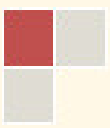
Рабочая группа разработки R

Определение языка R

Версия 3.1.1(2014-07-10) DRAFT

Москва, 2014



R Language Definition

Version 3.1.1(2014-07-10) DRAFT

R Core Team

Это руководство для R, версии 3.1.1 (2014-07-10).

Перевод с английского А.А.Фоменко
По общим вопросам обращаться по адресу:
http://www.aafomenko@yandex.ru

Определение языка *R*

Версия 3.1.1 (2014-07-10)

Перевод и редакция А.А.Фоменко. – Москва, 2014. 60 с. – (серия технической документации).

Данная книга является переводом одноименной книги из комплекта технической документации, поставляемой в составе дистрибутива ***R,*** и призвана восполнить пробел в русской локализации системы ***R***.

2014

1. Введение

R - система для статистических вычислений и графики. Она включает, между прочим, язык программирования, высокоуровневую графику, интерфейсы для других языков и средства отладки. Этот справочник детализирует и определяет язык ***R.***

Язык ***R*** - диалект ***S,*** который был разработан в 1980-ых и с тех пор находится в широком использовании в статистическом сообществе. Его основной разработчик, Джон М. Чемберс, был награжден за S Премией по системному программному обеспечению ACM 1998 года.

У синтаксиса языка есть поверхностное сходство с ***C,*** но семантика принадлежит к семейству FPL (языкам функционального программирования) с более сильной связью с Lisp и APL. В частности, возможно “вычислять на языке”, который поочередно позволяет писать функции, которые берут выражения в качестве входа, что-то, что часто полезно для статистического моделирования и графики.

Можно получить довольно примитивное использование ***R*** в интерактивном режиме, выполняя простые выражения из командной строки. Некоторые пользователи никогда, возможно, не уйдут с этого уровня, другие захотят написать свои собственные функции или оперативно систематизировать однообразную работу или на перспективу записать дополнительные пакеты для новой функциональности.

Цель этого справочника состоит в документировании языка по существу. Это означает, объекты показаны, так как они работают, и детали процесса вычисления выражений, которые полезно знать при программировании функций ***R.*** Главные подсистемы для определенных задач, таких как графика, описаны в этом справочнике лишь кратко и будут задокументированы отдельно.

Хотя большая часть текста одинаково применима к ***S,*** есть также некоторые существенные различия, и чтобы не перепутать проблему, сконцентрируемся на описании ***R.***

Проект языка содержит много тонкостей и распространенных ошибок, которые могут удивить пользователя. Большинство из них происходит из-за соображений непротиворечивости на более глубоком уровне, как мы объясним. Есть также много полезных ярлыков и идиом, которые позволяют пользователю выражать вполне сложные операции кратко. Многие из них становятся естественными по мере ознакомления с базовыми понятиями. В некоторых случаях существует много способов выполнить задачу, но некоторые из методов основаны на реализации языка, а другие работают на более высоком уровне абстракции. В таких случаях укажем на преимущественное использование.

Предполагается некоторое знакомство с ***R.*** Это не введение в ***R,*** а скорее справочник программиста. Другие справочники предоставляют дополнительную информацию: в особенности раздел "Предисловие" во Введении в ***R*** предоставляет введение в ***R,*** и раздел “Система и интерфейсы внешних языков” в Написание расширений ***R*** детализирует расширение ***R,*** используя скомпилированный код.

2. Объекты

На каждом машинном языке переменные обеспечивают средство доступа к данным, хранящимся в памяти. ***R*** не обеспечивает прямой доступ к памяти компьютера, а скорее обеспечивает много специализированных структур данных, именуемых объектами. На эти объекты ссылаются через символы или переменные. В ***R,*** однако, символы - самостоятельно объекты и могут управляться таким же образом как любой другой объект. Это отличается от многих других языков и имеет широко распространяющиеся следствия.

В этой главе даны предварительные описания различных структур данных, предоставленных в ***R.*** Более детальные обсуждения многих из них будут найдены в последующих главах. Функция определения в ***R*** *typeof* возвращает *тип* объекта ***R.*** Заметим, что в коде ***C,*** лежащем в основе ***R,*** все объекты являются указателями на структуру с определением типа SEXPREC; различные типы данных ***R*** представлены в ***C*** SEXPTYPE, который определяет, как используется информация в различных частях структуры.

Следующая таблица описывает возможное значение, возвращенное *typeof,* и их значение.

"NULL"	NULL
"symbol"	имя переменной
"pairlist"	парный объект (в основном внутренний)
"closure"	функция
"environment"	окружающая среда
"promise"	объект, используемый для отложенной оценки
"language"	конструкция языка **R**
"special"	внутренняя функция, которая не вычисляет свои аргументы
"builtin"	внутренняя функция, которая вычисляет свои аргументы
"char"	a 'scalar' строковый объект (только внутренний) ***
"logical"	вектор, содержащий логические значения
"integer"	вектор, содержащий целые значения
"double"	вектор, содержащий реальные значения
"complex"	вектор, содержащий комплексные значения
"character"	вектор, содержащий символьные значения
"..."	таргумент определенной переменной длины ***
"any"	специальный тип, который заменяет все типы: не существует объектов такого типа
"expression"	объект выражение
"list"	список
"bytecode"	код в байтах (только внутренне) ***
"externalptr"	объект внешнего указателя
"weakref"	объект слабой ссылки
"raw"	вектор, содержащий байты
"S4"	объект S4, который не является простым объектом

Пользователи не могут просто получить объекты, помеченные '***'.

Функциональный режим дает информацию о режиме объекта в смысле Becker, Chambers & Wilks (1988), и является более совместимым с другими реализациями языка ***S***. Наконец, функция *storage.mode* показывает режим хранения ее аргумента в смысле Беккера и др. (1988). Она обычно используется при вызове функции, записанной на другом языке, таких как ***C*** или ФОРТРАН для гарантирования, что объекты ***R*** имеют тип данных, который ожидает вызываемая подпрограмма. (На языке ***S*** векторы с целочисленными или действительными значениями имеют оба "числовой" режим, таким образом, их режимы хранения нужно отличать.)

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

Объекты в ***R*** часто преобразовываются к различным типам во время вычислений. Также имеется много доступных функций для выполнения явного преобразования.

При программировании на языке ***R*** тип объекта обычно не влияет на вычисления, однако, имея дело с внешними языками или операционной системой, часто необходимо гарантировать корректность типа объекта.

2.1. Основные типы

2.1.1. Векторы

Вектора рассматриваются как непрерывная последовательность ячеек, содержащих данные. Доступ к ячейкам осуществляется через операции индексирования, такими, как x[5]. Более детально рассмотрено в разделе 3.4 [индексирование].

R имеет шесть основных (‘атомарных’) типов векторов: *logical, integer, real, complex, string (or character)* и *raw*. Режим и режим хранения для разных типов векторов перечислены в следующей таблице.

Тип	mode	storage.mode
logical	*logical*	***logical***
integer	*numeric*	***integer***
double	*numeric*	***double***
complex	*complex*	***complex***
character	*character*	***character***
raw	***raw***	***raw***

Отдельные числа, такие как 4.2, и строка, такая как "*four point two*", все еще векторы, длины 1; нет больше основных типов. Возможны (и полезны) векторы с нулевой длиной.

У векторов строки есть режим и режим хранения "*character*". Отдельный элемент символьного вектора часто упоминается как *символьная ст рока.*

2.1.2. Списки

Списки (“универсальные векторы”) являются другим видом хранения данных. У списков есть элементы, каждый из которых может содержать любой тип объекта ***R,*** то есть элементы списка не обязательно имеют одинаковый тип. К элементам списка получают доступ посредством трех различных операций индексации. Они объяснены подробно в разделе 3.4 [Индексация].

Списки - векторы, и основные типы векторов упоминаются как атомарные векторы, где необходимо исключить списки.

2.1.3. Языковые объекты

Есть три типа объектов, которые составляют язык ***R.*** Это – *call* (вызов), *expressions* (выражения) и *name* (имя). Так как у ***R*** есть объекты типа "выражение", то мы попытаемся избежать использования слова «выражение» в других контекстах. В определенных синтаксически корректных высказываниях выражения будут упоминаться как *операт оры.*

У этих объектов есть режимы "*call*", "*expression*" и "*name*", соответственно.

Они могут быть созданы непосредственно из выражений, используя механизм кавычек и преобразованы «в» и «из» списков функциями *as.call* и *as.list.* Могут быть извлечены компоненты дерева синтаксического анализа, используя стандартные операции индексации.

2.1.3.1. Символьные объекты

Символы обращаются к объектам ***R.*** Обычно имя любого объекта ***R*** - символ.

Символы могут быть созданы через функции *as.name* и кавычку.

Символы имеют режим "*name*", режим хранения "*symbol*" и тип "*symbol*". Они могут быть преобразованы «в» и «из» символьных строк, используя *as.character* и *as.name*. Они естественно появляются как атомы проанализированных выражений, попробуй, например, *as.list(quote(x + y))*.

2.1.4. Выражения - объекты

В ***R*** можно иметь объекты типа " *expression*". Выражение содержит одно или более предложений. Оператор - синтаксически корректный набор маркеров. Объекты выражения - специальные объекты языка, которые содержат проанализированные, но неоцененные операторы ***R.*** Основное различие состоит в том, что объект выражения может содержать несколько таких выражений. Другие более тонкие различия состоят в томо, что объекты типа " *expression* " оцениваются лишь при явной передаче на вычисление, тогда как другие объекты языка могут быть оценены в некоторых неожиданных случаях.

Объект выражения ведет себя также как список, и к его компонентам можно получить доступ таким же образом как компонентам списка.

2.1.5. Объекты функции

В ***R*** функции - объекты и могут управляться почти таким же способом как любой другой объект. У функций (или более точно, обертка функции) есть три основных компонента: формальный список аргументов, тело и окружающая среда. Список аргументов - список разделенных запятой значений аргументов. Аргумент может быть символом, или конструкцией *«symbol = default»*, или специальным аргументом '...'. Вторая форма аргумента используется для указания значения по умолчанию для аргумента. Это значение будет использоваться, если функция будет вызвана без какого-либо значения, указанного для этого аргумента. Аргумент '...' является особенным и может содержать любое число аргументов. Он обычно используется, если число аргументов неизвестно или в случаях, где аргументы будут переданы другой функции.

Тело – синтаксически проанализированный оператор ***R.*** Это обычно набор операторов в фигурных скобках, но также может быть отдельный оператор, символ или даже константа.

Окружающая среда функции является средой, которая была активной при создании функции. Любой символ ограничен своей окружающей средой, связан и доступен функции. Эту комбинацию кода функции и привязки в ее окружающей среде называют 'оберткой функции', термином из теории функционального программирования. В этом документе обычно используется термин 'функция', но используется 'обертка', чтобы подчеркнуть значимость присоединенной среды.

Можно извлечь и управлять тремя частями обертки объекта, используя конструкции *formals*, *body* и *environment* (все три могут также использоваться на левой стороне присваивания). Последний из них может использоваться для удаления нежелательной привязки среды.

При вызове функции создается новая среда (называемая средой оценки), чье пространство (см. раздел 2.1.10 [объекты среды]) является средой от обертки функции. Эта новая среда первоначально заполнена неоцененными аргументами функции; поскольку оценка продолжается, локальные переменные создаются в ее пределах.

Есть также средство для преобразования функции «в» и «из» списочной структуры, используя *as.list* и *as.function*. Они были включены для совместимости с ***S*** и их использование обескураживает.

2.1.6. NULL

Существует специальный объект, называемый NULL. Он используется всякий раз, когда есть потребность идентифицировать или указать отсутствие объекта. Его не следует путать с вектором или списком нулевой длины.

Объект NULL не имеет типа и каких-либо поддающихся изменению свойств. В ***R*** есть только один объект NULL, к которому обращаются все экземпляры. Для проверки на NULL используют *is.null*. Нельзя установить атрибуты на NULL.

2.1.7. Встроенные объекты и специальные формы

Эти два вида объекта содержат встроенные функции ***R,*** то есть, те, которые выведены на экран как *.Primitive* в листингах кода (так же как те, к которым получают доступ через функцию *.Internal* и, следовательно, не видимые пользователем как объекты). Различия между ними заключается в обработке аргумента. Все собственные аргументы встроенных функций оцениваются и передаются внутренней функции в соответствии с *вызовом по значению,* тогда как специальные функции передают не оцененные аргументы внутренней функции.

Для языка ***R*** эти объекты - только другой вид функции. Функция *is.primitive* может отличить их от интерпретируемых функций.

2.1.8. Обещанные объекты

Объекты обещания - часть механизма отложенных вычислений ***R.*** Они содержат три слота: значение, выражение и окружающая среда. При вызове функции сравниваются аргументы, а затем каждый из формальных аргументов является обязательством к обещанию. Выражение, которое было дано для формального аргумента, и указатель на окружающую среду функции вызываются из сохраненных в обещании.

Пока к этому аргументу не получают доступ, не существует какого-либо значения, присоединенного с обещанием. Когда к аргументу получают доступ, сохраненное выражение оценивается в сохраненной окружающей среде и возвращается результат. Результат также сохранен обещанием. Функция замены извлечет контент слота выражения. Это позволяет программисту получать доступ или к значению или к выражению, присоединенному с обещанием.

В пределах языка ***R*** объекты обещания видимы почти неявно: фактические аргументы функции имеют этот тип. Есть также функция *delayedAssign*, которая сделает обещание из выражения. Отсутствует какой-либо способ в коде ***R*** для проверки, является ли объект обещанием или нет, и при этом нет способа использовать код ***R*** для определения окружающей среды обещания.

2.1.9. Точка-точка-точка

Тип объекта '*...*' хранится как тип *pairlist*. К компонентам '...' можно получить доступ обычным *pairlist* способом из кода ***C,*** но не легко получить доступ как к объекту в интерпретируемом коде. Объект может быть получен как список, так, например, в таблице ниже:

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

Если функция имеет '...' в качестве формального аргумента, который не соответствует формальным аргументам, то им сопоставляется '...'.

2.1.10. Окружающая среда

Будем считать, что окружающие среды могут состоять из двух частей. *Фрейм,* состоящий из набора пар символ-значение, и *оберт ка* - указатель на обертку

окружающей среды. Когда ***R*** ищет значение для символа, исследуется фрейм и, если соответствующий символ будет найден, его значение будет возвращено. В противном случае получают доступ к обертке окружающей среды, и процесс повторяется. Окружающие среды формируют древовидную структуру, в которой обертки играют роль родителей. Дерево окружающих сред имеет корень в пустой окружающей среде, доступной через *emptyenv(),* у которой нет родителя. Это - прямой родитель окружающей среды основного *(base)* пакета (доступной через функцию *baseenv()).* Прежде у *baseenv()* было специальное значение NULL, но начиная с версии 2.4.0 использование NULL в качестве окружающей среды не допускается.

Окружающие среды создаются неявно вызовами функции, как описано в разделе 2.1.5 [Объекты функции] и разделе 3.5.2 [Лексическая среда]. В этом случае окружающая среда содержит переменные, локальные для функции (включая аргументы), а ее обертка является окружающей средой вызванной в настоящий момент функции. Окружающие среды также можно создать непосредственно *new.env.* К контенту фрейма окружающей среды можно получить доступ и манипулирование при помощи *ls*, *get* и *assign*, включая *eval* и *evalq.*

Может использоваться функция *parent.env* для получения доступа к обертке окружающей среды.

В отличие от большинства других объектов ***R,*** окружающая среда не копируется при передаче в функцию или использовании в присвоениях. Таким образом, если присвоить одну и ту же окружающую среду нескольким символам и изменить одну из них, то другие изменятся также. В частности присвоение атрибутов окружающей среде может привести к неожиданностям.

2.1.11. Объекты парных списков

Объекты парных списков (Pairlist) подобны спискам точечной пары Lisp. Они интенсивно используются внутри ***R,*** но редко видимы в интерпретируемом коде, хотя они возвращаются *formals*, и могут быть созданы, например, функцией *pairlist*. *pairlist* с нулевой длиной равной NULL, как ожидалось бы в Lisp, но в отличие от списка нулевой длины. У каждого такого объекта есть три слота, значение CAR, значение CDR и значение TAG. Значение TAG - текстовая строка, и CAR и CDR обычно представляют, соответственно, элемент списка (голова) и остаток (хвост) списка с объектом NULL как разделитель (терминология CAR/CDR – традиционна для Lisp и первоначально упоминалась как адрес и декрементные регистры на компьютерах IBM в начале 60-ых).

