転 [Inverse of Atomic Representation]

表現の逆数は、「抽象」と呼ばれることもあります(例えば数字は数字で表される抽象的な数学的オブジェクトです)。原子表現の逆数は 5！：0 で、 これを ab と呼ぶことができます。

```
ab =: 5!:0
```

ab は名詞、動詞、副詞または結合詞のいずれかを生成できなければならないため、副詞です。例えば、我々は、の原子表現を抽象化することを見る時間がに等しく、H

h	r =: ar < 'h'	r ab
+ %	+---------+ |+-+-----+| ||2|+-+-+|| || ||+|%||| || |+-+-+|| |+-+-----+| +---------+	+ %

同様に任意の型の議論のために。たとえば名詞 N または結合詞 C の場合

N	rN=: ar <'N'	rN ab	C	(ar <'C') ab
6	+-----+ |+-+-+| ||0|6|| |+-+-+| +-----+	6	&	&

27.4.3 表現関数のまとめ [Summary of Representation Functions]

表現機能	名詞に適用される	動詞、副詞または接続詞に適用される
Format verb,monadic " :	結果は常にボックス描画文字などのグローバルパラメータによって制御される 文字配列です	適用できません
式の入力に応じた	表示はグローバル	display は 、

自動表示	パラメータによって制御されるフォーマット動詞の結果です。	現在の repr の結果に適用されるフォーマット動詞の結果です。（9！：3）によって選択される関数（5！：n ）n
boxed repr.verv 5！：2	結果は box 化されます	
線形写像。動詞 5！：5	結果は実行可能な文字列です	
カッコ内の repr 動詞 5！：6	結果は parentheses を多く入れた実行可能な文字列です	
Tree repr. 動詞 5！：4	適切ではありません	結果は構造を描く文字配列です
Atomic repr. 動詞 5！：1	結果は boxed の構造です	

27.4.4 再訪の実行 [Execute Revisited]
原子表現の使用の別の例を次に示します。Execute は動詞ではなく名詞を表す文字列を評価することを思い出してください。Execute はそれ自体動詞であるため、結果として動詞を提供することはできません。

". '1+2'	". '+'
3	

任意のクラスの値を表現する文字列を評価するために、副詞 eval say を定義することができます。副詞 eval は、そのアトミック表現を抽象化することによって結果を配信します。

```
   eval =: 1 : 0
". 'w =. ' , u
```

```
(ar < 'w') ab
)
```

'1+2' eval	mean =: '+/ % #' eval	mean 1 2
3	+/ % #	1.5

27.4.5 再びの連結詞 [The Tie Conjunction Revisited]
14 章から、私達が綱連結'によって動名詞を形成することを思い出しなさい。その議論は 2 つの動詞であるかもしれない。

```
G =: (+ %) ` h
```

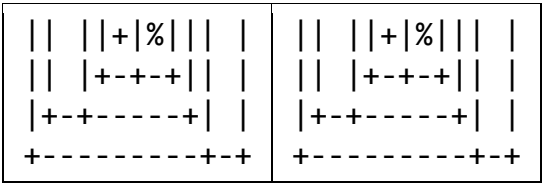
その結果は、原子表現のリストです。実証するために、私たちは 1 つを選択し、リストの最初のものを言い、動詞を抽象化します。

G	r =: 0 { G	r ab
+---------+-+ |+-+-----+|h| ||2|+-+-+|| | || ||+|%||| | || |+-+-+|| | |+-+-----+| | +---------+-+	+---------+ |+-+-----+| ||2|+-+-+|| || ||+|%||| || |+-+-+|| |+-+-----+| +---------+	+ %

この例は、Tie が動詞を表す式の引数を取ることができることを示しています。対照的に、原子表現関数(ar または 5！：1)は、引数を識別するためにボックス化された名前をとる必要があります。
Tie のように、動詞(名前ではない)を引数としてとり、アトム表現を生成 する結合 T があります。

```
T =: 2 : '(ar <''u.'') , (ar <''v.'')'
```

(+ %) T h	(+ %) ` h
+---------+-+ |+-+-----+|h| ||2|+-+-+|| |	+---------+-+ |+-+-----+|h| ||2|+-+-+|| |



27.5 バイナリデータの変換 [Conversions for Binary Data]

バイナリデータは、簡単に言えば、文字列としてコンパクトに表現された値です。ここでは、バイナリデータを扱うファイルを扱うために、J 配列とバイナリデータの値を変換する関数を見ていきます。データファイルについては、第 28 章で説明します。

以下では、32 ビットの PC が想定されているため、文字が 1 バイトを占め、浮動小数点数が 8 を占めると仮定します。

例えば、浮動小数点数の AJ 配列は、コンピュータのメモリに格納される。ストレージは、アレイのタイプ、ランク、形状、アレイ内の各番号のストレージに関する情報を保持する必要があります。配列内の各浮動小数点数には 8 バイトの記憶域が必要です。

浮動小数点数を長さ 8 の文字列に変換する組み込み関数があり、その逆もあります。

```
   cf8 =:   2 & (3!:5)   NB. float to 8 chars
   c8f =:  _2 & (3!:5)   NB. 8 chars to float
```

次の例では、n が浮動小数点であり、 n が変換されて長さが 8 の文字列 s が与えられ、s が n に等しい浮動小数点数に戻されることがわかります。

a =: 0.1 0.1	$ s =: cf8 a	c8f s
0.1 0.1	16	0.1 0.1

結果の文字はほとんど印刷できません。文字をアスキー文字セットで調べることができます:

```
   a. i. s
154 153 153 153 153 153 185 63
```

次に、数値の配列を変換することを検討してください。数字のリストは 1 つの文字列に変換され、その逆もあります::

a =: 0.1 0.1	$ s =: cf8 a	c8f s
0.1 0.1	16	0.1 0.1

cf8 のモナディックランクは無限大です: cf8 は引数全体に 1 回だけ適用されます。

```
   RANKS =: 1 : 'u b. 0'
   cf8 RANKS
_ _ _          NB.3 個の Under-bar を表示
```

引数はスカラまたはリストで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。

b =: 2 2 $ a	$ w =: cf8 b	$ w =: cf8"1 b
0.1 0.1 0.1 0.1	error	2 16

浮動小数点数は 8 文字に変換可能です。浮動小数点をより短い 4 文字の文字列に変換するオプションがあり、記憶の節約のために精度を犠牲にします。

```
   cf4 =:  1 & (3!:5)   NB. float to 4 chars
   c4f =: _1 & (3!:5)   NB. 4 chars to float
```

期待どおり、浮動小数点数を 4 文字に変換して戻すと、小さなエラーが発生する可能性があります。

```
   p =: 3.14159265358979323
```

p	$ z =: cf4 p	q =: c4f z	p - q
3.14159	4	3.14159	_8.74228e_8

AJ 整数は 4 バイトの記憶領域を必要とします。J の整数と 4 文字の文字列を変換する関数があります。

```
   ci4 =:  2 & (3!:4)  NB. integer to 4 char
  c4i =: _2 & (3!:4)  NB. 4 char  to integer
```

i =: 1 _100	$ s =: ci4 i	c4i s
1 _100	8	1 _100

s は 2 つの整数を表すので 、s の長さは 8 であることが わかります。
k が整数であり、c が k から 4 文字への変換で あると仮定する。

k =: 256+65	$ c =: ci4 k
321	4

c の 文字はほとんど印刷できないので、ASCII アルファベットでその場所を調べることで検査します。文字は、k の値の基数 256 桁であり、最初に(PC 上で)最も重要でない順に c に格納されていることがわかります。

k	a. i. c	256 256 256 256 #: k
321	65 1 0 0	0 0 1 65

範囲の整数_32768 に 32767 は、 2 文字の文字列とその逆に変換することができます。

```
ci2 =:  1 & (3!:4)  NB. integer to 2 char
c2i =: _1 & (3!:4)  NB. 2 char  to int
```

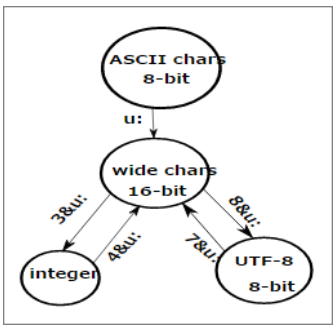
i	$ s =: ci2 i	c2i s
1 _100	4	1 _100

0～65535 の範囲の整数は 2 文字の文字列に変換することができ、その逆も可能です。 このような文字列は、「16bit unsigned」と記述されます。

m =: 65535	$ s =: ui2 m	u2i s
65535	2	65535

27.6 ユニコード [Unicode]
このセクションでは、Unicode の J サポートについて見ていきます。

J には 3 種類の文字データがあります
私たちがすでに 8 ビット ASCII として見てきた普通の文字データ
Unicode では「ワイド文字」と呼ばれる 16 ビット文字。
ファイルに Unicode を書き込む目的で、Unicode 文字を表す 8 ビット文字のシーケンス。
この表現は、UTF-8 エンコーディングと呼ばれます。
次の図は、文字データをある種類から別の種類に変換するために使用できる J 関数を示しています。関数は、u: ファミリのメンバーです。



J は文字データをサポートしていることが分かりました。例えば

```
   C =: 'this is a string'
```

組み込み動詞 3!: 0 は、データ値のタイプを示します。

```
   3!:0  C
 2
```

2 の結果は、C のデータ型が 「char」と呼ばれる 8 ビット文字であることを示します。
J はまた、「wchar」(ワイド文字)と呼ばれる 16 ビット文字の別のデータ型を提供します。組み込み関数 monadic u: char データを wchar に変換します。

```
   ] W =: u: C
this is a string
```

wchar データは前と同じように表示されますが、そのデータ型は 131072 として表示されます

```
   3!:0 W
131072
```

16 ビットの wchar 文字は、Unicode 標準の多くの文字の 1 つになります。組み込み関数 4&u: は、引数で指定された wchar 文字を生成します。これは、Unicode の "コードポイント"と呼ばれる 0〜65536 の範囲の整数です。
コードポイントは、しばしば 4 桁の 16 進数で与えられます。たとえば、ギリシャ文字アルファのコードポイントは、16b03b1 と書くことができる 16 進数 03b1 です

```
   ] alpha =: 4&u: 16b03b1
α
```

アルファは wchar です:

```
   3!:0  alpha
131072
```

アルファベットを含む wchar 文字列を作成することができます:

```
   ] U =: (u: 'the Greek letter alpha looks like this: ') , alpha
the Greek letter alpha looks like this:  α
```

ここで、wchar データ U をエクスポートする、たとえばデータファイルに書き込むとします。いくつかの認識された標準エンコーディング方式に従って、16 ビットの wchar データを 8 ビットのバイトのシーケンスとしてエンコードする必要があります。UTF-8 標準が適しています。
組み込み関数 8&u: は、wchar データの UTF-8 エンコーディングである文字列を生成します

```
   ] Z =: 8&u: U
the Greek letter alpha looks like this:  α
```

Z はデータ型 2(つまり 8 ビットの char 型)であり、Z のバイト 数は U の文字数より 1 多いことが わかります。アルファは 2 バイトとしてエンコードされるためです。

3!:0 Z	# U	# Z
2	42	43

8&u: の逆は組み込み関数 7&u: で、UTF-8 文字列から wchar 文字を生成します。

```
   ] A =: 7&u: Z
the Greek letter alpha looks like this:  α
```

A の文字の Unicode コードポイントを見ることができます。組み込み関数 3&u: は wchar データからコードポイント整数を生成します。A の最後の数文字を見ると、アルファのコードポイント整数は 10 進数 945、つまり 16 進数 03b1 であると予想されます。

```
   ] L =:  _6 {. A      NB. last few of A
is:  α

   3 & u: L
105 115 58 32 32 945
```

☆第 27 章終了

第 28 章: データファイル [Data Files]

一般的なファイル処理の主題と、ファイル内でのデータの構成方法は、それ自体が大きなテーマです。この章では、J で利用可能な施設の選択肢のみをカバーする予定です。
ファイルを読み込む J 関数は文字列の形で結果を生成し、同様にファイルを書き込む関数は文字列を引数として取ります。このような文字列は、コンピュータの利用可能なメモリが十分であるときに、ファイルの全データ内容とすることができる。
ここでは、文字列の入力と出力のための J 関数をいくつか見ていきます。次に、文字列をさまざまな形式のデータとして扱う例をいくつか見ていきます。最後に、マップされたファイルを従来のファイル処理の代替方法として検討します。

28.1 ファイルの読み書き [Reading and Writing Files]

28.1.1 組み込み動詞 [Built-in Verbs]

以下では、ファイル名は、J を実行しているコンピュータのオペレーティングシステムのファイル名として有効な文字列です。例:

```
   F =: 'c:\temp\demofile.xyz'    NB. a filename
```

組み込み動詞 1!:2 は、データをファイルに書き込みます。右の引数はボックスファイル名です。左の引数は文字列であり、書き込まれるデータです。効果は、ファイルがまだ存在しない場合にファイルが作成され、データがファイルの内容全体になることです。結果は null です。

```
   'some data' 1!:2 < F    NB. write to file F
```

ビルトイン動詞 1: 1 は、ファイルからデータを読み取ります。右の引数はボックスファイル名です。結果は文字列、つまり読み取られたデータです。

```
   data =: 1!:1 < F     NB.  read from file F
```

data	$ data
some data	9

28.1.2 ファイルとしての画面とキーボード [Screen and Keyboard As Files]

画面とキーボードはファイルとして扱うことができ、実行中のプログラムとのユーザー対話のためのシンプルな機能を提供します。
式 x(1！：2)2 は、x の値を「ファイル 2」、つまり画面に書き込みます。画面に表示する動詞は、次のように書くことができます。

```
   display =: (1!:2) & 2
```

例えば、以下はユークリッドのアルゴリズムによる最小共通分母の計算における段階を表示する動詞です。

```
   E =: 4 : 0
display x , y
if. y = 0 do. x else. (x | y) E x end.
)

   12 E 15
12 15

3 12
0 3
3 0
3
```

(1！： 2)&2 で表示される値は文字列に限定されません。上記の例では、数値のリストが表示されています。
ユーザ入力は、「ファイル 1」を読むことによって、すなわち(1！： 1)1 を評価することによって、キーボードから要求することができる 。結果は、ユーザーのキーストロークを含む文字ストリングです。たとえば、ユーザー対話の関数は次のようになります。

```
   ui =: 3 : 0
display 'please type your name:'
n  =.  (1!:1) 1
display 'thank you ', n
''
)
```

実行後

```
      ui ''
```

次のように画面にダイアログが表示されます。

```
please type your name:
Waldo
thank you Waldo
```

28.1.3 ライブラリ動詞 [Library Verbs]

"標準ライブラリ"(第 26 章)にはファイル処理のためのいくつかの有用な動詞があります。選択肢の概要を以下に示します。

s fwrite F	write string s to file F
fread F	read string from file F
s fappend F	append string s to file F
fread F;B,L	read slice from file F, starting at B, length L
s fwrites F	write text s to file F
freads F	read text from file F
fexist F	true if file F exists
ferase F	delete file F

今後は、これらのライブラリ動詞をファイル処理に使用します。

ライブラリ動詞 fwrite は、データをファイルに書き込みます。右の引数はファイル名です。左の引数は文字列であり、書き込まれるデータです。効果は、ファイルがまだ存在しない場合にファイルが作成され、データがファイルの内容全体になることです。

```
   'some data' fwrite F    NB. file write
9
```

結果は、書き込まれた文字の数を示します。_1 の結果は、左の引数が文字列でないか、または右の引数がファイル名とし

て有効でないか、または指定されたファイルが存在するが読み取り専用であるというエラーを示します。

```
   (3;4) fwrite F
_1
```

ライブラリ動詞 fread はファイルからデータを読み込みます。引数はファイル名で、結果は文字列です。

z =: fread F	$z
some data	9

_1 の結果、指定されたファイルが存在しないか、またはロックされています。

fread 'qwerty'	fexist 'qwerty'
_1	0

28.2 大きなファイル [Large Files]

大きなファイルの場合、コンピュータのメモリでファイルを単一の文字列として扱うには不十分な場合があります。このケースを非常に簡単に見ていきます。

初期の内容のファイルを書く:

```
   'abcdefgh' fwrite F
8
```

ライブラリ動詞 fappend を使ってファイルにデータを追加することができます。

```
   'MORE' fappend F
4
```

fappend の効果を見るために(このデモではもちろん、大きなファイルではない)、ファイル全体をもう一度読むことができます:

```
   fread F
abcdefghMORE
```

ファイルの選択されたスライス、たとえばバイト 4 から始まる 8 バイトを読み取ることができます 。この場合、fread は filename; start、size という形式の右引数を使用しま

す。

```
   start =: 4
   size  =: 8
   fread F ; start, size
efghMORE
```

28.3 データ形式 [Data Formats]

ここでは、データがファイル内にどのように編成されるか、つまり文字列で表される方法のいくつかの例を見ていきます。そこで、さまざまな内部構造を持つ文字列と J 変数の間の変換を調べます。

プログラム間でデータをやりとりするためにファイルを読み書きしています。「ライター」と「リーダー」という 2 つのプログラム。発生する質問は次のとおりです。

作家と読者はどちらも J プログラムになるでしょうか？もしそうなら、便利な J-only フォーマット、以下に説明する "バイナリ表現"があります。そうでない場合は、プログラミング言語に依存しないデータの記述から作業することが期待されます。

ライターとリーダーは、同じアーキテクチャのコンピューターで実行できますか？もしそうでなければ、J-to-J の状況でさえ、いくらかの精巧さが必要になるかもしれません。

データは、完全にいくつかの構造の繰り返しとして構成されていますか(「固定長レコード」など)。そうであれば、それを 1 つ以上の J 配列として扱うことができます。もしそうでなければ、私たちはいくつかの特別なプログラミングを必要とするかもしれません。

28.3.1 J-ファイルのバイナリ表現 [The Binary Representation for J-Files]

J プログラムでのみ特定のファイルを処理することを目的としているので、J プログラマにとって便利な任意のファイル形式を自由に選ぶことができます。「バイナリ表現」は特に便利です。

任意の配列 A に対して、

```
   A =:  'Thurs'; 19 4 2001
```

A のバイナリ表現は文字列です。配列と配列のバイナリ表現を変換するビルトインの動詞があります。

```
arrbin  =: 3!:1   NB. array to binary rep.
binarr  =: 3!:2   NB. binary rep. to array
```

場合 B はのバイナリ表現である A､我々がいることがわかり B は、一定の長さで、文字列です。

A	$ B =: arrbin A
+-----+---------+ |Thurs|19 4 2001| +-----+---------+	88

B をファイルに 書き込んで読み込み､逆変換を行って A の値を復元することができます。

B fwrite F	$ Z =: fread F	binarr Z
88	88	+-----+---------+ |Thurs|19 4 2001| +-----+---------+

J4.06 以降、さまざまなマシンアーキテクチャーを可能にするために、上記のバイナリ表現動詞のバリエーションがあります: 3 !: 1 の Dictionary を参照してください。

28.3.2 テキストファイル [Text Files]
式 a。(小文字の a ドット)は組み込みの名詞で、256 文字の ASCII 文字がすべて順番に含まれています。

65 66 67 { a.	$ a.
ABC	256

ASCII 文字セットでは、つまり a。位置 0 の文字はヌル、位置 10 は改行、位置 13 は改行文字です。J では、CR と LF という名前 は､キャリッジリターンと改行文字を意味する標準プロファイルであらかじめ定義されています。

```
   a. i. CR,LF
13 10
```

我々は見ての fread と fwrite の読み書き文字ファイルに使用します。テキストファイルは特別な種類の文字ファイルであり、その行は CR または LF 文字で区切られています。

いくつかのシステムでは、テキストの行は単一の LF によって区切られ、 他のシステムでは CR、LF のペアが期待されるという規則があります。J が実行されているシステムに関係なく、J のテキスト変数の場合、常に単一の LF と CR を使用して行を区切るという規則が適用されます。
次に、テキスト変数の例を示します。

```
   t =: 0 : 0
There is physics
and there is
stamp-collecting.
)
```

明らかに、3 文字の LF 文字と CR 文字のない文字列(つまり、1 次元の文字リスト)です。

$ t	+/t=LF	+/t=CR
49	3	0

このテキスト変数 t をテキストファイルに書き込むことを目指す場合は、LF または CRLF の単一の規則から選択する必要があります。 この状況に対処するには、2 つの便利なライブラリ動詞、fwrites および fread があります。
Windows では、X fwrites y は 、テキスト変数の書き込み X をファイルに y と各変換処理では、LF の 中の x に CRLF の中でペア Y。
Linux では、x fwrites y は変換なしでテキスト変数 x をファイル y に書き込みます。
Windows または Linux では 、Z =: freads y は 、ファイルの読み込み、y は任意の変換、

