R 言語定義 (R Language Definition)

Version 1.1.0 (2000 June 15) **DRAFT**

R Development Core Team

日本語訳注：この R-lang.texi[1] の日本語訳[2] は、英語原文と全く同じ条件の下で、自由に配布、利用、修正可能である。なおこの邦訳のソースはホームページ[3] から入手可能である。また同じホームページには R のヘルプファイル（html 形式のみ）の邦訳が翻訳済みのものから逐次公開されている。R の開発の早さから、こうした文章の日本語訳は常に"旧式化"していることをお断りしておく。最新のバージョンの R 付属の文章を適宜参照されたい。R-lang は GNU texinfo と呼ばれるマニュアル専用の TeX の方言で書かれており、TeX でコンパイル[4] する。最後に訳者[5] は R の一ユーザーにすぎず、" 解説された内容を十分理解した上で訳しているのでは決して無い" ということを警告とともにお断りしておく。(初訳：2002.1.12)

[1] バージョン 1.10。草稿段階。

[2] texinfo が日本語対応でないため完全には日本語化されていない。

[3] `http://is.titech.ac.jp/~mase/`

[4] 日本語版は、例えば日本語化 TeX の ascii ptex を用いるなら、まず "ptex R-lang.jp.v110.texi"、次に索引作成のため "texindex R-lang.jp.v110.*"、そしてもう一度 "ptex R-lang.jp.v110.texi" と三段階でコンパイルする。

[5] 間瀬茂。東京工業大学情報理工学研究科。

Table of Contents

1 序

R は統計計算とグラフィックスの為のシステムである。これは、就中、プログラミング言語、高水準グラフィックス、他の言語へのインタフェイス、そしてデバッグ用の機能を提供する。このマニュアルは R 言語の仕様の詳述し、定義を与える。

R 言語は、1980 年代にデザインされそれ以来統計関係者の間で広く使われてきた S の方言である。S の主要な開発者である John M. Chambers は 1998 年の ACM ソフトウェアシステム賞を受賞した。

言語の構文は C 言語と外見上の類似性を持つが、そのセマンティックスは Lisp と APL に強い親和性を持つ FPL (関数プログラミング言語) の変種である。特に、それは "言語による計算" を可能にし、それにより表現式を入力に取る関数を書くことを可能にする。これは統計モデリングとグラフィックスでしばしば有用になる。

コマンド行から入力した簡単な表現式を実行することによる、R の対話的な使用から極めて多くのものを得ることが出来る。このレベルを超える必要の全くないユーザーもいるであろうが、繰り返し行う作業を統一化するためにその場限りのやり方や、新しいアドオンパッケージを書くという目標のために、自分自身の関数を書きたいというユーザーもいるであろう。

このマニュアルの目的は言語自体のドキュメント化である。つまり、R の関数をプログラミングする際に知っておくと役に立つ、扱う対象と、表現式の評価の過程の詳細である。グラフィックスのような、特殊な作業用のサブシステムはこのマニュアルでは極く簡単にしか述べられず、別個にドキュメント化されるであろう。

このテキストに書かれた多くは S にも同じように該当するが、それでも幾つかの本質的な違いが存在し、ことがらを混乱させせないために R の記述に限定することにする。

この言語のデザインは、ユーザーを驚かせるかも知れない幾つかの精妙な点と、良くある陥穽を持つ。後で説明するように、これらのほとんどは深い水準での一貫性の考慮に由来する。また幾つかの有用な近道と慣用法があり、極めて複雑な操作を簡明に表現することが出来る。これらの多くは基礎にある概念に一旦なれれば自然になる。ある場合には一つの作業を行う複数の方法が存在するが、そのテクニックの幾つかは言語のインプリメント法に依存し、その他はより高い抽象度の水準において動作する。そうした場合、より好ましい使用法を指示するであろう。

R に多少の馴染みがあることを前提にする。これは R への入門書ではなく、むしろプログラマー用の参照マニュアルである。他のマニュアル、特に section "Preface" in *R Introduction*、は R への保管的な情報を提供し、section "System and foreign language interfaces" in *Writing R Extensions* はコンパイルしたコードで R を拡張する方法を詳述する。

2 オブジェクト

全ての計算機言語では変数 (又はシンボル) はメモリー中に保管されたデータにアクセスする手段を提供する。R は計算機のメモリーへの直接的なアクセス手段を提供せず、むしろ今後オブジェクトと呼ばれる幾つかの特殊化されたデータ構造を提供する。これらのオブジェクトはシンボルや変数を介して参照される。しかしながら R ではシンボルはそれ自身オブジェクトであり、他のオブジェクトと同じ方法で操作することが出来る。これは他の多くの言語と異なり、広範囲に渡る効果を持つ。

この章では R で提供される様々なデータ構造の準備的な記述を与える。それらの多くのより詳細な議論は引き続く章で与えられる。The R 固有の関数 `typeof` はある R オブジェクトの型 *type* を返す。R の基礎にある C のコードでは全てのオブジェクトは定義型 `SEXPREC` へのポインターであることを注意しよう。異なった R のデータ型は C で `SEXPTYPE` と表現されており、これは構造の様々な部分の情報がどのように使われているかを決定する。

次の表は `typeof` が返す全ての可能な値とその意味を示す。

`NULL`	空
`symbol`	変数名
`pairlist`	対リストオブジェクト
`closure`	関数
`environment`	環境
`promise`	遅延評価を実現するために使われるオブジェクト
`language`	R の言語構成
`special`	その引き数を評価しない組込み関数
`builtin`	その引き数を評価する組込み関数
`logical`	論理値を含むベクトル
`integer`	整数値を含むベクトル
`double`	実数値を含むベクトル
`complex`	複素数値を含むベクトル
`character`	文字値を含むベクトル
`...`	特殊な可変長引き数 ***
`any`	全ての型にマッチする特殊な型 ***
`expression`	表現式オブジェクト
`list`	リスト
`externalptr`	外部ポインターオブジェクト

ユーザーが '***' でマークされた項目を、すくなくとも苦労せずに、理用できるとは思えない。反例があれば教えてもらいたい。

関数 `mode` は Becker, Chambers & Wilks (1988) の意味でのオブジェクトの *mode* に関する情報を与え、他の S 言語のインプリメントと完全に互換性がある。最後に、関数 `storage.mode` は Becker et al. (1988) の意味での、その引き数の *storage mode* を返す。これは一般に、C や FORTRN といった、他言語で書かれた関数を呼び出すときに、R オブジェクトが呼び出しルーティンが期待するデータ型を持つことを保証するために使われる。(S 言語では、整数もしくは実数値ベクトルは共に`"numeric"` モードであり、従ってそれらの保管モードは区別する必要はない。)

```
> x <- 1:3
> typeof(x)
[1] "integer"
> mode(x)
[1] "numeric"
> storage.mode(x)
[1] "integer"
```

R オブジェクトはしばしば計算途中で他の型に強制変換される。明示的に強制変換を行うために使える非常に多数の関数が存在する。R 言語でプログラムを行う際、オブジェクトの型は一般に計算過程に影響を及ぼさないが、しかしながら、他の言語やオペレーティングシステムを使う際には、しばしばオブジェクトが適正な型を持つことを保証する必要が起きる。

2.1 基本型

2.1.1 ベクトル

ベクトルは等質的なデータを含む連続したセルと考えて良い。セルは `x[5]` といった添字操作によりアクセス出来る。詳細は ⟨undefined⟩ [Indexing], page ⟨undefined⟩ にある。.

R は 5 つの基本的なベクトル型を持つ: 論理、整数、実数、複素数そして文字列 (もしくは文字) である。これらのベクトルタイプのモードと保管モードが次表にある。

型	モード	保管モード
論理値	`logical`	`logical`
整数値	`numeric`	`integer`
倍精度実数値	`numeric`	`double`
複素数値	`complex`	`complex`
文字	`character`	`character`

文字列ベクトルは `"character"` というモードと保管モードを持ち、これは少し紛らわしい。

2.1.2 リスト

リスト (“総称的ベクトル”) はもう一つのデータ保管手段である。リストは要素を持ち、その各々は任意の型の R オブジェクトを含むことが出来る、つまり、リスとの要素は同じ型である必要はない。リスト要素は三つの異なった添字操作でアクセスされる。これらは ⟨undefined⟩ [Indexing], page ⟨undefined⟩ で詳しく説明される。

2.1.3 言語オブジェクト

R 言語を構成する三つのオブジェクトの型がある。*calls*, *expressions*, そして *names* である。R は型 `"expression"`のオブジェクトを持つので、他の文脈中で言葉による表現式の使用を避けることにする。特に構文的に正しい表現は *statements* という名前で参照する。

これらのオブジェクトは `"call"`, `"expression"` そして`"name"` というモードをそれぞれ持つ。

これらは表現式から `quote` 機構を用いて直接につくり出すことができ、`as.list` 関数と `as.call` 関数を用いてリストから、そしてリストへ、変換することができる。構文解析木の成分は標準的な添字操作を用いて抜き出すことができる。

2.1.3.1 シンボルオブジェクト

シンボルは R のオブジェクトを参照する。全ての R オブジェクトの名前は普通シンボルである。シンボルは関数 `quote` を通じてつくり出すことができる。.

シンボルはモード `"name"`、保管モード `"symbol"`、そして 型`"symbol"` を持つ。これらは `as.character` と `as.name` を用いて文字列から、そして文字列へ、強制変換できる。これらは構文解析済みの表現式のアトムとして自然に登場する、例えば `as.list(quote(x + y))` を試してみよ。

2.1.4 表現式オブジェクト

R では型 "expression" のオブジェクトを持つことができる。一つの *expression* は一つもしくは複数の文を含む。文とは構文的に正しいシンボルの集りである。表現式オブジェクトは、それが構文解析済みの、しかし未評価の R 文を含むという点で言語オブジェクトに似ている。主な違いは、表現式オブジェクトは複数のそうした表現をふくむことができることにある。もう一つのより微妙な差異は、型 "expression" のオブジェクトは明示的に eval に引き渡されたときだけ評価される一方、言語オブジェクトは思わぬところで評価される可能性があることにある。

表現式オブジェクトはリストと良く似た振舞を持ち、その成分はリストの成分と同様の仕方でアクセスできる。

2.1.5 関数オブジェクト

R では関数はオブジェクトであり、他のオブジェクトと良く似た方法で操作できる。関数は三つの基本成分、形式的引数のリスト、本体、そして環境を持つ。引数リストはコンマで区切られた引数のリストである。引数はシンボル、'*symbol* = *default*' という表現、もしくは特殊な引数 '...' のいずれかである。引数の第二形式はある引数の既定値を指定するのに使われる。この値は関数がその引数に対し特に値の指定なしで呼び出されたときに使われる。'...' 引数は特別であり、任意個数の引数を含むことができる。普通これは引数の数が未知の場合や、引数が他の関数に引き渡される際に使われる。

関数本体は構文解析済の R の文である。普通中括弧で挟まれた文の集りであるが、単一の文、シンボル、さらには定数でもよい。

関数の環境は関数がつくり出されたときに有効であった環境である。その環境に付随していた全てのシンボルは捕捉され、関数から使用可能になる。

クロージャーオブジェクトの三つの部分を formals, body そして environment を用いて取り出し、操作することが可能である (また三つのどれもが代入式の左辺項に使うことができる)。最後の一つはしばしば不必要な環境取り込み要素を取り除くのに使われる。as.list と as.function を用い、関数をリスト構造へ、そしてリスト構造から関数へ、変換する機能がある。これらは S との互換性を維持するために提供されているが、使わないことを勧める。

2.1.6 NULL

NULL と呼ばれる特殊なオブジェクトがある。これはあるオブジェクトが存在しないことを指示する必要があるたびに使われる。多くのリスプの実装では NULL は単に長さゼロのリストであるが、R は Scheme を模範としているもののそれとは異なった扱いがされている。しかしながら、長さゼロの pairlist は NULL である。

NULL オブジェクトは型も変更可能な性質も持たない。R には唯一つの NULL オブジェクトがあり、すべての個別例はそれを参照する。NULL に属性を設定することはできない。.

2.1.7 組込みオブジェクトと特殊形式

これらの二種類のオブジェクトは R の組込み関数を含む、つまり、これらはコードのリスティングでは .Primitive と表示される。両者の違いは引数の処理にある。組込み関数はその全ての引数を評価し、基本関数に値渡しで引き渡す。他方で特殊関数は未評価の表現式を内部関数に引き渡す。

R 言語から見れば、それらの定義が表示できないことを除けば、これらのオブジェクトは単に別種の関数にすぎない。typeof 関数を使ってこれらとコード解釈型関数とを区別できる。

2.1.8 Promise オブジェクト

予約[1] オブジェクトは R の遅延評価機構の一部である。これらは三つの中身、値、表現式そして環境を持つ。もしある関数が呼ばれると、引数が照合されそして形式的引数の各々が一つの予約に結びつけられる。その形式的引数に与えられた表現式と、その関数が呼び出された環境へのポインターがその予約に保管される。

その引数がアクセスされるまで、その予約に伴ういかなる値も存在しない。引数がアクセスされると、保管された表現式が保管された環境中で評価され、そして結果が返される。結果もまた予約により保存される。こうすることにより、プログラマーは予約に伴う値と表現式の双方にアクセスすることができる。

R 言語中では、予約オブジェクトはほとんど隠蔽されている。実際の関数引数はこの型である (これが遅延評価機構の核心である)。`substitute` は中身の表現式内容を取り出し、表現式から予約を作り出す `delay` 関数がある。あるオブジェクトが予約でるあるかどうか検査する確実な方法は無い。

2.1.9 ドット

ドットオブジェクト '`...`' はリストタイプとして保管される。'`...`' の要素は C コードから通常のリストとして呼び出せるが、R の内部からはオブジェクトとしてアクセスできない。しかしながら、オブジェクトはリストとして捕捉出来る。したがって例えば `table` 中では次のように使われている

```
     args <- list(...)
 ## ....
     for (a in args) {
 ## ....
```

もし関数が '`...`' を形式的引数として持てば、形式的引数としてマッチしない全ての実際の引数が '`...`' とマッチする。

2.1.10 環境

環境は二つのものからなると考えることができる。フレーム、つまりシンボルと値の対の集りと、それを取り巻く環境とである。あるシンボルの値が必要になると、フレームが検査され、マッチするシンボルが見付かると、それが返される。さもなければその外側の環境が次に調べられる、等が繰り返される。環境は木構造を作り、一つの環境は複数の子環境を持つことができるが、親は唯一つだけである。

環境は関数呼び出しで暗黙のうちに作り出されるが、直接 `new.env` により作り出すこともできる。一つの環境のフレームの内容は can be accesses and manipulated by `get` と `assign` そして `eval` と `evalq` によりアクセスし、操作できる。

現在のところ一つの環境の上位環境を直接アクセスする方法は無い。

他のあらゆる R オブジェクトとは異なり環境はコピーできない。したがって、もし同じ環境に複数のシンボルを割り当て、その一つを変更すると、他のものも変わってしまう。特に環境に属性を割り当てるとビックリするようなことが起きる可能性がある。

2.1.11 対リストオブジェクト

対リストオブジェクトはリスプのドット対リストに似る。これは R の内部で徹底的に使われているが、インタプリータコードではほとんど目に付くことは無い。しかしながらこれは `formals` の返

---

[1] 訳註 :「予約」は promise の訳。「約束」、「契約」?