Парные списки обрабатываются на языке ***R*** точно таким же образом как универсальные векторы ("*list*s"). В частности к элементам получают доступ, используя то же самый синтаксис [[]]. Использование парных списков является сомнительным, так как универсальные векторы обычно более эффективны в использовании. При получении доступа к внутреннему парному списку из ***R*** обычно (включая подмножества) преобразуют в универсальный вектор.

В очень немногих случаях парные списки видимы пользователем: одним из таких является *.Options*.

2.1.12. Тип “Any”

В действительности для объекта невозможно иметь тип "*Any*", но это, однако, допустимое значение типа. Это бывает при определенных (довольно редких) обстоятельствах, например, *as.vector(x,"any")* указывает, что не следует делать приведение типа.

2.2 . Атрибуты

У всех объектов, кроме NULL, могут быть один или более атрибутов, присоединенных к ним. Атрибуты сохранятся как *pairlist*, где все элементы именованы,

но их следует рассматривать как ряд пар *name=value*. Можно получить список атрибутов, используя *attributes* и устанавливая с помощью *attributes<-*, отдельные компоненты доступны с помощью *attr* и *attr <-*.

У некоторых атрибутов есть специальные функции доступа (например, *levels<-* для факторов), и они должны использоваться при доступности. В дополнение к скрытым деталям реализации они могут выполнить дополнительные операции. ***R*** пытается принять вызовы *attr<-* и *attributes<-*, которые включают специальные атрибуты, и осуществляет проверку непротиворечивости.

Матрицы и массивы - просто векторы с атрибутом размерности *dim* и дополнительно с именем размерности (*dimnames*), присоединенные к вектору.

Атрибуты используются для реализации структуры класса, используемой в ***R.*** Если у объекта есть атрибут класса, то атрибут будет исследован во время оценки. Структура класса в ***R*** описана подробно в главе 5 [Объектно-ориентированное программирование].

2.2.1. Имена

Атрибут имени (*names*) при его присутствии маркирует отдельные элементы вектора или списка. При печати объекта атрибут имени при его присутствии используется для маркировки элементов. Атрибут имен может также использоваться для индексирования результата, например, *quantile(x) ["25 %"]*.

Можно получить и определить имена, используя конструкции *names* и *names<-*. Последняя выполнит необходимые проверки непротиворечивости для гарантии, что у атрибута имен есть надлежащий тип и длина.

Отдельно обрабатываются парные списки и одномерные массивы. Для объектов парных списков (*pairlist*) используется виртуальный атрибут имен; в действительности атрибут имен создается из тегов компонентов списка. Для одномерных массивов атрибут имени (*name*) реально доступен из имен размерности (*dimnames [[1]]).*

2.2.2. Размерность

Атрибут размерности *dim* используется при реализации массивов. Контент массива сохраняется в векторе в порядке по столбцам, и атрибут размерности является вектором целых чисел, указывающие соответствующие пределы массива. ***R*** гарантирует, что длина вектора равна произведению длин размерностей. Длина одной или более размерностей может быть нулем.

Вектор не является одномерным массивом, так как у последнего атрибут размерности длиною один, тогда как у вектора атрибут размерности отсутствует.

2.2.3. Имена размерности

Массивы могут назвать каждую размерность, отдельно используя атрибут имен размерности (*dimnames*), который является списком символьных векторов. У списка имен размерностей самостоятельно могут быть имена, которые затем используются для заголовков пределов при печати массива.

2.2.4. Классы

У ***R*** есть тщательно продуманная система классов, которая преимущественно управляется через атрибут класса (*class*). Этот атрибут - символьный вектор, содержащий список классов, от которых наследовался объект. Он формирует основание “универсальных методов” функциональности ***R.***

К этому атрибуту можно получить доступ и управление фактически без ограничения пользователями. Отсутствует проверка, что объект фактически содержит компоненты, которые ожидают методы класса. Таким образом, изменение атрибута класса должно быть сделано с осторожностью, и при их доступности следует предпочесть определенные функции создания и преобразования.

2.2.5. Атрибуты временных рядов

Атрибут *tsp* используется для поддержания аргументов временного ряда: начало, конец и частота. Эта конструкция, главным образом, используется для обработки рядов с периодической подструктурой, например, месячные или квартальные данные.

2.2.6. Копирование атрибутов

Должны ли атрибуты быть скопированы при изменении объекта, находящегося в комплексной области, но есть некоторые общие правила (Becker, Chambers & Wilks, 1988, стр. 144-6).

Скалярные функции (те, которые работают поэлементно на векторе и чей выход подобен входу) должны сохранить атрибуты (кроме, возможно, класса).

Бинарные операции обычно копируют большинство атрибутов более длинного аргумента (и если они имеют одинаковую длину для обоих, то предпочитается значение для первого). Здесь 'большинство' означает все кроме имен, размерностей и имен размерностей, которые установлены соответственно кодом для оператора.

Подмножество (кроме индексированных пустым индексом) обычно отбрасывает все атрибуты кроме имен, размерностей и имен размерностей, которые сброшены как соответствующие. С другой стороны подприсвоение обычно сохраняет атрибуты, даже если длина изменена. Приведение отбрасывает все атрибуты.

Метод по умолчанию для сортировки отбрасывает все атрибуты кроме имен, которые сортируются наряду с объектом.

2.3. Специальные составные объекты

2.3.1. Факторы

Факторы используются для описания элементов, у которых может быть конечное количество значений (пол, социальный класс, и т.д.). У фактора есть атрибут уровней и класс "фактор" *(factor*). Дополнительно, он также может содержать атрибут *contrasts*, который управляет параметризацией при использовании факторов в функциях моделирования.

Фактор может быть просто номинальным или, возможно, иметь упорядоченные категории. В последнем случае он должен быть определен как таковой и иметь вектор класса (*class) c("ordered"," factor").*

В настоящий момент факторы реализованы с использованием целочисленного массива для указания фактических уровней, и второй массив имен, которые отображены на целые числа. Скорее, к сожалению, пользователи часто используют реализацию для облегчения некоторых вычислений. Это, однако, является проблемой реализации и, как гарантируют, не будет иметь место во всех реализациях ***R.***

2.3.2. Объект фрейм данных

Фреймы данных – это структуры ***R,*** которые наиболее близко подражают SAS или набору данных SPSS, то есть “переключение переменными ” матриц данных.

Фрейм данных является списком векторов, факторов, и/или матриц, имеющих одинаковую длину (число строк в случае матриц). Кроме того, у фрейма данных обычно есть атрибут имен, маркирующий переменные и атрибут *row.names* для маркировки переключателей *(cases*).

Фрейм данных может содержать список, который имеет одинаковую длину с другими компонентами. Список может содержать элементы разных длин, таким образом, обеспечивая структуру данных для массивов с переменной длиной строк. Однако для таких записей обычно массивы обрабатываются не правильно.

3. Оценка выражений

При вводе команды пользователем в запросе (или при считывании выражения из файла), сначала команды преобразуются синтаксическим анализатором во внутреннее представление. Средство оценки выполняет синтаксический анализ выражения ***R*** и возвращает значение выражения. У всех выражений есть значение. Это - основа языка.

Эта глава описывает основные механизмы средств оценки, но избегает обсуждения определенных функций или групп функций, которые описаны позже в отдельных главах или где страницы справки должны быть достаточной документацией.

Пользователи могут создать выражения и вызвать их оценку.

3.1. Простая оценка

3.1.1. Константы

Любое число, введенное непосредственно в запросе, является константой и оценивается.

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

Возможно, неожиданно, но число, возвращенное из выражения 1, является действительным. В большинстве случаев разница между целым числом и действительным значением будет незначительна, поскольку ***R*** делает округление справа при использовании чисел. Однако может возникнуть необходимость создания целочисленного значения для константы. Это можно сделать, вызывая функцию *as.integer* или используя различные другие методы. Но возможно самый простой подход состоит в сопровождении константы символьным суффиксом ‘*L*’. Например, для создания целочисленного значения 1 можно использовать:

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

Можно использовать суффикс *‘L’*, чтобы квалифицировать любое число с намерением создания из него явно целое число. Таким образом, *'0x10L'* создает целочисленное значение *16* из шестнадцатеричного представления. Константа *1e3L* дает *1000* как целое число, а не числовое значение и эквивалентна *1000L*. Заметим, что *‘L’* обработан как квалификация аргумента *1e3* а не *3*. Если мы квалифицируем значение с *‘L’*, которое не является целочисленным значением, например, *1e-3L,* мы получаем предупреждение, и создается действительное значение. Также предупреждают при наличии ненужной десятичной точки в числе, например, *1.L*.

Получим синтаксическую ошибку при использовании ‘***L***’ с комплексными числами, например, *12iL* дает ошибку.

Константы являются довольно скучными, и мы больше не хотим тратить слова.

3.1.2. Просмотр символов

При создании новой переменной у нее должно быть имя, что дает возможность на нее сослаться, и у нее обычно есть значение. Само имя - символ. При оценке символа возвращается его значение. Позже мы объясним подробно, как определить значение, которое имеет символ.

В этом небольшом примере ***y*** - символ и его значение ***4***. Символ также является объектом ***R,*** но редко возникает необходимость иметь дело с символами непосредственно, кроме случаев “Программирование на языке” (глава 6 [Вычисления на языке]).

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

3.1.3. Вызов функции

Большинство вычислений, выполненных в ***R,*** включает оценку функций. Мы будем также называть это как *вызов* функции. Функции вызываются по имени со списком аргументов, разделенных запятыми.

> mean(1:10)

[1] 5.5

В этом примере функция *mean* (средняя) была вызвана с одним аргументом, вектором целых чисел от 1 до 10.

R содержит огромное число функций с различными результатами. Большинство используется для получения результата, который является объектом ***R,*** но некоторые используются для вспомогательных целей, например, функции печати и рисования.

Вызовы функции могут тегировать (или называть) аргументы, как в *plot(x, y, pch = 3),* аргументы без тегов известны как позиционные, так как функция должна отличить их значение от их последовательных позиций среди аргументов вызова, например, что ***x*** обозначает переменную абсциссы, а ***y*** ординату. Использование тегов/имен - очевидное удобство для функций с большим количеством дополнительных аргументов.

Специальный тип вызовов функции может появиться на левой стороне оператора присваивания как в:

> class(x) <- "foo"

Что действительно делает эта конструкция – это вызов функции *class<-* с исходным объектом и правой стороной. Эта функция выполняет модификацию объекта и возвращает результат, который затем сохраняется обратно в исходной переменной. По крайней мере, концептуально это то, что происходит. Прилагаются дополнительные усилия для исключения ненужного дублирования данных.

3.1.4. Операторы

R позволяет использование арифметических выражений с помощью операторов, подобных таковым из языка программирования ***C,*** например:

> 1 + 2

[1] 3

Используя круглые скобки, выражения можно сгруппировать с включением вызовов функций, и прямым присвоением переменным:

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

R содержит много операторов. Они перечислены в таблице ниже.

-	*Минус, мож ет быт ь унарным или бинарным*
+	*Плюс, мож ет быт ь унарным или бинарным*
!	*Унарное нет*
~	*Тильда, используемая для формул модели, мож ет быт ь или унарным или бинарным*
?	*Справка*
:	*Последоват ельност ь, двоичная (в формулах модели: взаимодейст вие)*
***	*Умнож ение бинарное*
/	*Деление бинарное*
^	*Возведение в ст епень бинарное*
%x%	*Специальные бинарные операт оры, x могут быт ь заменены любым допуст имым именем*
%%	*Модуль бинарный*

% / %	*Целочисленное деление, бинарное*
*% * %*	*Мат ричное произведение, бинарное*
%o%	*Внешнее произведение, бинарное*
%x%	*Кронекерово умнож ение, бинарное*
%in%	*Соот вет ст вие операт ора, бинарного (в формулах модели: гнездованное)*
<	*Меньше чем, бинарный*
>	*Больше чем, бинарный*
==	*Равно, бинарное*
> =	*Больше чем или равно, бинарное*
<=	*Меньше чем или равно, бинарное*
&	*And, бинарное, векторизовано*
&&	*And, бинарное, не векторизовано*
/	*Или, бинарное, векторизовано*
//	*Или, бинарное, не векторизовано*
<-	*Левое присвоение, бинарное*
->	*Правое присвоение, бинарное*
$	*Подмнож ест во списка, бинарное*

За исключением синтаксиса, нет никакой разницы между применением оператора и вызовом функции. Фактически, *x + y* может эквивалентно быть записано *‘+‘ (x, y)*. Заметим, что так как ‘+’ не является именем стандартной функции, то он должен быть заключен в кавычки.

R имеет дело со всем вектором данных за один раз, и большинство элементарных операторов и основных математических функций, например, *log* являются векторизованными (как обозначено в таблице выше). Это означает, что например, добавление двух векторов одинаковой длины создаст вектор, содержащий поэлементные суммы, неявно индексируя циклическое выполнение по вектору. Это применяется также к другим операторам как - *, и / так же как к структурам более высокой размерности. Заметим в особенности, что умножение двух матриц не производит обычное матричное произведение (оператор ***%*%*** существует с этой целью). Некоторые тонкости, касающиеся векторизованных операций, будут обсуждены в разделе 3.3 [Элементарные арифметические операции].

Для получения доступа к отдельным атомарным элементам вектора используется конструкция *x[i]:*

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

Для доступа к компоненте списка обычно используется *x$a* или *x[[i]]*.

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

Также можно использовать индексацию конструкций на правой стороне присвоения.

Подобно другим операторам, индексация, в действительности, выполняется функциями, и можно использовать *‘[‘(x, 2)* вместо *x[2]*.

Операции индексации ***R*** содержат много расширенных функций, которые описаны далее в разделе 3.4 (Индексация).

3.2. Управляющие структуры

Вычисление в ***R*** состоит в последовательной оценке *операт оров*. Операторы, такие как *x <-1:10* или *mean(y)*, могут быть разделены или точкой с запятой или новой строкой. Всякий раз, когда средству анализа предоставляют синтаксически полный оператор, этот оператор оценивается и возвращается значение. Результат оценки оператора может упоминаться как значение оператора. Значение всегда присваиваться символу.

Для разделения операторов могут использоваться и точки с запятой, и новые строки. Точка с запятой всегда указывает на конец оператора, в то время как новая строка может указывать на конец оператора. Если текущий оператор синтаксически не полный, то новые строки просто игнорируются средством анализа. Если сеанс является интерактивным, запрос изменяется с *'>'* на *'+'*.

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

Операторы могут группироваться использованием фигурных скобок *'{'* и *'}'*. Группу операторов иногда вызывают блоком. Отдельные операторы оцениваются при вводе новой строки в конце синтаксически полного оператора. Блоки не оцениваются, пока новая строка не вводится после закрывающей фигурной скобки. В оставшейся части этого раздела оператор ссылается на отдельный оператор или блок.

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

3.2.1. Оператор *if*

Оператор *if/else* условно оценивает два оператора. Существует условие, которое подлежит оценке, и если значение равно TRUE, то первый оператор оценивается; иначе оценивается второй оператор. Оператор *if/else* возвращает в качестве своего значения значение выбранного оператора. Формальный синтаксис таков:

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

Во-первых, оценивается *statement1* для получения *value1*. Если *value1* - логический вектор с первым элементом, равным TRUE, то оценивается *statement2*. Если первый элемент *value1* равен FALSE, то оценивается *statement3*. Если *value1* - числовой вектор, то оценивается *statement3*, когда первый элемент *value1* равен нулю, а иначе оценивается *statement2*. Используется только первый элемент *value1*. Все другие элементы игнорируются. Если у *value1* есть какой-либо тип кроме логического или числового вектора, то сигнализируется ошибка.

Можно использовать оператор *if/else* для исключения числовых проблем, таких как взятие логарифма отрицательного числа. Поскольку, оператор *if/else* такой же, как другие операторы, то можно присвоить его значение. Два примера ниже эквивалентны.