CRLF の中でペアを y の単一の LF テキスト変数に Z。
t の ようなテキスト変数を扱う際の便宜のために、行に分けることができます。この目的の動詞は切り詰められます(第 17 章で詳しく説明しています)。

```
   cut =: < ;. _2
```

cut は、 各行の最後に LF を削除して、ボックス化された行のリストを生成します。

```
   lines =: cut t
```

```
   lines
+---------------+------------+---------------
+
|There is physics|and there is |stamp-collecting.|
+---------------+------------+---------------
+
```

カット の逆数は、カットされていないと呼ぶことができます。それ は各ボックスの最後に LF を復元してから、ひげをつけて文字列を作ります。

```
   uncut =: ; @: (,&LF &. >)
   uncut lines
There is physics
and there is
stamp-collecting.
```

28.3.3 固定長レコード [Fixed Length Records with Binary Data]

我々のデータは、2 つの J 変数であると仮定します。テーブル CNAMES 顧客名の、およびリスト AMTS 顧客ごとに金額を顧客のためには、バランスが言います。

cnames =: 'Mr Rochester' ,: 'Jane'	,. amts =: _10000 3
Mr Rochester Jane	_10000 3

16 バイトのレコードでフォーマットされたこのデータをファイルに書き込むことを目的とします。各レコードには 2 つのフィールドがあります。顧客名は 12 バイト、4 バイトは符号付き整数です。可能なアプローチがあります。

この計画では、レコードと呼ばれる n-by-16 文字テーブルを cnames と amts から構築することが計画されています。この例では、n = 2 であり、レコードは次のようになります。

```
Mr Rochester####
Jane        ####
```

ここで、####は、バイナリ形式の整数の 4 つの文字を表しています。

私たちは、顧客名フィールドの n-by-12 テーブルと金額フィ

ールドの n-by-4 テーブルを並べてステッチしてレコードテーブルを構築します。
customer-names フィールドには、すでに 12 バイト幅であるため、適切な cnames がすでにあります。

```
   $ cnames
2 12
```

金額フィールドのために我々は変換 AMTS を使用して、文字に CI4 をから第 27 章。結果は 1 つの文字列で、これは n-by-4 に再構成されます。

```
   ci4 =:  2 & (3!:4)  NB. integer to 4 char
   amtsfield =: ((# amts) , 4) $ ci4 amts
```

ここでは、2 つの「フィールド」テーブルを並べてステッチして、n-by-16 レコードテーブルを作成します。

```
   records =: cnames ,. amtsfield
```

レコード を調べるために、ここにはユーティリティー動詞であるところの非印字文字 # があります。

```
   inspect  =:  3  :  ('A=.a.{~32+i.96';'(A  i.y)
{ A,''#''')
```

inspect records	$ records
Mr Rochester#### Jane ####	2 16

ファイルに書き込まれる出力文字列は、レコードのラブです。

```
   (, records) fwrite F
32
```

このプロセスの逆は、ファイルから J 変数を回復することです。ファイルを読み込んで、受信した文字列を取得します。

```
   instr =: fread F
```

レコード長が 16 であることが分かっているため、レコード数は

```
   NR =: (# instr) % 16
```

レコードテーブル を取得するために入力文字列を変更します。

```
   inspect records =: (NR,16) $ instr
Mr Rochester####
Jane        ####
```

データを抽出する。得意先名は、レコードの列 0～11 として直接取得されます。

```
   cnames =: (i.12) {"1 records
```

金額については、列 12-15 を抽出し、単一の文字列にラブし、c4i で整数に変換します。

```
   c4i =: _2 & (3!:4)  NB. 4 char  to integer
   amts   =: c4i  , (12+i.4) {"1  records
```

cnames	,. amts
Mr Rochester	_10000
Jane	3

28.4 マップされたファイル [Mapped Files]

ファイルは、実行中のプログラムの仮想アドレス変換メカニズムにファイルが一時的に組み込まれたときにマップされると言われています。マップされたファイルのデータは、J 変数の値として配列として J プログラマに直接表示されます。変数の値の変更は、ファイル内のデータに対する変更です。そのような場合、現在の目的で、ファイルが変数にマップされているとか、変数がファイルにマップされていると言うことができます。

マップされたファイルには、次の利点があります。

利便性。ファイル内のデータは、任意の J 配列のデータと同様に処理されます。ファイルの読み書きはありません。

永続変数。ファイルにマップされた変数はファイル内に存在し、1 つの J セッションから別のセッションに持続することができます。

2 つのケースがあります。最初のケースでは、どんな種類の既存のファイルも変数にマップすることができます。与えられたように、ファイル内のデータの構造を取って、J プログラムは必要なマッピングの説明を与えなければなりません。たとえば、固定長レコードを持つファイルを文字テーブルにマップすることができます。

第 2 のケースでは、特に変数へのマッピングのために、特別なフォーマット(「jmf」と呼ばれる)でファイルを J で作成することができます。この場合、説明は自動的に変数から導出され、データと共にファイルに保存されます。したがって、"jmf"ファイルは自己記述的です。
最初に jmf ファイルを作成してから、指定されたファイルをマッピングします。

28.4.1 マップ ライブラリスクリプト [Library Script for Mapped Files]

マップされたファイルを処理するためのライブラリスクリプト jmf.ijs があります。 現在のところ、J アプリケーションライブラリから直接ダウンロードするのが最も簡単です。 ここに jmf.ijs へのリンクがあります。

C: \ temp などのディレクトリにダウンロードしたと仮定して、J セッションにロードすることができます。

```
   load 'c:\temp\jmf.ijs'
```

スクリプトはロケール jmf に自身をロードします。

28.4.2 jmf ファイルと永続変数 [jmf Files and Persistent Variables]

いくつかの貴重なデータで 配列 V を構築したとします。これからは、数多くの J セッションにわたって使用し維持することを目指しています。多分、V は今や貴重なものであり、おそらく後続のセッションが変更されて追加されるときに価値があるでしょう。
私たちの貴重なデータ V は、数字、文字、またはボックスの配列にすることができます。簡単な例として、数字の表として V を使います。

```
   ] V =:  2 2 $ 1 2 3 4
1 2
3 4
```

V から次のように 永続変数を作ることができます。
ステップ 1 は、V の値に必要なファイルのサイズをバイト単位で見積もることです。時間が経つにつれて、V が現在のサ

イズから最終的には 64 KB まで増加することが予想されるので 、我々の推定値 S は

```
   S =: 64000
```

疑義がある場合は十分に余裕を持ってください。サイズは 32 ビットマシンで正の整数(浮動小数点ではなく)、したがって 2147483648(2Gb)未満でなければなりません。
ステップ 2 では、ファイル名を選択し、便宜上、ファイル名を文字列として保持する変数 F を定義します。例えば:

```
   F =:  'c:\temp\persis.jmf'
```

ステップ 3 は、ファイル F を、S バイトのデータを保持するのに十分な大きさの jmf ファイルとして作成することです。この目的のために、ユーティリティ関数 createjmf は(ロケール jmf で)利用できるので、次のように書くことができます:

```
   createjmf_jmf_ F;S
```

(あなたのシステムでは、J の別のバージョンでは、ここに示されているものとは異なる応答が表示されることがあります)。
この時点で、ファイル F が存在します。それを調べると、ファイルを自己記述的にするヘッダレコードを格納するために、実際のサイズが S より少し大きいことがわかります。

```
   fdir F
+----------+------------------+-----+---+------+
| persis.jmf | 2012 12 16 8 37 22 | 64284 | rw- | -----a |
+----------+------------------+-----+---+------+
```

ファイル F の内容は、最初は createjmf_jmf_ によって J 値、実際は長さゼロのリストを表すように設定され ます。重要な点は、ファイル F には明確な値が含まれていることです。
ステップ 4 は、ファイル F の内容を 新しい変数にマップすることであり、そのために名前 P を選択する。

```
   map_jmf_ 'P'; F
```

この声明は、事実上:

```
              P =:  value-currently-in-file-F
```

P が空のリストになっている ことを確認できます。

P	$ P
	0

マッピングファイルの効果が特に注目してください F 変数 P は、の値を割り当てることである F と P の周りに他の方法はありません。したがって、V が F のプリセットされた初期値によって上書きされ、失われるため、ファイル F を貴重な配列 V に直接マッピングすることは避けました。
ステップ 5 は、P に所望の値、すなわち V の値を割り当てることである。

```
   P =: V
```

変数 P はファイル F にマップされているため、永続変数になりました。行 0 の列 1 の値を 99 に変更するなどして、P を修正できます。

P	P =: 99 (<0 1) } P
1 2 3 4	1 99 3 4

または新しい行を追加することによって:

```
   ] P =: P ,  0 0
1 99
3  4
0  0
```

現在のセッションを終了する前に、ステップ 6 が必要です。私たちは、マッピング解除、変数 P をファイル確保するために、F が 閉じられているが。

```
   unmap_jmf_ 'P'
0
```

0 の結果は成功を示します。変数 P は存在しなくなりました:

P	$ P

error	$ P

P の値がファイル F に存続することを示すために、このセッションまたは別のセッションでマッピング、処理、およびマッピング解除を繰り返します 。永続変数として選択した名前 P は、このセッションのためのものです。別のセッションでは、ファイル F 内の永続変数を任意の名前にマップできます。今回は、永続変数の名前 Q を選択します。ファイル F を Q にマッピングします。

```
   map_jmf_ 'Q' ; F

   Q
1 99
3  4
0  0
```

Q を 変更する:

```
   ] Q =: Q , 7 8
1 99
3  4
0  0
7  8
```

Q を アンマップしてファイル F を閉じます。

```
   unmap_jmf_ 'Q'
0
```

28.4.3 マップ ファイルは固定サイズです [Mapped Files are of Fixed Size]

S バイトのデータに十分な大き さのファイル F を作成したことを思い出してください。

```
   S
64000
   fdir F
+----------+------------------+-----+---+------+
|persis.jmf|2012 12 16 8 37 22|64284|rw-|-----a|
+----------+------------------+-----+---+------+
```

ファイル F の変数 は現在のところこれよりもはるかに小さく、ファイルの未使用の末尾部分は junk で埋められています。しかし、追加することで Q を修正し続けると、ファイル

を埋めることによって限界に達し、エラーが発生します。説明するために、目的の動詞を 記入してください:

```
   fill =: 3 : 0
try.   while. 1 do. Q =: Q , 99 99 end.
catch. 'full'
end.
)

   map_jmf_ 'Q'; F
   fill ''
full
```

Q に 現在入っているデータの量は、整数(Q は整数)に整数の 4 バイト、つまり 4 * * / $ Q の整数を掛けたものと見積もることができます。Q の最終的なサイズに対するこの結果は、元のサイズ推定値 S と一致する。

4 * */ $ Q	S
64000	64000

```
   unmap_jmf_ 'Q'
0
```

28.4.4 ファイルの指定 [Given Files]

ここでは、通常のデータファイル(上記で検討した特別な jmf 形式のファイル以外のファイル)のマッピングを見ていきます。
与えられたようにデータをレイアウトする方法と、このレイアウトを J 変数のタイプ、ランク、シェイプで表現する方法、つまり適切なマッピングを指定する方法を指定します。
例えば、与えられたファイル G を固定長レコードにレイアウトされたデータで読み込むことを目指すとします。各レコードは 8 文字です。ファイル G が元々、次のように作成されたとします。

```
    G =:  'c:\temp\data.xyz'
    'ABCD0001EFGH0002IJKL0003MNOP0004' fwrite G
32
```

次のステップは、どのような種類の変数がファイル G のデータをマッピングするのに適しているかを決定することです。私たちは、n 行 8 列の文字表を決定します。行数 n はファイ

ル内のデータ量によって決まるため、事前に n を指定しません。
私たちがプロトタイプと呼ぶ n 行 8 列の文字テーブルの小さな例から始めておくと便利です。n の選択は重要ではない。

```
   prototype =: 1 8 $ 'a'
```

これで、マッピングは次のように定義できます。

```
   ] mapping =: ((3!:0) ; (}. @: $)) prototype
+-+-+
|2|8|
+-+-+
```

マッピングはボックス化されたリストで あることがわかります。最初の項目はデータ型です。ここでは、「文字」を意味する 2 は、3 ! : 0 プロトタイプによって生成されます。2 番目の項目は、プロトタイプの末尾のディメンション(つまり、最初のものを除くすべて)です。ここでは 8 すべてが、最初のある 1 8 によって生成、 プロトタイプ: (。$} @) 。このようにして、マッピングは「n 行 8 文字」を表現または符号化する。
ここで、マッピングは左の引数として(ダイアディック)map_jmf_に渡されます。ファイル G を名前 W を選択する変数にマッピングします。

```
   mapping map_jmf_ 'W'; G
```

W は現在変数である ことがわかります。その値はファイル内のデータです。

W	$ W
ABCD0001 EFGH0002 IJKL0003 MNOP0004	4 8

私たちは慣習的な方法でデータを修正することができます:

```
   ] W =:  'IJKL9999' 2} W
ABCD0001
EFGH0002
IJKL9999
MNOP0004
```

私たちができないことは、ファイル G のすべてのスペースが既に持っているデータによって占有されているため、データに別の行を追加することです。

W	W =: W , 'WXYZ0000'
ABCD0001 EFGH0002 IJKL9999 MNOP0004	error

変数 W のマッピングを解除して ファイル G を閉じます。

```
   unmap_jmf_ 'W'
0
```

28.4.5 マップされた変数は特別です [Mapped Variables Are Special]

ファイルを変数にマッピングすることにより、プログラマは機能性と利便性に大きな利点があります。
これらの利点のために支払われるべき価格は、通常の変数には適用されないマップされた変数に適用するいくつかの考慮事項があるということです。プログラマは、これらの考慮事項を認識し、管理する必要があります。これは、このセクションと次のセクションでの話題です。
A が通常の変数であり、マッピングされていない場合は、割り当て B =: A では A の値が実際に B にコピーされます。その後の A への変更は B の値に影響しません。

A =: 1	B =: A	B	A =: 2	B
1	1	1	2	1

対照的に、ファイルにマップされた変数を考えてみましょう。ファイルが非常に大きい場合、値の別のコピーのための十分なスペースがない可能性があります。したがって、コピーは避けなければならない。
前の例を、A がマップされた変数である場合と比較してください 。

```
   map_jmf_ 'A'; F
```

A =: 1	B =: A	B	A =: 2	B
1	1	1	2	2

B は A の変化に伴って変化することが 分かる。効果で B = A があることを意味 B は、別の名前である A、しないの値のコピー。つまり、A と B の両方は、ファイル内の値と同じものを参照します。
したがって、A が B の変化に伴って変化する場合もある 。

A	B =: 7	A
2	7	7

マップされた変数に適用される明示的な動詞を考えてみましょう。ここで、y はファイル内のデータの別の名前になります。したがって、y への代入(ローカル代入さえも)によって、ファイル内のマップされた変数が意図せず変更される可能性があります。例えば

```
   foo =: 3 : ' 3 * y =. y + 1'
```

foo 2	A	foo A	A
9	7	24	8

28.4.6 アンマップ再検索 [Unmapping Revisited]

マッピングされたファイルと変数の現在のステータスは、J システムによって「マッピングテーブル」に保持されます。マッピングテーブルは式 showmap_jmf_ ''を入力することで表示できます が、現時点では選択した列のみを表示するユーティリティ関数があります。

```
   status =: 0 1 9 & {"1  @: showmap_jmf_
   status ''
+-------+-----------------+----+
|name   |fn               |refs|
+-------+-----------------+----+
|A_base_|c:¥temp¥persis.jmf|3   |
+-------+-----------------+----+
```

現在、ロケールベースの変数 A はファイル F(persis.jmf)にマップされています。
"refs"では、値 3 はファイル F のデータが 3 つの参照の対象であることを意味します。これらのうちの 1 つは変数 A です。もう 1 つは変数 B(A の別の名前であることがわかっています)で、3 つ目はシステムそのものです。
変数 A と B の両方が存在する:

A	B
8	8

簡単にするために、ファイルを閉じるための推奨される手順は、元のマップされた変数 A の代替名である B などのすべての変数をまず消去することです

```
   erase <'B'
1
```

ステータスは、参照数が減少したことを示します。

```
   status ''
+-------+-----------------+----+
|name   |fn               |refs|
+-------+-----------------+----+
|A_base_|c:¥temp¥persis.jmf|2   |
+-------+-----------------+----+
```

これで A をアンマップできます。

```
   unmap_jmf_ 'A'
0
```

結果が 0 の場合は、ファイルが閉じられ、A が消去されたことを意味します。ステータステーブルには、エントリがないこと、つまりファイルがマップされていないことが示されます。

```
   status ''
+----+--+----+
|name|fn|refs|
+----+--+----+
```

A が F にマッピングされ 、B が A の別の名前 である状況を再現しましょう。したがって、ファイル F への(データの)参

照は 3 つあります。

```
   map_jmf_ 'A'; F
   B =: A
   status ''
+-------+-------------------+----+
|name   |fn                 |refs|
+-------+-------------------+----+
|A_base_|c:¥temp¥persis.jmf|3   |
+-------+-------------------+----+
```

F を参照するすべての変数を消去するとどうなりますか？

```
   erase 'A';'B'
1 1
   status ''
+-------+-------------------+----+
|name   |fn                 |refs|
+-------+-------------------+----+
|A_base_|c:¥temp¥persis.jmf|1   |
+-------+-------------------+----+
```

変数 A がなくても 、名前 A の下に単一の参照があることがわかります。この単一の参照は、ファイル F がまだマップされていないという事実を反映しています。
したがって、以前にファイル F が変数 A にマップされると言ったとき、ファイル F が名前 A にマップされ、その名前の変数が作成されると言うのがより正確です。変数が後で消去されても、名前 A は引き続きマップされたファイルを識別し、unmap の引数として使用できます。

```
   unmap_jmf_ 'A'
0
   status ''
+----+--+----+
|name|fn|refs|
+----+--+----+
```

詳細は、「マップされたファイル」ラボを参照してください。

☆第 28 章終了

第 29 章エラー処理 [Error Handling]

この章の計画は、プログラミングエラーを見つけて処理するための J の機能のいくつかを調べることです。デバッグ戦略を検討することはこの章の範囲を超えていますが、(私の見解では)主張の使用が推奨されています。我々は見る:
主張
障害発生後も継続
中断された実行
プログラムされたエラー処理

29.1 主張 [Assertions]

ある程度プログラムを自己チェックすることができます。次に、カーペットの領域の価格を計算する動詞の例を示します。単位面積、長さ、幅の 3 つの数値のリストがあります。

```
   carpet =: 3 : 0
*/ y
)

   carpet 2 3 4
24
```

例として、計算* / y が大きく、問題があると仮定し、結果が正しいという確信を欲する。結果が合理的であることを少なくとも確認することができます。私たちは、カーペットの価格は、例えば$ 10〜$ 10,000 の間にあると考えています。結果 p は 10〜10000 でなければならないと主張し、カーペット を再定義することができます。

```
   carpet =: 3 : 0
p =. */y
assert. p >: 10
assert. p <: 10000
p
)
```

主張が真(または「すべて真」)として評価される場合、他の効果はなく、計算が進む。

```
   carpet 2 3 4
24
```

主張が偽と評価された場合、計算は終了し、指示が与えられる:

```
   carpet 0 3 4
|assertion failure: carpet
|   p>:10
```

主張は明示的な定義内でのみ行うことができます。「制御ワード」、すなわち、関数ではなく構文の要素です。
主張をどこに置くのが有用か、何が有効に宣言できるのかは、常に判断の対象となります。主張は、主張自体にエラーを起こすことは非常に望ましくないため、できるだけシンプルに保つのが最善です。
関数の引数の正当性をチェックするアサーションを作成すると便利なことがよくあります。たとえば、カーペットの引数 y は 3 つの厳密に正の数のリストでなければならないと主張することができます。
主張の順序は重要です。たとえば、それらの数値の値をチェックする前に、数値があることを確認する必要があります。名詞の型は 3 ! : 0 で与えられます。ここでは、整数(型= 4)または実数(型= 8)が必要です。

```
   carpet =: 3 : 0

assert. (3!:0 y) e. 4 8   NB. numeric
assert. 1 = # $ y         NB. a list (rank = 1)
assert. 3 = # y           NB. of 3  items
assert. *. / y > 0        NB. all positive

p =. */y

assert. p >: 10
assert. p <: 10000

p
)

   carpet 2 3 4
24

   carpet 'hello'
|assertion failure: carpet
|   (3!:0 y)e.4 8
```

29.1.1 主張と暗黙のスタイル [Assertions and the Tacit Style]