値であるし、例えば pairlist 関数で作り出すこともできる。そうしたオブジェクトの各々は三つの中身、CAR 値、CDR 値、そして TAG 値を持つ。TAG 値はテキスト文字列であり、CAR と CDR は普通それぞれ NULL オブジェクトを終結子に持つリストのリスト項目 (先頭) とその残り (末尾) を表す (CAR/CDR という概念はリスプに伝統的なものであり、もともと 60 年代初期の IBM の計算機のアドレスと減算レジスターを指していた)。

対リストは R 言語では総称ベクトル (“lists”) と全く同じように扱われる。特に、要素は同じ [[]] 構文を用いてアクセスされる。総称的ベクトルのほうが普通より効率的に使えるため、対リストの使用には異義が唱えられている。内部的な対リストが R からアクセスされると普通それは総称的ベクトルへ、そして総称的ベクトルから、変換される。

2.1.12 “any” 型

あるオブジェクトが “Any” 型であることは不可能であるが、それにも関らずこれは適正な型値である。これは或る種の (かなり稀な) 状況で使われる。例としては as.vector(x, "any") があり、型強制変換をすべきでないことを指示する。

2.2 属性

全てのオブジェクトはそれらに付随する一つもしくは複数の属性を持つことができる。属性はリストとして保管され、その全ての成分は名前を持つ。属性のリストは attributes を用いて得ることができ、attributes<- を用いて設定でき、個々の要素は attr と attr<- を用いてアクセスできる。

幾つかの属性は特別なアクセス用関数を持ち (例えば因子用の levels<-)、それらは普通付加的な操作を行うため利用できるなら使うべきである。R は特殊な属性が関係する attr<- と attributes<- への呼び出しを傍受し一貫性検査の強要を試みる。

行列と配列は属性 dim とそのベクトルに対するオプションの属性 dimnames を持つ単なるベクトルである。

属性は R で使われるクラス構造を実装するために使われる。もしオブジェクトが class 属性を持てば、その属性は評価の過程で検査される。R のクラス属性は Chapter 5 [オブジェクト指向プログラミング], page 22 で詳しく説明される。

2.2.1 名前

names 属性は存在すればベクトルもしくはリストの個々の要素のラベルとなる。オブジェクトが出力されるときは names 属性は存在すれば要素のラベルとして使われる。names 属性はまた添字操作にも使える、例えば quantile(x)["25%"]。

名前は names と names<- を使って獲得したり設定したりできる。後者は名前属性が適正な長さの型を持つかどうか必要な一貫性検査を行うだろう。

対リストオブジェクトに対しては仮想的な names 属性が使われ、names 属性は実際はリスト成分のタグから構築される。

一次元の配列に対しては names 属性は実際は dimnames[[1]] を参照する。

2.2.2 次元

dim 属性は配列を実装するために使われる。配列の中身は列主導の順序でベクトルに保存され、the dim 属性は整数のベクトルで配列の対応する範囲を指定する。R はベクトルの長さが次元の積であることを保証する。

ベクトルは一次元の配列ではなく、後者は長さ 1 の dim 属性をもつ一方、前者は dim 属性を持たない。

2.2.3 次元名

配列は文字ベクトルのリストである dimnames 属性を用いて各次元に名前を持つことができる。dimnames リスト自身も名前を持つことができ、もし持てば配列を出力するときに範囲の見出しに使われる。

2.2.4 クラス

R は洗練されたクラスのシステムを持っており、class 属性で制御される。この属性はオブジェクトが継承するクラスのリストを含む文字ベクトルである。これは R の "総称的メソッド" 機能の基礎をかたちづくる。

この属性はユーザーが制限無しに仮想的にアクセスし操作できる。オブジェクトが実際にクラスのメソッドが期待するような成分を持つかどうかの検査は行われない。したがって class 属性の変更は注意深く行われるべきであり、もし利用できるなら固有の構築・強制変換関数を使う方が好ましい。

2.2.5 時系列属性

tsp 属性は時系列のパラメータ、始点、終点、そして周期を保管するために使われる。この構成は主に月毎や四半期毎のデータのような周期的構造をもつ系列を扱うために使われる。

2.3 特殊な複合オブジェクト

2.3.1 因子

因子は有限個の値を持つ項目 (性別、社会的クラス等) を記述するために使われる。因子は labels 属性とクラス "factor" を持つ。オプションとして contrasts 属性を持つことができ、これは因子がモデル化関数中で使われるときパラメータ化を制御する。

因子は純粋に項目だけでも良く、順序付きのカテゴリーでも良い。後者では、そうしたものとして定義される class ベクトル c("ordered"," factor") を持つべきである。

因子は普通実際の水準を指定する整数配列と、その整数に対応する名前の二次的配列を用いて実装される。どちらからといえば不幸なことに、ユーザーはしばしばある計算をより簡単にするためにこの実装を利用する。しかしながら、これは実装の問題であり、 R の全ての移植でこれが正しいことが保証されるわけではない。

2.3.2 データフレームオブジェクト

データフレームは SAS もしくは SPSS のデータセット、 データの "変量による場合分け" 行列、をもっとも真似た R の構造である。

データフレームはすべて同じ長さ (行列の場合は行数) のベクトル、因子、そして (又は) 行列のリストである。加えて、データフレームは一般に変量のラベルである names 属性と場合のラベルである row.names 属性を持つ。

データフレームは他と同じ長さをもつリストを含んでも良い。リストは異なった長さの要素を含んでも良く、そうすることにより、不揃いのデータ構造を提供する。しかしながら、そうしたものとして、そうした配列は普通正確には処理されない。

3 表現式の評価

ユーザーがプロンプトに対しある命令をタイプする (もしくは表現式がファイルから読み込まれる) と、それに対し最初に起こることはパーサーによりその命令が内部表現に変換されることである。評価機構は構文解析された R の表現式を実行し、表現式の値を返す。全ての表現式は値を持つ。これがこの言語の核となる事実である。

この章は基本的な評価機構を説明するが、あとの別の章で説明される個々の関数や関数グループ、そしてヘルプ文章が十分な説明を含むことがらの議論は除外する。

ユーザーは表現式を構築し、それに対し評価機構を起動する。

3.1 単純な評価

3.1.1 定数

プロンプトに直接タイプされたあらゆる数は定数であり、評価される。

```
> 1
[1] 1
```

定数は全くつまらないものであり、より多くのことを行うにはシンボルが必要になる。

3.1.2 シンボル照合

新しい変数が作り出されると、それを参照するための名前を持つ必要があり、それは普通ある値を持つ。名前それ自体はシンボルである。あとでシンボルに付随する値をどのように決めるかを詳細に説明する。

以下の小さな例では y はシンボルで、その値は 4 である。シンボルも R のオブジェクトであるが、“言語自体のプログラミング” を行う場合を除き、ふつうシンボルを扱う必要はほとんど無い (Chapter 6 [言語自体のプログラミング], page 28)。

```
> y <- 4
> y
[1] 4
```

3.1.3 関数呼び出し

R で行われるほとんどの計算は関数の評価を含む。このことをまた関数の 起動 と呼ぶことにする。関数はコンマで区切られた引数のリストをもつ名前で起動される。

```
> mean(1:10)
[1] 5.5
```

この例では関数 `mean` が一つの引数、1 から 10 までの整数ベクトル、で起動されている。

R は異なった目的のための膨大な関数を含む。その多くは R のオブジェクトである結果を生成するために使われるが、例えば表示やプロット関数のような、その副次的効果のために使われるものもある。

関数呼び出しは タグ付き 引数を持つことができる、例えば `plot(x, y, pch = 3)`。タグ無しの引数は関数がそれらの意味を呼出時の引数の位置から区別するために 位置依存 と呼ばれる。つまり `x` は水平軸座標で `y` は垂直軸座標を意味する。タグの利用はたくさんのオプション引数を持つ関数に対し明白な便宜をもたらす。

次の例のように、特殊なタイプの関数呼び出しが代入演算の左辺項に現れることができる。

```
> class(x) <- "foo"
```

この構文が実際に行うことは、関数 `class<-` を元のオブジェクトと右辺のそれとで呼び出すことである。この関数はオブジェクトの変更を行い、結果を返し、それは元の変数に保管しなおされる。(少なくとも概念的にはこれが実際に起こることである。不要なデータの重複を避けるためにある付加的な工夫がされる。)

3.1.4 演算子

R では C 言語と類似した演算子を用いた数値表現式の使用が可能である。例えば、

```
> 1 + 2
[1] 3
```

表現式は括弧を用いたグループ化、関数呼び出しとの混合、直接的なやり方での変数への代入ができる。

```
> y <- 2 * (a + log(x))
```

R はいくつかの演算子を持つ。それらは以下の表にリストされる。

<table>
<tr><td><code>-</code></td><td>差、単項演算子としても二項演算子としても使える</td></tr>
<tr><td><code>+</code></td><td>和、単項演算子としても二項演算子としても使える</td></tr>
<tr><td><code>!</code></td><td>単項演算子の否定</td></tr>
<tr><td><code>~</code></td><td>チルダ、公式で使われ、単項演算子としても二項演算子としても使える</td></tr>
<tr><td><code>?</code></td><td>ヘルプ</td></tr>
<tr><td><code>:</code></td><td>数列、二項演算子</td></tr>
<tr><td><code>*</code></td><td>Multiplication, 二項演算子</td></tr>
<tr><td><code>/</code></td><td>Division, 二項演算子</td></tr>
<tr><td><code>^</code></td><td>Exponentiation, 二項演算子</td></tr>
<tr><td><code>%x%</code></td><td>Special 二項演算子 operators, x can be replaced by anything</td></tr>
<tr><td><code>%</code></td><td>Modulus, 二項演算子</td></tr>
<tr><td><code><</code></td><td>Less than, 二項演算子</td></tr>
<tr><td><code>></code></td><td>Greater than, 二項演算子</td></tr>
<tr><td><code>==</code></td><td>Equal to, 二項演算子</td></tr>
<tr><td><code>>=</code></td><td>Greater than or equal to, 二項演算子</td></tr>
<tr><td><code><=</code></td><td>Less than or equal to, 二項演算子</td></tr>
<tr><td><code>&</code></td><td>And, 二項演算子, ベクトル ized</td></tr>
<tr><td><code>&&</code></td><td>And, 二項演算子, not ベクトル ized</td></tr>
<tr><td><code>|</code></td><td>Or, 二項演算子, ベクトル ized</td></tr>
<tr><td><code>||</code></td><td>Or, 二項演算子, not ベクトル ized</td></tr>
<tr><td><code><-</code></td><td>Left assignment, 二項演算子</td></tr>
<tr><td><code>_</code></td><td>Left assignment, 二項演算子, shouldn't be used</td></tr>
<tr><td><code>-></code></td><td>Right assignment, 二項演算子</td></tr>
<tr><td><code>$</code></td><td>List subset, 二項演算子</td></tr>
</table>

構文の違いを別にすれば、演算子の適用と関数呼出しの間に差はない。実際 `x + y` は `"+"(x, y)` と書いても同じことである。'`+`' は非標準的な関数名を持つため引用符で囲むことが必要なことを注意しよう。

R はベクトルの全体を一度に処理でき、ほとんどの初等的演算子と `log` といった基本的数学関数はベクトル化されている (上の表のように)。これは例えば 同じ長さのベクトルの和は要素毎の和を含むベクトルをつくり出し、その際暗黙のうちにベクトルの添字に関するループ操作を行っている。これは `-`, `*` そして `/` といった演算子でも、またより高次元の構造に対しても成り立つ。特に、二つ

の行列の積は通常の行列積を生み出さない (このためには %*% 演算子が存在する) ことを注意しよう。ベクトル化された演算子に関するより詳細な点は Section 3.3 [基本的算術演算], page 13 で議論される。

ベクトルの個々の要素にアクセスするためには普通構文 x[i] を使う。

```
> x <- rnorm(5)
> x
[1] -0.12526937 -0.27961154 -1.03718717 -0.08156527  1.37167090
> x[2]
[1] -0.2796115
```

リストの成分には x$a や x[[i]] を使うことがより普通である。

```
> x <- options()
> x$prompt
[1] "> "
```

添字操作構文はまた代入文の右辺項に登場することができる。

他の演算子と同様に、添字操作は実際は関数で行われ、"["(x, 2) を x[2] の代わりに使うこともできる。

R の添字操作は Section 3.4 [添字操作], page 14 で更に説明される多くのより高度な特徴を持つ。

3.2 制御構造

R における計算は *文* を順次評価していくことによりなされる。x<-1:10 もしくは mean(y) といった文はセミコロンもしくは新しい行で区切ることができる。構文的に完全な文が評価機構に与えられると、その文は評価され *値* が返される。文の評価[1] の結果は文の値として参照できるこの値は常にシンボルに代入できる。

セミコロンと新しい行は共に文の区切りに使える。セミコロンは常に文の終りを意味するが、新しい行は文の終りを指示するかも知れない。もし現在の文が構文的に完全でないと、評価機構は新しい行を単に無視する。もしセッションが対話的なら、プロンプトが '>' から '+' へ代わる。

```
> x <- 0; x + 5
[1] 5
> y <- 1:10
> 1; 2
[1] 1
[1] 2
```

複数の文は括弧 '{' と '}' を用いてグループ化できる。文のグループは *ブロック* と呼ばれることもある。単一の文は構文的に完全な文の最後に新しい行がタイプされると評価される。ブロックは閉じ括弧の後に新しい行が挿入されるまで評価されない。この節の残りでは、*文* とは単一の文かブロックを意味する。

```
> { x <- 0
+ x + 5
+ }
[1] 5
```

[1] 評価は常にある環境で行われる。詳細は Section 3.5 [変数のスコープ], page 16 を見よ。

3.2.1 if 文

`if/else` 文は条件に応じて二つの文を評価する。評価される 条件 があり、もしその値 が `TRUE` なら最初の文が評価され、さもなければ二つ目の文が評価される。`if/else` 文はその値として選択された文の値を返す。形式的な構文は

```
if ( statement1 )
     statement2
else
     statement3
```

最初に *statement1* が評価され *value1* を返す。もし *value1* が論理値ベクトルでその最初の要素が `TRUE` なら *statement2* が評価される。もし *value1* の最初の要素が `FALSE` なら *statement3* が評価される。もし *value1* が数値ベクトルなら *value1* の最初の要素がゼロなら *statement3* が評価され、さもなければ *statement2* が評価される。*value1* の最初の要素だけが使われる。他の全ての要素は無視される。もし *value1* が論理値・数値以外の型を持てば、エラーが起きる。

if/else 文は負の数値の対数を取るといった数値的な問題を避けるのに使うことができる。 if/else 文は他の文と同じように値を指定できる。次の二つの例は同値である。

```
> if( any(x) <= 0 ) y <- log(1+x) else y <- log(x)
> y <- if( any(x)<= 0 ) log(1+x) else log(x)
```

`else` 節は無くても良い。文 `if(any(x) <= 0) x <- x[x <= 0]` は正しい構文である。もし `if` 文がブロック中に無ければ、`else` は、もしあるとすれば、*statement1* と同じ行上に無ければならない。さもなければ *statement1* の最後にある行換えは構文的に正しい文を生み出し、そのまま評価されてしまう。

if/else 文は入れ子になっても良い。

```
if ( statement1 )
     statement2
else if ( statement3 )
     statement4
else if ( statement5 )
     statement6
else
     statement8
```

偶数番号の文が評価され、結果の値が返される。もしオプションの `else` 節が省略される、全ての奇数番号の節が `FALSE` と評価されると、どの文も評価されず `NULL` が返される。

奇数番号の *statements* はどれかが `TRUE` になるまで順に評価され、それに対応する偶数番号の *statement* が評価される。この例では *statement6* は *statement1* が `FALSE`、*statement3* が `FALSE`、そして *statement5* が `TRUE` の時だけ評価される。`else if` 節の数はどれだけあっても良い。

3.2.2 繰り返し文

R は明示的な繰り返し[2] を行う三つの文を持つ `for`、`while` そして `repeat` 文である。二つの組込みの構成子 `next` と `break` は評価に関する付加的な制御を与える。三つの文の各々は評価された最後の文の値を返す。普通ではないが、これらの文の結果をシンボルに代入することが可能である。R は `tapply`、`apply` そして and `lapply` といった暗黙の繰り返しのための関数を提供する。加えて多くの演算、特に数値的なものは、ベクトル化されており、繰り返しを使う必要は無いかも知れない。.