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

Выражение *else* является дополнительным. Допустим, имеется оператор *if(any(x <= 0)) x <-x [x <= 0].* Если оператор *if* не находится в блоке, то *else,* если присутствует, должен появиться на той же самой строке как конец *statement2*. Иначе новая строка в

конце *statement2* завершается и *if* рассматривается как синтаксически полный оператор подлежащий оценке. Простое решение состоит в использовании составного оператора, обернутого в фигурные скобки, помещая *else* в ту же самую строку как закрывающая фигурная скобка, которая отмечает конец оператора.

Оператор *if/else* может быть гнездован.

```
    if ( statement1  ) {
statement
2
    } else  if ( statement3  ) {
statement
4
    } else  if ( statement5  ) {
statement
6
    } else
statement
8
```

Один из четных операторов будет оценен и будет возвращено полученное значение. Если дополнительное выражение *else* опущено, и все четные операторы оцениваются к FALSE, то ни один оператор не будет оценен, и возвращается NULL.

Перечисленные нечетные операторы оцениваются по порядку до оценки TRUE, а затем оценивается присоединенный четный пронумерованный оператор. В этом примере будет только оценен *statement6*, если *statement1* будет FALSE, и *statement3* FALSE, и *statement5* TRUE. Нет ограничений числа разрешенных выражений *else if*.

3.2.2. Циклы

У ***R*** есть три оператора для явных повторных выполнений. Это: *for, while* и *repeat*. Две встроенных конструкции *next* и *break* обеспечивают дополнительное управление оценкой. Каждый из этих трех операторов возвращает значение последнего оцененного оператора. Возможно, хотя редко, присвоить результат одного из этих операторов символу. ***R*** обеспечивает другие функции для неявного цикличного выполнения, такие как *tapply, apply* и *lapply*. Кроме того, много операций, особенно арифметических, которые являются векторизованными, таким образом, отсутствует необходимость использовать цикл.

Существует два оператора, используемых для явного управления выполнением цикла. Это - *next* и *break*. Оператор *break* вызывает выход из самого внутреннего цикла, выполняемого в настоящий момент. Оператор *next* сразу вызывает переход к началу цикла. Затем выполняется следующая итерация цикла (при наличии). Операторы ниже *next* в текущем цикле не оценивается.

Значение, возвращенное оператором цикла, всегда равно NULL и возвращается не явно.

3.2.3. Оператор *repeat*

Оператор *repeat* вызывает повторную оценку, пока не будет вызван оператор *break*. Это означает, что следует быть осторожными при использовании *repeat* из-за опасности бесконечного цикла. Синтаксис цикла *repeat* выглядит как:

```
repeat  statement
```

При использовании *repeat* следующим оператором должен быть блок операторов. Следует, как выполнить некоторое вычисление, так и проверить выход из цикла, что обычно требует двух операторов.

3.2.4. Оператор while

Оператор *while* очень похож на оператор *repeat*. Синтаксис оператора цикла *while* следующий:

while (statement1) statement2

где оценивается *statement1,* и если его значение TRUE, то оценивается *statement2*. Этот процесс продолжается, пока *statement1* не оценивается как FALSE.

3.2.5. Оператор *for*

Синтаксис оператора цикла *for* следующий:

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

где *vector* может быть или вектором или списком. Для каждого элемента в *вект оре* имя переменной устанавливается к значению элементов этого вектора и оценивается *statement1*. Побочный эффект состоит в том, что имя переменной продолжает существовать после окончания цикла закончился, и у нее есть значение последнего элемента вектора, для которого был оценен цикл.

3.2.6. Оператор switch

С технической точки зрения *switch* (переключатель) является лишь еще одной функцией, но ее семантика близка к таковым из управления структурами других языков программирования.

Синтаксис:

switch (statement, list)

где элементы *list* (списка) можно назвать. Во-первых, оператор оценивается, и получаем результат, значение. Если значение - число между 1 и длиной списка, то оценивается соответствующий элемент списка (*list*), и возвращается результат. Если значение является слишком большим, или слишком малым, то возвращается NULL.

```
> 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
```

Если value (значение) является символьным вектором, то оценивается элемент '...' с именем, которое точно соответствует значению. Если не существует соответствия, то будет использован отдельный не названный аргумент в качестве умолчания. Если умолчание отсутствует, то возвращается NULL.

```
> y <- "fruit"
> switch(y, fruit = "banana", vegetable = "broccoli", "Neither")
[1]      "banana"
> y <- "meat"
> switch(y, fruit = "banana", vegetable = "broccoli", "Neither")
[1]      "Neither"
```

Обычное использование переключателя состоит в переходе согласно символьного значения одного из аргументов функции.

```
> 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 возвращает или значение оцененного оператора, или NULL, если отсутствует оценка оператор.

Для выбора из списка существующих альтернатив, переключатель, возможно, не лучший способ выбора для оценки. Часто лучше использовать *eval* и оператор подмножества *[[* непосредственно через *eval(x [[условие]])*.

3.3. Элементарные арифметические операции

В этом разделе обсуждаются тонкости правил, которые применяются к основным операциям подобным сложению или умножению двух векторов или матриц.

3.3.1. Правило рецикличности

Если попытаться сложить две структуры с различным числом элементов, то более короткая будет циклично преобразована к длине более длинной. Таким образом, если, например, сложить вектор *c(1,2,3)* с вектором с шестью элементами, то в действительности прибавите *c(1,2,3,1,2,3)*. Если длина более длинного вектора не кратна короткому вектору, то выдается предупреждение.

Начиная с ***R*** 1.4.0, любой арифметический оператор, включающий вектор нулевой длиной, даст результат нулевой длины.

Одно исключение состоит в том, что прибавляя векторы к матрицам, предупреждение не дается, если длины несовместимые.

3.3.2. Распространение имен

Распространение имен (первый побеждает, я думаю - также, если у него нет имени?? – первый **with names** побеждает, рецикличность вызывает укорочение до потери имени).

3.3.3. Атрибуты размерности

матрица+матрица – размерности должны соответствовать. *вектор+матрица*: сначала рецикличность, затем проверка на соответствие, если не соответствует, то ошибка.

3.3.4. Обработка NA

Отсутствующие значения в статистическом смысле - это переменные, значение которых не известно, имеющие значение NA. Их не следует путать со свойством *missing* для не предоставленного аргумента функции (см. раздел 4.1.2 [Аргументы]).

Хотя атомарные элементы вектора должны иметь одинаковый тип, но существует много типов значений NA. Имеется особенно важный случай для пользователя. Тип по умолчанию NA равен *logical*, если не преобразовано к некоторому другому типу, таким образом, появление отсутствующего значения может

инициировать логическую, а не числовую индексацию (см. раздел 3.4 [Индексация] для подробностей).

Числовые и логические вычисления с NA обычно возвращают NA. В случаях, где результатом операции было бы одинаковое для всех возможных значений, то можно взять NA, операция может вернуть это значение. В частности 'FALSE & NA' FALSE, 'TRUE | NA' TRUE. NA не равен любому другому значению или себе; тестирование на NA выполняется использованием *is.na*. Однако значение NA будет соответствовать другому значению NA соответственно.

Числовые вычисления, результат которых не определен, к примеру '*0/0*', производят значение NaN. Оно существует только в типе двойной длины и для реальных или мнимых частей комплексных типов. Предоставляется функция *is.nan* для проверки на NaN, *is.na* также возвращает TRUE для NaN. Преобразование NaN к логическому или целочисленному типу дает NA соответствующего типа, но преобразование к символу присваивает строке "NaN". Значение NaN не сравниваются, поэтому тесты на равенство, или сопоставление, включающее NaN, приведут к NA. Они расцениваются как соответствующие любому значению NaN (и никакому другому значению, даже не NA) с помощью *match*.

NA символьного типа, начиная с R1.5.0, отличается от строки "NA". При необходимости явно указать NA программисты должны использовать '*as.character (NA)*', а не "NA", или установить элементы к NA, используя *is.na <-.*

Начиная с R2.5.0 существуют константы *NA_integer_, NA_real_, NA_complex_ и NA_character_,* которые будут генерировать (в синтаксическом анализаторе) значение NA соответствующего типа, и будет использоваться в обратном синтаксическом анализаторе при отсутствии иной возможно идентифицировать тип NA (и опции управления попросят это).

Отсутствуют значения NA для *raw* (строк символов) векторов.

3.4. Индексирование

R содержит несколько конструкций, которые предоставляют доступ к отдельным элементам или подмножествам посредством индексных операций. В случае основных типов векторов можно получить доступ к ***i***-му элементу, используя *x[i]*, но также существует индексация списков, матриц и многомерных массивов. Есть несколько форм индексации в дополнение к индексации с отдельным целым числом. Индексация может использоваться и чтобы извлечь часть объекта и заменить части объекта (или прибавить части).

У R есть три основных оператора индексации с синтаксисом, показанных следующими примерами:

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

Для векторов и матриц форма *[[* редко используются, хотя у нее есть некоторые небольшие семантические отличия от формы *[* (например, она отбрасывает любые имена или атрибут имен размерностей, и что частное соответствие используется для символьных индексов). Когда индексируются многомерные структуры с единственным индексом, то *x[[i]]* или *x[i]* вернет ***i***-й последовательный элемент ***x***.

Для списков обычно используют *[[* для выбора любого отдельного элемента,

тогда как *[* возвращает список выбранных элементов.

Форма *[[* позволяет выбрать только единственный элемент, используя целочисленные или символьные индексы, тогда как *[* позволяет индексировать векторами. Заметим, хотя это для списка или другого рекурсивного объекта, индексирование может быть вектором, и каждый элемент вектора применен поочередно к списку, выбранному компоненту, выбранному компоненту того компонента, и так далее. Результат - все еще единственный элемент.

Форма с использованием *$* применяется к рекурсивным объектам, таким как списки и парные списки. Она допускает только литеральную символьную строку или символ как индекс. Таким образом, индекс не вычисляется: для случаев, где следует оценить выражение, чтобы найти индекс, используйте *x[[expr]]*. Если *$* применен к не рекурсивному объекту, то результат обычно всегда будет NULL: начиная с *R* 2.6.0 это является ошибкой.

3.4.1. Индексирование векторами

R предоставляет мощные конструкции с использованием векторов в качестве индексов. Сначала обсудим индексацию простых векторов. Для простоты предположим, что выражением является *x[i]*. Затем следующие возможности существуют для типа *i*.

- **Integer** (Целое число). Все элементы *i* должны иметь одинаковый знак. Если они положительны, то выбраны элементы *x* с этими индексами. Если *i* содержит отрицательные элементы, то выбраны все элементы кроме обозначенных.

  Если *i* положительно и превышает *length(x)*, то соответствующий выбор равен NA. Отрицательное значение *i* вне границ вызывает ошибку.

  Особый случай – индекс, равный нулю, который имеет пустые эффекты: x[0] является пустым вектором, и, тем не менее, включает нули среди положительных или отрицательных индексов, имеющих тот же самый эффект, как будто они были опущены.
- **Другое числовое**. Перед использованием значение нецелого числа преобразуется в целое число (усечением к нулю).
- **Logical**. Индекс *i* должен иметь одинаковую длину с *x*. Если он будет короче, то его элементы будут циклически преобразованы как обсуждено в разделе 3.3 [Элементарные арифметические операции]. Если он длиннее, то *x* концептуально расширен с помощью NA. Выбранное значение *x* – это те, для которых *i* равно TRUE.
- **Символ**. Строки в *i* сопоставляются против атрибута имен *x*, и используются получающиеся целые числа. Для *[[* и *$* используется частное соответствие, если точное соответствие перестало работать, таким образом, *x$aa* будет соответствовать *x$aabb*, если *x* не будет содержать компонент, названный "*aa*", и "*aabb*" - единственное имя, у которого есть префикс "*aa*". Для *[[* частным соответствием можно управлять через аргумент *exact*, установленный по умолчанию в NA, указывающий, что частное соответствие позволено, но предупреждается, когда это происходит. Установка *exact* в TRUE препятствует тому, чтобы частное соответствие произошло, значение FALSE позволяет это и не предупреждает. Заметим, что *[* всегда требует точного совпадения. Строка "" обрабатывается особенно: она не указывает ‘ни на какое имя’ и не соответствует элементу (даже на те, которые без имени). Заметим, что частное соответствие используется только при извлечении, но не при замене.
- **Factor**. Результат идентичен *x[as.integer (i)]*. Факторные уровни никогда не используются. Раз так хочется, то используйте *x[as.character(i)]* или

подобную конструкцию.

- **Empty**. Выражение *x[]* возвращает ***x***, но отбрасывает "несоответствующие" атрибуты из результата. Сохраняются только имена, как в многомерных размерностях массивов, так и атрибуты имен размерностей (dimnames).
- NULL. Это обрабатывается, как будто это было *interger(0)*.

Индекс с пропущенным значением (то есть NA) дает результат NA. Это правило применяется также к случаю логической индексации, то есть элементам ***x***, у которых есть селектор NA в ***i,*** включенный в результат, но их значением будет NA.

Заметим, однако, что есть различные режимы NA - литеральная константа имеет режим "*logical*", но она часто автоматически преобразовывается к другим типам. Одно воздействие этого состоит в том, что у *x[NA]* есть длина ***x***, но у *x[c(1,NA)]* есть длина 2. Это потому, что правила для логических индексов применяются в прежнем случае, но для целочисленных индексов в последнем.

Индексация с *[* также выполнит соответствующее подмножество любых атрибутов имен.

3.4.2. Индексирование матриц и массивов

Подмножества многомерных структур обычно следуют аналогичным правилам, как одномерная индексация для каждой переменной индекса с соответствующим компонентом *dimnames* взятие места names. Применяется несколько специальных правил, хотя:

Обычно, к структуре получают доступ, используя число индексов, соответствующих ее размерности. Однако также возможно использовать отдельный индекс в случае игнорирования атрибута размерности и имени размерности, а результат является эффективным для *c(m)[i]*. Заметим, что *m[1]* обычно очень отличается от *m[1,]*, или *m[,1]*.

Можно использовать матрицу целых чисел в качестве индексов. В этом случае число столбцов матрицы должно соответствовать число размерностей структуры, и результатом будет вектор с длиной как число строк матрицы. Следующий пример показывает, как извлечь элементы *m[1, 1]* и *m[2, 2]* в одной операции.