主張は正しかったです。暗黙のスタイルは、鮮明さと明快さには良いです。
しかしながら、2 つは容易に組み合わされない。明らかに、主張の自然な場所は明示的な定義の行です。対照的に、暗黙の定義は主張のための場所を提供しません。
純粋に暗黙の定義に主張を追加するにはどうすればよいでしょうか？関数の引数についての主張を行うことができるのは、何もないよりはるかに良いでしょう。ここに可能性があります。
純粋に暗黙の定義の例があるとします。

```
   sq =: *:
```

sq の 引数は数値でなければならない、つまり述語を満たさなければならないと主張したい:

```
   is_number =: 4 8 16 128 e。〜(3！: 0)
```

今、私たちの目的は、再定義することである SQ を以前の定義を使用しながら、SQ を。この目的のために便利なのは、以下に定義されている結合 ASSERTING である。
私たちは書くことができます

```
   sq =: sq ASSERTING is_number
```

我々は見る:

```
   sq
3 : 0
assert. (is_number) y
(*:) y
)
   sq 3
9
   sq 'abc'
|assertion failure: sq
|   (is_number)y

The definition of ASSERTING is:
   ASSERTING
2 : 0
    U =. 5!:5  < 'u'
```

```
    if. (< U) e. nl 3 do. U =. 5!:5 < U end.
    V =. 5!:5  < 'v'
    z =: 'assert. (', V , ') y', LF
    z =. z , '(', U,  ') y'
    3 : z
)
```

ASSERTING 結合詞は、この結果を簡単に検査できるように、この文字列構築スタイルで記述されています。新しい sq は述語 is_number と古い sq の値(名前ではない！)を組み合わせていることがわかります。最後に、ここで定義されている ASSERTING はモナド動詞に対してのみ良いことに注意してください。

29.1.2 アサーションの有効化と無効化 [Enabling and Disabling Assertions]

正確さが確信できたら、特にパフォーマンスが問題である場合には、プログラムからアサーションを削除することを検討できます。別の可能性は、アサーションをそのまま残すことですが、アサーションを無効にすることです。この場合、アサーションされた式は評価されず、アサーションは常に成功します。アサーションを有効または無効にする組み込みの関数 9: 35 があります。例えば:

```
   (9!:35) 0      NB. disable assertions
   carpet 0 3 4   NB. an error
0
   (9!:35) 1      NB. enable assertions
   carpet 0 3 4   NB. an error
|assertion failure: carpet
|   */y>0
```

組み込み関数 9: ！34 のアサーションが有効になっているかどうかをテストします。現在、彼らは:

```
   9!:34 ''   NB. check that assertions are enabled
1
```

29.2 失敗後の継続 [Continuing after Failure]

障害発生後も継続する方法はいくつかあります。

29.2.1 ノンストップスクリプト [Nonstop Script]

プログラムをテストする際には、一連のテストのためのスク

リプトを記述すると便利です。次に、テストスクリプトの例を示します。

```
   (0  :  0)  (1!:2)  <'test.ijs'      NB.  create
test-script

NB. test 1
carpet 10  0 30

NB. test 2
carpet 10 20 30
)
```

テストで間違った結果が得られるか、または完全に失敗する可能性があります。つまり、システムによって終了する可能性があります。テストに失敗してもスクリプトは、組み込みの動詞 0！：10 または 0！：11 を使用してスクリプトを実行することによって、スクリプトを続行することができます

```
   0!:11  <'test.ijs'                    NB.  execute
test-script

   NB. test 1
   carpet 10  0 30
|assertion failure: carpet
|   */y>0

   NB. test 2
   carpet 10 20 30
6000
```

29.2.2 制御構造の試行とキャッチ [Try and Catch Control Structure]

ここでは、単語リストを使って英語の単語をフランス語に翻訳する動詞の例を示します。

```
   English =: 'one'; 'two';  'three'
   French  =: 'un';  'deux'; 'trois'

   ef =:  3 : '> (English i. < y) { French'
```

リストにない単語はエラーになります。

```
   EF =: 3 : 'try. ef y catch. ''don''''t know'' end.'
```

ef 'two'	ef 'seven'
deux	error

このエラーは try で処理できます。キャッチ。終わり。制御構造。(第 12 章で は制御構造を紹介しています)

```
   EF =: 3 : 'try. ef y catch. ''don''''t know'' end.'
```

EF 'two'	EF 'seven'
deux	don't know

このスキームは、

```
try. B1 catch. B2 end.
```

ブロック B1 を実行することを意味する。B1 が失敗した場合にのみ、ブロック B2 を実行します。

29.2.3 有害な結合 [Adverse Conjunction]

最後の例の暗黙のバージョンは、 "有害な"連想: :(コロンコロン)で書くことができます 。

```
   TEF =: ef ::( '分からない' _)
```

TEF 'two'	TEF 'seven'
deux	don't know

::の 左と右の引数は両方の動詞であることに注意してください。スキームは次のとおりです。

```
          (f :: g)y
```

評価する: fy。場合にのみ、FY は失敗し、評価 GY

29.3 延期された実行 [Suspended Execution]

プログラムの一例として、我々は持っていると仮定すると、デバッグする動詞メイン支援動詞を使用していますプラス

```
   main  =: 3 : 0
k =. 'hello'
z =. y plus k
'result is'; z
)
   plus =: +
```

明らかに、メインにエラーがあります。文字列 k は、プラスで期待される数値引数と矛盾しています。

たとえば、 キーボードに main 1 と入力すると、エラーが検

出されるとプログラムが終了し、エラー・レポートが表示され、ユーザーはキーボードからの入力を求められます。

```
   main 1
|domain error: plus
|   z=.y     plus k
```

エラーの原因に関する詳細情報を収集するために、制御がキーボードに戻ったときにプログラムが終了するのではなく一時停止されるようにすることができます。一時停止を有効にするには、main を再度実行する前にコマンド(13 ! : 0)1 を使用します。

```
   (13: 0)1
```

今とき、メイン再実行され、我々はわずかに異なるエラーメッセージが表示されます

```
   main 1
|domain error: plus
|plus[:0]
```

この時点で実行は中断されます。サスペンド状態では、式を入力して評価することができます。プロンプトは、中断された状態を識別する 6 つのスペース(通常の 3 ではなく)です。

```
      1 + 1
2
```

キーボードでこの式を入力することにより、「実行スタック」と呼ばれるもの(選択列)を表示することによって、計算の現在の状態を見ることができます。

```
      0 2 6 7 8 { ″ 1 (13!:13 '')
+----+-+---------+----------+-+
|plus|0|+-+-----+|          |*|
|    | ||1|hello||          | |
|    | |+-+-----+|          | |
+----+-+---------+----------+-+
|main|1|+-+      |+-+-----+| |
|    | ||1|      ||k |hello|| |
|    | |+-+      |+-+-----+| |
|    | |         ||y |1    || |
|    | |         |+-+-----+| |
+----+-+---------+----------+-+
```

スタックはテーブルであり、現在実行中の各関数の行があり

ます。私たちは、ことがわかりプラスは、エラーが検出された機能であり、プラスが から呼び出されるメイン。
スタックには 9 個のカラムがあり、そのうちの 5 個を表示用に選択しました(カラム 0 2 6 7 8)。列は次のとおりです。

0	中断された関数の名前。名前付き関数だけがスタックに表示されます。
1	(上に示されていない)エラー番号またはエラーでない場合は 0
2	行番号。 プラスはライン 0 でサスペンドされ、メインはライン 1 でサスペンドされます
3	(上には示されていない)名前クラス: 副詞、結合詞または動詞を表す 1,2 または 3
4	(上には示されていない)サスペンド機能の線形表現
5	(上に示されていない)定義がロードされたスクリプトの名前
6	引数の値 プラスは引数 1 と'hello'で呼び出されました
7	ローカル変数の名前と値。プラスながら暗黙動詞であることは、ローカル変数を持っていない 主が有する K、また y は、明示的な関数の引数はローカル変数と見なされるので、。
8	アスタリスク、または空白。プラス、それはサスペンションが発生された機能であることを示すために*印を付しています。通常これはスタック上のトップ関数です(必ずしもそうではありませんが、以下で説明します)。

中断状態では、通常の方法で名前を入力することによって、グローバル変数を検査できます。この単純な例には何もありません。
最後に、中断された実行を終了し、中断状態から抜け出すには、次の式を入力します。

```
   (13: 0)1
```

29.4 プログラムされたエラー処理 [Programmed Error Handling]

デフォルトでは、サスペンドが有効になっているときにエラーが発生すると、プログラムは中断し、キーボードからの入力を待ちます。
キーボードから入力するのではなく、エラーが発生したときに、私たち自身のエラー処理ルーチンが自動的に入力されるようにすることができます。
次のようにしてエラーを処理することにしました。
システムによって生成されたエラーメッセージを表示する
スタックの選択された列を表示する
サスペンドされた関数の実行を短くし、返されるものではなく値'error'を返すようにします。
プログラムの実行を再開する。(これは、さらなるエラーのカスケードにつながる場合もあります)。
この一連のアクションを実行する動詞は次のとおりです。

```
   handler =: 3 : 0
(1!:2&2) 13!:12 ''              NB. display error message
(1!:2&2) 0 2 6 7 8 {"1 (13!:13 '') NB. display stack
13!:6 'error'                   NB. resume returning 'error'
)
```

次のステップは、この動詞をエラーハンドラとして宣言することです。これを行うために、「潜伏表現」と呼ばれるものに適切な値を設定します。潜在的な式は文字列で表され、システムがサスペンド状態になる直前に自動的に実行されます。
潜在的な表現は、13 !: 14 と 13 !: 15 と照会して設定することができます。潜在的な式の現在の値は何ですか？

```
   13: 14 ''
```

ヌル文字列。潜在式を文字列に設定し、 "動詞ハンドラを実行する"という意味の式を表します。

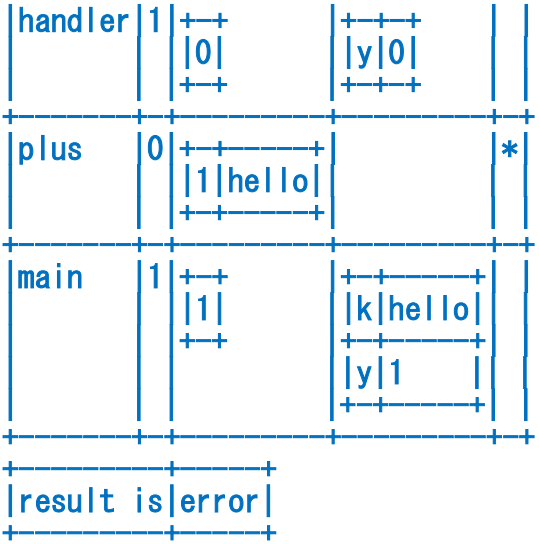
```
   13!:15 'handler 0'
```

今度は、停止が有効になっていることを確認します。

```
   (13!:0) 1 NB. enable suspension
```

メインでデバッグを実行しよう

```
   main 1
|domain error: plus
|plus[:0]
+-------+-+--------+--------+-+
```



一番上のスタックフレームはハンドラ用であることがわかります。なぜなら、スタックを表示する要求が発行されたときにハンドラに入っているからです。一時停止機能はプラスです。

表示結果がエラーであることが示され、plus がハンドラによって提供され た値('error')を返したことを示します。

☆第 29 章終了

第 30 章: 疎配列 [Sparse Arrays]

30.1 はじめに [Introduction]

J の疎配列機能は、アレイの要素の多くがすべて同じであれば、大規模な配列をコンピュータに適度な量のメモリに格納することを可能にします。この場合、何度も発生する値は 1 回だけ格納する必要があります。
一例として、ネットワークを記述している接続配列に対して、疎な表現が考えられるかもしれない。この章では、疎配列を扱うための J 機構を見ていきます。
D がその要素の大部分が同じ配列で あるとすると、

```
   ] D =: 2 3 4(2 2; 3 6; 4 4)} 16 16 $
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0
0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

この配列は、「疎配列」と呼ばれるコンパクトな形式で格納できます。ここでは、非ゼロ要素だけが格納領域を占有します。Sparse(疎)でない通常の配列は、「dense(稠密)」な配列と呼ばれることがあります。
組み込み関数 $. があります。(ドルドット)を使用して密な配列から疎配列を計算します。

```
   S =: $. D
```

多くの目的のために密な配列 D とスパース配列 S は等価である: S は一致 D を、従ってそれは同じ寸法を有し、および索引付けに同じ結果を与えます。

S -: D	($S) -: ($D)	((< 0 0){ S) -: (<0 0) { D

1	1	1

30.2 疎配列はコンパクト [Sparse Array is Compact]
マトリックス D と比較すると、マトリックス S は、D において何度も現れる値が S 内に 1 回だけ格納されるため、経済的である。この値はの「まばらな要素」として知られている S、または「ゼロ」要素 S。S の場合は 0 になりますが、常に 0 にする必要はありません。
組み込みの 7!: 5 で配列が占有するストレージのサイズを測定することができます。S の大きさ(D の疎な形)が D の大きさより小さいことがわかります。

7!:5 <'S'	7!:5 <'D'
384	2048

30.3 疎配列の検査 [Inspecting A Sparse Array]
標準ライブラリには 有用な関数データ型があります。引数の型を示します。

datatype D	datatype S
integer	sparse integer

構築された動詞 3!: 0 もその引数の型を与えることを思い出してください。疎配列の場合、3!:0 によって報告される可能な型は

1024	スパースブール	sparse boolean
2048	疎文字	sparse character
4096	スパース整数	sparse integer
8192	まばらな浮動小数点	sparse floating point
16384	疎な複合体	sparse complex
32768	スパースボックス	sparse boxed

S を通常の方法で 表示すると、配列の使い慣れた表現ではなく、インデックス値のペアのリスト(それぞれの例では非ゼロ要素)のペアが表示されます。

```
   S
```

```
2 2 | 2
3 6 | 3
4 4 | 4
```

この表示は、S の疎要素が実際には整数 0 であることを示していません。これを示すために、動詞 3&$を使用して疎要素を抽出することができます。

se =: 3 $. S	datatype se
0	integer

ここで、S から新しい配列を計算すると

```
T =: S + 5
```

我々はそれを見る T はまばらである、とのまばらな要素 T はゼロではないが、5

T	3 $. T
2 2 | 7 3 6 | 8 4 4 | 9	5

まばらな配列を表示するもう 1 つの方法は、単に 0&$で密に変換することです。

```
   view =: 0 & $.
```

T	view T
2 2 | 7 3 6 | 8 4 4 | 9	5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 7 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 8 5 5 5 5 5 5 5 5 5 5 5 5 5 5 9 5

	5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

30.4 疎配列による計算 [Computing with Sparse Arrays]
スパース配列を持つ計算は、結果として疎配列を生成する傾向があることを除いて、密配列とほぼ同じです。上記の S + 5 でこれを確認しました。別の例があります。上合計 T はまばらである列和のベクトルを生成します。

```
   ] V =: + / T
2 | 82
4 | 84
6 | 83
```

しかしの「ゼロ」要素 V は、の「ゼロ」要素の列の合計であり、T

```
   3 $. V
80
```

執筆時点では、高密度アレイと比較して疎アレイを用いて行うことができるものにはまだいくつかの制限があります。詳細については $. の下にある辞書を参照してください。

30.5 疎配列の作成 [Constructing A Sparse Array]
この時点で、いくつかの用語を定義すると役に立ちます。文脈によれば、数字 0 または 1 または 0.0 または 1.0 は、ブール値または整数または実数として有効であり得ることに最初に留意されたい。しかし、コンテキストが存在しない場合、J システムはそれらすべてを実際にブール値にします。

datatype 0	datatype 1	datatype 0.0	datatype 1.0
boolean	boolean	boolean	boolean

あいまいな型の値をいくつか定義すると便利です。

INTEGERZERO =: 3 - 3	datatype INTEGERZERO

0	integer

INTEGERZERO =: 3 - 3	datatype INTEGERZERO
0	integer

REALZERO =: 0.0*0.1	datatype REALZERO
0	floating

REALONE =: ^ 0	datatype REALONE
1	floating

現在、疎配列に戻すには、必要な形状と型の空の配列を作成することから始めて、実際のデータは作成しないでください。空の配列は、式を評価することによって構築されます。

```
1 $. shape;axes;zero
```

どこで、

●Shape(形状)は、寸法を指定する。

●Axes(軸)は、これらの次元のどれが疎であるかを軸番号のリストとして指定します。たとえば、2 次元の場合、両方がスパースであれば、リストは 0 1 になります。

今のところ、疎な配列の例では、すべての軸が疎ですが、以下では、疎軸と密な軸が混在していることがわかります。

●zero は "zero"要素の値を指定し、したがって配列全体の型を指定します。あいまいな値が明らかに必要です。

場合は、ゼロが省略されているデフォルトです REALZERO。両方の軸とゼロが省略されている場合、デフォルトはすべての軸スパースと REALZERO です。

だから、すべての次元(つまり、軸 0 と軸 1)に疎な 6 行 6 列の配列を構築するには、0 というゼロ要素を持つ整数型を書くことができます:

```
U = 1 $。6 6; 0 1; INTEGERZERO
```

この時点で、U は空で、つまりすべてゼロであるため、何も表示されません。

```
   U
```

それに少数の非ゼロ要素を挿入することでそれを埋め込む

```
   U =: 4 5 6 7(0 0; 1 1; 2 2; 3 3)} U
```

そして、することを確認し U は、我々はそれを見て期待するものです。

```
   U
4 0 0 0 0 0

5 0 0 0 0
0 6 0 0
0 0 0 7 0
0 0 0 0 0 0
0 0 0 0 0 0
```

30.6 疎軸と稠密(ちゅうみつ)軸 [Sparse and Dense Axes]

配列は、いくつかの軸では疎であり、他の軸では(稠)密であるかもしれません。 以下の例では、W はその第 1 の軸上で疎であり、その第 2 の軸上では密集している。疎軸のリストはちょうど 0 です。

```
   saw   =: ,0   NB. sparse axes for W
   W =: 1 $. 3 5; saw ; INTEGERZERO
   W =: 4 5 6 (0 1; 0 2; 1 3) } W
```

期待通りに見える:

```
   view W
0 4 5 0 0
0 0 0 6 0
0 0 0 0 0
```

しかし、2 つの密な行としてのみ格納されていることがわかります。

```
   W
0 | 0 4 5 0 0
1 | 0 0 0 6 0
```

第 2 軸の軸だけの配列との比較は、疎軸のリストが 1 であるためです

```
   saz=: ,1   NB. sparse axes for Z
```

```
   Z =: 1 $. 3 5; saz; INTEGERZERO
   Z =: 4 5 6 (0 1; 0 2; 1 3) } Z
```

Z は W のように見える

```
   view Z
0 4 5 0 0
0 0 0 6 0
0 0 0 0 0
```

しかし、それは 3 つの密な列として格納されていることがわかります。

```
   Z
1 │ 4 0 0
2 │ 5 0 0
3 │ 0 6 0
```

30.7 疎配列の解体 [Deconstructing a Sparse Array]

上で述べたように、U 自身を表示すると、配列の使い慣れた表現ではなく、各非ゼロ要素の 1 組のインデックスと値のペアのリストが表示されます。

```
   U
0 0 │ 4
1 1 │ 5
2 2 │ 6
3 3 │ 7
```

各ペアからインデックスを抽出して、U のインデックス配列と呼ばれるものを得ることができます。これは普通の高密度配列です。

```
   4 $. U
0 0
1 1
2 2
3 3
```

各ペアから値を抽出するには、

```
   5 $. U
4 5 6 7
```

上記のとおり、0&$。スパースから密な配列を生成します:

```
   0 $. U
4 0 0 0 0 0
0 5 0 0 0 0
```

```
0 0 6 0 0 0
0 0 0 7 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

30.8 疎配列と関係 [Sparse Array From Relation]

次に、データをリレーション(つまりテーブル)として表現する代わりに、データをスパース配列として表現する方法を検討します。
ポイントは、スパース配列がデータとのいくつかの計算の関係よりも便利かもしれないということです。このように、疎な配列とリレーションの間の変換に興味があります。
例えば、所与の関係 R が、様々な都市における様々な商品の売上を表すと仮定する。

```
   'Pa Qu Ro Sy' =: s: ' Paris Quebec Rome Sydney'
   'Ap Ba Ch Da' =: s: ' Apples Bananas Cherries Damsons'

   R =: (". ;. _2) 0 : 0
Ap ; Pa; 99
Ap ; Qu ; 50
Ba ; Qu ; 10
Ch ; Ro ; 19
Da ; Sy ; 110
Da ; Pa ; 88
)

   R
+---------+-------+---+
|`Apples  |`Paris |99 |
+---------+-------+---+
|`Apples  |`Quebec|50 |
+---------+-------+---+
|`Bananas |`Quebec|10 |
+---------+-------+---+
|`Cherries|`Rome  |19 |
+---------+-------+---+
|`Damsons |`Sydney|110|
+---------+-------+---+
|`Damsons |`Paris |88 |
+---------+-------+---+
```

関係 R を以下のように疎配列に変換することができます。
最初に、最初の列のすべての可能な値のセットであるドメインを確立する必要があります。 R から計算することができま