---
[2] 繰り返し文は文もしくはブロック文を繰り返し評価することを意味する

繰り返しを明示的に制御するのに使える二つの文がある。`break` と `next` である。`break` 文は現在実行中の最も内意の繰り返しから抜け出す。`next` 文は直ちに繰り返しの最初へ制御を移す。それから繰り返しの次が (もしそれがあれば) 実行される。現在の繰り返しの `next` 以降の文は評価されない。

3.2.3 reapet 文

`reapet` 文 は特に中断が要求されるまで本体を繰り返し評価させる。これは `repeat` 文を使うと果てしない繰り返しを行う危険があることを意味し注意が必要になることを意味する。`repeat`による繰り返しの構文は次のようになる。

`repeat` *statement*

`repeat` を使うときは *statement* はブロック文でなければならない。ある計算とともに繰り返しの中断の検査が必要になり、従って普通二つの文が必要になる。

3.2.4 while 文

`while` 文は `repeat` によく似ている。`while` による繰り返しの構文は次のようになる。

`while (` *statement1* `)` *statement2*

ここで *statement1* が評価され、もしこれの値が `TRUE` なら *statement2* が実行される。この過程は *statement1* の評価が `FALSE` になるまで続く。もし *statement2* が一度も評価されないと `while` は `NULL` を返し、さもなければ *statement2.* の最後の評価が値として返される。

3.2.5 for

`for` による繰り返しの構文は次のようになる。

`for (` *name* `in` *vector* `)`
  statement1

ここで *vector* はベクトルでもリストでも良い。*vector* 中の各要素に対し、その値が変数 *name* に設定され *statement1* が評価される。副次的効果として変数 *name* は繰り返しが終了した後も存続し、その値は繰り返しが評価した *vector* の最後の要素の値を持つ。

3.2.6 switch

技術的には `switch` は単なる一つの関数であるが、その構文は他のプログラム言語の制御構造のそれに近い。

その構文は次のようになる。

`switch (`*statement*`,` *list*`)`

ここで *list* の成分は名前を持っても良い。最初に *statement* が評価されその結果 varvalue が得られる。もし *value* が 1 と *list* の長さの間の数であれば、対応する *list* の成分が評価され結果が返される。もし *value* が大きすぎるか小さすぎると `NULL` が返される。

```
> x <- 3
> switch(x, 2+2, mean(1:10), rnorm(5))
[1]  2.2903605  2.3271663 -0.7060073  1.3622045 -0.2892720
> switch(2, 2+2, mean(1:10), rnorm(5))
[1] 5.5
> switch(6, 2+2, mean(1:10), rnorm(5))
```

```
NULL
```

もし *value* が文字ベクトルなら *value* に正確に照合する名前を持つ '...' の成分が返される。もしどれとも一致しなければ NULL が返される。

```
> y <- "fruit"
> switch(y, fruit = "banana", vegetable = "broccoli", meat = "beef")
[1] "banana"
```

switch の普通の使用法は関数のある引数の文字値に応じて分岐処理することである。

```
> centre <- function(x, type) {
+ switch(type,
+        mean = mean(x),
+        median = median(x),
+        trimmed = mean(x, trim = .1))
+ }
> x <- rcauchy(10)
> centre(x, "mean")
[1] 0.8760325
> centre(x, "median")
[1] 0.5360891
> centre(x, "trimmed")
[1] 0.6086504
```

switch は評価された文の値か、どの文も評価されなければ NULL を返す。

既存の選択肢のリストから選ぶためには switch は評価すべき項目を選ぶ最良の方法では無いかもしれない。しばしば eval(x[[condition]]) 経由で eval と部分集合演算子 [[を直接使う方が望ましい。

3.3 基本的算術演算

この節では二つのベクトルもしくは行列の和や積といった基本的な演算に適用される細かな点を議論する。

3.3.1 リサイクル規則

もし異なった数の要素を持つ二つの構造の和を取ろうとすると、短い方が最も長い方の長さまでリサイクル使用される。つまり、例えばもし c(1, 2, 3) を 6 つの要素のベクトルに加えると、実際は c(1, 2, 3, 1, 2, 3) を加えることになる。もし長い方のベクトルの長さが短い方の長さの倍数でないと、警告が出される。

一つの例外としてベクトルを行列に足すときは、長さが不揃いでも警告は出されない。

3.3.2 名前の伝播

名前の伝播 (最初のものが勝者になる、と思う。もしそれが名前を持たなければ?? —- *名前を持つ* 最初のものが勝者になる。リサイクル規則は短い方が名前を失うようにする)

3.3.3 次元属性

(行列同士の和では次元が一致する必要。ベクトルと行列の和では、ベクトルがリサイクル使用され、次元が適合するかどうか検査され、もし適合しなければエラーとなる)

3.3.4 NA の処理

3.4 添字操作

R には添字の操作により個別の要素もしくは部分集合にアクセスすることを許すいくつかの機構がある。基本であるベクトル型では i-番目の要素は x[i] で得られるが、またリスト、行列、そして多次元配列も添字操作できる。単一の整数による添字操作に加えいくつかの添字操作がある。

R は三つの基本的な添字演算子を持ち、その構文は次の例で示される。

```
x[i]
x[i, j]
x[[i]]
x[[i, j]]
x$a
x$"a"
```

ベクトルや行列に対しては [[はほとんど使われないが、これは構文的には [形と少し意味が異なり、names や dimnames 属性を取り除く。多次元の構造を単一の添字で添字操作する際、x[[i]] や x[i] は x. の系列的に i 番目の要素を返す。

リストに対しては、単一の要素を選択するには一般に [[を使うが、[ではベクトルによる添字指定ができる。

$ を使った形式はリストオブジェクトに対し使われる。これは文字通りの文字列もしくは添字を意味するシンボルだけを使う。つまり、添字は計算できる量ではない。添字を見いだすためにある表現式を評価する必要があれば x[[expr]] が使える。

3.4.1 ベクトルによる添字操作

R にはベクトルを添字として使ういくつかの強力な構成法がある。単純なベクトルの添字操作を最初に議論しよう。単純のために表現は x[i] であるとしよう。そうすると i の型に応じて以下の可能性が存在する。

- 整数。 i の全ての要素は同じ符号でなければならない。もしそれらが正ならば x のこれらの添字を持つ要素が選択される。もし i が負の要素からなれば、これらの添字持つ要素以外の全てが選択される。

  もし i が正で length(x) を越えれば、対応する選択は NA である。i が負で限界を越えるならばエラーになる。

  特殊な場合としてゼロの添字があり、これは何も惹き起こさない。x[0] は空のベクトルであり、また正もしくは負の添字中のゼロはまるでそれが存在しないかのように扱われる。
- 他の数値。 整数でない値は使用前に整数に変換される。
- 論理値。添字 i は x と同じ長さでなければならない。もしこれが短ければ、その要素は Section 3.3 [基本的算術演算], page 13 で述べられたようにリサイクル使用される。もしこれが長ければ、x は NA を用いて延長される。選択される x の要素は i が TRUE であるものである。.
- 文字。i 中の文字列は x の名前属性と照合され、対応する整数が使われる。
- 因子。結果は x[as.integer(i)] と同じである。因子の水準は決して使われない。もしその方が良ければ、x[as.character(i)] または類似の構成を使う。
- 空。 表現 x[] は x を返すが、結果から “関係の無い” 属性を取り除く。names と、多次元配列の場合は dim と dimnames 属性が残される。

欠損値 (つまり NA) を持つ添字操作は NA を返す。この規則は論理値添字指定の場合も適用される。つまり、 i 中に NA 選択子を持つ x の要素は結果に含まれるが、その値は NA になるであろう。

しかしながら NA には異なったモードがあることを注意しよう。文字通りの定数はモード "logical" であるが、これはしばしば自動的に他の型に強制変換される。これの一つの効果は x[NA] は x と同じ長さを持つが x[c(1, NA)] は長さ 2 を持つことである。なぜなら論理値添字に対する規則が前者に適応されるが、項者では整数添字に対する規則が適応されるからである。

[を用いた添字操作もまた関連する全ての名前属性の部分集合化を実行する。

3.4.2 行列と配列の添字操作

多次元構造の部分集合抽出は一般的に、names の代わりに dimnames の成分を用いた、各添字変数に対する一次元添字操作と同じ規則に従う。しかしながら、いくつかの特殊規則が適用される。

普通、一つの構造はその次元に対応するいくつかの添字を用いてアクセスされる。しかしながら又単独の添字を使うことも可能で、その時は dim と dimnames 属性は無視され、結果は実際には c(m)[i] のそれと同じになる。m[1] は普通 m[1,] もしくは m[, 1] とかなり違うものになることを注意しよう。

添字の行列を添字として使うこともできる。この場合、行列の列数は構造の次元の数と一致する必要があり、結果は行列の列数を長さに持つベクトルになる。次の例は要素 m[1, 1] と m[2, 2] を一回の操作で取り出す。

```
> m <- matrix(1:4, 2)
> m
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> i <- matrix(c(1, 1, 2, 2), 2, byrow = TRUE)
> i
     [,1] [,2]
[1,]    1    1
[2,]    2    2
> m[i]
[1] 1 4
```

単一の添字を用いる場合も行列添字を用いる場合も、names 属性が存在すれば、構造が一次元であるかのように使用される。三次元行列から例えば m[2, ,] を用いて単一の切片を選び出す場合のように、もし添字操作により結果がある方向に長さ 1 を持つようになれば、対応する次元は一般に結果から取り除かれる。もし一次元の構造が結果として得られれば、結果はベクトルになる。これは場合によると好ましくなく、添字操作に 'drop = FALSE' を付け加えれば無効にできる。これは関数 [への追加の引数であり、添字には影響しないことを注意しよう。従ってある行列の第一行を 1 掛ける n 行列として選択する正しいやり方は m[1, , drop=FALSE] となる。次元を取り去る機能を無効にすることを忘れることは、添字が時おり、しかし一般にではなく長さ 1 を持つ添字を持つ一般的なサブルーチンにおける良くある失敗の原因になる。

一次元の配列は dim と dimnames 属性を持つ (ともに長さ 1) 点でベクトルとは異なることを注意しよう。こうした構造は部分集合取り出し演算では容易に得ることはできないが、明示的に構成することができる。 dimnames リストの要素はそれ自身名前をもつことがある一方、names 属性はそうでないため、これは時々役に立つ。

m[FALSE,] のような演算は次元がゼロの構造を作り出すことがある。R はこうした場合を注意深く処理するように努める。

3.4.3 他の構造の添字操作

演算し `[` は総称的な関数であり、クラス毎にメソッドを加えることができ、演算し `$` と `[[` も同様である。任意の構造に対しユーザー定義の添字演算子を定義することができる。そうした関数、例えば `[.foo`、は幾つかの引数を持ち、その最初は添字処理される構造であり、残りは添字である。`$` の場合は添字引数は `x$"abc"` 形式を使う場合もモード `"symbol"` を持つ。

ユーザー定義の添字演算の最も重要な例はデータフレームに対するものである。ここで詳しく述べることはしないが、広い意味で、基本的に同じ長さのベクトルからなるリストである構造に対する行列風の添字操作を定義できる。

3.4.4 部分集合指定

(訳注：この部分は未完成)

3.5 変数のスコープ

ほとんど全てのプログラミング言語は幾つかのスコープ規則を持ち、異なったオブジェクトが同じ名前を持つことができるようになる。これは例えば関数中の局所的な変数が大局的なオブジェクトと同じ名前を持つことを許す。

R はパスカルと同様 辞書式スコープモデルを持つ。しかしながら R は 関数プログラミング言語 であり関数と言語オブジェクトを動的につくり出すことを許し、この事実を反映し追加的な特徴を持つ。

3.5.1 大局的環境

大局的環境はユーザーの作業スペースの基礎である。命令行から代入演算を入力すると対応するオブジェクトは大局的環境に属する。

3.5.2 辞書式環境

ある関数への全ての呼出は関数中で生成される局所的変数を含む フレーム をつくり出し、それからある環境中で評価され、この二つが併せて新しい環境をつくり出す。

用語に注意しよう: フレームとは変数の集まりであり、環境とは入れ子状のフレームである (または同じことだが: いちばん奥のフレームとそれを囲む環境)。

環境は変数に代入できるし、他のオブジェクトに含めることもできる。しかしながら、それらはそれ自身オブジェクトではなく、特に代入によってコピーされることはない。

閉包 (`"function"` モード) オブジェクトはそれがその定義の一部分としてつくり出された環境を含む。(既定では。環境は `environment<-` を用いて操作できる)。それから関数が続いて呼び出され、その評価環境はその定義環境中に入れ子になる。これは呼び出し側の環境であるとは限らないことを注意しよう。

このように、ある変数が関数の内部で要求されると、それは先ず評価環境中、次にその外側の環境、さらにその外側の環境、といった風に探される。

3.5.3 呼び出しスタック

関数が起動されるたびに、新しい評価フレームがつくり出される。計算過程の任意の時点で現在活動的な環境に 呼出しスタック 経由でプログラムのアクセスがある。関数が起動されるたびに、文脈[3]