```
> 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
```

Отрицательные индексы запрещены в индексации матриц. Разрешены NA и нулевое значение: строки в индексе матрице, содержащей нуль, игнорируются, тогда как строки, содержащие NA, производят NA в результате.

Как в случае использования отдельного индекса, так и в индексации матриц индексации, используется атрибут *name*, если представлен, как имел бы одномерную структуру.

Если операция индексации вызывает результат одного из его расширений длиной единица в качестве выбора отдельного среза трехмерной матрицы (скажем) *m[2,,]*, то соответствующая размерность обычно отбрасывается из результата. Если результатом является одномерная структура, то получается вектор. Это иногда нежелательно и может быть отключено путем добавления ‘drope = FALSE’ к операции индексации. Заметим, что это дополнительный аргумент функции *[*, и не добавляется к счетчику индекса. Следовательно, корректным способом выбрать первую строку матрицы как *1 на n* матрицы является m[1, drop = FALSE]. Упущение отключить отбрасывающую опцию является частой причиной отказа в общих подпрограммах, где индексирование иногда, но не обычно, имеет длину один. Это правило все еще применяется к одномерному массиву, где любое подмножество даст векторный результат, если ‘drop = FALSE’ не будет использоваться.

Заметим, что векторы отличны от одномерных массивов в этом, последние имеют атрибуты размерности и имени размерности (dimnames) (обе из длины один). Одномерные массивы не легко получить из операций подмножества, но они могут быть созданы явно и возвращены таблицей. Это иногда полезно, потому что элементы списка имен размерностей (dimnames) можно самостоятельно назвать, что не имеет места для атрибута имен.

Некоторые операции, такие как m[FALSE], приводят к структурам, у которых размерность имеет нулевое расширение. Обычно ***R*** пробует обработать эти структуры подобающим образом.

3.4.3. Индексирование других структур

Оператор *[* является универсальной функцией, которая позволяет добавлять методы класса, и операторы ***$*** и *[[* аналогичны. Таким образом, можно иметь определяемые пользователем операции индексации для любой структуры. Такая функция, скажем *[.foo*, вызывается с набором аргументов, первым из которых является индексируемая структура, а остальные - индексы. В случае ***$*** индексный аргумент имеет режим "символ", используя форму *x$"abc"*. Важно знать, что методы класса не обязательно ведут себя таким же образом как основные методы, например, относительно частного соответствия.

Самый важный пример метода класса для *[* является использование для фреймов данных. Здесь это подробно не описывается (см. страницу справки для *[.data.frame)*, но в общих чертах, если два индекса предоставлены (даже, если один из них пуст), то это создает индексацию, подобную матрице, для структуры, которая в основном является списком векторов одинаковой длины. Если предоставлен отдельный индекс, то это интерпретируется как индексация списка столбцов, и в этом случае аргумент отбрасывания игнорируется с предупреждением.

Основные операторы ***$*** и *[[* могут быть применены к среде окружения. Позволены только символьные индексы, и никакое частное соответствие не делается.

3.4.4. Присвоение подмножества

Присвоение подмножеству структуры является частным случаем общего механизма для сложного присвоения:

```
x[3:5] <- 13:15
```

Результат этой команды соответствует следующему выполнению:

```
‘*tmp*‘ <- x
x <- "[<-"(‘*tmp*‘, 3:5, value=13:15)
rm(‘*tmp*‘)
```

Заметим, что индекс сначала преобразован в числовой индекс, а затем элементы заменены последовательно вдоль числового индекса, как будто использовался для цикла *for*. Любая существующая переменная с именем *‘*tmp*‘* будет перезаписана и

удалена, и это имя переменной не должно использоваться в коде.

Одинаковый механизм может быть применен к функциям кроме *[*. Заменяющая функция имеет одинаковое имя на вставленную <-. Ее последним аргументом, который нужно вызвать значением, является новое значение, подлежащее присвоению. Например:

```
names(x) <- c("a","b")
```

эквивалентно

```
‘*tmp*‘ <- x
x <- "names<-"(‘*tmp*‘, value=c("a","b"))
rm(‘*tmp*‘)
```

Гнездование сложных присвоений оценивается рекурсивно:

```
names(x)[3] <- "Three"
```

эквивалентно

```
‘*tmp*‘ <- x
x <- "names<-"(‘*tmp*‘, value="[<-"(names(‘*tmp*‘), 3, value="Three"))
rm(‘*tmp*‘)
```

Также разрешены сложные присвоения в обертке окружающей среды (используя <<-):

```
names(x)[3] <<- "Three"
```

эквивалентно:

```
‘*tmp*‘ <<- get(x, envir=parent.env(), inherits=TRUE)
names(‘*tmp*‘)[3] <- "Three"
x <<- ‘*tmp*‘
rm(‘*tmp*‘)
```

а также для

```
‘*tmp*‘ <- get(x,envir=parent.env(), inherits=TRUE)
x <<- "names<-"(‘*tmp*‘, value="[<-"(names(‘*tmp*‘), 3, value="Three"))
rm(‘*tmp*‘)
```

Только целевая переменная оценивается в обертке окружающей среды, итак:

```
e<-c(a=1,b=2)
i<-1 local({
    e <- c(A=10,B=11)
    i <-2
    e[i] <<- e[i]+1
})
```

использует локальное значение ***i*** и на LHS и на RHS, и локальном значении ***e*** на RHS супер оператора присваивания. Это устанавливает ***e*** во внешней среде к:

```
a   b
1   12
```

Таким образом, сверх присвоение эквивалентно этим четырем строкам:

```
‘*tmp*‘ <- get(e, envir=parent.env(), inherits=TRUE)
‘*tmp*‘[i] <- e[i]+1
e <<- ‘*tmp*‘
rm(‘*tmp*‘)
```

Подобно:

```
x[is.na(x)] <<- 0
```

эквивалентно:

```
‘*tmp*‘ <- get(x,envir=parent.env(), inherits=TRUE)
```

```
‘*tmp*‘[is.na(x)] <- 0
x <<- ‘*tmp*‘
rm(‘*tmp*‘)
```

и не эквивалентно для:

```
‘*tmp*‘ <- get(x,envir=parent.env(), inherits=TRUE)
‘*tmp*‘[is.na(‘*tmp*‘)] <- 0
x <<- ‘*tmp*‘
rm(‘*tmp*‘)
```

Эти две интерпретации варианта отличаются, только если есть также локальная переменная ***x.*** Это - хорошая идея исключить локальную переменную с тем же самым именем как целевая переменная сверх присвоения. Поскольку этот случай был обработан неправильно в версиях 1.9.1 и ранее, то не должно быть серьезной потребности в таком коде.

3.5. Область действия переменных

Почти у каждого языка программирования есть ряд правил области действия переменных, что позволяет использовать одно имя для различных объектов. Это позволяет, например, локальной переменной в функции иметь одинаковое имя с глобальным объектом.

R использует *лексическую модель* определения контекста, подобную таким языкам как Паскаль. Однако, ***R*** - *язык функционального программирования* и позволяет динамическое создание и манипулирование функциями и объектами языка, и имеет дополнительные функции, отражающие этот факт.

3.5.1. Глобальная окружающая среда

Глобальная окружающая среда - корень пользовательской рабочей области. Операторы присвоения в командной строке заставят соответствующий объект принадлежать глобальной окружающей среде. Ее обернутая окружающая среда - следующая окружающая среда на пути поиска, и так далее назад к пустой окружающей среде, которая является оберткой основной окружающей среды.

3.5.2. Лексическая окружающая среда

Каждый вызов функции создает *фрейм,* который содержит локальные переменные, создаваемые в функции, и оцененные в окружающей среде, которая в комбинации создает новую окружающую среду.

Заметим терминологию: фрейм является набором переменных, окружающая среда является вложением фреймов (или эквивалентно: самый внутренний фрейм плюс обертка окружающей среды).

Окружающие среды могут быть присвоены переменным или содержаться в других объектах. Однако Заметим, что они не стандартные объекты - в частности они не копируются при присвоении.

Обертка (режим "функция") объекта будет содержать окружающую среду, в которой она создается как часть ее определения. По умолчанию. Окружающей средой можно управлять, используя *environment* <-. При последующем вызове функции создается как обертка ее оцененная окружающая среда с оберткой. Заметим, что это не обязательно окружающая среда вызывающей стороны!

Таким образом, когда переменную требуют в функции, она сначала разыскивается в окружающей среде оценки, затем в обертке, обертке обертки, и т.д.; при достижении глобальной окружающей среда или окружающей среды пакета поиск продолжается вверх по пути поиска в окружающей среде основного (base) пакета. Если переменная не будет найдена там, то поиск продолжится далее в пустой окружающей средой, а затем прекращается.

3.5.3. Стек вызова

При каждом вызове функции создается новый фрейм оценки. В любой момент времени во время вычисления в настоящий момент активная окружающая среда доступна через *стек вызова*. При каждом вызове функции внутренне создается специальная конструкция, называемая контекст, и помещается в список контекстов. При завершении функцией оценки своего контекста ее контекст удаляется из стека вызова.

Создание переменных, определивших выше доступный стек вызова, назван динамическим контекстом. Связывание для переменной затем решается самым последним (на время) определением переменной. Это противоречит правилам области действия данных по умолчанию в ***R,*** которые используют привязку в окружающей среде, в которой функция была определена (лексический контекст). Некоторые функции, особенно те, которые используют и управляют формулами модели, должны имитировать динамический контекст, непосредственно получая доступ к стеку вызова.

Доступ к стеку вызова обеспечен через семейство функций, у которых есть имена, которые начинаются с *'sys'..*, и они перечислены кратко ниже:

sys.call	Получает вызов для определенного контекста.
sys.frame	Получает оценку фрейма для активного контекста.
sys.nframe	Получает оценку фрейма для определенного контекста.
Gsys.function	Получает вызванную функцию в определенном контексте.
sys.parent	Получает родителя вызова текущей функции
sys.calls	Получает вызовы для всех активных контекстов
sys.frames	Получает оценку фреймов для всех активных контекстов
sys.parents	Получает числовую метку для всех активных контекстов.
sys.on.exit	Устанавливает функцию, подлежащую выполнению, при запуске указанного контекста.
sys.status	Вызовы *sys.frames, sys.parents and sys.calls.*
parent.frame	Получает фрейм оценки для указанного родительского контекста.

3.5.4. Путь поиска

В дополнение к структуре окружающей среды оценки у ***R*** есть путь поиска окружающей среды, в которой ищутся переменные, не найденные в другом месте. Используется для двух случаев: пакеты функций и присоединенные пользовательские данные.

Первый элемент пути поиска – это глобальная окружающая среда, а последний – это основной (*base*) пакет. Окружающая среда *Autoloads* (Автозагрузок) используется для содержания объектов прокси, которые могут быть загружены по требованию. Другая окружающая среда вставлена в использование пути *attach* или *library*.

Пакеты с *пространством имен* имеют разные пути поиска. Когда поиск объекта ***R*** начат с объекта в таком пакете, ищется сначала сам пакет, затем он импортируется, затем основное пространство имен и, наконец, глобальная окружающая среда, и остальная часть регулярного пути поиска. Эффект состоит в том, что ссылки на другие объекты в этом же пакете будут разрешены в пакете, и объекты не могут быть замаскированы объектами с этим же именем в глобальной окружающей среде или в других пакетах.

4. Функции

4.1. Написание функций

Хотя ***R*** может быть очень полезным как инструмент анализа данных, большинство пользователей очень быстро приступает к написанию собственных функций. Это - одно из реальных преимуществ ***R.*** Пользователи могут программировать функции, и они, при необходимости, могут изменить функции уровня системы на функции более предпочтительные для пользователя.

R также предоставляет инструменты, которые облегчают документировать любые созданные функции. См. раздел “Написание документации R” в Написании расширений ***R.***

4.1.1. Синтаксис и примеры

Синтаксис написания функции выглядит как:

function (arglist) body

Первый компонент объявления функции - ключевое слово *function*, которое указывает ***R,*** что создается функция.

Список аргументов - список разделенных запятой значений формальных аргументов. Формальный аргумент может быть символом, оператором формы ‘*symbol = expression*’, или специальный формальный аргумент ‘...’.

Тело может быть любым допустимым выражением ***R.*** Обычно, тело – это группа выражений в фигурных скобках (‘*{*’ и ‘*}*’).

Обычно функции присвоены символам, но это не обязательно. Значение, возвращенное вызовом *function*, является функцией. Если имя ей не дано, то она упоминается как анонимная функция. Анонимные функции наиболее часто используют в качестве аргументов другие функции, такие как семейство *apply* или *outer.*

Вот простая функция: *echo <- function(x) print(x).* Таким образом, *echo* является функцией, у которой единственный аргумент и при вызове *echo* она печатает свой аргумент.

4.1.2. Аргументы

Формальные аргументы функции определяют переменные, значения которых будут предоставлены при вызове функции. Имена этих аргументов могут использоваться в пределах тела функции, где они получают значение, предоставленное во время вызова функции.

Можно указать значения по умолчанию для аргументов, используя специальную форму ‘*name = expression*’. В этом случае, если пользователь не будет указывать значение для аргумента и функция будет вызвана, то выражение будет связано с соответствующим символом. Если значение необходимо, то выражение оценивается во фрейме оценки функции.

Поведение по умолчанию также можно указать путем использования функции *missing*. Когда *missing* вызывается с именем формального аргумента, то возвращает TRUE, если формальный аргумент не соответствовал какому-либо фактическому аргументу и не был впоследствии изменен в теле функции. Аргумент, который является *missing*, таким образом, получит свое значение по умолчанию, или любое. Функция *missing* не вызывает оценки аргумента.

Специальный тип аргумента ‘...’ может содержать любое число предоставленных аргументов. Он используется для множества целей. Он позволяет писать функцию, которая берет произвольное число аргументов. Это может использоваться для перемещения некоторых аргументов в промежуточную функцию, которые затем могут быть извлечены функциями, вызванными впоследствии.

4.2. Функции как объекты

Функции являются объектами первого класса в ***R.*** Они могут использоваться где угодно при необходимости в объекте ***R***. В частности их можно передать как аргументы функциям и возвратить как значение функций. См. раздел 2.1.5 [Функции-объекты] для подробностей.

4.3 . Оценка

4.3.1. Окружающая среда оценки

При вызове функции создается новый фрейм оценки. В этом фрейме формальные аргументы сопоставляются предоставленным аргументам согласно правилам, изложенным в разделе 4.3.2 [Соответствие аргументов]. Операторы в теле функции последовательно оцениваются в этом фрейме окружающей среды.

Обворачивающий фрейм фрейма оценки является фреймом окружающей среды, связанной с вызываемой функцией. Это может отличаться от ***S***. Хотя у многих функций имеется *GlobalEnv* в качестве их окружающей среды, что не всегда верно, и у функций, определяемых в пакетах с пространствами имен (обычно) есть пространство имен пакета как их окружающая среда.

4.3.2. Соответствие аргумента

Этот подраздел применяется к оберткам, а не к примитивным функциям. Последние обычно игнорируют теги и выполняют позиционное соответствие, но следует проконсультироваться с их справкой об исключениях, которые включают *log, round, signif, rep* и *seq.int*.

Вначале оценка функций состоит в сопоставлении формальных и фактических или предоставленных аргументов. Это сделано в трех шаговом процессе:

1. **Точное соответствие на тегах.** Для каждого именованного предоставленного аргумента ищется список формальных аргументов с элементом, имя которого соответствует точно. Является ошибкой иметь соответствие одного и того же формального аргумента нескольким реально существующим или наоборот.

2. **Частное соответствие на тегах.** Каждый остающийся именованный предоставленный аргумент по сравнению с остающимися формальными аргументами, использует частное соответствие. Если имя предоставленного аргумента соответствует точно с первой частью формального аргумента, то эти два аргумента считаются соответствующими. Является ошибкой иметь много частных соответствий. Заметим это, если *f<- function(fumble, fooey) fbody, f (f = 1, fo = 2)* недопустима, даже при том, что 2-ой фактический аргумент только соответствует *fooey.* f (f = 1, fooey = 2) *является* законным, хотя второй аргумент соответствует точно и удален из рассмотрения для частного соответствия. Если формальные аргументы содержат '...' тогда частное соответствие только применимо к аргументам, которые предшествуют ему.

3. **Позиционное соответствие.** Любые несогласованные формальные аргументы ограничены *неназванным*и предоставленным аргументам в следующем порядке. Если будет '...' аргумент, то он приведет в рабочее состояние остающиеся аргументы, тегированные или нет.

Если какие-либо аргументы остаются несогласованными, то объявляется ошибка.

Соответствие аргументов обеспечивается функциями *match.arg, match.call* и *match.fun*. Доступ к частному алгоритму соответствия, используемому ***R,*** через *pmatch.*

4.3.3. Оценка аргументов

Одним из самых важных положений в оценке аргументов функции состоит в том, что предоставленные аргументы и аргументы по умолчанию обрабатываются по-

разному. Предоставленные аргументы функции оценены во фрейме оценки функции вызова. Аргументы по умолчанию функции оценены во фрейме оценки функции.

Семантика вызова функции с аргументами в ***R*** является вызовом по значению. Вообще, предоставленные аргументы ведут себя так, как будто они являются локальными переменными, инициализированными с предоставленным значением и с именем соответствующего формального аргумента. Изменение значения предоставленного аргумента в пределах функции не будет влиять на значение переменной во фрейме вызова.

У R есть форма отложенных вычислений аргументов функции. Аргументы не оцениваются, пока отсутствует необходимость. Важно понять, что в некоторых случаях аргумент никогда не будет оцениваться. Таким образом, плохим стилем является использование аргументов функций для вызова побочных эффектов. В то время как в ***C*** распространено использование формы *foo (x = y)*, чтобы вызвать *foo* со значением ***y*** и одновременно присвоить значение ***y*** к ***x***, то такой стиль не должен использоваться в ***R.*** Нет никакой гарантии, что аргумент когда-либо будет оценен, и, следовательно, присвоение, возможно, не будет выполнено.

Также стоит заметить, что воздействие *foo (x <-y)* при оценке аргумента состоит в изменении значения ***x*** в окружающей среде вызова, а не в окружающей среде оценки *foo.*

Можно получить доступ к фактическому (не по умолчанию) выражению, используемому в качестве аргументов внутри функции. Механизм реализован через обещания. При оценке функции фактическое выражение, используемое в качестве аргумента, сохраняется в обещании вместе с указателем на окружающую среду функции откуда она была вызвана. Когда (если) аргумент оценен, сохраненное выражение оценивается в окружающей среде, из которой была вызвана функция. Так как используется только указатель на окружающую среду, любые изменения, произведенные в той окружающей среде, будут действительны во время этой оценки. Затем также сохранится полученное значение в отдельном месте в обещании. Последующие оценки получают эту хранимую сумму (вторая оценка не выполняется). Также возможен доступ к неоцененному выражению, использованием *substitute*.

При вызове функции каждый формальный аргумент приписан к обещанию в локальной окружающей среде вызова со слотом выражения, содержащим фактический аргумент (если он существует), и слот окружающей среды, содержащий окружающую среду вызывающей стороны. Если отсутствует фактический аргумент для формального аргумента в данном вызове, и существует выражение по умолчанию, то он так же присваивается слоту выражения формального аргумента, но с окружающей средой, установленной к локальной окружающей среде.

Процесс заполнения обещания значениями слота путем оценки содержания выражения слота в окружающей среде обещаний называют *принуждением* обещания. Обещание только однажды будет принуждено, контент значения слота будет непосредственно использоваться позже.

Обещание является принудительным, когда его значение необходимо. Это обычно происходит во внутренних функциях, но обещание может также быть принудительным прямой оценкой обещания непосредственно. Это иногда полезно, когда выражение по умолчанию зависит от значения другого формального аргумента или другой переменной в окружении. Это замечено в следующем примере, где одинокая *label* гарантирует, что метка будет основана на значении ***x*** прежде, чем она будет изменена в следующей строке.