す：

```
   ] Fru =:  > ~. 0 { |: R
`Apples `Bananas `Cherries `Damsons
```

第 2 列のドメインについても同様です。

```
   ] Cit =: > ~. 1 { |: R
`Paris `Quebec `Rome `Sydney
```

これで、最初の列がそのドメインにインデックスに変換されました。

```
   ] r =: Fru i. > 0 { |: R
0 0 1 2 3 3
```

2 番目の列についても同様です。

```
   ] c =: Cit i. > 1 { |: R
0 1 1 2 3 0
```

3 番目の値

```
   ] v =: > 2 { |: R
99 50 10 19 110 88
```

今度は#Citによって次元#Fruの空の疎な配列を構築します。デフォルトでは、疎軸は 0 と 1 になり、 "ゼロ"要素はREALZEROになります。関数1&$。空の配列を生成します。

```
   A =: (1 & $.) (#Fru) , (#Cit)
```

通常の方法で値を挿入します。

```
   A =: v (<"1 r,.c) } A
```

私たちが期待していることを確認してください：

```
   view A
99 50  0   0
 0 10  0   0
 0  0 19   0
88  0  0 110
```

行と列のラベル付けで A を表示するには、行ラベルのリストは上記で計算された Fru であり、列ラベルのリストは Cit：

```
   (a:, <"0 Cit), (<"0 Fru) ,. (<"0 view A)
+--------+------+------+----+-------+
|        |`Paris|`Quebec|`Rome|`Sydney|
+--------+------+------+----+-------+
```

```
|`Apples  |99    |50    |0    |0     |
+--------+------+------+-----+------+
|`Bananas |0     |10    |0    |0     |
+--------+------+------+-----+------+
|`Cherries|0     |0     |19   |0     |
+--------+------+------+-----+------+
|`Damsons |88    |0     |0    |110   |
+--------+------+------+-----+------+
```

これで元のリレーションから疎配列を作成し終えました。データとして配列として計算できます。
たとえば、各都市の売上の合計値は、次の式で表されます。

```
   +/ A
0 | 187
1 |  60
2 |  19
3 | 110
```

これは疎配列なので、通常の表示を取る：

```
   view +/ A
187 60 19 110
```

30.9 疎配列からの関係

サークルを完成させるために、次に、疎配列 A からリレーションを生成する方法を次に見ていきます 。

```
   A
0 0 | 99
0 1 | 50
1 1 | 10
2 2 | 19
3 0 | 88
3 3 | 110
```

最初のステップは、非ゼロ要素のインデックス配列を取得することです。

```
   ] INDS =: 4 $。A
0 0
0 1
1 1
2 2
3 0
3 3
```

そして次の値、

```
   ] VALS =: 5 $。A
99 50 10 19 88 110
```

上記で計算し たドメイン Fru を索引付けすることによって生成される関係の最初の列。2 番目の列は Cit と同様に生成されます。

```
   ] c0 =: (0 { |: INDS) { Fru
`Apples `Apples `Bananas `Cherries `Damsons `Damsons
   ] c1 =: (1 { |: INDS) { Cit
`Paris `Quebec `Quebec `Rome `Paris `Sydney
```

最後に、スパース配列から回収された関係が、

```
   (<"0 c0) ,. (<"0 c1) ,. (<"0 VALS)
+---------+-------+---+
|`Apples  |`Paris |99 |
+---------+-------+---+
|`Apples  |`Quebec|50 |
+---------+-------+---+
|`Bananas |`Quebec|10 |
+---------+-------+---+
|`Cherries|`Rome  |19 |
+---------+-------+---+
|`Damsons |`Paris |88 |
+---------+-------+---+
|`Damsons |`Sydney|110|
+---------+-------+---+
```

☆第 30 章終了

第 31 章: パフォーマンス [Performance]

この章では、パフォーマンス、つまり計算を実行するのにかかる時間と、それを改善する方法について説明します。
J プログラムで優れたパフォーマンスを達成するための 1 つのゴールデンルールがあります。このルールは、いつでもできるだけ多くのデータに動詞を適用しようとすることです。言い換えれば、組み込み関数が配列引数を取ることができるという事実を最大限に活用するために、スカラーではなくベクトルまたは一般的に配列である動詞引数に与えることを試みてください。
この章の残りの部分は、ほとんどがこの単一の点でハープすることから成っています。

31.1 測定された時間の測定 [Measuring the Time Taken]

組み込みの動詞 6！: 2 があります。引数として式を文字列として取り、式を実行する時間を秒単位で返します。例えば、与えられた:

```
   mat =: ? 256 256 $ 256  NB. a random matrix
```

行列を逆にする時間(秒)は次の式で与えられます。

```
   6!:2 '%. mat'
0.075273
```

同じ表現に再び時間を置くと、次のようになります。

```
   6!:2 '%. mat'
0.0734835
```

明らかに、この測定には不確実性がある。6: 2 の二項の場合、いくつかの測定値を平均します。しかし、現在の目的のために、モナディック 6！: 2 を使用して、大まかで準備ができていますが、適切な測定を行います。
いくつかの式の時間を比較する場合、少しのユーティリティ関数を導入すると便利です:

```
   compare =: (; (6!:2)) @: > "0      NB. compare timings

   compare '+/ i. 1000' ; '*/ i. 1000'
+----------+----------+
|+/ i. 1000|2.14228e_5|
+----------+----------+
```

```
|*/ i. 1000|8.44842e_6|
+----------+----------+
```

31.2 パフォーマンスモニタ [Performance Monitor]

6！: 2 と 同様に、パフォーマンスモニタと呼ばれる実行時間を測定するためのもう 1 つの便利な手段があります。明示的な動詞の各行に費やされる時間を示します。
ここでは、メインプログラムと補助機能を使用した例を示します。私たちはそれが何をするのに興味がなく、時間を費やすのに過ぎない。

```
   main =: 3 : 0
    m =. ? 10 10 $ 100x   NB. random matrix
    u =. =/ ~ i. 10       NB. unit matrix
    t =. aux m           NB. inverted
    p =. m +/ . * t
    'OK'
)

   aux =: 3 : 0
    assert. 2 = # $ y        NB. check y is square
    assert. =/ $ y
    %. y
)
```

我々はモニターを始める:

```
   load 'jpm'
   start_jpm_ ''
178570
```

分析する式を入力します

```
   main 0                    NB.  expression to be analyzed
OK
```

利用可能なレポートを表示するには: まず、主な機能:

```
   showdetail_jpm_ 'main'     NB. display measurements
 Time (seconds)
+--------+--------+---+----------------+
|all     |here    |rep|main            |
+--------+--------+---+----------------+
|0.000004|0.000004|1  |monad           |
|0.000079|0.000079|1  |[0] m=.?10 10$100|
|0.000011|0.000011|1  |[1] u=.=/~i.10  |
|0.042640|0.000005|1  |[2] t=.aux m    |
|0.062151|0.062151|1  |[3] p=.m+/ .*t  |
```

```
|0.000015|0.000015|1  |[4] 'OK'          |
|0.104900|0.062265|1  |total monad       |
+--------+--------+---+------------------+
```

私たちは補助機能を見たいかもしれません:

```
   showdetail_jpm_ 'aux'
 Time (seconds)
+--------+--------+---+------------------+
|all     |here    |rep|aux               |
+--------+--------+---+------------------+
|0.000004|0.000004|1  |monad             |
|0.000004|0.000004|1  |[0] assert. 2=#$y |
|0.000010|0.000010|1  |[1] assert. =/$y  |
|0.042617|0.042617|1  |[2] %.y           |
|0.042635|0.042635|1  |total monad       |
+--------+--------+---+------------------+
```

明らかに、メインは、ほとんどの時間を 2 行目と 3 行目に費やしています。ライン 2 の「すべて」の下の時間は、ライン 2 の「ここ」(すなわち、メイン)の時間に aux の「トータル」の時間。

31.3 ゴールデンルール: 例 1 [The Golden Rule: Ex. 1]

スカラー引数を取ることを明確に意図した関数の例を次に示します。

```
   collatz =: 3 : 'if. odd y do. 1 + 3 * y else. halve y end.'
       odd =: 2 & |
       halve  =: -:
```

ベクトル引数を使うと、結果が間違っています。

```
With a vector agument it gives the wrong results
   collatz 2 3 4 5 6 7 8 9
1 1.5 2 2.5 3 3.5 4 4.5
```

したがって、引数をスカラーにするランクを指定する必要があります。

```
   (collatz "0) 2 3 4 5 6 7 8 9
1 10 2 16 3 22 4 28
```

これはゴールデンルールのための機会です。ここでベクトル引数のために設計されたバージョンです：

```
   veco =: 3 : 0
    c =. odd y
    (c * 1+3*y) + (1 - c) * (halve y)
```

```
)
```

結果は同じです：

```
   data =: 1 + i. 1e5
   (collatz"0 data) -: (veco data)
1
```

ここにバリエーションがあります。 副詞 } は、ビットベクトル奇数 y に適用されて動詞を生成し、この動詞のモナドの場合は、その引数の第 1 および第 2 の項目を一緒に噛み合わせる。

```
   meco =: 3 : 0
    c =. odd y
    (c }) (halve y) ,: (1+3*y)
)

1
   (meco data) -: veco data
```

ベクターのバージョンは、より高速です：

```
   compare 'collatz"0 data '  ;  'veco data ' ; 'meco data'
+--------------+----------+
|collatz"0 data |0.315611  |
+--------------+----------+
|veco data      |0.0280563 |
+--------------+----------+
|meco data      |0.00665947|
+--------------+----------+
```

31.4 GREx.2: Conway の「人生」[Golden Rule Ex.2: Conway's "Life"]

JH Conway の著名な「Game of Life」には紹介は必要ありません。Rosetta Code の J には次のようなバージョンがあります。

```
   pad=: 0,0,~0,.0,.~]
   life=: (_3 _3 (+/ e. 3+0,4&{)@,;._3 ])@pad
```

開始パターンを提供するために、y 行 y 列のブール値行列で r ペントミノを生成する関数 rp があります。

```
   rp =: 3 : '4 4 |.  1 (0 1; 0 2; 1 0; 1 1; 2 1) } (y,y) $ 0'

   ] M =: rp 8
```

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

   life  M
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 0
0 0 0 0 1 0 0 0
0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0
```

私たちは、ことがわかりライフ動詞が含まれている; ._ 3 - それは、そのセルの 3×3 近辺に取り組んで、個別に各セルの隣人の数を計算します。
これに対して、ここでは、プレーン全体を移動させて各セルをその隣接セルと整列させることによって、すべての隣接セル数を一度に計算するバージョンがあります。

```
   sh   =: |. !. 0
   E    =: 0 _1 & sh
   W    =: 0 1  & sh
   N    =: 1    & sh
   S    =: _1   & sh
   NS   =: N + S
   EW   =: E + W
   NeCo =: NS + (+ NS) @: EW       NB. neighbour-count
   evol =: ((3 = ]) +. ([ *. 2 = ])) NeCo
```

最後の行は、(neighbor-count が 3)または("alive"とcountが2)という条件を表します。シフト法evolとRosetta法の寿命は同じ結果を与える。

```
   (life M) -: (evol M)
1
```

ただし、シフト方法は高速です。

```
   G =: rp 200   NB. a 200-by-200 grid
```

```
   e3 =: 'r3 =: life ^: 100 G NB. 100 iterations, Rosetta'
   e4 =: 'r4 =: evol ^: 100 G NB. 100 iterations, shifting'
   compare e3;e4
+-----------------------------------------------+-------+
|r3 =: life ^: 100 G NB. 100 iterations, Rosetta |9.66706|
+-----------------------------------------------+-------+
|r4 =: evol ^: 100 G NB. 100 iterations, shifting|0.24398|
+-----------------------------------------------+-------+
```

もう一度正しいかどうかを確認する:

```
   r3 -: r4
1
```

31.5 GR Ex.3: 関係の参加 [Golden Rule Ex.3: Join of Relations]

31.5.1 予選 [Preliminaries]

第 18 章 から、著者タイトルとタイトル主題の関係を思い出してください。さまざまなサイズのこれらの関係の形でテストデータが必要になります。ランダムな整数からテストデータを生成する動詞を定義すると便利です。(整数は、本目的のためのシンボルの代用として適切である)。引数 y が必要とされる異なるタイトルの数です。

```
   maketestdata =: 3 : 0
    T  =. i. y                          NB. titles domain
    A  =. i. <. 4 * y % 5               NB. authors domain
    S  =. i. <. y % 2                   NB. subjects domain
    AT =. (?. (#T) $ # A) ,. (?. (#T) $ #T)      NB. AT relation
    TS =. (?. (#T) $  _1+# T) ,. (?. (#T) $ #S)  NB. TS relation
    AT;TS
)

   'AT1 TS1' =: maketestdata 12      NB. small   test-data
   'AT2 TS2' =: maketestdata 1000    NB. medium
   'AT3 TS3' =: maketestdata 10000   NB. large
```

31.5.2 ファーストメソッド [First Method]

また、第 18 章から、関係の結合の動詞を思い出してください。これは、さらなる比較の出発点として取り上げます。これを「最初の方法」と呼ぶことができます。

```
   VPAIR =: 2 : 0
    :
    z =.  0 0 $ ''
    for_at. x do. z=.z , |: v (#~"1  u) |: at , "1 y end.
```

```
    ~. z
)

   first  =: (1&{ =  2&{) VPAIR (0 3 & {)
```

AT1	TS1	AT1 first TS1
0 6	0 0	2 2
2 8	3 2	1 4
1 4	7 4	1 1
3 6	0 0	4 4
2 5	0 5	8 2
1 4	4 4	7 4
4 10	1 4	5 2
1 4	10 4	
8 8	8 2	
7 7	4 1	
3 9	1 3	
5 8	5 2	

例えば、著者 5 がタイトル 8 を、タイトル 8 がサブジェクト 2 について､著者 5 がサブジェクト 2 について書いたとする。

31.5.3 第 2 の方法: ブール行列 [Second Method: Boolean Matrix]

もう一つの方法があります。これは、タイトルの等価性のブール値行列を計算します。行 i 列 j は、i {AT の中のタイトルが j {TS のタイトルと等しい場合に真である。著者およびタイトルは、ブール行列を疎表現に変換し、次にそのインデックス行列を取ることによって回復される。

```
   second =: 4 : 0
    'a t'  =. |: x
    'tt s' =. |:  y
    bm     =. t =/ tt      NB. boolean matrix of matches
    sm     =. $. bm        NB. convert to sparse
    im     =: 4 $. sm      NB. index-matrix
    'i j'  =. |: im
    (i { a),. (j { s)
)
```

今度は、第 2 の方法の正当性をチェックします。つまり、最

初のメソッドと同じ結果を返します。私たちは順序を気にしませんし、反復については気にしないので、ソートしたナブが一致すれば、2 つのリレーションが同じであるとしましょう。

```
   (AT2 second TS2)  same (AT2 first TS2)
1
```

今何度目(何回目)か

```
   t1 =: 6!:2 'AT2 first TS2'
   t2 =: 6!:2 'AT2 second AT2'
   t3 =: 6!:2 'AT3 first TS3'
   t4 =: 6!:2 'AT3 second TS3'

   3 3 $ ' '; (#AT2) ; (#AT3) ; 'first' ; t1; t3 ; 'second' ; t2;
t4
+------+---------+--------+
|      |1000     |10000   |
+------+---------+--------+
|first |0.0356744|7.916   |
+------+---------+--------+
|second|0.00337303|0.321806|
+------+---------+--------+
```

ブール値行列を計算する時間はサイズが 2 次的であるため、2 番目の方法の利点はより大きなサイズでは減少することがわかります。パフォーマンスモニタを使用して、時間の経過を確認することができます。

```
   require 'jpm'
   start_jpm_ ''
178570
   z =: AT3 second TS3
   showdetail_jpm_ 'second'  NB. display measurements
 Time (seconds)
+--------+--------+---+---------------+
|all     |here    |rep|second         |
+--------+--------+---+---------------+
|0.000007|0.000007|1  |dyad           |
|0.000329|0.000329|1  |[0] 'a t'=.|:x  |
|0.000208|0.000208|1  |[1] 'tt s'=.|:y |
|0.177003|0.177003|1  |[2] bm=.t=/tt  |
|0.131638|0.131638|1  |[3] sm=.$.bm   |
|0.000009|0.000009|1  |[4] im=:4$.sm  |
|0.000160|0.000160|1  |[5] 'i j'=.|:im |
|0.017383|0.017383|1  |[6] (i{a),.(j{s)|
```

```
|0.326736|0.326736|1  |total dyad      |
+--------+--------+---+----------------+
```

明らかに、2 行目のブール値行列を計算することに多くの時間が費やされました。私たちはこれ以上のことができますか？

31.5.4 第 3 法: 再帰的分割行列 [boolean matrix with recursive splitting]

二次的な時間を避けるための試みです。 引数が特定のサイズよりも小さい場合は、上記の 2 番目の方法を使用します（これは 2 次ですが、小さな引数に対してはそれほど悪くはありません）。
引数が特定のサイズより大きい場合は、それを 2 つの小さな部分に分割し、その間にタイトルが共有されないようにします。 次に、このメソッドは再帰的にパーツに適用されます。
実験すると、「特定のサイズ」はコンピュータ上で約 256 と表示されます。

```
   third =: 4 : 0
    if. 0 = # x do. return.  0 2 $ 3 end.
    if. 0 = # y do. return.  0 2 $ 3 end.
    'a t'  =. |: x
    'tt s' =. |: y
    if. 256 > # x do.
        bm     =. t =/ tt   NB. boolean matrix of matches
        sm     =. $. bm
        im     =. 4 $. sm  NB. index-matrix
        'i j'  =. |: im
       (i { a),. (j { s)
    else.
        p  =:  <. -: (>./t) + (<./t)  NB. choose "pivot" title
        pv =: t <: p
        x1 =. pv # x
        x2 =. (-. pv) # x
        assert. (#x1) < (#x)
        assert.  (#x2) < (#x)
        qv =. tt <: p
        y1 =. qv # y
        y2 =. (-. qv) # y
        assert. (#y1) < (#y)
        assert. (#y2) < (#y)
        (x1 third y1) , (x2 third y2)
    end.
```

```
)

Check correctness :
   (AT2 third TS2) same (AT2 second TS2)
1
```

そしてタイミング。 私のコンピュータ上での実験では、3 番目の方法が成功するスペースを使い果たす 2 番目の方法が示されています。

```
   'AT4 TS4' =: maketestdata  30000
   'AT5 TS5' =: maketestdata 100000
   t4a =: 6!:2 'AT4 second TS4'
   t5  =: 6!:2 'AT2 third TS2'
   t6  =: 6!:2 'AT3 third TS3'
   t7  =: 6!:2 'AT4 third TS4'
   t8  =: 6!:2 'AT5 third TS5'
   a =:  ' ';      (#AT2); (#AT3) ; (#AT4); (#AT5)
   b =:  'second'; t2;     t4;       t4a;  'limit error'
   c =:  'third' ; t5;     t6;        t7  ;  t8

   3 5 $a,b,c
+------+---------+---------+---------+-----------+
|      |1000     |10000    |30000    |100000     |
+------+---------+---------+---------+-----------+
|second|0.00337303|0.321806 |3.05887  |limit error|
+------+---------+---------+---------+-----------+
|third |0.00134601|0.0173196|0.0553821|0.192562   |
+------+---------+---------+---------+-----------+
```

結論として、第 3 の方法は明らかに優れているが、かなり複雑である。

31.6 GR 例 4: マンデルブロセット [Golden Rule Example 4: Mandelbrot Set]

マンデルブロセットは、多くの計算を必要とするフラクタル画像です。以下を書いて、私はこの Wikipedia の記事に役立つことが判明しました 。

画像の計算には、画像内のすべてのピクセルに対して、条件が満たされるまで単一のスカラー関数の反復が必要です。異なるピクセルは異なる反復回数を必要とする。最終的な結果は、各ピクセルの反復カウントの配列です。したがって、マンデルブロセットは不可避的にスカラー計算であるように見

えるかもしれません。次のような意味ではありません。ゴールデンルールが適用されます。

31.6.1 スカラ・バージョン [Scalar Version]

画像の構成は、複素平面上の点のグリッドを選択することから始まり、画像内の各画素に対して 1 つずつである。グリッドを便利に構築するための動詞がここにあります。

```
   makegrid =: 4 : 0
'm n' =. x
'px py qx qy ' =. , +. y
(|. py + (i.m) * ((qy-py) % m-1)) (j.~ /) (px + (i.n)
* ((qx-px) % n-1))
)
```

左の引数は、m 行 n 列のピクセルとしてグリッドを指定します。右の引数は、画像の左下隅と右上隅の位置を複素平面内の点として指定します。
小さなグリッドでデモンストレーションするには:

```
   3 4 makegrid 0j0 3j4
0j4 1j4 2j4 3j4
0j2 1j2 2j2 3j2
  0   1   2   3
```

マンデルブロ画像の場合、これはより適しています:

```
   GRID =:  400 700 makegrid _2.5j_1 1j1
```

ここで は、マンデルブロ画像の全体が複素平面のこの領域内に含まれることが知られているので、右の引数は_2.5j_1 1j1 となるように選択される。この領域は、比率 4:7 の辺を持つ矩形であることに注意してください。左の引数 m-by-n ピクセルは、同じ比率になるように選択されます。
画像は、グリッド内の各ピクセルにマンデルブロ関数を適用することによって計算されます。適切なマンデルブロ関数を次に示します。これは、Wikipedia の記事で概説されている設計に従います。

```
   mfn1 =: 3 : 0
    NB.        y is one pixel-position
    v =. 0j0
    n =. 0
    while. (2 > | v) *. (n < MAXITER) do.
       v  =. y + *: v
```

```
        n =. n+1
    end.
    n
)
```

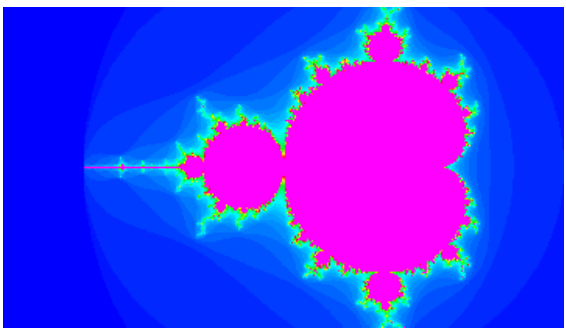
反復の最大回数である MAXITER の 値を選択する必要があります。最大値が高いほど、結果の画像はより複雑になります。現在の目的では、認識可能な画像を与える最大 64 回の反復を選択してみましょう。

```
   MAXITER =: 64                NB. maximum number of
iterations
   image1 =: mfn1 " 0  GRID
```

結果 image1 は整数の行列であり、色にマッピングして画面上に表示することができます。

```
require '~addons/graphics/viewmat/viewmat.ijs'
viewmat image1
```

次のような画像が生成されます。



31.6.2 平面バージョン [Planar Version]

今度は、すべてのピクセルを一度に計算するバージョンを見ていきます。ここでは最初の試みです。これは mfn1 の簡単な開発です が、ここではすべてのピクセルのすべての計算が最大回数の反復で実行できるようになっていま MAXITER の値が小さい場合は OK です。 クイックデモンストレーション、最初の正確さ：それは mfn1 と同じ結果をもたらします。

```
   MAXITER =: 9

   (mfn1 " 0 GRID) -: (mfn3 " 2 GRID)
1
```

```
And it's faster:
   e3a =: 'mfn1 " 0 GRID  NB. Wikip. with MAXITER=9'
   e3b =: 'mfn3 " 2 GRID  NB. Planar with MAXITER=9'
   compare e3a;e3b
+---------------------------------------+-------
-+
|mfn1 " 0 GRID  NB. Wikip. with MAXITER=9|6.06934 |
+---------------------------------------+-------
-+
|mfn3 " 2 GRID  NB. Planar with MAXITER=9|0.168251|
+---------------------------------------+-------
-+
```

残念ながら、MAXITER の値が大きくなると問題が発生します。v の複素数の繰り返しの二乗は無限大ではなく最終的に無限大を生成しますが、無限大を減算することによって生じる「NaN 誤差」です。 観察する：

```
   (*: ^: 9) 1j3  NB. this is OK
1.96052e255j_9.80593e255

   (*: ^: 30) 1j3  NB. but this is not
|NaN error
|       (*:^:30)1j3
|[-589] c:\users\homer\X15\temp.ijs
```

残念ながら、MAXITER の値が大きくなると問題が発生します。v の複素数の繰り返しの二乗は 無限大ではなく最終的に無限大を生成しますが、無限大を減算することによって生じる「NaN 誤差」です。観察する:

```
  mfn4 =: 3 : 0
   N =. ($ y) $ 0
   v =. 0j0
   for_k.  i. MAXITER - 1  do.
       v =. y + *: v
       N =. N + 2 > | v
       if. 0 = 8 | k do.
           e =. N < k
           v =. e } v ,: 1.5j1.5
       end.
   end.
   N+1
)
```

NaN エラーを回避するための試みです。8 回ごとに 1 サイクル、「エスケープ」された(すなわち最終結果 N に寄与しない)v の値は、それらが制限なしに増加するのを防ぐために小さい値にリセットされる。

```
   mfn4 =: 3 : 0
    N =. ($ y) $ 0
    v =. 0j0
    for_k.  i. MAXITER - 1  do.
        v =. y + *: v
        N =. N + 2 > | v
        if. 0 = 8 | k do.
            e =. N < k
            v =. e } v ,: 1.5j1.5
        end.
    end.
    N+1
)
```

リセットの負担にもかかわらず、タイミングはスカラー法より約 10 倍速く見えます。

```
MAXITER =: 64
   e1 =: 'image1 =: mfn1 " 0 GRID NB. Scalar'
   e4 =: 'image4 =: mfn4 " 2 GRID NB. Planar, resetting'
   compare e1 ; e4
+-------------------------------------------+-------+
|image1 =: mfn1 " 0 GRID NB. Scalar         |17.6847|
+-------------------------------------------+-------+
|image4 =: mfn4 " 2 GRID NB. Planar, resetting|1.50849|
+-------------------------------------------+-------+
```

結果が正しいことを確認します。

```
   image4 -: image1
1
```

31.7 辞書 B のコード [The Special Code of Appendix B of the Dictionary]

内 J 辞書の付録 B 性能を改善するためにインタプリタによって特別な治療を与えられている約 80 種類の表現が記載されています。より多くの表現がリリースノートに記載されています

例は+ /です。スピードアップは、+ /(それと同等のものとは対照的に)自身が認識された場合にのみ発生することに注

意してください。

```
   data =: ? 1e6 $ 1e6  NB. a million random integers
   plus =: +

   compare 'plus / data' ; '+/ data'
+-----------+----------+
|plus / data|0.220362  |
+-----------+----------+
|+/ data    |0.00157533|
+-----------+----------+
```

特別な式は f でマスク解除できます。すべての定義された名前を組み込み関数に変換する副詞。

```
   foo =: plus /
   compare 'foo data'; 'foo f. data'
+-----------+----------+
|foo data   |0.221123  |
+-----------+----------+
|foo f. data|0.00160007|
+-----------+----------+
```

ここでの推奨は、プログラマがこれらの特殊なケースを使用する機会を探すべきではないということです。推薦は、適切な場合には、暗黙の定義に最終的な小さな磨きを与えることによって、通訳者がそれらを見つけることができるようにすることだけです。

☆第 31 章終了

第 32 章: ツリー [Trees]

32.1 はじめに [Introduction]

ボックス内のボックスからなるデータ構造は、ツリーと呼ばれることがあります。J は、木を使った計算をサポートするためのいくつかの特別な関数を提供します。
ツリーの例を次に示します。

```
   ] T =:  'the cat' ; 'sat' ; < 'on' ; < ('the';'mat')
+---------+---+---------------+
|the cat  |sat|+---+---------+|
|         |   ||on |+---+---+||
|         |   ||   ||the|mat|||
|         |   ||   |+---+---+||
|         |   |+---+---------+|
+---------+---+---------------+
```

内箱のない箱は葉と呼ばれます。T に は 7 つのボックスがあり、そのうちの 5 つは葉であることがわかります。

32.2 フェッチ [Fetching]

明らかに、任意のボックスの内容は 、インデックス付けとアンボクシングの組み合わせによってツリーT から取り出すことができます。

```
   ] a =: > 2 { T
+--+---------+
|on|+---+---+|
|  ||the|mat||
|  |+---+---+|
+--+---------+

   ] b =: > 1 { a
+---+---+
|the|mat|
+---+---+

   ] c =: > 1 { b
mat
```

この目的のために 、組み込みの動詞 "Fetch"(ダイアディック{: :)があります。その左の引数は、一連のインデックス(パスと呼ばれる)です。

```
   (2;1;1) {:: T
mat
```

さらなる例:

```
   2 {:: T
+--+---------+
|on|+---+---+|
|  ||the|mat||
|  |+---+---+|
+--+---------+

   (2 ;1) {:: T
+---+---+
|the|mat|
+---+---+
```

32.3 フェッチの領域 [The Domain of Fetch]

{:: の右の引数はスカラーではなく、より上位のベクトルでなければならず、そうでなければエラーが発生します。(1 つのボックスがスカラーであることを思い出してください)。

0 {:: , <'hello'	0 {:: < 'hello'
hello	error

フルレングスパスは、リーフからデータコンテンツをフェッチするパスであるとしましょう。
全長のパスに沿って、すべてのインデックスはスカラ、ボックスを選択する必要があります。そうでない場合はエラーが発生します。言い換えれば、単一のパスを持たなければなりません。

T	(2; 1 ; 0 1) {:: T
+-------+---+--------------+ |the cat|sat|+--+---------+| | | ||on|+---+---+|| | | || ||the|mat||| | | || |+---+---+|| | | |+--+---------+| +-------+---+--------------+	error

リーフからフェッチされたデータは、パスに沿って選択された最後のボックスを開いた結果です。このデータは、上で見

たように、配列である可能性があります。

```
   (2;1;1) {:: T
mat
```

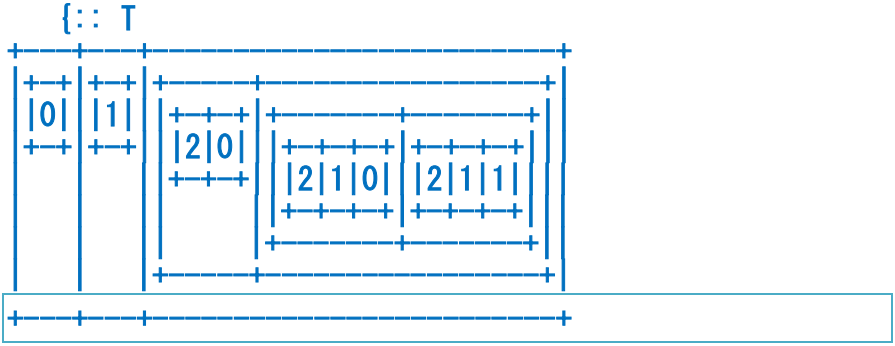
このデータが索引付け可能な配列である場合、追加の索引を全長のパスに追加して、オーバーレイのパスを与えて、さらに 1 つのスカラーをフェッチすることができます。次の例は、'mat'から' m'を取り出す方法を示しています。

```
   (2;1;1;0) {:: T
m
```

32.4「マップ」動詞 [he "Map" Verb]

Monadic {::は "Map"と呼ばれます。それは葉へのすべての経路を示しています。

```
   {:: T
```



32.5 このツリーの高さは幾つですか？ [What is the Height of This Tree?]

動詞 L.("LevelOf")は、ツリー内の最長パスの長さ、すなわち、葉の非ボックス化データコンテンツをフェッチするパスの最大長

T	L.T
+-------+---+--------------+ |the cat|sat|+--+----------+| | | ||on|+---+---+|| | | || ||the|mat||| | | || |+---+---+|| | | |+--+----------+| +-------+---+--------------+	3

を報告する。「プログラミング言語」という本では、Kenneth Iverson は木の最長経路の長さに「高さ」という用語を使用しています。
パスの長さは、必要なインデックス作成とアンボクシングのステップ数です。T から任意のデータコンテンツをフェッチするには、最大 3 つのステップが必要であることは明らかです

T	L.T
+-------+---+--------------+ |the cat|sat|+--+----------+| | | ||on|+---+---+|| | | || ||the|mat||| | | || |+---+---+|| | | |+--+----------+| +-------+---+--------------+	3

単一の葉のみからなる樹木の葉の内容、例えば<6 を取り出すには、1 つのステップが必要です。ステップは> @: (0&{)

A =: ,<6	L. A	(> @: (0&{)) A	0 {:: A
+-+ |6| +-+	1	6	6

「こんにちは」の 内容を取得する手順は必要ありません

L. 'hello'	(0$0) {:: 'hello'
0	hello

ここで、LevelOf a Tree はその Map から計算できます。つまり、L. T は、{T

```
   {:: T                    NB. Map giving the paths to leaves
+---+---+-----------------------+
|+-+|+-+|+-----+---------------+|
||0|||1|||+-+-+|+-------+-------+||
|+-+|+-+||2|0|||+-+-+-+|+-+-+-+|||
|   |   |+-+-+||2|1|0|||2|1|1|||||
```

```
   # S: 1 {:: T         NB. the length of each path
1 1 2 3 3

   >. / # S: 1 {:: T    NB. the maximum of the lengths
3

   L. T                 NB.  the LevelOf T
3
```

32.6 レベル [Levels]

本のテキストが整理される方法を考えてみましょう。

- 本は一連の章です。異なる本の章数は異なる場合があります。
- 章は一連の段落です。異なる章には段落の数が異なる場合があります。
- 段落は一連の文章です。異なる段落の文章数が異なる場合があります
- 文は一連の単語である。異なる文章は、異なる数の単語を有することができる。

章、段落、文章、言葉は、組織内の異なるレベルであると言います。組織の原則は明白です:

少数のレベルがあります。

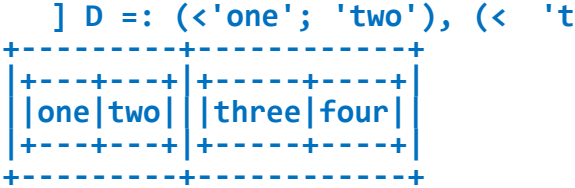
レベル数は予め固定されている。

すべての葉への経路は同じ長さです。

J は、このように固定レベルに編成されたツリーを処理するのに特に有用ないくつかの組み込み関数を提供しています。

このようなツリーの例を次に示します。

```
   ] D =: (<'one'; 'two'), (<  'three' ; 'four')
```



内箱がない箱(葉)は、レベル 0 にあると言われています。

与えられた関数を、L 内の値に適用することができます。つ

まり、レベル 0 で、L: 連結の助けを借りて("レベル At"と呼ばれます)。
それぞれのレベル 0 ノードの内容、つまり各リーフを逆転します:

```
   |. L: 0 D
+---------+------------+
|+---+---+|+-----+----+|
||eno|owt||||eerht|ruof||
|+---+---+|+-----+----+|
+---------+------------+
```

レベル 1 で反転:

```
   |. L: 1 D
+---------+------------+
|+---+---+|+----+-----+|
||two|one||||four|three||
|+---+---+|+----+-----+|
+---------+------------+
```

レベル 2 で:

```
   |. L: 2 D
+------------+---------+
|+-----+----+||+---+---+|
||three|four|||one|two||
|+-----+----+||+---+---+|
+------------+---------+
```

各レベル 0 1 2 で関数を適用できることがわかります。関数が適用されるレベルは、負の数として指定することもできます。

```
   |. L: _2 D
+---------+------------+
|+---+---+|+-----+----+|
||eno|owt||||eerht|ruof||
|+---+---+|+-----+----+|
+---------+------------+
```

```
   |. L: _1 D
+---------+------------+
|+---+---+|+----+-----+|
||two|one||||four|three||
|+---+---+|+----+-----+|
+---------+------------+
```

ツリーのレベルは、配列のランクに似ています。 L: ランク接続詞の類似体です」。

32.7 フラット接続詞[The Spread Conjunction]

我々は、L: 結合の結果が引数と同じ木構造を持っていることを上に見ました。レベルで関数を適用するには L: と同じように、ただし、L: とは異なり 、結果はツリーではなく単なるフラットリストとして渡されるという点で、S: (「スプレッド」と呼ばれる)があります。

```
   D
+---------+------------+
|+---+---+|+-----+----+|
||one|two||||three|four||
|+---+---+|+-----+----+|
+---------+------------+

   |. S: 0 D
eno
owt
eerht
ruof
```

上記の結果は、各項目が文字列である 4 つの項目のリスト(「フラットリスト」)です。

```
   |. S: 1 D
+----+-----+
|two |one  |
+----+-----+
|four|three|
+----+-----+
```

上記の結果は、2 つのアイテムのリストであり、各アイテムは 2 つのボックスのリストです。

```
   |. S: 2 D
+------------+---------+
|+-----+----+|+---+---+|
||three|four|||one|two||
|+-----+----+|+---+---+|
+------------+---------+
```

上記の結果は 2 つの項目のリストであり、各項目はボックスです。

32.8 さまざまなパス長のツリー [Trees with Varying Path-lengths]

上記 の例のツリーD では、すべての葉までの経路長は同じ長さです。しかしながら、一般に、経路長は変化し得る。ツリーT の例では、

```
   T
+--------+---+---------------+
|the cat|sat|+---+----------+|
|       |   ||on|+----+----+||
|       |   ||  ||the|mat|||
|       |   ||  |+----+----+||
|       |   |+---+----------+|
+--------+---+---------------+
```

経路は{:: T で示され、経路の長さは

```
   (# S: 1) {:: T
1 1 2 3 3
```

レベル 0 のノードの内容を逆にすると、驚きはありません。

```
   |. L: 0 T
+--------+---+---------------+
|tac eht|tas|+---+----------+|
|       |   ||no|+----+----+||
|       |   ||  ||eht|tam|||
|       |   ||  |+----+----+||
|       |   |+---+----------+|
+--------+---+---------------+
```

レベル 1 のノードの内容を逆にすると、レベル 0 の葉のすべてではなく一部がレベル 1 に再び現れることがわかります。

```
   |. L: 1 T
+--------+---+---------------+
|tac eht|tas|+---+----------+|
|       |   ||no|+----+----+||
|       |   ||  ||mat|the|||
|       |   ||  |+----+----+||
|       |   |+---+----------+|
+--------+---+---------------+
```

説明は、レベル 1 では与えられた動詞が
それらのノードは厳密にレベル 1 にあり、すなわち 1 = L のノードである。ノード AND
厳密にレベル 0 にあるノードは、レベル 1 ノード内に含まれ

ることによってまだ説明されていない。
同様に、レベル 2 のノードの内容を逆にすると、次のようになります。

```
   |. L: 2 T
+--------+---+--------------+
|tac eht|tas|+------------+-+|
|       |   ||+---+---+|on||
|       |   |||the|mat|||  ||
|       |   ||+---+---+|  ||
|       |   |+------------+-+|
+--------+---+--------------+
```

この例では、逆の結果の一部は文字列であり、一部はボックスのリストです。彼らはさまざまな種類のものです。これらの異なるタイプの結果は、S で試行されるようにフラットなリストに単純に組み立てることはできません。
したがって、u s: 1 は、動詞 u 自体がすべてのノードで一様な結果を出さない限り、失敗する可能性があります。次の 2 つの例を比較してください。

|. S: 1 T	(< @: |.) S: 1 T
error	+-------+---+--+---------+ |tac eht|tas|no|+---+---+| | | | ||mat|the|| | | | |+---+---+| +-------+---+--+---------+

Level 結合 L: 同じ方法でツリーを歩きます。つまり、同じノードをヒットして元に戻します。

```
   |. L: 0 T
+--------+---+--------------+
|tac eht|tas|+-+-----------+|
|       |   ||no|+---+---+||
|       |   |||  |eht|tam|||
|       |   ||  |+---+---+||
|       |   |+-+-----------+|
+--------+---+--------------+
```

しかし、レベルは結果のフラットなリストを構築しようとせず、個々の結果をツリー内のその位置に戻します。したがっ

て、フラットリストを構築しようとするために Spread が失敗する場合、Level は成功します。

|. S: 1 T	|. L: 1 T
error	+-------+---+--------------+ |tac eht|tas|+--+---------+| | | ||no|+---+---+|| | | || ||mat|the||| | | || |+---+---+|| | | |+--+---------+| +-------+---+--------------+

32.9 ツリーの変換 [Transforming Trees]

次に、ツリー、リレーションおよび配列の間の変換を調べます。これらの変換の動機は、構造をデータに変換することによって、いくつかの操作がより簡単になるということです。例えば、ブランチの追加や削除(グラフトやプルーニング)が容易になります。

このセクション全体を通して、問題の樹木は一定数のレベルの特性を持ち、すべての葉に同じ経路長を持つと仮定されます。

ここに出発点の木があります。

```
   p1 =: (< 'aa' ;'bb' ; 'cc'), (< 'dd' ; 'ee' )
   p2 =: (,< ( 'ff' ; 'gg'))
   ] T  =: (< p1) , (< p2)          NB. a tree
+---------------------+---------+
|+----------+-------+|+-------+|
||+--+--+--+|+--+--+||+--+--+||
|||aa|bb|cc|||dd|ee|||||ff|gg|||
||+--+--+--+|+--+--+||+--+--+||
|+----------+-------+|+-------+|
+---------------------+---------+
```

32.9.1 ツリーからの関係 [Relation from Tree]

ツリーとの関係を作る関数です。これは、葉へのすべてのパスのセットであるインデックス行列と、対応する葉のデータ値を組み合わせます。