[3] 訳注:文脈 (context)

と呼ばれる特殊な構造物が内部的につくり出され、文脈のリストの置かれる。関数が評価を終えると、その文脈は呼び出しスタックから取り除かれる。

変数の値を呼び出しスタック経由で得ることは普通動的なスコープと呼ばれる。この際ある変数はその (時間的に) 直近の変数の定義で決定される。R の既定のスコープ規則は辞書式スコープであるが、動的なスコープが本質的な場合がある。

呼び出しスタックへのアクセスは 'sys.' で始まる名前を持つ関数の族を通じて提供される。

`sys.call` 指定された文脈に対する呼び出しを得る。

`sys.frame`
: 指定された文脈に対する評価フレームを得る。

`sys.nframe`
: 全ての活動的な文脈に対する環境フレームを得る。

`sys.function`
: 指定された文脈中で起動された関数を得る。

`sys.parent`
: 現在の関数起動の親を得る。

`sys.calls`
: 全ての活動的な文脈に対する呼び出しを得る。

`sys.frames`
: 全ての活動的な文脈に対する評価フレームを得る。

`sys.parents`
: 全ての活動的な文脈に対する名前ラベルを得る。

`sys.on.exit`
: 指定された文脈が活性化された時に実行される関数を設定する。

`sys.status`
: `sys.frames`, `sys.parents` そして `sys.calls` への呼び出し。

`parent.frame`
: 指定された親文脈に対する評価フレームを得る。

3.5.4 検索パス

評価環境構造に加えて R はどこにも見付からない変数を検索するための環境の検索パスを持つ。これは二つの目的に使われる: 関数のパッケージと付随するユーザーデータである。

検索パスの最初の要素は大局的環境で最後のものは基本パッケージである。必要に応じ読み込まれるかもしれない代理[4] オブジェクトを保持するために `Autoloads` 環境というものが使われる。他の環境は `attach` もしくは `library` を用いてこのパスに含めることができる。

[4] 訳注:代理 (proxy)

4 関数

4.1 関数を書く

R はデータ解析の道具として非常に有用であり得るが、多くのユーザーはすぐに自分自身の関数を書きたくなるであろう。これこそが R の真の長所の一つである。ユーザーはそうした関数をプログラムでき、もしそうしたければシステムレベルの関数を最も適当と判断した関数に変更することができる。

R はまたユーザーが作った関数に対する説明を容易に加えることができる機能を提供する。See section "Writing R documentation" in *Writing R Extensions*.

4.1.1 構文と例

関数を書く構文は次のようになる

`function ( 引数リスト ) 関数本体`

関数宣言の最所の要素はキーワード `function` であり、これは R に関数を作り出したいことを知らせる。

引数リストは形式的な引数をコンマで区切って並べたリストである。形式的引数はシンボル、'*symbol* = *expression*' の文、または特殊形式的引数 '`...`' である。

関数本体 は任意の適正な R 表現式である。一般に、本体は波括弧 ('{' と '}') で囲った表現式のグループである。

普通関数はシンボルに付値されるが、必ずしもそうする必要は無い。`function` の呼び出しの結果は一つの関数になる。もしこれが名前を与えられなければ、それは普通匿名関数と呼ばれる。

次の簡単な関数を考えよう: `echo <- function(x) print(x)`. すると `echo` は単一の引数を取る関数であり、`echo` を呼び出すとその引数を印字する。

4.1.2 引数

関数の形式的引数は、関数が呼び出されたときにその値が補われる変数を定義する。こうした引数の名前は関数本体中で使うことができ、関数が呼び出されたときに補われる値を取る。

引数の既定値を特殊な形式 '*name* = *expression*' で指定できる。この場合、もしユーザーが関数呼び出しの際に値を指定しなければ、この表現が対応するシンボルに与えられる。もし値が必要なら *expression* が関数の評価フレーム中で評価される。

既定値はまた関数 `missing` を用いて指定できる。`missing` がある引数名で呼び出されると、もしユーザーがその引数に値を与えていなければ `TRUE` を返し、さもなければ `FALSE` を返す。`missing` の呼び出しは引数の評価を行わない。

特殊なタイプの引数である '`...`' は与えられた任意の数の引数を含むことができる。これは色々な目的のために使うことができる。任意個数の引数を取る関数を書くのに使うことができる。中間的な関数へのいくつかの引数を吸収し、それから後から呼び出される関数によって取り出されるようにすることができる。

4.2 関数オブジェクト

R では関数は三つの部分から成ると考えることができる。関数本体、形式的引数 そして 環境 である。

関数本体 は関数が呼び出されたとき評価される表現式である。形式的引数 はユーザーが補う値と関数の本体に現れるシンボルもしくは名前との間の対応を与える。最後に 環境 は関数が作り出されたとき活動的だった環境である。この環境は (適切に除去されない限り) 関数が存在する限り関数に連係している。

関数は R の第一級のオブジェクトである。これは基本的にオブジェクトが必要とされるあらゆる場所に登場できることを意味する。特に、関数の引数として渡すことができるし、関数の返値として返すことができる。

4.3 評価

4.3.1 評価環境

関数が呼び出されると新しい評価フレームが作り出される。このフレーム中で形式的引数は Section 4.3.2 [引数照合], page 19 中で与えられた規則に従い補われた値と照合される。関数の本体中の文はそれからこの環境フレーム中で評価される。

評価フレームの親もしくは抱合フレームとは起動された関数に連係する環境フレームである。これは S とは違ったものになる可能性がある。ほとんどの関数は親として `.GlobalEnv` を持つが、いつでもそうなるわけではない。

4.3.2 引数照合

関数の評価で最初に起こることは形式的引数と、実際もしくは補われた引数との照合である。これは三段階の過程で行われる。

1. タグに関する正確な照合。各名前付きの引数に対し、形式的引数中の名前が正確に一致する項目が探される。同じ形式的引数が複数の実際の引数に一致する、もしくはその逆はエラーとなる。
2. タグに関する部分的な照合。各名前付きの引数が残りの形式的引数に対し部分的照合により比較される。もし形式的引数が正確にある形式的引数の最初の部分に一致すれば、両者は一致すると見なされる。複数の部分的一致はエラーになる。もし `f <- function(fumble,fooey) fbody` ならば `f(f = 1, fo = 2)` は二つ目の実際の引数だけが `fooey` に一致するものの不正になることを注意しよう。しかしながら `f(f = 1, fooey =2)` は第二の引数が正確に一致し、部分的照合の対象から外されるため正しい構文になる。もし形式的引数が ‘`...`’ を含めば、部分的照合はそれに先立つ引数に対してだけの適用される。
3. 位置に関する照合。未照合の全ての形式的引数は其の順に 名前無しの 引数と結びつけられる。もし ‘`...`’ 引数があれば、残りの引数を名前の有無にかかわらず引き受ける。

一つでも未照合の引数があればエラーが宣言される。

引数照合は関数 `match.arg`, `match.call` そして `match.fun` により改善される。R が用いる部分的照合アルゴリズムへは `pmatch` 経由でアクセスできる。

4.3.3 引数評価

関数への引数の評価に関して知っておくべき最も重要な点は、値が提供された引数と既定値引数が異なって扱われるということである。関数への値補充引数は呼び出し関数の評価フレームで評価される。既定値引数は関数の評価フレームで評価される。

R での関数起動は 値渡し で行われる。一般に補われた引数は与えられた値と対応する形式的引数名で初期化された変数であるかのように振舞う。補われた引数の値を関数内で変えても呼び出しフレーム中の変数の値は変わらない。

R は一種の関数引数の遅延評価を持つ。引数は必要になるまで評価されない。場合によると引数は決して評価されないことを知ることは重要である。従って、関数への引数を或る種の副作用を起こすために使うことは好ましくないやり方である。**C** のような言語では `foo(x = y)` といった形式を引数 `foo` を値 `y` で起動すると同時に `y` の値を `x` に代入するために使うことは普通であるが、R ではこれはこの引数が必ず評価される保証が無く、従って代入が行われないかも知れないためため使ってはならない。

また引数が評価された際の `foo(x <- y)` の効果は `x` の値を呼び出し環境で変更することであり `foo` の評価環境ででではないことを注意しよう。

関数の内部で引数として使われる実際の表現式にアクセスすることができる。この機構は予約を用いて行われる。予約 はユーザーが利用不可能な R のデータ構造である。しかしながら 予約 がどのように動作するのかを知ることはより良いコードを書く助けになる。ある関数が評価されている際、引数として実際に使われている表現式は予約中に保管され、同時に関数がそこから呼び出された環境へのポインターも保管される。引数が (もし) 評価されるならば、保管された表現式は関数が呼び出された環境で評価される。環境へのポインターだけが使われるためその環境へのあらゆる変更はこの評価の間有効になる。結果の値はそれからまた予約 中の別の場所に保管される。それ以降の評価はこの保管された値を検索してくる (二度目の評価は行われない)。未評価の表現式へのアクセス (内部コードで) もまた可能である。R は非常に柔軟なプログラムであるので、インタープリタ言語中で予約に出くわす可能性がある。しかしユーザーは自分自身のプログラムではそれらを当てにしないほうが良い。

ある関数が呼び出されると、各形式的引数は呼び出しの局所環境中で予約を割り当てられ、その表現式スロットは実際の引数を含み、環境スロットは親環境を含む。呼び出しにおいて形式的引数へ実際の引数が指定されていず、既定表現式が存在すると、それが同様に形式的引数の形式的引数のスロットに割り当てられるが、その環境は局所環境に設定される。

予約 環境中の表現式スロットの内容を評価することにより予約の値スロットを埋める過程は予約の 強制執行 と呼ばれる。

予約はその値が必要になると強制執行される。これは普通内部関数の内側で行われるが、予約はまた予約自身の直接的な評価でも強制執行される。これは既定の表現式が他の引数の値に依存する場合に時おり有用になる。このことは次の例で見ることができ、そこでは孤立した `label` という文がそれが表現式引数に基づくことを明らかにしている。

```
function(x, label = deparse(x)) {
    label
    x <- x + 1
    print(label)
}
```

予約の表現式スロットはそれ自身他の予約を含むことができる。これは未評価の引数が他の関数の引数として引き渡された毎に起こる。予約を強制徴収する際には、その表現式中の他の全ての予約はまた再帰的に強制執行される。

4.3.4 スコープ

スコープもしくはスコープ規則は単にあるシンボルの値を見出すために評価子が用いる一群の規則である。すべての計算機言語はそうした規則を持つ。R ではこうした規則はかなり単順であるが、通常もしくは既定の規則を置き換える機構が存在する。

R は 辞書式スコープ と呼ばれる規則を固守する。これは表現式が作り出されたときに有効であった変数の束縛が表現式中の全ての未拘束のシンボルの値を提供するために使われることを意味する。

スコープのほとんどの興味ある性質は関数の評価に関係するので、此の点に焦点をしぼる。あるシンボルは拘束されているか未拘束かのどちらかである。関数の全ての形式的引数は関数本体中の拘束変数を提供する。関数本体の他の全てのシンボルは局所変数かまたは未拘束変数のどちらかである。局所変数は関数中で定義された変数である。R は変数の公式な定義を持たず、単に必要に応じて使われるため、ある変数が局所的かどうかを決定するのは困難である。局所変数が最初に定義されるべきであり、これは典型的にはそれを付値文の左辺項に置くことでなされる。

評価の過程で未拘束のシンボルが検出されると R はその値を見付けようとする。スコープ規則はこの過程がどのように進行するかを決める。R では関数の環境が最初に検索され、それから先頭環境に至るまで、その親から親へと検索が続く。

先頭環境はシンボルに関する検索が行われる環境の検索リストである。そうして最初に見つかった値が使われる。

この規則集が、関数は他の関数の返り値になり得るという事実と結び付くと、かなり素晴しい、しかし一見すると奇妙な、性質が得られる。

簡単な例をあげる：

```
f <- function(x) {
    y <- 10
    g <- function(x) x + y
    return(g)
}
h <- f()
h(3)
```

かなり興味ある問題は `h` が評価されるとき何が起きるのかということである。これを説明するためにはもう少し用語が必要になる。関数本体では変数は拘束、局所、未拘束の状態を取り得る。拘束変数は関数の形式的引数に一致する変数である。局所変数は関数本体中で生成もしくは定義されたものである。関数本体が評価されるとき、局所または拘束変数の値を決定することには困難は無い。スコープ規則は未拘束変数の値をどのように見出すかを決定する。

`h(3)` が評価される際、その本体が `g` のそれであることが分かる。. その本体中では `x` と `y` は未拘束である。辞書式スコープを持つ言語では `x` は値 3 に、`y` は値 10 に結びつけられ、したがって `h()` は値 13 を返すはずである。これが実際 R で起こることである。

S では異なったスコープ規則が使われており、変数 `y` が作業スペースにあり、必要ならその値が使われるはずなのに `y` が見つからないというエラーが生じる。

S のスコープ規則はまず現在のフレームを探し、それから大局的環境または作業スペースを探すというものである。この規則は `C` 言語で使われているものと大変良く似ている。

4.4 閉包

閉包 とは関数とその各自由変数に対する拘束を提供する環境を一緒に考えたものである。多くの R の関数は環境と結びつけられているために、しばしば閉包と呼ばれる。

5 オブジェクト指向プログラミング

オブジェクト指向プログラミングは最近人気のプログラムスタイルである。その人気の多くは複雑なシステムを書きそして保守することが容易になるという事実からきている。これは幾つかの異なった機構によってなされる。

オブジェクト指向言語の核心はクラスとメソッドの概念である。クラス とはオブジェクトの定義である。典型的にはクラスは幾つかのクラス固有の情報を持つために使われる スロット を持つ。言語中のあるオブジェクトはあるクラスのインスタンスでなければならない。プログラミングはオブジェクトもしくはクラスのインスタンスに基づいて行われる。

計算は メッソド を利用して行われる。メソッドは基本的には関数であり、普通特定のクラスのオブジェクトに対する特定の計算を実行するように特殊化されている。このことが言語をオブジェクト指向と呼ぶ理由である。R では 総称的関数 が適切なメソッドを決定するために使われる。総称的関数はその引数のクラスを決め、この情報を適切なメソッドを選ぶために使用する。

多くのオブジェクト指向言語のもう一つの特長は継承の概念である。多くのプログラミング言語では普通お互いに関連しあった多くのオブジェクトが存在する。もし幾つかの成分が再使用できればプログラミングは飛躍的に単順化できる。

あるクラスが他のクラスを継承すれば、それは普通親クラスの全てのスロットを受け継ぎ、新しいスロットを加えることによりそれを拡張できる。メッソドの実行 (総称的関数経由で) において、もしそのクラスに対するメッソドが存在しなければ親のメソッドが用いられる。