```
function(x, label = deparse(x)) {
label
```

```
x <- x + 1 print(label)
}
```

Слот выражения обещания может самостоятельно включать другие обещания. Это происходит всякий раз, когда неоцененный аргумент передают как аргумент другой функции. При принуждении обещания другие обещания в его выражении также будут принуждены рекурсивно из-за их оценки.

4.3.4. Область действия

Контекст или правила области действия просто являются набором правил, используемых средством анализа для поиска значения символа. У каждого машинного языка есть ряд таких правил. В ***R*** правила довольно просты, но в них существуют механизмы для изменения обычности, или правил по умолчанию.

R придерживается ряда правил, которые называют *лексическим контекстом*. Это означает реальное связывание переменных при создании выражения, и обычно обеспечивают значения для любых несвязанных символов в выражении.

Большинство интересных свойств области действия связано с оценкой функций, и мы сосредоточимся на этой проблеме. Символ может быть или связанным, или несвязанным. Все формальные аргументы функции обеспечивают связанные символы в теле функции. Любые другие символы в теле функции - или локальные переменные или несвязанные переменные. Локальная переменная - та, которая определена в пределах функции. Поскольку у ***R*** нет никакого формального определения переменных, они просто используются при необходимости, и может быть сложно определить, локальна ли переменная или нет. Локальные переменные должны сначала быть определены, это обычно делается при наличии присвоения от них слева.

Во время процесса оценки при обнаружении несвязанного символа ***R*** пытается найти значение для него. Правила области действия определяют выполнение этого процесса. В ***R*** сначала ищется окружающая среда функции, затем ее обертка и так далее, пока не будет достигнута глобальная окружающая среда.

A simple example:

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

В ***S*** из-за различных правил области действия данных возникнет ошибка, указывающая, что ***y*** не найден, если переменной ***y*** нет в рабочей области при использовании ее значения.

Довольно интересный вопрос состоит в том, что происходит при оценке ***h***. Для описания этого потребуется еще несколько определений. В пределах тела функции переменные могут быть связанными, локальными или несвязанными. Связанные переменные - те, которые соответствуют формальным аргументам функции. Локальные переменные - те, которые были созданы или определены в пределах тела функции. Несвязанные переменные - те, которые являются ни локальными, ни связанными. После оценки тела функции отсутствуют проблемы определения значений для локальных переменных или для связанных переменных. Правила обзора данных определяют поиск значения для несвязанных переменных в языке.

При оценке *h(3)* видим, что его тело является телом ***g***. В пределах этого тела ***x*** связан с формальным аргументом, а ***y*** является несвязанным. На языке с лексическим

контекстом, ***x*** будет присоединен со значением 3, а ***y*** со значением 10 локальной ***f***, итак *h(3)* должен вернуть значение 13. В ***R*** это действительно происходит.

В ***S*** из-за различных правил обзора данных получаем ошибку с указанием, что ***y*** не найден, если нет переменной ***y*** в рабочей области при использовании ее значения.

5. Объектно-ориентированное программирование

Объектно-ориентированное программирование – это стиль программирования, который стал популярным в последние годы. Большая часть популярности определяется облегчением написания и поддержки сложных систем. Делается это через несколько различных механизмов.

Центральным к любому объектно-ориентированному языку является понятие класса и методов. *Класс* (*class*) является определением объекта. Обычно класс содержит несколько *слот ов* (slot), которые используются для содержания информации о классе. Объект на языке должен быть экземпляром некоторого класса. Программирование основано на объектах или экземплярах классов.

Вычисления выполнены через *мет оды (method)*. Методы в основном являются функциями, которые специализированы выполнения определенных вычислений на объектах, обычно определенного класса. Именно это делает язык объектно-ориентированным. В ***R*** используются *универсальные функции* для определения соответствующего метода. Универсальная функция ответственна за определяющие класс ее аргументы и использует эту информацию для выбора соответствующего метода.

Другая особенность большинства объектно-ориентированных языков - понятие наследования. В большинстве запрограммированных задач обычно существует много объектов, которые связаны между собой. Программирование значительно упрощается при возможности повторного использования некоторых компонент.

Если класс наследовался от другого класса тогда обычно, что он получает все слоты в родительском классе и может расширить их, прибавляя новые слоты. При посылке метода (через универсальные функции), если метод для класса не существует, он разыскивается метод у родителя.

В этой главе обсуждается реализация этой общей стратегии в ***R*** и обсуждаются некоторые из ограничений в пределах текущего проекта. Одно из преимуществ, которые передается большинству объектных систем, является большая непротиворечивость. Это достигнуто через правила, которые проверены компилятором или интерпретатором. К сожалению, из-за способа, которым объект системы включен в ***R,*** данное преимущество не возникает. Пользователей предостерегают использовать объекта систему прямым способом. В то время как возможно выполнить некоторые довольно интересные подвиги, они имеют тенденцию приводить к запутанному коду и могут зависеть от деталей реализации, которые не будут продвинуты.

Самое широкое использование объектно-ориентированного программирования в ***R*** осуществляется через методы печати (*print*), методы сводок (*summary*) и методы рисования (*plot*). Эти методы предоставляют одинаковый вызов универсальной функции, скажем рисования, который зависит от типа его аргумента и вызывает функцию рисования, которая является определенной для полученных данных.

Для прояснения концепции рассмотрим реализацию небольшой системы, разработанной для обучения студентов теории вероятности. В этой системе объекты - функции вероятности, а рассматриваемые методы предназначаются для поиска моментов и рисования. Вероятности могут всегда представляться с точки зрения кумулятивной функции распределения, но могут часто представляться другими

способами. Например, как плотность, когда она существует или как производная функции момента при ее существовании.

5.1. Определение

Вместо законченной объектно-ориентированной системы, у R есть система класса и механизм для посылки объекта, основанного на классе. Механизм посылки для интерпретируемого кода полагается на четыре специальных объекта, которые хранятся во фрейме оценки. Этими специальными объектами являются *.Generic, .Class, .Method* и *.Group*. Существует отдельный механизм посылки, используемый для внутренних функций и типов, которые будут обсуждены в другом месте.

Класс системы определяется через атрибут *class*. Этот атрибут является вектором символов имен классов. Итак, для создания объекта класса "*foo*" просто присоединяется атрибут класса со строкой ‘"*foo*"’ в нем. Таким образом, фактически что-либо может быть возвращено объекту класса *"foo*".

Системный объект использует *универсальные функции* через две функции посылки: *UseMethod* и *NextMethod.* Типичное использование системного объекта должно начинаться вызовом универсальной функции.

Она типично очень простая функция и состоит из одной строки кода. Системная функция *mean* как раз является такой функцией:

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

При вызове *mean*, у нее может быть любое число аргументов, но ее первый аргумент является особенным, и класс этого первого аргумента используется для определения необходимого метода, подлежащего вызову. Переменная *.Class* установлена в атрибут класса ***x***, *.Generic* установлен в строку *«mean»*, и поиск сделан для вызова корректного метода. Игнорируются атрибуты класса любых других аргументов *mean*.

Предположим, что у ***x*** был атрибут класса, который содержал "*foo*" и "*bar*" в этом порядке. В этом случае ***R*** сначала искал бы функцию, названную *mean.foo* и если б не нашел, то затем искал бы функцию *mean.bar*, а в случае неудачности этого поиска выполнил бы заключительный поиск *mean.default*. Если и последний поиск неудачен, то ***R*** выдаст ошибку. Хорошей идеей является указание метода по умолчанию. Заметим, что функции *mean.foo* и т.д. упомянуты, в этом контексте, как методы.

NextMethod обеспечивает другой механизм для поставки. У функции вызов *NextMethod* может быть в любом месте. Определение подлежащего вызову метода базируется, прежде всего, на текущем значении *.Class* и *.Generic*. Это несколько проблематично, так как метод является обычной функцией, и пользователи могут вызвать его непосредственно. Если так сделать, то будут отсутствовать значения для *.Generic* или *.Class*.

Если метод вызван непосредственно, и он содержит вызов *NextMethod,* то используется первый аргумент *NextMethod* для определения универсальной функции. Печатается ошибка при отсутствии этого аргумента; поэтому хорошей идеей является всегда предоставлять такой аргумент.

В случае непосредственного вызова метода атрибут класса первого аргумента метода используется в качестве значения *.Class*.

Сами методы используют *NextMethod* для обеспечения формы наследования. Обычно определенный метод выполняет несколько операций для установки данныхе, а затем вызывает следующий соответствующий метод посредством вызова *NextMethod*.

Рассмотрите следующий простой пример. Точка в двумерном Евклидовом пространстве может указываться ее Декартовыми ***(x-y)*** координатами или полярными (r-тета) координатами. Следовательно, для хранения информации о расположении точки можно определить два класса, "*xypoint*" и "*rthetapoint*". Все '*xypoint*' структуры данных - списки с ***x***-компонентом и ***y***-компонентом. Все '*rthetapoint*' объекты - списки с ***r***-компонентом и компонентом теты.

Теперь, предположим, что желательно получить ***x***-позицию любого типа объекта. Это легко можно сделать через универсальные функции. Определяем универсальную функцию *xpos* следующим образом.

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

Теперь можно определить метод:

xpos.xypoint <- function(x) x$x

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

Пользователь просто вызывает функцию *xpos* с любым представлением в качестве аргумента. Внутренний метод поставки находит класс объекта и вызывает соответствующие методы.

Довольно легко прибавить другие представления. Отсутствует необходимость написания новых универсальных функций, а только методы. Это облегчает добавление к существующей системе, так как пользователь ответственен лишь за контакт с новым представлением, а не с любым из существующих представлений.

Объем использования этой методологии к предоставленной специализированной печати для объектов различных типов равен приблизительно 40 методам для *print*.

5.2. Наследование

У атрибута класса объекта может быть несколько элементов. Когда родовую функцию вызывают, первое наследование, главным образом, обработано через *NextMethod*. *NextMethod* решает, что метод, в настоящий момент, будучи оцененным, находит следующий класс от

FIXME: здесь в оригинале что-то пропущено

5.3. Поставка метода

Универсальные функции должны иметь единственный оператор. Они должны обычно иметь вид: *foo <-function(x...) UseMethod ("foo", x).* При вызове *UseMethod* определяется соответствующий метод, а затем вызывается этот метод с теми же самыми аргументами в том же самом порядке как универсальный вызов, как будто был вызван непосредственно метод.

Для определения корректного метод получают и используют атрибут класса первого аргумента для поиска корректного метода. Имя универсальной функции объединено с первым элементом атрибута класса в форму *generic.class*, и разыскивается функция с этим именем. Если функция найдена, то она используется. Если функция отсутствует, то используется второй элемент атрибута класса до исчерпания всех элементов атрибута класса. Если метод не был найден, то в этой точке используется метод *generic.default*. Если у первого аргумента универсальной функции отсутствует атрибута класса, то используется *generic.default*. Начиная с введения пространств имен методы, возможно, не доступны своими именами (то есть получение ("*generic.class*") может перестать работать), но они будут доступны через *getS3method("generic","class")*.

У любого объекта может быть атрибут класса. У этого атрибута может быть любое число элементов. Каждый из них - строка, которая определяет класс. При вызове универсальной функции исследуется класс ее первого аргумента.

5.4. UseMethod

UseMethod - специальная функция и ведет себя по-другому от других вызовов функции. Синтаксисом вызова этого является *UseMethod(generic, object),* где *generic* является именем универсальной функции, *object* является объектом, который используется для определения выбора метода.. *UseMethod* можно вызвать только из тела функции.

UseMethod изменяет модель оценки двумя способами. Во-первых, при вызове определяется следующий метод (функция), подлежащий вызову. Затем вызывается функция с использование текущей оценки окружающей среды; этот процесс будет описан коротко. Второй путь, при котором *UseMethod* изменяет окружающую среду оценки, состоит в том, что не возвращается управление функции вызова. Это означает, гарантированное исключение выполнения любых операторов после вызова *UseMethod*.

При вызове *UseMethod* универсальная функция является указанным значением в вызове *UseMethod*. Объект для поставки является либо предоставленным вторым аргументом, либо первым аргументом текущей функции. Класс аргумента определен, и первый элемент этого объединен с именем универсальности для определения соответствующего метода. Так, если у универсальности было имя *foo*, и класс объекта - "*bar*", то ***R*** будет искать метод с именем *foo.bar*. Если такой метод не существует, то используется выше описанный механизм наследования для определения местоположения соответствующего метода.

После определения метода ***R*** вызывает его специальным способом. Вместо создания новой окружающей среды оценки ***R*** использует окружающую среду текущего вызова функции (вызов универсальности). Любые присвоения или оценки, которые были сделаны перед вызовом *UseMethod*, будут в действовать. Аргументы, которые использовались в вызове универсальности, являются сверх соответствующими к формальным аргументам выбранного метода.

При вызове метода он вызывается с аргументами, которые одинаковы по числу и именам как в вызове универсальной функции. Они соответствуют аргументам метода согласно стандартным правилам ***R*** для соответствия аргумента. Однако объект, то есть первый аргумент оценивается.

Вызов *UseMethod* имеет эффект размещения некоторых специальных объектов во фрейме оценки. Это - *.Class., Generic* и *.Method*. Эти специальные объекты обычно используются ***R*** для обработки поставки метода и наследования. *.Class* - класс объекта *.Generic* - имя универсальной функции и *.Method* - имя вызываемого метода в настоящий момент. При вызове метода через один из внутренних интерфейсов может оказаться еще один объект именем *.Group*. Это будет описано в разделе раздела 5.6 [Групповые методы]. После начального вызова *UseMethod* эти специальные переменные, а не объект непосредственно, управляют выбором последующих методов.

Затем тело метода оценивается стандартным способом. Специальный просмотр переменных в теле следует правилам для метода. Так, если у метода есть присоединенная окружающая среда, то она используется. В действительности мы заменили вызов универсальной функции вызовом метода. Любые локальные присвоения во фрейме универсальной функции будут перенесены в вызов метода. Использование этой особенности обескураживает. Важно понять, что управление никогда не будет возвращено универсальной функции и, следовательно, не будут выполняться любые выражения после вызова *UseMethod*.

Любые аргументы универсальной функции, которые были оценены до вызова *UseMethod*, остаются оцененными.

Аргументы в вызове универсальной функции повторно сопоставляются с аргументами для метод, используя стандартный механизм соответствия аргумента.

Первый аргумент, то есть объект, будет оценен.

Если первый аргумент *UseMethod* не предоставлен, то предполагается, что это имя текущей функции. Если два аргумента предоставлены *UseMethod,* то первым является имя метода, и вторым предполагается объект, подлежащий поставке. Это оценивается таким образом, чтобы можно было определить необходимый метод. В этом случае первый аргумент в вызове универсальной функции не оценивается и отбрасывается. Отсутствуют способы изменения других аргументов в вызове метода, и они остаются в виде, как они были при вызове универсальной функции. В этом отличие от *NextMethod*, где аргументы в вызове следующего метода могут быть изменены.

5.5. NextMethod

NextMethod используется для обеспечения простого механизма наследования.

Методы, вызванные в результате вызова *NextMethod*, ведут себя так, как будто они были вызваны из предыдущего метода. Аргументы наследованному методу находятся в том же самом порядке и имеют те же самые имена как вызов текущего метода. Это означает, что они одинаковы по отношению к вызову универсальной функции. Однако, выражения для аргументов - имена соответствующих формальных аргументов текущего метода. Таким образом, у аргументов будет значение, которое соответствует их значению во время вызова *NextMethod*.

Неоцененные аргументы остаются неоцененными. Пропущенные аргументы остаются пропущенными.

Синтаксисом для вызова *NextMethod* является *NextMethod*(*generic, object, ...*). Если не предоставлена универсальная функция, то используется значение *.Generic*. Если объект не предоставлен, ьл используется первый аргумент в вызове текущего метода. Используется значение в аргументе '...' для изменения аргументов следующего метода.

Важно понять, что выбор следующего метода зависит от текущего значения *.Generic и .Class* а не на объекте. Так изменение объекта в вызове *NextMethod* влияет на аргументы, полученные следующим методом, но не влияет на выбор следующего метода.

Методы можно вызвать непосредственно. Если они имеются, то будут отсутствовать *.Generic.Class* или *.Method.* В этом случае следует указывать универсальный аргумент *NextMethod*. Принимается значение *.Class* для установки атрибута класса объекта, который является первым аргументом текущей функции. Значение *.Method* - имя текущей функции. Эти варианты для значений по умолчанию гарантируют, что поведение метода не изменяется в зависимости от того, вызывают ли его непосредственно или через вызов обобщения.

Проблема для обсуждения - поведение '...' аргумент *NextMethod*. Книга White описывает поведение следующим образом:

- аргументы, передаваемые по имени, заменяют соответствующие аргументы в вызове текущего метода. Неназванные аргументы идут в начале списка аргументов.

То, что я хотел бы сделать:

- сначала делают аргумент, соответствующий для *NextMethod*;
- если объект или универсальная функция изменены, то прекрасно
- сначала, если именованный элемент списка соответствует аргумент (названный или нет) значение списка заменяет значение аргумента.
- первый неназванный элемент списка

Значение для поиска: Класс: на первом месте из *.Class*, второй из первого

аргумента из метода и последний из объекта, указанного в вызове *NextMethod.*

Универсальная функция: на первом месте из *.Generic,* если ничего, то из первого аргумента из метода, и если все это пропущено, то из вызова *NextMethod.*

Метод: он должен быть только текущим именем функции.

5.6. Групповой метод

Для нескольких типов внутренних функций ***R*** предоставляет механизм поставки операторам. Это означает, что операторам, таким как == или <, можно было изменить их поведение для элементов специальных классов. Функции и операторы были сгруппированы в три категории, и групповые методы могут быть записаны для каждой из этих категорий. В настоящий момент отсутствует механизма для увеличения числа групп. Можно записать методы, определенные для любой функции в пределах группы.

Следующая таблица приводит функции для различных групп.

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

Для операторов в группе *Ops* вызывается специальный метод, если его два операнда, взятые вместе, предлагают единственный метод. Определенно, если оба операнда соответствуют одному и тому же методу или если один операнд соответствует методу, который имеет приоритет по отношению другого операнда. Если они не предлагают единственный метод, то используется метод по умолчанию. Или групповой метод или метод класса доминируют, если у другого операнда отсутствует соответствующий метод. Метод класса доминирует над групповым методом.

Когда группа является *Ops*, то специальная переменная *.Method* является вектором строки с двумя элементами. Элементы *.Method* установлены в имя метода, если соответствующий аргумент - элемент класса, который использовался, чтобы определить метод. Иначе соответствующий элемент *.Method* установлен в нулевую строку длины "".

5.7. Написание методов

Пользователи могут легко написать свои собственные методы и универсальные функции. Универсальная функция - просто функция с вызовом *UseMethod*. Метод - просто функция, которая была вызвана через поставку метода. Это может быть в результате вызова или *UseMethod* или *NextMethod*.

Стоит помнить, что методы можно вызвать непосредственно. Это означает, что они могут быть напечатаны без вызова *UseMethod*, и следовательно, специальные переменные *.Generic, .Class* и *.Method* не устанавливаются. В этом случае правила по умолчанию, детализированные выше, будут использоваться для их определения.

Наиболее популярный способ использования универсальных функций должен обеспечить печать и сводные методы для статистических объектов, обычно вывод некоторого процесса подгонки модели. Для этого каждая модель присоединяет атрибут класса к своему выводу, а затем обеспечивает специальный метод, который берет этот вывод и обеспечивает хорошую читаемую его версию. Затем пользователь должен только помнить, что печать или сводка обеспечат хороший вывод для результатов любого анализа.

6. Вычисление на языке

R принадлежит к классу языков программирования, в которых у подпрограмм есть возможность изменить или создать другие подпрограммы и оценить результат как неотъемлемую часть языка непосредственно. Это подобно Lisp и Scheme и другим разным языкам "функционального программирования", но отличается от семейств ФОРТРАНА и ALGOL. Семейство Lisp доводит эту особенность до крайности: парадигма "все - список", в которой отсутствуют различия между программами и данными.

R представляет более дружественный интерфейс программированию, чем это делает Lisp, по крайней мере, привыкшим к математическим формулам и структурам управления, подобным ***C,*** но механизм действительно очень подобен Lisp. ***R*** позволяет прямой доступ к проанализированным выражениям и функциям и позволяет изменять и впоследствии выполнять их, или создавать полностью новые функции с нуля.

Есть много стандартных применений этого средства, таких как вычисление аналитических производных выражений, или генерации многочленных функций из вектора коэффициентов. Однако существует более фундаментальное использование для работы с интерпретатором ***R.*** Некоторые из них важны для повторного использования функций как компонентов в других функциях, как (по общему признанию не очень симпатично) вызовы *model.frame*, которые созданы в нескольких подпрограммах моделирования и рисования. Другое использование просто позволяет изящные интерфейсы полезной функциональности. Как пример, рассмотрим функцию *curve*, которая позволяет получить график любой функции, заданной подобно *sin(x)*, или средства для рисования математических выражений.

В этой главе дадим введение в набор средств, которые доступны для вычислений на языке.

6.1. Прямое манипулирование языковым объектом

Есть три вида объектов языка, которые доступны для модификации *calls, expressions* и *functions*. Здесь сконцентрируемся на объектах *calls* (вызов). Они иногда упоминаются как "неоцененные выражения", хотя эта терминология несколько сбивает с толку. Наиболее прямой метод получения объекта вызова состоит в использовании кавычки с аргументом выражения, например,

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

Аргументы не оценены, результат - просто проанализированный аргумент. Объекты *e1* и *e2* могут быть оценены позже, используя *eval* или просто управляя как данными. Возможно, наиболее очевидно, почему у *e2* объекта есть режим "вызов", так как он включает вызов функции рисования с некоторыми аргументами. Однако *e1* фактически имеет точно ту же самую структуру как вызов бинарного оператора *+* с двумя аргументами, факт, который ясно выведен на экран следующим:

```
> quote("+"(2, 2))
2 + 2
```

К компонентам объекта вызова получают доступ, используя подобный списку синтаксис, и можно, фактически, преобразовать «в» и «из» списков, используя *as.list* и *as.call:*

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

> e2[[3]]

y

Если ключевого слово аргумента соответствия использовано, то ключевые слова могут использоваться в качестве тегов списка:

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

> e3$x age

> e3$y weight

У всех компонентов объекта вызова есть режим "имя" в предыдущих примерах. Это - верно для идентификаторов в вызовах, но компоненты вызова могут также быть константами - которые могут иметь любой тип, хотя первый компонент должен быть функцией, если вызов должен быть оценен успешно - или другие объекты вызова, соответствуя подвыражениям. Объекты с режимом *name* могут быть созданы из символьных строк, используя *as.name*, таким образом, можно было бы изменить *e2* объект следующим образом:

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

> e2

x + y

Чтобы иллюстрировать факт, что подвыражения - просто компоненты, которые являются самостоятельно вызовами, рассмотрим:

> e1[[2]] <- e2

> e1

x + y + 2

В проанализированных выражениях сохранены все круглые скобки группировки во вводе. Они представлены как вызов функции с одним аргументом, так, чтобы 4 - (2 - 2) стал "-, "(4, "(" (" -" (2, 2))) в префиксной нотации. В оценках оператор *'('* только возвращает свой аргумент.

Это немного неудачно, но не легко записать *parser/deparser* комбинацию, которая и сохраняет ввод данных пользователем, сохранит его в минимальной форме и гарантирует, что *parser/deparser* выражения даст то же самое выражение.

Как это происходит, синтаксический анализатор ***R*** не является совершенно обратимым, ни является своим *deparser* как далее показано в качестве примера:

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

Выражения *Deparsed* должны, однако, оценить к эквивалентному значению к исходному выражению (до погрешности округления).

... внутренняя память конструкций управления потоками... отмечает несовместимость **S**plus...

6.2. Подстановка

Фактически не часто возникает желание изменить внутренние выражения в предыдущем разделе. Более часто желательно просто получить выражение для его декомпиляции и использования его для маркировки рисунка, например. Пример этого показан в начале *plot.default*:

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

Это заставляет переменную или выражение, данную как аргумент ***x*** для *plot,* использовать позже для маркировки оси ***X***.

Обычно функция достигает этого заменой, которая берет выражение ***x*** и заменяется выражением, которое передали через формальный аргумент ***x***. Заметим, что для этого ***x*** должен перенести информацию о выражении, которое создает его значение. Это связано со схемой отложенных вычислений ***R*** (см. раздел 2.1.8 [Объекты обещания]). Формальный аргумент - действительно обещание, объект с тремя слотами, один для выражения, которое определяет его, один для окружающей среды, в которой можно оценить это выражение, и один для значения однажды оцененного выражения. Замена (*substitute*) распознает переменную обещания и заменит значением ее слота выражения. Если замена вызвана в функции, локальные переменные функции также подвергаются подстановке.

Аргумент для замены не должен быть простым идентификатором, это может быть выражение, включающее несколько переменных, и подстановка произойдет для каждого из них. Кроме того, у замены есть дополнительный аргумент, который может быть окружающей средой или списком, в котором ищутся переменные. Например:

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

Заметим, что заключение в кавычки было необходимо, чтобы заменить ***x***. Этот вид конструкции пригождается в соединении со средствами для помещения математического выражения на графики как показано в следующих случаях:

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

Важно понять, что подстановки являются просто лексическими; нет никакой проверки, что получающиеся объекты вызова имеют смысл, если они оценены. *substitute(x <- x + 1, list(x =2))* будет счастливо возвращение 2 <-2 + 1. Однако, некоторые части ***R*** составляют свои собственные правила для того, что имеет смысл, а что нет и могло бы фактически иметь использование для таких плохо сформированных выражений. Например, при использовании “математики на графиках” особенно часто включаются конструкции, которые синтаксически корректны, но которые были бы бессмысленны для оценки, как‘ {}> =40 *" годы"’.

Замена не будет оценивать свой первый аргумент. Это приводит к проблеме подзамены на объекте, который содержится в переменной. Решение состоит в повторном использовании замены, как здесь:

```
> 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
```

Точные правила для подстановок следующие: каждый символ в дереве синтаксического анализа для первого является соответствующим против второго аргумента, который может быть теговым списком или фреймом окружающей среды. Если это - простой локальный объект, его значение вставляется, *except,* если соответствует глобальной окружающей среды. Если это - обещание (обычно аргумент функции), заменяют выражением обещания. Если символ не является соответствующим, это оставляют нетронутым. Специальное исключение для того, чтобы занять место на верхнем уровне является по общему признанию специфическим. Это было наследовано от ***S,*** и объяснение наиболее вероятно, что нет никакого управления, по которому переменные могли бы быть границей на том уровне так, чтобы было лучше только заставить замену действовать как кавычка.

Правило подстановки обещания немного отличается от ***S,*** если локальная переменная изменена прежде, чем используется замена. ***R*** будет затем использовать новое значение переменной, тогда как ***S*** будет безоговорочно использовать выражение аргумента - если это не была константа, у которой есть любопытное следствие, что *f((1))* может очень отличаться от ***f(1)*** в ***S***. Правило ***R*** значительно более чисто, хотя у него действительно есть следствия в соединении с отложенными вычислениями, которые становятся неожиданностью для некоторых. Рассмотрим:

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

Это выглядит прямолинейным, но можно обнаружить, что метка ***y*** становится уродливым выражением *c(...)*. Это происходит, потому что правила отложенных вычислений заставляют *eval(ylab)* выражения происходить после того, как ***y*** был изменен. Решение сначала произвести оценку *ylab* состоит в следующем:

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

Заметим, что нельзя использовать *eval(ylab)* в этой ситуации. Если бы *ylab* являлся языком или объектом выражения, то это произвело бы оценку объекта не в соответствии с требованиями, так если бы передали математическое выражение подобное *quote(log[e](y))*.

Разновидность для *substitute* является *bquote*, которая используется для замены некоторых подвыражений их значением. Пример сверху:

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

может быть записан более компактно:

```
plot(0)
for(i in 1:4)
        text(1, 0.2*i, bquote( x[.(i)] == .(pnorm(i)) ))
```

Выражение заключено в кавычки за исключением содержания *.()* подвыражения, которое заменено их значением. Есть дополнительный аргумент для вычислений значений в различной окружающей среде. Синтаксис для *bquote* заимствован из макроса одинарной обратной кавычки LISP.

6.3. Еще об оценке

Функция оценки *eval* была представлена ранее в этой главе как средство оценки объектов вызова. Однако, это не все сведения. Также можно указать окружающую среду, в которой должна иметь место оценка. По умолчанию - это фрейм оценки, из которого вызывают оценку, но довольно часто это следует установить во что-то еще.

Очень часто соответствующий фрейм оценки – это фрейм родителя текущего фрейма (cf.???). В частности, когда объект для оценки будет результатом работы замены аргументов функции, то он будет содержать переменные, которые имеют смысл только в вызывающей стороне (заметим, что нет причин ждать, что переменные вызывающей стороны находятся в лексическом контексте вызываемого). Так как часто оценка происходит в родительском фрейме, то функция *eval.parent* существует как сокращение для *eval*(*expr, sys.frame (sys.parent*)).

Другой случай, который происходит часто, является оценка в списке или фрейме данных. Например, это происходит в соединении с функцией *model.frame* при наличии аргумент данных. Обычно, аргументы формулы модели подлежат оценке на данных, но они могут иногда также содержать ссылки на элементы в вызывающей стороне *model.frame*. Это иногда полезно в соединении с имитационными исследованиями. Таким образом, с этой целью нужно не только оценить выражение в списке, но также и специфицировать обертку, в которой продолжается поиск, если переменная не находится в списке. Следовательно, у вызова есть вид:

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

Заметим, что оценка в данной окружающей среде может фактически изменить эту окружающую среду, что наиболее очевидно в случаях включения оператора присваивания, как например:

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

Это также верно при оценке в списках, но исходный список не изменяется, потому что в действительности обрабатывается копия.

6.4. Оценка объектов выражение

Объекты режима "выражение" определены в разделе 2.1.4 [Объекты выражения]. Они подобны спискам объектов вызова.

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

Заметим, что оценка объекта выражения оценивает каждый вызов поочередно, но заключительное значение - значение последнего вызова. В этом отношении она ведет себя почти тождественно составной кавычке объекта языка *({2 + 2; 3 + 4}).* Однако, есть тонкие различия: объекты *Call* неотличимы от подвыражений в дереве синтаксического анализа. Это означает, что они автоматически оценены как если бы имелось подвыражение. Объекты выражения могут быть распознаны во время оценки и в некотором смысле сохранить свое отсутствие кавычек. Средство оценки не будет оценивать объект выражения рекурсивно, а лишь в случае его передачи

непосредственно функции *eval* как выше. Различия могут быть замечены здесь:

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

Декомпилятор представляет объект выражения вызовом, который создает его. Это подобно способу, по которому он обрабатывает числовые векторы и несколько других объектов, у которых нет определенного внешнего представления. Однако, это действительно приводит к следующему некоторому беспорядку:

```
> 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"
```

То есть, ***e*** и ***ee*** выглядят идентично при печати, но первый является вызовом, который генерирует объект выражения, и другой является непосредственно объектом.

6.5. Манипулирование вызовами функции

Для функции можно узнать, как ее вызвали, просмотрев результат *sys.call* как в следующем примере функции, которая просто возвращает свой собственный вызов:

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

Однако, в действительно это не полезно за исключением отладки, потому что требуется, чтобы функция отследила соответствие аргумента для интерпретации вызова. Например, она должна в состоянии видеть, что 2-ой фактический аргумент является соответствующим первому формальному аргументу (***x*** в вышеупомянутом примере).

Чаще требуется вызов со всеми фактическими аргументов, связанными соответствующими формальными. С этой целью используется функция *match.call*. Вот разновидность предыдущего примера, функция, которая возвращает ее собственный вызов с соответствующими аргументами:

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

Заметим, что теперь второй аргумент соответствует ***x*** и появляется в соответствующей позиции в результате.

Основное использование этого метода состоит в вызове другой функции с теми же самыми аргументами, возможно удаляя некоторые и прибавляя другие. Типичное применение видно в начале функции *lm*:

```
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()))
```

Заметим, что полученный вызов оценен в родительском фрейме, в котором можно быть уверенным, что включенные выражения имеют смысл. Вызов может быть обработан как объект списка, где первый элемент - имя функции, и остающиеся элементы - фактические аргументы выражений с соответствующими формальными именами аргумента в качестве тегов. Таким образом, метод для устранения нежелательных аргументов должен присвоить NULL, как видно в строках 2 и 3, и прибавить аргумент, использующий теговое присвоение списка (здесь, чтобы передать *drop.unused.levels = TRUE)* как в строке 4. Чтобы изменить название вызванной функции, присвойте первому элементу списка и удостоверьтесь, что значение - имя, или здесь используется конструкция *as.name* ("*model.frame*") или кавычка (*model.frame*).

У функции *match.call* есть аргумент *expand.dots*, который является переключателем, если он установлен в FALSE, позволяя собрать все аргументы '...' как отдельный аргумент с тегом '...'.

```
> 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))
```

Аргумент '...' является списком (для точности парным списком), а не вызовом списка как это имеет место в ***S****:*

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

Одной из причин использования этой формы *match.call* является желание просто избавиться от любых аргументов '...', чтобы не передавать неуказанные аргументы функции, которые, возможно, не знают их. Вот пример, перефразируемый от *plot.formula:*

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

Более тщательно продуманное применение находится в *update.default,* где ряд дополнительных аргументов можно добавить, заменить или отменить таковые из исходного вызова:

```
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)
        }
}
```

Заметим проявленную заботу по индивидуальному изменению существующих аргументов в случае *extras[[a]] == NULL*. Связывание не работает над объектами

вызова без приведения как показано; это - возможно ошибка.

Две дальнейших функции существуют для конструкции вызовов функции, а именно: *call* и *do.call*. Вызов функции позволяет создать объект вызова с имени функции и списка аргументов:

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

Как видно, значение ***x***, а не символа, вставлено в вызов, таким образом, отчетливо отличается от *round(x)*. Форма используется довольно редко, но иногда полезна, где имя функции доступно как символьная переменная.

Функция *do.call* связывается, но сразу оценивает вызов и берет аргументы из объекта режима "список", содержащего все аргументы. Естественное использование этого состоит в том, что каждый хочет применить функцию как *cbind* ко всем элементам фрейма данных или списка.

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

Другое использование включает изменения для конструкций, подобным *do.call("f",list(...))*. Однако нужно знать, что этим включается оценка аргументов перед фактическим вызовом функции, что может победить аспекты отложенных вычислений и подстановки аргумента непосредственно в функции. Подобный комментарий применяется к функции *call*.

6.6. Манипулирование функциями

Часто полезно иметь возможность управлять компонентами функции или обертки. ***R*** обеспечивает ряд функций интерфейса с этой целью.

body	Возвращает выражение, которое является телом функции.
formals	Возвращает список формальных аргументов функции. Это парный список (pairlist).
environment	Возвращает окружающую среду, связанную с функцией.
body<-	Устанавливает тело функции к предоставленному выражению.
formals <-	Устанавливает формальные аргументы функции к предоставленному списку.
environment<-	Устанавливает окружающую среду функции к указанной окружающей среде.

Также возможно изменить привязку различных переменных в окружающей среде функции, используя код вроде *evalq(x <- 5, environment(f))*.

Также возможно преобразовать функцию в список, используя *as.list*. Результат - связь списка формальных аргументов с телом функции. Наоборот такой список может быть преобразован в функцию, используя *as.function*. Эта функциональность, главным образом, включена для совместимости с ***S***. Заметим, что информация об окружающей среде теряется при использовании *as.list*, тогда как у *as.function* есть аргумент, который позволяет установить окружающую среду.

7. Интерфейсы системы и внешних языков

7.1. Оперирование доступом к системе

Доступ к обертке операционной системы через систему функции ***R***. Детали будут отличаться платформой (см. онлайновую справку), и все, что может безопасно быть предположено – это то, что первым аргументом будет командная строка, которую передадут для выполнения (не обязательно оберткой), а второй аргумент будет внутренним, который, если *true*, соберет вывод команды в символьный вектор ***R.***

Для синхронизации доступны функции *system.time* и *proc.time* (хотя доступная информация может быть ограничена на платформах "не похожих на Unix").

К информации от окружающей среды операционной системы можно получить доступ и управлять с:

Sys.getenv	переменные окружения OS
Sys.putenv	
Sys.getlocale	локализация системы
Sys.putlocale	
Sys.localeconv	
Sys.time	текущее время
Sys.timezone	временная зона

Универсальный набор функций доступа к файлу обеспечен на всех платформах:

file.access	доступность файла
file.append	соединить файлы
file.choose	показать пользователю имя файла
file.copy	копировать файлы
file.create	создать или отсечь файл
file.exists	проверка на существование
file.info	дополнительная информация о файле
file.remove	удалить файл
file.rename	переименовать файл
file.show	показать текстовый файл
unlink	отсоединить файл или директорию.

Есть также функции для управления именами файла и путями независимым от платформы способом.

базовое имя	имя файла без каталога
dirname	имя каталога
file.path	создайте путь к файлу
path.expand	разверните ~ в путь Unix

7.2. Интерфейсы внешних языков

См. раздел “Система и интерфейсы внешнего языка” в Написании расширений ***R*** для деталей добавления функциональности ***R*** через скомпилированный код.

Функции *.**C*** и *.Fortran* обеспечивают стандартный интерфейс для скомпилированного кода, который был связан с ***R,*** либо во время создания, либо через *dyn.load*. Они, прежде всего, предназначены для скомпилированного кода ***C*** и ФОРТРАНА, соответственно, но функция *.**C*** может использоваться с другими языками,

которые могут генерировать интерфейсы ***C,*** например ***C++***.

Функции *.**C*** и *.Fortran* обеспечивают интерфейсы для скомпилированного кода (прежде всего, компилированного кода C) для управления объектами ***R***.

7.3. .Internal и .Primitive

Интерфейсы *.Internal* и *.Primitive* используются для вызова кода ***C*** компилированного в ***R*** во время построения. Смотри раздел "*.Internal vs .Primitive*" в ***R.***

8. Обработка прерываний

Услуги обработки прерываний в ***R*** предоставлены через два механизма. Функции, такие как остановка (*stop)* или предупреждение (*warning),* могут быть вызваны непосредственно, или могут использоваться опции, такие как "*warn*", для управления обработкой проблем.

8.1. Стоп

Вызов остановки прерывает оценку текущего выражения, печатает аргумент сообщения и возвращается к выполнению на верхнем уровне.

8.2. Предупреждение

Функция *warning* (предупреждения) берет единственный аргумент, который является символьной строкой. Поведение вызова предупреждения зависит от значения опции "*warn*". Если "*warn*" является отрицательным, то предупреждение игнорируются. Если оно равно нулю, то они сохранятся и печатаются после завершения высокоуровневой функции. Если это единица, то они печатаются при совершении, и если равны 2 (или больше), предупреждения превращены в ошибки.

Если "*warn*" равно нулю (по умолчанию), то создается переменная *last.warning,* и сообщения, связанные с каждым вызовом предупреждения последовательно хранятся в его векторе. Если имеется менее 10 предупреждений, то они печатаются после окончания оценки функции. Если имеется более 10, то печатается сообщение, указывающее количество произошедших предупреждений. В любом случае *last.warning* содержит вектор сообщений, и предупреждения обеспечивают способ получения доступа к их печати.

8.3. on.exit

В любой точке в теле функции можно вставить вызов *on.exit.* Действие вызова *on.exit* состоит в сохранении значение тела так, чтобы оно было выполнено при выходе из функции. Это позволяет функции изменять некоторые системные аргументы и гарантировать их сброс, чтобы приспособить значение при завершении функции. *on.exit* будет гарантированно выполнен при выходе из функции или непосредственно, или как результат предупреждения.

Ошибка в оценке кода *on.exit* вызывает непосредственный переход к верхнему уровню без дальнейшей обработки кода *on.exit.*

on.exit берет единственный аргумент, который является выражением, которое будет оценено при выходе из функции.

8.4. Опции ошибок

Существует набор опций, которые могут использоваться в ***R*** для управления обработкой ошибок и предупреждений. Перечислен в таблице ниже.

'warn' Управление печатью предупреждений.

'warning.expression'

Устанавливает подлежащее оценке выражение при возникновении ошибки. Подавляется нормальная печать предупреждений при

включении опции.

‘*error*’ Устанавливает подлежащее оценке выражение при возникновении ошибки. Нормальная печать сообщений об ошибках и предупреждающих сообщений предшествует оценке выражения.

Выражения, установленные *options("error")*, оцениваются до выполнения вызовов *on.exit*.

Можно использовать *options(error = expression(q("yes")))*, чтобы заставить ***R*** выйти после сообщения об ошибке. В этом случае ошибка заставит ***R*** завершать работу, и глобальная окружающая среда будет сохранена.

9. Отладка

Отладка кода всегда была чем-то вроде искусства. ***R*** обеспечивает несколько инструментов, которые помогают пользователям найти проблемы в своем коде. Эти инструменты останавливают выполнение в определенных точках в коде, и текущее состояние вычислений может быть осмотрено.

Большинство средств отладки вызывается или посредством вызовов браузера (*browser),* или посредством отладки *(debug)*. Обе эти функции основаны на одинаковом внутреннем механизме, и обе предоставляют пользователю специальный запрос. Любая команда может быть введена в запросе. Окружающая среда оценки для команды - в настоящий момент активная окружающая среда. Это позволяет исследовать текущее состояние любых переменных и т.д.

Есть пять специальных команд, которые ***R*** интерпретирует по-разному. Это:

‘*RET*’ Переход на следующий оператор при отладке функции. Продолжить выполнение при вызове браузера.

‘*c*’

‘*cont*’ Продолжить выполнение.

‘*n*’ Выполнить следующий оператор в функции. Также работает в браузера.

‘*where*’ Показать стек вызовов

‘*Q*’ Остановить выполнение и сразу перейти к верхнему уровню.

Если существует локальная переменная с тем же самым именем, как одна из специальных команд, упомянутых выше, то к ее значению можно получить доступ при использовании *get*. Вызов *get* с именем в кавычках получит значение в текущей окружающей среде.

Отладчик обеспечивает доступ только к интерпретируемым выражениям. Если вызываются функции внешнего языка (такого как ***C***), то не предоставляется доступ к операторам на этом языке. Выполнение остановится на следующем операторе, который оценен в ***R.*** Можно использовать символьный отладчик, такой как *gdb*, для отладки скомпилированного кода.

9.1. Браузер

Вызов функции браузера заставляет ***R*** останавливать выполнение в точке и предоставляет пользователю специальный запрос. Аргументы браузеру игнорируются.

```
> 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 Отладка/неотладка

Отладчик может быть вызван для любой функции при использовании команды *debug(fun).* Впоследствии, каждый раз при оценке функции, вызывается отладчик. Отладчик позволяет управлять оценкой операторов в теле функции. До выполнения каждого оператора, этот оператор распечатывается, и предоставляется специальный запрос. Может быть дана любая команда, но перечисленные в таблице выше имеют особое значение.

Отладка выключается вызовом *undebug* с функцией как аргумент:

```
> 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. Трассировка/нетрассировка

Другой способ контролировать поведение ***R*** состоит в механизме трассировки. Вызывают *trace* с единственным аргументом, который является именем функции, подлежащей отслеживанию. Имя не должно быть заключено в кавычки, но для некоторых функций следует заключить имя в кавычки, чтобы избежать синтаксической ошибки.

Если трассировка была вызвана для функции, то каждый раз при оценке функции ее вызов распечатывается. Этот механизм удаляется при *untrace* с функцией в качестве аргумента:

```
> trace("[<-")
```

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

9.4. Обратная трассировка

Когда ошибка вызвала переход к верхнему уровню, специальная переменная, названная *.Traceback*, помещается в основную окружающую среду. *.Traceback* - символьный вектор с одной записью для каждого вызова функции, который был активным во время возникновения ошибки. Исследование *.Traceback* может быть выполнено вызовом *traceback*.

10. Синтаксический анализатор

Синтаксический анализатор преобразовывает текстовое представление кода ***R*** во внутреннюю форму, которую затем передается к средству оценки в ***R,*** которое выполняет указанные инструкции. Внутренняя форма является самостоятельным объектом ***R*** и может быть сохранена и управляться в пределах системы ***R.***

10.1. Процесс синтаксического анализа

10.1.1. Режимы синтаксического анализа

Синтаксический анализ в ***R*** выполняется в трех различных разновидностях:

- Цикл "считал-оценил-напечатал".
- Синтаксический анализ текстовых файлов.
- Синтаксический анализ символьных строк.

Цикл "считал-оценил-напечатал" формирует основной интерфейс командной строки ***R.*** Читается текстовый ввод, пока не доступно полное выражение ***R.*** Выражения могут быть разделены на несколько входных строк. Основной запрос (по умолчанию‘> ’) указывает, что синтаксический анализатор готов к новому выражению, а запрос продолжения (по умолчанию '+') указывает, что синтаксический анализатор ожидает остаток от неполного выражения. Выражение преобразуется во внутреннюю форму во время ввода, и проанализированное выражение передается средству оценки, а результат печатается (если специально не сделан невидимым). Если синтаксический анализатор оказывается в состоянии, которое является несовместимым с синтаксисом языка, то отмечается "Синтаксическая ошибка", и синтаксический анализатор сбрасывается и возобновляет ввод в начале следующей входной строки.

Можно проанализировать текстовые файлы, используя функцию синтаксического анализа. В частности это делается во время ввода функции, которая сохраняет команды во внешнем файле, и выполняет их, как будто они были введены на клавиатуре. Заметим, тем не менее, весь файл анализируется и проверяется на синтаксис до выполнения любой оценки.

Символьные строки, или векторы, могут быть проанализированы, используя синтаксический анализатор для *text= argument*. Строки обрабатываются так, как будто они были строками входного файла.

10.1.2. Внутреннее представление

Проанализированные выражения сохранятся в объекте ***R,*** содержащем дерево синтаксического анализа. Более полное описание таких объектов может быть найдено в разделе 2.1.3 [Объекты языка] и разделе 2.1.4 [Объекты выражения]. Кратко, каждое элементарное выражение ***R*** хранится в форме вызова функции как список с первым элементом, содержащим имя функции, и остаток, содержащий аргументы, которые в дальнейшем могут быть выражениями ***R***. Элементы списка можно назвать, соответствуя теговому соответствию формальных и фактических аргументов.

Заметим, что все элементы синтаксиса ***R*** обработаны следующим образом, например, присвоение *x <-1* закодировано как *" <-"(x, 1).*

10.1.3. Обратное преобразование

Любой объект ***R*** может быть преобразован в выражение ***R,*** используя обратное преобразование (*deparse*). Это часто используется в соединении с выводом результатов, например, для маркировки рисунков. Заметим, что только объекты режима "выражение", как можно ожидать, будут неизменны, повторно анализируя вывод обратного преобразования. Например, числовой вектор 1:5 будет выглядеть при обратном преобразовании как "*c(1,2,3,4,5)*", который повторно проанализирует как вызов функции ***c***. В максимально возможной степени оценка обратного преобразования и повторно проанализированного выражения дает одинаковый результат как оценка оригинала, но есть несколько неуклюжих исключений, главным образом включая выражения, которые не генерировались из текстового представления на первом этапе.

10.2. Комментарии

Комментарии в ***R*** пропускаются синтаксическим анализатором. Любой текст, начиная с символа #, и до конца строки принимается за комментарий, если символ # не заключен в кавычки в строке. Например:

```
> x <- 1    # Это комментарий...
> y <- "    #... а этот нет."
```

10.3. Стандартные блоки - Tokens

Маркеры - элементарные стандартные блоки языка программирования. Они распознаются во время лексического анализа, который (концептуально, по крайней мере) имеет место до синтаксического анализа, непосредственно выполняемого синтаксическим анализатором.

10.3.1. Константы

Есть пять типов констант: *integer, logical, numeric, complex* и *string*. Кроме того, есть четыре специальные константы: *NULL, NA, Inf* и *NaN*.

NULL используется для указания на пустой объект. *NA* используется для отсутствующего (“Не доступного”) значения данных. *Inf* обозначает бесконечность, и *NaN* - не-число в исчислении плавающей точки IEEE (результаты операций соответственно 1/0 и 0/0, например).

Логические константы – либо TRUE (ИСТИНА), либо FALSE (ЛОЖЬ).

Числовые константы следуют за подобным синтаксисом из языка C. Они состоят из целой части, состоящей из нуля или большего количества цифр, сопровождаемых дополнительно '.', и дробной части из нуля или большего количества цифр, дополнительно сопровождаемых символом экспоненты, состоящей из ‘E’ или ‘e’, дополнительного знака и строки одной или более цифр. Или дробная или десятичная часть могут быть пустыми, но не обе сразу.

Допустимые числовые константы: 1 10 0.1 .2 1e-7 1.2e+7

Числовые константы могут также быть шестнадцатеричными, начинаясь с *‘0x’*, или *‘0x’*, сопровождаемого нулем или большим количеством цифр ***‘a-f’*** или ***‘A-F’.*** Поддерживаются шестнадцатеричные константы с плавающей точкой, используя синтаксис C99, т.е. *‘0x1.1p1’*.

Теперь существует отдельный класс целочисленных констант. Они создаются при использовании спецификатора ***L*** в конце числа. Например, *123L* дает целочисленное значение, а не действительное значение. Можно использовать суффикс ***L*** для квалификации любого не комплексного числа с намерением создания целого

числа. Таким образом, это может использоваться с числами, представленными в шестнадцатеричном или экспоненциальном виде. Однако, если значение не допустимое целое число, выдается предупреждение, и создается действительное значение. Следующие примеры показывают допустимые целочисленные константы, значение которых будет генерировать предупреждение и давать действительные константы и синтаксические ошибки.

Допустимые целочисленные константы: 1L, 0x10L, 1000000L, 1e6L

Допустимые числовые константы:1.1L, 1e-3L, 0x1.1p-2

Синтаксическая ошибка: 12iL 0x1.1

Предупреждение выдаются для значения, которое содержит ненужную десятичную точку, например, 1.L. Это - ошибка иметь десятичную точку в шестнадцатеричной константе без двоичной степени.

Заметим также, что предыдущий знак (+ или -) обработан как унарный оператор, а не как часть константы.

Актуальная на настоящий момент информация по принятым форматам может быть найдена путем *?NumericConstants.*

У комплексных констант есть форма десятичной числовой константы, сопровождаемой *'i*. Заметим, что только просто мнимые числа - фактические константы, другие комплексные числа проанализированы унарные или бинарные операции на числовых и мнимых числах.

Допустимые комплексные константы: 2i 4.1i 1e-2i

Строковые константы разграничены парой одинарных (‘ ’ ’) или двойных (‘ " ’) кавычек и могут содержать все другие печатаемые символы. Кавычки и другие специальные символы в пределах строк специфицируются, используя escape-последовательности:

|’ одинарная кавычка

|" двойная кавычка

|*n* новая строка

|*r* возврат каретки

|*t* символ вкладки

|*b* клавиша Backspace

|*a* звонок

|*f* перевод формата

|*v* вертикальная вкладка

|| наклонная черта влево непосредственно

|*nnn* символ с данным восьмеричным кодом - последовательности один, две или три цифры в диапазоне 0... 7 приняты.

|*xnn* символ с данным шестнадцатеричным кодом - последовательности одной или двух шестнадцатеричных цифр (с записями 0 *... 9 A... F... f).*

|*unnnn |u {nnnn}*

(где много байтовые локали поддерживаются, иначе ошибка). Символ Unicode с данным шестнадцатеричным кодом - последовательности до четырех шестнадцатеричных цифр. Символ должен быть допустимым в текущей локали.

*Unnnnnnnn \U{nnnnnnnn***}***

(где много байтовые локали поддерживаются а не на Windows, иначе

ошибка). Символ Unicode с данным шестнадцатеричным кодом - последовательности до восьми шестнадцатеричных цифр.

Одинарная кавычка может также быть встроена непосредственно в разграниченной строке двойной кавычки и наоборот.

Начиная с ***R*** 2.8.0, 'nul' (\0) не позволен в символьной строке, так как использование \0 в строковой константе завершает константу (обычно с предупреждением): дальнейшие символы до кавычки на момент закрытия отсканированы, но игнорируются.

10.3.2. Идентификаторы

Идентификаторы состоят из последовательности букв, цифр, точки (' . ') и подчеркивания. Они не должны начинаться ни с цифры, ни с подчеркивания, ни с точки, сопровождаемой цифрой.

Определение буквы зависит от текущей локализации: точный набор допустимых символов дается выражением *C(isalnum (c) // c =='.' // c == '_')* и будет включать акцентируемые буквы во многие западноевропейские локализации.

Заметим, что идентификаторы, начинающиеся с точки, по умолчанию не перечислены функцией *ls,* и что '...' и '.. 1', '.. 2', и т.д. являются специальными.

Заметим также, что у объектов могут быть имена, которые не являются идентификаторами. К ним обычно получают доступ через *get* и *assign,* хотя они могут также быть представлены текстовыми строками при некоторых ограниченных обстоятельствах, когда нет никакой неоднозначности (например, *"x" <-1*). Как *get* и *assign*, не ограничены именам, которые являются идентификаторами, они не распознают операторы преобразования в нижний индекс или заменяющие функции. Следующие пары не эквивалентны:

x$a<-1	*assign("x$a",1)*
x[[1]]	*get("x[[1]]")*
names(x)<-nm	*assign("names(x)",nm)*

10.3.3. Зарезервированные слова

Следующие идентификаторы имеют особое значение и не могут использоваться для имен объектов:

if else repeat while function for in next break
TRUE FALSE NULL Inf NaN
*NA NA_integer_ NA_real_ NA_complex_ NA_character_*
... ..1 ..2 etc.

10.3.4. Специальные операторы

R позволяет определяемые пользователем инфиксные операторы. У них вид строки символов, разграниченных символом '%'. Строка может содержать любой печатаемый символ, кроме '%'. Escape-последовательности для строк здесь не применяются:

Заметим, что предопределены следующие операторы:

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

10.3.5. Разделители

Хотя не строго маркеры, фрагменты пробельных символов (пробелы, табуляции и переводы формата в Windows и локалях UTF-8 другие символы пробелов Unicode) служат для разграничения маркеров в случае неоднозначности, (сравните *x <- 5* и *x < -5*).

У новых строк есть функция, которая является комбинацией разделителя маркера и разделителя выражения. Если выражение может завершиться в конце строки,

синтаксический анализатор предположит, что это так и есть, иначе новая строка обрабатывается как пробел. Точки с запятой (‘;’) могут использоваться для разделения элементарных выражений на одной строке.

Специальные правила применяются к ключевому слову *else*: в составном выражении прежде, чем *else* завершится, на наиболее удаленном уровне, новая строка завершает конструкцию *if* и последующее *else* вызывают синтаксическую ошибку. Это несколько аномальное поведение происходит из-за применения ***R*** в интерактивном режиме, что позволяет определить, является ли входное выражение полным, неполным или недопустимым, как только пользователь нажимает RET.

Запятая (‘,’) используется для разделения аргументов функции и многих индексов.

10.3.6. Оператор-символ

R использует следующие маркеры операторов:

*+ - * / %% ^*	*арифметические*
> >= <	
*<= == != *	*отношений*
! & |	*логические*
~	*формулы модели*
-> <-	*присвоение*
$	*индексация списка*
:	*последовательность*

У некоторых операторов имеется иное значение в формулах модели.

10.3.7. Группировка

Обычные круглые скобки - ‘(’ и ‘) ’ - используются для явной группировки в пределах выражений и разграничения списков аргументов для определений и вызовов функции.

Фигурные скобки - ‘{’ и‘} ’- разделяют блоки выражений в определениях функций, условных выражениях и итеративных конструкциях.

10.3.8. Символ индекса

Индексация массивов и векторов выполняется использованием отдельных и двойных скобок ‘***[ ]***’ и ‘***[[ ]]***’. Кроме того, индексация теговых списков может быть сделана, используя оператор ‘***$***’.

10.4. Выражения

Программа ***R*** состоит из последовательности выражений ***R.*** Выражение может быть простым выражением, состоящим только из константы или идентификатора, или оно может быть составным выражением, созданным из других частей (которые могут самостоятельно быть выражениями).

Следующие разделы детализируют различные доступные синтаксические конструкции.

10.4.1. Вызов функций

Вызов функции принимает форму функциональной ссылки, сопровождаемой списком разделенных запятой значений ряда аргументов в пределах круглых скобок.

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

Также функциональная ссылка может быть:

- идентификатором (именем функции);
- текстовой строкой (так же, но удобнее, если у функции есть имя, которое не является допустимым идентификатором),
- выражением (которое должен оцениваться к функциональному объекту),

Каждый аргумент может быть тегирован *(tag=expr)*, или быть лишь простым выражением. Также он может быть пустым, или он может быть одним из специальных маркеров '...', '.. 2', и т.д.

Тег может быть идентификатором или текстовой строкой. Примеры:

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

10.4.2 . Суффикс и префикс операторов

Порядок очередности (самый высокий первый) операторов:

```
::
$ @
^
- +              (unary)
:
%xyz%
* /
+ -              (binary)
> >= < <= == !=
!
& &&
| ||
~                (unary and binary)
-> ->>
=                (as assignment)
<- <<-
```

Заметим, что: предшествует двоичный +/-, но не ^. Следовательно, 1:3-1 равняется 012, но 1:2^3 равняется 1:8.

Оператор возведения в степень '^' и левые операторы присваивания плюс-минус '<-= <<-' группа справа налево, вся другая группа операторов слева направо. Таким образом, 2^ 2^3 равно 2^8, а не 4^3, тогда как 1 - 1 - 1 равно −1, а не 1.

Заметим, что операторы %% и %/% для целочисленного остатка и деления, имеют более высокий приоритет, чем умножение и деление.

Хотя он не строго оператор, но он также нуждается в упоминании: знак '=' используется для тегирования аргументов в вызовах функции и для присвоения значения по умолчанию в функциональных определениях.

Знак ***'$'*** является в некотором смысле оператором, но не позволяет произвольные правые стороны и обсужден в разделе 10.4.3 [Индексные конструкции]. У него более высокий приоритет по сравнению с любым другим оператором.

Проанализированная форма унарной или бинарной операции абсолютно эквивалентна вызову функции с оператором как имя функции и операндами в качестве аргументов функции.

Круглые скобки записаны как эквивалентные унарному оператору с именем "(" даже в случаях, где круглые скобки могли быть выведены из приоритета оператора, например, a * (b + c).

Заметим, что символы присвоения - операторы точно такие же, как арифметические, реляционные и логические операторы. Также разрешены выражения на целевой стороне присвоения настолько, насколько синтаксического анализатора касается допустимость выражения. (*2 + 2 <-5* является допустимым выражением

настолько, насколько касается синтаксического анализатора. Хотя средство анализа является объектом). Подобные комментарии применяются к оператору формулы модели.

10.4.3. Конструкция индексов

У R есть три конструкции индексации, две из которых синтаксически подобны, хотя с несколько различной семантикой:

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

Формально объект может быть любым допустимым выражением, но это, как понимают, обозначает или оценивает подустанавленный объекту. Аргументы обычно оценивают к действительным или символьным индексам, но возможны другие виды аргументов (особенно при *drop = FALSE*).

Внутренние индексные конструкции сохранятся как вызовы функции с именем функции "[" или, соответственно, "[[".

Третья конструкция индексов имеет вид:

object $ tag

Здесь, объект как выше, тогда как тег - идентификатор или текстовая строка. Внутренне сохранится как вызов функции с именем "$".

10.4.4. Составные выражения

Составное выражение имеет вид:

{ expr1 ; expr2 ; ..... ; exprn }

Точки с запятой можно заменить переводом строки. Внутренне сохранится как вызов функции с "{" как имя функции и выражения как аргументы.

10.4.5. Элементы управления потоком

R содержит следующие структуры управления как специальные синтаксические конструкции:

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

Выражения в этих конструкциях обычно будут составными выражениями.

В пределах конструкций цикла **(***while, repeat, for***)** можно использовать *break* (для завершения цикла) и *next* (для перехода к следующей итерации).

Внутренне, конструкции сохранятся как вызовы функции:

"if"(cond, expr) "if"(cond, expr1, expr2)

"while"(cond, expr) "repeat"(expr)

"for"(var, list, expr) "break"()

"next"()

10.4.6. Определение функций

Определение функции имеет вид:

function (arglist) body

Тело функции - выражение, часто составное выражение. *arglist* - список разделенных запятой значений элементов, каждый из которых может быть идентификатором, или вида '*identifier = default*', или специальный маркер '...'. По умолчанию может быть любым допустимым выражением.

Заметим, что у аргументов функции в отличие от тегов списка, и т.д., не может

быть “странных имен”, предоставленных в виде текстовых строк.

Внутренне определение функции хранится как вызов функции с именем функции *function* и двумя аргументами: *arglist* и *т елом*. *arglist* хранится как теговый парный список, где теги - имена аргументов, а значение - выражения по умолчанию.

10.5. Директивы

Синтаксический анализатор в настоящий момент поддерживает только одну директиву *#line*. Она подобна директиве препроцессора ***C*** с этим же именем. Синтаксис имеет вид:

#line nn ["filename"]

где *nn* - целочисленный номер строки, и дополнительное *имя файла* (при необходимости в двойных кавычках):

names the source file.

В отличие от директивы ***C,*** *#line* должна появиться как первые пять символов на строке. Как в ***C,*** *nn* и запись "*filename*" может быть разделена пробелами. И в отличие от ***C,*** что-либо после текста на строке будет обрабатываться как комментарий и игнорироваться.

Эта директива говорит синтаксическому анализатору, что следующая строка является строкой *nn* из файла с именем *filenane*. Если имя файла не дано, то предполагается, что имя следует взять из предыдущей директивы. Обычно не используется пользователями, но может использоваться препроцессорами для отнесения диагностических сообщений к исходному файлу.

Приложение A. Ссылки

Richard A. Becker, John M. Chambers and Allan R. Wilks (1988), The New S Language. Chapman & Hall, New York. This book is often called the “Blue Book ”.