```
   rfromt =: 3 : 0
    P =. indmat y        NB. index-matrix
```

```
    V =. dava   y       NB. the value of each word
    (<"1 P) (, "0 0) V   NB. relation
)

   indmat =: 3 : 0         NB. index-matrix of given tree
  (; @: >) S: 1 ({:: y)
)

   dava =: < S: 0           NB. data-values from leaves
```

各行は、リーフへのパスとそのリーフのデータ内容を関連付けることがわかります。

```
   T
+---------------------+---------+
|+----------+-------+|+-------+|
||+--+--+--+|+--+--+||+--+--+||
||aa|bb|cc|||dd|ee|||||ff|gg|||
||+--+--+--+|+--+--+||+--+--+||
|+----------+-------+|+-------+|
+---------------------+---------+
   ] R =: rfromt T
+-----+--+
|0 0 0|aa|
+-----+--+
|0 0 1|bb|
+-----+--+
|0 0 2|cc|
+-----+--+
|0 1 0|dd|
+-----+--+
|0 1 1|ee|
+-----+--+
|1 0 0|ff|
+-----+--+
|1 0 1|gg|
+-----+--+
```

32.9.2 関係からのツリー [Tree from Relation]

関係から木を作る関数:

```
   tfromr =: 3: 0の
    間に。(1 <: #> {。{。y)do。y =。ステップy終了。
    1( "1 y
)
どこで
   step =: 3: 0 NB。木から関係
    k =の
```

```
1ステップ。rkeys y      v =。k < (1 {"1 y)
    (～.k)、。v
)
   rkeys = :(}: &)>)@ :( {。" 1)NB。短縮キー
```

プロセスの各ステップは、パス長を１分の１に減らします。

```
R                step R
+-----+--+       +---+----------+
|0 0 0|aa|       |0 0|+--+--+--+|
+-----+--+       |   ||aa|bb|cc||
|0 0 1|bb|       |   |+--+--+--+|
+-----+--+       +---+----------+
|0 0 2|cc|       |0 1|+--+--+   |
+-----+--+       |   ||dd|ee|   |
|0 1 0|dd|       |   |+--+--+   |
+-----+--+       +---+----------+
|0 1 1|ee|       |1 0|+--+--+   |
+-----+--+       |   ||ff|gg|   |
|1 0 0|ff|       |   |+--+--+   |
+-----+--+       +---+----------+
|1 0 1|gg|
+-----+--+
```

```
step step R                 tfromr R
+-+--------------------+     +--------------------+---------+
|0|+----------+-------+|     |+----------+-------+|+-------+|
| ||+--+--+--+|+--+--+||     ||+--+--+--+|+--+--+|||+--+--+||
| |||aa|bb|cc|||dd|ee|||     |||aa|bb|cc|||dd|ee|||||ff|gg|||
| ||+--+--+--+|+--+--+||     ||+--+--+--+|+--+--+|||+--+--+||
| |+----------+-------+|     |+----------+-------+|+-------+|
+-+--------------------+     +--------------------+---------+
|1|+-------+           |
| ||+--+--+|           |
| |||ff|gg||           |
| ||+--+--+|           |
| |+-------+           |
+-+--------------------+
```

```
   T  - : tfromr R
1
```

32.9.3 配列からのツリーとツリーからの配列

ここでは、ツリーから配列を構築する動詞があります。結果はボックスの配列であると仮定されているため、パディングのために空のボックスが必要になることがあります。

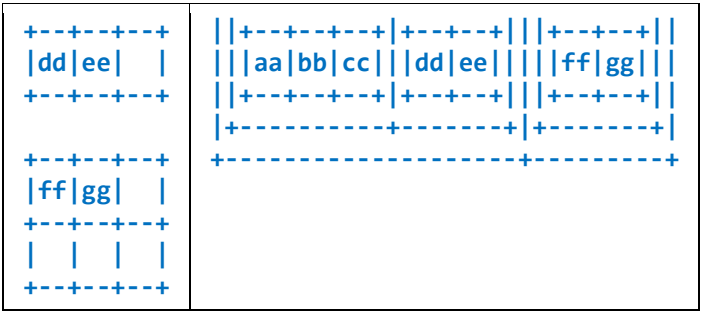
```
   afromt =: 3 : 0        NB. array from tree
    a =. indmat y
    b =. 1+ >./ a         NB. dimensions of array
    c =. b $ a:           NB. recipient array
    d =. dava T
    e =. < @: ( ;/) "1 a  NB. indices
    d e } c
)
```

```
T                                   A =: afromt T
+--------------------+---------+    +--+--+--+
|+----------+-------+|+-------+|    |aa|bb|cc|
||+--+--+--+|+--+--+|||+--+--+||    +--+--+--+
|||aa|bb|cc|||dd|ee|||||ff|gg|||    |dd|ee|  |
||+--+--+--+|+--+--+|||+--+--+||    +--+--+--+
|+----------+-------+|+-------+|
+--------------------+---------+    +--+--+--+
                                    |ff|gg|  |
                                    +--+--+--+
                                    |  |  |  |
                                    +--+--+--+
```

逆の関数は、すべての空のボックスがパディングされて破棄されると仮定して、配列からツリーを構築することです。最初に、指定された配列をその indexmatrix-value の関係に変換し、その関係を tfromr の木に変換します。

```
   tfroma =: 3: 0NB。ツリーから配列
    b =。、y NB。データ内容の単位
    c =。(: #: i。@(* /))$ y NB。インデックス行列
    d =。 - 。b  - :  "0 a:
    e =。(<1 c)、。b NB。関係
    f =。d: #e NB。空のボックスを
    破棄する t f
)
```

```
A          Z =: tfroma A
+--+--+--+ +--------------------+---------+
|aa|bb|cc| |+----------+-------+|+-------+|
```



正当性のチェック：

```
   Z  - : T
1
```

☆第 32 章終了

付録 1: 式の評価 [Evaluating Expressions]

A1.1 はじめに [Introduction]

ここでは、J の式を評価するプロセスを見ていきます。完全な式を評価するには、名前に割り当てられた値を取得するか、引数に関数を適用するなどの一連の基本ステップを実行します。例えば、与えられた、

```
   x =: 3
```

式

```
   4+5*x
19
```

次のステップで(アウトラインで)評価されます。

1. 3 に与える x に割り当てられた値を得る
2. 計算する 5 * 3 15 を与える
3. 4 + 15 を計算する 19

ステップが行われるシーケンスは、J 言語の文法(または「構文解析」)ルールによって管理されます。解析ルールにはさまざまな影響があります。たとえば、次のように非形式的に記述できます。

1. 動詞は長い正しい範囲を持っています。例えば、式 2 * 3 + 4 において、*の右の引数は 3 + 4 であり、2 * 3 + 4 は 2 *(3 + 4)を意味する。これは先に「右端優先」ルールと呼ばれていました。
2. 動詞は左の範囲が短い。たとえば、2 * 3 + 4 の場合、+の左の引数は 3 です。
3. 副詞と連言は動詞の前に適用されます。たとえば、+&1%2 は (+&1)%2 を意味します。
4. 副詞と結合詞は長い間スコープと短い権利スコープを残しています。

これらの効果は、式が暗黙的にカッコで括られる方法を記述します。もちろん、明示的なかっこを書くことで、たとえ必要ではないとしても、常に望ましい効果を生み出すことができます。さらなる効果は次のとおりです。

- ●名詞を表す名前は、直ちに評価されます。
- ●関数を示す名前は、関数が適用されるまで評価されません。
- ●値が割り当てられていない名前は動詞。
- ●長い動詞の動詞は長さ 2 または 3 の列に分解されます。

解析ルールがこれらの影響をどのように引き起こすかを見ていきます。このプロセスを説明するために、評価プロセスをモデル化またはシミュレートする機能をステップごとに使用してスローモーションで作業中に示す機能を使用できます。この関数(EVM という副詞)は、J ディクショナリのセクション IIE で与えられた解析アルゴリズムの記述に基づいています。これは、ダウンロード可能な J スクリプトで定義されてい ます。

A1.2 最初の例 [First Example]

2 + 3 などの式の評価は、モデリング副詞 EVM に引数'2 + 3'(文字列、通知)を提供することによってモデル化できます。

2 + 3	'2 + 3' EVM
5	5

我々は、参照「+ 3 2」EVM が同じ値を計算する 2 + 3 が、EVM は、評価プロセスのステップごとの記録、または履歴を生成します。この履歴は、式 hist ''を入力すると表示されます。

```
  hist ''
 Queue          Stack                  Rule
+----------+   +------+---+---+---+   +----+
|mark 2 + 3|   |      |   |   |   |   |    |
+----------+   +------+---+---+---+   +----+
|mark 2 +  |   | 3    |   |   |   |   |    |
+----------+   +------+---+---+---+   +----+
|mark 2    |   | +    | 3 |   |   |   |    |
+----------+   +------+---+---+---+   +----+
|mark      |   | 2    | + | 3 |   |   |    |
+----------+   +------+---+---+---+   +----+
|          |   | mark | 2 | + | 3 |   |dyad|
+----------+   +------+---+---+---+   +----+
|          |   | mark | 5 |   |   |   |    |
+----------+   +------+---+---+---+   +----+
```

我々はプロセスの連続的な段階を見る。この例では 6 つの段階があります。各ステージは 2 つの変数の値によって定義されます。最初に、「キュー」があります。最初に、評価される式が含まれ、単語に分割され、先頭にマークを付けるシン

ボルが前に付いています。第 2 に、最初は空の「スタック」があります。最初の段階では、最初にキューとスタックが表示されます。
各段階で、スタックが検査され、何かができるかどうか、つまり、スタック内の最初の数単語が規則が適用されるパターンを形成するかどうかが確認されます。これらのルールのうち 9 つがあり、それぞれが順番に試行されます。ルールが適用されない場合は、キューの末尾からスタックの先頭にワードが転送され、次のステージに移動して再試行します。このプロセスは、第 1 ステージから第 5 ステージにわたります。
第 5 段階では、ルールが適用可能であることがわかります。このルールは右端の列にダイアドとして識別されます。非公式には、ダイアド ルールは次のとおりです。
スタック内の最初の 4 つの項目が何かであれば、名詞、動詞、名詞に動詞を適用して新名詞を取得し、名詞。
6 番目と最後のステージは、前のステージで認識された「ダイアド」ルールを適用した結果を示しています。ルールは再試行され、結果は得られず、待ち行列には単語がなくなり、終了しました。最終的な結果は、スタックの 2 番目の項目です。履歴は Qh Sh と Rh の 3 つのグローバル変数で維持されます。式 hist ''は、これらの変数からフォーマットされた表示を計算します。

A1.3 解析ルール [Parsing Rules]

このセクションでは、9 つの解析ルールのそれぞれの例を示します。各ルールは、動詞の名詞動詞のような、スタックの前面にあるアイテムのパターンを探します。
スタックの各項目は、動詞、名詞、副詞、結合、名前、左括弧、右括弧、代入記号(=。または=:)または開始記号のいずれかに分類されます。
ルールの簡潔な記述を助けるために、より大きなクラスのアイテムを形成することができる。たとえば、項目が開始記号、割り当て記号または左括弧である場合、項目は「EDGE」として分類されます。
ルールは、常に同じ順序で試されます。モナドルールから始めて「カッコルール」で終わる、下に表示される順序です。

A1.3.1 モナド ルール [Monad Rule]

スタックの最初の 3 つの項目が "EDGE"で、続いて動詞の後に名詞がある場合、動詞は名詞に適用され、結果値は Z say によって記号化され、値 Z は動詞と名詞がスタックにあります。スタックの項目を変換するスキームは次のとおりです。

```
monad rule: EDGE VERB NOUN etc  =>   EDGE Z etc
```

ここで、Z は NOUN に VERB を 適用して計算した結果です。例えば:

*: 4	'*: 4' EVM
16	16

```
  hist ''
Queue          Stack                Rule
+----------+   +------+----+---+   +-----+
|mark *: 4|    |      |    |   |   |     |
+----------+   +------+----+---+   +-----+
|mark *:  |    | 4    |    |   |   |     |
+----------+   +------+----+---+   +-----+
|mark     |    | *:   | 4  |   |   |     |
+----------+   +------+----+---+   +-----+
|         |    | mark | *: | 4 |   |monad|
+----------+   +------+----+---+ +-----+
|         |    | mark | 16 |   |   |     |
+----------+   +------+----+---+   +-----+
```

A1.3.2 第 2 モナ ドルール [Second Monad Rule]

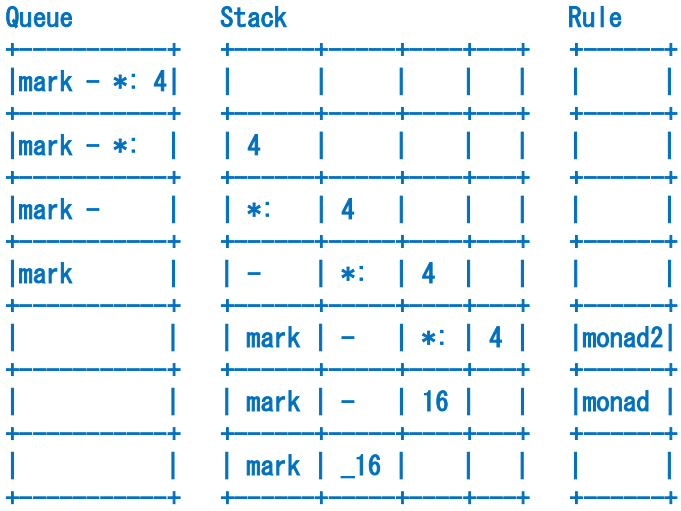
スタック内のアイテムは、EDGE、副詞または動詞または名詞である場合、「EAVN」として分類されます。スキームは次のとおりです。

```
monad2 rule: EAVN VERB1 VERB2 NOUN etc => EAVN VERB1 Z  etc
```

ここで、Z はである VERB2 monadically NOUN に適用しました。例えば:

- *: 4	' - *: 4' EVM
_16	_16

```
  hist ''
```



A1.3.3 ダイアド ルール [Dyad Rule]

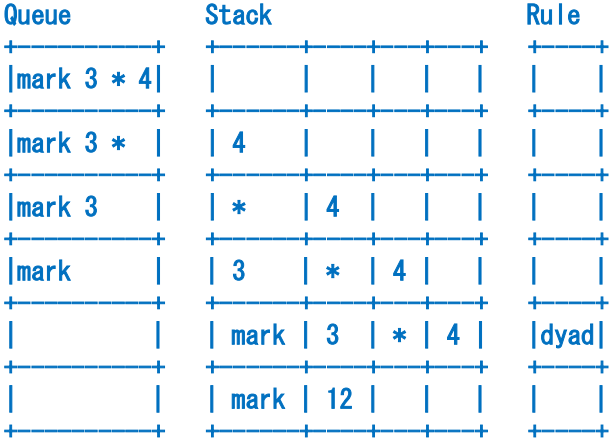
スキームは

```
dyad rule:  EAVN NOUN1 VERB NOUN2 etc => EAVN Z etc
```

ここで、Z は、NOUN1 と NOUN2 に対話的に適用される VERB です。例えば。

3 * 4	'3 * 4' EVM
12	12

```
  hist ''
```



A1.3.4 副詞 規則 [Adverb Rule]

動詞または名詞である項目は「VN」に分類されます。スキームは次のとおりです。

```
adverb rule: EAVN VN ADVERB etc => EAVN Z etc
```

ここで、Z は VN に ADVERB を適用した結果です。例えば:

+ / 1 2 3	'+ / 1 2 3' EVM
6	6

```
  hist ''
 Queue              Stack                                Rule
+--------------+   +-------+-------+-------+-------+   +-----+
|mark + / 1 2 3|   |       |       |       |       |   |     |
+--------------+   +-------+-------+-------+-------+   +-----+
|mark + /      |   | 1 2 3 |       |       |       |   |     |
+--------------+   +-------+-------+-------+-------+   +-----+
|mark +        |   | /     | 1 2 3 |       |       |   |     |
+--------------+   +-------+-------+-------+-------+   +-----+
|mark          |   | +     | /     | 1 2 3 |       |   |     |
+--------------+   +-------+-------+-------+-------+   +-----+
|              |   | mark  | +     | /     | 1 2 3 |   |adv  |
+--------------+   +-------+-------+-------+-------+   +-----+
|              |   | mark  | +/    | 1 2 3 |       |   |monad|
+--------------+   +-------+-------+-------+-------+   +-----+
|              |   | mark  | 6     |       |       |   |     |
+--------------+   +-------+-------+-------+-------+   +-----+
```

A1.3.5 結合規則 [Conjunction Rule]

スキームは次のとおりです。

```
        conjunction  EAVN VN1 CONJ VN1 etc => EAVN
Z etc
```

ここで Z は CONJ を VN1 と VN2 に適用した結果です。例えば:

1 & + 2	'1 & + 2' EVM
3	3

```
  hist ''
 Queue              Stack                          Rule
+------------+   +------+-----+---+---+---+   +-----+
|mark 1 & + 2|   |      |     |   |   |   |   |     |
```

```
-------------+   +------+-----+---+---+---+   +-----+
|mark 1 & +  |   | 2    |     |   |   |   |   |     |
+-------------+   +------+-----+---+---+---+   +-----+
|mark 1 &    |   | +    | 2   |   |   |   |   |     |
+-------------+   +------+-----+---+---+---+   +-----+
|mark 1      |   | &    | +   | 2 |   |   |   |     |
+-------------+   +------+-----+---+---+---+   +-----+
|mark        |   | 1    | &   | + | 2 |   |   |     |
+-------------+   +------+-----+---+---+---+   +-----+
|            |   | mark | 1   | & | + | 2 |   |conj |
+-------------+   +------+-----+---+---+---+   +-----+
|            |   | mark | 1&+ | 2 |   |   |   |monad|
+-------------+   +------+-----+---+---+---+   +-----+
|            |   | mark | 3   |   |   |   |   |     |
+-------------+   +------+-----+---+---+---+   +-----+
```

A1.3.6 トライデントルール [Trident Rule]

スキームは次のとおりです。

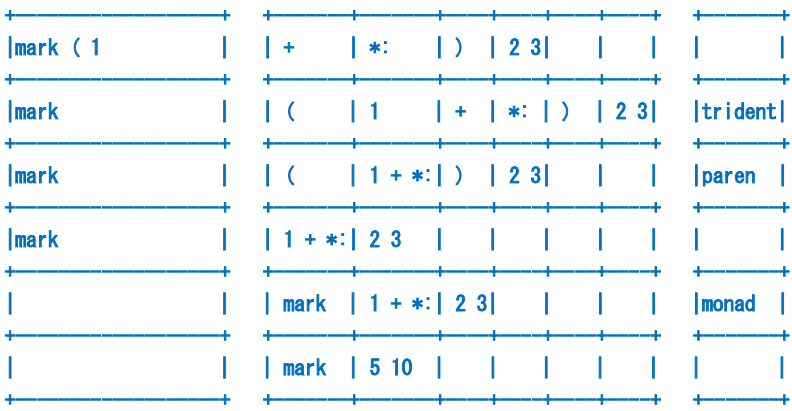
```
        trident rule: EAVN VN1 VERB2 VERB3 etc =>
EAVN Z etc
```

2 つの場合があります。VN1 は動詞または名詞です。
VN1 が動詞 VERB1 で あれば、 Z はフォーク VERB1 VERB2 VERB3 として定義された単一動詞 です。
場合 VN1 が名詞である NOUN1 次いで、 Z は、フォークの略語として定義された単一の動詞である NOUN1 VERB2 VERB3。
フォークのフォークと略語については、第 9 章を参照してください。
次に例を示します。1 + *: fork 1 の略語 : + *:

(1: + * :) 2 3	(1 + * :) 2 3	'(1 + * :) 2 3' EVM
5 10	5 10	5 10

```
   hist ''
Queue                 Stack                                Rule
+-------------------+  +------+------+---+---+---+---+   +------+
|mark ( 1 + *: ) 2 3|  |      |      |   |   |   |   |   |      |
+-------------------+  +------+------+---+---+---+---+   +------+
|mark ( 1 + *: )    |  | 2 3  |      |   |   |   |   |   |      |
+-------------------+  +------+------+---+---+---+---+   +------+
|mark ( 1 + *:      |  | )    | 2 3  |   |   |   |   |   |      |
+-------------------+  +------+------+---+---+---+---+   +------+
|mark ( 1 +         |  | *:   | )    | 2 3|   |   |   |   |      |
```