この章ではこの一般的な戦略がどのように R で実現されているかと、現在のデザインにおける幾つかの制限に付いてに議論する。オブジェクトシステムが持つ利点の一つはより大きな一貫性である。これはコンパイラーもしくはインタプリターが行う検査の規則によって実現される。不幸にもオブジェクトシステムが R に移植されている仕組みからこの利点は得られない。ユーザーはオブジェクトシステムを注意して直接使わなければならない。かなり興味深い幾つかの芸当を行うことが可能であるが、これは不明瞭なコードを生みやすく、必ずしも実現されていない移植の詳細に依存する可能性がある。

R におけるオブジェクト指向プログラミングの最大の利用は `print` メッソド、`summary` メソッド、そして `plot` メソッドを通じて得られる。これらのメソッドは総称的な関数呼び出し、例えば `plot`、を利用することを可能にし、これはその引数の型に基づいて手配を行い、提供されたデータに固有のプロット関数を呼び出す。

概念を明瞭にするために、学生に確率に付いて教えるためにデザインされた小さなシステムの移植を考えてみよう。このシステムではオブジェクトは確率関数であり、考慮するメソッドはモーメントを見出すこととプロットである。確率は常に累積分布関数の言葉で表現できるが、しばしば他の方法でも表現できる。例えば存在すれば密度関数、もしくはモーメント母関数である。

(訳注：この項未完成)

5.1 定義

完全なオブジェクト指向システムの代わりに R はクラスシステムとオブジェクトのクラスに基づくメッソド指名[1] の機構を持つ。インタープリタコードに対する指名機構は評価フレームに保管された四つの特殊なオブジェクトに依存する。これらの特殊なオブジェクトは `.Generic`, `.Class`, `.Method` そして`.Group` である。どこか他で議論される内部的な関数と型に対して使われる別個の指名機構がある。

[1] dispatching: 動的タスク指名、人材派遣、作業分配

クラスシステムは class 属性によってを通じて attribute. この属性はクラス名のリストである。クラス "foo" のオブジェクトを作り出すには文字列 ‘"foo"’ を内部に持つクラス属性を単に付加すれば良い。したがって、事実上すべてのものがクラス "foo" のオブジェクトにすることができる。

オブジェクトシステムは二つの指名関数 UseMethod と NextMethod を用いて 総称的関数を利用する。典型的なオブジェクトシステムの利用は総称的関数を呼び出すことで始まる。この関数は普通極めて単順で一行のコードからなる。システム関数 mean は正にそうした関数の例である。

```
> mean
function (x, ...)
UseMethod("mean")
```

mean が呼び出されるときは任意個数の引数を持つことができるが、その最初の引数は特殊で、その最初の引数がどのメソッドを使うかを決めるのに使われる。変数 .Class が x のクラス属性に設定され .Generic は文字列"mean" に設定され、起動すべき正しいメソッドを探索する。mean に対する他の全ての引数のクラス属性は無視される。

x が "foo" と "bar" をこの順で含むクラス属性を持つとしよう。R は最初 mean.foo という名前の関数を探し、もしそれが無ければ次に関数 mean.bar を探す。もしそれも見つからなければ、最後に foo.default を探す。もし最後の探索も不成功ならば R はエラーを宣告する。常に既定のメソッドを用意するのは良い考えである。関数 mean.foo 等 はこの文脈ではメソッドとして参照されることを注意しよう。

NextMethod は指名に対するもう一つの機構を提供する。一つの関数はその中のどこ出でも NextMethod を呼び出せる。どのメソッドが起動されるかの決定は第一に .Class と .Generic の現在値に依存する。これはメソッドが実際は通常の関数でありユーザーがそれを直接呼び出せるためすこし問題無しとはしない。もしユーザーがそうすれば .Generic と .Class に対する値は存在しないことになる。

もしメソッドが直接起動され、それが NextMethod の呼び出しを含めば NextMethod への最初の引数が総称的関数を決定するのに使われる。もしこの引数が提供されないとエラーが生じる。したがってこの引数を常に与えることはよい考えである。

メソッドが直接起動された場合は、このメソッドへの最初の引数のクラス属性が.Class の値として使われる。

メソッドそれ自身は一種の継承を提供するために NextMethod を使用する。普通特定のメソッドはデータを整えるために幾つかの操作を行い、それから NextMethod への呼び出しを通して適当な次のメソッドを呼び出す。

次の簡単な例を考えよう。2 次元のユークリッド空間の点はそのデカルト座標 (x-y) もしくは曲座標 (r-theta) で指定できる。したがって点の位置の情報を保管するためには二つのクラス"xypoint" と "rthetapoint" を定義することが考えられる。‘xy’ 点データ構造の全体は x-成分と y-成分をもつリストになる ‘r − theta’ データ構造の全体は r-成分と theta-成分をもつリストになる。

ここで、双方のタイプのオブジェクトから x-位置を得ることを考えよう。これは総称的関数を用いて簡単に実現できる。総称的関数 xpos を次のように定義しよう。

```
xpos <- function(x, ...)
    UseMethod("xpos")
```

次にメソッドを定義する。

```
xpos.xypoint <- function(x) x$x
xpos.rthetapoint <- function(x) x$r * cos(x$theta)
```

ユーザーは単に関数 xpos を引数としてどちらかの表現を用いて呼び出せば良い。内部の dispatching メソッドがオブジェクトのクラスを見出し、適切なメソッドを呼び出す。

他の表現を加えるのはとても簡単である。新しい総称的関数を書く必要は無く、メソッドだけを書けば良い。ユーザーは既存の他の表現ではなく新しい表現を処理することだけに責任を持てば良いので、これは既存のシステムに付け加えるを簡単にする。

この手法の使用の大部分は異なったオブジェクトに対する印字出力法を提供することにある。約４０のそうしたメソッドがある。

あるオブジェクトのクラス属性は複数の要素を持つことができる。ある総称的関数が呼び出されると、最初の継承は主に NextMethod を通じて処理される。 NextMethod は現在評価中のメソッドを決定し、次のクラスを[2]。

総称的関数は単一の文からなるべきである。それらは常に　foo <- function(x, ...) UseMethod("foo", x) の形式を持つべきである。UseMethod の構文は、適当なメソッドが決定され、それからそのメソッドが、あたかも呼び出しが直接メソッドになされたかのように総称的関数への呼び出しと同じ順序で、同一の引数で起動されるというものである。

正しいメソッドを決定するためには総称的関数への最初の引数のクラス属性が得られ、正しいメソッドを見付けることができなければならない。総称的関数の名前はクラス属性の最初の要素と形式 *generic*.*class* で結合され、それからその名前を持つ関数が探される。もし関数が見付かればそれが使われる。もしそうした関数が見付からなければ、クラス属性の第２要素が使われる等、クラス属性の全ての要素が無くなるまで続く。もしいかなるメソッドもその時点で見付からなければ、メソッド *generic*.*default* が使われる。もし総称的関数への最初の引数がクラス属性を持たなければ *generic*.*default* が使われる。

全てのオブジェクトは class 属性を持つことができる。この属性は任意個数の要素を持つことができる。これらの各々はクラスを定義する文字列である。総称的関数が起動されると、その最初の引数のクラスが検査される。

UseMethod は特別な関数で他の関数呼び出しとは異なった挙動をする。その呼び出しの構文は UseMethod(*generic*, *object*) であり、ここで *generic* は総称的関数の名前、*object* はどのメソッドを選ぶかを決定するオブジェクトである。UseMethod はある関数の内部からだけ呼び出すことができる。S における UseMethod の定義は追加の引数 ‘...’ を持つが、それ以降の計算に何らの効果も持たないようである。 R は UseMethod へ２つ以上の引数があると警告を出し、これらの余分の引数を無視する。

UseMethod は評価モデルを２種類の仕方で変更する。最初に、それが起動されたときに次に呼び出されるメソッド (関数) を決定する。次に、現在の評価環境の下でその関数を起動する。この過程を手短に説明しよう。UseMethod が評価環境を変更する２つ目の動作では、呼び出し関数に制御を戻さない。これは UseMethod の呼び出しに続く如何なる文も実行されるとは限らないということを意味する。

UseMethod が起動されると、総称的関数が UseMethod への呼び出しへの特殊値としてか、もし如何なる値も指定されないと現在の関数の名前として、決定される。dispatch されるオブジェクトは提供される第２引数、または現在の関数への第１引数のどちらかである。引数のクラスが決定され、その最初の要素が適切なメソッドを決定するために総称的関数の名前と結合される。したがって、もし総称的関数が名前 foo を持ち、オブジェクトのクラスが "bar" ならば、R は名前 foo.bar を持つメソッドを探す。もしそうしたメソッドが存在しなければ、先に説明された継承メカニズムが適当なメソッドを特定するために使われる。

メソッドが一旦決定されると R はそれを特殊な方法で起動する。新しい評価環境をつくり出す代わりに R は現在の関数呼び出し (総称的関数への呼び出し) の環境を使う。UseMethod の呼び出し

[2] 訳注：以下欠

に先立って行われた如何なる代入・評価も無効になる。総称的関数への呼び出しに使われた引数は選ばれたメソッドの形式的引数と再照合される。

メソッドが起動されると、それは総称的関数への呼び出しにおけるのと同じ個数・名前を持つ引数でよびだされる。それらはメソッドの引数と R の標準的照合手順にしたがって照合される。しかしながら、オブジェクト、つまり最初の引数は既に評価済みである。

`UseMethod` の呼び出しはある特殊なオブジェクトを評価フレームに置くという効果を持つ。それは `.Class`、`.Generic` そして `.Method` である。これらの特殊オブジェクトは R がメソッド dispatch と継承を処理する際に使われる。`.Class` はオブジェクトのクラスであり、`.Generic` は総称的関数の名前、そして`.Method` は現在起動中のメソッドの名前である。もしメソッドが内部インタフェイスの一つにより起動されると `.Group` という名前のオブジェクトも存在するかも知れない。これは節 ⟨undefined⟩ [Group methods], page ⟨undefined⟩ で説明されるであろう。 `UseMethod` への最初の呼び出しの後に、オブジェクト自身でなく、これらの変数が引き続くメソッドの選択を制御する。

メソッドの本体が標準的なやりかたでそれから評価される。特に本体中の変数の検索はメソッドに対する規則に従う。従って、メソッドが関連する環境を持てば、それが使われる。実際には総称的関数への呼び出しはメソッドへの呼び出しに置き換えられている。総称的関数のフレーム中での任意の局所的付値はメソッドへの呼び出しに引き継がれるであろう。この 特徴 の使用は勧められない。制御は決して総称的関数に返されないことを認識することが肝要で、従って `UseMethod` への呼び出しに続く如何なる表現式も決して実行されることは無いであろう。

`UseMethod` への呼び出しに先だって評価された総称的関数への任意の引数は評価済みとされるであろう。

総称的関数への呼び出し中の引数は標準的な引数照合機構を用いてメソッドへの引数と再照合されるであろう。最初の引数、つまりオブジェクト、は評価済みとされるであろう。

`UseMethod` への最初の引数が与えられないと、それは現在の関数の名前とされる。もし 2 つの引数が `UseMethod` に与えられると、2 つ目は手配されるべきオブジェクトであると仮定される。それが評価されると必要なメソッドが決定できる。この場合総称的関数への呼び出し中の最初の引数は評価されず、無視される。呼び出し中の他の引数を、それらがあたかも総称的関数への呼び出し中にあったかのように変換する方法は無い。これは、次のメソッドへの呼び出し中の引数を変更できる `NextMethod` と対照的である。

`NextMethod` は単純な継承機構を提供するために使われる。

`NextMethod` の呼び出しへの結果として起動されるメソッドは、それがあたかも先立つメソッドから起動されているかのように振舞う。継承メソッドへの引数は現在のメソッドへの呼び出しと同じ順序であり、同じ名前を持つ。これはそれらが総称的関数の呼び出しに対するのと同じであることを意味する。しかしながら、引数に対する表現は現在のメソッドの対応する形式的引数の名前である。このように引数は NextMethod が起動された時点でのそれらの値に対応する値を持つであろう。

未評価の引数は未評価のままである。欠損値は欠損したままである。

`NextMethod` への呼び出しの構文は `NextMethod(総称的関数, オブジェクト, ...)` である。もし 総称的関数 が提供されていないと `.Generic` の値が使われる。もし `object` が提供されないと、現在のメソッドの呼び出し中の最初の引数が使われる。‘`...`’ 中の値は次のメソッドの引数を変更するために使われる。

次のメソッドの選択が `.Generic` と `.Class` の現在値に依存し、オブジェクトには依存しないことを理解することが重要である。したがって `NextMethod` の呼び出し中のオブジェクトを変更することは、次のメソッドによって受け継がれる引数に影響するが、次のメソッドの選択には影響しない。

メソッドは直接呼び出すことができる。もしそうすると `.Method` は存在しないことになる。この場合は `NextMethod` の `.Class` の値が現在の関数の第一引数であるオブジェクトのクラス属性とさ

れる。`.Method` の値は現在の関数の名前である。既定値のこうした選択はメソッドの動作が、それが直接呼ばれたか、またはある総称的関数への呼び出し経由で行われたかによって変わらないことを保証する。

一つの論点は `NextMethod` に対する '`...`' 引数の挙動である。白本はこの挙動を次のように説明している：

- 名前付きの引数は現在の呼び出し中の対応する引数を置き換える。名前無しの引数は引数リストの先頭に行く。

私が行いたいのは次の通りある：

-最初に Nextmethod に対する引数照合を行う；-オブジェクトもしくは総称的関数がうまく置き換わるか-最初に名前付きのリストの要素が引数 (名前付きもしくは名前無し) にマッチしたら、リストの値は引数値を置き換える。-最初の名前無しのリスト要素

照合のための値：クラス：最初に .Class から、次にメソッドへの最初の引数から、そして最後に NextMethod への呼び出し中で指定されたオブジェクトから取る。

総称的関数：最初に .Generic から取る。もし何もなければ次にメソッドへの最初の引数から、そしてそれも無ければ NextMethod への呼び出しから取る。

メソッド：これは単に現在の関数名であるべきである。

5.2 グループメソッド

組込関数の幾つかのタイプに対し R は演算子に対する指名機構を提供する。これは `==` や `<` といった演算子が特殊なクラスのメンバーに対し変更された挙動を持ち得ることを意味する。関数と演算子は三つのカテゴリーにグループ化されており、これらのグループそれぞれに対しグループメソッドを書くことができる。現在のところグループを付け加える機能は無い。一つのグループ内の関数全てに特有のメソッドを書くことが可能である。

次の表は異なったグループに対する関数のリストである。

'`Math`'	abs, acos, acosh, asin, asinh, atan, atanh, ceiling, cos, cosh, cumsum, exp, floor, gamma, lgamma, log, log10, round, signif, sin, sinh, tan, tanh, trunc
'`Summary`'	all, any, max, min, prod, range, sum
'`Ops`'	+, -, *, /, ^, < , >, <=, >=, !=, ==, %%, %\%, &, |, !