```
+-----------------+  +-----+-----+---+---+---+---+  +-----+
|mark ( 1         |  | +   | *:  | ) | 2 3|   |   |  |     |
+-----------------+  +-----+-----+---+---+---+---+  +-----+
|mark             |  | (   | 1   | + | *: | ) | 2 3|  |trident|
+-----------------+  +-----+-----+---+---+---+---+  +-----+
|mark             |  | (   | 1 + *:| ) | 2 3|   |   |  |paren  |
+-----------------+  +-----+-----+---+---+---+---+  +-----+
|mark             |  | 1 + *:| 2 3 |   |   |   |   |  |     |
+-----------------+  +-----+-----+---+---+---+---+  +-----+
|                 |  | mark | 1 + *:| 2 3|   |   |   |  |monad  |
+-----------------+  +-----+-----+---+---+---+---+  +-----+
|                 |  | mark | 5 10 |   |   |   |   |  |     |
+-----------------+  +-----+-----+---+---+---+---+  +-----+
```



A1.3.7 Bident Rule [Bident Rule]

スキームは次のとおりです。

```
bident rule: EDGE CAVN1 CAVN2 etc => EDGE Z etc
```

そして、これらの 6 つの事例がすべて事実上の規則のために存在する。

CAVN1	CAVN2	Z
動詞	動詞	動詞(フック)
副詞	副詞	副詞
連合	動詞	副詞
連合	名詞	副詞
名詞	連合	副詞
動詞	連合	副詞

最初のケース(フック)は第 03 章で、残りのケースは第 15 章のビンデムのスキームで説明されてい ます。
次の例では、式(1&)は noun という形式の象徴です。したがって、それは副詞です。

+(1&)2	'+(1&)2' EVM
3	3

```
   hist ''
Queue                Stack                        Rule
+----------------+  +-----+-----+---+---+---+  +-----+
|mark + ( 1 & ) 2|  |     |     |   |   |   |  |     |
```

```
+----------------+   +------+-----+----+---+---+   +------+
|mark + ( 1 & )  |   | 2    |     |    |   |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
|mark + ( 1 &    |   | )    | 2   |    |   |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
|mark + ( 1      |   | &    | )   | 2  |   |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
|mark + (        |   | 1    | &   | )  | 2 |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
|mark +          |   | (    | 1   | &  | ) | 2 |   |bident|
+----------------+   +------+-----+----+---+---+   +------+
|mark +          |   | (    | 1&  | )  | 2 |   |   |paren |
+----------------+   +------+-----+----+---+---+   +------+
|mark +          |   | 1&   | 2   |    |   |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
|mark            |   | +    | 1&  | 2  |   |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
|                |   | mark | +   | 1& | 2 |   |   |adv   |
+----------------+   +------+-----+----+---+---+   +------+
|                |   | mark | 1&+ | 2  |   |   |   |monad |
+----------------+   +------+-----+----+---+---+   +------+
|                |   | mark | 3   |    |   |   |   |      |
+----------------+   +------+-----+----+---+---+   +------+
```

A1.3.8 割当ルール [Assignment Rule]

私たちは、名詞または名前を表すために NN を書いています。そして、ASGN 代入記号について=または=。。スキームは次のとおりです。

```
assign rule: NN Asgn CAVN etc => Z etc
```

ここで、Z は CAVN の値です。

1 + x =: 6	'1 + x =: 6' EVM
7	7

```
   hist ''
Queue                Stack                  Rule
+----------------+   +------+----+---+---+   +------+
|mark 1 + x =: 6|   |      |    |   |   |   |      |
+----------------+   +------+----+---+---+   +------+
|mark 1 + x =:   |   | 6    |    |   |   |   |      |
+----------------+   +------+----+---+---+   +------+
|mark 1 + x      |   | =:   | 6  |   |   |   |      |
+----------------+   +------+----+---+---+   +------+
|mark 1 +        |   | x    | =: | 6 |   |   |assign|
```

```
+---------------+   +------+----+---+---+   +------+
|mark 1 +       |   | 6    |    |   |   |   |      |
+---------------+   +------+----+---+---+   +------+
|mark 1         |   | +    | 6  |   |   |   |      |
+---------------+   +------+----+---+---+   +------+
|mark           |   | 1    | +  | 6 |   |   |      |
+---------------+   +------+----+---+---+   +------+
|               |   | mark | 1  | + | 6 |   |dyad  |
+---------------+   +------+----+---+---+   +------+
|               |   | mark | 7  |   |   |   |      |
+---------------+   +------+----+---+---+   +------+
```

A1.3.9 括弧のルール [Parenthesis Rule]

スキームは次のとおりです。

```
paren rule: ( CAVN ) etc => Z etc
```

ここで、Z は CAVN の値です。例えば:

(1+2)*3	'(1+2)*3' EVM
9	9

```
   hist ''
Queue                 Stack                          Rule
+-----------------+   +------+---+---+---+---+---+---+   +-----+
|mark ( 1 + 2 ) * 3|   |      |   |   |   |   |   |   |   |     |
+-----------------+   +------+---+---+---+---+---+---+   +-----+
|mark ( 1 + 2 ) *  |   | 3    |   |   |   |   |   |   |   |     |
+-----------------+   +------+---+---+---+---+---+---+   +-----+
|mark ( 1 + 2 )    |   | *    | 3 |   |   |   |   |   |   |     |
+-----------------+   +------+---+---+---+---+---+---+   +-----+
|mark ( 1 + 2      |   | )    | * | 3 |   |   |   |   |   |     |
+-----------------+   +------+---+---+---+---+---+---+   +-----+
|mark ( 1 +        |   | 2    | ) | * | 3 |   |   |   |   |     |
+-----------------+   +------+---+---+---+---+---+---+   +-----+
|mark ( 1          |   | +    | 2 | ) | * | 3 |   |   |   |     |
+-----------------+   +------+---+---+---+---+---+---+   +-----+
|mark (            |   | 1    | + | 2 | ) | * | 3 |   |   |     |
+-----------------+   +------+---+---+---+---+---+---+   +-----+
|mark              |   | (    | 1 | + | 2 | ) | * | 3 |   |dyad |
+-----------------+   +------+---+---+---+---+---+---+   +-----+
|mark              |   | (    | 3 | ) | * | 3 |   |   |   |paren|
+-----------------+   +------+---+---+---+---+---+---+   +-----+
|mark              |   | 3    | * | 3 |   |   |   |   |   |     |
+-----------------+   +------+---+---+---+---+---+---+   +-----+
|                  |   | mark | 3 | * | 3 |   |   |   |   |dyad |
```

```
+-----------------+   +-----+---+---+---+---+---+---+   +-----+
|                 |   | mark | 9 |   |   |   |   |   |   |     |
+-----------------+   +-----+---+---+---+---+---+---+   +-----+
```

A1.3.10 転送の例 [Examples of Transfer]

次の例は、名前がキューからスタックに転送されるときに、名前が名詞である値を示す場合、名前ではなく値がキューに移動することを示しています。

a =: 6	(a = 7)、a
6	7 6
a =: 6	'(a =: 7)、' EVM
6	7 6

```
   hist ''
Queue                  Stack                                Rule
+-------------------+   +-----+-----+---+---+---+---+   +-----+
|mark ( a =: 7 ) , a|   |     |     |   |   |   |   |   |     |
+-------------------+   +-----+-----+---+---+---+---+   +-----+
|mark ( a =: 7 ) ,  |   | 6   |     |   |   |   |   |   |     |
+-------------------+   +-----+-----+---+---+---+---+   +-----+
|mark ( a =: 7 )    |   | ,   | 6   |   |   |   |   |   |     |
+-------------------+   +-----+-----+---+---+---+---+   +-----+
|mark ( a =: 7      |   | )   | ,   | 6 |   |   |   |   |     |
+-------------------+   +-----+-----+---+---+---+---+   +-----+
|mark ( a =:        |   | 7   | )   | , | 6 |   |   |   |     |
+-------------------+   +-----+-----+---+---+---+---+   +-----+
|mark ( a           |   | =:  | 7   | ) | , | 6 |   |   |     |
+-------------------+   +-----+-----+---+---+---+---+   +-----+
|mark (             |   | a   | =:  | 7 | ) | , | 6 |   |assign|
+-------------------+   +-----+-----+---+---+---+---+   +-----+
|mark (             |   | 7   | )   | , | 6 |   |   |   |     |
+-------------------+   +-----+-----+---+---+---+---+   +-----+
|mark               |   | (   | 7   | ) | , | 6 |   |   |paren |
+-------------------+   +-----+-----+---+---+---+---+   +-----+
|mark               |   | 7   | ,   | 6 |   |   |   |   |     |
+-------------------+   +-----+-----+---+---+---+---+   +-----+
|                   |   | mark | 7   | , | 6 |   |   |   |dyad  |
+-------------------+   +-----+-----+---+---+---+---+   +-----+
|                   |   | mark | 7 6 |   |   |   |   |   |     |
+-------------------+   +-----+-----+---+---+---+---+   +-----+
```

対照的に、名前が動詞の名前である場合、名前は評価されず

にスタックに転送されます。したがって、その後の割り当てによって、適用される動詞が変更されます。

f =: +	((f = :-)、f)4
+	_4 _4

f =: +	'((f =: -)、f)4' EVM
+	_4 _4

```
   hist ''
Queue                        Stack                                  Rule
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|mark ( ( f =: - ) , f ) 4|  |-------+-------+-+-+-+-+-+-+-+  +-------+
+-------------------------+  |-------+-------+-+-+-+-+-+-+-+  +-------+
|mark ( ( f =: - ) , f )  |  | 4     |       | | | | | | |  |       |
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|mark ( ( f =: - ) , f    |  | )     | 4     | | | | | | |  |       |
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|mark ( ( f =: - ) ,      |  | f     | )     | 4| | | | | |  |       |
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|mark ( ( f =: - )        |  | ,     | f     | )| 4| | | | |  |       |
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|mark ( ( f =: -          |  | )     | ,     | f| )| 4| | | |  |       |
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|mark ( ( f =:            |  | -     | )     | ,| f| )| 4| | |  |       |
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|mark ( ( f               |  | =:    | -     | )| ,| f| )| 4| |  |       |
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|mark ( (                 |  | f     | =:    | -| )| ,| f| )| 4|  |assign |
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|mark ( (                 |  | -     | )     | ,| f| )| 4| | |  |       |
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|mark (                   |  | (     | -     | )| ,| f| )| 4| |  |paren  |
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|mark (                   |  | -     | ,     | f| )| 4| | | |  |       |
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|mark                     |  | (     | -     | ,| f| )| 4| | |  |trident|
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|mark                     |  | (     | - , f| )| 4| | | | |  |paren  |
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|mark                     |  | - , f| 4     | | | | | | |  |       |
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|                         |  | mark | - , f| 4| | | | | |  |monad  |
+-------------------------+  +-------+-------+-+-+-+-+-+-+-+  +-------+
|                         |  | mark | _4 _4| | | | | | |  |       |
```

A1.3.11 解析ルールの見直し [Examples of Transfer]

ルール		前にスタックする					スタックした後				ここで Z は...
モナド		EDGE	動詞	名詞	等		EDGE	Z	等		名詞に動詞動詞
モナド 2		EAVN	動詞 1	動詞 2	名詞		EAVN	動詞 1	Z		名詞に適用された動詞 2
二人組		EAVN	名詞 1	動詞	名詞 2		EAVN	Z	等		動詞が Noun1 と Noun2 に適用されました
副詞		EAVN	VN	Adv	等		EAVN	Z	等		VN に適用された Adv
conj		EAVN	VN1	コンジ	VN2		EAVN	Z	等		VN1 と VN2 に適用される Conj
トライデント		EAVN	VN1	動詞 2	動詞 3		EAVN	Z	等		フォーク (VN1 Verb2 Verb3)
二人		EDGE	CAVN1	CAVN2	等		EDGE	Z	等		ビデント (CAVN1 CAVN2)
割り当てます		NN	Asgn	CAVN	等		Z	等	等		CAVN
パレン		(	CAVN	)	等		Z	等	等		CAVN

A1.4 解析ルールの効果 [Examples of Transfer

ここでは、パースルールのいくつかの効果を見ていきます。以下では、実際に解析ルールがどのように暗黙のかっこを発生させるかに注目してください。

A1.4.1 ダイアルは長い右の範囲を持っている [Dyad Has Long Right Scope]

式 4 + 3-2 を考えてみましょう。これは 4+(3-2)を意味します。

4 + 3 - 2	4 + (3-2)	'4+3-2' EVM
5	5	5

```
  hist ''
Queue                Stack                    Rule
+--------------+     +------+---+---+---+     +----+
|mark 4 + 3 - 2|     |      |   |   |   |     |    |
+--------------+     +------+---+---+---+     +----+
|mark 4 + 3 -  |     | 2    |   |   |   |     |    |
+--------------+     +------+---+---+---+     +----+
```

```
|mark 4 + 3    |   | -     | 2 |   |   |   |    |
+--------------+   +-------+---+---+---+   +----+
|mark 4 +      |   | 3     | - | 2 |   |   |    |
+--------------+   +-------+---+---+---+   +----+
|mark 4        |   | +     | 3 | - | 2 |   |dyad|
+--------------+   +-------+---+---+---+   +----+
|mark 4        |   | +     | 1 |   |   |   |    |
+--------------+   +-------+---+---+---+   +----+
|mark          |   | 4     | + | 1 |   |   |    |
+--------------+   +-------+---+---+---+   +----+
|              |   | mark  | 4 | + | 1 |   |dyad|
+--------------+   +-------+---+---+---+   +----+
|              |   | mark  | 5 |   |   |   |    |
+--------------+   +-------+---+---+---+   +----+
```

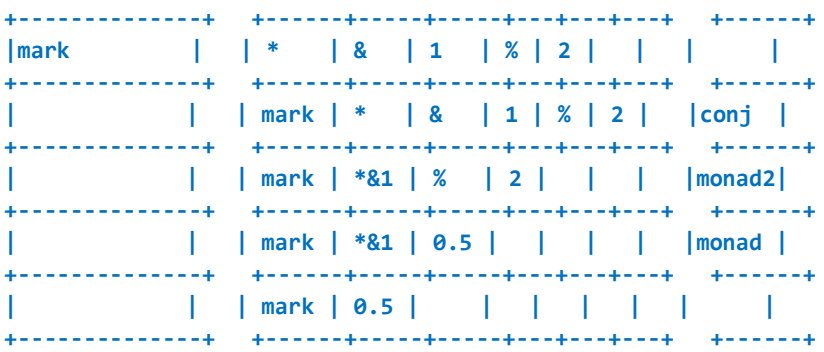
ここでは、一般的なルールの例を持っている：二項動詞は、この例ではので、可能な限りその右引数として取り+に適用される 3-2 だけではなく、3。
さらに、二項動詞は、できるだけ左引数として取る。この例での左引数は、-だけである 3 ではなく 4 + 3。したがって、二項動詞は「長い正しい範囲」と「短い左の範囲」を持つと言われています。

A1.4.2 動詞の前の演算子 [Operators Before Verbs]

副詞と結合詞が最初に適用され、次に得られる動詞が適用されます。

* & 1 % 2	(*&1) % 2	'* & 1 % 2' EVM
0.5	0.5	0.5

```
  hist ''
Queue             Stack                          Rule
+--------------+   +------+-----+-----+---+---+---+   +------+
|mark * & 1 % 2|   |      |     |     |   |   |   |   |      |
+--------------+   +------+-----+-----+---+---+---+   +------+
|mark * & 1 %  |   | 2    |     |     |   |   |   |   |      |
+--------------+   +------+-----+-----+---+---+---+   +------+
|mark * & 1    |   | %    | 2   |     |   |   |   |   |      |
+--------------+   +------+-----+-----+---+---+---+   +------+
|mark * &      |   | 1    | %   | 2   |   |   |   |   |      |
+--------------+   +------+-----+-----+---+---+---+   +------+
|mark *        |   | &    | 1   | %   | 2 |   |   |   |      |
```

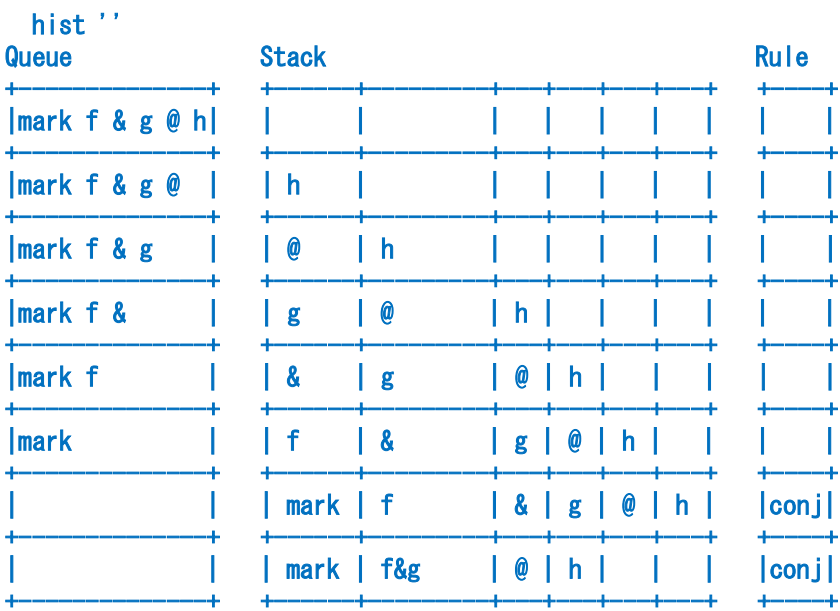


A1.4.3 演算子は長い左の範囲を保持 [Operators Have Long Left Scope]

次の例では、動詞の値がその構造を示すために"カッコで表した表現"(第 27 章を参照)に示されていることに注意してください。

副詞または結合詞は、可能な限り左の議論をとる。これらの動詞の構造を見てください: 明らかに/副詞と@結合詞はすべて左に取っています:

f @ g /	f & g @ h	'f&g@h' EVM
(f@g)/	(f&g)@h	(f&g)@h



```
|              |   | mark | (f&g)@h |   |   |   |   |   |   |
+--------------+   +------+---------+---+---+---+---+   +---+
```

したがって、オペレータは「長い左スコープ」と言われています。例では 、F&G @ h を我々は右引数ことを確認&はちょうどであるグラム、ないグラム@ h を。したがって、接続詞には「短い権利範囲」があります。

A1.4.4 左の trains [Train on the Left]

副詞の長い左のスコープは trains を通って延びていません。希望の効果を得るには括弧が必要な場合があります。fgh が train として意図されていると仮定し、次に以下を比較する:

(f g h) /	f g h /	'f g h / ' EVM
(f g h)/	f g (h/)	f g (h/)

```
  hist ''

Queue            Stack                          Rule

+------------+   +------+----------+----+----+   +-------+
|mark f g h /|   |      |          |    |    |   |       |
+------------+   +------+----------+----+----+   +-------+
|mark f g h  |   | /    |          |    |    |   |       |
+------------+   +------+----------+----+----+   +-------+
|mark f g    |   | h    | /        |    |    |   |       |
+------------+   +------+----------+----+----+   +-------+
|mark f      |   | g    | h        | /  |    |   |adv    |
+------------+   +------+----------+----+----+   +-------+
|mark f      |   | g    | h/       |    |    |   |       |
+------------+   +------+----------+----+----+   +-------+
|mark        |   | f    | g        | h/ |    |   |       |
+------------+   +------+----------+----+----+   +-------+
|            |   | mark | f        | g  | h/ |   |trident|
+------------+   +------+----------+----+----+   +-------+
|            |   | mark | f g (h/) |    |    |   |       |
+------------+   +------+----------+----+----+   +-------+
```

連結のために同様に（右の引数によって）

f g h @ +	'f g h @ +' EVM

f g (h@+)	f g (h@+)

```
  hist ''

Queue                Stack                                Rule

+--------------+     +------+-----------+-----+-----+     +-------+
|mark f g h @ +|     |      |           |     |     |     |       |
+--------------+     +------+-----------+-----+-----+     +-------+
|mark f g h @  |     | +    |           |     |     |     |       |
+--------------+     +------+-----------+-----+-----+     +-------+
|mark f g h    |     | @    | +         |     |     |     |       |
+--------------+     +------+-----------+-----+-----+     +-------+
|mark f g      |     | h    | @         | +   |     |     |       |
+--------------+     +------+-----------+-----+-----+     +-------+
|mark f        |     | g    | h         | @   | +   |     |conj   |
+--------------+     +------+-----------+-----+-----+     +-------+
|mark f        |     | g    | h@+       |     |     |     |       |
+--------------+     +------+-----------+-----+-----+     +-------+
|mark          |     | f    | g         | h@+ |     |     |       |
+--------------+     +------+-----------+-----+-----+     +-------+
|              |     | mark | f         | g   | h@+ |     |trident|
+--------------+     +------+-----------+-----+-----+     +-------+
|              |     | mark | f g (h@+) |     |     |     |       |
+--------------+     +------+-----------+-----+-----+     +-------+
```

しかし、どの右の議論なしの連結のためにも、左の範囲は trains を通って拡張しない：

f g h @	'f g h @' EVM
(f g h)@	(f g h)@

```
  hist ''

Queue                Stack                                Rule

+------------+     +------+-----------+---+---+---+     +-------+
|mark f g h @|     |      |           |   |   |   |     |       |
+------------+     +------+-----------+---+---+---+     +-------+
|mark f g h  |     | @    |           |   |   |   |     |       |
+------------+     +------+-----------+---+---+---+     +-------+
|mark f g    |     | h    | @         |   |   |   |     |       |
+------------+     +------+-----------+---+---+---+     +-------+
|mark f      |     | g    | h         | @ |   |   |     |       |
```

```
+------------+   +------+----------+---+---+---+   +-------+
|mark        |   | f    | g        | h | @ |   |   |       |
+------------+   +------+----------+---+---+---+   +-------+
|            |   | mark | f        | g | h | @ |   |trident|
+------------+   +------+----------+---+---+---+   +-------+
|            |   | mark | f g h    | @ |   |   |   |bident |
+------------+   +------+----------+---+---+---+   +-------+
|            |   | mark | (f g h)@ |   |   |   |   |       |
+------------+   +------+----------+---+---+---+   +-------+
```

対照的に、f / g /の場合、「副詞」規則を適用する前に「conj」規則がどのように適用されるかに注目する

f @ g /	'f @ g / ' EVM
(f@g)/	(f@g)/

```
  hist ''

Queue            Stack                          Rule

+------------+   +------+---------+---+---+---+   +----+
|mark f @ g /|   |      |         |   |   |   |   |    |
+------------+   +------+---------+---+---+---+   +----+
|mark f @ g  |   | /    |         |   |   |   |   |    |
+------------+   +------+---------+---+---+---+   +----+
|mark f @    |   | g    | /       |   |   |   |   |    |
+------------+   +------+---------+---+---+---+   +----+
|mark f      |   | @    | g       | / |   |   |   |    |
+------------+   +------+---------+---+---+---+   +----+
|mark        |   | f    | @       | g | / |   |   |    |
+------------+   +------+---------+---+---+---+   +----+
|            |   | mark | f       | @ | g | / |   |conj|
+------------+   +------+---------+---+---+---+   +----+
|            |   | mark | f@g     | / |   |   |   |adv |
+------------+   +------+---------+---+---+---+   +----+
|            |   | mark | (f@g)/  |   |   |   |   |    |
+------------+   +------+---------+---+---+---+   +----+
```

A1.4.5 動詞の推定 [Presumption of Verb]

値が割り当てられていない名前は動詞と見なされます。たとえば、次の 2 つの名前はフックを作成します。

Blue Skies	'Blue Skies' EVM
Blue Skies	Blue Skies

```
   hist ''
Queue                Stack                              Rule
 +---------------+   +-------+-----------+-------+   +------+
 |mark Blue Skies|   |       |           |       |   |      |
 +---------------+   +-------+-----------+-------+   +------+
 |mark Blue      |   | Skies |           |       |   |      |
 +---------------+   +-------+-----------+-------+   +------+
 |mark           |   | Blue  | Skies     |       |   |      |
 +---------------+   +-------+-----------+-------+   +------+
 |               |   | mark  | Blue      | Skies |   |bident|
 +---------------+   +-------+-----------+-------+   +------+
 |               |   | mark  | Blue Skies |       |   |      |
 +---------------+   +-------+-----------+-------+   +------+
```

☆付録 1 終了。

付録 2: 収集された用語

本書では、「データ」、「関数」、「引数」、「式」という単語は、プログラミングで通常使用される意味で使用されます。
本書では、以下に示す意味で他の言葉が使用されていますが、各単語の説明は前に説明した言葉にのみ依存しています。

値 VALUE	式を評価することによって生成できるものはすべて値であると言われています。すべての値はデータ値または関数です。
名詞 NOUN	データ値
動詞 VERB	名詞から名詞を計算する関数。
モナド MONAD	単一の引数を取る動詞。 Monadic
ダイアド DYAD	2 つの引数を取る動詞。すべての動詞はモナドまたはダイアドです。 Dyadic
アンビバレント AMBIVALENT	式は、(1 つまたは 2 つの引数が指定されているかどうかに応じて)モナドまたはダイアドのいずれかを示すときには相反すると言います。
オペレータ OPERATO	それらの引数(単数または複数)として名詞または動詞をとり、結果として名詞または動詞または演算子を生成する関数。すべての J 関数は動詞または演算子です。
副詞 ADVERB	単一の引数を取る演算子。
結合 CONJUNCTION	2 つの引数を取る演算子。すべての演算子は副詞または結合詞です。

入会者 BIDENT	J 文法が単一の関数として解釈を提供する 2 つの式のシーケンス。
トライデント TRIDENT	J 文法が解釈を単一の関数として提供する 3 つの式のシーケンス。
結合 TRAIN	J 文法が解釈を単一の関数として提供する 2 つ以上の式のシーケンス。
フック HOOK	2 つの動詞のシーケンス、すなわち、bident として定義される動詞。
フォーク FORK	3 つの動詞、すなわち三角形のシーケンスとして定義される動詞。
明らかに EXPLICIT	関数は、値が代入される引数変数を含む式で定義されている場合、明示的に定義されている、または明示的であると言われます。
暗黙 TACIT	関数は、引数変数を使用せずに定義された場合、暗黙的に定義されるか、または単に暗黙のうちに定義されると言われます。すべての J 関数は、組み込み関数または明示関数または暗黙関数のいずれかです。
アレイ ARRAY	名詞、すなわち、長方形の座標または軸に配列されたより単純な値からなるデータ値。すべての名詞は、0 以上の軸を持つ配列です。 行列。
寸法 DIMENSION	(配列の)軸の長さ
形状 SHAPE	(配列の)次元のリスト
スカラー SCALAR	次元のない名詞。スカラの形は空のリストです。

ランク SCALAR	(名詞の)次元の数、すなわちその形状の長さ。
ボックス BOX	その値が任意の配列を表すことができるような特別な型のスカラー。
細胞 CELL	任意の配列の次元のリストは、任意に、先行する次元に続いて後続の次元に分割することができます。したがって、元の配列は、各セルが後続の次元のみを有するセルの配列として記述される。主要な寸法は、それらのセルのフレームと呼ばれます。
フレーム FRAME	Cell を参照してください。
ランク RANK	(動詞の)自然な、または内在する、その議論のランク。固有ランクよりも高いランクの引数を指定すると、その動詞は引数の各固有ランクのセルに個別に適用されます。モナドには 1 つのランクがあり、ダイアドには 2 つのランクがあり(左右の引数にそれぞれ 1 つずつ)、したがって相反する動詞には 3 つの動詞があります。

著作権©ロジャー・ストークス 2015
最終更新 2015 年 7 月 1 日

付録３
J 言語プロセッサ PC Windows 版、Android 版のインストール

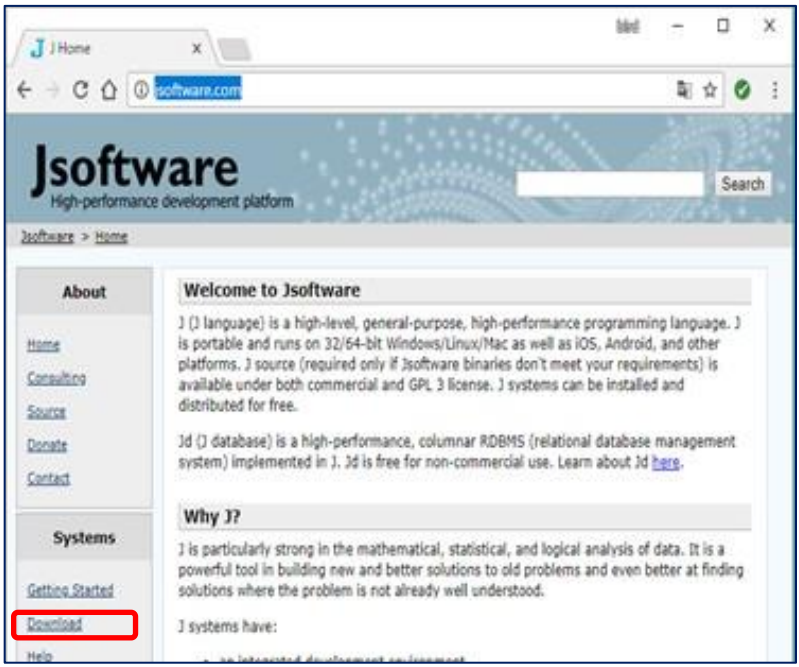
2017/12/9 JAPLA 鳥邊錬太郎

１．インストーラのダウンロード

J Processor は J 言語の総本山 jsoftware.com のサイトから Download します。DL は現在のところ、無料です。以下をクリックしてください。

jsoftware.com

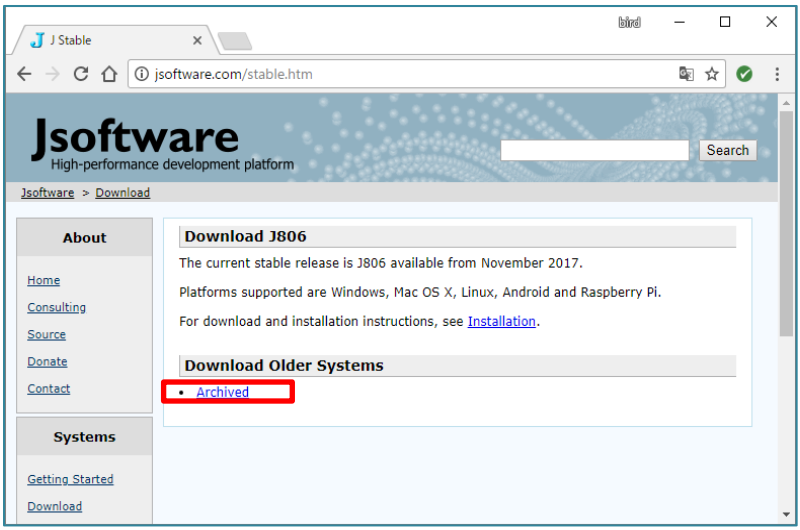
1.1.jsoftware.com の Main Page が以下のように表示されます。



1.2.J プロセッサの Download ページが表示されます。

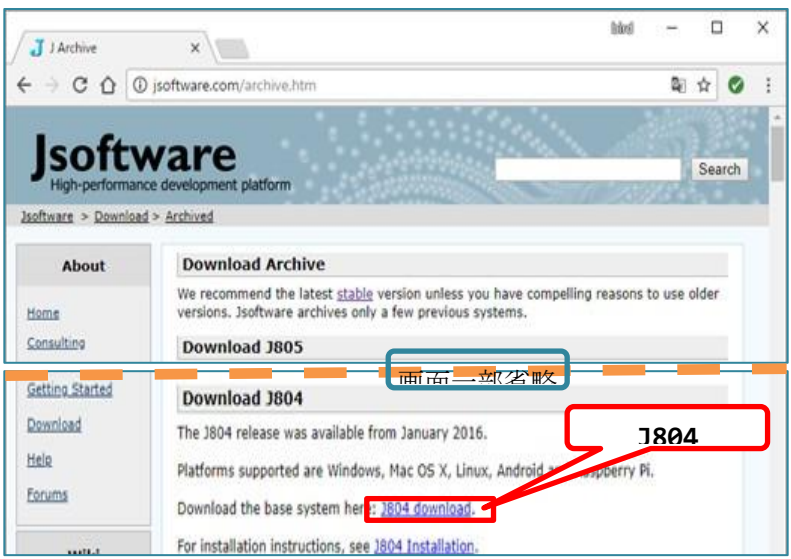
2017 年 12 月 9 日現在の最新バージョンは j806 です。しかし、ここでは比較的安定しているバージョンである j804 を対象に Download してみます。

この画面では最新の j806 以前のものは Archive されていますので、赤枠の Archived をクリックします。

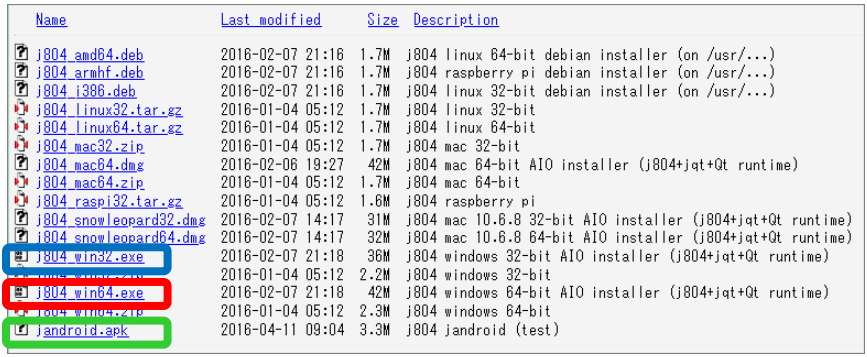


1.3.J プロセッサの Archived ページが表示されます。

目的の Download804 欄の赤枠から J804Download をクリックします。



1.3.J804 の Installer Download のページが表示されます。



Name	Last modified	Size	Description
j804_amd64.deb	2016-02-07 21:16	1.7M	j804 linux 64-bit debian installer (on /usr/...)
j804_armhf.deb	2016-02-07 21:16	1.7M	j804 raspberry pi debian installer (on /usr/...)
j804_i386.deb	2016-02-07 21:16	1.7M	j804 linux 32-bit debian installer (on /usr/...)
j804_linux32.tar.gz	2016-01-04 05:12	1.7M	j804 linux 32-bit
j804_linux64.tar.gz	2016-01-04 05:12	1.7M	j804 linux 64-bit
j804_mac32.zip	2016-01-04 05:12	1.7M	j804 mac 32-bit
j804_mac64.dmg	2016-02-06 19:27	42M	j804 mac 64-bit AIO installer (j804+jqt+Qt runtime)
j804_mac64.zip	2016-01-04 05:12	1.7M	j804 mac 64-bit
j804_raspi32.tar.gz	2016-01-04 05:12	1.6M	j804 raspberry pi
j804_snowleopard32.dmg	2016-02-07 14:17	31M	j804 mac 10.6.8 32-bit AIO installer (j804+jqt+Qt runtime)
j804_snowleopard64.dmg	2016-02-07 14:17	32M	j804 mac 10.6.8 64-bit AIO installer (j804+jqt+Qt runtime)
j804_win32.exe	2016-02-07 21:18	36M	j804 windows 32-bit AIO installer (j804+jqt+Qt runtime)
j804_win32.zip	2016-01-04 05:12	2.2M	j804 windows 32-bit
j804_win64.exe	2016-02-07 21:18	42M	j804 windows 64-bit AIO installer (j804+jqt+Qt runtime)
j804_win64.zip	2016-01-04 05:12	2.3M	j804 windows 64-bit
jandroid.apk	2016-04-11 09:04	3.3M	j804 jandroid (test)

Windows PC 版の場合は、

J804 win32.exe Windows 32Bit 版
J804 win64.exe Windows 64Bit 版

のどちらかの好みをを選択して、Download して下さい。
Install は、Download した Installer をダブルクリックすると、瞬時にインストールは完了します。

また、Android 版スマホの場合では、

jandroid.apk ndroid Smapho 版

を選択して、Download して下さい。
スマホへの Install は、次の手順で行ってください。

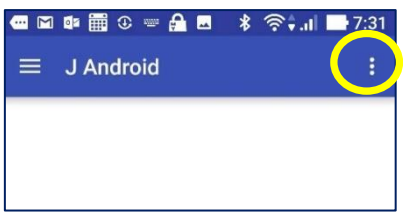
Smapho1. Download したアプリ jandroid.apk をあなたの android スマホにコピーする。

Smapho2. スマホのファイル処理アプリから、コピーした,android.spk をダブルタップするとインストールが始まります。機種によっては、インストール中に「開発者向けオプションが許可されていない」と表示がされることがあります。この時は「許可する」を選択してください。インストールされると、ホーム画面に jandroid のアイコンが表示されます（右図アイコン）。

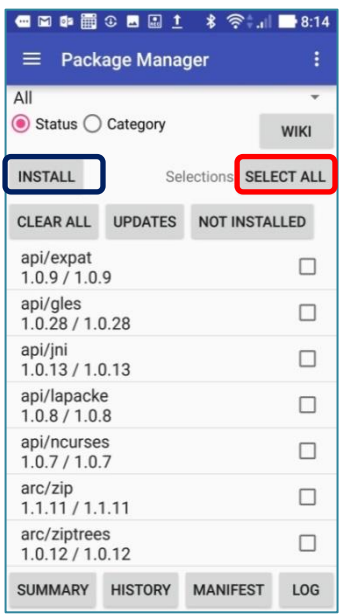
Smapho3. J 言語の実行計算式は、PC，スマホのアスキーコードをすべて使用しています。したがって、通常のキーボードでは、使いずらいのです。jAndroid は、実はインストールすると J 専用のキーボードがインストールされます。

J 専用のキーボードは、
⇒スマホの設定
⇒言語と入力
⇒仮想キーボード
から選択できます。
4 種のキーボードは、左下のキーで切り替えが
出来ます（キーボード**白枠図**参照）。

Smapho4. スマホ版 J 言語は、基本機能に関しては、インストール後、即利用出来ますが、各種アドオン機能は、各自追加インストールする必要があります。



J-Android 画面(上図)の右上（黄色〇）Menu をクリックして、
Tools ⇒ Package Manager
の順に進み、J-Andriod の必要アドオンを選択してインストールします（右図）。Adoon Name は Reqire 宣言して利用しますが、登録 Adoon グループ名と必ずしも、一致していません。



そこで、ここではすべてをインストールすることをお勧めします。J804 の場合は、すべてをインストールしても 30MB 強です。

赤枠の SELECT ALL を選択して、青枠の INSTALL をタップして下さい。

インストール時間は 5 分前後です。Adoon をインストールしておけば、グラフや数値解析などのプロセッサも利用できます。

是非、お楽しみください。

付録４

J 言語プロセッサ Android 版の基本的な使い方

2018/05/19 JAPLA 鳥邊錬太郎

1.あなたの Android スマホにインストールされた J 言語アプリはスマホのホーム画面に下図のようなアイコンで表示されています（黄色枠）。

図1: J 言語のアイコン
黄枠をタップして、J を起動する。

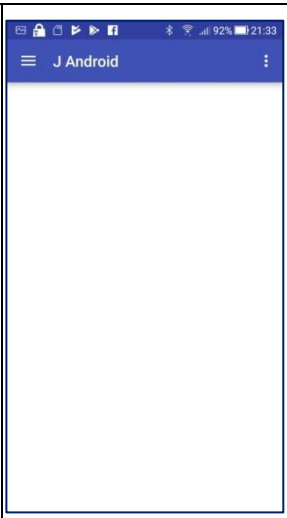


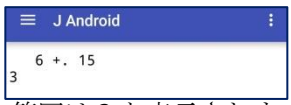
図2:J 言語の Main 画面
ここに J の計算式を記入して、タップする。

例 1： 6 と 15 の最大公約数 GCD を求める場合、

```
   6 +. 15
```

とタイプして Enter をタップします。

結果は、下の行に表示されます。



答回は 3 と表示されました。

図 3:J 言語は即時計算
インタラクティブ(対話形式)で実行します。

例 2: 複数組の最大公約数を計算する場合

```
   6 8 21 +. 15 12 48
3 4 21
```

のように、(6 と 15),(8 と 12),(21,48)の組み合わせを同時に計算できる。J 言語では、関数に与える変数はベクトルで与えることが出来る。自分で、試してみよう。

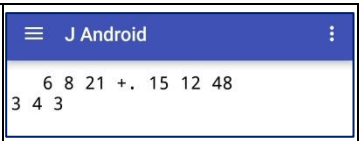
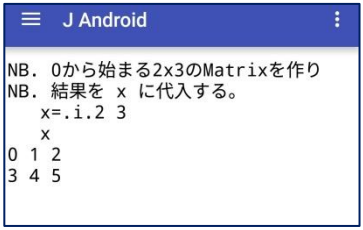


図 4:ベクトル形式計算

左 (黄色)	右(赤色)
J Android	File
Editor	Script
其の他適宜	Window
	Tools
	Help
	其の他

図 5:J Android のメニューの説明



NB.はコメント。
X=.i.2 3 の x=.のドット .
はxがローカル変数であること を定義している。ちなみに グローバル変数の時は x=: とする。
関数 i. は0から始まる整数列を生成する。

```
   i.5
0 1 2 3 4
```

となる。

```
   i.2 3
0 1 2
3 4 5
```

と 2x3 の行列を生成する。

本書の更新履歴

★ Ｊを学ぶ・スマホ版 V.1.0 2018_01_19 改定

★ Ｊを学ぶ・スマホ版 V.1.1 2018_05_19 更新

2017/12/07 By Rentaro Toribe, At Sakado

最終印刷 2018-05-19