Ops グループ中の演算子に対しては、もし二つの被演算項が対として単一のメソッドを示唆するなら特殊なメソッドが起動される。特に、二つの被演算項が同じメソッドに対応する場合や、一つの被演算項が他の被演算項のそれに優先するメソッドに対応する場合が該当する。もし両者が単一のメソッドを示唆しないならば既定のメソッドが使われる。もし他の被演算項が対応するメソッドを持たなければグループメソッドとクラスメソッドのどちらも優位にはならない。クラスメソッドがグループメソッドよりも優先する。

もし グループ が Ops ならば、特殊変数 `.Method` は二つの要素を持つ文字列ベクトルである。`.Method` の要素は、もし対応する引数がメソッドを決定するのに使われたクラスのメンバーならば、メソッドの名前にセットされる。さもなければ `.Method` の対応する要素は長さゼロの文字列`""` にセットされる。

5.3 メソッドを書く

ユーザーは容易に自分自身のメソッドや総称的関数を書くことができる。総称的関数とは単に UseMethod への呼び出しを伴う関数である。メソッドとは単にメソッド指名を経由して起動された関数のことである。これは UseMethod もしくは NextMethod への呼び出しの結果であり得る。

メソッドは直接に呼び出せることを覚えておくことは価値がある。これは UseMethod を予め使うこと無く入力でき、したがって特殊変数 .Generic, .Class そして .Method が特定の値を与えられないことを意味する。この場合上に説明された既定の規則がこれらを決定するために使われるであろう。

最も普通の総称的関数の使用は、あるモデル当てはめ過程の出力である統計的オブジェクトに対する print そして summary メソッドを提供することである。これを行うために、各モデルはその出力にあるクラス属性を付与し、それからその出力を受け取り、その適正な可読版を提供する特殊なメソッドを提供する。そうするとユーザーは print もしくは summary が任意の解析結果に対する適正な出力を提供するであろうことだけを覚えていればよい。

6 言語自体のプログラミング

R はサブルーチンが他のサブルーチンを修正したり構成したりでき、その結果を言語自体を構成する一部分として評価できる能力を持つような一群の言語のクラスに属する。これは Lisp や Scheme そして “関数プログラミング” 言語と呼ばれる他の言語に類似しており、FORTRN や ALGOL 風言語とは一線を画す。Lisp 風言語族は、プログラムとデータの間に一切の区別が無い “全てがリスト” というパラダイムにより、この特徴を最左翼に位置する。

R は、少なくとも数学公式や C 風の制御構造になれた者に取って、Lisp よりはよりとっつき易いプログラミングへのインタフェイスを提供するが、そのエンジンは実際はかなり Lisp 風である。R は構文解析された表現式や関数への直接のアクセスを許し、それらを変更した後実行したり、さらにはまったく新しい関数を一からつくり出すことを許す。

表現式の導関数を計算したり、係数ベクトルから多項式を生成したりといった、この機能の幾つかの標準的応用がある。しかしながら、また R のインタープリタ部分の動作にとりより基本的な使用法が存在する。これらの幾つかは、幾つかのモデリング・プロットルーチン中でつくり出される `model.frame` への呼び出し (決して手際良いとはいえないかも知れないが) に於けるように、関数を他の関数の一部分として再使用するために本質的である。別の使用法は単に有用な機能へのエレガントなインタフェイスを与えることである。一例として `curve` 関数がある。これは `sin(x)` といった表現式で与えられた関数のグラフを書いたり、数学的表現式のプロットに対する機能を与える。

この章では、言語自体に関する計算に対して利用できる一揃いの機能への紹介を与える。

6.1 言語オブジェクトの直接的操作

変更に対して利用できる三種類の言語オブジェクト、呼び出し (call)、表現式 (expression) そして関数 (function)、がある。ここでは呼び出しオブジェクトにのみ注目する。これらはしばしば “未評価表現式” とも呼ばれるが、この表現は少々混乱を招く。呼び出しオブジェクトを得る最も直接的な方法は `quote` を表現式引数で用いることである、つまり、

```
> e1 <- quote(2 + 2)
> e2 <- quote(plot(x, y))
```

引数は評価されず、結果は単に構文解析された引数である。オブジェクト `e1` と `e2` は後で `eval` を用いるか、または単にデータとして扱うことで評価できる。恐らくなぜ `e2` オブジェクトがモード `"call"` を持つかは、それがある引数を伴った `plot` 関数への呼び出しを含むので、すぐさま理解されるであろう。しかしながら `e1` は、次の例で明らかに示されるように、実際には二つの引数を持つ二項演算子 `+` への呼び出しと全く同じ構造をもつ。

呼び出しオブジェクトの成分はリスト-風の構文を用いてアクセス可能であり、実際 `as.list` と `as.call` を用いてリストへ、そしてリストから変換できる。

```
> e2[[1]]
plot
> e2[[2]]
x
> e2[[3]]
y
```

キーワード引数照合が使われると、キーワードはリストタグとして使うことができる：

```
> e3 <- quote(plot(x = age, y = weight))
> e3$x
age
```

```
> e3$y
weight
```

先の例では、呼び出しオブジェクトの全ての成分はモード "name" を持つ。これは呼び出し中の識別子に対しては真であるが、呼び出しの成分はまた定数であっても良く、それは如何なる型であっても良い。しかしながら、呼び出しが成功裡に評価されるためには最初の成分は関数、もしくは副表現式に対応する他の呼び出しオブジェクト、であるほうが好ましい。モード名オブジェクトは文字列から as.name を用いて構成でき、したがって e2 オブジェクトを次のように修正できる

```
> e2[[1]] <- as.name("+")
> e2
x + y
```

副表現式がそれ自身呼び出しである単なる成分であるという事実を解説するために、次の例を考えよう

```
> e1[[2]] <- e2
> e1
x + y + 2
```

一つの注意すべき点は R の逆構文解析器は表現式の括弧を補わないということである。たとえば次のようにすると

```
> e1 <- quote(4 - 2)
> e1[[3]] <- quote(2 - 2) # replacing the ‘2’
> e1
4 - 2 - 2
```

e1 は実際には左連結規則が示唆するように $4 - (2 - 2)$ ではなく $(4 - 2) - 2$ となる。これは次のように呼び出しを評価すると自ずから明らかになる

```
> eval(e1)
[1] 4
```

入力中の全てのグルーピング用括弧は構文解析された表現式中に保存される。それらは単一の引数を持つ関数呼び出しとして表現され、したがって 4 - (2 - 2) は前置表記で "-"(4, "(" ("-"(2, 2))) となる。

これは少々不幸なことであるが、ユーザーの入力を保存し、それを最小の形式で保管し、逆構文解析された表現式を構文解析すると前と同じ表現になるような、構文・逆構文解析器を書くことは易しくはない。

次の例が示すように、困ったことに R の構文解析器も逆構文解析器も完全には可逆でない。

```
> deparse(quote(c(1, 2)))
[1] "c(1, 2)"
> deparse(1:2)
[1] "c(1, 2)"
> quote("-"(2, 2))
2 - 2
> quote(2 - 2)
2 - 2
```

しかしながら, 構文・逆構文解析器は基本的な表現式に対してはかなり優れた仕事をする。

... 流れ制御構造物の内部保管... Splus との非互換性を注意する...

6.2 代入

実際は先の節に於けるようなある表現式の内容を修正したくなることはそんなに多くはない。より多くは、逆構文解析し、そしてそれを例えばプロットのラベルに使うために単に表現式を得たいことの方が多い。これの一つの例は plot.default の最初の部分で見られる:

```
xlabel <- if (!missing(x))
    deparse(substitute(x))
```

これは plot の x 引数として与えられた変数または表現式を後で x-軸のラベルとして使えるようにする。

これを実行するために使われる関数は substitute で、表現式 x を引数に取り、形式的引数 x を通じて引き渡された表現式を代入する。これがうまく働くためには x はその値をつくり出した表現式に関する情報を備えている必要があることを注意しよう。これは R の遅延評価機構 (see ⟨undefined⟩ [予約オブジェクト], page ⟨undefined⟩) と関連している。形式的引数は実際は二つのスロットを持つ予約オブジェクトであり、一つはそれを定義する表現式用であり、もう一つはその表現式の値用である。substitute は予約変数を認識し、その表現式スロットの値を代入する。もし substitute がある関数の内部で起動されると、その関数の局所引数は同様に代入の対象となる。

substitute への引数は単純な識別子である必要は無く、複数の変数を含む表現式であって良く、代入はこれらの各々に対して起こる。また substitute は追加の引数を持ち、これは変数がその中で検索される環境もしくはリストであって良い。

```
> substitute(a + b, list(a = 1, b = quote(x)))
1 + x
```

x を代入するためには引用化が必要なことを注意しよう。この種の構成法は、次の例が示すように、グラフ中に数学的表現を置くための機能ととの関連で便利となる。

```
> plot(0)
> for (i in 1:4)
+   text(1, 0.2 * i,
+         substitute(x[ix] == y, list(ix = i, y = pnorm(i))))
```

代入は純粋に語彙的であることを理解することが重要である；結果の呼び出しオブジェクトが評価されたとき意味をなすかどうかは検査されない。substitute(x <- x + 1, list(x = 2)) は無邪気に 2 <- 2 + 1 を返す。しかしながら R の一部分は何が意味を持つか・持たないかに付いて独自の規則を持っており、そうした不整合な表現式を実際に使ってしまうかも知れない。たとえば "グラフ中の数式" 機能は '{}>=40*" years"' といった、構文的には正確だが評価すると無意味になる構成をしばしば含む。

代入はその最初の引数を評価しない。これは変数中に含まれるオブジェクトに対し代入をどのようにして行うかというパズルをもたらす。解決法は次の例のように substitute をもう一度使うことである

```
> expr <- quote(x + y)
> substitute(substitute(e, list(x = 3)), list(e = expr))
substitute(x + y, list(x = 3))
> eval(substitute(substitute(e, list(x = 3)), list(e = expr)))
3 + y
```

代入の正確な規則は次のとうりである: 構文解析木中の各シンボルは最初第二引数に対して照合される。第二引数はタグ付きのリストまたは環境フレームであって良い。もしそれが単純な局所オブジェクトならば, 大局的環境に対してマッチする場合を除き、その値が挿入される。もしそれが予約 (普通関数引数) ならば予約表現式が代入される。もしシンボルがマッチしなければ、それはそのまま

にされる。トップレベルにおける代入に対する特殊な例外は正直なところ奇妙である。これは S から引き継がれており、根本的理由は恐らく、変数がそのレベルで拘束されるべき制御が存在せず、したがって代入を単に quote として振舞わせるのが好ましい、ということであろう。

予約代入の規則は、もし局所変数が substitute が使われる前に修正されると、S のそれと少々異なる。R はそうした時変数の新しい値を使うが、一方 S は、それが定数でない限り、無条件に引数の表現式を使う。これは S では f((1)) が f(1) と非常に異なるという奇妙な結論をもたらす。R の規則はより明晰であるが、しかしながら遅延評価との関連で驚くような結果をもたらす。次の例を考えよう

```
logplot <- function(y, ylab = deparse(substitute(y))) {
    y <- log(y)
    plot(y, ylab = ylab)
}
```

これは単純明快であるように思われるが、y ラベルが醜い c(...) という表現になっていることを気づくであろう。これは遅延評価が ylab が変更された後で ylab 表現式を評価することによる。解決策は ylab を最初に評価するように強制することである、つまり

```
logplot <- function(y, ylab = deparse(substitute(y))) {
    ylab
    y <- log(y)
    plot(y, ylab = ylab)
}
```

この代入では eval(ylab) を使うべきではないことを注意しよう。もし ylab が言語もしくは表現式オブジェクトならば、これはオブジェクトもまた評価してしまうが、これはもし quote(log[e](y)) といった数学式が引き渡されたときは決して望ましいことではないだろう。

6.3 評価に関する追加

eval 関数はこの章の冒頭に呼び出しオブジェクトを評価する手段として紹介された。しかしながら、これが全てではない。同様に評価が行われる環境を指定することができる。この環境の既定値は eval がそこから呼び出された評価フレームであるが、頻繁に他のものを設定する必要が起きる。

関係する評価フレームが現在のフレームの親のそれであることが良くある (cf. ???)。特に、評価されるべきオブジェクトが関数の引数の substitute 操作の結果であるとき、それが呼び出し側にとってだけ意味を持つ変数を含むであろう (呼び出し側の変数が呼び出された側の lexical スコープにあることを期待する如何なる理由も無いことを注意しよう)。親フレーム中で評価することは頻繁に起こるので eval(expr, sys.frame(sys.parent())) に対する簡略形として eval.parent 関数が存在する。

しばしば起きるもう一つのケースはリストもしくはデータフレーム中での評価である。例えば、これは data 引数が与えられたときの model.frame 関数との関連で起きる。一般に、モデル公式の項は data 中で評価される必要があるが、ときとしてそれらはまた model.frame の呼び出し側の中の項目への参照を含むことがある。したがってこのためにはリスト中の表現式を評価するだけでなく、変数がリスト中に無い場合に探索を続行する閉包を指定する必要がある。

```
eval(expr, data, sys.frame(sys.parent()))
```

与えられた環境丶での評価は実際にはその環境を変更する可能性があることを注意しよう。これは以下のような付値演算子を含む場合に明白となる

```
eval(quote(total <- 0), environment(robert$balance)) # rob Rob
```

これはまたリスト中で評価する際も真であるが、この際は実際はコピーを扱っているため元のリストは変わることは無い。

6.4 表現式オブジェクトの評価

モード "expression" のオブジェクトは ⟨undefined⟩ [Expression objects], page ⟨undefined⟩ で定義されている。これは呼び出しオブジェクトのリストと非常に良く似ている。

```
> ex <- expression(2 + 2, 3 + 4)
> ex[[1]]
2 + 2
> ex[[2]]
3 + 4
> eval(ex)
[1] 7
```

表現式オブジェクトの評価は各呼び出しを順に評価するが、最終的な値は最後の呼び出しのそれである。この点で複合言語オブジェクト quote({2 + 2; 3 + 4}) とほとんど同じ挙動を持つ。しかしながら微妙な相違点がある：呼び出しオブジェクトは構文解析木中では副表現式と判別不能である。これはそれが副表現式と同じ仕方で自動的に評価されることを意味する。表現式オブジェクトは評価の間も認識可能であり、ある意味でその引用化された状態を保つ。評価器は表現式オブジェクトを再帰的には評価せず、そうなるのはそれが上のように eval 関数に直接引き渡されたときだけである。この違いは次の例で見られる：

```
> eval(substitute(mode(x), list(x = quote(2 + 2))))
[1] "numeric"
> eval(substitute(mode(x), list(x = expression(2 + 2))))
[1] "expression"
```

逆構文解析器は表現式オブジェクトを、それを生み出した呼び出しにより表現する。これはそれが数値ベクトルや特別な外部表現を持たない幾つかの他のオブジェクトを扱うのと似たやり方である。しかしながら、これは次のようなちょっとした混乱を招く。

```
> e <- quote(expression(2 + 2))
> e
expression(2 + 2)
> mode(e)
[1] "call"
> ee <- expression(2 + 2)
> ee
expression(2 + 2)
> mode(ee)
[1] "expression"
```

つまり e と ee は出力されると同じに見えるが、一方は表現式オブジェクトを生成した呼び出しであり、他方はオブジェクト自身である。

6.5 関数呼び出しの操作

次の単にそれ自身の呼び出しを返す関数の例で見られるように、sys.call の結果を眺めることにより、ある関数がどのようにして呼び出されたかを見付けることが可能である。

```
> f <- function(x, y, ...) sys.call()
> f(y = 1, 2, z = 3, 4)
f(y = 1, 2, z = 3, 4)
```

しかしながら、これは呼び出しを解釈するために関数が引数の照合結果を記録しておかなければならないという意味で、デバッグの目的を除いては実際は有用でない。例えば、二番目の実際の引数が最初の形式的なそれ (上の例では x) にマッチされたことがわかる必要がある。

より頻繁には、全ての実際の引数が対応する公式に拘束された呼び出しを必要とする。このためには関数 match.call が用いられる。次は先の例の変形であり、引数照合済みのそれ自身の呼び出しを返す関数である。

```
> f <- function(x, y, ...) match.call()
> f(y = 1, 2, z = 3, 4)
f(x = 2, y = 1, z = 3, 4)
```

二番目の引数がここでは x にマッチされ結果の対応する位置に現われてることを注意しよう。

このテクニックの第一の使用法は、幾つかの引数が除去されたり追加されたりしたかもしれない、同じ引数を持つ他の関数を呼び出すことである。典型的な応用が関数 lm の先頭部分にある。

```
mf <- cl <- match.call()
mf$singular.ok <- mf$model <- mf$method <- NULL
mf$x <- mf$y <- mf$qr <- mf$contrasts <- NULL
mf$drop.unused.levels <- TRUE
mf[[1]] <- as.name("model.frame")
mf <- eval(mf, sys.frame(sys.parent()))
```

結果の呼び出しはその親フレームで評価され、そこでは含まれる表現式が意味を持つことが確実であることを注意しよう。呼び出しはリストオブジェクトとして扱うことができ、リストの最初の要素は関数名、残りの要素は対応する形式名がタグになっている実際の引数表現式である。このように、不要な引数を取り去るテクニックは、第 2・3 行に見られるように、NULL を代入することであり、引数を付け加えるには、第 4 行のように、タグ付きのリスト代入 (ここでは drop.unused.levels = TRUE を引き渡す) を行う。呼び出される関数の名前を変更するには、この場合のように as.name("model.frame") 構成を使うか、または quote(model.frame) を用いて、値が名前であることを保証して、リストの先頭要素に代入する。

関数 match.call は、もし FALSE に設定されると全ての '...' 引数をタグ '...' を持つ単一の引数に一括化するスイッチである引数 expand.dots を持つ。

```
> f <- function(x, y, ...) match.call(expand.dots = FALSE)
> f(y = 1, 2, z = 3, 4)
f(x = 2, y = 1, ... = list(z = 3, 4))
```

'...' 引数はリスト (正確には対リスト) であり、S に於けるような list への呼び出しでは無い。

```
> e1 <- f(y = 1, 2, z = 3, 4)$...
> e1
$z
[1] 3

[[2]]
[1] 4
```

この形の match.call を使う理由は単に、任意の '...' 引数を取り除き、それを知らないかもしれない関数に引き渡さないようにするためである。次の例は plot.formula を書き換えた例である:

```
m <- match.call(expand.dots = FALSE)
m$... <- NULL
m[[1]] <- "model.frame"
```

より精妙な応用が update.default にあり、そこでは当初の呼び出し中にある引数に一揃いのオプション引数を追加、変更、もしくは消去が可能である。

```
extras <- match.call(expand.dots = FALSE)$...
if (length(extras) > 0) {
    existing <- !is.na(match(names(extras), names(call)))
    for (a in names(extras)[existing]) call[[a]] <- extras[[a]]
    if (any(!existing)) {
        call <- c(as.list(call), extras[!existing])
        call <- as.call(call)
    }
}
```

extras[[a]] == NULL である場合に存在する引数を個別に変更する場合は用心がいることを注意しよう。例示されたように強制変換をしない限り呼び出しオブジェクトに対する連結操作は失敗する。これは恐らくバグであろう。

もう二つの関数が関数呼び出し構成用に存在する、つまり call と do.call である。

関数 call を使って関数名と引数リストから呼び出しオブジェクトを作ることができる。

```
> x <- 10.5
> call("round", x)
round(10.5)
```

このようにシンボルではなく x の値が呼び出しに挿入されており、round(x) とは全く異なったものである。この形式は滅多に使われないが、関数名が文字変数として得られる場合にはたまに役に立つ。

関数 do.call は似たようなものであるが、呼び出しを即座に評価し、全ての引数を含むモード "list" のオブジェクトを引数に取る。これの自然な用法は cbind のような関数をリストやデータフレームのすべての要素に適用したいときである。

```
is.na.data.frame <- function (x) {
    y <- do.call("cbind", lapply(x, "is.na"))
    rownames(y) <- row.names(x)
    y
}
```

他の使用法は do.call("f", list(...)) といった構成の変種を含む。しかしながら、これは実際の関数呼び出し以前の引数の評価を含み、遅延評価の側面や関数自体への代入を無効にするかも知れないことに留意する必要がある。同様の注意が call 関数にも当てはまる。

6.6 関数の操作

しばしば関数や閉包の成分を操作することが可能であれば有用である。R はこの目的のために一組のインタフェイスを提供する。

body
: 関数の本体である表現式を返す。

formals
: 関数への形式的変数のリストを返す。これは pairlist である。

environment
: 関数に付随する環境を返す。

body<-
: これは関数本体に与えられた表現式をセットする。

formals<-
: 形式的引数に与えられたリストをセットする。

環境<-　　関数の環境を指定された環境にセットする。

また関数の環境中の異なった変数の結合を変更することができ、`evalq(x<- 5, environment(f))` といったコードを使う。

更に `as.list` を用い関数リストに変換することができる。結果は形式的変数リストと関数本体を連結したものになる。逆に、そうしたリストは `as.function` を用いて関数に変換できる。この機能は主に S との互換性のために用意されている。`as.list` が使用されたとき環境の情報が失われることと、一方で `as.function` は環境をセットすることを許す引数を持つことを注意しよう。

7 システムと外部言語とのインタフェイス

7.1 オペレーティングシステムへのアクセス

オペレーティングシステムへのアクセスは R 関数 `system` を用いてなされる。詳細はプラットフォームにより異なり (オンラインヘルプを見よ)、間違いなく仮定できることは、最初の引数は実行 (必ずしもシェルによってではない) のために引き渡される文字列 `command` であり、第 2 引数はもしそれが真なら命令の出力を R の文字列に集める be `internal` である。

関数 `system.time` が命令実行の所用時間を知るために使える (しかしながら利用できる情報は Unix タイプのプラットフォームに限られるかもしれない)。

7.2 外部言語へのインタフェイス

コンパイルされたコードを用いて R に機能を加える詳細に付いては section "システムと外部言語とのインタフェイス" in *Writing R Extensions* を参照のこと。

関数 `.C` と `.Fortran` は、構築時もしくは `dyn.load` を使って R にリンクされたコンパイル済みコードへの標準的なインタフェイスを提供する。これらは主にコンパイルされた **C** と FORTRAN のコードにそれぞれ対応するが、`.C` 関数は C++ のような C 風のインタフェイスを生成できる他の言語に対しても使うことができる。

関数 `.Call` と `.External` はコンパイルされたコード (主に **C** コード) を使って R オブジェクトを操作するためのインタフェイスを提供する。

7.3 .Internal と .Primitive

`.Internal` と `.Primitive` インタフェイスは構築時に R にコンパイルされて組み込まれた **C** コードを呼び出すために使われる See section ".Internal と .Primitive" in *Writing R Extensions*.

8 例外処理

R の例外処理機能は二つのメカニズムにより提供される。T `stop` や `warning` といった関数は直接に呼び出すことができ、また `"warn"` のようなオプションは問題の処理を制御するために使うことができる。

8.1 stop

`stop` の呼び出しは現在の表現式の評価を停止し、メッセージ引数を出力し、実行を最先頭に戻す。

8.2 warning

関数 `warning` は単一の文字列引数を取る。`warning` の呼び出しの挙動はオプション `"warn"` の値に依存する。もし `"warn"` が負値なら警告は無視される。もしそれが零なら警告は蓄積され最上位関数が完了したときに出力される。もしそれが 1 ならば警告はそのたびに出力され、もし 2 (またはそれ以上) ならば警告はエラーに変わる。

もし `"warn"` が零 (既定値) ならば、最上位変数 `last.warning` が作り出され、`warning` の呼び出しに伴う各メッセージはこのベクトルに逐次蓄積される。もし警告の数が 10 未満ならば関数の評価が終了した際に出力される。もし １０ 以上の警告があれば幾つ警告があったかが出力される。どちらの場合も `last.warning` はメッセージのベクトルを保持している。

8.3 on.exit

一つの関数はその本体のどこにでも `on.exit` の呼び出しを含むことができる。`on.exit` の呼び出しの効果は関数終了時に実行されるように本体の値を保管することである。これにより、関数はシステムパラメータを変更し、関数終了時にそれらが適当な値にリセットされることを保証することができる。`on.exit` は関数が直接に終了しても、また警告の結果として終了しても実行されることが保証されている。

`on.exit` コードの評価に於けるエラーは、`on.exit` コードのそれ以上の実行無しに、直ちにトップレベルへのジャンプを惹き起こす。

`on.exit` は関数が終了した際に評価される表現式である単一の引数を取る。

8.4 restart

`restart` の呼び出しは、もしその関数 (もしくはそれが呼び出している関数の一つ) の評価中にエラーが生じれば、実効的に関数を可能な復帰点に位置させる。

`restart` は論理値変数である単一の引数を取る。もし論理値の値が `TRUE` ならばジャンプ点が設定される。もしその値が `FALSE` ならばジャンプ点は除去される。

ジャンプが実行されるとジャンプ点は取り除かれる。

もしエラーが生じ、一つもしくは複数のジャンプ点が有効なら制御はジャンプ点を設定した一番奥の関数に戻される。実行は選択された本体の最初の実行文から始まる。引き続く評価に対する環境は、ジャンプを誘発したエラーが発行された時に効果を持っていた環境である。

8.5 エラー時オプション

R がエラーや警告を処理する仕方を制御するのに使える幾つかの options 変数がある。それらは次の表にまとめられている。

'warn' 警告の出力を制御する。

'warning.expression'
: 警告が発生したときに評価される表現式を設定する。このオプションが設定されると通常の警告の出力は抑制される。

'error' エラーが発生したときに評価される表現式を設定する。通常のエラーメッセージと警告メッセージがこの表現式の評価の前に出力される。

options("error") によって導入された表現式は on.exit への呼び出しが行われる前に評価される。

エラー信号が発生した際に R を終了するために options(error = expression(q("yes"))) を使うことができる。この場合にエラーが発生すると R とその環境はセーブされる。

9 デバッグ

コードのデバッグは常に少々芸術的側面を持つ。R はユーザーがそのコード中の問題を見付けることを手助けする幾つかの道具を提供する。これらの道具はコードの特定の箇所で実行を停止させ、計算の現在の状態を吟味することができる。

ほとんどのデバッグは browser もしくは debug への呼び出し経由で実行される。この関数はともにある内部的な機構に依存しており、両者ともにある特殊なプロンプトをユーザーに提示する。このプロンプトに対してはあらゆる命令が入力できる。この命令の評価環境は現在活動的な環境である。これにより如何なる変数等の現在の状況を調べることができる。

R が異なって解釈する四つの命令がある。それらは、

'⟨RET⟩' 関数がデバッグされたら次の実行文へ移る。ブラウザーが起動されたら実行を継続する。

'c'
'cont' 実行を継続する。

'n' 関数の次の実行文を実行。これはブラウザーからも使える。

'Q' 実行を停止し、即座にトップレベルにジャンプする。

もし上に羅列したのものの一つと同じ名前の局所変数があれば、その値は get を使って得ることができる。引用符で囲まれた名前を用いた get は現在の環境での値を取り出すであろう。

デバッガーでアクセスできるのは解釈実行される表現式だけである。関数が (**C** のような) 他言語コードを呼び出を行うならば、その言語中の実行文への如何なるアクセスも不可能である。実行は R により評価中の次の実行文で停止するであろう。

9.1 browser

関数 browser の呼び出しは R をその時点で停止させ、ユーザーに特殊なプロンプトを提示する。browser への引数は無視される。

```
> foo <- function(s) {
+ c <- 3
+ browser()
+ }
> foo(4)
Called from: foo(4)
Browse[1]> s
[1] 4
Browse[1]> get("c")
[1] 3
Browse[1]>
```

9.2 debug/undebug

デバッガは命令 debug(*fun*) を用いて如何なる関数に対しても起動できる。そうすると、この関数が評価されるたびにデバッガが起動される。デバッガは関数本体中の実行文の評価を制御することを可能にする。各実行文が実行される前に、実行文が出力され、特別なプロンプトが提示される。あらゆる命令が実行できるが、上の表中の命令は特殊な意味を持つ。

デバッグは undebug を関数とともに呼び出すことにより終了される。

```
> debug(mean.default)
> mean(1:10)
debugging in: mean.default(1:10)
debug: {
    if (na.rm)
        x <- x[!is.na(x)]
    trim <- trim[1]
    n <- length(c(x, recursive = TRUE))
    if (trim > 0) {
        if (trim >= 0.5)
            return(median(x, na.rm = FALSE))
        lo <- floor(n * trim) + 1
        hi <- n + 1 - lo
        x <- sort(x, partial = unique(c(lo, hi)))[lo:hi]
        n <- hi - lo + 1
    }
    sum(x)/n
}
Browse[1]>
debug: if (na.rm) x <- x[!is.na(x)]
Browse[1]>
debug: trim <- trim[1]
Browse[1]>
debug: n <- length(c(x, recursive = TRUE))
Browse[1]> c
exiting from: mean.default(1:10)
[1] 5.5
```

9.3 trace/untrace

R の挙動を監視するもう一つの方法は trace 機能を使うことである。trace は追跡したい関数名を表す単一の引数とともに呼び出される。この名前は引用符で囲む必要は無いが、或る種の関数に対しては構文エラーを避けるために引用符で囲む必要があるかも知れない。

trace がある関数に対して起動されると、関数が評価されるたびにその呼び出しが出力される。この機能は関数名を引数に取った untrace 関数を呼び出すことで取り除くことができる。

```
> get("[<-")
.Primitive("[<-")
> trace("[<-")
> x <- 1:10
> x[3] <- 4
trace: "[<-"(*tmp*, 3, value = 4)
```

9.4 traceback

あるエラーが最上位へのジャンプを惹き起こすと .Traceback と呼ばれる特殊変数が最上位の作業スペースに置かれる。.Traceback はエラーが生じた時に活動していた関数呼び出し各々に対する項目を持つ文字ベクトルである。.Traceback のチェックは traceback を呼び出すことで行うことができる。

10 構文解析

構文解析器とは R のテキストコードを、指定された指示を実行する R の評価機構へ引き渡すされるかもしれない内部形式に変換するものである。内部形式はそれ自身 R のオブジェクトであり、保管したり、さもなければ、R システムの内部で処理される。

10.1 構文解析の過程

10.1.1 構文解析のモード

R に於ける構文解析は三つの異なる変種を持つ：

- 読み取り・評価・出力ループ
- テキストファイルの構文解析
- 文字列の構文解析

読み取り・評価・出力ループは R に対する基本的な命令行インタフェイスを形成する。テキスト入力は完全な R の表現式が得られるまで読み取られる。表現式は複数の入力行に分割されていても良い。初期プロンプト (既定で '> ') は構文解析器が新しい表現式を読み取ることが可能であることを示し、継続プロンプト (既定では '+ ') は構文解析器が不完全な表現式の残りを期待していることを意味する。表現式は入力中に内部形式に変換され、構文解析済みの表現式が評価機構に引き渡され、そして結果が出力される (不可視の指定がされていない限り)。もし構文解析器が言語の仕様と矛盾する状態にあることを見出すと "Syntax Error" のフラグが立てられ、構文解析器は自分自身をリセットし、次の入力行の先頭の入力から再開する。

テキストファイルは `parse` 関数を使って構文解析できる。特に、これは `source` 関数の実行中に実行され、命令を内部ファイルに保管し、それらがあたかもキーボードからタイプされたかのように実行する。しかしながら、どれかの評価が行われる前に、全てのファイルが構文解析され、構文が検査されることを注意しよう。

文字列やそれらのベクトルは `parse` 関数への `text=` 引数を使って構文解析できる。文字列はあたかのそれらが入力ファイルの行であるのと全く同じに扱われる。

10.1.2 内部表現

構文解析された表現式は構文解析木を含む R オブジェクトに保管される。そうしたオブジェクトのより詳しい記述は Section 2.1.3 [言語オブジェクト], page 3 と Section 2.1.4 [表現式オブジェクト], page 4 にある。簡単にいえば、R の全ての初等的表現式は関数呼び出し形式で保管される、つまり、最初の要素が関数名で、残りはそれ自身さらに R の表現式であり得る引数からなるリストである。このリスト要素は名前が付けられ、形式的引数と実際の引数のタグ付き照合に対応する。全ての R 構文要素はこうした方法で扱われることを注意しよう、たとえば代入 `x <- 1` は `"<-"(x, 1)` とコード化される。

10.1.3 逆構文解析

任意の R オブジェクトは `deparse` を用いて R の表現式に変換できる。これはしばしば出力結果との関連で使われる、たとえばプロットのラベルが例である。モード `"expression"` のオブジェクトだけが逆構文解析結果を再構文解析しても変わらないことが期待できる。例えば数値ベクトル `1:5` は `"c(1, 2, 3, 4, 5)"` と逆構文解析され、これは関数 `c` の呼び出しとして再構文解析されるであ

ろう。可能な限り、逆構文解析された表現式と、その再構文解析された表現式はもとのそれを評価したのと同じ結果を与えるように努められるが、もともとテキスト表現から生成されたのではないようなものを含む、幾つかの厄介な例外がある。

10.2 トークン

トークン are the elementary building blocks of a programming language. They are recognised during *lexical analysis* which (conceptually, at least) takes place prior to the syntactic analysis performed by the parser itself.

10.2.1 定数

４つの定数型、論理値、数値、複素数値、そして文字列がある。

更に４つの特殊定数、`NULL`, `NA`, `Inf` そして `NaN` がある。

`NULL` は空のオブジェクトを表す。`NA` は欠損 (“Not Available”) データ値を表す。`Inf` と `NaN` はそれぞれ IEEE 浮動小数計算における無限大と非数 (例えば、それぞれ演算 1/0 と 0/0 の結果) を表す。

論理定数は `TRUE` か `FALSE` のどちらかである。

数値定数は **C** 言語と類似の構文に従う。それらは、オプションとして ‘`.`’ を従える零かそれ以上の数字からなる整数部分、オプションとして ‘`E`’ もしくは ‘`e`’ そして符号と零またはそれ以上の数字からなる文字列からなる指数部分を従える、零もしくはそれ以上の数字からなる小数部分、からなる。小数、整数部分のどちらかはなくても良いが、同時に両方無いことは許されない。

適正な数値定数： `1 10 0.1 .2 1e-7 1.2e+7 2e 3e+`

最後の二つの例は実際にはほとんど使われないであろうが、それぞれ ‘2’ と ‘3’ として認識される。

整数値定数という特別なクラスが無いことを注意しよう。

また先頭の符号は定数の一部ではなく、単項演算子として扱われることを注意しよう。

複素数値定数は数値定数とその後に続く ‘`i`’ という形を持つ。純虚数だけが実際の複素数値であり、他の複素数値は数値と虚数に対する単項、もしくは二項演算として構文解析されることを注意しよう。

適正な複素数値： `2i 4.1i 1e-2i`

文字列定数は一重引用符 (‘`'`’) または二重引用符 (‘`"`’) の対で囲まれ、他の印字可能な全ての文字を含むことができる。引用符もしくは他の特殊文字は エスケープ文字列 を用いて指定できる：

`\'` 一重引用符

`\"` 二重引用符

`\n` 改行

`\t` タブ文字

`\b` バックスペース

`\`*nnn* 与えられたコードを持つ文字

一重引用符はまた直接二重引用符による囲み中にいれることができ、逆もまた真である。文字列 `"NA"` は “欠損値” という特別な意味を持つ。

10.2.2 識別子

識別子は文字、数字、そしてピリオッド記号 (‘.’) からなる。数字や、ピリオッドに続く数字から始まってはならない。

文字の定義は現在のロケールに依存する。もしこれが正しく指定され (これはシステム依存である) いれば、ユーザーの母語文字を識別子に使うことができる。

ピリオッドで始まる識別子は既定では `ls` 関数で表示されず、‘...’ そして ‘..1’, ‘..2’, 等は特殊であることを注意しよう。

オブジェクトはまた識別子では無い名前を持つことができることを注意しよう。これらは一般に `get` と `assign` でアクセスできるが、ある限られた環境では曖昧さが無い限りテキスト文字列でも表示できる (例えば `"x" <- 1`)。

10.2.3 予約語

以下の識別子は特殊な意味を持ち、オブジェクト名として使うべきではない：

```
if else repeat while function for in next break
TRUE FALSE NULL NA Inf NaN
... ..1 ..2 etc.
```

10.2.4 特殊演算子

R ではユーザーが独自の中置 (infix) 演算子を定義できる。これらは文字 ‘%’ で挟まれた文字列という形式を持つ。文字列は ‘%’ を除く全ての印字可能な文字を含むことができる。エスケープ文字列はこの限りではない。

以下の演算子が予め定義されている。

```
%% %*% %/% %in% %o% %x%
```

10.2.5 分離記号

文字通りにはトークンでは無いが、引き続く空白文字 (スペースとタブ) は曖昧さが残る場合に区切りとして使うことができる (`x<-5` と `x < -5` を比較せよ)。

セミコロン (‘;’) は初等的表現式を分離するのに使われる。

改行はトークン分離子と表現式の終了指示を兼ね備えた機能を持つ。もし表現式が行末で終了することができるなら、構文解析器はそれをそうしたものとして扱い、さもなければ改行は空白として扱われる。

キーワード `else` には特殊な規則が適用される： 複合表現式の内部では、`else` の前の改行は無視され、一方で最も外側のレベルでは改行は `if` 構文を終了させ `else` は構文エラーを惹き起こす。この少々変則的な挙動は、R が対話的に使用できるように作られており、したがってユーザーが ⟨RET⟩ を押すや否や、入力された表現式が完全であるか、不完全であるか、または不正であるかを判断しなければならないことによる。

コンマ (‘,’) は関数引数と多重添字を分離するために使われる。

10.2.6 演算子のトークン

R uses the following operator tokens

`+ - * / %% ^`	算術
`> >= < <= == !=`	関係
`! & |`	論理
`~`	モデル公式
`-> <-`	代入
`$`	リスト添字操作
`:`	数列

(これらの演算子はモデル公式の内部では異なった意味を持つ)

10.2.7 グループ化シンボル

通常の括弧—'(' と ')'—は表現式内部での厳密なグループ化と、関数定義および関数呼び出しに対する引数リストを区切るために使われる。

中括弧—'{' and '}'—は関数定義中の表現式のブロック、条件付き表現式、そして繰返し構成、を区切る。

10.2.8 添字操作のトークン

配列やベクトルの添字操作は一重もしくは二重の鉤括弧記号 '`[]`' と '`[[]]`' を使ってなされる。同様に、タグ付きリストの添字操作は '`$`' 演算子を用いて行われる。

10.3 表現式

R のプログラムは R 表現式の列からなる。一つの表現式は定数と識別子だけからなる単純な表現式であるか、他の部品 (それ自身表現式であって良い) から構成された複合表現式である。

次の節では利用可能な様々な構文的構成を詳しく述べる。

10.3.1 関数呼び出し

関数呼び出しは関数参照に引き続く、何組かの括弧中のコンマで区切られた引数のリストという形式を持つ。

function_reference (*arg1*, *arg2*, ... , *argn*)

関数参照は次のどれかである

- 一つの識別子 (関数名)
- 文字列 (上と同じであるが、関数名が適正な識別子でない名前を持つとき簡略である)
- 表現式 (関数オブジェクトとして評価できる必要がある)

各引数はタグを持つことができる (*tag=expr*)。もしくは単に単純表現式である。空であっても良く、特殊記号 '`...`', '`..2`', 等であっても良い。

タグは識別子または文字列である。

例：

```
f(x)
g(tag = value, , 5)
"odd name"("strange tag" = 5, y)
(function(x) x^2)(5)
```

10.3.2 間置演算子と前置演算子

演算子の優先度 (高いもの程先に適用) は

```
^
- +                    (単項演算)
:
%xyz%
* / %%
+ -                    (二項演算子)
> >= < <= == !=
!
&
|
~                      (単項または二項演算子)
->
<-
```

巾乗演算子 '^' と左代入演算子 '<-' は右から左へとグループ化される。他の全ての演算子は左から右へとグループ化される。つまり `2 ^ 2 ^ 3` は 2^8 であり 4^3 ではなく、他方 `1 - 1 - 1` は -1 であり 1 では無い。

厳密には演算子では無いけれど、シンボル '=' symbol が関数呼び出しにおけるタグ付き引数と、関数定義における既定値代入に対して使われることを述べる必要がある。

'$' 記号はある意味で演算子であるが、任意の右辺値を許すわけではなく、⟨undefined⟩ [Index constructions], page ⟨undefined⟩ で説明されている。これは他のどの演算子よりも高い優先度を持つ。

単項または二項演算子の構文解析済みの形式は、演算子名を関数名に持ち、被演算項を関数引数に持つ関数呼び出しと全く同値である。

括弧は、演算子の優先度から括弧が推測される場合 (例えば `a * (b + c)`) を含め、名前 `"("` を持つ単項演算子として記録される。

代入記号は算術、関係そして論理演算子と全く同類の演算子であることを注意しよう。構文解析器に関する限り、任意の表現式が代入の被代入項に許される (`2 + 2 <- 5` は、評価機構ははねつけるものの、構文解析に関する限り適正な表現式である。同じ注意はモデル公式演算子に付いても当てはまる。

10.3.3 添字構成

R は三つの添字構成を持ち、そのうち二つは構文的には少し異なるが意味的には類似している。

object [*arg1*, ... , *argn*]
object [[*arg1*, ... , *argn*]]

object は形式的には任意の適正な表現式で良いが、部分集合演算が可能なオブジェクトを表すか、そうしたオブジェクトに評価できることが仮定されている。引数は一般に数値または文字添字として評価されるが、他の種類の引数も可能である (特に `drop = FALSE`)。

内部的にはこれらの添字構成はそれぞれ関数名 `"["` と `"[["` を持つ、関数呼び出しとして保管される。

三番目の添字構成は

object `$` *tag*

ここで *object* は上と同じであるが *tag* は識別子か文字列である。内部的にはこれは名前 `"$"` を持つ関数呼び出しとして保管される。

10.3.4 複合表現式

複合表現式は次の形式を持つ

`{` *expr1* `;` *expr2* `;` `......` `;` *exprn* `}`

セミコロンは改行で置き換えることができる。内部的には、これは`"{"` を関数名とし表現式を引数に持つ関数呼び出しとして保管される。expressions as arguments.

10.3.5 流れ制御要素

R は次のような制御構造を特殊な構文要素として持つ constructs

`if (` *cond* `)` *expr*
`if (` *cond* `)` *expr1* `else` *expr2*
`while (` *cond* `)` *expr*
`repeat` *expr*
`for (` *var* `in` *list* `)` *expr*

これらの構成要素中の表現式は典型的に複合表現式である。

繰返し構造 (`while`, `repeat`, `for`) 中では `break` (ループを停止するために) そして `next` (次の繰返しをスキップするために) 使うことができる。

内部的にはこれらの構成は関数呼び出しとして保管される：

`"if"(`*cond*`,` *expr*`)`
`"if"(`*cond*`,` *expr1*`,` *expr2*`)`
`"while"(`*cond*`,` *expr*`)`
`"repeat"(`*expr*`)`
`"for"(`*var*`,` *list*`,` *expr*`)`
`"break"()`
`"next"()`

10.3.6 関数定義

関数の定義はつぎの形式を持つ

`function (` *arglist* `)` *body*

関数の本体は表現式であり、しばしば複合表現式である。*arglist* はコンマで区切られた項目のリストであり、各々は識別子、'*identifier* = *default*' の形式、もしくは特殊シンボル... のどれでも良い。*default* は任意の適正な表現式であって良い。

関数引数はリストのタグ等とは異なり、テキスト文字列で与えられる "奇妙な名前" を持つことはできないことを注意しよう。

内部的には関数定義は関数名 `function` と二つの引数、*arglist* と *body*、をもつ関数呼び出しとして保管される。*arglist* はタグ付きの対リストとして保管され、タグは引数名、そして値は既定の表現式である。

関数と変数の索引

Appendix A 参考文献

Richard A. Becker, John M. Chambers and Allan R. Wilks (1988), *The New S Language.* Chapman & Hall, New York. この本はしばしば “青本 *(Blue Book)*” と呼ばれる。